问题1：

以front\_h60\_intrinsics.yaml文件为例

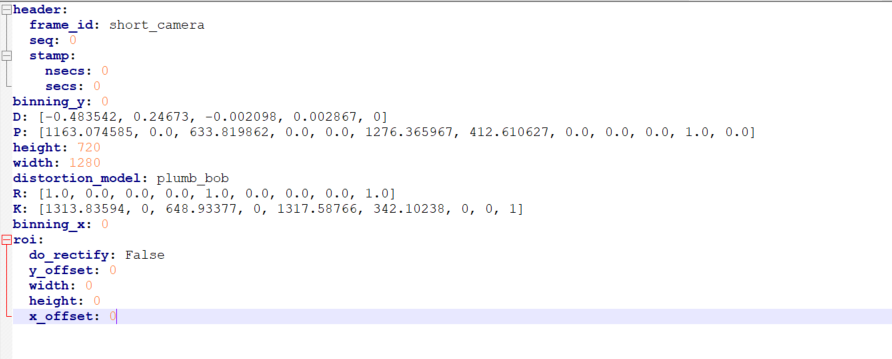


Figure 1 front\_h60\_intrinsics.yaml

请说明一下个参数代表的意义是什么？哪些是内参矩阵、哪些是畸变系数、鱼眼还是针孔相机等？

解答：

D:畸变系数矩阵

K:内参矩阵

相机内参只有D和K有用，其余参数暂时未用

针孔相机

另外一个问题个front\_h30\_intrinsics.yaml内参文件和front\_h60\_intrinsics.yaml缺少k 参数，这个k代表什么？

