《最优控制与设计》Final report

姓名：王博彬　侯雪冰　龚子利　　学号：12212711　12212761　12210209

Problem I: State Estimation for a Biped Robot

1 Background Information

状态估计的重要性: 状态估计在机器人控制系统中起着至关重要的作用，特别是在涉及双足机器人的应用中。它涉及使用传感器数据以及状态空间模型来估计系统内部无法直接观测到的状态。这对于有效控制机器人并确保其能够与环境安全、高效地交互至关重要。

卡尔曼滤波器的应用: 在状态估计中，最广泛使用的技术之一是卡尔曼滤波器。卡尔曼滤波器是一种算法，它利用一系列随时间观测到的含有统计噪声和其他不精确性的测量值，来产生未知变量的估计，这些估计通常比基于单一测量的估计更为精确。

项目目标：在这个项目中，我们将实现一个双足机器人的状态估计器。状态估计器的主要作用是为双足机器人的控制策略提供观测输入，具体来说，提供浮动基座的线速度。

2 Derive the State Space Model

通过对 IMU 原始数据的调整以对齐 IMU 坐标系与机体坐标系，转动角度和转动方向的速度：

$$^{\text{imu}}\alpha_{\text{original}}, ^{\text{imu}}\alpha_{\text{original}}, ^{\text{imu}}\alpha_{\text{original}}$$

Body 坐标系下的旋转速度和 IMU 坐标系下的旋转速度关系为：

$$^{B}w_{\text{OB}} = ^{B}R_{\text{imu}} w_{\text{original}}$$

因为 imu 在 body 内固定，所以 $^{B}R_{\text{imu}}$ 由此得到 $^{B}w_{\text{OB}}$。

body 至 world 坐标系下的旋转矩阵和 IMU 到 world 的旋转矩阵可通过以下关系获得：

$$^{O}R_{B} = ^{O}R_{\text{imu0}} \cdot ^{\text{imu0}}R_{\text{imu}} \cdot ^{\text{imu}}R_{B}$$

即 imu0 坐标系和 world 坐标系是固定的，imu 在 body 内固定，所以 ${}^{O}R_{\text{imu0}}.{}^{\text{imu0}}R_{\text{imu}}.{}^{\text{imu}}R_B$ 相同。

而 R_{imu} 可以通过 IMU 的测量值 ${}^{\text{imu}}\alpha_{\text{original}}$ 直接得到，由此可以求出 world 坐标系下 body 的旋转矩阵 OR_B 和 world 坐标系下 body 的速度 ${}^{o}w_{\text{OB}}$。因为

$$ {}^{o}w_{\text{OB}} = {}^{B}\ R_{\text{o}}^{B} w_{\text{OB}} \times {}^{o}\ R_{\text{B}} $$

从而求取位移和。

此外，我们可以通过 IMU 角度测量之解算出初始方位角的一半，接下来重置使用上/保持测量中给出 world 坐标系下 body 的位置重置。

状态向量定义如下：

$$ \mathbf{x} = [p_{\text{com}}, v_{\text{com}}, p_1, p_2] $$

这里，p_{com} 和 v_{com} 分别是机器人质心的位置和速度，p_1, p_2 是机器人两个足部在世界坐标系中的位置。

动态方程定义如下：

$$ \begin{aligned} \dot{p}_{\text{com}} &= v_{\text{com}} \\ \dot{v}_{\text{com}} &= a_{\text{com}} + g \\ \dot{p}_i &= 0, \quad \text{for } i = 1, 2 \\ p_{\text{foot},k+1} &= [p_1^{k+1}, p_2^{k+1}] \end{aligned} $$

其中，a_{com} 是质心加速度，g 是重力加速度。

$$ \begin{bmatrix} \dot{p}_{\text{com}} \\ \dot{v}_{\text{com}} \\ p_{\text{foot},k+1} \end{bmatrix}_{x_{k+1}} = \begin{bmatrix} I_3 & \Delta t \cdot I_3 & 0_{3\times 6} \\ 0_{3\times 3} & I_3 & 0_{3\times 6} \\ 0_{6\times 3} & 0_{6\times 3} & I_6 \end{bmatrix} \begin{bmatrix} p_{\text{com}} \\ v_{\text{com}} \\ p_{\text{foot}} \end{bmatrix}_{x_k} + \begin{bmatrix} \Delta t \cdot I_3 \\ 0_{3\times 3} \\ 0_{6\times 3} \end{bmatrix} \dot{v}_{\text{com}} $$

状态转移方程为：

$$ \mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k $$

其中，$\mathbf{A}$ 和 $\mathbf{B}$ 是系统矩阵和控制矩阵。

噪声模型定义如下：

$$ Q_k = \begin{bmatrix} Idt \cdot \text{noise}_p & 0 & 0 & 0 \\ 9.8Idt \cdot \text{noise}_v & 20 & 0 & 0 \\ 0 & 0 & Idt \cdot \text{noise}_{p1} & 0 \\ 0 & 0 & 0 & Idt \cdot \text{noise}_{p2} \end{bmatrix} $$

这里，I 是单位矩阵，noise_p 和 noise_v 分别是位置和速度的噪声参数。

3 Establish the Observation Equation：

状态向量 z 定义为：

$$ z = \left[p^0 - p_1^0, p^0 - p_1^0, v_1^0, v_2^0, z_1^0, z_2^0 \right]^T $$

分别为在 world 坐标系下 body 和 foot 的位置差，world 坐标系下 foot 的速度，world 坐标系下 foot 位置的 z 分量（特意加入 z 是因为 foot 的高度方向误差会直接影响机器人的质心高度和抬腿高度，在每次触地时清零以消除累计误差）

使用 Pinocchio 功能包可以直接获取机器人 urdf 文件中的模型文件和数据，然后初始化位置变量计算关节的雅可比矩阵和该帧的正向运动学，同理使用数据得到足部相对基座的位置变量，再使用雅可比矩阵与足部各个关节角速度相乘得到足部相对基座的速度变量。

设 foot 的位置在 world 坐标系下表示为：

$$^{o}p = ^{o} p_{\text{com}} + ^{O} R_B \cdot ^{B} p_i$$

为了求 world 坐标系下 foot 的速度，对上式求导：

$$\frac{dp_i^0}{dt} = ^{o} v_{\text{com}} + ^{o} R_B \left(^{B}w_{\text{OB}} \cdot ^{B} p_i + \frac{^{B}p_i}{t} \right)$$

由 foot 在支撑相下不打滑的假设，得：

$$\frac{dp_i^o}{dt} = ^{o} v_{\text{com}} + ^{o} R_B \left(^{B}w_{\text{OB}} \cdot p + \frac{^{B}p}{t} \right) = 0$$

