课程编号：111111　　课程性质：专业必修

组合导航课程设计

GNSS-INS 组合导航程序设计

学院：测绘学院
专业：导航工程
教师：朱峰
班级：导航 3 班
姓名：叶通
学号：2020302142138

2023 年 2 月 23 日

Contents

Chapter 1　程序设计概述

§1.1　程序设计内容

1. 根据给定的一组高精度 IMU 数据、GNSS 位置数据，进行 GNSS-惯性导航松组合算法解算，导航结果与参考结果进行对比分析，验证自己的程序。

2. 分析初始方差 P0，过程噪声 Q，观测噪声 R 的数值大小调节对状态收敛性的影响。

3. 实现零速更新（ZUPT），非完整性约束（NHC），里程计更新（ODO），航向角更新（Heading）算法，分析观测更新间隔与观测方差设置的影响，并给出这些更新算法使用前后的效果对比与分析。

4. 在 GNSS 正常以及模拟 GNSS 失锁时，分别加入 NHC、ODO、NHC+ODO 观测更新，对比其加入前后的位置、速度、姿态结果变化。其中，模拟 5 个失锁时长：10 秒、30 秒、60 秒、90 秒、120 秒。

5. 撰写课程报告，提交后采取线上面试再次考核。

§1.2　程序设计要求

需提交作业文档（基本原理、实现细节和精度分析）及能够运行的源代码（Python, Matlab，C, C++ 语言选一种）

Chapter 2　算法与程序设计

§2.1　GNSS-INS 松组合算法简介

2.1.1　系统状态方程

GNSS/INS 松组合常采用误差状态卡尔曼滤波（间接卡尔曼滤波）进行组合导航解算，以解决系统的非线性问题。因此，卡尔曼滤波的状态向量包含导航状态误差和传感器误差，定义为

$$\delta \boldsymbol{x}(t)=\left[\begin{array}{ccccccc}(\delta \boldsymbol{r}^n)^T & (\delta \boldsymbol{v}^n)^T & \boldsymbol{\phi}^T & \boldsymbol{b}_g^T & \boldsymbol{b}_a^T & \boldsymbol{s}_g^T & \boldsymbol{s}_a^T\end{array}\right]^T \tag{2.1}$$

其中(2.1)表示将陀螺和加速度计的零偏及比例因子误差增广到卡尔曼滤波的状态量中进行估计。为得到系统状态方程，首先需要得到 $\sigma x(t)$ 的连续时间微分方程，写作如下形式：

$$\sigma \dot{x}=\boldsymbol{F}(t)\sigma x(t)+\boldsymbol{G}(t)\boldsymbol{\omega}(t) \tag{2.2}$$

其中将状态向量 $\sigma x(t)$ 向量求导即对向量的各分量分别对时间 t 求导，可以得到如下位置、速度和姿态误差微分方程

$$\delta \dot{\boldsymbol{r}}^n=-\boldsymbol{\omega}_{en}^n\times\delta \boldsymbol{r}^n+\delta \boldsymbol{\theta}\times \boldsymbol{v}^n+\delta \boldsymbol{v}^n \tag{2.3}$$

$$\delta \dot{\boldsymbol{v}}^n=\mathbf{C}_b^n\delta \boldsymbol{f}^b+\boldsymbol{f}^n\times\boldsymbol{\phi}-(2\boldsymbol{\omega}_{ie}^n+\boldsymbol{\omega}_{en}^n)\times\delta \boldsymbol{v}^n+\boldsymbol{v}^n\times(2\delta \boldsymbol{\omega}_{ie}^n+\delta \boldsymbol{\omega}_{en}^n)+\delta \boldsymbol{g}_p^n \tag{2.4}$$

$$\dot{\boldsymbol{\phi}}=-\boldsymbol{\omega}_{in}^n\times\boldsymbol{\phi}+\delta \boldsymbol{\omega}_{in}^n-\delta \boldsymbol{\omega}_{ib}^n \tag{2.5}$$

而剩下的陀螺和加速度计零偏及比例因子误差采用一阶高斯马尔科夫模型进行建模。

将上式带入式(2.2), 可得 $\boldsymbol{F}$ 矩阵：

$$\mathbf{F}=\left[\begin{array}{ccccccc}\mathbf{F}_{rr} & \mathbf{I}_{3\times3} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{F}_{vr} & \mathbf{F}_{vv} & \left[\left(\mathbf{C}_b^n\boldsymbol{f}^b\right)\times\right] & \mathbf{0} & \mathbf{C}_b^n & \mathbf{0} & \mathbf{C}_b^n\,\mathrm{diag}\left(\boldsymbol{f}^b\right)\\ \mathbf{F}_{\phi r} & \mathbf{F}_{\phi v} & -\left(\omega_{in}^n\times\right) & -\mathbf{C}_b^n & \mathbf{0} & -\mathbf{C}_b^n\,\mathrm{diag}\left(\boldsymbol{\omega}_{ib}^b\right) & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \frac{-1}{T_{gb}}\mathbf{I}_{3\times3} & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \frac{-1}{T_{ab}}\mathbf{I}_{3\times3} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \frac{-1}{T_{gs}}\mathbf{I}_{3\times3} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \frac{-1}{T_{as}}\mathbf{I}_{3\times3}\end{array}\right] \tag{2.6}$$

其中各分量的具体表达式可参见组合导航讲义，在此不再详述。

状态方程的系统噪声向量 $\boldsymbol{\omega}(t)$ 可以写为如下形式：

$$\boldsymbol{w}=\left[\begin{array}{cccccc}\boldsymbol{w}_v^T & \boldsymbol{w}_\phi^T & \boldsymbol{w}_{gb}^T & \boldsymbol{w}_{ab}^T & \boldsymbol{w}_{gs}^T & \boldsymbol{w}_{as}^T\end{array}\right]^T \tag{2.7}$$

其中 $\boldsymbol{\omega}$ 表示一个三维的白噪声向量。

对(2.2)进行离散化处理，并根据解微分方程求出离散系统的状态转移矩阵。在时间间隔较短，$\boldsymbol{F}$ 在积分区间变化不太剧烈时，可以把求解状态转移矩阵的矩阵积分作以下的简化：

$$\boldsymbol{\Phi}_{k/k-1}=\exp\left\{\mathbf{F}\left(t_{k-1}\right)\Delta t\right\}\approx\mathbf{I}+\mathbf{F}\left(t_{k-1}\right)\Delta t \tag{2.8}$$

同样，当 $\boldsymbol{G}(t)$ 在较短积分区间内变化不太剧烈的时候，离散化后系统噪声的方差阵 $\boldsymbol{Q}$ 可以写为

$$\mathbf{Q}_k \approx \frac{1}{2}\left[\boldsymbol{\Phi}_{k/k-1}\mathbf{G}\left(t_{k-1}\right)\mathbf{q}\left(t_{k-1}\right)\mathbf{G}^{\mathrm{T}}\left(t_{k-1}\right)\boldsymbol{\Phi}_{k/k-1}^{\mathrm{T}} + \mathbf{G}\left(t_k\right)\mathbf{q}\left(t_k\right)\mathbf{G}^{\mathrm{T}}\left(t_k\right)\right]\Delta t \tag{2.9}$$

其中连续时间系统噪声方差阵 $\boldsymbol{q}(t)$ 一般情况下是常值矩阵，由 IMU 的传感器误差参数模型决定。$\boldsymbol{q}(t)$ 与 $\boldsymbol{G}$ 的具体表达式参加组合导航讲义。

这样即构造出了等效离散系统满足离散卡尔曼滤波的基本方程。

2.1.2　GNSS 位置观测方程

按理来说，GNSS 直接提供了位置观测结果，则观测方程的系数矩阵应该为在位置分量上的一个单位阵和在其他分量均为零矩阵的一个大矩阵，观测值依然表示为 INS 推算出来的位置减去 GNSS 观测值的位置。但是由于 GNSS 定位解算给出的是天线相位中心（或其它参考点）的位置坐标，INS 机械编排给出的是 IMU 测量中心的导航结果，二者在物理上不重合，因此在数据融合解算时需进行杆臂效应改正。

IMU 中心位置与 GNSS 位置的关系可写为如下形式（考虑各状态量误差）：

$$\hat{\boldsymbol{r}}_G^n = \hat{\boldsymbol{r}}_I^n + \mathbf{D}_R^{-1}\hat{\mathbf{C}}_b^n\boldsymbol{l}^b \tag{2.10}$$

其中 $\hat{\boldsymbol{r}}_I^n$ 为 INS 推算的 IMU 中心位置（经纬度表示），$\hat{\boldsymbol{r}}_G^n$ 为惯导推算出来的 GNSS 观测值的天线相位中心位置（经纬度表示），$\boldsymbol{l}^b$ 为 INS 指向 GNSS 的向量在 b 系下的投影结果（视为无误差），$\boldsymbol{D}_R^{-1}$ 为将导航系的北东地分量转为经纬度的对角矩阵（视为无误差）。设 GNSS 直接观测的位置为 $\tilde{\boldsymbol{r}}_G^n$, 则观测向量（单位为 m）可以表示为：

$$\sigma\boldsymbol{z}_r = \boldsymbol{D}_R(\hat{\boldsymbol{r}}_G^n - \tilde{\boldsymbol{r}}_G^n) \tag{2.11}$$

将(2.10)做扰动分析，写成矩阵形式为：

$$\sigma\boldsymbol{z}_r = \boldsymbol{H}_r\sigma\boldsymbol{x} + \boldsymbol{n}_r \tag{2.12}$$

其中

$$\boldsymbol{H}_r = \begin{bmatrix}\boldsymbol{I}_3 & \mathbf{0}_3 & (\boldsymbol{C}_b^n\boldsymbol{l}^b\times) & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3\end{bmatrix} \tag{2.13}$$

实际解算时，观测矩阵 $\boldsymbol{H}_r$ 中的 $\boldsymbol{C}_b^n$ 用最新时刻的惯导姿态进行赋值。

2.1.3　GNSS 速度观测方程

与上述位置观测方程同理，考虑到杆臂以及姿态误差，速度误差和陀螺角速度误差的影响，并把速度观测误差向量表示为 INS 推算的速度与 GNSS 解算的速度之差：

$$\begin{aligned}\delta\boldsymbol{z}_v &= \hat{\boldsymbol{v}}_G^n - \tilde{\boldsymbol{v}}_G^n \\ &= \delta\boldsymbol{v}^n - \left(\boldsymbol{\omega}_{in}^n\times\right)\left(\mathbf{C}_b^n\boldsymbol{l}^b\times\right)\boldsymbol{\phi} - \left[\mathbf{C}_b^n\left(\boldsymbol{l}^b\times\boldsymbol{\omega}_{ib}^b\right)\right]\times\boldsymbol{\phi} - \mathbf{C}_b^n\left(\boldsymbol{l}^b\times\right)\delta\boldsymbol{\omega}_{ib}^b + \boldsymbol{n}_v\end{aligned} \tag{2.14}$$

写成矩阵形式有：

$$\sigma\boldsymbol{z}_v = \boldsymbol{H}_v\sigma\boldsymbol{x} + \boldsymbol{n}_v \tag{2.15}$$

其中

$$\mathbf{H}_v = \begin{bmatrix}\mathbf{0}_3 & \mathbf{I}_3 & \mathbf{H}_{v3} & -\left(\mathbf{C}_b^n\boldsymbol{l}^b\times\right) & \mathbf{0}_3 & \mathbf{H}_{v6} & \mathbf{0}_3\end{bmatrix} \tag{2.16}$$

$$\begin{cases}\mathbf{H}_{v3} = -\left(\boldsymbol{\omega}_{in}^n\times\right)\left(\mathbf{C}_b^n\boldsymbol{l}^b\times\right) - \left[\mathbf{C}_b^n\left(\boldsymbol{l}^b\times\boldsymbol{\omega}_{ib}^b\right)\times\right] \\ \mathbf{H}_{v6} = -\mathbf{C}_b^n\left(\boldsymbol{l}^b\times\right)\mathrm{diag}\left(\boldsymbol{\omega}_{ib}^b\right)\end{cases} \tag{2.17}$$

§2.2　编程实现过程

2.2.1　结构体以及类的定义

由于程序冗长，工程浩大，细节繁琐，涉及变量和状态较多，故先将必要的结构体和类进行定义。在”InertialNavigation.h” 中进行如下结构体的定义

```
struct ANGLE
{
	double yaw;
	double pitch;
	double roll;

	ANGLE(double y=0.0,double p=0.0,double r=0.0)
	{
		yaw = y;
		pitch = p;
		roll = r;
	}
};
```

其中 IMUError 用来存放估计出来的零偏和比例因子，而不能直接从外界获取，故没有写构造函数。

```
struct IMUError
{
	Vector gyrbias;//标准单位rad/s
	Vector accbias;//标准单位m/s^2
	Vector gyrscale;//标准单位 1
	Vector accscale;//标准单位 1

	IMUError():gyrbias(3),accbias(3),gyrscale(3),accscale(3)
	{

	}
};
```

而 IMUNoise 用来存放 IMU 的噪声参数，可以直接从外界给出的值进行赋值，故重写构造函数并将外界传入的单位转换为标准单位。

```
struct IMUNoise
{
	Vector ARW;//角度随机游走
	Vector VRW;//速度随机游走
	Vector gb_std;//陀螺仪零偏标准差
	Vector ab_std;//加速度计零偏标准差
```

```
        Vector gs_std;//陀螺仪比例因子标准差
        Vector as_std;//加速度计比例因子标准差
        double corr_t;//相关时间


        IMUNoise(Vector arw , Vector vrw , Vector gb, Vector ab, Vector gs ,
 Vector as , double t):ARW(3),VRW(3),gb_std(3),ab_std(3),gs_std(3),as_std(3)
        {
                ARW = arw * D2R * (1.0 / 60.0);//输入参数单位为[deg/s/sqrt(h)]
                VRW = vrw * (1.0 / 60.0);//输入参数单位为[m/s/sqrt(h)]
                gb_std = gb * D2R * (1.0 / 3600.0);//输入参数单位为[deg/h]
                ab_std = ab * 1.0E-5;//输入参数单位为[mGal]
                gs_std = gs * 1.0E-6;//输入参数单位为[ppm]
                as_std = as * 1.0E-6;//输入参数单位为[ppm]
                corr_t = t;
        }


};
```

IMUDATA 用来存放读入的 IMU 数据（增量形式）。

```
struct IMUDATA
{
        double time;

        Vector AngInc;
        Vector VelInc;

        IMUDATA() :AngInc(3), VelInc(3)
        {

        }
};
```

GNSSDATA 用来存放 GNSS 数据以及可用性信息

```
struct GNSSDATA
{
        double time;

        Vector Blh;
        Vector Vel;
```

```
7
8         Vector Blh_std;
9         Vector Vel_std;
10
11        bool DataAvailable;
12
13        GNSSDATA():Blh(3),Vel(3),Blh_std(3),Vel_std(3)
14        {
15                time = 0.0;
16                DataAvailable = false;
17        }
18  };
```

