# 一、MATLAB

## 插值

|  |
| --- |
| data=load(‘C: \Users\17901\Desktop\data.txt’);  data(:,1) = interp1(1:1004, data\_gt(:,1), linspace(1, 1004, 5082), 'spline');  %此处举例将1004行的原数据插值扩充为5082行数据  dlmwrite('C:\Users\17901\Desktop\new\_data.txt', data, 'delimiter', '\t', 'precision', 13);  %保存文件，设置分隔符及精度 |

## 读取csv保持精度

|  |
| --- |
| 使用excel打开csv文件，设置单元格格式，设置为数值，并将小数点设置为足够的位数。 |

# 二、Python

## 批量修改文件名

|  |
| --- |
| import os  path\_src =’home/xxx/xx’  filename=oslistdir(path\_src)  filename.sort(key=lamda x:int(x[1:4]))  for file in filename  new\_name=xxx.png  os.rename(os.path.join(path\_src,file),os.path.join(path\_src,new\_name)) |

读取文件顺序可能会发生变化，所以重命名前需要对文件名进行排序。

## Pycharm使用不同的python环境

|  |
| --- |
| 在终端中输入：activate <环境名>  然后执行：python <xxx.py> |

## python工程找不到自定义包

|  |
| --- |
| 1、Windows下：  在程序开头添加：  import sys  sys.path.append("C:\\Users\\LEGION\\Desktop\\gcn-parking-slot-main")  #将工程目录加入到系统路径中，注意自定义包一定要有\_\_init\_\_.py文件（可以是空的）  包结构  2、Ubuntu下：  sudo gedit ~/.bashrc  最后一行加入以下自定义库的路径：  export PYTHONPATH=$PYTHONPATH:/home/xxx/xx  source ~/.bashrc  3、使用pycharm：  右键自定义库目录；  Mark Directory as;  Sources root; |

## 中文注释报错

|  |
| --- |
| 文件开头加  #coding=utf-8  或者  #coding=gbk |

## python3与melodic自带cvBridge冲突

|  |
| --- |
| 1、编译python3环境的cvbridge  mkdir cvBridge\_ws  mkldir src  git clone https://github.com/ros-perception/vision\_opencv.git src/vision\_opencv  apt-cache show ros-melodic-cv-bridge | grep Version  Version: 1.13.1-1xenial-20180416-143935-0800 #版本不一定一致  cd src/vision\_opencv/  git checkout 1.13.1  cd …/…/  catkin config -DPYTHON\_EXECUTABLE=/usr/bin/python3  -DPYTHON\_INCLUDE\_DIR=/usr/include/python3.5m  -DPYTHON\_LIBRARY=/usr/lib/x86\_64-linux-gnu/libpython3.5m.so  #寻找该文件对应位置，可使用locate命令  catkin config --install  catkin build  2、在需要执行python3的终端下添加source  source {dir}/install/setup.bash –extend |

## 修改bag中话题的时间戳

|  |
| --- |
| in\_bag = '/home/nrc/130/bevlidar2\_2024-01-30-15-29-21.bag'  out\_bag = '/home/nrc/130/output2.bag'  input\_bag = rosbag.Bag(in\_bag, "r")  output\_bag = rosbag.Bag(out\_bag, "w")  first\_flag\_odo = True  for topic, msg, t in input\_bag.read\_messages():  if first\_flag\_odo and topic =='/Odometry':  t\_first\_odo = msg.header.stamp.to\_sec()  msg.header.stamp = Time.from\_sec(1)  first\_flag\_odo = False  output\_bag.write(topic, msg, t)  elif topic =='/Odometry':  msg.header.stamp = Time.from\_sec(msg.header.stamp.to\_sec() - t\_first\_odo)  output\_bag.write(topic, msg, t)  output\_bag.close()  注意修改的是msg.header.stamp，而不是for循环语句中的t。  最后注意要关闭输出bag，否则需要reindex。 |

# C++

## cout相关

### (1). 设置cout精度

|  |
| --- |
| std：：cout<<setpresion(10)<<a<<std::endl; |

### (2). 原始字符串

|  |
| --- |
| std：：cout<<R“(xxx \ “”\n )”<<std::endl;  R做前缀，“(和)”做定界符，则可在字符串中直接使用反斜杠、双引号等  如果字符串中存在)”，编译器会认为字符串提前结束，导致错误。  需要使用R“+\*(和)+\*”做开头、结尾  std：：cout<<R“+\*(xxx)+\*”<<std::endl; |

### (3). cout布尔值

|  |
| --- |
| #include <iostream>  bool flag=false;  std::cout<<std::boolalpha<<flag<<std::endl; |

## 引用第三方库

|  |
| --- |
| 1、由头文件和源文件组成  .cpp文件中引用.h文件  #include “xxx.h”  用双引号，不在同一级目录下时要写对路径  CmakeListx.txt中：  add\_library(my\_lib xxx.cpp)  target\_link\_libraries(my\_node my\_lib)  2、仅由头文件组成  CmakeListx.txt中：  include\_directories( xxx.hpp) |

## C++数据类型极值

|  |
| --- |
| #include <climits>  //某类型最大、最小值  numeric\_limits<T>::max()  numeric\_limits<T>::min()  //如需修改，重置宏定义，尝试在用户cpp文件中宏定义INT\_MAX,但是numeric\_limits<T>::max()仍未改变，一般情况下不要修改类型极值 |

## 常量

|  |
| --- |
| //c++中一般常用const设置常量，而不使用宏定义  const int a=1；  //跨cpp文件使用时，需要extern  extern const int a； |

## 外部变量

|  |
| --- |
| xxxClass.cpp中声明变量  extern int x=1;  int y;  int z=10;//也可声明并定义  main.cpp使用外部变量  extern int x；  extern int y；  extern int z； |

## 单双引号

|  |
| --- |
| “ ”用于字符串常量，隐含‘/0’  ‘ ’用于字符常量 |

## 指针声明

|  |
| --- |
| int \*a,b;  //a是指针，b是整型  //定义指针时要明确指向，避免以下情况出现：  int \*a;  int b;  \*a=1;  \*a=b;  //指针地址不明确，可能修改程序内存，导致难以发现的错误  //实际操作时程序发生段错误  //可利用关键字new分配内存,并使用delete释放内存  int \*a = new int;  int \*b = new int [10];  \*a=1;  delete a;  delet [] b;  //直接定义一个变量，其存储在栈空间中，而new方法定义的变量将存储在堆空间中  //栈由操作系统自动分配释放 ，用于存放函数的参数值、局部变量等，其操作方式类//似于数据结构中的栈。  //堆由开发人员分配和释放， 若开发人员不释放，程序结束时由 OS 回收，分配方  //式类似于链表。 |

## 打印变量类型

|  |
| --- |
| //以整型为例  int a;  typeid(a).name() |

## 从yaml文件读取参数

|  |
| --- |
| 一、填写yaml文件  注意开头加上%yaml:1.0  数值类型：  IMU: 1  字符串类型：  path: “xxx”  二、函数读取参数  void get\_parameters(std::string config\_file){  cv::FileStorage fsSettings(config\_file,cv::FileStorage::READ);  if(!fsSettings.isOpen())  {  ROS\_WARN(“WRONG PATH!\n”)  }else{  int xxx=fsSettings[“xxx”];  //数值类型  std::string yyy;  fsSettings[“yyy”]>>yyy;  //字符串类型  //也可以写成std::string xxx=(std::string)fsSettings[“yyy”];  fsSettings.release();  }  }  //fsSetting()>>传回相当于地址,数值类型如果想使用该方法，如下:  //int \*xxx =new int;//不使用new关键字会发生内存错误  // fsSettings[“xxx”]>>\*xxx;//实测可以正常读取参数 |

## 同名函数参数不同

|  |
| --- |
| 一、声明时重写函数，如：  para(int a);  para();  二、声明函数时默认赋值(vins\_fusion中的Estimator::inputImage()函数)  //声明时：  void Estimator::inputImage(double t,const cv::Mat &\_img, const cv::Mat &\_img1=cv::Mat());  //在函数内部判断\_img1.empty()是否为真,可知传入了几个参数  //两种调用方式：  inputImage(t,img0);  inputImage(t,img0,img1); |

## 指针和引用

|  |
| --- |
| //指针：  int \*a=new int;  a存放地址，\*a为该地址存储的数据 |
| //引用：  int b;  int &c=b; |
| 1、定义和性质不同  指针是一个变量，存储的地址，指向内存单元  引用是变量的别名，跟原始变量是同一块内存  int a = 10;  int \*p = &a; // p是指针, &在此是取地址运算  int &r = a; // r是引用, &在此起标识作用  2、指针可以不初始化，引用必须初始化  int\* ptr; //正确  int& ref; //error:引用变量"ref"需要初始值设定项  3、指针可以有多级，引用只能是一级，&&表示右值引用  int \*\*p; // 合法  int &&a; //右值引用  4、指针可以指向NULL，引用不可以为NULL  int \*p = NULL; // 合法  int &r = NULL; // 不合法  5、指针初始化之后可以再改变指向，引用不可以  int a = 996;  int \*p = &a; // 初始化, p 是 a 的地址  int &r = a; // 初始化, r 是 a 的引用  int b = 885;  p = &b; // 合法, p 更改为 b 的地址  r = b; // 不合法, r 不可以再变  6、sizeof 的运算结果不同  int a = 996;  int \*p = &a;  int &r = a;  cout << sizeof(p); // 返回 int\* 类型的大小  cout << sizeof(r); // 返回 int 类型的大小  7、自增运算意义不同  int a = 996;  int \*p = &a;  int &r = a;  p++;  r++;  8、指针和引用作为函数参数时，指针需要检查是否为空，引用不需要  void fun\_p(int \*p)  {  // 需要检查P是否为空  if (p == NULL)  {  // do something  }  }  void fun\_r(int &r)  {  // 不需要检查r  // do something  } |
| 函数接口区分取地址和引用  void funcA(int& a,int& b)//这是引用  {  //TODO  }    void funcB(int\* input)//这是要用取地址  {  //TODO  }    int main()  {  int a = 1;  int b = 2;  funcA(a, b);//引用，调用函数的时候不需要取地址符    int c = 3;  funcB(&c);//取地址，调用函数的时候，因为函数要求输入为指针，因此要取地址  return 1;  } |
| 1、移动语义并不总是比拷贝语义更快。在某些情况下，拷贝语义可能比移动语义更快，例如拷贝的对象较小或者拷贝的对象类型不支持移动语义。  2、右值引用并不是所有场景都适用。例如在函数返回值的情况下，通常情况下最好使用值传递而非右值引用。  3、右值引用会导致对象的生命周期缩短，所以需要格外小心，确保在使用右值引用时对象的生命周期仍然正确，避免出现悬空指针等问题。  4、右值引用有时会导致代码的可读性降低，所以需要根据具体情况权衡使用。对于简单的场景，可能不需要使用右值引用，而对于复杂的场景，使用右值引用可能会提高代码的可读性和维护性。 |

