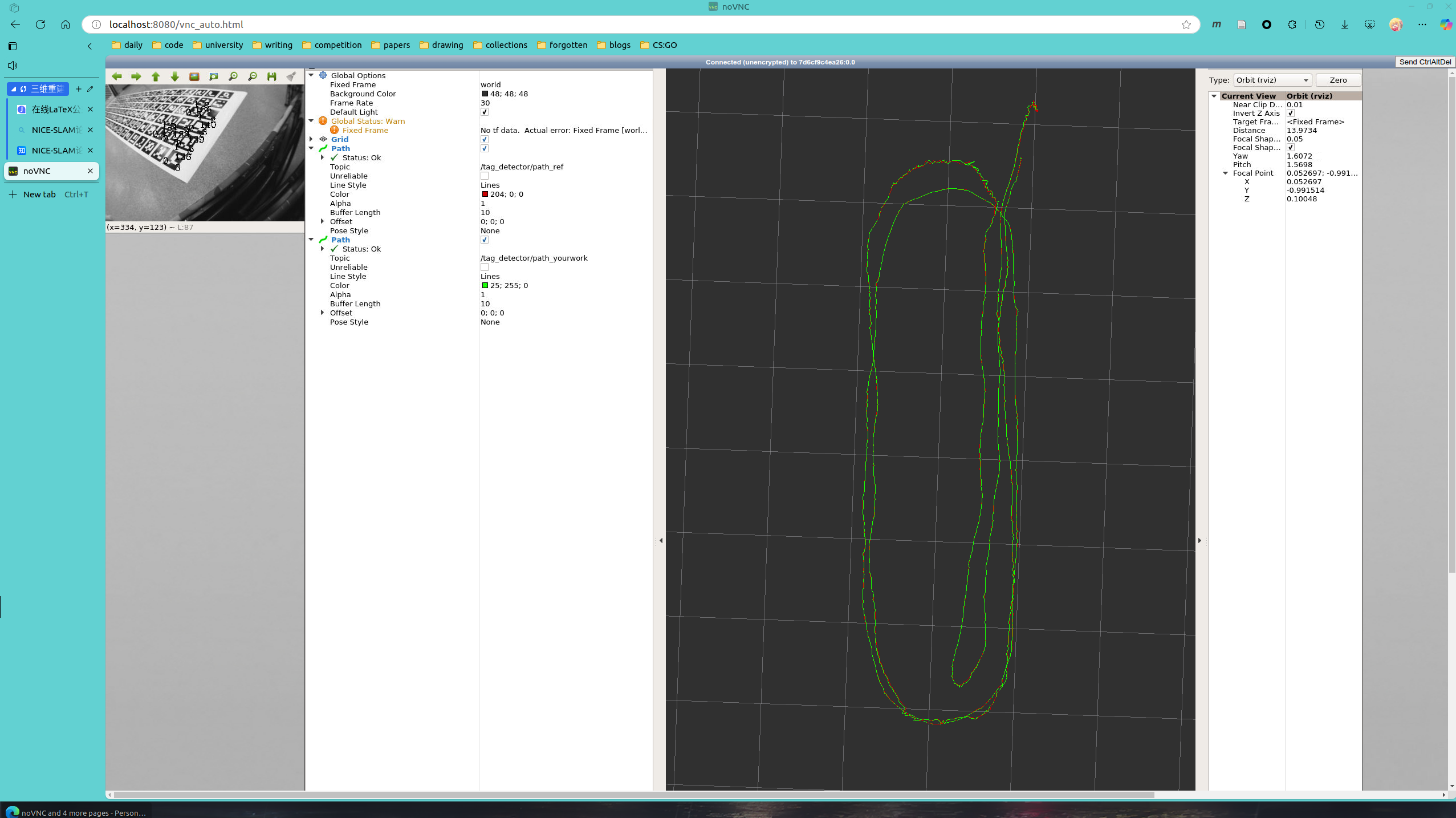
proj2phase1-徐涵-202293010207

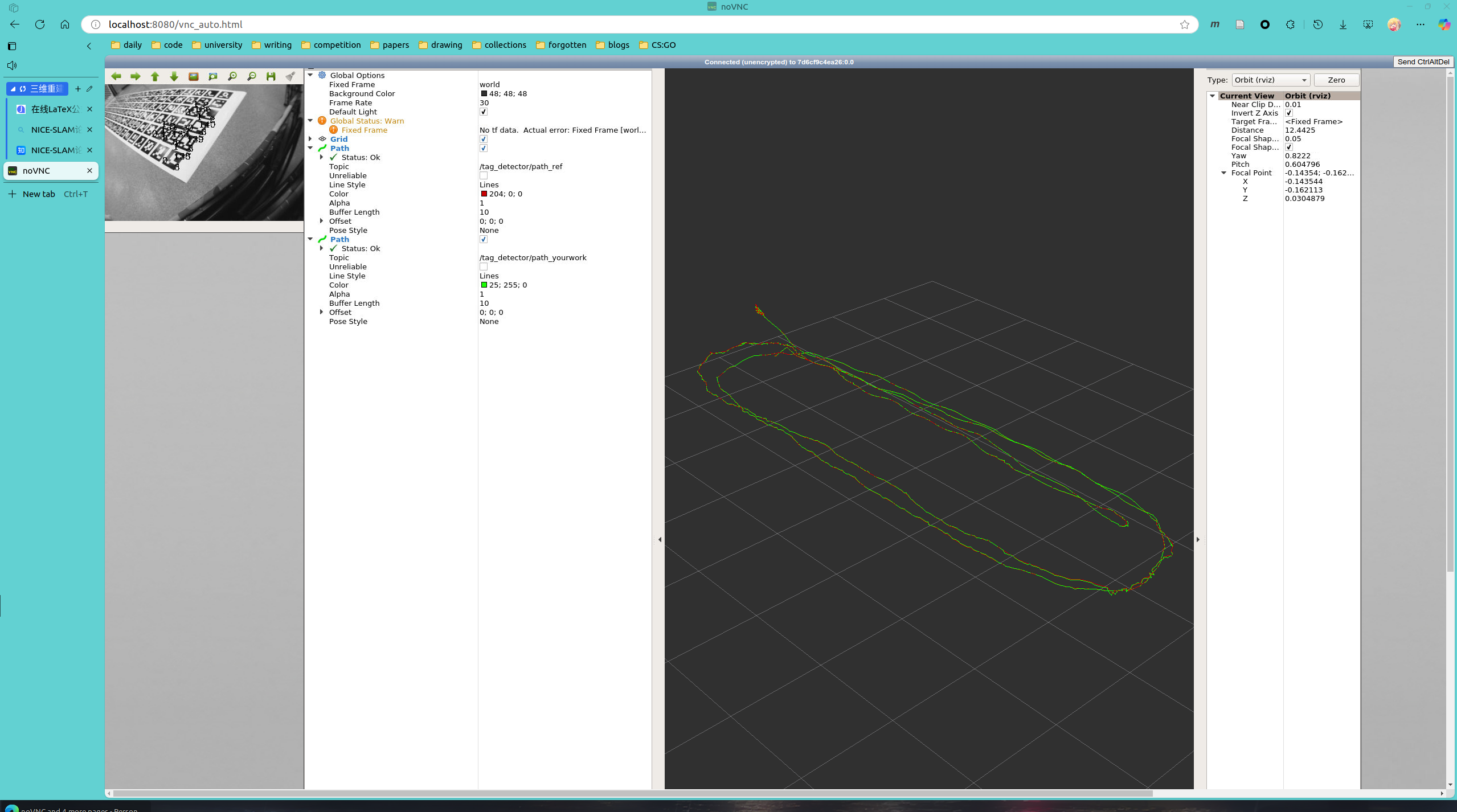