解答：

K：内参矩阵 每个内参文件都有；

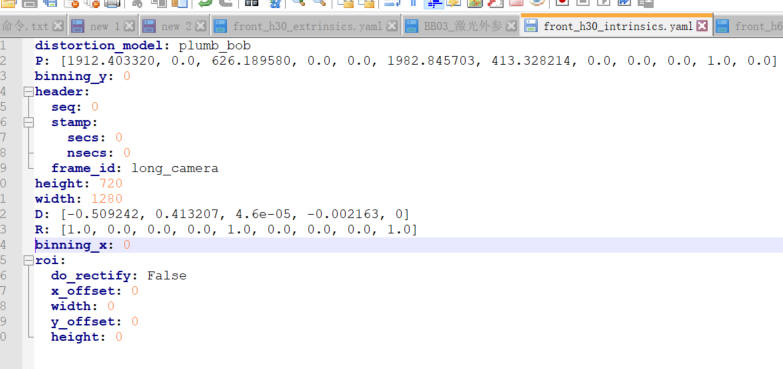


Figure 2 front\_h30\_intrinsics.yaml

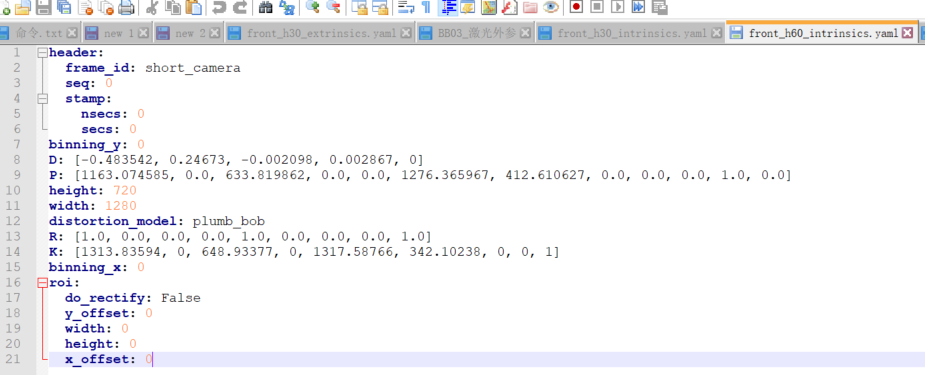


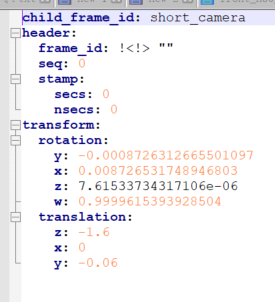
Figure 3 front\_h60\_intrinsics.yaml

问题2：

相机外参文件，这里的外参是Camera-to-LiDAR的外参？

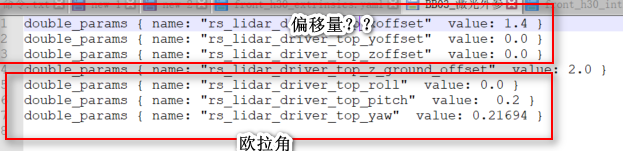
解答：

是的



问题3：

Lidar 的外参文件，上面的三个是外参矩阵的偏移量？



这里的外参是如下图LIDAR-to-OTS的外参？

解答：是的

问题4：雷达到摄像机融合问题：

基于front\_h60\_intrinsics.yaml 提取的相机内参矩阵是

[[1313.83594 0. 648.93377 0. ]

[ 0. 1317.58766 342.10238 0. ]

[ 0. 0. 1. 0. ]]

基于front\_h60\_extrinsics.yaml 提取的四元数组和平移，转换得到的外参矩阵是

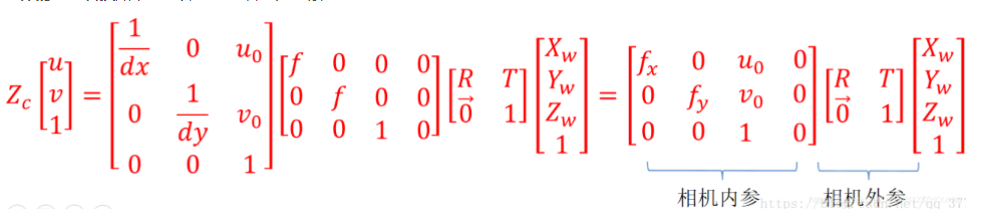
[[ 0.99999848 -0.00003046 -0.00174506 0. ]

[-0. 0.9998477 -0.01745241 -0.06 ]

[ 0.00174533 0.01745238 0.99984617 -1.6 ]

[ 0. 0. 0. 1. ]]