得到 v_{com} 的表达式为：

$$^{o}v_{\text{com}} = ^{O} R_B \left(^{B}w_{\text{OB}} \cdot ^{B} p_i + \frac{^{B}p_i}{t} \right)$$

以下是状态更新矩阵的表示：

$$\begin{bmatrix} P - P_1 \\ P - P_2 \\ -^{o}R_B \cdot (^{B}w_{OB} \cdot ^{B} p_1 + ^{B}\dot{p}_1) \\ -^{o}R_B \cdot (^{B}w_{OB} \cdot ^{B} p_2 + ^{B}\dot{p}_2) \\ z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} I_{3\times3} & 0_{3\times3} & -I_{6\times6} \\ I_{3\times3} & 0_{3\times3} & \\ 0_{3\times3} & I_{3\times3} & 0_{6\times6} \\ 0_{3\times3} & I_{3\times3} & \\ 0_{1\times6} & 001 & 0_{1\times3} \\ 0_{1\times9} & 001 & 0_{1\times0} \end{bmatrix} \begin{bmatrix} \dot{P}_{\text{com}} \\ \dot{V}_{\text{com}} \\ \dot{P}_1 \\ \dot{P}_2 \end{bmatrix}$$

更新矩阵方程为：

$$z_{k+1} = C_{k+1} z_{k+1}$$

噪声矩阵 R 表达为：

$$R = \begin{bmatrix} I_{6\times6} \cdot \text{noise}_{p_{\text{foot}}} & 0 & \cdots & 0 \\ \vdots & I_{6\times6} \cdot \text{noise}_{v_{\text{foot}}} & \cdots & 0 \\ 0 & \cdots & 0 & I_{2\times2} \cdot \text{noise}_{z} \end{bmatrix}$$

由于足底里程计支撑相时 foot 在 world 坐标系下速度为 0 的假设，所以当 foot 在摆动相时需要额外做补充, 腿处于摆动象时，扩大腿部位置预测误差的协方差值，以测量值为准。

处理方法：引入每条腿的置信度 trust 和腿支撑状态相位（有一种通过计算足底触地的时间累计计算相位的大小，在程序中介绍），摆腿时 phase=0，支撑时 phase 在 0 到 1 之间变化：

信任度 *trust* 的函数根据相位 *phase* 和置信窗口 *trust window* 来定义，它的表达式如下：

$$\text{trust} = \begin{cases} \frac{\text{phase}}{\textit{trust window}} & \text{for phase} \leq \textit{trust window} \\ 1 & \text{for } \textit{trust window} < \text{phase} < \textit{1-trust window} \\ \frac{1-\text{phase}}{\textit{trust window}} & \text{for phase} \geq \textit{1-trust window} \end{cases}$$

world 坐标系下 foot 的速度模型：

$$v_i = (1 - \text{trust}) \cdot v_{\text{com}} + \text{trust} \cdot ({}^{O}R_B(v_{\text{rel}} + {}^{o}\omega_B \times r_{\text{rel}}))$$

其中 v_0 是假设速度得到的结果，v_{rel} 是 foot 在 body 坐标系下的速度，r_{rel} 是 foot 在 body 坐标系下的位置。

world 坐标系下 foot 在 Z 方向上的高度模型：

$$z_i = (1 - \text{trust}) \cdot (p_f(2) + p_0(2))$$

其中 p_f 表示 foot 在 world 坐标系下的位置，p_0 表示 body 在 world 坐标系下的位置。且上式只在摆动相时不为 0，在支撑相时为了消除累计误差恒为 0。

4 Kalman Filter

将位置、速度和高度信息合并到观测向量 y:

$$y = \begin{bmatrix} p_{rel} \\ v \\ z \end{bmatrix}$$

状态估计更新公式：

$$\hat{x} = A\hat{x} + Ba$$

预测协方差 P_m:

$$P_m = APA^T + Q$$

预测的观测 y_{Model}:

$$y_{\text{Model}} = C\hat{x}$$

观测残差 e_y:

$$e_y = y - y_{\text{Model}}$$

观测协方差 S:

$$S = CP_mC^T + R$$

确保对称性:

$$S = \frac{1}{2}(S + S^T)$$

卡尔曼增益 K:

$$K = P_m C^T S^{-1}$$

状态更新:

$$\hat{x} = \hat{x} + K e_y$$

协方差更新:

$$P = (I - KC)P_m$$

确保对称性:

$$P = \frac{1}{2}(P + P^T)$$

对协方差矩阵的子矩阵进行校正，避免数值问题:

$$\text{如果 } \det(P[:2,:2]) > 1 \times 10^{-6}\text{，则进行校正}$$

$$P[:2,:2] = \frac{P[:2,:2]}{10}, \quad P[:2,2:] = 0, \quad P[2:,:2] = 0$$

5 Compare state estimator results against the true state:

在 x 方向上，估计速度与真实速度对比：

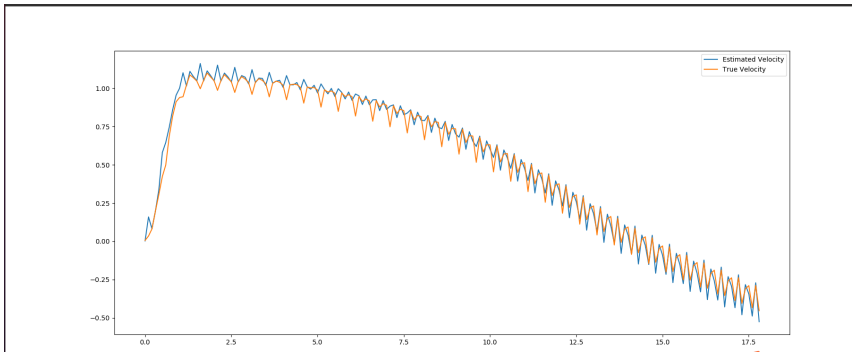

Figure 1: x：Estimated velocity vs True velocity

在 y 方向上，估计速度与真实速度对比：

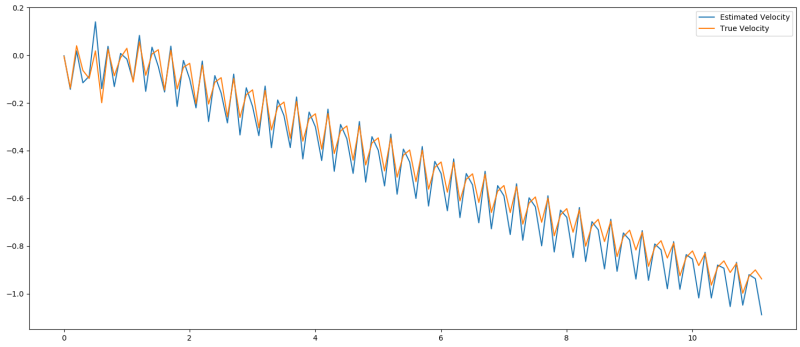

Figure 2: y：Estimated velocity vs True velocity

在 z 方向上，估计速度与真实速度对比：

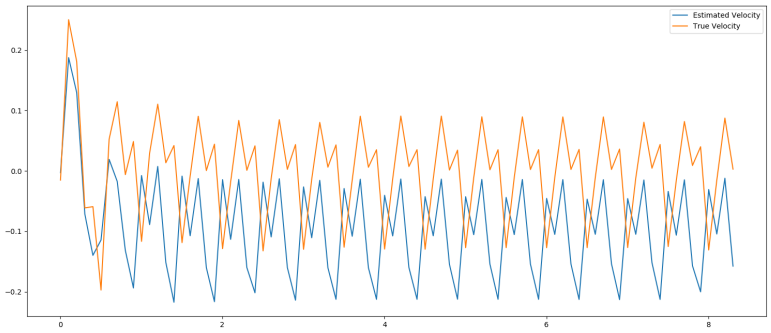

Figure 3: z：Estimated velocity vs True velocity

6 Try different Covariance Matrices to see how they impact Estimation Error

在代码中，通过改变传感器的噪声大小，高怀疑数和信任度来改变 Q 和 R 的协方差矩阵：

初始化过程噪声协方差矩阵 Q：

$$Q = \begin{bmatrix} \mathrm{I}[3:3,3:3] \cdot \mathrm{process\_noise\_pimu} & 0 & 0 \\ 0 & \mathrm{I}[3:6,3:6] \cdot \mathrm{process\_noise\_vimu} & 0 \\ 0 & 0 & \mathrm{I}[6:,6:] \cdot \mathrm{process\_noise\_pfoot} \end{bmatrix}$$

初始化测量噪声协方差矩阵 R：

$$R = \begin{bmatrix} \mathrm{I}[6:6,6:6] \cdot \mathrm{sensor\_noise\_pimu\_rel\_foot} & 0 & 0 \\ 0 & \mathrm{I}[6:12,6:12] \cdot \mathrm{sensor\_noise\_vimu\_rel\_foot} & 0 \\ 0 & 0 & \mathrm{I}[12:,12:] \cdot \mathrm{sensor\_noise\_zfoot} \end{bmatrix}$$

我们可以通过设置信任度窗口来计算整体的信任度：

$$\text{trust} = \begin{cases} \frac{\text{phase}}{\textit{trust window}} & \text{for phase} \leq \textit{trust window} \\ 1 & \text{for } \textit{trust window} < \text{phase} < \textit{1-trust window} \\ \frac{1-\text{phase}}{\textit{trust window}} & \text{for phase} \geq \textit{1-trust window} \end{cases}$$

更新过程：

$$Q_{\text{sub}} = Q[qindex : qindex + 3, qindex : qindex + 3] \text{ （提取 Q 的子矩阵）}$$

$$Q_{\text{sub}} \leftarrow Q_{\text{sub}} \cdot (1 + (1 - \text{trust}) \cdot \text{高怀疑数}) \text{ （调整子矩阵）}$$

$$R_{\text{sub1}} = R[rindex1 : rindex1 + 3, rindex1 : rindex1 + 3] \text{ （提取 R 的第一个子矩阵）}$$

$$R_{\text{sub1}} \leftarrow R_{\text{sub1}} \cdot (1 + (1 - \text{trust}) \cdot \text{高怀疑数})$$

$$R_{\text{sub2}} = R[rindex2 : rindex2 + 3, rindex2 : rindex2 + 3] \text{ （提取 R 的第二个子矩阵）}$$

$$R_{\text{sub2}} \leftarrow R_{\text{sub2}} \cdot (1 + (1 - \text{trust}) \cdot \text{高怀疑数})$$

$$R_{\text{sub3}} = R[rindex3, rindex3] \text{ （提取 R 的第三个子矩阵，这里是一个元素）}$$

$$R_{\text{sub3}} \leftarrow R_{\text{sub3}} \cdot (1 + (1 - \text{trust}) \cdot \text{高怀疑数})$$

第一种情况：

process_noise_pimu = 0.002
process_noise_vimu = 0.002
process_noise_pfoot = 0.002
sensor_noise_pimu_rel_foot = 0.00000002
sensor_noise_vimu_rel_foot = 0.00000002
sensor_noise_zfoot = 0.00000002
high_suspect_number = 1000000.0
trust_window=0.2
在 x 方向上，估计误差：

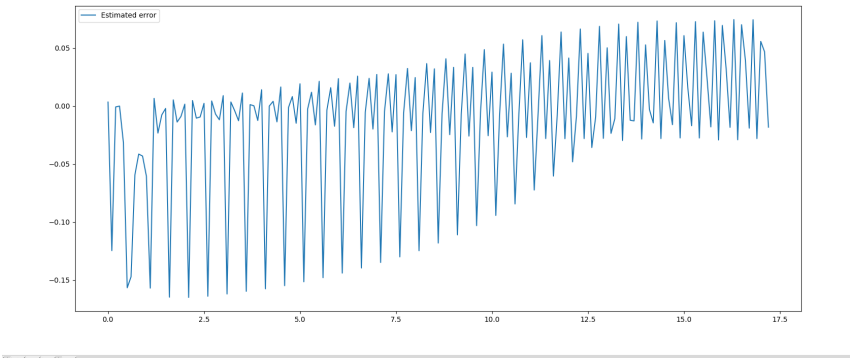

Figure 4: x：Estimated error

第二种情况:

process_noise_pimu = 0.02

process_noise_vimu = 0.02

process_noise_pfoot = 0.02

sensor_noise_pimu_rel_foot = 0.000002

sensor_noise_vimu_rel_foot = 0.000002

sensor_noise_zfoot = 0.000002

high_suspect_number = 1000000.0

trust_window=0.2

在 x 方向上，估计误差:

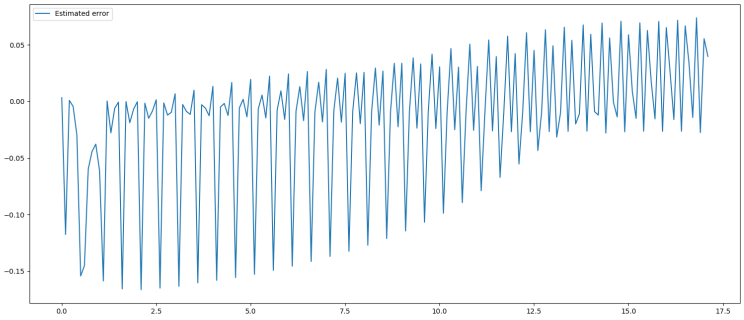

Figure 5: x：Estimated error

第三种情况:

process_noise_pimu = 0.02

process_noise_vimu = 0.02

process_noise_pfoot = 0.02

sensor_noise_pimu_rel_foot = 0.002

sensor_noise_vimu_rel_foot = 0.002

sensor_noise_zfoot = 0.002

high_suspect_number = 1000000.0

trust_window=0.2

在 x 方向上，估计误差：

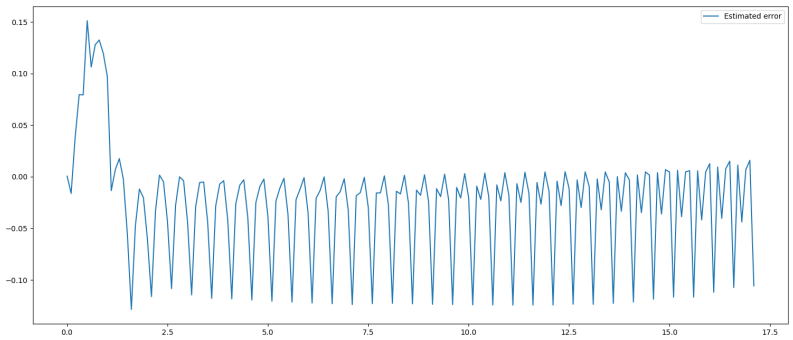

Figure 6: x：Estimated error

分析：

根据第一、二、三种情况可知当噪音的方差逐渐增大时，估计的速度准确性逐渐降低，尤其是在增加了传感器测量噪声方差之后，估计的误差一直处于 0 到-0.15 之间，无法改善。相反在传感器测量噪声方差较少时，估计误差随着时间推移逐渐向 0 靠近，在一定时间后误差在 0 附近波动，范围为-0.05 到 0.05，说明估计速度与真实速度近似相同。