另外，出现的 Vector 和 Matrix 都是自己实现的矢量和矩阵类，功能包括基本的矩阵和向量运算。

定义 Body 类，继承自 IMUBOX 类 (包含上一时刻与当前时刻的位置速度姿态和姿态矩阵等结构体成员, 已在前期课程中实现了输入 IMUDATA 完成纯惯导机械编排)，包含前一时刻和当前 IMUDATA,GNSS 数据，IMU 误差结构体，IMU 噪声结构体，杆臂向量，时间标记，状态协方差 Cov，系统噪声矩阵 q, 待估参数 x。

```
1   class Body :public IMUBOX
2   {
3
4   public:
5
6         Body(NavState inistate, NavState inistate_std, IMUNoise imun, Vector lb);
7
8         void AddIMUData(double* OneLineIMU);
9         void AddGNSSData(double* OneLineGNSS);
10
11        void SetTimestamp(double time);
12        bool IsGNSSDataAvailable();
13
14        void work();
15
16        void BodyShow();
17        void BodyOutputToFile(fstream& fout);
18        void CovOutputToFile(fstream& fout);
19
20  private:
21
22        IMUDATA IMUData_pre, IMUData_cur;//前一时刻和现在的IMU数据
```

```
        GNSSDATA GNSSData;//GNSS数据

        double timestamp;//每次要进行内插的时间记号（通常为初始时刻和有GNSS数据整秒）

        IMUError ImuError;
        IMUNoise ImuNoise;

        Vector antlever;//杆臂（由b->GNSS,在b系下测量）

        Matrix Cov;//状态协方差（21*21）
        Matrix q;//系统噪声矩阵（18*18）
        Matrix x;//待估的参数（21*1）

        void ImuInterpolate(IMUDATA& IMUData_pre, IMUDATA& IMUData_cur, double
 midtime, IMUDATA& IMUData_mid);

        void ImuPropagation(IMUDATA& IMUData_pre, IMUDATA& IMUData_cur);
        void GNSSUpdating(IMUDATA& IMUData_mid);

        void Feedback();
        void Compensate();

};
```

并写构造函数，用传入初始的位置速度姿态信息调用 IMUBOX 父类的构造函数对父类的成员（当前与上一时刻位置速度姿态）赋值，用当前的方差信息对 Cov 矩阵赋值，用 IMUNoise 的噪声信息对 q 阵赋值。

2.2.2　IMU 状态传播实现

首先调用父类机械编排函数实现此时的状态更新，然后使用内部成员上一时刻位置速度姿态实现 F 矩阵和 G 矩阵各分块矩阵的构造，并最终按(2.6)重组 F 矩阵和 G 矩阵（已在矩阵类中重载运算符实现了矩阵拼接功能）。

```
        F = (Frr, I_33, Zero_33, Zero_33, Zero_33, Zero_33)
& (Fvr, Fvv, Cn_bfb.AntiSym(), Zero_33, Cn_b_pre, Zero_33, Cn_b_pre * fb.diag())
& (Fphir, Fphiv, -wn_in.AntiSym(), -Cn_b_pre, Zero_33, -Cn_b_pre * wb_ib.diag(), Zero_33)
& (Zero_33, Zero_33, Zero_33, _1_T_I_33, Zero_33, Zero_33, Zero_33)
& (Zero_33, Zero_33, Zero_33, Zero_33, _1_T_I_33, Zero_33, Zero_33)
& (Zero_33, Zero_33, Zero_33, Zero_33, Zero_33, _1_T_I_33, Zero_33)
& (Zero_33, Zero_33, Zero_33, Zero_33, Zero_33, Zero_33, _1_T_I_33);


        G = (Zero_33, Zero_33, Zero_33, Zero_33, Zero_33, Zero_33)
& (Cn_b_pre, Zero_33, Zero_33, Zero_33, Zero_33, Zero_33)
& (Zero_33, Cn_b_pre, Zero_33, Zero_33, Zero_33, Zero_33)
```

```
& (Zero_33, Zero_33, I_33, Zero_33, Zero_33, Zero_33)
& (Zero_33, Zero_33, Zero_33, I_33, Zero_33, Zero_33)
& (Zero_33, Zero_33, Zero_33, Zero_33, I_33, Zero_33)
& (Zero_33, Zero_33, Zero_33, Zero_33, Zero_33, I_33);
```

构造完成后，按(2.8)(2.9)给出的简化公式计算状态转移 Phi 矩阵和离散系统噪声方差阵 Q。再按卡尔曼滤波的一步预测公式，实现状态方差的传播

```
    Q = G * q * G.T() * delta_t;
    Q = (Phi * Q * Phi.T() + Q) * 0.5;

    //由于状态总为0矩阵，故此步可以省略
    //x = Phi * x;
    Cov = Phi * Cov * Phi.T() + Q;
```

2.2.3　GNSS 测量更新实现

传入 IMUDATA 类对象 (需要用到当前的加速度计输出和角速度输出信息)，使用已读入的内部成员 GNSSDATA 中的数据，实现 GNSS 观测方程的构造和状态量 x 的计算更新。

构造观测向量：

```
    //位置观测向量表示为 INS 推算的位置与 GNSS 位置观测之差
    Vector Zr(3);
    Zr = Dr * (IMU_to_GNSSBlh - GNSSData.Blh );

    //速度观测向量可表示为 INS 推算的速度与 GNSS 解算的速度之差
    Vector Zv(3);
    Zv = Vel - GNSSData.Vel;
```

计算中间变量并完成 H 矩阵构造。并将位置观测方程和速度观测方程的两个 H 矩阵上下拼接。

```
    Matrix Hv3(3, 3);
    Hv3 = -1.0 * wn_in.AntiSym() * Cn_blb.AntiSym() - Cn_b_lb_wbib.AntiSymMatrix();
    Matrix Hv6(3, 3);
    Hv6 = -1.0 * Cn_b * antlever.AntiSym() * wb_ib.diag();

    H = (I_33, Zero_33, Cn_blb.AntiSym(), Zero_33, Zero_33, Zero_33, Zero_33)
        & (Zero_33, I_33, Hv3, -1.0 * (Cn_blb).AntiSym(), Zero_33, Hv6, Zero_33);
```

再使用 GNSSData 中给出的观测标准差信息，组成测量噪声的 R 矩阵

```
    //观测噪声矩阵
    Matrix pos_var(3, 3);
    Matrix vel_var(3, 3);

    pos_var = GNSSData.Blh_std.diag() * GNSSData.Blh_std.diag();
    vel_var = GNSSData.Vel_std.diag() * GNSSData.Vel_std.diag();

    //Matrix R(6, 6);
    Matrix R = (pos_var, Zero_33) & (Zero_33, vel_var);
```

之后，套用卡尔曼滤波给出的公式，进行状态的更新和状态方差阵更新。更新完毕后，将 GNSSData 的可用性设置为 false。

```
        //卡尔曼滤波测量更新
        Matrix K = Cov * H.T() * (H * Cov * H.T() + R).inv();
        //x = x + K * (Z - H * x);
        x = K * Z;
        Cov = (I_2121 - K * H) * Cov * (I_2121 - K * H).T() + K * R * K.T();

        //设置GNSS数据状态为不可用
        GNSSData.DataAvailable = false;
```

2.2.4　INS-GNSS 松组合导航整体实现

首先定义 Body 类对象，并用已知的初始条件传入构造函数将其初始化。之后设置时间标签为开始时刻，并通过循环读掉开始时刻之前的 IMU 和 GNSS 数据。

之后，在循环中，每添加一行 IMU 数据，进行调用 body.work() 函数。进入 work 函数内部后，先检查当前 IMU 数据的时间是否小于时间标记（设置为 GNSSData 的时间），小于则只进行 IMU 的传播，若大于，则说明需要进行 GNSS 更新。具体更新步骤为，先内插出一个在 timestamp 的 IMU 数据，然后用此 IMU 数据实现前半 IMU 传播，得到最新的当前时刻状态。然后使用这个内插 IMU 数据作为参数带入 GNSS 更新函数中，完成状态更新，更新完后立刻将状态进行然后再次进行一个 IMU 传播，得到当前读入 IMU 数据时刻的状态。

```
void Body::work()
{
        //如果当前时间小于时间标记，则继续传播IMU状态
        if (IMUData_cur.time < timestamp)
        {
                ImuPropagation(IMUData_pre, IMUData_cur);
        }
        //当前时间大于时间标记，则首先进行对时间标记时刻的状态内插，再判断是否进行GNSS更新
        //一般起始时刻不更新，后面每个GNSS整秒都进行更新
        else if (IMUData_cur.time > timestamp)
        {
                IMUDATA IMUData_mid;

                //内插出中间时刻IMU增量数据
                ImuInterpolate(IMUData_pre, IMUData_cur, timestamp, IMUData_mid);

                //前半状态传播
                ImuPropagation(IMUData_pre, IMUData_mid);

                if ( GNSSData.DataAvailable == true )
                {
                        //测量更新
                        GNSSUpdating(IMUData_mid);
                }
```

```
25
26                  Feedback();
27
28                  ImuPropagation(IMUData_mid, IMUData_cur);
29
30          }
31
32          return;
33
34  }
```

最后在 work 调用后，判断 GNSS 数据是否可用，不可用则读取新的 GNSS 数据并将可用性设置为 true，并自动将时间标记设置为这个 GNSS 数据的时间。这样一来，IMU 将持续进行状态传播直到当 IMU 数据的时间再次超过此时间标记再进行 GNSS 更新。循环此步骤即实现了松组合导航算法。

以下是在 main 函数的部分实现代码。

```
1           while (!fileloader_INS.isEof())
2           {
3                   fileloader_INS.load(OneLineImu);
4
5                   //跳过起始时刻之前的数据
6                   if (OneLineImu[0] < STARTTIME)
7                   {
8                           body.AddIMUData(OneLineImu.data());
9                           continue;
10                  }
11                  //添加IMU数据
12                  body.AddIMUData(OneLineImu.data());
13
14                  //进行松组合算法传播误差状态和测量更新工作
15                  body.work();
16
17                  //若GNSS更新后读取一组新GNSS数据
18                  if (body.IsGNSSDataAvailable() == false)
19                  {
20                          fileloader_GNSS.load(OneLineGNSS);
21                          body.AddGNSSData(OneLineGNSS.data());
22                  }
23          }
```

§2.3　结果展示

2.3.1　纯松组合结果展示

经过上述代码编译运行后，将数据输出到 txt 文件中，再导入 MATLAB 中进行成图，得到组合导航解算数据与参考数据中载体的位置速度姿态随时间的变化图并比较二者差异。