## 智能指针

|  |
| --- |
| 智能指针分哪几种？std::unique\_ptr, std::shared\_ptr, std::weak\_ptr各有何用途？  C++11标准中就这三种。  ( 1 ) unique\_ptr:独占语义。在任何时间点，资源只能唯一地被一个unique\_ptr占有。当unique\_ptr离开作用域，所包含的资源被释放。如果资源被其它资源重写了，之前拥有的资源将被释放。所以它保证了他所关联的资源总是能被释放。  ( 2 ) shared\_ptr:共享占有语义。shared\_ptr的目标非常简单:多个共享指针可以同时指向—个对象，当最后一个shared\_ptr离开作用域时，内存才会自动释放。  ( 3 ) weak \_ptr:共享非占有语义。通常与shared\_ptr配对使用，其弱引用计数可以解决shared\_ptr循环引用的问题。 |

## std::cerr重定向调试

|  |
| --- |
| 在printf标准输出流占用等情况需要调试时，可以将cerr重定向到文本文件，打印过程参数，便于调试。  #include <iostream>  #include <fstream>  int main() {  std::ofstream outfile("error.txt");  if (!outfile.is\_open()) {  std::cout << "无法打开文件" << std::endl;  return 1;  }  std::streambuf\* originalBuffer = std::cerr.rdbuf();  std::cerr.rdbuf(outfile.rdbuf());  std::cerr << "这是一个错误信息" << std::endl;  std::cerr.rdbuf(originalBuffer);  outfile.close();  return 0;  } |

## 非连通区域提取ORB特征点

|  |
| --- |
| cv::Ptr<cv::ORB> orb= cv::ORB::create();  //创建提取器  cv::Ptr<cv::DescriptorMatcher> matcher = cv::DescriptorMatcher::Create(“BruteForce-Hamming”);  //匹配方式  cv::Mat mask=cv::Mat::zero(width,height,CV\_8UC1);  //创建与待提取图像相同大小的蒙版  cv::Rect roi\_1(start\_x1,start\_y1,w\_1,h\_1);  //设置第一个ROI  cv::Rect roi\_2(start\_x2,start\_y2,w\_2,h\_2);  //设置第二个ROI  mask(roi\_1)=255;  mask(roi\_2)=255;  //将蒙版上ROI区域像素值设置为255  cv::Mat des;  //定义描述子存储变量  std::vector<cv::KeyPoint> kps;  //定义特征点存储变量  orb->detectAndCompute(image,mask,kps,des);  //在蒙版为零的区域提取特征点并计算描述子 |

## 对象成员

|  |
| --- |
| 在定义类A时，想要实例化另外一个类B的对象b作为A的属性。而类B构造函数需要参数。  A.h中：  class A{  A();  B b；  };  A.cpp中：  A::A():b(“/home/nrc/xx”){  } |

## 编译选项

|  |
| --- |
| 编译C++代码时常用选项：  set(CMAKE\_CXX\_STANDARD 11)  set(CMAKE\_CXX\_STANDARD\_REQUIRED ON)  设置编译选择的C++标准版本。  set(CMAKE\_BUILD\_TYPE Release)  这将设置构建类型为"Release"，通常表示编译器会启用优化选项。CMake支持多种构建类型，如Debug、Release、RelWithDebInfo和MinSizeRel。  -Wall  表示开启所有警告信息。具体来说，编译器会生成所有可能的警告信息，包括一些可能会被忽略的警告信息。开启-Wall选项可以帮助开发者发现潜在的问题，提高代码的质量和可靠性。但是，由于-Wall会生成大量的警告信息，有时候会影响开发效率，因此需要根据具体情况来决定是否开启该选项。  -O3  是GCC编译器的优化级别选项之一，用于指定编译器生成高度优化的代码。-O3选项表示进行最高级别的优化，编译器将尽可能地对代码进行各种优化，以提高代码的执行效率和性能。这些优化包括但不限于内联函数展开、循环展开、常量传播、循环变量优化、向量化等。使用-O3选项的优点是，它可以显著提高代码的执行速度和性能。编译器会根据代码的结构和特性进行深度优化，以减少指令级别的开销、提高并行性和利用处理器的特殊功能。  -g  这是一个调试选项，它指示编译器生成调试信息。这允许开发者使用调试器来调试程序。 |

## 多线程

|  |
| --- |
| 传统的C++（C++11之前）中并没有引入线程这个概念，在C++11出来之前，如果我们想要在C++中实现多线程，需要借助操作系统平台提供的API，比如Linux的<pthread.h>，或者windows下的<windows.h> 。  C++11提供了语言层面上的多线程，包含在头文件<thread>中。它解决了跨平台的问题，提供了管理线程、保护共享数据、线程间同步操作、原子操作等类。C++11 新标准中引入了5个头文件来支持多线程编程，如下图所示：    一个进程至少有一个线程，线程是进程的一部分。线程是轻量级的进程。 |

## 多线程安全读写相同变量

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1. mutex互斥锁   mutex（互斥锁）是一种同步原语，通过在访问共享资源之前获取锁，以确保一次只有一个线程可以访问该资源。当一个线程获取到锁时，其他线程将被阻塞，直到该线程释放锁。mutex适用于需要对共享资源进行复杂操作的情况，因为它可以提供更细粒度的控制，但需要开发者显式地加锁和解锁。   1. atomic原子类型   atomic（原子类型）是一种特殊的数据类型，它提供了原子操作的功能，可以确保对共享变量的读写操作是不可分割的。这意味着在任何给定时刻，只有一个线程可以对该变量进行读写操作，而其他线程需要等待。atomic适用于对简单数据类型（如整数、指针等）进行原子操作的情况，因为它提供了更高效的方式来保证线程安全，避免了显式的加锁和解锁。   |  |  | | --- | --- | | 成员函数 | 描述 | | store() | 存储一个值到原子对象，等价于使用等号。 | | load() | 加载原子对象中存入的值，等价与直接使用原子变量 | | exchange() | 返回原来里面存储的值，然后this会再存储新的值，相当于将上面两个load()和store()合成起来 | | compare\_exchange\_weak( T&expected, T desired,....） | 交换-比较操作是比较原子变量值和所提供的期望值，如果二者相等，则this存储提供的期望值desired，如果不等则将期望值expected更新为原子变量的实际值，更新成功则返回true反之则返回false | | compare\_exchange\_strong (T& expected, T desired, ......) | 如果当前的变量this的值等于expected值时，则将this值改为desired，并返回true；否则，不对this的值进行修改，expected被赋值为this的值，并返回false. | |

## C和C++的混合编程

|  |
| --- |
| extern”C”{  #include “src/lsd.h”  }; |

## 内联函数的声明与定义

|  |
| --- |
| 报错示例：  xx.h中声明函数  int getNum();  xx.cpp定义函数  int getNum(){  return num;  }  main.cpp使用函数  报错未定义的引用  正确示例：  xx.h中声明并定义  int getNum(){  return num;  }  main.cpp使用  原因：  内联函数在编译时展开，编译器需要在编译时就看到内联函数的定义，因此需要在头文件中同时提供声明和定义。 |

## 记录代码耗时

|  |
| --- |
| #include <iostream>  #include <chrono>  int main() {  // 记录开始时间点  auto start = std::chrono::high\_resolution\_clock::now();  // 这里是你想要计时的指令  // 例如：std::cout << "Hello, World!" << std::endl;  // 记录结束时间点  auto end = std::chrono::high\_resolution\_clock::now();  // 计算时间间隔  auto duration = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);  // 输出执行时间  std::cout << "指令执行时间：" << duration.count() << " 微秒" << std::endl;  return 0;  } |

# 四、Linux/ros

## 一键安装ROS

|  |
| --- |
| wget http://fishros.com/install -O fishros && . fishros |

## bag包导出图片

|  |
| --- |
| 安装图片处理依赖的包：  sudo rosrun apt-get install mjpegtools  sudo rosrun apt-get install ffmpeg |
| rosrun image\_view extract\_images \_sec\_per\_frame:=0.01 image:=<IMAGETOPIC>  播放bag开始导出图片  注：导出图片少于包内实际含有的图片信息时，可能是因为播放速度太快。可使用  rosbag play -r 0.2 xxx.bag调节播放速度 |

## rosbag操作

### (1). 查看数据包信息

|  |
| --- |
| rosbag info xx.bag // 查看数据包信息 |

### (2). 数据播包

|  |
| --- |
| rosbag play xx.bag // 播放xx数据包  rosbag play \*.bag // 播放目录下所有数据包  rosbag play bag1 bag2 // 播放多个bag包  rosbag play --bags=bag1 bag2 // 同上，播放多个bag包  rosbag play –s 10 –r 0.7 –l \*.bag // -s指定从几秒开始，-r指定播放速度，-l循环播放  rosbag play –s 10 –r 0.7 -u 60 \*.bag // 从10s开始0.7倍速播包，持续播60s  rosbag play --pause \*.bag // 以暂停方式启动  // 等播放到topic\_1、topic\_2相关话题的时候暂停播放，等待命令行输入空格时继续播放  rosbag play xxx.bag --pause-topics /topic\_1 /topic\_2  rosparam set /use\_sim\_time time // 设置使用仿真时间，播包或者录包时间戳对不上时  rosbag play xx.bag --clock // 系统以msg对应的仿真时间运行，而不是wall clock time（UTC）；避免在做msg的tf转换时把更早时间的msg丢掉  rosbag play xxx.bag --topic /topic\_name1 /topic\_name2 // 只播放特定topic1/2  rosbag play xxx.bag --topics /topic\_1 /topic\_2 // 同上 |

### (3). 节点查看

|  |
| --- |
| rosnode list // 获得正在运行的节点列表  rosnode info /node-name // 获得特定节点的信息  rosnode ping node-name // 测试节点是否连通  rosnode kill node-name // 终止节点 |

### (4). 节点运行

|  |
| --- |
| roscore // 启动master节点；启动ros  // 起单个节点  rosrun 功能包名称 节点名称 // 执行功能包中的一个节点  // 起多个节点  roslaunch 功能包名称 launch文件 // 执行功能包中一个或多个节点或设置执行选项  // 用来启动定义在launch文件中的多个节点,采用xml格式对要运行的节点进行描述  rospack find package\_name // 查找package  rospack depends package\_name // 查看package依赖  rospack depends1 package\_name // 查看package的直接依赖 |

### (5). 查看topic

|  |
| --- |
| rostopic –h // 查看帮助  rostopic list // 列出所有topic  rostopic info /topic // 查看topic由谁发布和订阅  rostopic type topic\_name // 查看topic的message类型  rostopic hz topic\_name // 查看topic的频率  rostopic echo topic\_name // 显示topic上发布的message （python版本有要求）  // rostopic echo topic\_name命令要在节点运行的情况下才能查看实时的消息输出  修改topic名字  rosbag play xxx.bag oldTopicName:=newTopicName  同时录包 rosbag record -o bagName.bag /newTopicName /otherTopicName  保存某个topic内容  rostopic echo -b xxx.bag -p /topic\_name > xxx.csv  rostopic echo -b xxx.bag -p /topic\_name > xxx.txt  /\* csv格式  目标1消息字段... 目标2消息字段... 目标3消息字段...  目标1 time1 目标2 time1 目标3 time1  目标1 time2 目标2 time2 目标3 time2  ... ... ...  \*/ |

### (6). 查看msg

|  |
| --- |
| rosmsg –h // 查看帮助  rosmsg list // 查看所有消息  rosmsg show msg // 查看msg中有哪些字段 |