第四种情况：

process_noise_pimu = 0.002

process_noise_vimu = 0.002

process_noise_pfoot = 0.002

sensor_noise_pimu_rel_foot = 0.00000002

sensor_noise_vimu_rel_foot = 0.00000002

sensor_noise_zfoot = 0.00000002

high_suspect_number = 10000.0

trust_window=0.2

在 x 方向上，估计误差：

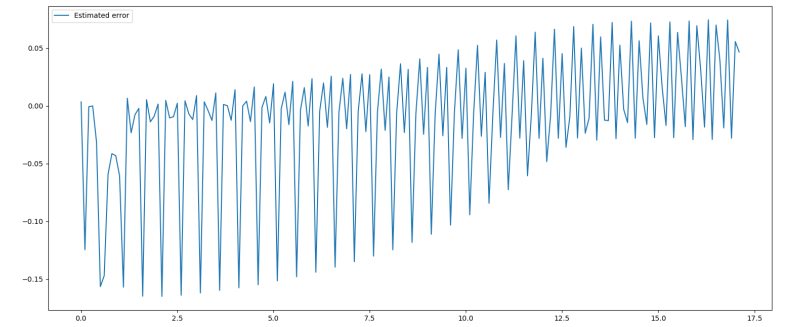

Figure 7: x：Estimated error

第五种情况：

process_noise_pimu = 0.002

process_noise_vimu = 0.002

process_noise_pfoot = 0.002

sensor_noise_pimu_rel_foot = 0.00000002

sensor_noise_vimu_rel_foot = 0.00000002

sensor_noise_zfoot = 0.00000002

high_suspect_number = 10.0

trust_window=0.2

在 x 方向上，估计误差：

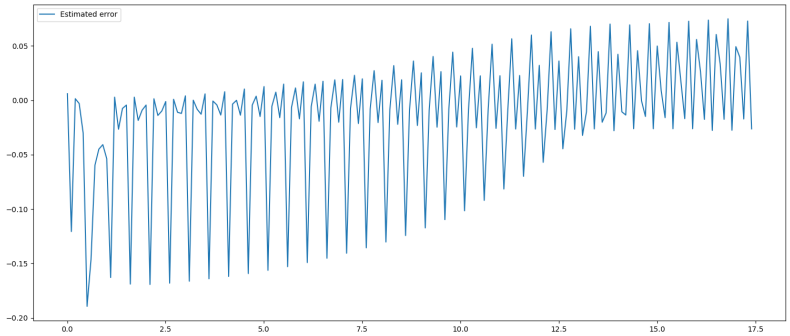

Figure 8: x：Estimated error

分析：

根据第四、五种情况可知，改变高怀疑数对估计误差的影响并不明显。

第六种情况：

process_noise_pimu = 0.02

process_noise_vimu = 0.02

process_noise_pfoot = 0.02

sensor_noise_pimu_rel_foot = 0.000002

sensor_noise_vimu_rel_foot = 0.000002

sensor_noise_zfoot = 0.000002

high_suspect_number = 1000000.0

trust_window=0.3

在 x 方向上，估计误差：

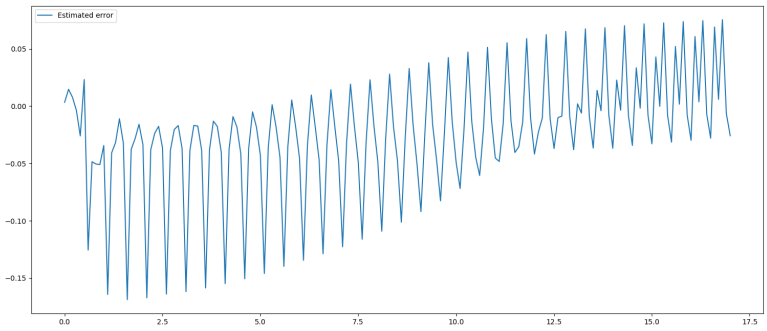

Figure 9: x：Estimated error

第七种情况：

process_noise_pimu = 0.02

process_noise_vimu = 0.02

process_noise_pfoot = 0.02

sensor_noise_pimu_rel_foot = 0.000002

sensor_noise_vimu_rel_foot = 0.000002

sensor_noise_zfoot = 0.000002

high_suspect_number = 1000000.0

trust_window=0.1

在 x 方向上，估计误差：

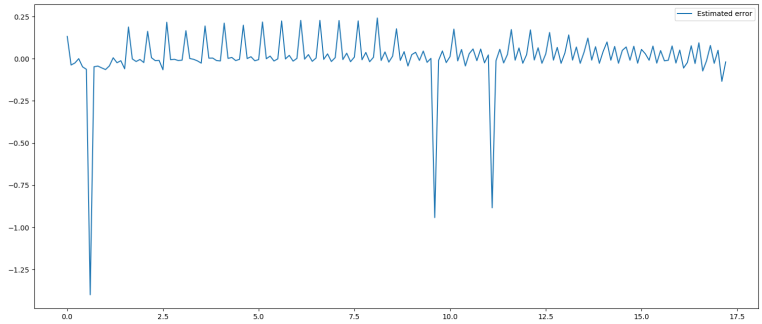

Figure 10: x：Estimated error

第八种情况：

process_noise_pimu = 0.02

process_noise_vimu = 0.02

process_noise_pfoot = 0.02

sensor_noise_pimu_rel_foot = 0.000002

sensor_noise_vimu_rel_foot = 0.000002

sensor_noise_zfoot = 0.000002

high_suspect_number = 1000000.0

trust_window=0.4

在 x 方向上，估计误差：

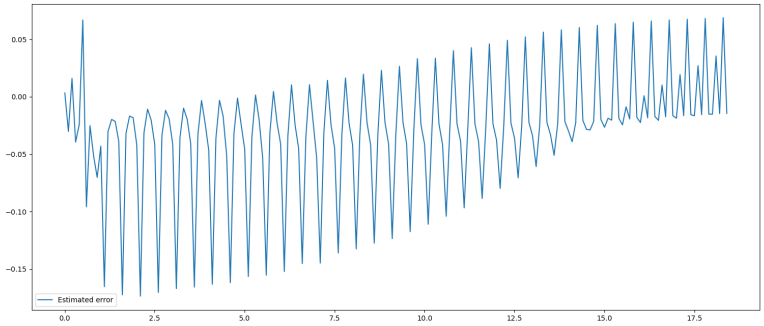

Figure 11: x：Estimated error

分析：由第一、六、七、八种情况对比与分析，我们可以看出当置信窗口为 0.1 的时候，稳定后的误差较大，在 0.25 到-0.25 之间震荡，而置信窗口 0.2 及以上的时候稳定误差为 0.05 到-0.05 之间震荡，且在置信度为 0.2 和 0.3 时稳定误差较小。因此可以认为置信窗口在 0.2-0.3 时，估算出来的速度更为准确。

总结：在这种情况下：process_noise_pimu = 0.002

process_noise_vimu = 0.002

process_noise_pfoot = 0.002

sensor_noise_pimu_rel_foot = 0.00000002

sensor_noise_vimu_rel_foot = 0.00000002

sensor_noise_zfoot = 0.00000002

high_suspect_number = 1000000.0

trust_window=0.2（0.2-0.3）

卡尔曼滤波器的表现最好，此时估计出来的速度更为准确

Problem II: RL training of inverted singular and double pendulums

1 背景信息

倒立摆是一个典型的非线性、强耦合、自然不稳定的系统。自 20 世纪 60 年代以来，倒立摆系统一直是控制理论领域的研究热点。倒立摆的控制问题不仅涉及到非线性、鲁棒性、随动性等控制理论中的关键问题，而且其控制过程能有效反映控制理论在实际应用中的有效性。倒立摆控制原理在多个领域得到广泛应用。在游戏角色的移动控制中，

倒立摆原理使得角色行走更逼真；在空中机器人控制中，倒立摆原理实现了机器人的稳定飞行；在人形机器人设计中，倒立摆帮助机器人保持行走平衡；此外，倒立摆还应用于自主驾驶车辆的动态控制和稳定性控制中。而倒立摆作为控制理论教学和科研中的物理模型，其重要性不言而喻。它不仅作为姿态控制的装置，用于控制机器人、飞行器等复杂系统的稳定性，还作为非线性系统的研究工具，在控制系统设计中起到了关键作用。通过倒立摆的研究，我们可以深入理解控制原理，提高控制性能，为各个领域的发展提供基础。

强化学习算法作为一种机器学习的重要分支，通过智能体与环境间的不断交互与试错学习，来优化决策过程。它在机器人控制、游戏对战等领域已有广泛应用，如 AlphaGo 成功击败人类围棋高手，展示了其处理复杂决策问题的优越性。随着人工智能的不断发展，强化学习算法在无人驾驶、智能家居等领域的应用前景也日益广阔。其重要性在于，通过智能体的主动探索与反馈学习，能够推动系统性能的持续提升，为智能系统设计和应用提供新的可能性。

项目目标：本项目旨在通过强化学习的算法分别实现一阶倒立摆在微小扰动下的平衡控制与起摆后的平衡控制

2 平衡控制问题的原理

：在第一小问中，我们主要使用了 reinforce 算法来实现一阶倒立摆在微小扰动情况下的平衡。REINFORCE 算法（也称为蒙特卡洛策略梯度算法）是 Policy Gradient 算法的一种，它基于蒙特卡洛（MC）更新方式。MC 更新方式意味着算法在每个完整的 episode（或轨迹）结束后才进行更新，而不是在每个时间步或每个 mini-batch 后进行更新。策略梯度算法的核心思想是先定义一个评价指标（通常是期望回报），然后利用随机梯度上升（或梯度下降，取决于是最大化还是最小化目标）的方法来更新策略的参数，以使得这个评价指标不断上升（或下降）。在 REINFORCE 算法中，评价指标通常是累积折扣奖励的期望，即：

$$J(\theta) = E_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \gamma^t r_t \right]$$

其中，τ 是从策略 π_θ 采样得到的轨迹，r_t 是时刻 t 的奖励，γ 是折扣因子，θ 是策略的参数。

随机梯度上升算法用于更新策略参数　的公式可以表示为：

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

但在 REINFORCE 算法中，由于期望的计算通常是不可行的（因为它涉及到对整个状态-动作空间的遍历），我们采用蒙特卡洛采样的方法来近似这个期望。具体来说，我们从一个轨迹中采样得到一系列的奖励，并使用这些奖励来近似累积折扣奖励的梯度。

因此，REINFORCE 算法的更新公式可以写为：

$$\theta \leftarrow \theta + \alpha G_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

其中, G_t 是从时间 t 开始的累积折扣奖励（即 $G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$）, $\pi_\theta(a_t|s_t)$ 是在状态 s_t 下采取动作 a_t 的概率。

3 实现的类与方法

3.1 Agent 类介绍

`Agent` 类是一个用于学习解决倒立摆任务的智能体, 它基于策略梯度算法进行学习。以下是该类的主要特点:

(1) **init**: 在初始化时，智能体设置学习率、折扣因子和一个小数值用于数学稳定性。它初始化了一个空的列表来存储采样动作的概率值和对应的奖励值。此外，智能体还创建了一个策略网络和一个 AdamW 优化器。

(2) **sample action**: 根据当前状态，智能体使用策略网络采样一个动作。首先，它将状态转换为 PyTorch 张量，并传递给策略网络以获取动作均值和标准差。然后，它根据这些均值和标准差创建一个正态分布，并从该分布中采样一个动作。采样得到的动作被转换为 NumPy 数组，并存储其概率值以备后用。

(3) **update**: 该方法是 `Agent` 类中的一个重要方法，它使用 REINFORCE 算法根据收集的奖励和动作的对数概率来更新策略网络。该方法的主要流程如下:

   (1) 初始化当前累积回报（*running_g*）为 0，并创建一个空列表（*gs*）来存储折扣回报。

   (2) 通过逆序遍历收集的奖励列表，计算每个时间步的折扣回报，并将它们插入到 *gs* 列表的开头。

   (3) 将 *gs* 列表转换为 PyTorch 张量，存储在 *deltas* 变量中。

   (4) 初始化损失（*loss*）为 0，并遍历对数概率列表和折扣回报张量。对于每对对数概率和折扣回报，计算损失项并将其累加到 *loss* 中。

   (5) 清空优化器的梯度缓存，并对 *loss* 执行反向传播以计算梯度。

   (6) 使用优化器（AdamW）更新策略网络的参数。

   (7) 清空存储的奖励（*rewards*）和对数概率（*probs*）列表，为下一个 episode 做准备。

3.2 Policy_Network 类介绍

`Policy_Network` 类是一个神经网络模型，用于参数化策略，通过预测动作分布的参数来指导决策。以下是该类的主要特点和功能：

1. **初始化 (`__init__` 方法)**：
   (a) 接收两个主要参数：观察空间的维度 (`obs_space_dims`) 和动作空间的维度 (`action_space_dims`)。
   (b) 定义了神经网络层，包括两个隐藏层和激活函数（双曲正切），以及用于预测动作分布均值和标准差的全连接层。
2. **前向传播 (`forward` 方法)**：
   (a) 接收一个状态观测值 `x` 作为输入。
   (b) 通过神经网络进行前向传播，使用双曲正切激活函数。
   (c) 预测并返回动作分布的均值 (`action_means`) 和标准差 (`action_stddevs`)。注意，标准差通过一个指数函数进行处理，以确保其值始终为正。

这个类的主要目的是，基于给定的状态观测，预测出一个合理的动作分布，从而指导智能体在强化学习环境中的行为。

3.3 训练部分

对于每个回合（从 1 到总回合数）：

1. 重置环境以获取初始观测和信息。
2. 初始化回合奖励列表和日志概率列表。
3. 当回合未结束时：
   (a) 从代理中采样动作。
   (b) 在环境中执行动作并获取新的观测、奖励和终止标志。
   (c) 将奖励添加到代理的奖励列表和回合奖励列表中。
   (d) 检查回合是否终止。
4. 计算回合总奖励并添加到奖励列表中。
5. 更新代理的策略网络。
6. 如果总奖励达到或超过奖励阈值，训练终止，保存此时的策略网络。
7. 如果回合是 25 的倍数，计算最近 50 个回合的平均奖励并打印信息以便观察训练情况。

3.4 奖励函数设计

为了控制倒立摆能在竖直向上的位置保持较长时间，我们将 gymnasium 库中的环境进行打包，自定义设计了奖励函数。设计的奖励函数综合考虑了五个部分：存活时间、倒立摆角度、小车位置和倒立摆角速度和小车速度。

1. **存活奖励**
   智能体每一步都会获得一个固定的奖励，用来鼓励智能体尽可能长时间地存活在任务中。当倒立摆的角度达到终止条件后，即此回合控制失败，存活奖励是为了鼓励每个回合可以运行更多的时间步数。

2. **角度奖励**
   角度奖励根据智能体的倾斜角度来计算。初始奖励是根据角度的余弦值来确定的，表示智能体越接近垂直位置，奖励越高。如果智能体的倾斜角度超过了预设的阈值，则会给予一个大的负奖励，表示智能体在这个状态下表现非常差。

3. **位置奖励**
   位置奖励根据智能体在水平位置的偏移量来计算。如果智能体的位置在预设的阈值范围内，则奖励是一个正值，表示智能体越接近中心位置，奖励越高。如果位置超过了这个阈值范围，则奖励是一个负值，表示位置越远离中心位置，惩罚越大。

4. **速度惩罚**
   速度惩罚根据智能体的水平速度和角速度来计算。惩罚的目的是鼓励智能体保持低速稳定，而不是快速变化。速度越高，惩罚越大。

5. **综合奖励**
   最终的自定义奖励是将上述的存活奖励、角度奖励和位置奖励相加，再减去速度惩罚。通过这种设计，奖励函数综合考虑了智能体的多个状态变量，目的是鼓励智能体在垂直、接近中心位置且低速稳定的情况下尽可能长时间地存活。

3.5 训练结果与分析

通过反复的调整奖励函数的每部分奖励或惩罚的数值，我们得到了最终的合适的奖励函数。经过训练，在 2334 次训练起，训练结果已经达到我们的目标，为了实现更长时间的控制，我们选择了第 2337 次训练的结果。整个训练过程的奖励变化如下：

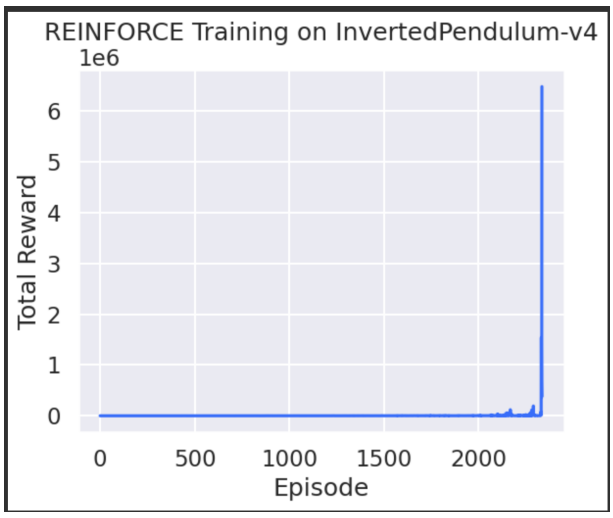

Figure 12: reward

使用经过 2337 次训练得到的模型进行仿真，我们能够实现远超过 10 分钟对倒立摆的控制。但是由于 Reinforce 算法的限制，没有办法实现倒立摆的竖直向上的稳定状态，只能尽可能保持其不倒下。但是在第二问中，我们改变了算法，通过学习，能够实现倒立摆的稳定。

4 倒立摆起摆问题

在第二小问中，我们最开始使用 reinforce 方法来进行控制，但是此方法在进行了多次学习尝试时候并不能成功。起先我们尝试使用过程分解的方法来完成完整任务，分别为起摆部分和稳摆部分：我们尝试使用同样结构的策略网络，使用奖励函数将倒立摆从竖直向下摆动到距离竖直向上的位置有较小角度和加速度，但是我们始终无法设计出合适的奖励函数与终止条件来达到目标。之后我们又尝试将起摆的步骤分解为摆到水平位置以上以及继续上摆并控制角速度两个部分，但很遗憾，在第一步中我们发现 reinforce 的算法很难使得 reward 收敛，过程中的训练图片如下：