1. rivz结果展示：完整截图保存在assets目录中。

绿色为使用梯度下降算法的优化后的相机pose，红色为使用PnP解算出的相机Pose。

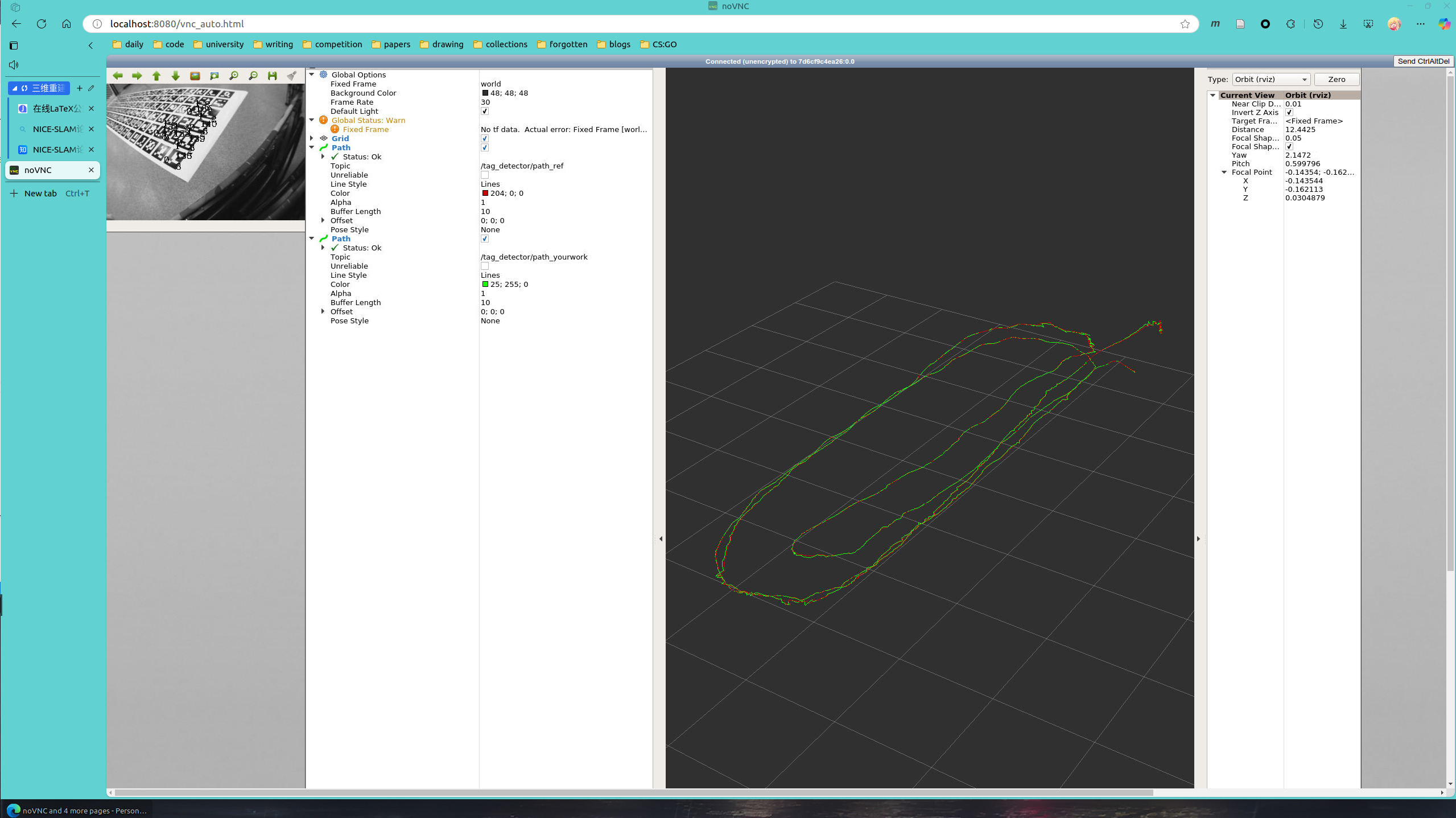
**(bonus)如果有毛刺，是由于优化算法收敛于局部最优导致的。可以看到，我的结果几乎消除了所有毛刺。**