### (7). 日志清理

|  |
| --- |
| rosclean check // 检查日志文件  rosclean purge // 清理日志文件 |

### (8). 参数设置和加载

|  |
| --- |
| rosparam list // 当前使用到的参数  rosparam get / // 获取参数列表中所有参数的值  rosparam set param\_name true // 设置参数取值为true  rosparam set param\_name 150 // 设置参数取值为150  rosparam load param\_file.yaml // 加载文件中的所有参数  rosparam dump params.yaml // 将所有参数写到params.yaml文件  rosparam load params.yaml copy // 参数从文件中导出  rosparam get copy/param\_name // 获取参数取值 |

### (9). 录包

|  |
| --- |
| // cd到目标文件夹下  rosbag record –a // 录制所有topic的数据包,不光包括正在播的bag里有的  rosbag record /topic\_a /topic\_2 // 录制topic\_1/topic\_2的数据包  rosbag record -O bag\_name.bag /topic1 // 录制topic\_1，指定生成数据包的名字，否则都是 日期.bag  rosbag record -o bag\_name.bag /topic1 // 录制topic\_1，指定生成数据包的名字，格式会自动变成 bag\_name\_时间.bag  rosbag record -a -o bag\_name.bag // 录制所有topic  // 录包时间没对齐的话，需要设置rosparam set /use\_sim\_time true  // 然后播包时加 --clock  rosbag record --duration=30 /topic // 录制topic，录制时长30s；还可设置为--duration=5m 或 --duration=2h  rosbag record --split --duration=30 /topic\_1 /topic\_2 // 录制topic\_1和2，录制持续30s后分文件存储  rosbag record --split --duration=5m /topic\_1 /topic\_2 // 录制topic\_1和2，录制持续5m后分文件存储  rosbag record --split --duration=2h /topic\_1 /topic\_2 // 录制topic\_1和2，录制持续2h后分文件存储  rosbag record --split --size=1024 /topic // 录制topic，录制大小1024M后分文件存储  rosbag record -l 1000 /topic // 录制topic满1000个消息 |

### (10). 切割bag

|  |
| --- |
| 根据topic进行切割/过滤  // 根据topic过滤/切割  rosbag filter my.bag only-tf.bag "topic == '\tf'"  // 根据多个topics进行过滤/切割  rosbag filter my.bag output.bag "topic == '/velodyne\_point\_cloud' or topic =='/visensor/imu'"  根据时间段进行切割/过滤  // 根据时间过滤/切割  rosbag filter my.bag output.bag "t.to\_sec() <= 1659100376.473893"  // 根据某个时间段过滤/切割  rosbag filter my.bag output.bag "t.to\_sec() >= 1659100335.039506 and t.to\_sec() <= 1659100376.473893"    切割/过滤某个时间段内的多个topics  // 根据topic和时间过滤/切割  rosbag filter my.bag output.bag "topic == '/odometry/gps' and t.to\_sec() <= 1659100376.473893"  // 过滤/切割某个时间段内的多个topics  rosbag filter my.bag output.bag "(topic == '/velodyne\_point\_cloud' or topic =='/visensor/imu')  and (t.to\_sec() >= 1659100335.039506 and t.to\_sec() <= 1659100376.473893)" |

### (11). bag话题转txt

|  |
| --- |
| rostopic echo -b bagname.bag /topic/name > xxx.txt |

### (12). 修复意外中断的包

|  |
| --- |
| rosbag reindex 2024-03-19-15-20-36.bag.active  rosbag fix 2024-03-19-15-20-36.bag.active result.bag |

## 查看话题信息

|  |
| --- |
| rostopic echo /image\_raw --noarr  可以查看话题信息，在查看图像等数据量大的话题时，data一栏显示内容过多，可以--noarr参数使其隐藏，方便查看图像分辨率等信息。 |

## 自定义话题类型

|  |
| --- |
|  |

## rosdep update失败

|  |
| --- |
| sudo pip install rosdepc  sudo rosdepc init  rosdepc update  若没安装pip 执行sudo apt install python-pip |

## 减小串口延迟

|  |
| --- |
| sudo apt install setserial  setserial /dev/ttyUSB0 low\_latency  注意需要每次开启电脑时都进行一次修改，可写入程序开启脚本中 |

## DBOW3链接

以slam14代码为例，使用DBoW3词袋模型时，安装DBoW3库存在编译不通过的情况，需要更改CMakeLists.txt

|  |
| --- |
| set（DBoW3\_LIBS “/usr/local/lib/libDBoW3.a”）  更改为：  set（DBoW3\_LIBS “/usr/local/lib/libDBoW3.so”）  注：.a 为静态链接库；.so 为动态链接库。 |

## Cmake

|  |
| --- |
| CMake是一个跨平台项目构建工具，帮助用户生成Makefile |

### CMakeLists.txt结构：

|  |
| --- |
| 指定最小cmake版本  cmake\_minimum\_required(VERSION X.XX)  为项目命名  project(PROJECT\_NAME)  以OpenCV为例，添加外部软件包，先找到对应路径  find\_package(OpenCV REQUIRED)  包含头文件路径  include\_directories(${OpenCV\_INCLUDE\_DIRS})  添加库文件  add\_library(LIBRARY\_NAME XXX.CPP)  添加可执行文件  add\_executable(E\_FILE\_NAME XXX.CPP)  链接库  target\_link\_libraries( E\_FILE\_NAME ${OpenCV\_LIBS}) |

### find\_package

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| find\_package(OpenCV REQUIRED)  REQUIRED关键字将该目标库标记为必须，未找到对应包时将报错。  find\_package(OpenCV COMPONENTS XXX)  COMPONENTS关键字用来导入该软件包中的部分库。  find\_package(PackageName)命令寻找第三方软件包的.cmake文件，有两种模式：  Module 模式  这种模式下，find\_package 需要解析 Findxxx.cmake 文件，这也是 find\_package 默认工作的模式  Config 模式  这种模式下，find\_package 需要解析 xxxConfig.cmake 文件或者 xxx-config.cmake 文件。实际上只有 Findxxx.cmake 文件无法找到，即 Module 模式执行不成功，才会进入 Config 模式。  有三个相关变量：   |  |  | | --- | --- | | PackageName\_FOUND | 如果找到对应的.cmake则为true | | PackageName\_INCLUDE\_DIR | 头文件目录 | | PackageName\_LIBRARY | 库文件 |   list(APPEND CMAKE\_MODULE\_PATH "/path/to/your/cmake/modules")  find\_package(PackageName [PATHS Path1])  include()  用于包含本地的或CMake标准中提供的\*.cmake文件。  Module 模式下需要查找到名为 Find<PackageName>.cmake 的配置文件。该文件的查找过程只涉及到两个路径：CMAKE\_MODULE\_PATH 和 CMake 安装路径下的 Modules 目录。即搜索路径依次为：  先在 CMAKE\_MODULE\_PATH 变量对应的路径中查找。如果路径为空，或者路径中查找失败，则在 CMake 安装目录（即 CMAKE\_ROOT 变量）下的 Modules 目录中查找。这个目录通常为 /usr/share/cmake-x.xx/Modules。这两个变量可以在CMakeLists.txt文件中打印查看具体内容：  其中 CMAKE\_MODULE\_PATH 默认为空，可以利用 set 命令对该变量进行设置。  与 Module 模式不同，Config 模式需要查找的路径很多，也要匹配很多的可能性，因此有些路径是首先作为根目录，然后进行子目录的匹配，而有些则是直接指定到具体的 XXXConfig.cmake 文件。具体将会按照如下顺序进行查找：  1、<PackageName>\_DIR  2、CMAKE\_PREFIX\_PATH、CMAKE\_FRAMEWORK\_PATH、  CMAKE\_APPBUNDLE\_PATH  3、PATH环境变量路径  4、CMAKE\_SYSTEM\_PREFIX\_PATH等系统变量路径  <PackageName>\_DIR  首先搜索名为<PackageName>\_DIR的 CMake 变量或环境变量路径，这个变量默认为空。这个路径需要直接指定到<PackageName>Config.cmake或<lower-case-package-name>-config.cmake文件所在目录才能找到，可参考以下示例：  CMAKE\_PREFIX\_PATH 等变量  如果按照<PackageName>\_DIR搜索不到相应的XXXConfig.cmake文件的话，则会查找名为CMAKE\_PREFIX\_PATH、CMAKE\_FRAMEWORK\_PATH、CMAKE\_APPBUNDLE\_PATH的 CMake 变量或环境变量路径。这些变量指定的路径将作为查找时的根目录，它们默认都为空，使用message(STATUS ${CMAKE\_PREFIX\_PATH})打印时将没有输出。  PATH 环境变量路径  若还找不到，则是搜索 PATH 环境变量路径，这个路径也是根目录，默认为系统环境 PATH 环境变量值。  其实这个路径才是 Config 模式大部分情况下能够查找到安装到系统中各种库的原因。这个路径的查找规则为：遍历 PATH 环境变量中的各路径，如果该路径如果以 bin 或 sbin 结尾，则自动回退到上一级目录得到根目录。在终端键入以下内容以查看系统环境变量路径：echo $PATH  CMAKE\_SYSTEM\_PREFIX\_PATH 等变量  我们在安装 OpenCV 的时候，可以在cmake-gui中修改CMAKE\_INSTALL\_PREFIX变量的内容，把 OpenCV 安装在/opt/opencv下，我们在使用find\_package时也不需要指名路径，说明除了上面的一些变量之外，还存在另外一系列路径。这里列出最重要的一个变量：CMAKE\_SYSTEM\_PREFIX\_PATH。 |

### 链接指定路径下的包

|  |
| --- |
| 1. 设置DIR  set(Torch\_DIR /home/libtorch)  find\_package(Torch REQUIRED)  2. 设置PATHS  find\_package(Protobuf REQUIRED  PATHS /home/libtorch  NO\_DEFAULT\_PATH)  3. 指定  set(CMAKE\_PREFIX\_PATH /home/libtorch) |

### 注释

|  |
| --- |
| CMake 使用 # 进行行注释，可以放在任何位置。  CMake 使用 #[[ ]] 形式进行块注释。 |

### 打印信息

|  |
| --- |
| MESSAGE(STATUS “This is ” ${NAME\_SOURCE\_DIR})  STATUS ：非重要消息  WARNING：CMake 警告, 会继续执行  AUTHOR\_WARNING：CMake 警告 (dev), 会继续执行  SEND\_ERROR：CMake 错误, 继续执行，但是会跳过生成的步骤  FATAL\_ERROR：CMake 错误, 终止所有处理过程 |

### 设置变量

|  |
| --- |
| set(SRC\_LIST add.c;div.c;main.c;mult.c;sub.c)  add\_executable(app ${SRC\_LIST})  用于定义一个变量，存储多个文件名对应的字符串。 |

### 设置C++标准

|  |
| --- |
| 方式一：  #增加-std=c++11  set(CMAKE\_CXX\_STANDARD 11)  方式二：  #增加-std=c++11  cmake CMakeLists.txt文件路径 -DCMAKE\_CXX\_STANDARD=11  在执行cmake时可以修改宏的值 |