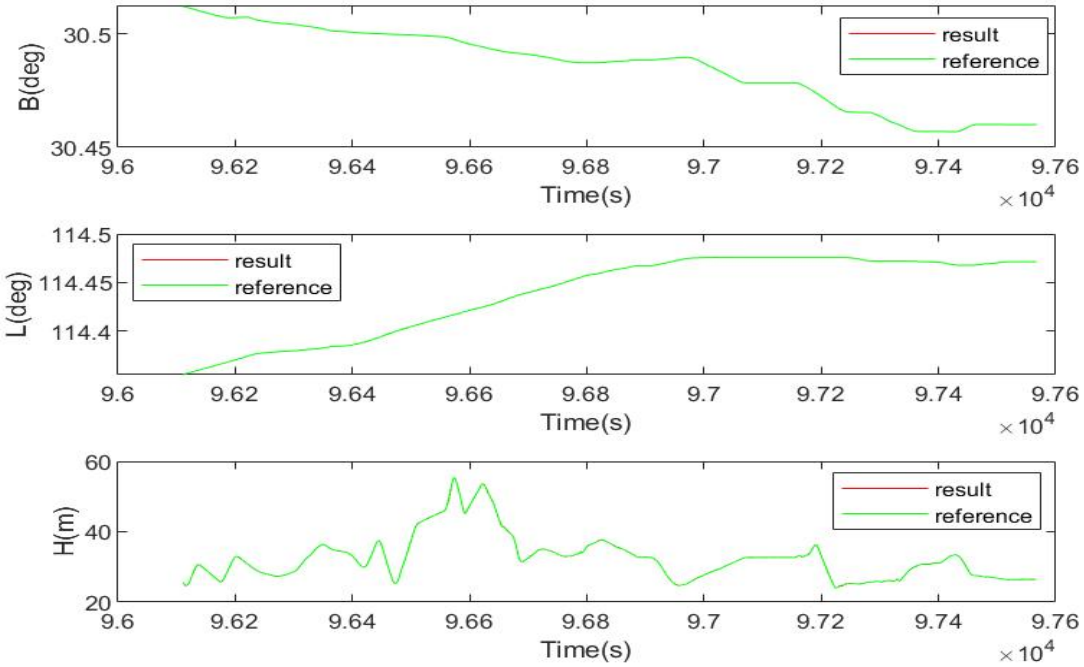

Figure 2.1: 载体位置解算结果

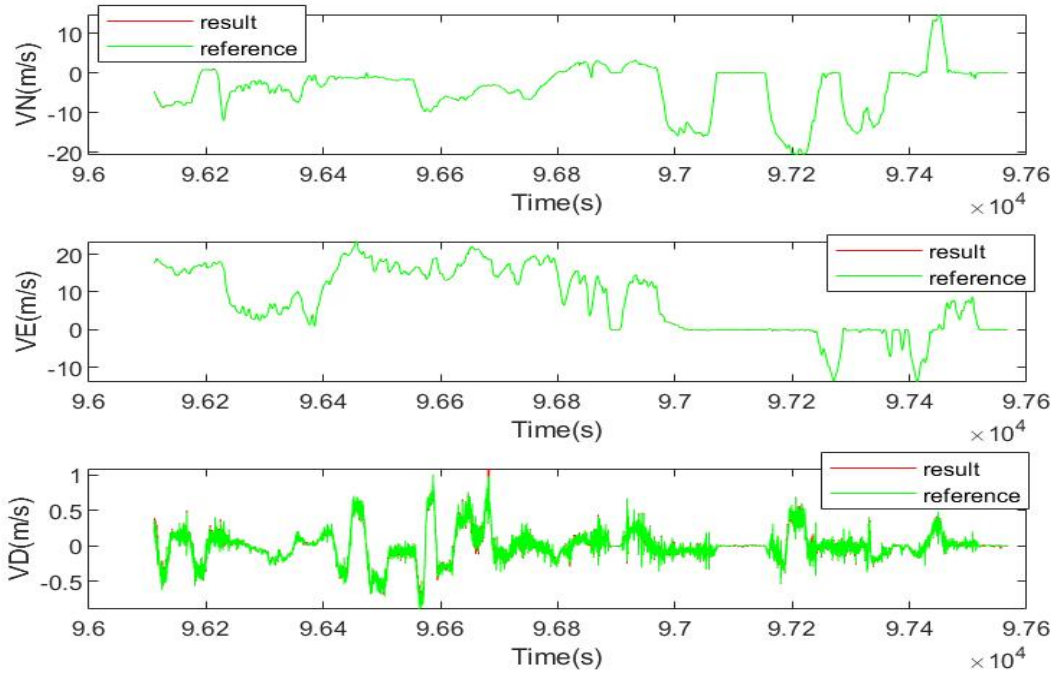

Figure 2.2: 载体速度解算结果

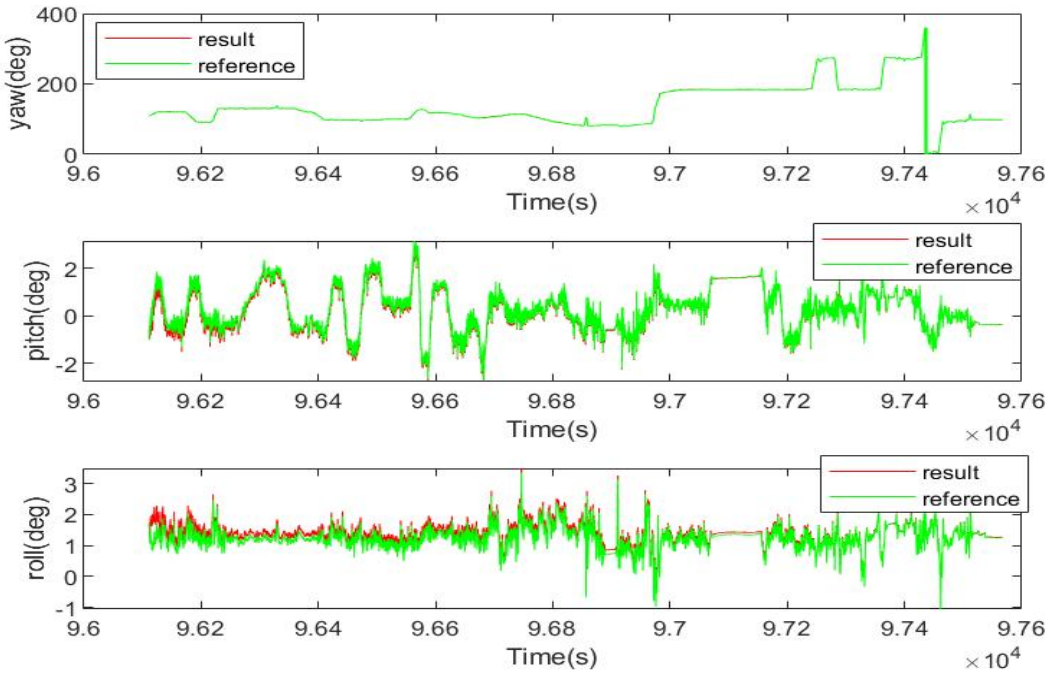

Figure 2.3: 载体姿态解算结果

可以看到解算结果与参考结果的曲线几乎完美的重合在一起，说明解算成果较为正确，松组合算法程序成功运行。

2.3.2　初始方差和噪声阵对收敛结果分析

将正常按课程要求中给出的初始方差设置 P 阵，用(2.9)式求取 Q 阵，按 GNSS 观测数据给出的观测噪声构造 R 阵进行松组合解算，将所得的各状态的方差随时间变化作图可得：

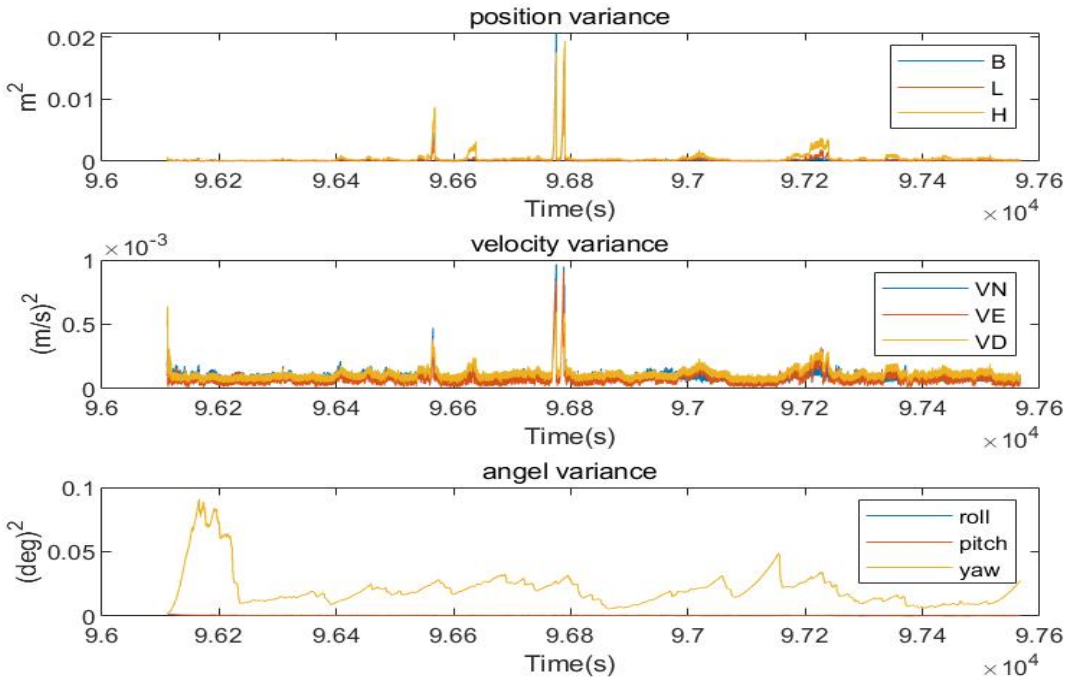

Figure 2.4: 载体位置速度姿态方差解算结果

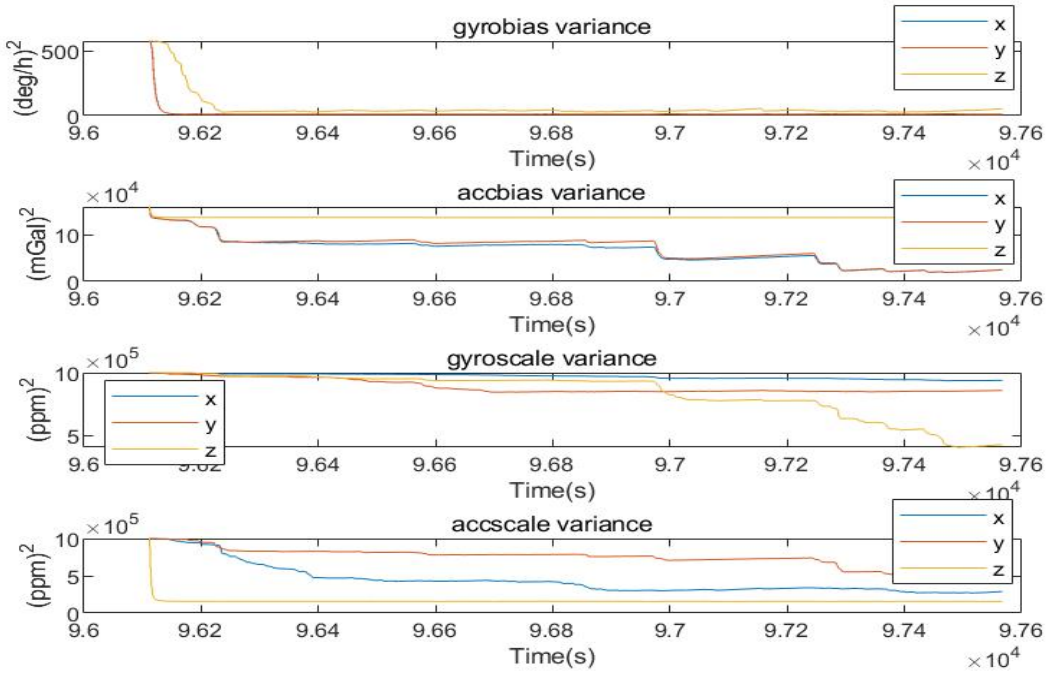

Figure 2.5: 载体零偏项比例因子项方差解算结果

可以看到位置速度姿态的方差大都一开始就稳定在一个比较小的量级，而陀螺和加速度的零偏

与比例因子项则存在慢慢收敛的过程。

对图像进行放大后可以看到，方差曲线存在随时间呈锯齿状的上下波动，这是因为在没有 GNSS 数据的时刻，卡尔曼滤波不断进行状态的一步预测，推演状态的传播并进行方差的传递和累积，而到了有 GNSS 数据时刻进行测量更新，更新后状态的方差也一并下降。图中位置与速度的方差在 96800s 前后有一个十分显著的峰值，也是因为这几个历元的 GNSS 数据缺失导致的方差传递所致。

接下来设置初始方差阵为原先的两倍

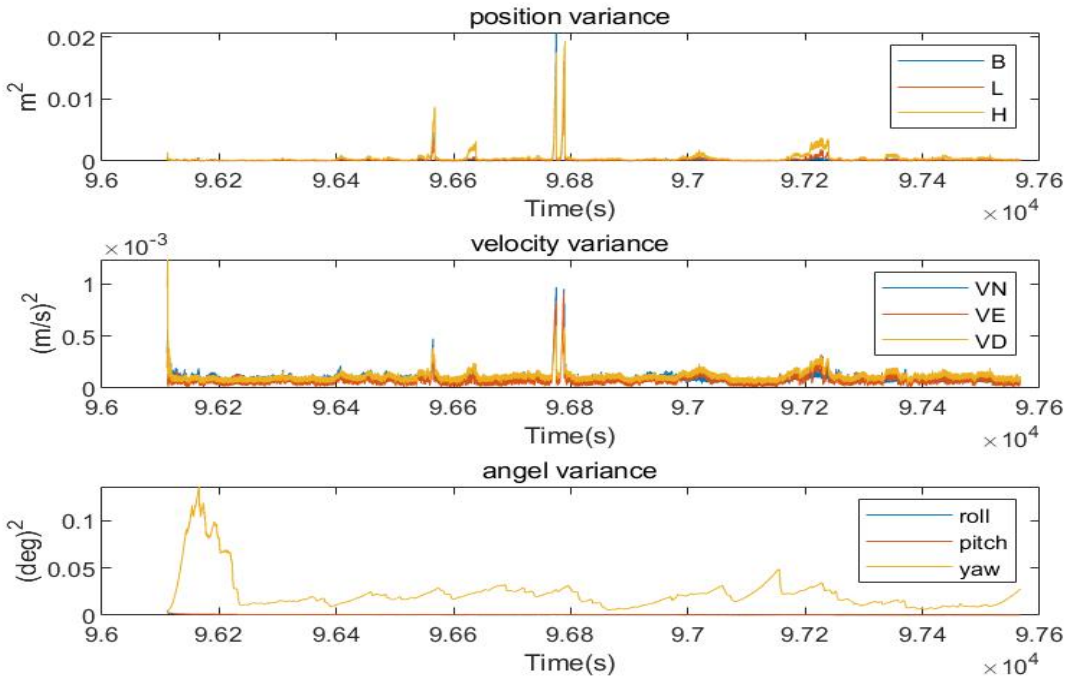

Figure 2.6: 两倍 P 阵载体位置速度姿态方差解算结果

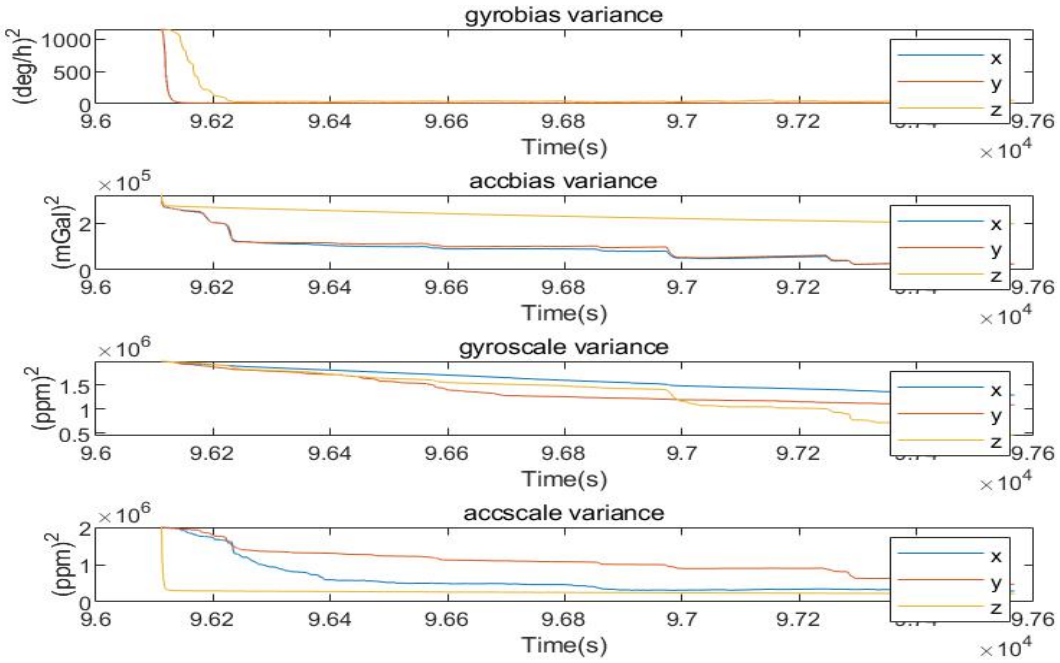

Figure 2.7: 两倍 P 阵载体零偏项比例因子项方差解算结果

可以看到图像与原先差异不大，仅仅是初值有差异，整个曲线形状几乎不变，说明初值的设置对卡尔曼滤波的影响不太大，只要卡尔曼滤波方程正确，方差都能随着测量更新快速下降收敛。