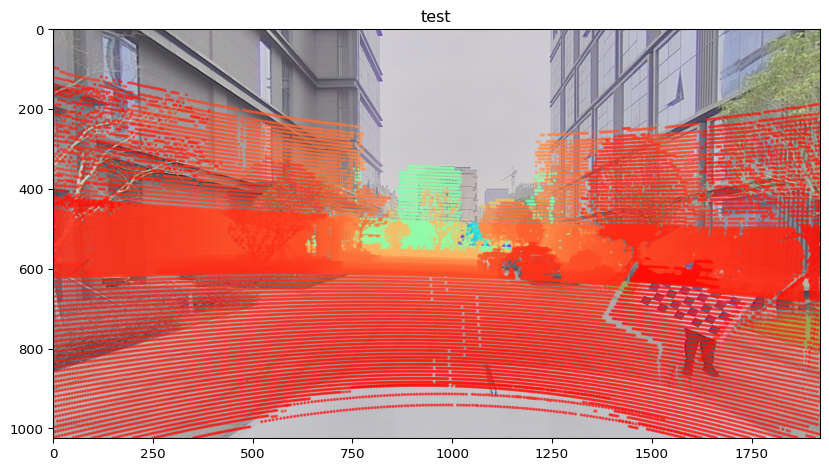
根据坐标转换



做雷达到相机的投影后，你们给的数据雷达和相机无法融合。



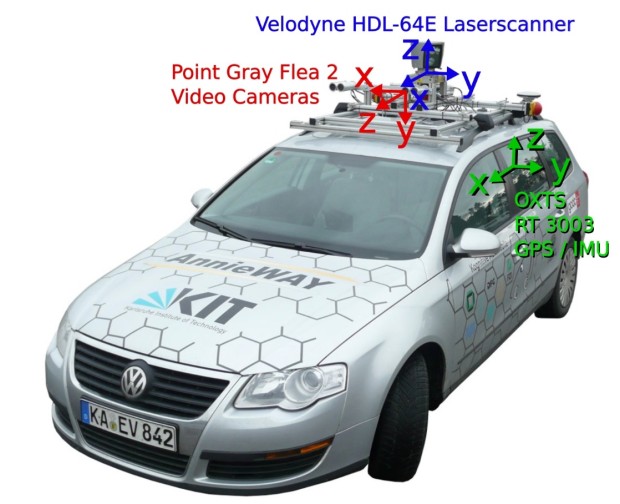
而我们的采集的雷达是有的，如下图



请帮忙检查一下问题？

解答：

相机坐标系是向前z+，向右x+,向下y+与激光雷达坐标系不同如下图所示



附激光点云到图像投影示意代码

1、eigen变量定义

Eigen::Matrix<float, 3, 3> intr\_K\_;

Eigen::Matrix<float, 1, 5> intr\_D\_;

Eigen::Matrix<float, 3, 3> extr\_R\_init\_;

Eigen::Matrix<float, 3, 3> extr\_R\_;

Eigen::Matrix<float, 3, 1> extr\_t\_;

1. 相机内参加载

bool LoadCameraIntrinsics(const std::string &yaml\_file) {

if (!FileExists(yaml\_file)) {

return false;

}

YAML::Node node = YAML::LoadFile(yaml\_file);

if (node.IsNull()) {

ROS\_ERROR\_STREAM("Load " << yaml\_file << " failed! please check!");

return false;

}

Eigen::VectorXf params(9 + 5);

try {

camera\_width\_ = node["width"].as<float>();

camera\_height\_ = node["height"].as<float>();

for (size\_t i = 0; i < 9; ++i) {

params(i) = node["K"][i].as<float>();

}

for (size\_t i = 0; i < 5; ++i) {

params(9 + i) = node["D"][i].as<float>();

}

} catch (YAML::Exception &e) {

ROS\_ERROR\_STREAM("load camera intrisic file " << yaml\_file

<< " with error, YAML exception: " << e.what());

return false;

}

// fill in camera k and d

intr\_K\_n.row(0) << params(0), params(1), params(2);

intr\_K\_.row(1) << params(3), params(4), params(5);

intr\_K\_.row(2) << params(6), params(7), params(8);

intr\_D\_.row(0) << params(9), params(10), params(11), params(12), params(13);

return true;

}

1. 相机外参加载

bool LoadCameraExtrinsics(const std::string &yaml\_file) {

if (!FileExists(yaml\_file)) {

return false;

}

YAML::Node config = YAML::LoadFile(yaml\_file);

if (!config) {

ROS\_ERROR\_STREAM("Open TransformationMatrix File:" << yaml\_file << " failed.");

return false;

}

float x, y, z;

float qx, qy, qz, qw;

try {

if (!config["transform"]) {

ROS\_ERROR\_STREAM("Open TransformationMatrix File:" << yaml\_file

<< " has no transform.");

return false;

}

if (config["transform"]["translation"]) {

x = config["transform"]["translation"]["x"].as<float>();

y = config["transform"]["translation"]["y"].as<float>();

z = config["transform"]["translation"]["z"].as<float>();

} else {

ROS\_ERROR\_STREAM("TransformationMatrix File:" << yaml\_file

<< " has no transform:translation.");

return false;

}

// fill rotation

if (config["transform"]["rotation"]) {

qx = config["transform"]["rotation"]["x"].as<float>();

qy = config["transform"]["rotation"]["y"].as<float>();

qz = config["transform"]["rotation"]["z"].as<float>();

qw = config["transform"]["rotation"]["w"].as<float>();