### 搜索文件

|  |
| --- |
| 方法一：使用aux\_source\_directory  cmake\_minimum\_required(VERSION 3.0)  project(CALC)  include\_directories(${PROJECT\_SOURCE\_DIR}/include)  # 搜索 src 目录下的源文件  aux\_source\_directory(${CMAKE\_CURRENT\_SOURCE\_DIR}/src SRC\_LIST)  add\_executable(app ${SRC\_LIST})  方法二：使用file  file(GLOB/GLOB\_RECURSE 变量名 要搜索的文件路径和文件类型)  GLOB: 将指定目录下搜索到的满足条件的所有文件名生成一个列表，并将其存储到变量中。  GLOB\_RECURSE：递归搜索指定目录，将搜索到的满足条件的文件名生成一个列表，并将其存储到变量中。  file(GLOB MAIN\_SRC ${CMAKE\_CURRENT\_SOURCE\_DIR}/src/\*.cpp)  file(GLOB MAIN\_HEAD ${CMAKE\_CURRENT\_SOURCE\_DIR}/include/\*.h)  CMAKE\_CURRENT\_SOURCE\_DIR 宏表示当前访问的 CMakeLists.txt 文件所在的路径。  关于要搜索的文件路径和类型可加双引号，也可不加:  file(GLOB MAIN\_HEAD "${CMAKE\_CURRENT\_SOURCE\_DIR}/src/\*.h") |

### 添加头文件

|  |
| --- |
| cmake\_minimum\_required(VERSION 2.0)  project(CALC)  set(CMAKE\_CXX\_STANDARD 11)  set(HOME /home/robin/Linux/calc)  set(EXECUTABLE\_OUTPUT\_PATH ${HOME}/bin/)  include\_directories(${PROJECT\_SOURCE\_DIR}/include)  file(GLOB SRC\_LIST ${CMAKE\_CURRENT\_SOURCE\_DIR}/src/\*.cpp)  add\_executable(app ${SRC\_LIST}) |

### 指定输出路径

|  |
| --- |
| 根目录下的CMakeLists.txt:  cmake\_minimum\_required(VERSION 2.8)  project(HelloSLAM)  add\_subdirectory(src bin)  Src目录下的CMakeLists.txt:  正常编写，add\_executable( )等等  在build下经过cmake .. 和 make -j，src路径下编译得到的中间二进制和目标二进制存放在build/bin路径下。  EXECUTABLE\_OUTPUT\_PATH宏是可执行文件输出路径。  LIBRARY\_OUTPUT\_PATH宏是库文件输出路径，这个对应静态库文件和动态库文件都适用。 |

### 制作库文件

|  |
| --- |
| (1).a静态库  add\_library(库名称 STATIC 源文件1 [源文件2] ...)  在Linux中，静态库名字分为三部分：lib+库名字+.a，此处只需要指定出库的名字就可以了，另外两部分在生成该文件的时候会自动填充。  cmake\_minimum\_required(VERSION 3.0)  project(CALC)  include\_directories(${PROJECT\_SOURCE\_DIR}/include)  file(GLOB SRC\_LIST "${CMAKE\_CURRENT\_SOURCE\_DIR}/src/\*.cpp")  add\_library(calc STATIC ${SRC\_LIST})  这样最终会生成静态库libcalc.a。  (2).so动态库  add\_library(库名称 SHARED 源文件1 [源文件2] ...)  在Linux中，动态库名字分为三部分：lib+库名字+.so，此处只需要指定出库的名字就可以了，另外两部分在生成该文件的时候会自动填充。  cmake\_minimum\_required(VERSION 3.0)  project(CALC)  include\_directories(${PROJECT\_SOURCE\_DIR}/include)  file(GLOB SRC\_LIST "${CMAKE\_CURRENT\_SOURCE\_DIR}/src/\*.cpp")  add\_library(calc SHARED ${SRC\_LIST})  这样最终会生成静态库libcalc.so。  。 |

### 链接库文件

|  |
| --- |
| cmake\_minimum\_required(VERSION 3.0)  project(TEST)  file(GLOB SRC\_LIST ${CMAKE\_CURRENT\_SOURCE\_DIR}/\*.cpp)  # 指定源文件或者动态库对应的头文件路径  include\_directories(${PROJECT\_SOURCE\_DIR}/include)  # 指定要链接的动态库的路径  link\_directories(${PROJECT\_SOURCE\_DIR}/lib)  # 添加并生成一个可执行程序  add\_executable(app ${SRC\_LIST})  # 指定要链接的动态库  target\_link\_libraries(app pthread calc)  静态库和动态库均可通过target\_link\_libraries链接。测试时发现，库文件使用到其他库文件时，如opencv，在链接库文件的CMakeLists.txt中也应该链接opencv，否则会出现未定义的引用报错。  顺序问题：  例如进程test依赖a库,b库, a库又依赖b 库，那么顺序如下： target\_link\_libraries (test a b)  也可以通过以下方式而不用考虑库文件的顺序问题  target\_link\_libraries(  x  -Wl,--start-group  libX1.a  libX3.a  libX2.a  -Wl,--end-group  )  注：此处-Wl中W为大写，l为小写。 |

### 变量操作

|  |
| --- |
| (1)追加  有时候项目中的源文件并不一定都在同一个目录中，但是这些源文件最终却需要一起进行编译来生成最终的可执行文件或者库文件。如果我们通过file命令对各个目录下的源文件进行搜索，最后还需要做一个字符串拼接的操作，关于字符串拼接可以使用set命令也可以使用list命令。  使用set拼接  如果使用set进行字符串拼接，对应的命令格式如下：  set(变量名1 ${变量名1} ${变量名2} ...)  关于上面的命令其实就是将从第二个参数开始往后所有的字符串进行拼接，最后将结果存储到第一个参数中，如果第一个参数中原来有数据会对原数据就行覆盖。  cmake\_minimum\_required(VERSION 3.0)  project(TEST)  set(TEMP "hello,world")  file(GLOB SRC\_1 ${PROJECT\_SOURCE\_DIR}/src1/\*.cpp)  file(GLOB SRC\_2 ${PROJECT\_SOURCE\_DIR}/src2/\*.cpp)  # 追加(拼接)  set(SRC\_1 ${SRC\_1} ${SRC\_2} ${TEMP})  message(STATUS "message: ${SRC\_1}")  使用list拼接  如果使用list进行字符串拼接，对应的命令格式如下：  list(APPEND <list> [<element> ...])  list命令的功能比set要强大，字符串拼接只是它的其中一个功能，所以需要在它第一个参数的位置指定出我们要做的操作，APPEND表示进行数据追加，后边的参数和set就一样了。  cmake\_minimum\_required(VERSION 3.0)  project(TEST)  set(TEMP "hello,world")  file(GLOB SRC\_1 ${PROJECT\_SOURCE\_DIR}/src1/\*.cpp)  file(GLOB SRC\_2 ${PROJECT\_SOURCE\_DIR}/src2/\*.cpp)  # 追加(拼接)  list(APPEND SRC\_1 ${SRC\_1} ${SRC\_2} ${TEMP})  message(STATUS "message: ${SRC\_1}")  在CMake中，使用set命令可以创建一个list。一个在list内部是一个由分号;分割的一组字符串。  (2) 字符串移除  我们在通过file搜索某个目录就得到了该目录下所有的源文件，但是其中有些源文件并不是我们所需要的，比如：  $ tree  .  ├── add.cpp  ├── div.cpp  ├── main.cpp  ├── mult.cpp  └── sub.cpp  0 directories, 5 files  在当前这么目录有五个源文件，其中main.cpp是一个测试文件。如果我们想要把计算器相关的源文件生成一个动态库给别人使用，那么只需要add.cpp、div.cp、mult.cpp、sub.cpp这四个源文件就可以了。此时，就需要将main.cpp从搜索到的数据中剔除出去，想要实现这个功能，也可以使用list  list(REMOVE\_ITEM <list> <value> [<value> ...])  通过上面的命令原型可以看到删除和追加数据类似，只不过是第一个参数变成了REMOVE\_ITEM。  cmake\_minimum\_required(VERSION 3.0)  project(TEST)  set(TEMP "hello,world")  file(GLOB SRC\_1 ${PROJECT\_SOURCE\_DIR}/\*.cpp)  # 移除前日志  message(STATUS "message: ${SRC\_1}")  # 移除 main.cpp  list(REMOVE\_ITEM SRC\_1 ${PROJECT\_SOURCE\_DIR}/main.cpp)  # 移除后日志  message(STATUS "message: ${SRC\_1}")  可以看到，在第8行把将要移除的文件的名字指定给list就可以了。但是一定要注意通过 file 命令搜索源文件的时候得到的是文件的绝对路径（在list中每个文件对应的路径都是一个item，并且都是绝对路径），那么在移除的时候也必须将该文件的绝对路径指定，否则移除操作不会成功。  (3)获取 list 的长度。  list(LENGTH <list> <output variable>)  LENGTH：子命令LENGTH用于读取列表长度  <list>：当前操作的列表  <output variable>：新创建的变量，用于存储列表的长度。  (4)读取列表中指定索引的的元素，可以指定多个索引  list(GET <list> <element index> [<element index> ...] <output variable>)  <list>：当前操作的列表  <element index>：列表元素的索引  从0开始编号，索引0的元素为列表中的第一个元素；  索引也可以是负数，-1表示列表的最后一个元素，-2表示列表倒数第二个元素，以此类推  当索引（不管是正还是负）超过列表的长度，运行会报错  <output variable>：新创建的变量，存储指定索引元素的返回结果，也是一个列表。  (5)将列表中的元素用连接符（字符串）连接起来组成一个字符串  list (JOIN <list> <glue> <output variable>)  <list>：当前操作的列表  <glue>：指定的连接符（字符串）  <output variable>：新创建的变量，存储返回的字符串  (6)查找列表是否存在指定的元素，若果未找到，返回-1  list(FIND <list> <value> <output variable>)  <list>：当前操作的列表  <value>：需要在列表中搜索的元素  <output variable>：新创建的变量  如果列表<list>中存在<value>，那么返回<value>在列表中的索引  如果未找到则返回-1。  (7)将元素追加到列表中  list (APPEND <list> [<element> ...])  (8)在list中指定的位置插入若干元素  list(INSERT <list> <element\_index> <element> [<element> ...])  (9)将元素插入到列表的0索引位置  list (PREPEND <list> [<element> ...])  (10)将列表中最后元素移除  list (POP\_BACK <list> [<out-var>...])  (11)将列表中第一个元素移除  list (POP\_FRONT <list> [<out-var>...])  (12)将指定的元素从列表中移除  list (REMOVE\_ITEM <list> <value> [<value> ...])  (13)将指定索引的元素从列表中移除  list (REMOVE\_AT <list> <index> [<index> ...])  (14)移除列表中的重复元素  list (REMOVE\_DUPLICATES <list>)  (15)列表翻转  list(REVERSE <list>)  (16)列表排序  list (SORT <list> [COMPARE <compare>] [CASE <case>] [ORDER <order>])  COMPARE：指定排序方法。有如下几种值可选：  STRING:按照字母顺序进行排序，为默认的排序方法  FILE\_BASENAME：如果是一系列路径名，会使用basename进行排序  NATURAL：使用自然数顺序排序  CASE：指明是否大小写敏感。有如下几种值可选：  SENSITIVE: 按照大小写敏感的方式进行排序，为默认值  INSENSITIVE：按照大小写不敏感方式进行排序  ORDER：指明排序的顺序。有如下几种值可选：  ASCENDING:按照升序排列，为默认值  DESCENDING：按照降序排列 |