设置系统噪声阵 Q 为原先的两倍

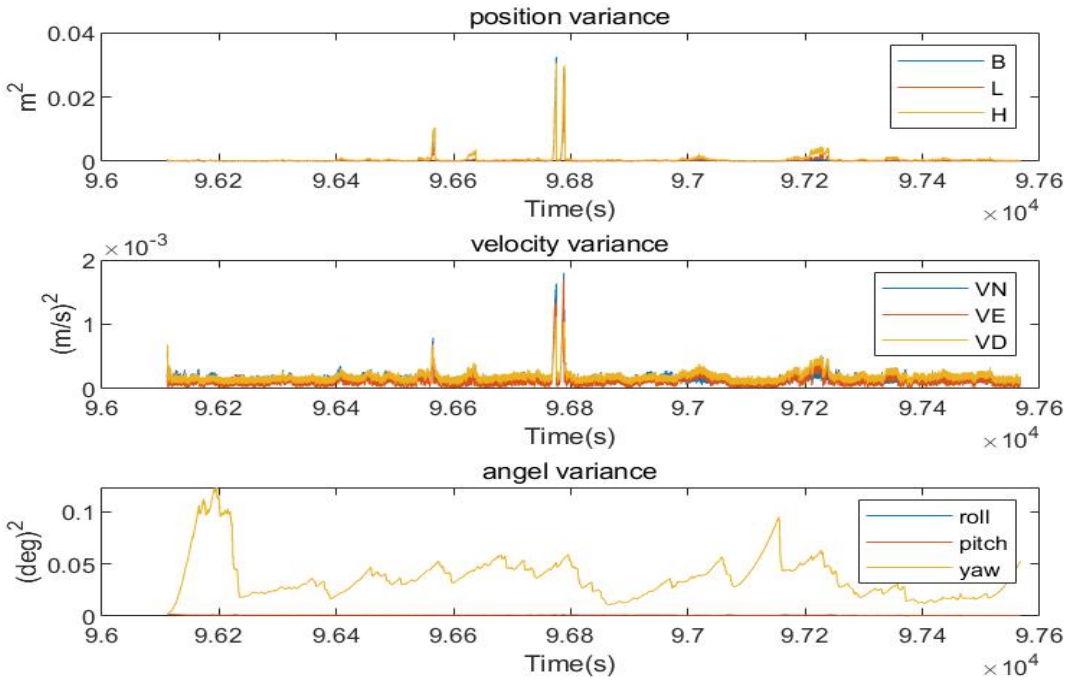

Figure 2.8: 两倍 Q 阵载体位置速度姿态方差解算结果

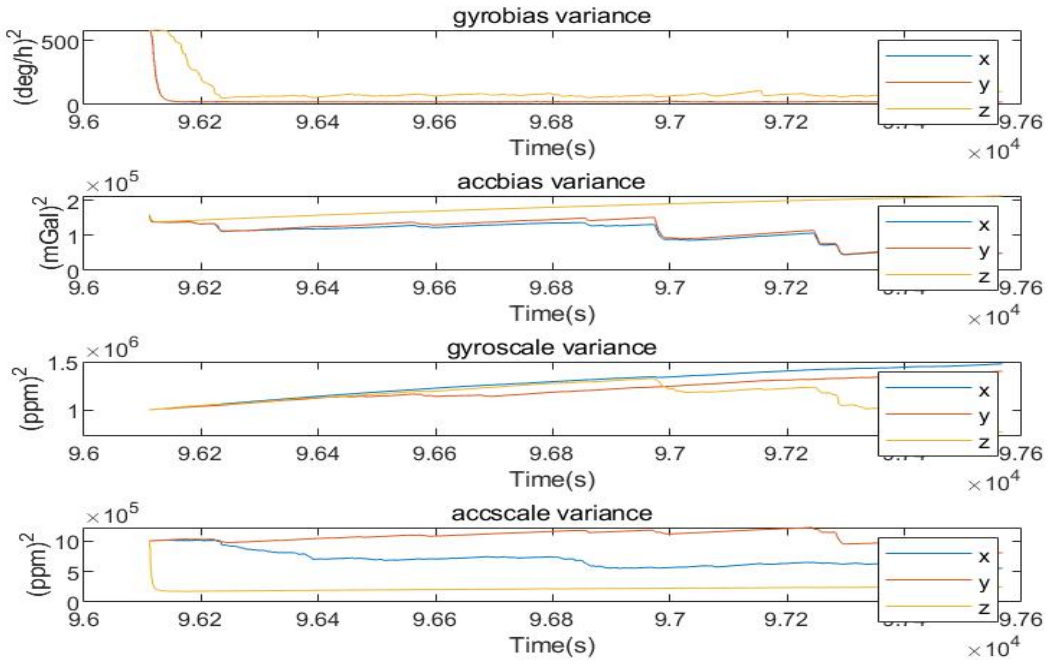

Figure 2.9: 两倍 Q 阵载体零偏项比例因子项方差解算结果

可以看到，在位置速度姿态的收敛情况变化不大，但是在加速度计陀螺仪误差项时，有部分方

差曲线会持续增大，无法收敛。说明了 Q 阵过大会导致状态的方差矩阵变大，即使测量更新也无法将其拉回，最终导致状态发散。

设置观测噪声阵 R 为原先的两倍

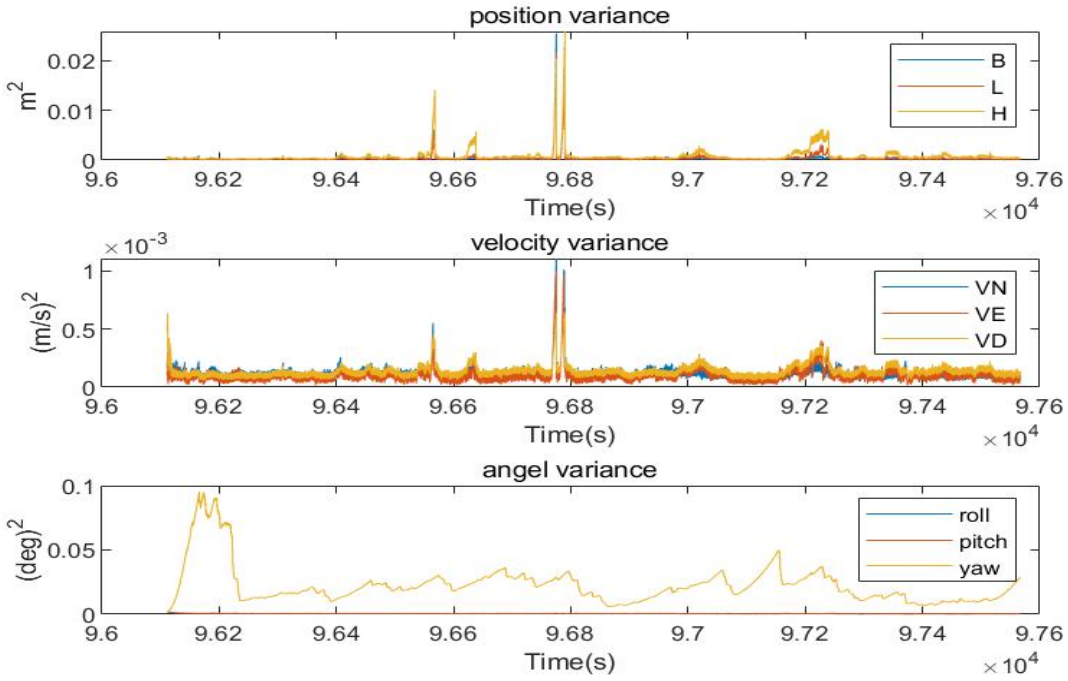

Figure 2.10: 两倍 R 阵载体位置速度姿态方差解算结果

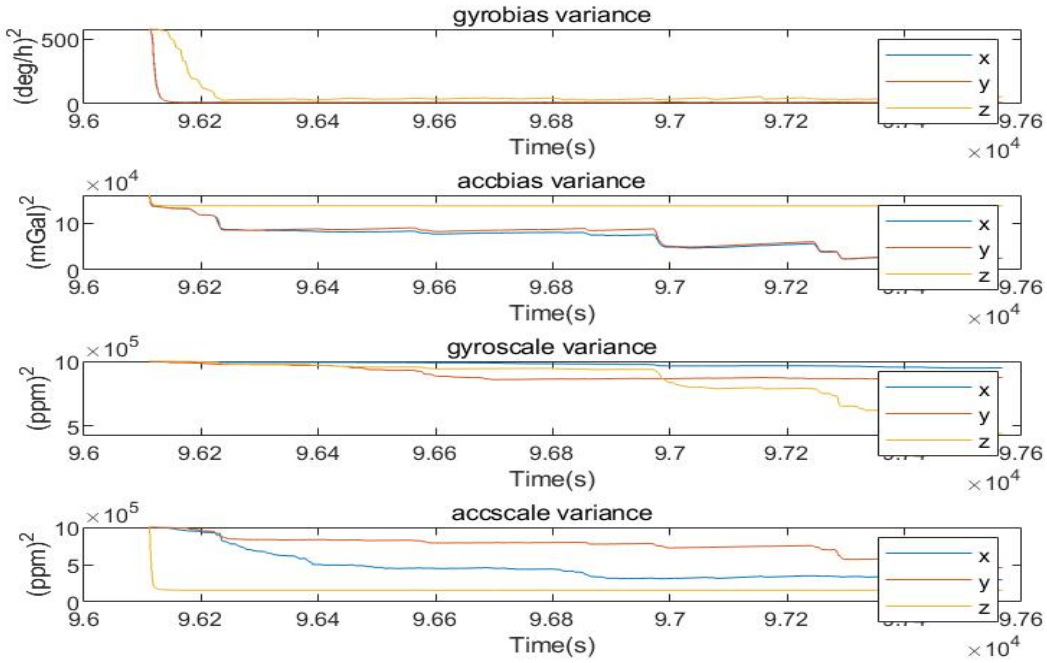

Figure 2.11: 两倍 R 阵载体零偏项比例因子项方差解算结果

发现与原来相比的变化也不太明显，可能是由于 GNSS 数据给出的信息比较精确，方差比较小，即使两倍也相比于惯导状态的方差仍为小量，测量更新依然会很好地收敛。合理猜测继续放大观测噪声阵 R 将会导致收敛速度变慢。

Chapter 3　其他辅助算法简介与实现

§3.1　辅助算法简介

3.1.1　零速更新算法（ZUPT）

零速更新算法是指当运动载体静止的时候，在 N 系下的三轴速度都为 0. 利用这一信息，当判断载体为 0 速时，则可以假想有某一神秘仪器对载体的速度进行了观测并给出了 0 速的观测值，建立观测方程带入卡尔曼滤波中即可对载体状态进行更新，用来抑制惯导在 GNSS 失锁时刻的状态飘移。

设惯导推算出的速度为 $\hat{v}$, 观测到的 0 速为 $\tilde{v}$, 有如下关系式：

$$\hat{\boldsymbol{v}} = \boldsymbol{v} + \boldsymbol{\sigma v} \tag{3.1}$$

$$\tilde{\boldsymbol{v}} = \boldsymbol{v} - \boldsymbol{n} \tag{3.2}$$

$$\hat{\boldsymbol{v}} - \tilde{\boldsymbol{v}} = \sigma \boldsymbol{v} + \boldsymbol{n} \tag{3.3}$$

其中 n 表示观测噪声，代表了对这个虚拟 0 值的相信程度。令观测向量为 $Z = (\hat{v} - \tilde{v}) = \hat{v}$, 有

$$\boldsymbol{Z} = \boldsymbol{H}\delta\boldsymbol{x} + \boldsymbol{n} \tag{3.4}$$

其中

$$\boldsymbol{H} = \begin{bmatrix} \mathbf{0} & \boldsymbol{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \tag{3.5}$$

3.1.2　非完整性约束算法（NHC）与里程计更新算法（ODO）

非完整性约束只正常车辆行驶过程中，不发生飘移，打滑，车轮不离开地面。这样一来，载体的侧向速度以及垂向速度理论上应一直保持为 0. 里程计则可以一直提供车体的前向速度，可以作为卡尔曼观测更新的数据使用。

设里程计坐标系的三轴朝向与 IMU 保持一致，即二者仅有杆臂上的差异,$\mathbf{C}_b^v$ 为单位阵。使用 INS 机械编排推导的车轮速度为：

$$\begin{aligned} &\hat{\boldsymbol{v}}_{wheel}^v = \mathbf{C}_b^v \hat{\mathbf{C}}_n^b \hat{\boldsymbol{v}}_{\mathrm{IMU}}^n + \mathbf{C}_b^v \left(\hat{\boldsymbol{\omega}}_{nb}^b \times\right) \boldsymbol{I}_{\mathrm{wheel}}^b \\ &\approx v_{\mathrm{wheel}}^v + C_b^v C_n^b \delta v_{IMU}^n - C_b^v C_n^b \left(v_{IMU}^n \times\right) - C_b^v \left(l_{\mathrm{wheel}}^b \times\right) \delta\omega_{ib}^b \end{aligned} \tag{3.6}$$

车体坐标系的速度为

$$\tilde{\boldsymbol{v}}_{wheel}^v = \boldsymbol{v}_{wheel}^v - \boldsymbol{n} \tag{3.7}$$

其中车体的观测速度向量 $\boldsymbol{v}_{wheel}$ 为,v 有 ODO 里程计提供，第二三项为虚拟的 0 值

$$\tilde{\boldsymbol{v}}_{wheel}^v = \begin{bmatrix} v & 0 & 0 \end{bmatrix} \tag{3.8}$$

同理可构造观测方程

$$\boldsymbol{Z} = \hat{\boldsymbol{v}}_{wheel}^v - \tilde{\boldsymbol{v}}_{wheel}^v = \boldsymbol{H}\delta\boldsymbol{x} + \boldsymbol{n} \tag{3.9}$$

其中 $\boldsymbol{H}$ 为：

$$\mathbf{H} = \begin{bmatrix} \mathbf{0}_3 & \mathbf{C}_b^v\mathbf{C}_n^b & -\mathbf{C}_b^v\mathbf{C}_n^b\left(\boldsymbol{v}_{LMU}^n\times\right) & -\mathbf{C}_b^v\left(l_{\text{whel}}^b\times\right) & \mathbf{0}_3 & \mathbf{H}_{vw,6} & \mathbf{0}_3 \end{bmatrix} \tag{3.10}$$

其中,

$$\mathbf{H}_6 = -\mathbf{C}_b^v\left(l_{\text{whel}}^b\times\right)\text{diag}\left(\omega_{ib}^b\right) \tag{3.11}$$

需要进行单独 NHC 与 ODO 的测量更新时，可以直接用上述方程，在观测值向量 $\boldsymbol{Z}$ 和观测矩阵 $\boldsymbol{H}$ 前加入两行和一行的单位阵将整个方程降维即可使用。

§3.2　结果展示

使用在一个窗口的加速度计输出与静止时的理论常值作差，小于阈值则判断为 0 速，进行 ZUPT 更新。对惯导数据进行解算（其中在 97000s 开始有一段 120s 的 GNSS 的失锁）进行解算，作图如下。