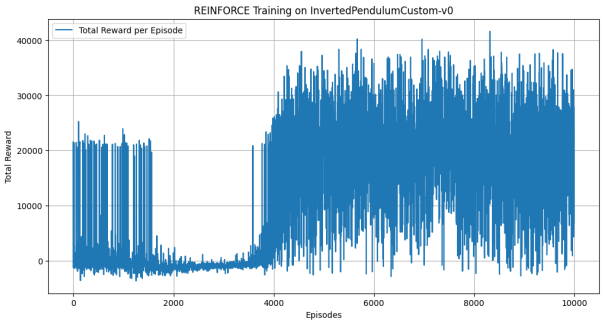

Figure 13: reward

最终，通过寻找资料，我们使用了 DDPG 方法来进行尝试。

4.1 DDPG 方法介绍

DDPG 是一种基于深度学习和确定性策略梯度的强化学习算法，适用于解决连续动作空间的问题。其核心思想是通过优化价值网络和策略网络，使得智能体能够在给定的状态下输出最优的连续动作。

4.2 网络结构

DDPG 的网络结构由四个网络组成：当前 Actor 网络、目标 Actor 网络、当前 Critic 网络和目标 Critic 网络。

- 当前 Actor 网络和目标 Actor 网络结构相同，输入为状态，输出为动作。当前 Actor 网络根据当前状态输出当前动作，而目标 Actor 网络根据下一状态输出下一动作。
- 当前 Critic 网络和目标 Critic 网络结构也相同，输入为状态和动作，输出为该状态下采取该动作的 Q 值。当前 Critic 网络根据当前状态和当前动作输出当前 Q 值，而目标 Critic 网络根据下一状态和下一动作输出下一 Q 值。

4.3 算法流程

DDPG 算法的基本工作流程包括以下几个步骤：

1. 初始化当前 Actor 网络、当前 Critic 网络，并将它们的参数分别复制到目标 Actor 网络和目标 Critic 网络。
2. 使用当前 Actor 网络与环境进行交互，生成经验样本。每个样本包括当前状态、当前动作、奖励和下一状态。
3. 使用这些样本更新当前 Critic 网络的参数，使其能够更准确地估计 Q 值。
4. 通过确定性策略梯度算法更新当前 Actor 网络的参数，使其输出的动作在价值网络那里能够得到更高的评分。
5. 定期将当前 Actor 网络和当前 Critic 网络的参数复制到目标 Actor 网络和目标 Critic 网络，以实现稳定的学习过程。

5 起摆问题使用的类与方法

5.1 Critic 类

描述: 该 'Critic'类是一个神经网络模型，用于估计给定状态-动作对的值函数（Q 值）。

属性:

- obs_dim: 观测空间的维度。
- action_dim: 动作空间的维度。

网络结构:

- 第一层线性层，输入维度为 obs_dim，输出维度为 1024。
- 第二层线性层，输入维度为 1024 与 action_dim 之和，输出维度为 512。
- 第三层线性层，输入维度为 512，输出维度为 300。
- 输出层线性层，输入维度为 300，输出维度为 1（Q 值）。

方法:

- forward(x, a): 计算给定观测 x 和动作 a 的 Q 值。

5.2 Actor 类

描述: 该 'Actor'类是一个神经网络模型，用于根据给定状态输出动作。
属性:

- obs_dim: 观测空间的维度。
- action_dim: 动作空间的维度。

网络结构:

- 第一层线性层，输入维度为 obs_dim，输出维度为 512。
- 第二层线性层，输入维度为 512，输出维度为 128。
- 输出层线性层，输入维度为 128，输出维度为 action_dim。

方法:

- forward(obs): 根据给定观测 obs 输出动作。输出的动作经过 tanh 函数缩放，并乘以一个常数（这里是 10）来确保动作在合适的范围内。

5.3 DDPGAgent 类介绍

DDPGAgent 类实现了深度确定性策略梯度（Deep Deterministic Policy Gradient, DDPG）算法，该算法用于解决连续动作空间中的强化学习问题。以下是该类的详细介绍。

构造函数

```
__init__(self, env, gamma, tau, buffer_maxlen, critic_learning_rate,
         actor_learning_rate, train, decay, device)
```

构造函数初始化了 DDPGAgent 对象，并设置了相关的超参数和网络。

方法

get_action

```
get_action(self, obs, t=0)
```

根据当前状态选择动作。

- **obs**: 当前观察到的状态。
- **t**: 当前时间步。

该方法将状态输入到演员网络，得到相应的动作。如果处于训练模式，还会添加探索噪声。

update

```
update(self, batch_size)
```

从经验回放缓冲区中采样一批数据，并更新评论家和演员网络。

- **batch_size**: 采样批次的大小。

该方法首先计算当前 Q 值和目标 Q 值之间的均方误差（MSE），并更新评论家网络。然后，计算演员网络的策略损失，并更新演员网络。最后，使用软更新规则更新目标网络。

5.4 BasicBuffer 类介绍

'BasicBuffer'类实现了一个基本的经验回放缓冲区，用于存储强化学习过程中智能体的交互经验。以下是该类的详细介绍。

构造函数

```
__init__(self, max_size)
```

构造函数初始化了 BasicBuffer 对象，并设置了缓冲区的最大容量。

- **max_size**：缓冲区的最大容量，即可以存储的经验条目的最大数量。

成员变量

- **max_size**：缓冲区的最大容量。
- **buffer**：使用 `deque` 数据结构实现的循环缓冲区，用于存储经验条目。

方法

push

```
push(self, state, action, reward, next_state, done)
```

将一个新的经验条目推入缓冲区。

- **state**：当前状态。
- **action**：智能体采取的动作。
- **reward**：智能体获得的奖励。
- **next_state**：智能体执行动作后的下一个状态。
- **done**：布尔值，指示当前回合是否结束。

该方法将经验条目存储为元组 `(state, action, reward, next_state, done)`,并将其添加到缓冲区中。

sample

```
sample(self, batch_size)
```

从缓冲区中随机采样一批经验条目。

- **batch_size**：采样批次的大小。

该方法返回五个列表，分别包含采样的状态、动作、奖励、下一个状态和回合结束标志。

```
return (state_batch, action_batch, reward_batch, next_state_batch, done_batch)
```

__len__

```
__len__(self)
```

返回缓冲区中当前存储的经验条目的数量。

- 该方法返回缓冲区的长度，即缓冲区中存储的经验条目的数量。

5.5 训练方法介绍

本文中的 `train` 函数实现了一个基于 DDPG (Deep Deterministic Policy Gradient) 算法的训练过程，用于训练智能体在 `InvertedPendulumCustom-v0` 环境中进行控制任务。函数的详细介绍如下：