## VSCODE调试ROS/CMAKE程序

|  |
| --- |
| roslaunch启动方式调试：   1. catkin\_make编译工程，VSCODE安装ROS插件   可能会出现ROS拓展包无法加载的问题，是因为python等包的版本和vscode的版本不一致，降级拓展包或者升级vscode即可解决。   1. 当前目录下 . /devel/setup.bash 2. code . 使用vscode打开项目文件夹 3. 设置CMakeLists.txt中编译类型为dubug   SET(CMAKE\_BUILD\_TYPE ”Debug”)  如果不设置默认为Release编译，设置断点不起效果，并且无法查看变量值。   1. 设置调试方式   roslaunch选择ROS:launch  修改launch.json文件，填写launch文件的绝对路径至”target”:之后     1. 设置断点并启动   在左侧“变量”、“监视”两栏查看变量  rosrun启动方式调试：  1-4步与roslaunch启动方式相同   1. 设置调试方式   rosrun选择ROS:attach   1. 在终端启动程序并允许访问   rosrun package node arg       1. 设置断点并attach进程   选择需要调试的进程      Cmake编译可执行文件的调试：   1. 安装cmake插件 2. code . 使用vscode打开项目文件夹 3. 设置CMakeLists.txt中编译类型为dubug 4. 设置调试方式   选择Cmake调试程序    注：有时会无法加载扩展包，设置CMake调试程序之前可以先查看vscode左侧扩展列表一次。  选择(gdb)启动    将编译生成的可执行文件的路径填写在program处：     1. 设置断点并逐步调试 |

## Ubuntu装错软件包无法开机

|  |
| --- |
| 开机时选择ubuntu高级设置  选择recovery mode    进入root，命令行语句卸载对应的包（无英伟达显卡的电脑如果装了nvidia-smi在开机时会无法初始化显卡，导致开机卡在检测界面）    或者按下Ctrl+Alt+F2 输入用户名和密码后，以显卡驱动为题为例执行sudo apt-get purge nvidia-\*，然后重启。 |

## 显卡驱动

|  |
| --- |
| 1、nvidia-smi  为了避免装完驱动无法正常开机显示图像界面，在安装驱动时先执行以下命令，避免内核跟随驱动一起升级。  sudo apt-mark hold linux-image-generic linux-headers-generic  之后在系统的软件与更新->附加驱动中安装nvidia驱动，注意此时安装后，安装cuda时应该取消勾选nvidia driver。  如果发生“NVIDIA-SMI has failed because it couldn‘t communicate with the NVIDIA driver”报错。  ls -l /usr/src/查看具体版本    sudo dkms install -m nvidia -v 535.171.04  reboot  nvidia-smi指令正常显示显卡信息  2、cuda、cudnn、pytorch  根据nvidia-smi右上角为支持的最大cuda版本，登录pytorch官网，选择合适的cuda版本，复制命令行。启动conda虚拟环境后执行安装，科学上网能够加快下载速度和提高成功率。 |

## 找不到wifi适配器

|  |
| --- |
| 查看内核版本  sudo uname -a  升级内核  <https://kernel.ubuntu.com/mainline/>  在上述网站下载对应版本的四个安装包    在ubuntu放置这四个安装包的目录下执行 sudo dpkg -i \*.deb  安装完成后重启电脑  如果还是找不到wifi需要安装驱动  <https://github.com/q3aql/drivers-linux-firmware>  下载deb文件，sudo dpkg -i 安装，之后重启电脑  (文件夹中给出drivers-linux-firmware\_20220818-1\_all.deb) |

## Ubuntu蓝牙无法打开

|  |
| --- |
| sudo dmesg |grep -i bluetooth查看蓝牙问题  Failed to load Intel firmware file intel/ibt-0040-0041.sfi(-2)  表示载入该文件失败  /lib/firmware/intel路径下没有对应文件  从<https://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git/tree/intel>找到对应文件下载并复制到/lib/firmware/intel，重启电脑解决问题 |

## ORB\_SLAM3编译通过但是运行发生段错误

|  |
| --- |
| 检查ThirdParty中三方库的CmakeLists.txt，将Opencv版本改为当前安装版本  删除所有的 -march=native  navite表示允许编译器自动探测目标架构（即：编译主机）并生成针对目标架构优化的目标代码。使用时可能出现内存溢出导致段错误。  编译安装一些库时，如g2o，如果默认没有使用-march=native，因此里面的Eigen代码没有使用向量化加速，所以它们并没有内存对齐。而在程序中启用了向量化加速后，所有的Eigen对象都是内存对齐的。两个程序链接起来之后，g2o中未对齐的Eigen对象一旦传递到代码中，向量化运算的指令就会触发异常。综上，要么都用-march=native，要么都不用。 |

## Eigen内存对齐

|  |
| --- |
| Eigen矩阵运算快的主要原因之一就是向量化运算。  所谓向量化运算，就是利用SSE、AVX等SIMD（Single Instruction Multiple Data）指令集，实现一条指令对多个操作数的运算，从而提高代码的吞吐量，实现加速效果。SSE是一个系列，包括从最初的SSE到最新的SSE4.2，支持同时操作16 bytes的数据，即4个float或者2个double。AVX也是一个系列，它是SSE的升级版，支持同时操作32 bytes的数据，即8个float或者4个double。  但向量化运算是有前提的，那就是内存对齐。SSE的操作数，必须16 bytes对齐，而AVX的操作数，必须32 bytes对齐。也就是说，如果我们有4个float数，必须把它们放在连续的且首地址为16的倍数的内存空间中，才能调用SSE的指令进行运算。  因此，对于Eigen对象来说，一旦需要使用SSE指令集进行向量化运算，那必然要求它从内存地址16 byte整数倍的地方开始。如果不满足这个条件，在程序的运行过程中，一旦开始使用SSE指令集，就会产生错误，甚至崩溃。  Eigen对象都需要内存对齐么？  Eigen类型一般可以分为两类：固定大小、非固定大小。  能够向量化运算的固定大小的Eigen类，也被称为固定大小向量化Eigen类型（fixed-size vectorizable Eigen types）。这要求Eigen对象：具有固定大小并且该大小是16个字节的倍数（即16字节对齐的）。 |

## 一个终端多个source

|  |
| --- |
| source xxx/devel/setup.bash –-extend  后缀加—extend，否则会覆盖之前路径 |

## python发布图像话题

|  |
| --- |
| from sensor\_msgs.msg import Image as rosimg  # 定义发布器  bev\_msg = rosimg()  # 新建待发布的消息变量  result = cat\_img(l\_img, r\_img ,settings.xr-settings.xl)  # 待发布的图片  bev\_msg.encoding = 'bgr8'  # 编码方式  bev\_msg.height=result.shape[1]  bev\_msg.width=result.shape[0]  # 尺寸  bev\_msg.step = 1536  # 宽\*通道数，此处为512\*3  bev\_msg.header = msgl.header  # 时间戳等header信息  bev\_msg.data = np.array(image0).tobytes()  # 将图片数组转为byte  global bev\_pub  bev\_pub.publish(bev\_msg) |

## python发布点云话题

|  |
| --- |
| from sensor\_msgs.msg import PointCloud2, PointField  localcloud = [  [1.0, 2.0, 3.0],  [1.0, 2.0, 5.0],  [1.0, 4.0, 3.0],  ]  print(localcloud)  fields =[  PointField('x', 0, PointField.FLOAT32, 1),  PointField('y', 4, PointField.FLOAT32, 1),  PointField('z', 8, PointField.FLOAT32, 1),  ]  # fields[0]、fields[1]、fields[2]分别为每个点的x、y、z坐标字段，均为float类型，4  # 个字节。fields[3]为点云反射强度、float类型，4个字节。fields[4]为每个点对应的# 时间戳，double类型，8个字节，64比特，fields[5]为每个点对应的线ID，值为 # 0~15， 2个字节。在该代码中仅定义了三轴坐标，每个坐标32byte，4字节。  cloudarray = np.array(localpoint\_list, dtype=np.float32)  localcloud\_msg = PointCloud2()  localcloud\_msg.header = msgl.header  localcloud\_msg.header.frame\_id = "local"  localcloud\_msg.width = len(localpoint\_list)  localcloud\_msg.height = 1  # uint32 height：data的高度，一帧点云通常height=1，即表示无序点云；  # uint32 width：data的宽度，即每行对应点的数量；  localcloud\_msg.fields = fields  # sensor\_msgs/PointField[] fields：包含每个点的字段属性信息。  localcloud\_msg.data = cloudarray.tobytes()  # uint8[] data：序列化后的点云二进制数据，所有点信息都在一串字符中，无法通过 # data[i]取值。  localcloud\_msg.point\_step = 12  # 三维坐标，三个4字节数据，共12个字节一个点。  localcloud\_msg.row\_step = localcloud\_msg.point\_step\*len(localpoint\_list)  # uint32 row\_step：每行占用的比特数，=点的数量\*Point\_step;  # bool is\_bigendian：点云是否按正序排列  # uint32 point\_step：每个点占用的比特数，1字节=8比特，与PointField里所有字节# 数之和相等。  localcloud\_msg.is\_dense = True  # bool is\_dense：是否存在无效点。  localpoint\_pub.publish(localcloud\_msg) |

## 话题信息传输协议

|  |
| --- |
| * TransportHints().reliable() * TransportHints().unreliable() * TransportHints().udp() * TransportHints().tcpNoDelay() * TransportHints().tcp()   使用方式：  ros::Subscriber sub\_imu = n.subscribe(IMU\_TOPIC, 2000, imu\_callback, ros::TransportHints().tcpNoDelay());  有工程师经过测试，得到延迟情况如下：  Subscriber queue size: 1, Publisher queue size: 1 **TransportHint** --- **Average communication time (ms)** reliable -----------------7.5 unreliable --------------0.4 udp ----------------------0.4 tcpNoDelay ------------0.4 tcp -----------------------7.7 |

## 录包使用低延迟模式

|  |
| --- |
| rosbag record /xxx -o xxx.bag –tcpnodelay  可尝试与四-5一起作为解决粘包问题 |

## VsCode添加头文件路径

|  |
| --- |
| Ctrl+Shift+P打开搜索栏，搜索configuration打开c\_cpp\_properties.json  在”includePath”:[]中添加头文件路径，不同的路径用逗号分隔。完成后代码补全，代码跳跃功能正常。 |