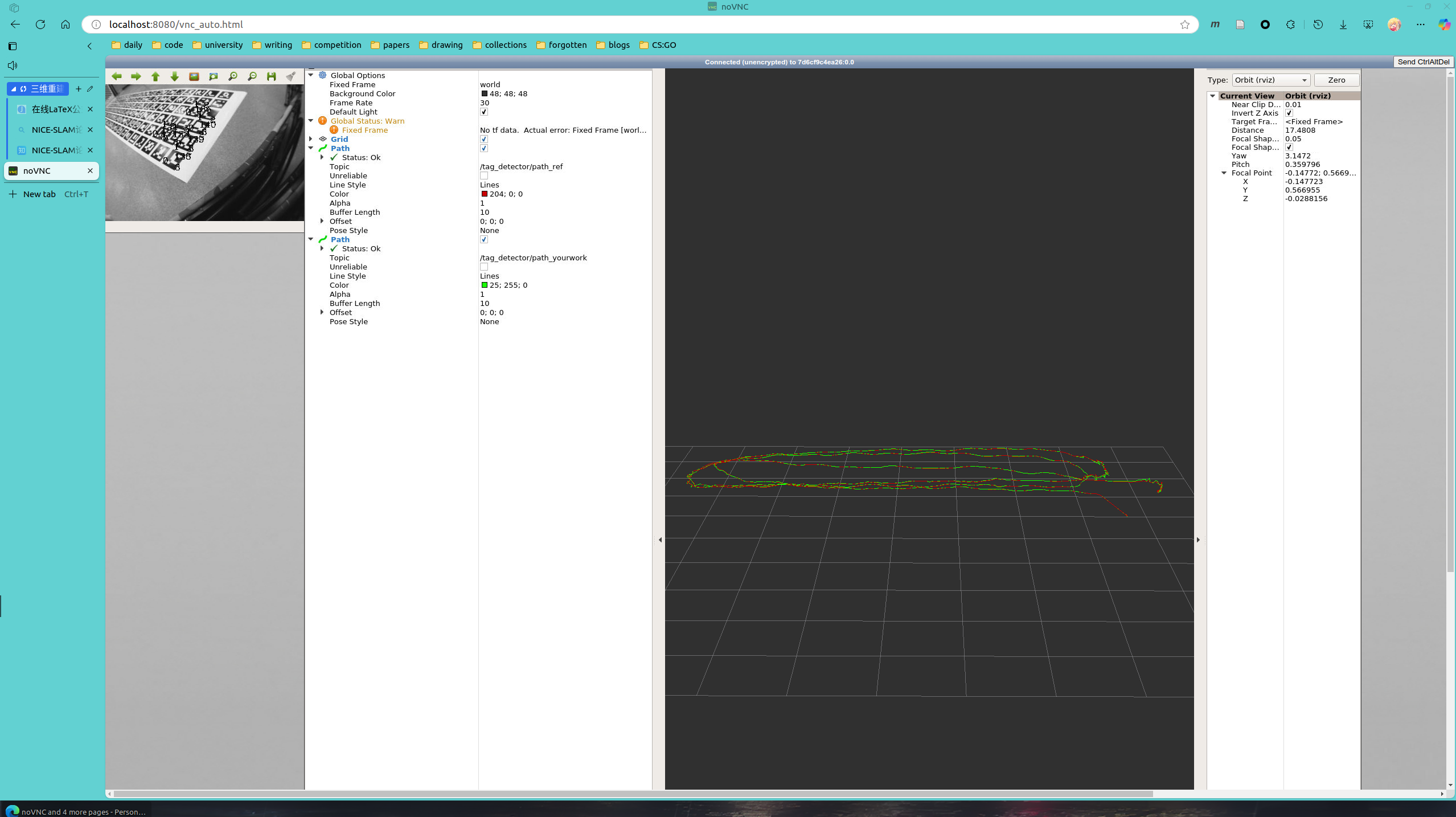
result1



result2



result3



result4

(b) Statistics about the result.这里我计算了像素坐标系下和归一化相机坐标系下两种情况的error。可以看到像素坐标系下error数值比较大，这是因为一个像素对应的物理尺寸是非常小的，所以这个结果也是合理的。而在归一化相机坐标系中，error就非常小了，所以rviz图中的绿线和红线基本是完全吻合的。

像素坐标系下的error：

[ INFO] [1732962306.785898367]: RMSE X, Y, Z, roll, pitch, yaw:

0.001188, 0.001016, 0.006289, 0.003171, 0.001591, 0.000319

ref error75.4489

res error74.7122

[ INFO] [1732962306.826466623]: RMSE X, Y, Z, roll, pitch, yaw:

0.001188, 0.001015, 0.006291, 0.003170, 0.001593, 0.000319

ref error74.2416

res error72.8336

[ INFO] [1732962306.860473438]: RMSE X, Y, Z, roll, pitch, yaw:

0.001187, 0.001014, 0.006295, 0.003169, 0.001595, 0.000318

ref error90.1345

res error88.1427

[ INFO] [1732962306.888490535]: RMSE X, Y, Z, roll, pitch, yaw:

0.001186, 0.001013, 0.006295, 0.003167, 0.001602, 0.000318

ref error74.0466

res error73.119

[ INFO] [1732962306.921782520]: RMSE X, Y, Z, roll, pitch, yaw:

0.001185, 0.001012, 0.006292, 0.003164, 0.001606, 0.000318

ref error67.4535

res error66.3266

归一化相机坐标系下的error：  
[ INFO] [1732962553.021147856]: RMSE X, Y, Z, roll, pitch, yaw:

0.001009, 0.001212, 0.003643, 0.002641, 0.001275, 0.000326

ref error0.399912

res error0.400227

[ INFO] [1732962553.054152883]: RMSE X, Y, Z, roll, pitch, yaw:

0.001007, 0.001218, 0.003665, 0.002640, 0.001273, 0.000328

ref error0.425551

res error0.426367

[ INFO] [1732962553.082345136]: RMSE X, Y, Z, roll, pitch, yaw:

0.001005, 0.001221, 0.003679, 0.002636, 0.001271, 0.000328

ref error0.424916

res error0.422953

[ INFO] [1732962553.115326532]: RMSE X, Y, Z, roll, pitch, yaw:

0.001002, 0.001223, 0.003689, 0.002633, 0.001268, 0.000328

ref error0.433257

res error0.431232

[ INFO] [1732962553.153791722]: RMSE X, Y, Z, roll, pitch, yaw:

0.001001, 0.001229, 0.003718, 0.002633, 0.001265, 0.000329

ref error0.410064

res error0.411601

对应计算代码：

void calculateReprojectionError(

const vector<cv::Point3f> &pts\_3, // the input 3D points

const vector<cv::Point2f> &pts\_2, // the input 2D features that are corresponding to the 3D points

const cv::Mat R, // the under-estimated rotation matrix

const cv::Mat t){ // the under-estimated translation{

// 本地初始化重投影误差

double sum=0;

// 校正畸变，undistortPoints得到的是归一化相机坐标系

vector<cv::Point2f> corrected\_pts\_2;

cv::undistortPoints(pts\_2, corrected\_pts\_2, K, D);

for (size\_t i = 0; i < pts\_3.size(); i++){

cv::Mat point\_w(3, 1, CV\_64FC1);

point\_w.at<double>(0, 0) = pts\_3[i].x;

point\_w.at<double>(1, 0) = pts\_3[i].y;

point\_w.at<double>(2, 0) = pts\_3[i].z;

// 转相机坐标系并归一化

cv::Mat p\_camera = (R \* point\_w + t);

cv::Mat p\_camera\_homogeneous = p\_camera/p\_camera.at<double>(2);

// 如果在像素坐标系下...

// cv::Mat p\_pixel = K \* p\_camera\_homogeneous;

// float fx = K.at<double>(0, 0);

// float fy = K.at<double>(1, 1);

// float cx = K.at<double>(0, 2);

// float cy = K.at<double>(1, 2);

// cv::Point2f temp\_pts\_2\_pixel(fx\*corrected\_pts\_2[i].x+cx,fy\*corrected\_pts\_2[i].y+cy);

// sum = sum + sqrt(pow(temp\_pts\_2\_pixel.x-p\_pixel.at<double>(0),2)+pow(temp\_pts\_2\_pixel.y-p\_pixel.at<double>(1),2));

// 如果在归一化相机坐标系下...

sum = sum + sqrt(pow(corrected\_pts\_2[i].x-p\_camera\_homogeneous.at<double>(0),2)+

pow(corrected\_pts\_2[i].y-p\_camera\_homogeneous.at<double>(1),2));

}

reproj\_error = sum;

}

(c) Descriptions about the implementation.

