정리:

1) 토크 수준으로 제어 (참고 문헌 참조, 예: 200 Hz)
   - observation + last_action
   - action space 범위 지정 (최소, 최대 값)
   - SAC vs PPO (or DDPG)

2) baseline : rule-based (extened kalman filter 깃헙 코드 같은)

3) IMU 지금 수준을 왜곡 적용 (orientation, linear acceleration, angular velocity)

4) 학습 & 테스트 시나리오 동일. 학습시 20, 20, 20 ==> 0, 0, 1

1. 토크 수준의 기체 제어

moveByMotorPWMsAsync(*front_right_pwm, rear_left_pwm, front_left_pwm, rear_right_pwm, duration, vehicle_name=''*) [source]

- Directly control the motors using PWM values

Parameters:
- **front_right_pwm** (*float*) – PWM value for the front right motor (between 0.0 to 1.0)
- **rear_left_pwm** (*float*) – PWM value for the rear left motor (between 0.0 to 1.0)
- **front_left_pwm** (*float*) – PWM value for the front left motor (between 0.0 to 1.0)
- **rear_right_pwm** (*float*) – PWM value for the rear right motor (between 0.0 to 1.0)
- **duration** (*float*) – Desired amount of time (seconds), to send this command for
- **vehicle_name** (*str, optional*) – Name of the multirotor to send this command to

해당 방식은 pwm(펄스 폭 변조) 방식을 사용하는데 이를 토크 수준에서 추력을 설정하게 하려면 다음과 같은 코드 사용이 필요

```
Eigen::Vector4f forces_to_pwm(const Eigen::Vector4f &f_u,const msr::airlib::Environment::State &env)
{
    float propeller_diameter = 0.2286, standard_air_density = 1.225, d = 0.2275; //d is arm_length
    float c_T = 0.109919*pow(propeller_diameter, 4)*standard_air_density;
    float c_Q = 0.040164*pow(propeller_diameter, 5)*standard_air_density/(2*M_PI);

    Eigen::Matrix<float,4,4> M;
    M <<    c_T,    c_T,   c_T,    c_T,
         -d*c_T,  d*c_T, d*c_T, -d*c_T,
          d*c_T, -d*c_T, d*c_T, -d*c_T,
            c_Q,    c_Q,  -c_Q,   -c_Q;

    Eigen::Vector4f thrust =  c_T*M.inverse()*f_u;

    float max_thrust = 4.179446268;
    float air_density_ratio = env.air_density/standard_air_density;
    Eigen::Vector4f pwm;

    for(int i = 0;i<thrust.size();++i){
        pwm[i] = std::max<float>(0.0,std::min<float>(1.0,thrust[i]/(air_density_ratio*max_thrust)));
    }
    return pwm;
}
```

해당 방식을 통해서 토크 수준의 드론 제어가 가능
- 앞에서 언급한 pwm api만을 사용해 간단한 이동 테스트를 진행해보았는데 값 변화에 매우 민감한 모습을 보임, 토크를 이용했을 때에는 어떤지는 아직 확인X => 적절한 action space 찾는 것이 중요

2. imu raw data
   imu sensor에는 크게 자이로센서와 가속도 센서를 통해 기체의 각속도와 선형 가속도를 측정
   orientation 값은 일반적으로는 자이로 센서에서 얻어진 각속도를 시간에 따라 적분하면 값을 추출할 수 있지만 실제에서는 오류 값 또한 적분되기 때문에 오차가 점점 늘어남 => 따라서 실제로는 가속도, 자이로, 지자계 센서 조합을 통해 계산 (하지만 이러한 부분은 airsim에서는 구현 x)
   airsim에는 각속도와 동일한 shape의 rollpitchyaw 방향 정보를 quaternion으로 변환하는 부분 존재

3. 왜곡 방식
   다음 3개의 링크는 최근 px4 드론 운행 중 log에 imu값에 대한 WARNING이나 CRITICAL이 나타났을 때 orientation, angular velocity 및 linear acceleration에 어떤식으로 왜곡이 들어가는지 보여줌
   https://review.px4.io/plot_app?log=2381c82f-a1bf-4edf-889d-74b7dbfc8f31

   https://review.px4.io/plot_app?log=67466a39-f663-4df1-97d2-5bd4131294b1

   https://review.px4.io/plot_app?log=a3f6c3b5-7525-4746-85ed-34bf44d13ed0
   결론은 현실에서의 왜곡은 flatten noise 보다 가우시안 노이즈에 더 근접한 모습을 보여줌

4. 룰베이스
   - 기존 코드 AirSimSimpleFlightEstimator.hpp

```
virtual simple_flight::Axis3r getLinearVelocity() const override
{
    return AirSimSimpleFlightCommon::toAxis3r(kinematics_->twist.linear);
}

virtual simple_flight::Axis4r getOrientation() const override
{
    return AirSimSimpleFlightCommon::toAxis4r(kinematics_->pose.orientation);
}
```

- 칼만 필터 구현 코드 AirSimSimpleFlightEstimator.hpp

```
virtual simple_flight::Axis3r getLinearVelocity() const override
{
    if (ekf_enabled_) {
        return getEkfLinearVelocity();
    }
    else {
        return getTrueLinearVelocity();
    }
}

virtual simple_flight::Axis4r getOrientation() const override
{
    if (ekf_enabled_) {
        return getEkfOrientation();
    }
    else {
        return getTrueOrientation();
    }
}
```

다음과 같은 방식으로 moveToPosition과 같은 api를 사용할 때 kinematics에서 값을 전달하는 것이 아닌 센서값을 전달하는 방식으로 기존의 문제점을 해결 가능 => 실제로 가능한지는 구현 후 test 필요

5. observation space로 imu sensor의 데이터와 gps sensor의 데이터를 사용.
   - 기존에는 position과 velocity 정보를 kinematics에서 갖고온 실제 pos, vel을 사용하였음.
   - gps_to_position 함수를 통해 gps로부터 나온 위도, 경도, 고도의 WGS84 좌표계를 airsim simpleflight의 x,y,z 좌표계로 변환하여 사용함.