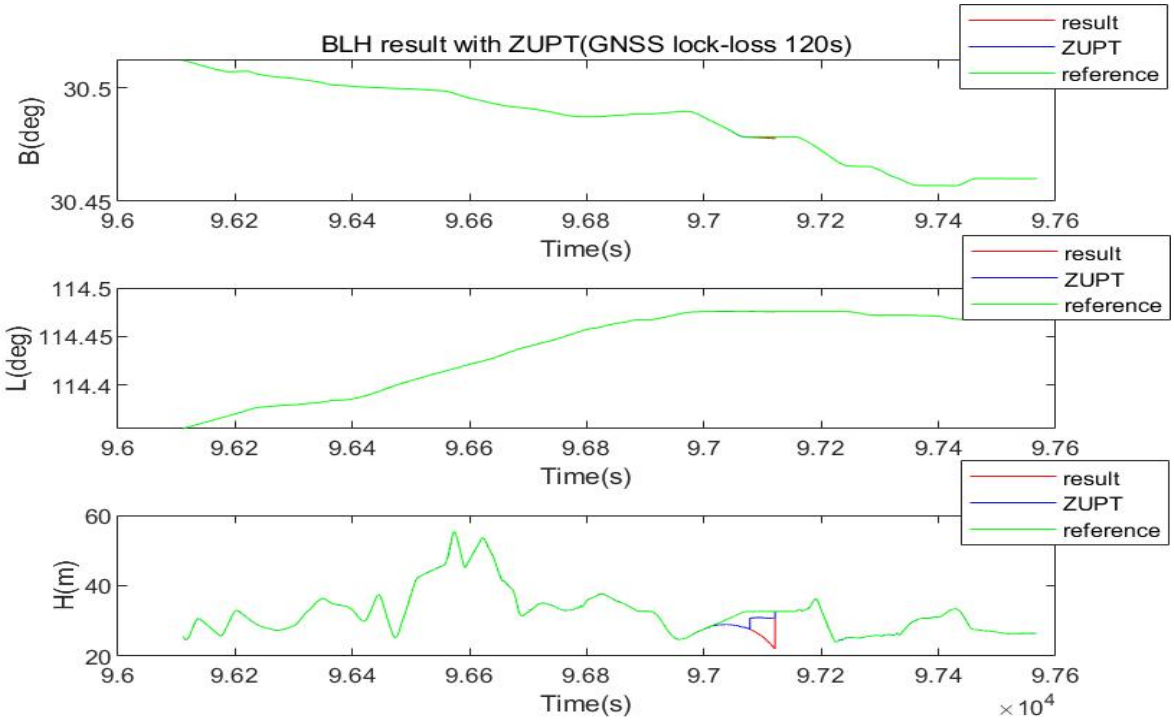

Figure 3.1: 载体位置解算结果 (零速更新)

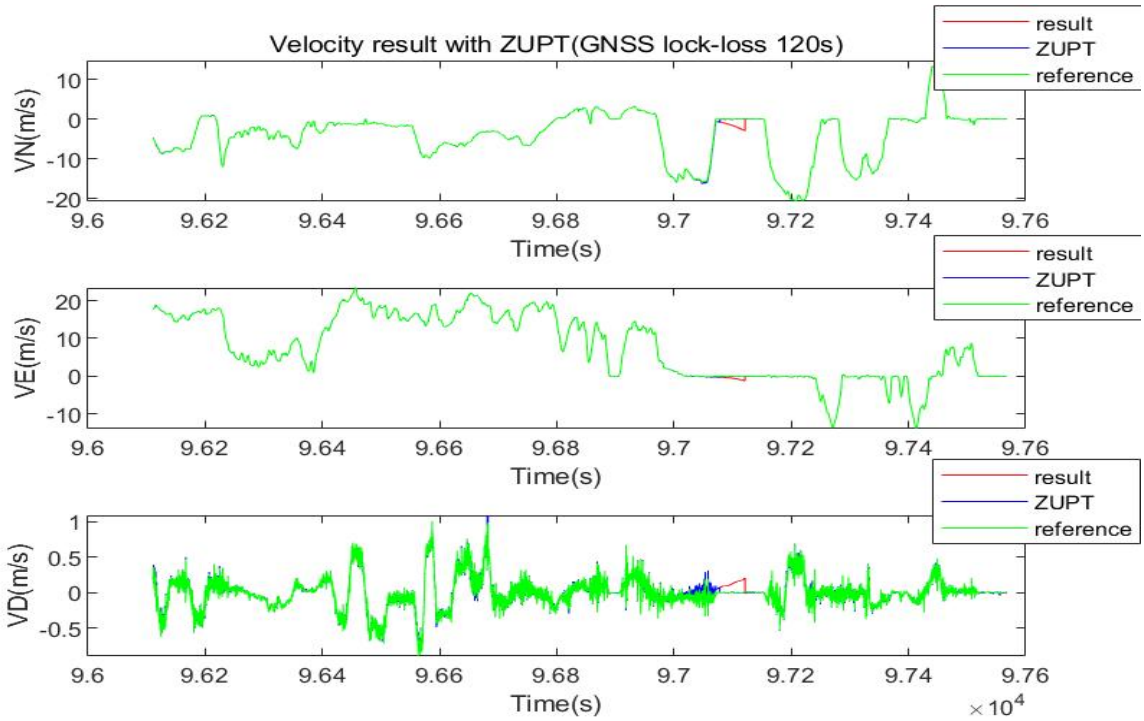

Figure 3.2: 载体速度解算结果 (零速更新)

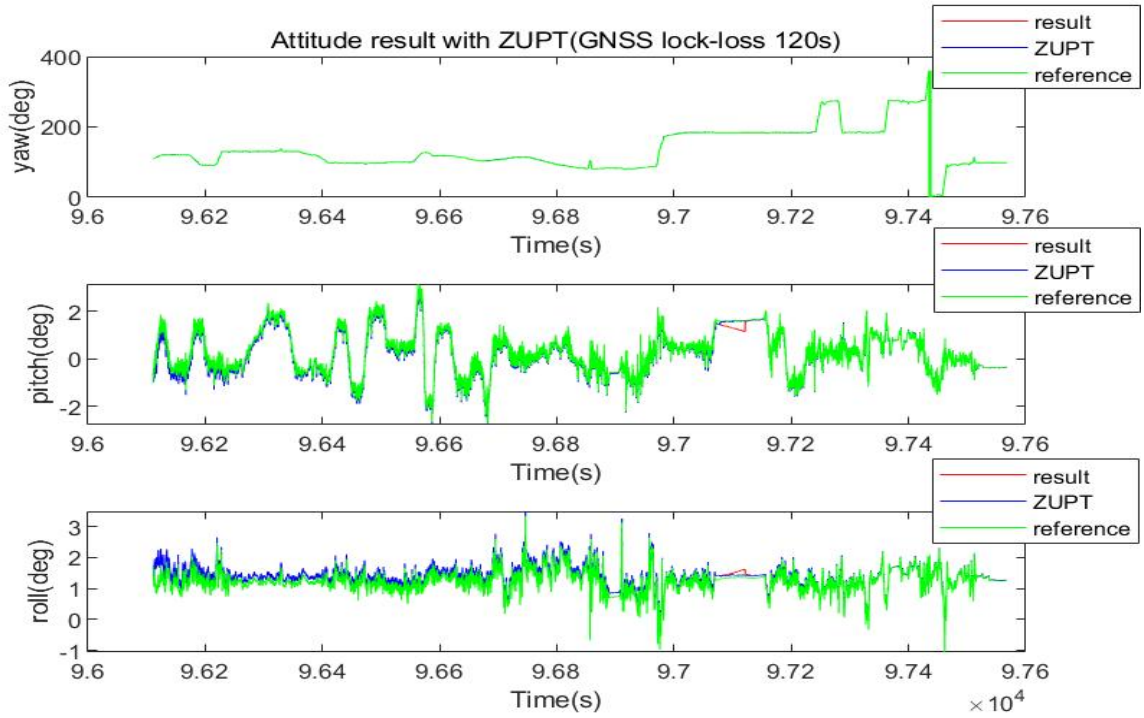

Figure 3.3: 载体姿态解算结果 (零速更新)

可以看出，在其他时段，开启零速更新算法后的解算与无零速更新算法时的结果几乎一致，而在 GNSS 失锁时段，在使用零速更新算法后能非常有效地抑制位置速度姿态的飘移。

读取里程计数据，可以发现里程计的数据时刻与惯导是对齐的，于是可以对每一个惯导状态的传播后进行 NHC 与 ODO 的更新。

在选取失锁时段 96500s 开始失锁 10s, 之后分别为 96600s（30s）,96700s(60s),96800s(90s),97000s(120s) 的 GNSS 数据进行组合导航解算，并作图如下。

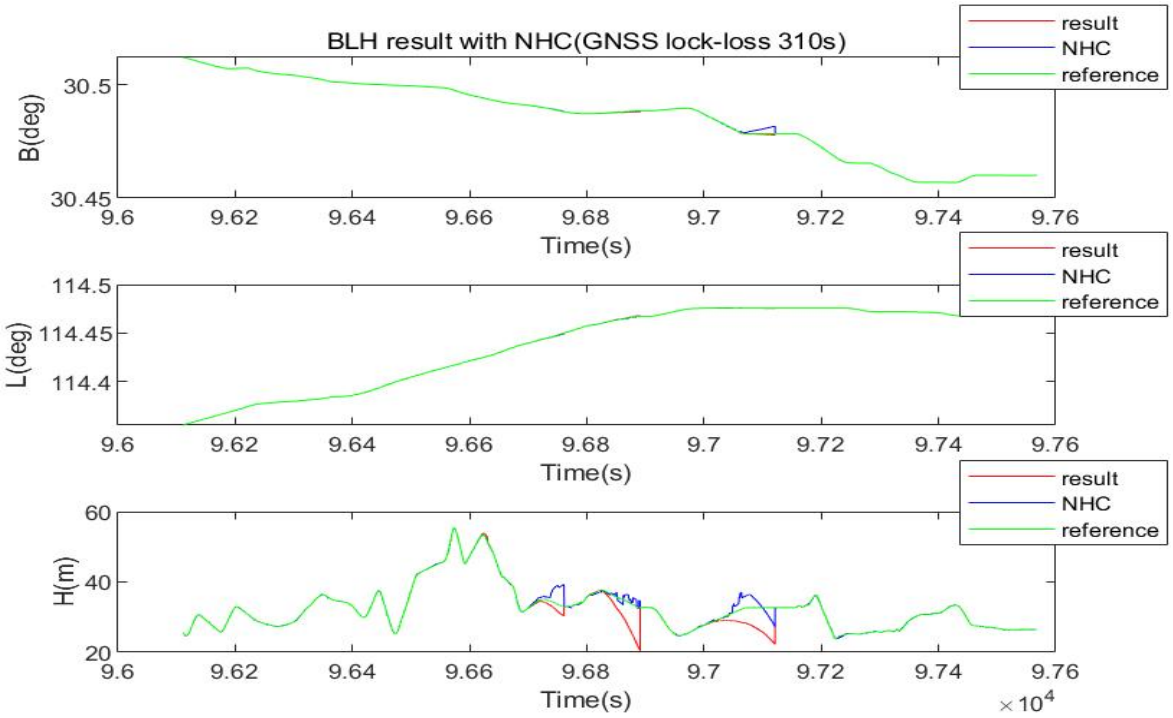

Figure 3.4: 载体位置解算结果 (NHC 更新)

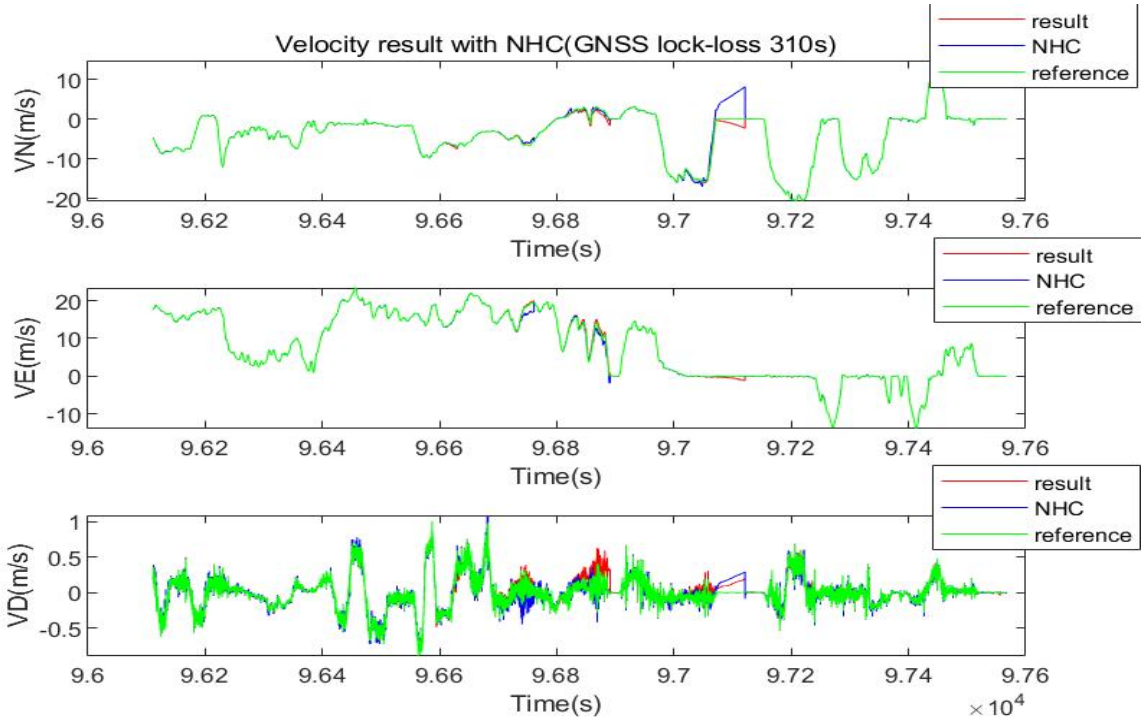

Figure 3.5: 载体速度解算结果 (NHC 更新)

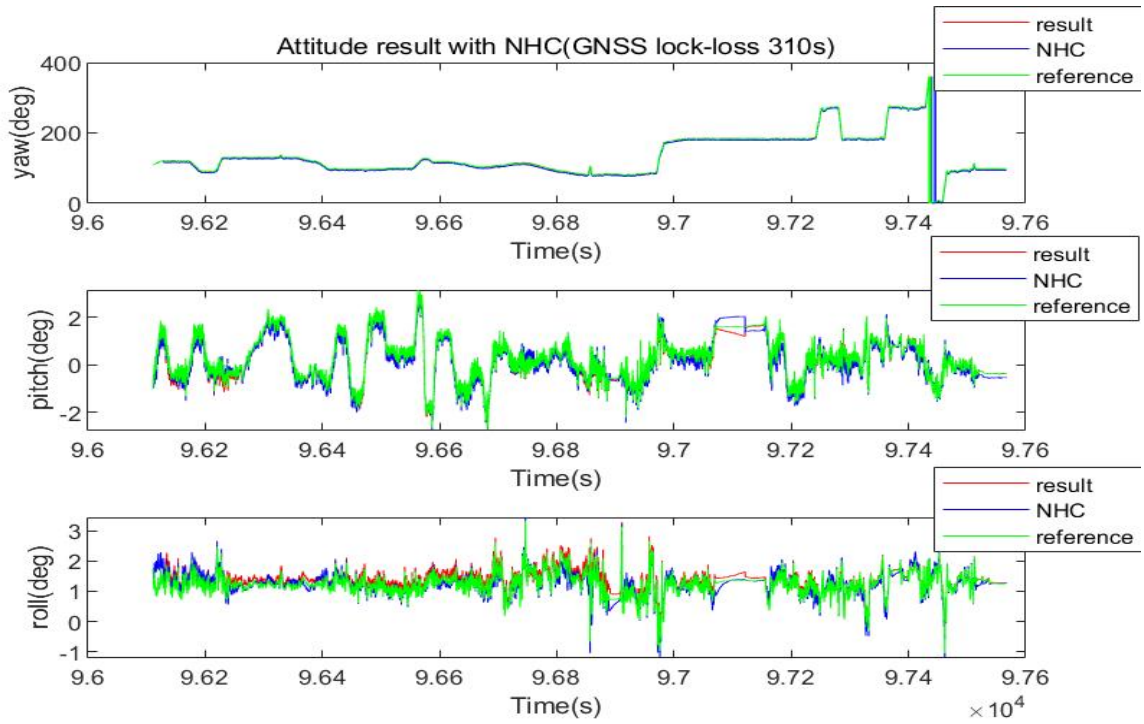

Figure 3.6: 载体姿态解算结果 (NHC 更新)