1.使用PnP算法计算和

cv::solvePnPRansac(pts\_3, pts\_2, K, D, rvec, t);

cv::Rodrigues(rvec, r);

2.发布到rviz，需要和

// publish reference path. Use publishPath().

Quaterniond Q\_ref;

Eigen::Vector3d t\_ref;

for(size\_t i = 0; i < 3; i++)

t\_ref(i) = t.at<double>(i,0);

Q\_ref = R\_ref.inverse();t\_ref = -R\_ref.inverse()\*t\_ref;

publishPath(frame\_time, t\_ref, Q\_ref, path\_ref, pub\_path\_ref);

3.计算初始解，按照ppt中对应方法即可

Eigen::MatrixXd A(2\*pts\_3.size(),9);

for (uint i=0;i<pts\_3.size();i++){

A.block<2,9>(2\*i,0) << pts\_3[i].x, pts\_3[i].y, 1, 0,0,0, -corrected\_pts\_2[i].x\*pts\_3[i].x, -corrected\_pts\_2[i].x\*pts\_3[i].y, -corrected\_pts\_2[i].x,

0,0,0, pts\_3[i].x, pts\_3[i].y, 1, -corrected\_pts\_2[i].y\*pts\_3[i].x, -corrected\_pts\_2[i].y\*pts\_3[i].y, -corrected\_pts\_2[i].y;

}

Eigen::JacobiSVD<MatrixXd> A\_svd(A, Eigen::ComputeThinU | Eigen::ComputeThinV);

Eigen::VectorXd init\_solve = A\_svd.matrixV().rightCols(1);

Eigen::Matrix3d H\_hat, H\_orthogonal;

H\_hat << init\_solve[0], init\_solve[1], init\_solve[2], init\_solve[3], init\_solve[4], init\_solve[5], init\_solve[6], init\_solve[7], init\_solve[8];

if (H\_hat(2,2)<0){

H\_hat = -H\_hat;

}

H\_orthogonal << H\_hat.col(0), H\_hat.col(1), H\_hat.col(0).cross(H\_hat.col(1));

Eigen::JacobiSVD<MatrixXd> H\_hat\_svd(H\_orthogonal, Eigen::ComputeThinU | Eigen::ComputeThinV);

Eigen::Matrix3d R\_hat;Eigen::Vector3d T\_hat;R\_hat.setIdentity();T\_hat.setZero();

R\_hat = H\_hat\_svd.matrixU()\*(H\_hat\_svd.matrixV().transpose());

T\_hat = H\_hat.col(2)/(H\_hat.col(0).norm());

4.使用梯度下降算法进行优化迭代：这里我写了在像素坐标系下和归一化相机坐标系下两个版本，都是可以的。

4.1 准备阶段，我选择按ZXY顺序分解

// 残差初始化

Eigen::MatrixXd gamma(2, pts\_3.size());

// 雅可比矩阵的列表

Eigen::MatrixXd JacobianMatrix[pts\_3.size()];

// 最大迭代次数

int iter\_threshold = 1000;

// 待优化参数初始化

Eigen::VectorXd parameters(6, 1);

// 按psi,phi,theta顺序分解并连接它们

Eigen::Vector3d eulerAngles = R\_hat.eulerAngles(2, 0, 1);

parameters << eulerAngles[0],eulerAngles[1],eulerAngles[2],T\_hat[0],T\_hat[1],T\_hat[2];

Eigen::MatrixXd tmp\_J(2,6);

4.2 迭代过程

4.2.1 计算残差，注意残差是target在前，prediction在后，因此所有的Jacobian在求导后要加上负号

// compute residual,target在前，prediction在后

Eigen::Vector3d pt\_3\_eigen(pts\_3[i].x, pts\_3[i].y, pts\_3[i].z);

Eigen::Vector3d p\_camera = temp\_R \* pt\_3\_eigen + temp\_T;

Eigen::Vector3d p\_camera\_homogeneous = p\_camera / p\_camera[2];

// 如果在像素坐标系下...