## GLIBCXX not found

|  |
| --- |
| ImportError: /lib/x86\_64-linux-gnu/libstdc++.so.6: version `GLIBCXX\_3.4.29' not found (required by /home/nrc/anaconda3/envs/gcn-parking/lib/python3.9/site-packages/cv2/pyth  on-3.9/cv2.cpython-39-x86\_64-linux-gnu.so)  #查看是否包含对应文件  strings /home/nrc/anaconda3/envs/gcn-parking/lib/libstdc++.so.6.0.29 | grep GLIBCXX  strings /usr/lib/x86\_64-linux-gnu/libstdc++.so.6 | grep GLIBCXX  strings 在对象文件或二进制文件中查找可打印的字符串；  grep是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来。用于过滤/搜索的特定字符。可使用正则表达式能多种命令配合使用。  # 复制  sudo cp /home/nrc/anaconda3/envs/gcn-parking/lib/libstdc++.so.6.0.29 /usr/lib/x86\_64-linux-gnu/  # 删除之前链接  sudo rm /usr/lib/x86\_64-linux-gnu/libstdc++.so.6  # 创建新的链接  sudo ln -s /usr/lib/x86\_64-linux-gnu/libstdc++.so.6.0.29 /usr/lib/x86\_64-linux-gnu/libstdc++.so.6  ln命令 用来为文件创件连接，连接类型分为硬连接和符号连接两种，默认的连接类型是硬连接。如果要创建符号连接必须使用"-s"选项。软链接像快捷方式，可以对文件和目录做软链接。软链接记录的只是源文件。软链接失去源文件不可用。 |

## libtiff报错

|  |
| --- |
| OSError:/usr/lib/libgdal.so:undefined symbol:TIFFReadRGBATileExt,version LIBTIFF\_4.0  libtiff库版本冲突，降级至4.4.0  conda install libtiff==4.4.0 |

## grep命令

|  |
| --- |
| 在Ubuntu系统中，你可以使用`grep`命令来搜索目录下所有文件中包含某个单词的内容。`grep`是一个非常强大的文本搜索工具，它能够快速地在文件中查找匹配的文本行。以下是一些常用的`grep`命令选项和示例，帮助你在目录下搜索特定的单词。  **基本用法：**  grep [选项] '搜索的单词' 文件或目录  **常用选项：**  - `-r` 或 `--recursive`：递归搜索所有子目录。  - `-i` 或 `--ignore-case`：不区分大小写。  - `-v` 或 `--invert-match`：反转匹配，只显示不包含指定单词的行。  - `-w` 或 `--word-regexp`：匹配整个单词，而不是部分匹配。  - `-n` 或 `--line-number`：显示匹配行的行号。  - `-l` 或 `--files-with-matches`：只列出包含匹配行的文件名，而不显示匹配内容。  - `--include` 或 `--include=PATTERN`：只搜索匹配给定模式的文件。  - `-e` 或 `--regex`：使用正则表达式进行搜索。  **示例：**  1. \*\*递归搜索包含单词 "example" 的所有文件\*\*：  grep -r "example" /path/to/directory  2. \*\*递归搜索且不区分大小写\*\*：  grep -ri "example" /path/to/directory  3. \*\*递归搜索并显示行号\*\*：  grep -rin "example" /path/to/directory  4. \*\*递归搜索特定文件类型\*\*：  grep -r --include="\*.sh" "example" /path/to/directory  5. \*\*只列出包含匹配行的文件名\*\*：  grep -rl "example" /path/to/directory  6. \*\*使用正则表达式搜索\*\*：  grep -r --regex="example.\*world" /path/to/directory  7. \*\*反转匹配，查找不包含 "example" 的行\*\*：  grep -rv "example" /path/to/directory  8. \*\*查找整个单词 "example"\*\*：  grep -rw "example" /path/to/directory |

## 命令行语句说明文档

|  |
| --- |
| 使用man <命令行语句>查看官方文档  如：man grep |

## 话题时间同步

|  |
| --- |
| #include <message\_filters/subscriber.h>  #include <message\_filters/synchronizer.h>  #include <message\_filters/sync\_policies/approximate\_time.h>  using namespace sensor\_msgs;  void callback(const ImageConstPtr& image1, const ImageConstPtr& image2 ){}  int main(){  message\_filters::Subscriber<Image> image1\_sub(nh, "image1", 1);  message\_filters::Subscriber<Image> image2\_sub(nh, "image2", 1);  typedef message\_filters::sync\_policies::ApproximateTime<Imamge, Image> SyncPolicy;  SyncPolicy mySyncPolicy(10);  mySyncPolicy.setAgePenalty(0.1);  mySyncPolicy.setInterMessageLowerBound( 0, ros::Duration(0.05) );  mySyncPolicy.setInterMessageLowerBound( 0, ros::Duration(0.1) );  mySyncPolicy.setMaxIntervalDuration( ros::Duration(0.2) )  message\_filters::Synchronizer<SyncPolicy> sync(mySyncPolicy, image1\_sub, image2\_sub);  sync.registerCallback(boost::bind(&callback,\_1,\_2));  }  Age penalty: 当比较后面的时间间隔会乘上一个惩罚因子(1+AgePenalty)，一个非零的Age penalty会更早的输出匹配消息集，或输出更多的消息集，以牺牲质量为代价  inter message lower bound: 如果特定的消息两帧之间不能比已知的间隔更接近，给定这个间隔下界会帮助算法更早的得出一个给定消息集是最优匹配，从而减小延迟。不正确的下界将导致选择次优集合。一个典型的bound：相机帧率的一半。  Max interval duration: 如果候选匹配消息集合的时间间隔大于此值则不考虑。丢弃最早的消息，转向下一个消息。 |

## ubuntu下文件所有者及权限

|  |
| --- |
| 更改文件所有者  sudo chown nrc xxx.launch  修改文件所有者为当前用户，或者root，所有者为root时修改读写权限后还是不可修改或执行，需要修改所有者  -c或——changes：效果类似“-v”参数，但仅回报更改的部分；  -f或--quite或——silent：不显示错误信息；  -h或--no-dereference：只对符号连接的文件作修改，而不更改其他任何相关文件；  -R或——recursive：递归处理，将指定目录下的所有文件及子目录一并处理；  -v或——version：显示指令执行过程；  更改文件权限  sudo chmod +x xxx.launch  u # 操作对象简称，用户user，文件或目录的所有者。  g # 操作对象简称，同组用户group，文件或目录所属群组  o # 操作对象简称，其它用户others  a # 操作对象简称，所有用户all，系统默认使用此项  + # 权限操作符，添加某些权限  - # 权限操作符，取消某些权限  = # 权限操作符，设置文件的权限为给定的权限  r # 权限设定（英文），表示可读权限  w # 权限设定（英文），表示可写权限  x # 权限设定（英文），表示可执行权限  - # 权限设定（英文字符），表示没有权限  X # 权限设定，如果目标文件是可执行文件或目录，可给其设置可执行权限  s # 权限设定，设置权限suid和sgid，使用权限组合“u+s”设定文件的用户的ID位，“g+s”设置组ID位  t # 权限设定，只有目录或文件的所有者才可以删除目录下的文件  -c或——changes # 效果类似“-v”参数，但仅回报更改的部分，如果文件权限已经改变，显示其操作信息；  -f或--quiet或——silent # 操作过程中不显示任何错误信息；  -R或——recursive # 递归处理，将指令目录下的所有文件及子目录一并处理；  -v或——verbose # 显示命令运行时的详细执行过程；  --reference=<参考文件或目录> # 把指定文件或目录的所属群组全部设成和参考文件或目录的所属群组相同；  <权限范围>+<权限设置> # 开启权限范围的文件或目录的该选项权限设置；  <权限范围>-<权限设置> # 关闭权限范围的文件或目录的该选项权限设置；  <权限范围>=<权限设置> # 指定权限范围的文件或目录的该选项权限设置；  有时会使用数字来代表权限  第一个数字表示文件所有者的权限。  第二个数字表示与文件所有者同属一个用户组的其他用户在该文件夹上的权限。  第三个数字表示其他用户组在该文件夹上的权限。  权限分为三种：读（r=4）、写（w=2）、执行（x=1）。  根据不同的权限组合方式，可以得到以下模式和对应的数字：  可读+可执行（rx=5）对应数字5，  可读+可写（rw=6）对应数字6，  可读+可写+可执行（rwx=7）对应数字7。  举例，chmod 755 设置用户的权限为：  文件所有者可读可写可执行  与文件所有者同属一个用户组的其他用户可读可执行  其它用户组可读可执行 |

## launch文件

|  |
| --- |
| 以下示例为开启两个节点，设置布尔、字符串、数值类型参数，remap修改字符串。  <launch>  <node name="" pkg="" type="" args=""/>  <node name="" pkg="" type="" args="">  <param name="flag" value="true"/>  <param name="frame\_id" type="string" value="map"/>  <param name="x" value="0.6"/>  <remap from="/talker" to="/re\_talker" />  </node>  <include file="xxx.launch"/>  </launch>  标蓝处为两种node结束方法。launch文件中可以通过include方法执行多个launch文件。  在launch文件中，${find xxx\_package\_name}返回某个包的路径，该路径是对应包CMakeLists.txt所在路径，名字是CMakeLists.txt中project(xxx\_package\_nage)处设置的包名。 |

## ROS日志流

|  |
| --- |
| 五个级别：DEBUG、INFO、WARN、ERROR、FATAL  ROS\_DEBUG\_STREAM("xxx is "<<a)  ROS\_INFO\_STREAM()  ROS\_WARN\_STREAM()  ROS\_ERROR\_STREAM()  ROS\_FATAL\_STREAM()  一个终端使用roslaunch启动多个节点时，cout无法打印信息，需要使用ros日志流来打印参数，便于调试。 |

## Ubuntu挂载硬盘

|  |
| --- |
| 问题背景：电脑装有两块固态硬盘，现在需要安装ubuntu系统并分配存储空间。  在通过启动U盘安装ubuntu时，尝试选择A硬盘上剩余的空间和B硬盘全部都挂载到/home上，发生报错。只能先将A硬盘上剩余的空间挂载到/home，安装完成后将另一块硬盘挂载到新路径下(/home下的某一个目录，如：/home/nrc/DISK2)  lsblk -f 查看硬盘挂载情况和UUID  sudo mkfs.ext4 /dev/sda 以ext4格式格式化硬盘  sudo mount /dev/sda /home/nrc/DISK2  sudo gedit /etc/fstab 在其中加入：  /dev/sda /home/nrc/DISK2 ext4 defaults 0 0  这一步是为了计算机掉电再次开机时依然可以挂载硬盘。  /dev/sda 可以替换为UUID=d023b043-d291….-ecdb7a58050，作用是标识待挂载的硬盘。  挂载后可能出现文件夹权限问题。  使用sudo chown nrc /home/nrc/DISK2 修改挂载点文件夹的所属者。 |

## Anaconda相关

### conda activate 失败

|  |
| --- |
| 1. 使用source activate xxx启动环境 2. source ~/.bashrc之后再conda activate |

### 取消默认进入base环境

|  |
| --- |
| 1. conda config --set auto\_activate\_base false 2. 修改~/.bashrc   把export PATH="~/anaconda/bin:$PATH"  修改为export PATH="$PATH:~/anaconda/bin"  默认先搜索系统的python解释器，之后才使用conda环境的python。 |

### .sh脚本中激活conda虚拟环境

|  |
| --- |
| **方法一：**先source conda.sh  #!/bin/bash  source ~/anaconda3/etc/profile.d/conda.sh  conda activate wrfpy && python test.py  **方法二**：使用完整路径  # 在shell中直接使用目标环境下python的绝对路径  ~/anaconda3/envs/wrfpy/bin/python test.py  **方法三：**bash hook  #!/bin/bash  eval "$(conda shell.bash hook)"  conda activate wrfpy && python test.py  **方法四**：source activate  #!/bin/bash  source ~/anaconda3/bin/activate wrfpy && python test.py  **方法五：**通过bash -i test.sh执行  #!/bin/bash  conda activate wrfpy && python test.py  **方法六**：通过source test.sh执行  #!/bin/bash  conda activate wrfpy && python test.py |