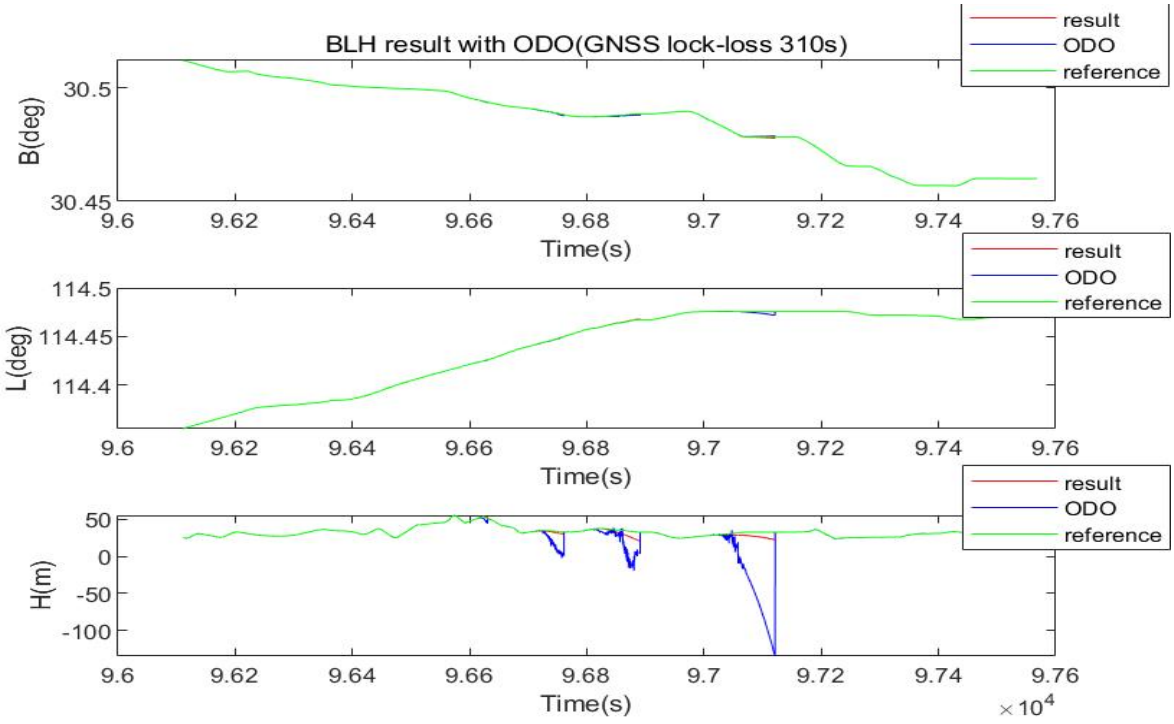

Figure 3.7: 载体位置解算结果 (ODO 更新)

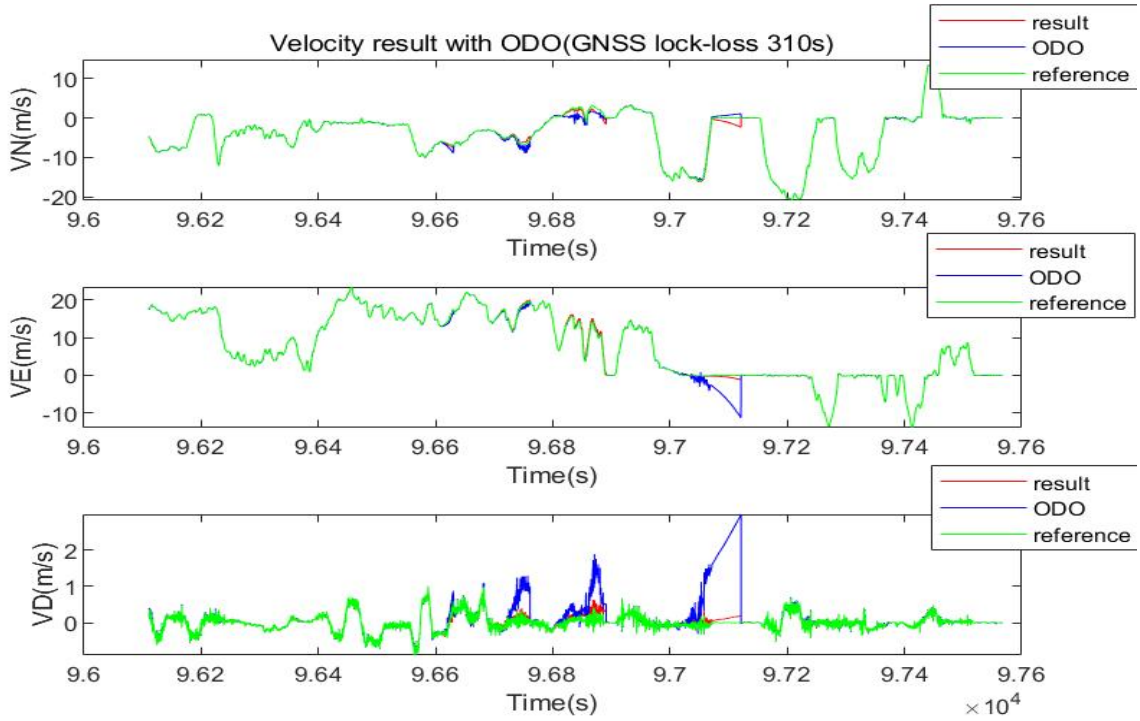

Figure 3.8: 载体速度解算结果 (ODO 更新)

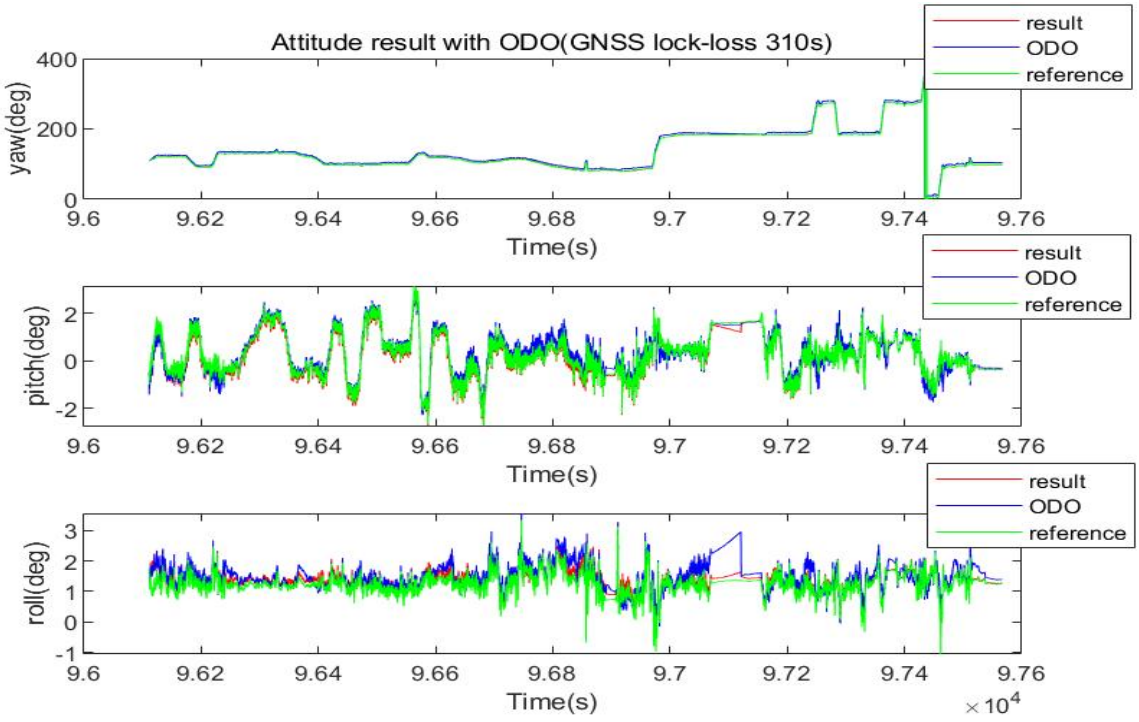

Figure 3.9: 载体姿态解算结果 (ODO 更新)

可以发现 NHC 和 ODO 的效果都不好，加入 NHC 和 ODO 后，在失锁时段甚至出现了反向飘移或比不加入时飘移更大的现象。

在经过非常艰辛的反复检查代码和公式，不断调试更新间隔和观测方差 R 阵，效果都非常不太理想。最终发现选取的失锁时段载体速度不高或接近零速。在更换失锁时段为 96300s(120s),96500(90s), 96600(60s),96700(30s),96800s(10s) 后作图如下：

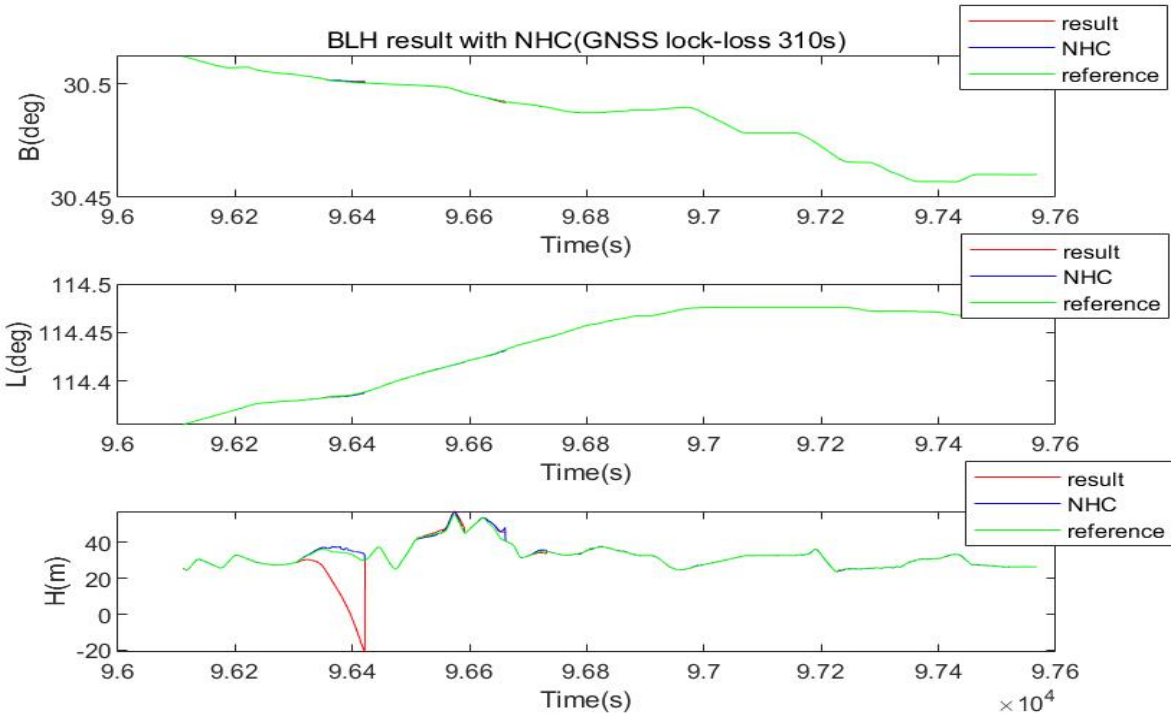

Figure 3.10: 载体位置解算结果 (NHC 更新)

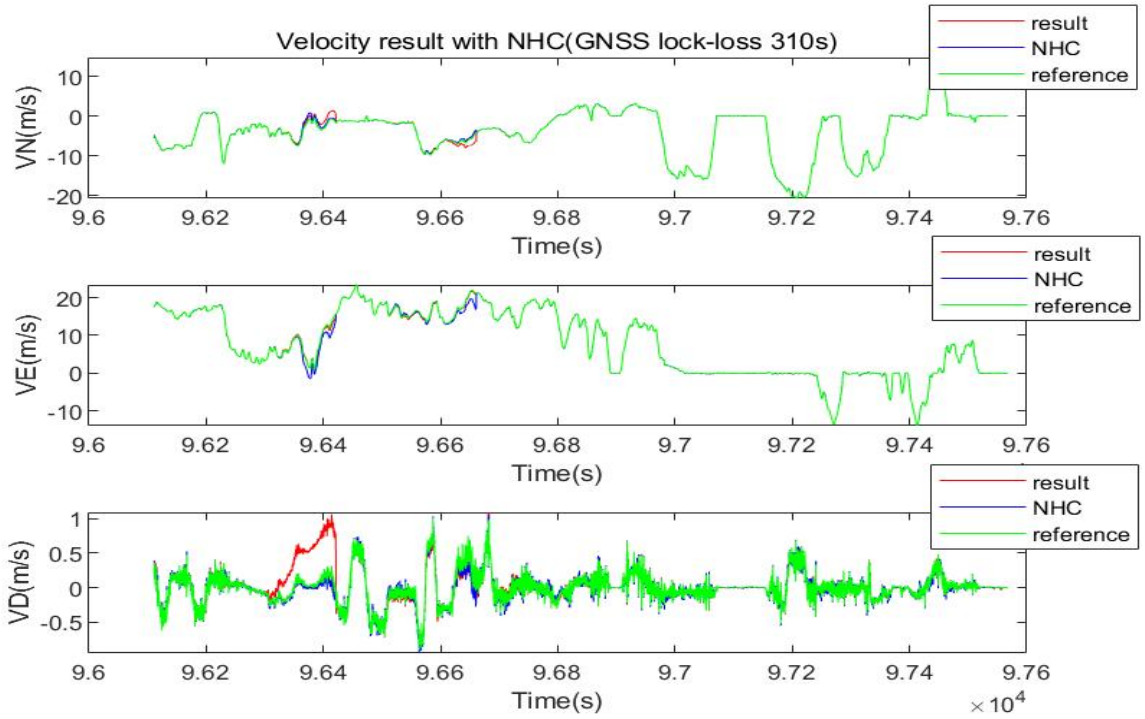

Figure 3.11: 载体速度解算结果 (NHC 更新)