// 貌似版本过低，无法使用这个函数，手动赋值

// Eigen::Matrix3d K\_eigen;

// // Eigen::cv2eigen(K,K\_eigen);

// for (size\_t i = 0; i < 3; i++){

// for (size\_t j = 0; j < 3; j++){

// K\_eigen(i, j) = K.at<double>(i, j);

// }

// }

// Eigen::Vector3d p\_pixel = K\_eigen \* p\_camera\_homogeneous;

// float fx = K.at<double>(0, 0);

// float fy = K.at<double>(1, 1);

// float cx = K.at<double>(0, 2);

// float cy = K.at<double>(1, 2);

// cv::Point2f temp\_pts\_2\_pixel(fx\*corrected\_pts\_2[i].x+cx,fy\*corrected\_pts\_2[i].y+cy);

// gamma.block<2,1>(0,i) << temp\_pts\_2\_pixel.x - p\_pixel[0],

// temp\_pts\_2\_pixel.y - p\_pixel[1];

// 如果在归一化相机坐标系下...

gamma.block<2,1>(0,i) << corrected\_pts\_2[i].x - p\_camera\_homogeneous[0],

corrected\_pts\_2[i].y - p\_camera\_homogeneous[1];

4.2.2 计算Jacobian，如果在像素坐标系下，对应的Jacobian只需要多乘以一个fx或fy。Jacobian的数学计算公式过程从略。

// 如果在归一化相机坐标系下...

gamma.block<2,1>(0,i) << corrected\_pts\_2[i].x - p\_camera\_homogeneous[0],

corrected\_pts\_2[i].y - p\_camera\_homogeneous[1];

double x,y,z;x = pts\_3[i].x;y = pts\_3[i].y;z = pts\_3[i].z;

double dpcamera1\_dpsi = (-sin(psi)\*cos(theta)-cos(psi)\*sin(phi)\*sin(theta))\*x

-cos(psi)\*cos(phi)\*y

+ (-sin(psi)\*cos(theta)+cos(theta)\*cos(psi)\*sin(phi))\*z;

double dpcamera3\_dpsi = 0;

double dpcamera1\_dphi = -sin(psi)\*cos(phi)\*sin(theta)\*x

+sin(phi)\*sin(psi)\*y

+cos(theta)\*sin(psi)\*cos(phi)\*z;

double dpcamera3\_dphi = sin(phi)\*sin(theta)\*x

+cos(phi)\*y

-sin(phi)\*cos(theta)\*z;

double dpcamera1\_dtheta = (-cos(psi)\*sin(theta)-sin(psi)\*sin(phi)\*cos(theta))\*x

+(-cos(psi)\*sin(theta)-sin(theta)\*sin(psi)\*sin(phi))\*z;

double dpcamera3\_dtheta = -cos(phi)\*cos(theta)\*x

-cos(phi)\*sin(theta)\*z;

double dpcamera2\_dpsi = (cos(theta)\*cos(psi)-sin(psi)\*sin(phi)\*sin(theta))\*x

-sin(psi)\*cos(phi)\*y

+(cos(psi)\*sin(theta)+sin(psi)\*sin(phi)\*cos(theta))\*z;

double dpcamera2\_dphi = cos(psi)\*cos(phi)\*sin(theta)\*x

-cos(psi)\*sin(phi)\*y

-cos(psi)\*cos(phi)\*cos(theta)\*z;

double dpcamera2\_dtheta = (-sin(theta)\*sin(psi)+cos(psi)\*sin(phi)\*cos(theta))\*x

+(sin(psi)\*cos(theta)+cos(psi)\*sin(theta)\*sin(phi))\*z;

// 如果是在像素坐标系下...

// du/dt1

// tmp\_J(0, 3) = -fx\*(1/p\_camera[2]);

// // du/dt2

// tmp\_J(0, 4) = 0;

// // du/dt3

// tmp\_J(0, 5) = -fx\*(-p\_camera[0]/(p\_camera[2]\*p\_camera[2]));

// // dv/dt1

// tmp\_J(1, 3) = 0;

// // dv/dt2

// tmp\_J(1, 4) = -fy\*(1/p\_camera[2]);

// // dv/dt3

// tmp\_J(1, 5) = -fy\*(-p\_camera[1]/(p\_camera[2]\*p\_camera[2]));

// // du/dpsi

// tmp\_J(0, 0) = -fx\*(dpcamera1\_dpsi\*p\_camera[2]-p\_camera[01]\*dpcamera3\_dpsi)/(p\_camera[2]\*p\_camera[2]);