} else {

ROS\_ERROR\_STREAM("TransformationMatrix File:" << yaml\_file

<< " has no transform:rotation.");

return false;

}

} catch (const YAML::Exception &e) {

ROS\_ERROR\_STREAM(yaml\_file << " load failed. error:" << e.what());

ROS\_ERROR\_STREAM("Please ensure param file is exist or format is correct");

return false;

}

// fill rotation and translation

Eigen::Quaternion<float> rotation(qw, qx, qy, qz);

extr\_R\_ = rotation.toRotationMatrix();

extr\_t\_.col(0) << x, y, z;

float roll = 90 \* M\_PI / 180; // 3-x-roll

float pitch = -90 \* M\_PI / 180; // 2-y-pitch

float yaw = 0 \* M\_PI / 180; // 1-z-yaw

extr\_R\_init\_ = Eigen::AngleAxisf(yaw, Eigen::Vector3f::UnitZ())

\* Eigen::AngleAxisf(pitch, Eigen::Vector3f::UnitY())

\* Eigen::AngleAxisf(roll, Eigen::Vector3f::UnitX());

return true;

}

1. 点云到图像映射

void PixelDenormalize(cv::Point3f & pt) {

// add distortion

Eigen::Matrix<double, 2, 1> pt2d(pt.x, pt.y);

double r\_sq = pt2d[0] \* pt2d[0] + pt2d[1] \* pt2d[1];

Eigen::Matrix<double, 2, 1> pt2d\_radial =

pt2d \* (1 + intr\_D\_[0] \* r\_sq + intr\_D\_[1] \* r\_sq \* r\_sq +

intr\_D\_[4] \* r\_sq \* r\_sq \* r\_sq);

Eigen::Matrix<double, 2, 1> dpt2d;

dpt2d[0] = 2 \* intr\_D\_[2] \* pt2d[0] \* pt2d[1] +

intr\_D\_[3] \* (r\_sq + 2 \* pt2d[0] \* pt2d[0]);

dpt2d[1] = intr\_D\_[2] \* (r\_sq + 2 \* pt2d[1] \* pt2d[1]) +

2 \* intr\_D\_[3] \* pt2d[0] \* pt2d[1];

Eigen::Matrix<double, 2, 1> pt2d\_distort;

pt2d\_distort[0] = pt2d\_radial[0] + dpt2d[0];

pt2d\_distort[1] = pt2d\_radial[1] + dpt2d[1];

// add intrinsic K

double focal\_length\_x = intr\_K\_(0, 0);

double focal\_length\_y = intr\_K\_(1, 1);

double center\_x = intr\_K\_(0, 2);

double center\_y = intr\_K\_(1, 2);

pt.x = pt2d\_distort[0] \* focal\_length\_x + center\_x;

pt.y = pt2d\_distort[1] \* focal\_length\_y + center\_y;

}

bool CameraUtils::ConvertPointVCS2IMG(const cv::Point3f &p3fVcs, cv::Point3f &p3fImg) {

Eigen::Matrix<float, 3, 1> point\_vcs;

Eigen::Matrix<float, 3, 1> point\_camera;

point\_vcs << p3fVcs.x, p3fVcs.y, p3fVcs.z;

point\_camera = extr\_R\_ \* (extr\_R\_init\_ \* point\_vcs + extr\_t\_);

if (point\_camera[2] <= 0) {

return false;

}

float x = point\_camera[0] / point\_camera[2];

float y = point\_camera[1] / point\_camera[2];

float depth = point\_camera[2];

p3fImg = cv::Point3f(x, y, depth);

PixelDenormalize(p3fImg);

if (p3fImg.x >= 0 && p3fImg.x < camera\_width\_ && p3fImg.y >= 0 && p3fImg.y < camera\_height\_) {

return true;

}

return false;

}