函数参数

- **batch_size**：每个训练批次的大小，默认为 128。
- **critic_lr**：评论家网络的学习率，默认为 1×10^{-3}。
- **actor_lr**：演员网络的学习率，默认为 1×10^{-4}。
- **max_episodes**：训练的最大回合数，默认为 1000。
- **max_steps**：每回合的最大步数，默认为 500。
- **gamma**：折扣因子，默认为 0.99。
- **tau**：软更新参数，默认为 1×10^{-3}。
- **buffer_maxlen**：经验回放缓冲区的最大长度，默认为 100000。

训练过程

- 首先，函数通过调用 `make` 函数创建一个名为 `InvertedPendulumCustom-v0` 的环境。
- 然后，基于提供的超参数，创建一个 `DDPGAgent` 智能体。
- 接着，调用 `mini_batch_train` 函数进行训练，并记录每个回合的总奖励。
- 训练完成后，函数会生成一个奖励随回合变化的曲线图，并保存图像。
- 最后，如果当前目录下没有 `models` 文件夹，函数会创建该文件夹，并将训练好的智能体模型保存为 `pendulum_swingup_ddpg.pkl` 文件。

6 奖励函数的设计

奖励函数的设计目标是评估系统当前状态与目标状态之间的差距，并通过负奖励值鼓励系统朝着目标状态进行调整。具体方法如下：

6.1 定义矩阵和状态向量

首先，定义了一个 3x3 矩阵 invT，其中包含系统的参数。然后，定义当前状态向量 **j**，它包含位置 x 以及角度 θ 的正弦和余弦值，同时定义目标状态向量 $\mathbf{j}_{\text{target}}$，表示系统希望达到的状态，即位置为 0 且角度为 0 度（直立位置）。

6.2 计算差异和奖励

计算当前状态与目标状态之间的差异向量，然后将其与矩阵 invT 相乘，再将结果与差异向量相乘，得到一个标量值。这个标量值被代入一个负指数函数，最终得到的奖励值用于评估当前状态的优劣。

6.3 奖励机制

这个奖励函数采用负指数形式，使得当系统状态接近目标状态时，奖励值趋近于 0，而当状态偏离目标状态时，奖励值为负且绝对值增大。通过这样的设计，系统被鼓励不断调整自身，尽量接近目标状态，以获得更高的奖励（即更少的负值）。

7 训练结果

通过训练部分设置的早停策略，在训练 250 个回合之后，我们得到了学习好的模型。训练过程中的 reward 变化如下：

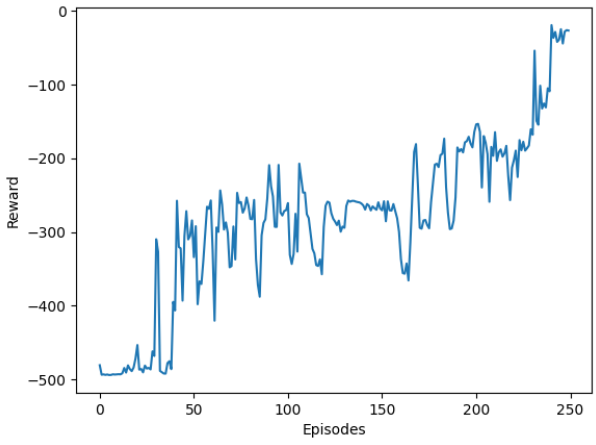

Figure 14: plot action

通过仿真我们可以看到，训练出的模型能够很好的完成任务，在 100 个时间步以内，倒立摆就能实现从竖直向下到摆起并在竖直向上的位置保持稳定的全部过程。

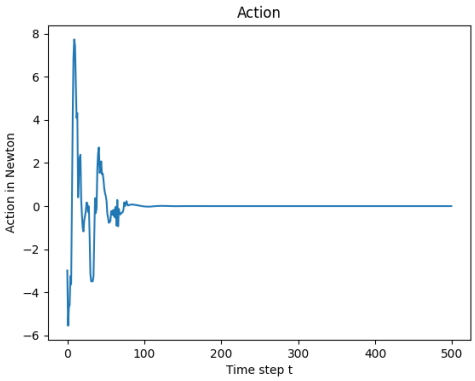

Figure 15: Plot action

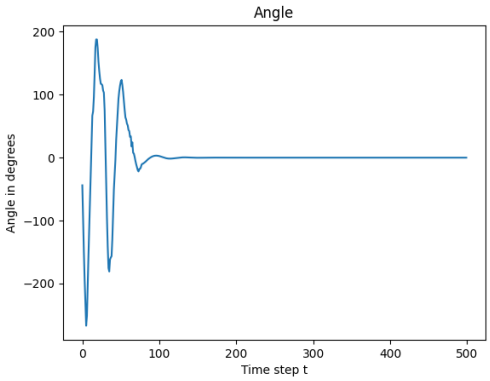

Figure 16: Plot angle

8 RL training of inverted double pendulums

我们并未成功实现倒立双摆的控制，但在探索的过程中我们找到了一些或许能够完成此任务的强化学习算法：PILCO（即 Probabilistic Inference and Learning for Control）算法，以下是对于此算法的一些介绍

8.1 PILCO 算法的原理

PILCO（Probabilistic Inference and Learning for Control）是一个针对复杂控制任务的数据高效强化学习方法。它通过学习系统的概率动力学模型，并利用该模型进行预测和控制。

8.2 系统动力学学习

PILCO 通过测量数据学习系统的动力学模型，该模型将当前状态 x_t 和控制输入 u_t 的联合分布映射到后继状态 x_{t+1} 的分布上，即：

$$\Phi : p(x_t, u_t) \mapsto p(x_{t+1})$$

其中，$p(x_{t+1})$ 是真实分布的高斯近似（通过矩匹配得到）。

对于 GP 模型，先验均值函数设置为零，对应于对基础动力学函数的无先验知识，同时使用加性噪声的平方指数（SE）协方差函数。

8.3 控制策略优化

对于控制策略 $u_t = \pi(x_t; \theta)$，PILCO 使用确定性高斯过程（GP），其中后验方差设置为零。这确保了当前状态唯一地映射到特定的输入向量。

GP 控制器依赖于多个参数，包括伪输入、伪目标和权重矩阵。

8.4 成本函数

立即成本函数 $c(x_t)$ 可以由不同的子成本函数组成，如：

$$c(x_t) = \sum_i C_i(x_t)$$

PILCO 的原始版本考虑了特定的成本函数，如饱和二次函数，用于实现仿真中无限长轨道上双摆的摆动和平衡。但在现实应用中，还需要考虑状态空间约束，因此引入了另一种成本函数——双铰链函数。

8.5 PILCO 算法流程

Algorithm 1: PILCO算法框架

1: 初始化控制器参数为随机值
2: 对系统应用一个随机控制序列
3: 根据当前可用的测量数据学习系统动力学模型
4: 优化控制器参数以最小化当前的成本函数
5: 重复3、4步直到优化完毕