// // du/dphi

// tmp\_J(0, 1) = -fx\*(dpcamera1\_dphi\*p\_camera[2]-p\_camera[0]\*dpcamera3\_dphi)/(p\_camera[2]\*p\_camera[2]);

// // du/dtheta

// tmp\_J(0, 2) = -fx\*(dpcamera1\_dtheta\*p\_camera[2]-p\_camera[0]\*dpcamera3\_dtheta)/(p\_camera[2]\*p\_camera[2]);

// // dv/dpsi

// tmp\_J(1, 0) = -fy\*(dpcamera2\_dpsi\*p\_camera[2]-p\_camera[1]\*dpcamera3\_dpsi)/(p\_camera[2]\*p\_camera[2]);

// // dv/dphi

// tmp\_J(1, 1) = -fy\*(dpcamera2\_dphi\*p\_camera[2]-p\_camera[1]\*dpcamera3\_dphi)/(p\_camera[2]\*p\_camera[2]);

// // dv/dtheta

// tmp\_J(1, 2) = -fy\*(dpcamera2\_dtheta\*p\_camera[2]-p\_camera[1]\*dpcamera3\_dtheta)/(p\_camera[2]\*p\_camera[2]);

// // 如果是在归一化相机坐标系下...

// du/dt1

tmp\_J(0, 3) = -(1/p\_camera[2]);

// du/dt2

tmp\_J(0, 4) = 0;

// du/dt3

tmp\_J(0, 5) = -(-p\_camera[0]/(p\_camera[2]\*p\_camera[2]));

// dv/dt1

tmp\_J(1, 3) = 0;

// dv/dt2

tmp\_J(1, 4) = -(1/p\_camera[2]);

// dv/dt3

tmp\_J(1, 5) = -(-p\_camera[1]/(p\_camera[2]\*p\_camera[2]));

// du/dpsi

tmp\_J(0, 0) = -(dpcamera1\_dpsi\*p\_camera[2]-p\_camera[01]\*dpcamera3\_dpsi)/(p\_camera[2]\*p\_camera[2]);

// du/dphi

tmp\_J(0, 1) = -(dpcamera1\_dphi\*p\_camera[2]-p\_camera[0]\*dpcamera3\_dphi)/(p\_camera[2]\*p\_camera[2]);

// du/dtheta

tmp\_J(0, 2) = -(dpcamera1\_dtheta\*p\_camera[2]-p\_camera[0]\*dpcamera3\_dtheta)/(p\_camera[2]\*p\_camera[2]);

// dv/dpsi

tmp\_J(1, 0) = -(dpcamera2\_dpsi\*p\_camera[2]-p\_camera[1]\*dpcamera3\_dpsi)/(p\_camera[2]\*p\_camera[2]);

// dv/dphi

tmp\_J(1, 1) = -(dpcamera2\_dphi\*p\_camera[2]-p\_camera[1]\*dpcamera3\_dphi)/(p\_camera[2]\*p\_camera[2]);

// dv/dtheta

tmp\_J(1, 2) = -(dpcamera2\_dtheta\*p\_camera[2]-p\_camera[1]\*dpcamera3\_dtheta)/(p\_camera[2]\*p\_camera[2]);

4.2.3 应用更新并进行收敛性判断

// update the optimization variables

Eigen::VectorXd delta\_p = An.inverse()\*bn;

parameters += delta\_p;

// std::cout << "update:" << delta\_p<< endl;

if (delta\_p.cwiseAbs().sum() < 0.0001){

break;

}

5.优化结果发布到rviz，仍然需要和

Eigen::Matrix3d R;R = AngleAxisd(parameters[0], Vector3d::UnitZ()) \* AngleAxisd(parameters[1], Vector3d::UnitX()) \* AngleAxisd(parameters[2], Vector3d::UnitY());

Eigen::Vector3d T(parameters[3],parameters[4],parameters[5]);

// publish path

Quaterniond Q\_yourwork;

Q\_yourwork = R.transpose();

Eigen::Vector3d T\_wc = -R.transpose()\*T;

publishPath(frame\_time, T\_wc, Q\_yourwork, path, pub\_path);