## 系统环境变量

|  |
| --- |
| 可以通过在终端执行  echo $PATH  来打印系统环境变量的路径，各路径用冒号(:)分隔。运行命令时，PATH是要搜索的目录列表。输入一个指令后，系统按环境变量的顺序查找该命令。全部路径下都找不到报错：command not found！  修改环境变量可以在终端中执行  PATH=$PATH:/home/nrc/xxx  使得当前终端加入该环境变量，要想永久加入某环境变量，需要在~/.bashrc文件中添加  export PATH=$PATH:/home/nrc/xxx  注：=两边不能有空格  可以使用printenv命令查看所有环境变量 |

## D435i查看内外参

|  |
| --- |
| rs-enumerate-devices -c  输出IMU和相机在不同分辨率下的内参，以及相机到IMU的外参。 |

## D435i驱动安装

|  |
| --- |
| 1. 注册服务器公钥   sudo apt-get update && sudo apt-get upgrade && sudo apt-get dist-upgrade  sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-key F6E65AC044F831AC80A06380C8B3A55A6F3EFCDE || sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-key F6E65AC044F831AC80A06380C8B3A55A6F3EFCDE   1. 将服务器添加到存储库列表中   echo "deb [signed-by=/etc/apt/keyrings/librealsense.pgp] https://librealsense.intel.com/Debian/apt-repo `lsb\_release -cs` main" | \  sudo tee /etc/apt/sources.list.d/librealsense.list  sudo apt-get update  这一步使用官方的方法可能出错  sudo add-apt-repository "deb https://librealsense.intel.com/Debian/apt-repo $(lsb\_release -cs) main" -u   1. 将服务器添加到存储库列表中   sudo apt-get install librealsense2-dkms  sudo apt-get install librealsense2-utils  sudo apt-get install librealsense2-dev  sudo apt-get install librealsense2-dbg |

## Ubuntu科学上网

|  |
| --- |
| github上下载qv2ray：https://github.com/Qv2ray/Qv2ray/releases，进入GitHub中的仓库，找到Linux下的对应软件[Qv2ray-v2.7.0-linux-x64.AppImage](https://github.com/Qv2ray/Qv2ray/releases)在/home/xxx中新建文件夹qv2ray，并将对应软件解压至/home/xxx/qv2ray/中然后给该软件赋可执行权限。  双击打开qv2ray，左上角点击“首选项”-“内核设置(Kernel Settings)”，需要配置V2Ray核心可执行文件路径和V2Ray资源目录。下载[核心文件](https://github.com/v2fly/v2ray-core/releases/download/v4.32.0/v2ray-linux-64.zip)后解压至/home/xxx/.config/qv2ray。该文件夹此处重命名为vcore。检验内核设置中的路径是否正确。如果不通过可以检查一下vcore文件夹中的v2ctl和v2ray文件是不是没赋予可执行权限。    添加链接：首页-（左上角）分组-默认分组-订阅设置-勾选“此分组是一个订阅”-填写“订阅地址”-“订阅类型”选择“Basic Base64”。订阅链接需要上各大机场上订阅，订阅链接名为<https://xxx>格式。  此时只能浏览器翻墙，浏览器翻墙时需要将浏览器的设置->proxy设置成使用系统代理。终端翻墙需要安装工具包proxychains，如下图所示    编辑配置文件        如上图：sock4是127.0.0.1 1089修改好之后，保存配置文件除了用sock4，也可以用http，上图的http为：127.0.0.1 8889。  用这个指令测试终端翻墙是否配置成功。sudo proxychains apt update。成功的话显示如下： <>-ok。proxychains等级仅小于sudo，如果指令没有sudo，那就把proxychains指令放在最前面。 |

## Sophus库缺失fmt库依赖

|  |
| --- |
| fatal error:fmt/format.h:没有那个文件或目录  下载format源码安装，仍存在许多报错undefined inference to fmt::v8::vprint ……  使用命令行sudo apt-get install -y libfmt-dev  在工程CMakeLists.txt中添加target\_link\_libraries(vins\_lib fmt) |

## 双击运行sh脚本

|  |
| --- |
| sudo apt-get install dconf-tools安装插件  终端运行dconf-editor  可视化界面中依次打开org->gnome->nautilus->preferences  executable-text-activation中可以设置为launch或者ask    此处设置为ask，双击sh文件会弹窗，可以选择想要的操作    在实际运行时发现，可能存在找不到roscore、rviz、catkin\_make等ros指令的情况，需要在对应的指令前加上source /opt/ros/melodic/setup.bash  如  #!/bin/sh  {  gnome-terminal -t “core” -x bash -c “source /opt/ros/melodic/setup.bash;roscore;exec bash”  }&  或者  #!/bin/sh  source /opt/ros/melodic/setup.bash  gnome-terminal -t “core” -x bash -c “roscore;exec bash” |

## Ubuntu包管理工具

|  |
| --- |
| 1. apt-get(Advanced Package Tool)   官方包管理工具。   1. get   apt-get易用升级版本。   1. aptitude   aptitude 在处理依赖问题上更佳一些。举例来说，aptitude 在删除一个包时，会同时删除本身所依赖的包。这样，系统中不会残留无用的包，整个系统更为干净。安装包时也会考虑软件依赖关系，给用户多个解决方案。 |

## 多线程回调函数

|  |
| --- |
| 1、ros::spinOnce()和spin()  while (ros::ok())  {  send1();  send2();  ros::spinOnce();  r.sleep();  }  ros::spin()可以看作r.sleep()=0的ros::spinOnce()  2、ros::MultiThreadedSpinner()  ros::MultiThreadedSpinner可以实现多线程的回调函数执行，需要和spinner.spin()一起使用，因此是阻塞的， MultiThreadedSpinner的用法是：  ros::MultiThreadedSpinner spinner(4); // 数字表示调用的线程数  spinner.spin(); // spin() will not return until the node has been shutdown  3、ros::AsyncSpinner()  ros::AsyncSpinner 是一种和ros::MultiThreadedSpinner类似的多线程回调函数执行器，不过AsyncSpinner不是阻塞的，因此可以和while()配合去执行一些别的任务，常见的用法如下：  ros::Rate loop\_rate(m\_frameRate);  // 使用 ros::AsyncSpinner 可以使这些回调在后台线程中异步地执行，而不是顺序地执行。  // 这允许你的主线程继续执行其他任务，而不是被回调阻塞。  ros::AsyncSpinner spinner(m\_threadNum);  spinner.start();  //当当前节点没有关闭时  while (ros::ok()) {  // 可以在这里执行一些别的任务...  // ...  ros::spinOnce();  loop\_rate.sleep();  }  //如果当前节点关闭  rosShutdown(); |

## | 与 &

|  |
| --- |
| 1. 竖线‘|’ ，在linux中是作为管道符的，将‘|’前面命令的输出作为'|'后面的输入。 2. 双竖线‘||’，用双竖线‘||’分割的多条命令，执行的时候遵循如下规则，如果前一条命令为真，则后面的命令不会执行，如果前一条命令为假，则继续执行后面的命令。 3. &同时执行多条命令，不管命令是否执行成功。 4. && 可同时执行多条命令，当碰到执行错误的命令时，将不再执行后面的命令。如果一直没有错误的，则执行完毕。 |

## 网口数据传输

|  |
| --- |
| 1. IP、子网掩码、网关   IP地址 = 网络地址 + 主机地址。  子网掩码必须和IP地址成对出现，否则没有意义。  子网掩码是用于给IP地址划分网络地址与主机地址的。  和子网掩码1对应的IP地址，代表网络位；和子网掩码0对应的IP地址，代表主机位。  子网中只要1是连续的，就是合理子网掩码。  网关实质上是一个网络通向其他网络的IP地址。  公网IP（Internet Protocol）是指用于互联网通信的IP地址，它是互联网上每个设备在网络中的唯一标识。与公网IP相对的是私有IP，私有IP用于内部网络通信，无法直接访问互联网。   1. 设置巨帧   以太网（Ethernet）协议对数据帧的大小有一个限制，默认是1500字节，这个数值就是以太网的MTU（Maximum Transfer Unit）。  sudo ifconfig th0 down  sudo ifconfig eth0 mtu 8192  sudo ifconfig th0 up   1. 设置网口缓冲区大小   在/etc/sysctl.confw文件中加入  #接收区默认大小  net.core.rmem\_default=8388688  #接收区最大大小  net.core.rmem\_max=16777216  #发送区最大大小  net.core.wmem\_max=16777216 |

## 同局域网ssh控制

|  |
| --- |
| 1、Ubuntu安装SSH  sudo apt-get install openssh-server  2、windows终端控制  键盘win+r，输入cmd，打开命令行窗口  ssh （Ubuntu系统名称）@（Ubuntu的IP地址）  如：ssh [nrc@192.168.31.171](mailto:nrc@192.168.31.171)  输入密码后可以使用终端控制ubuntu端 |

## 非局域网远程控制

|  |
| --- |
| Ubuntu安装x11vnc，windows安装VNCviewer。  使用cpolar内网穿透，windows端会报错The connection closed unexpect。 |

## ubuntu进入tty模式

|  |
| --- |
| 修改Display Manager可能出现电脑重启后进不去图形界面  # 输入  sudo dpkg-reconfigure lightdm  # 或者  sudo dpkg-reconfigure gdm3  选择默认的Display Manager，重启电脑即可。 |

# ros2

## 初始化

# WORD

## 打开word报错无法创建工作空间

|  |
| --- |
| 取消勾选查看中的预览窗格，问题解决。 |

# 六、EXCEL

## 格式化公式

|  |
| --- |
| ="ren "&B1&" "&C1  注意英文双引号，固定字符用双引号括起来，列名用&开头，后续有内容时还要加&表示截止（如上述例子的C1、B1） |

## 自动调整行高

|  |
| --- |
| 选中表格，在单元格-格式下拉菜单中有自动调整行高。 |

# 七、PPT

# 常用网站

## Linux命令行查询

|  |
| --- |
| https://www.lzltool.com/LinuxCommand |

## python API查询

|  |
| --- |
| <https://pypi.org/> |

## ROS API

|  |
| --- |
| <https://docs.ros.org/en/api/>  <https://index.ros.org/>  以sensor\_msgs/NavSatFix为例：  <http://docs.ros.org/en/api/sensor_msgs/html/msg/NavSatFix.html>  网站有该话题类型，以及包含的数据 |

## Pytorch中文文档

|  |
| --- |
| <https://pytorch-cn.readthedocs.io/zh/latest/> |

## PCL官方文档

|  |
| --- |
| <https://pointclouds.org/documentation/> |

# 硬件

## USB接口电磁干扰卫星信号

|  |
| --- |
| USB线可能会存在电磁干扰，影响卫星定位结果。在使用大疆M300进行航路点飞行时，经常出现RTK信号差的问题。  解决方案：   * 1. 接口处包裹铜箔   2. 更换USB线   3. 设备使用金属壳 |