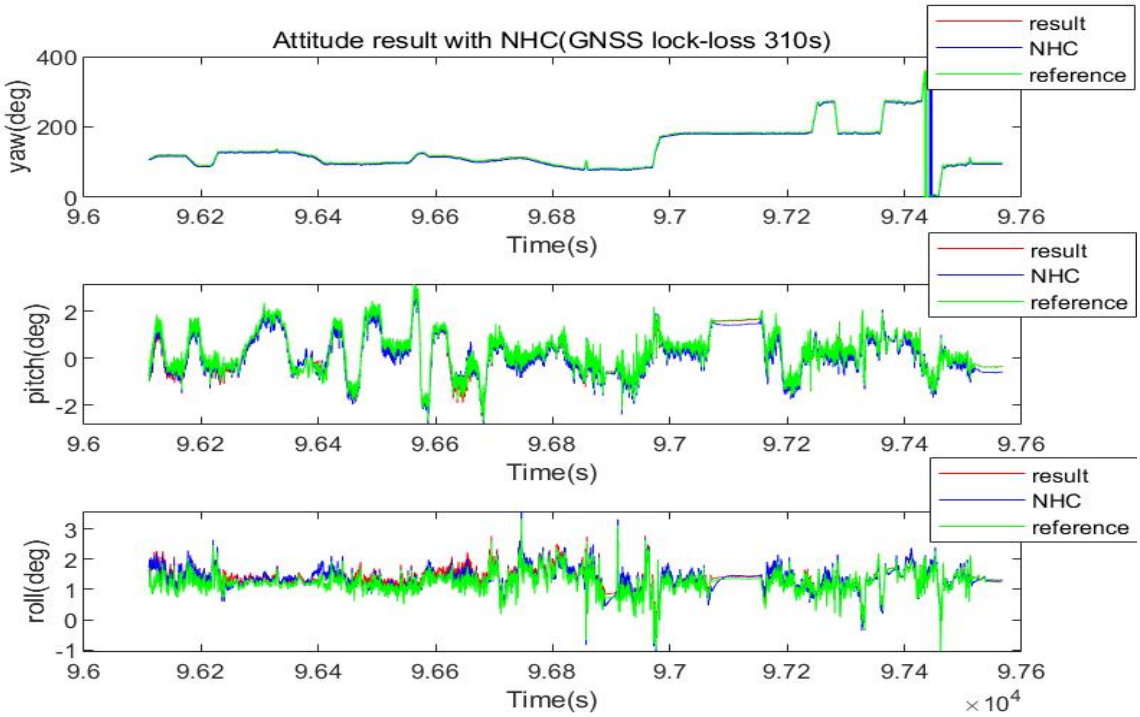

Figure 3.12: 载体姿态解算结果 (NHC 更新)

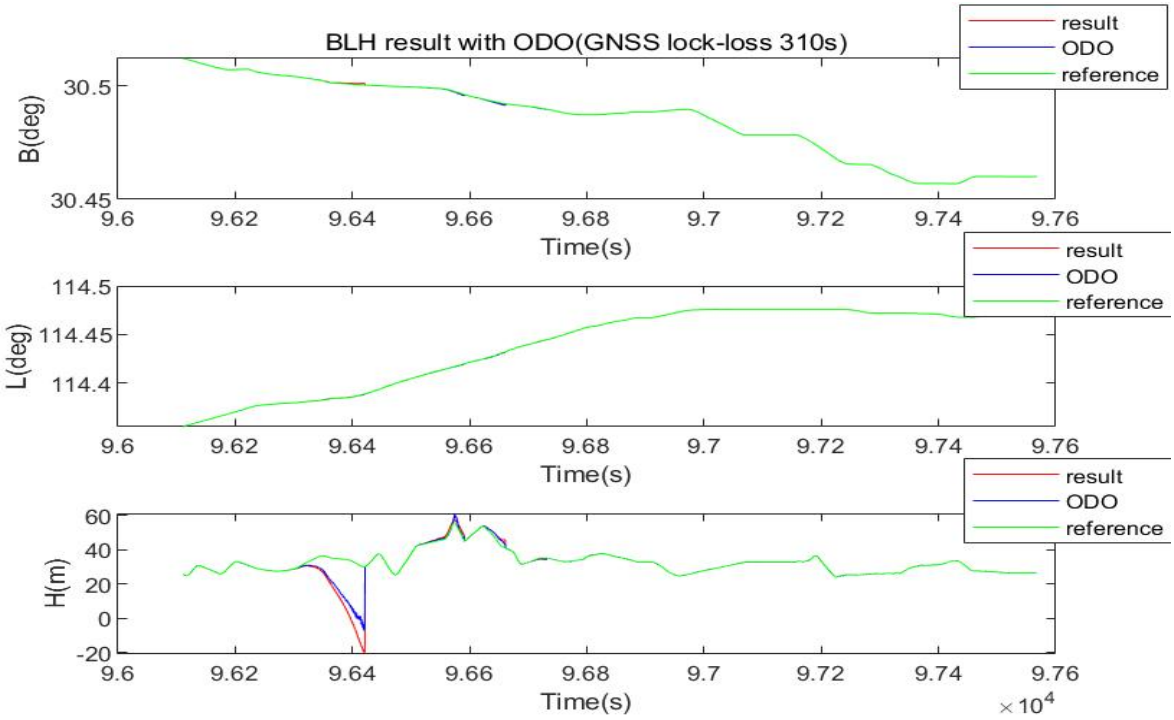

Figure 3.13: 载体位置解算结果 (ODO 更新)

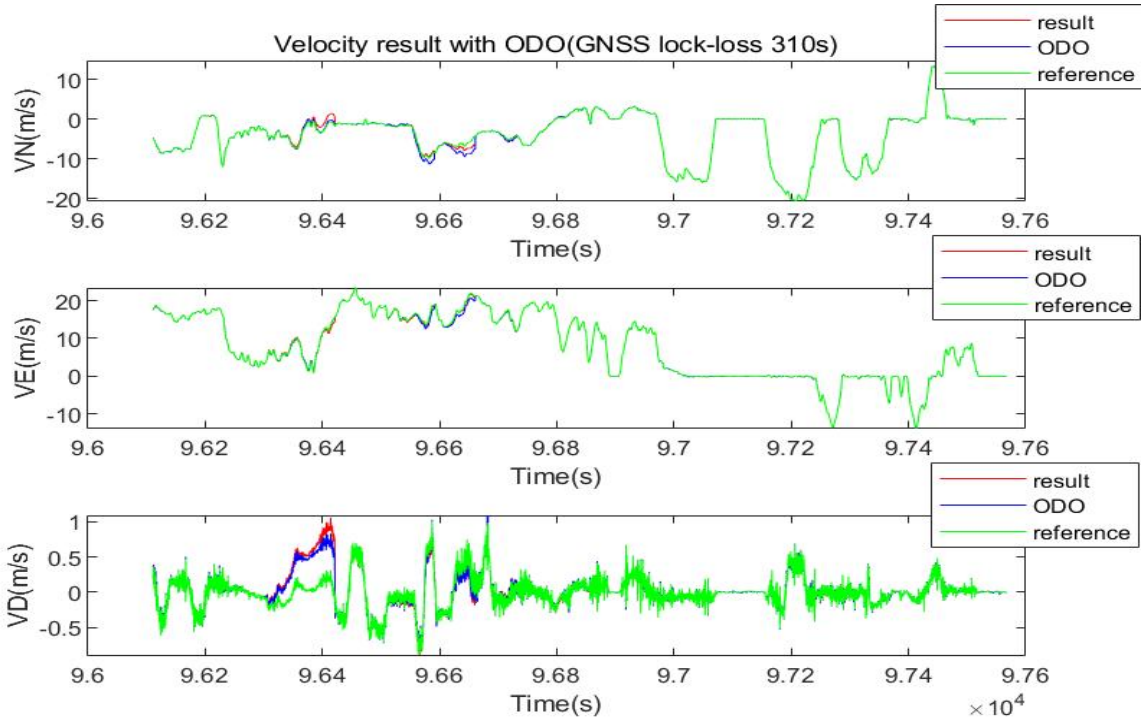

Figure 3.14: 载体速度解算结果 (ODO 更新)

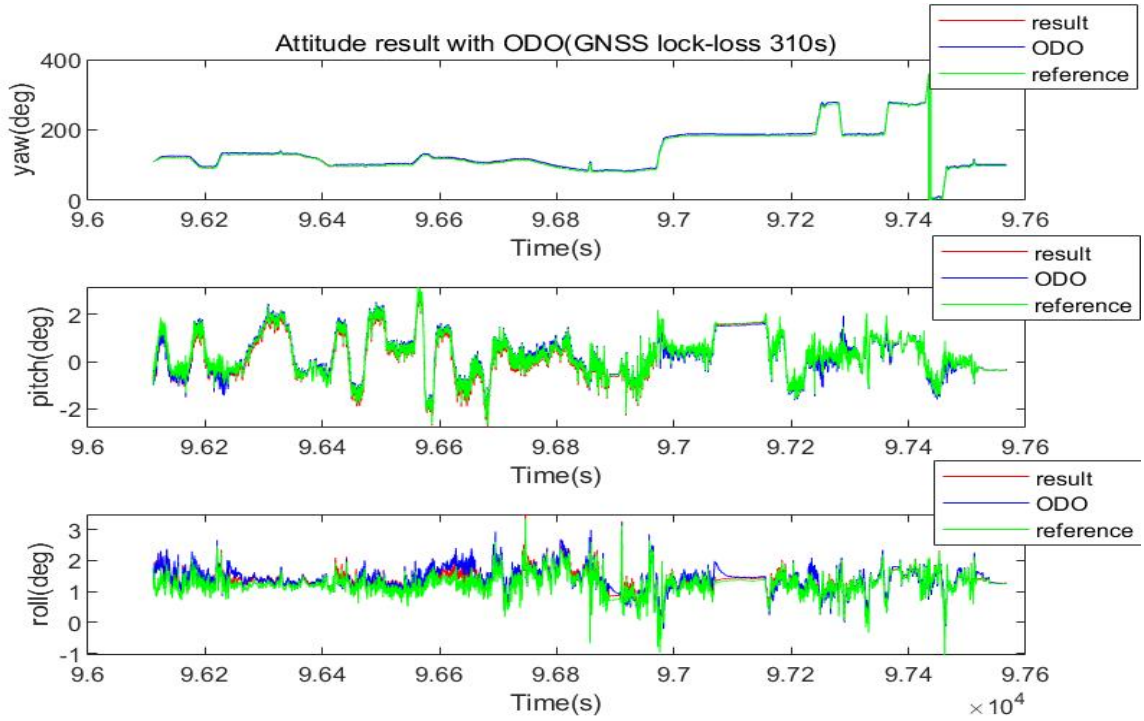

Figure 3.15: 载体姿态解算结果 (ODO 更新)

可以看到 NHC 的效果在失锁时段十分显著，非常有效的抑制了惯导在 GNSS 失锁时刻的误差累计和状态飘移，加入 NHC 后与参考值几乎一致。加入 ODO 后虽然也会发生失锁时段后的飘移，但是会稍稍对飘移程度有抑制作用，使用 ODO 后比不使用的结果较参考真值更为接近。

猜测造成 NHC 与 ODO 的效果与失锁时段有关的原因是由于载体运动速度不高的情况下，再去引用虚拟的侧向 0 速度值和前向的测量速度值进行状态更新对惯导本身状态发散的拉回效果不大，反而还加入了更多的噪声和不确定状态，造成比不加入时发散更快的现象。

使用 NHC 加 ODO 的组合算法加入组合导航中，设置 NHC 的 R 阵为 $\begin{bmatrix} 0.01 & 0.00 \\ 0.00 & 0.01 \end{bmatrix}$,ODO 的 R 阵为 [0.01] 时，进行数据解算后，发现会对解算航向角造成约有 3 到 5 度的偏差，北向速度则有约 0.5m/s 的偏差。再次调整 NHC 的 R 阵为 $\begin{bmatrix} 0.02 & 0.00 \\ 0.00 & 0.02 \end{bmatrix}$，ODO 为 [0.04] 时，效果较好，绘制图如下：

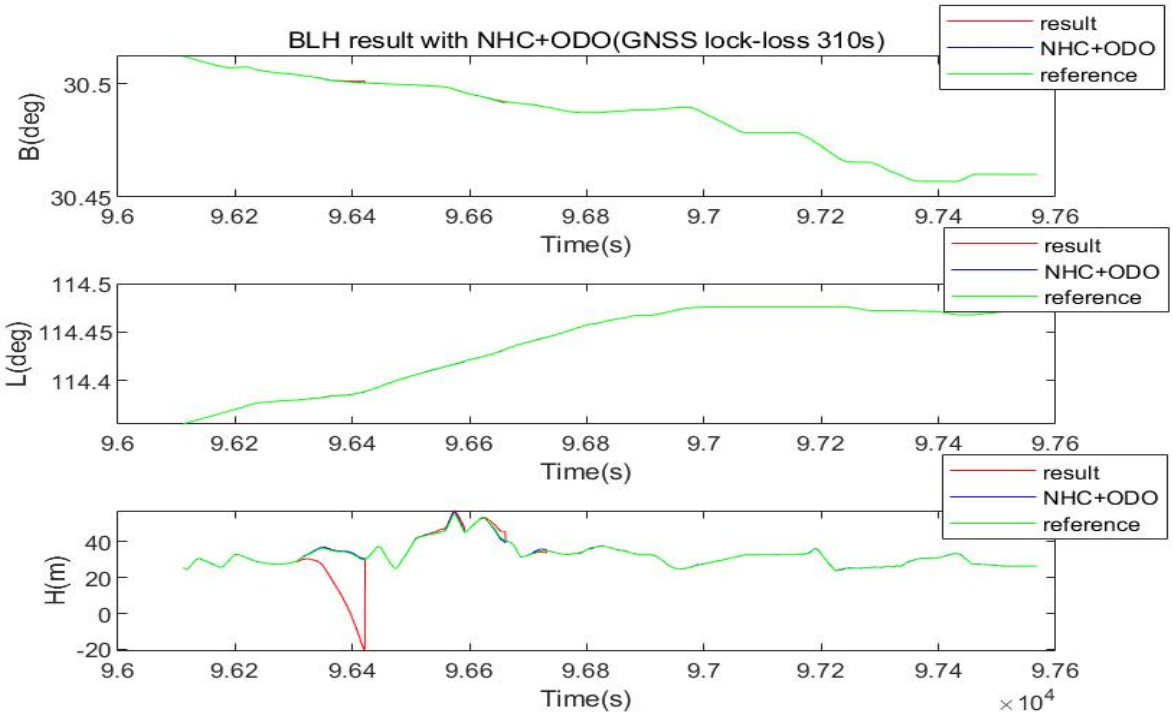

Figure 3.16: 载体位置解算结果 (NHC+ODO 更新)

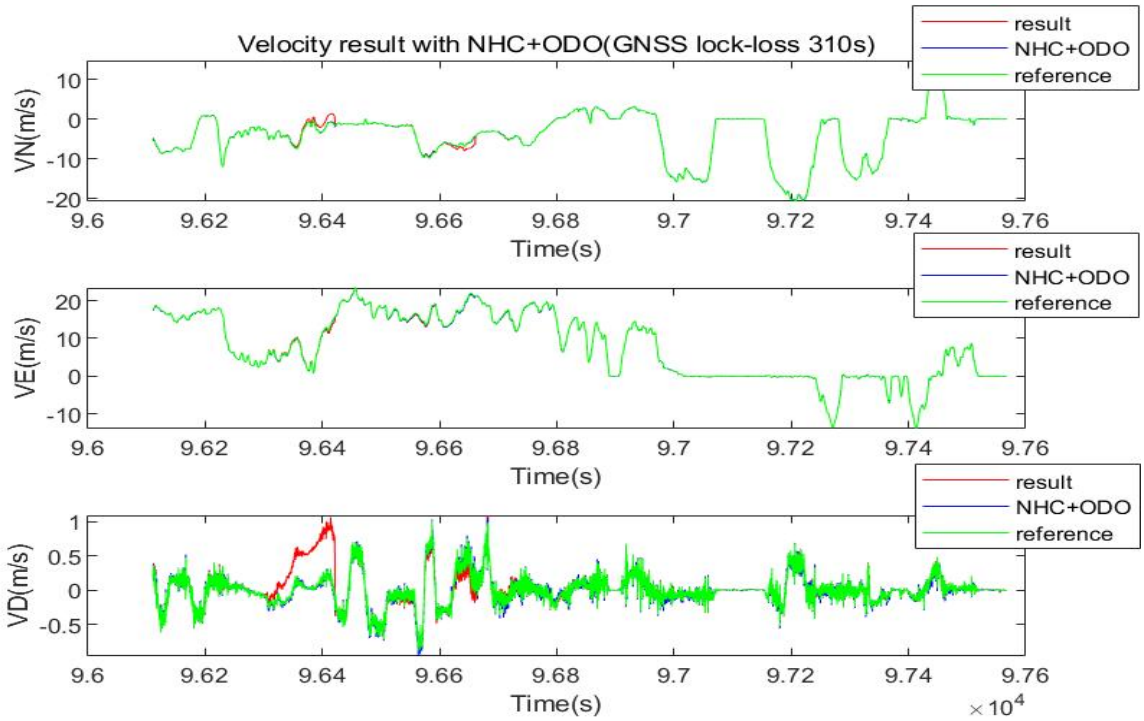

Figure 3.17: 载体速度解算结果 (NHC+ODO 更新)

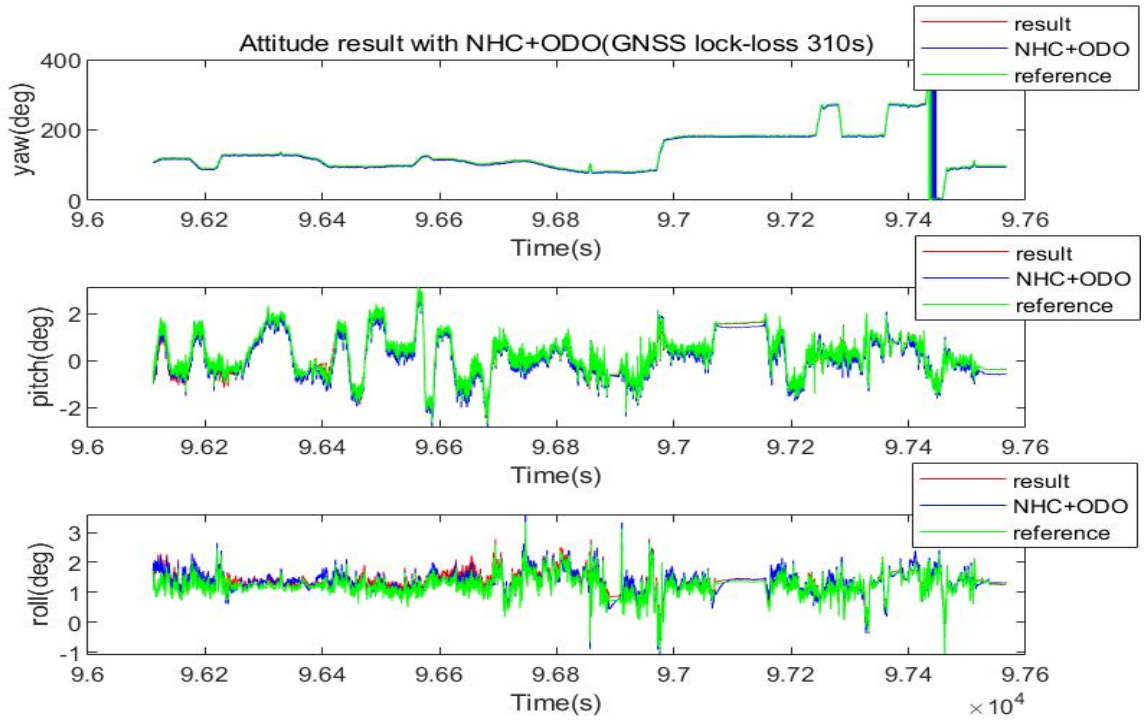

Figure 3.18: 载体姿态解算结果 (NHC+ODO 更新)

可以看到 NHC 加 ODO 对惯导状态在 GNSS 失锁时段的漂移的抑制作用十分显著，加入后几乎与参考曲线完全重合，实现了 GNSS 失锁状态下惯导依然能持续运行并提供较为可靠的位置速度姿态信息。

接下来将上述几种方法一并绘制图像比较差别：

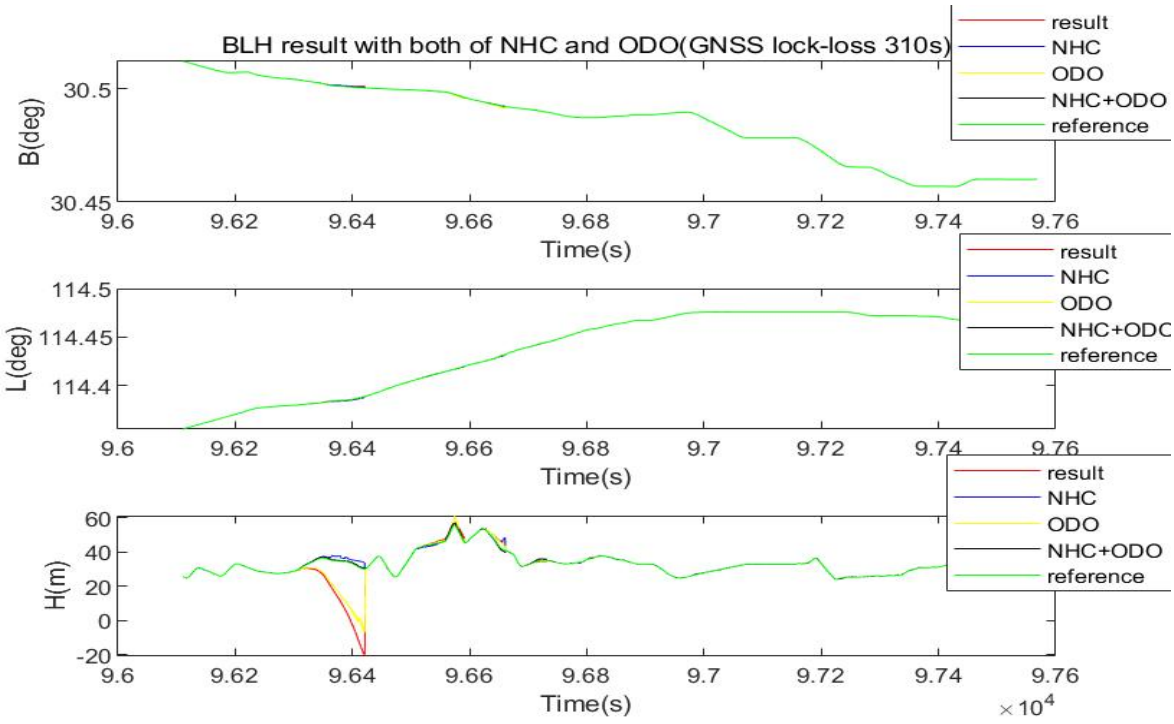

Figure 3.19: 载体位置解算结果 (NHC+ODO 更新)

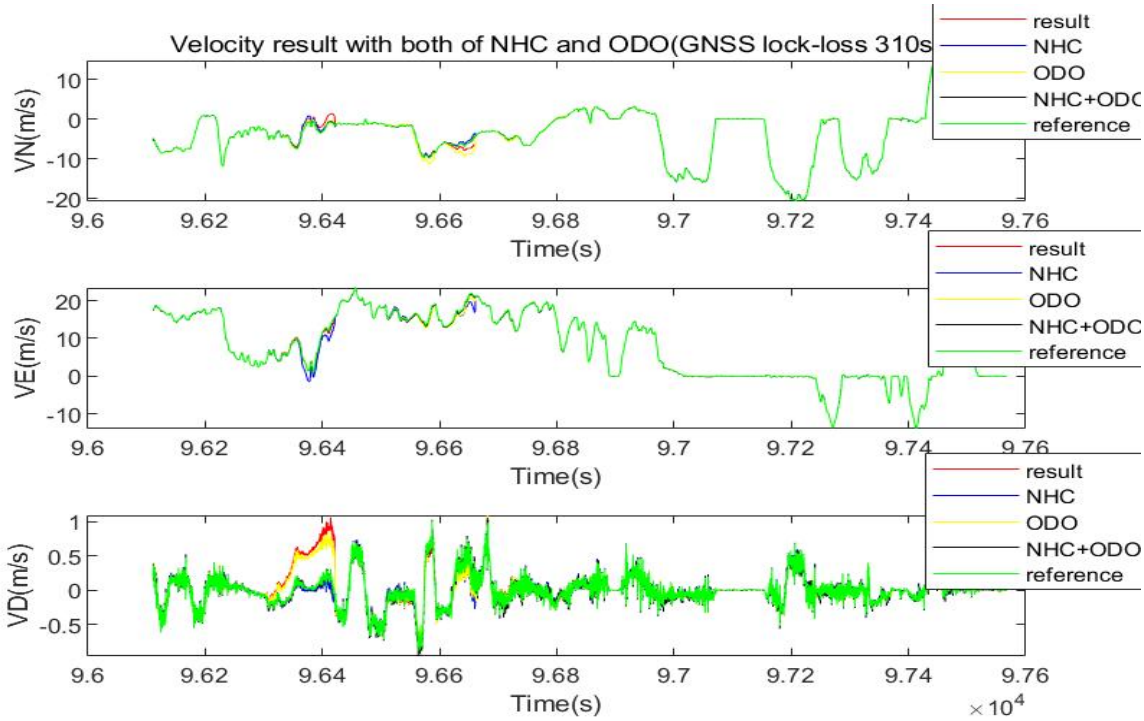

Figure 3.20: 载体速度解算结果 (NHC+ODO 更新)

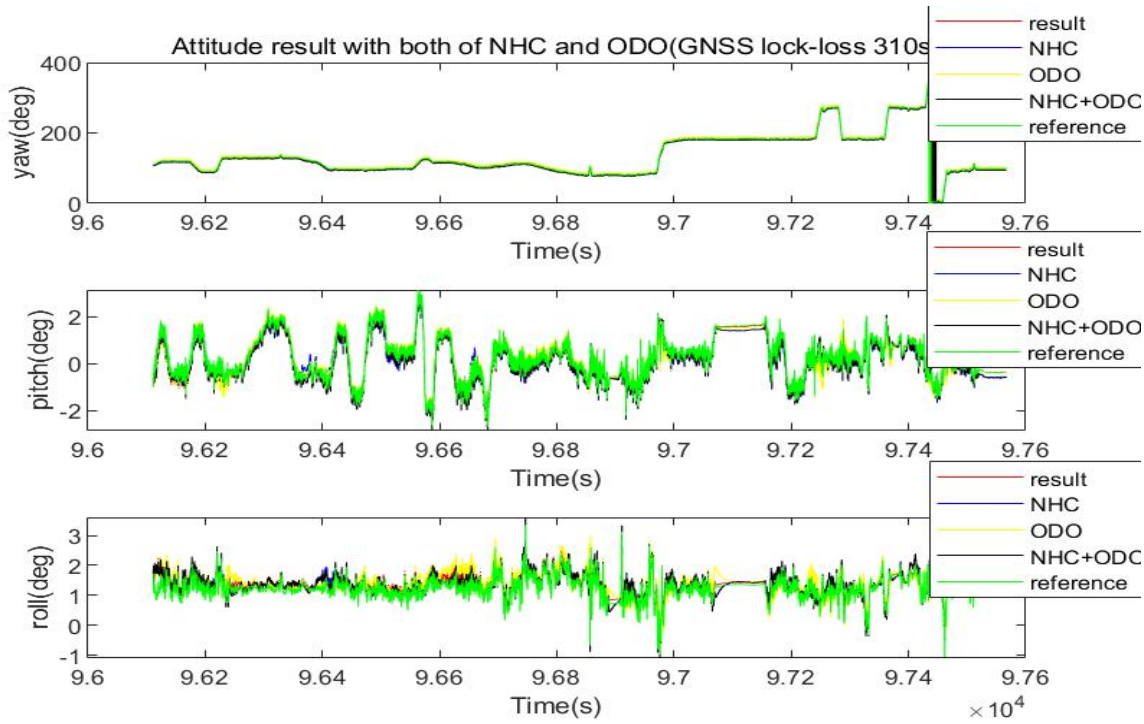

Figure 3.21: 载体姿态解算结果 (NHC+ODO 更新)

比较可以得出 NHC+ODO 同时使用时对惯导漂移的抑制效果最好，且在非失锁时刻对于卡尔曼滤波的正常运行，惯导的状态传递和 GNSS 测量更新不会造成负面影响。

Chapter 4　心得体会

经过这次惯导松组合的实现，我对本来只是模糊的一个概念的松组合算法有了更清晰的认识，对它的每一个公式的细节和实际编程实现需要注意的操作也有了较为全面的认知。所谓的松组合算法的简单可能只是相对其他算法而言的，但真正实现一遍松组合算法，将惯导与 GNSS 数据进行有效组合还是一个非常复杂且繁琐的过程，如起始的历元是否需要内插以及如何内插，各个 IMU 误差参数如何转换为标准单位等等问题，并且只要有一处疏忽，整个程序的结果将会离实际值越偏越大，最后检查也无从查起。所以以后应养成一个习惯，把外部的数据进来是就换成标准单位，想输出到控制台或文件时再转为常用单位。

此外，对于实验中遇到的问题，这次是对于 NHC 与 ODO 的改正效果非常不好，比原来不加的效果漂移更大，并且无论如何检查代码都没有找到任何错误，最后抱着试一试的心态将数据做了改动，总算得到了预期效果。看来从多方面分析问题也是十分重要的一环。

最后对课程提一些意见。希望可以在课堂上在多花时间介绍松组合具体的算法，对其具体的卡尔曼滤波状态量，状态方程，观测方程以及对惯导和 GNSS 的数据时间同步能有详细的讲解，否则在动手实现的过程中还是要花大量时间找资料理解公式等，导致最终实现中花了一整个寒假还没写完。另外在课程要求中给出的一个单位转换有个小错误，应该是，$1Gal = 1cm/s^2$, 而不是 $1mGal = 1cm/s^2$。