## 三脚插头接线

|  |
| --- |
| 插头端定义：  三线插头怎么接  插座端定义：    线路定义：火线（棕色）、零线（蓝色）、地线（黄绿双色）    以明纬开关电源接线为例：  输入端：  AC/L(Alternating Current/live line)>>火线>>棕色接线  AC/N(Alternating Current/null line)>>零线>>蓝色接线  地线(ground line)>>黄绿双色线    输出端：  V+相连  V-相连 |

# 算法

## Lsd直线检测

|  |  |
| --- | --- |
| 输入 | 灰度图像I，采样率S，方向变化最大值τ，截断阈值D0，错误控制阈值ε |
| 输出 | 检测直线段数据RECT |
| 步骤一 | 如果图像比较大，需要高斯降采样压缩图像 |
| 步骤二 | 计算各像素点梯度值和方向，按梯度值从大到小排序 |
| 步骤三 | 将梯度值小于ρ的点的状态设置为USED，其余设置为UNUSED |
| 步骤四 | 取出排序表中余下状态为UNUSED的点 |
| 步骤五 | 设置方向变化最大阈值τ进行区域增长Rigion Grow。Rigion Grow就是判断起始点周围的像素是否小于梯度方向变化阈值，若小于，认为这些像素同属于该条直线。 |
| 步骤六 | 计算生长所得到区域的同性点密度D |
| 步骤七 | 如果D<D0，截断区域，转至步骤六  如果D>D0，计算错误控制函数NFA |
| 步骤八 | 如果NFA<ε，认为检测到直线，输出直线段数据（起始点）。  判断排序表中是否还有状态为UNUSED的点，若有转步骤四，若无结束程序 |

## 深度优先

|  |
| --- |
| Depth First Search, 简称 DFS。从图中一个未访问的顶点 V 开始，沿着一条路一直走到底，然后从这条路尽头的节点回退到上一个节点，再从另一条路开始走到底…，不断重复此过程，直到所有的顶点都遍历完成。  递归的表达性很好，也很容易理解。但是如果层级过深，很容易导致栈溢出。以下为非递归的实现方法。  /\*\*  \* 使用栈来实现对二叉树的 dfs  \* @param root  \*/  public static void dfsWithStack(Node root) {  if (root == null) {  return;  }    Stack<Node> stack = new Stack<>();  // 先把根节点压栈  stack.push(root);  while (!stack.isEmpty()) {  Node treeNode = stack.pop();  // 遍历节点  process(treeNode)    // 先压右节点  if (treeNode.right != null) {  stack.push(treeNode.right);  }    // 再压左节点  if (treeNode.left != null) {  stack.push(treeNode.left);  }  }  } |

## 广度优先

|  |
| --- |
| Breadth First Search 简称 BFS。广度优先遍历，指的是从图的一个未遍历的节点出发，先遍历这个节点的相邻节点，再依次遍历每个相邻节点的相邻节点，直到所有的顶点都遍历完成。  /\*\*  \* 使用队列实现 bfs  \* @param root  \*/  private static void bfs(Node root) {  if (root == null) {  return;  }  Queue<Node> stack = new LinkedList<>();  stack.add(root);    while (!stack.isEmpty()) {  Node node = stack.poll();  System.out.println("value = " + node.value);  Node left = node.left;  if (left != null) {  stack.add(left);  }  Node right = node.right;  if (right != null) {  stack.add(right);  }  }  } |

## 迪杰斯特拉(Dijkstra)

|  |
| --- |
| 弗洛伊德算法(Floyd)计算图中各个顶点之间的最短路径，迪杰斯特拉算法(Dijkstra)用于计算图中任意一个顶点到其他顶点的最短路径。  首先将起始节点的距离标记为0，其他节点的距离因为还不确定所以先需要标记为无穷大。然后，在图中找到距离起始节点最近的节点，更新其相邻节点的距离，距离为从起始节点到该节点的距离加上该节点到相邻节点的距离。不断循环此过程，直到所有节点都被访问过。 |

## 弗洛伊德(Floyd)

|  |
| --- |
| 弗洛伊德算法(Floyd)计算图中各个顶点之间的最短路径，迪杰斯特拉算法(Dijkstra)用于计算图中任意一个顶点到其他顶点的最短路径。  将每个点作为路线中间点进行迭代。    public class FloydAlgorithm {  public static void main(String[] args) {  // 测试看看图是否创建成功  char[] vertex = { 'A', 'B', 'C', 'D', 'E', 'F', 'G' };  //创建邻接矩阵  int[][] matrix = new int[vertex.length][vertex.length];  final int N = 65535;  matrix[0] = new int[] { 0, 5, 7, N, N, N, 2 };  matrix[1] = new int[] { 5, 0, N, 9, N, N, 3 };  matrix[2] = new int[] { 7, N, 0, N, 8, N, N };  matrix[3] = new int[] { N, 9, N, 0, N, 4, N };  matrix[4] = new int[] { N, N, 8, N, 0, 5, 4 };  matrix[5] = new int[] { N, N, N, 4, 5, 0, 6 };  matrix[6] = new int[] { 2, 3, N, N, 4, 6, 0 };  //创建 Graph 对象  Graph graph = new Graph(vertex.length, matrix, vertex);  //调用弗洛伊德算法  graph.floyd();  graph.show();  }  }  // 创建图  class Graph {  private char[] vertex; // 存放顶点的数组  private int[][] dis; // 保存，从各个顶点出发到其它顶点的距离，最后的结果，也是保留在该数组  private int[][] pre;// 保存到达目标顶点的前驱顶点  // 构造器  /\*\*  \*  \* @param length  \* 大小  \* @param matrix  \* 邻接矩阵  \* @param vertex  \* 顶点数组  \*/  public Graph(int length, int[][] matrix, char[] vertex) {  this.vertex = vertex;  this.dis = matrix;  this.pre = new int[length][length];  // 对 pre 数组初始化, 注意存放的是前驱顶点的下标  for (int i = 0; i < length; i++) {  Arrays.fill(pre[i], i);  }  }  // 显示 pre 数组和 dis 数组  public void show() {  //为了显示便于阅读，我们优化一下输出  char[] vertex = { 'A', 'B', 'C', 'D', 'E', 'F', 'G' };  for (int k = 0; k < dis.length; k++) {  // 先将 pre 数组输出的一行  for (int i = 0; i < dis.length; i++) {  System.out.print(vertex[pre[k][i]] + " ");  }  System.out.println();  // 输出 dis 数组的一行数据  for (int i = 0; i < dis.length; i++) {  System.out.print("("+vertex[k]+"到"+vertex[i]+"的最短路径是" + dis[k][i] + ") ");  }  System.out.println();  System.out.println();  }  }  //弗洛伊德算法, 比较容易理解，而且容易实现  public void floyd() {  int len = 0; //变量保存距离  //对中间顶点遍历， k 就是中间顶点的下标 [A, B, C, D, E, F, G]  for(int k = 0; k < dis.length; k++) { //  //从 i 顶点开始出发 [A, B, C, D, E, F, G]  for(int i = 0; i < dis.length; i++) {  //到达 j 顶点 // [A, B, C, D, E, F, G]  for(int j = 0; j < dis.length; j++) {  len = dis[i][k] + dis[k][j];// => 求出从 i 顶点出发，经过 k 中间顶点，到达 j 顶点距离  if(len < dis[i][j]) {//如果 len 小于 dis[i][j]  dis[i][j] = len;//更新距离  pre[i][j] = pre[k][j];//更新前驱顶点  }  }  }  }  }  } |

## A\*路径规划

|  |
| --- |
|  |

## 卡尔曼滤波(KF)

|  |
| --- |
| 预测：    其中  更新：    ：上一时刻状态  ：上一时刻系统输入  ：测量噪声  ：当前先验状态，通过预测获得的状态  ：状态转移矩阵  ：上一时刻状态的协方差  ：当前时刻先验估计协方差  ：卡尔曼增益  ：观测量  卡尔曼滤波输入为观测值，将状态量看作一个符合正态分布的随机变量，需要求解的最终值是该随机变量的均值，目的是使其方差最小。 |

## 扩展卡尔曼滤波(EKF)

## EKF融合相机和IMU信息估计姿态

|  |
| --- |
| 坐标系定义：  IMU坐标系：右下前  IMU回调函数中调整为前左上坐标系(Right-Down-Front->Front-Left-Up)，均为右手系。    相机坐标系：右下前。  状态方程：  状态量设置为四元数和陀螺漂移    设为载体的旋转角速度，则用四元数的形式可以表示为    四元数微分方程为    按照四元数乘法展开上式可得    写成矩阵形式    其中为机体坐标系相对于地理坐标系的角速度在载体坐标系上的分量，也就是陀螺仪的输出    由一阶龙格-库塔法可得姿态更新离散模型为    为系统采样时间间隔；为 时刻的四 元数；为时刻的四元数。    对  求取 Jacobi 矩阵    量测方程：  IMU测量的是比力，静止状态下测量的是支持力，大小与重力相等，方向与重力相反。在前左上坐标系中，IMU量测为：      其中    量测方程为    对求取 Jacobi 矩阵，可得系统量测矩阵为    式中，  在实际安装中，初始状态下imu读数不会是标准的，存在安装角。，其中  2D图像求解航向角：  OpenCV中的recoverPose函数求解得到相邻两帧的旋转矩阵通过该函数求解出来的 、，它表示的是points1到points2的变换，也就是。程序中points1处输入k-1时刻特征点，程序中points2处输入k时刻特征点，特征点变换关系有    以第零帧IMU系为基准系    可以求得时刻相对于基准系的旋转矩阵：    该旋转矩阵是相对于两个右下前坐标系的，特征点在右下前坐标系与前左上坐标系中的坐标相差一个变换矩阵      解算位姿四元数时应从该旋转矩阵求解 |

## ESKF

## 光束法平差 (Bundle Adjustment)

|  |
| --- |
| BA是指将相机位姿和观测信息(如特征点)进行联合优化。  优化目标：所有重投影误差的和最小。  优化方法：   1. 最速下降法     函数梯度方向是上升最快的方向，反梯度方向则是下降最快的方向。最速下降法保证每次迭代方向都是下降的，当步长过长时，可能也会出现上升的情况。按照一定步长沿梯度相反方向迭代，成折线下降趋势，因此迭代速度较慢。   1. Newton型方法   导数为0处就是函数最小处。首先将函数利用泰勒展开到二次项：    其中H为Hessian矩阵，是二次偏导矩阵（J为雅可比（Jacobian)矩阵，即一次偏导矩阵）。    Newton型方法收敛很快，尤其是对于二次函数而言一步就可以得到结果。但是该方法有个最大的缺点就是Hessian矩阵计算十分复杂，并且Newton型方法的迭代并不保证每次迭代都是下降的。   1. Gauss-Newton方法   在范数符号内将e一阶泰勒展开，我们可以得到    其中J为Jacobi矩阵，函数e对x求一次偏导而来，梯度也是对向量函数求一次偏导而来。将标量考虑为1 × 1的矩阵，将向量考虑为n × 1的矩阵，其实这些求导都是求Jacobi矩阵。此时不需要计算Hessian矩阵。  同样，二次函数导数为0时取极值，则有    由此在方向上迭代直至最小。  Gauss-Newton方法就避免了求Hessian矩阵，并且收敛很快。但是依旧无法保证每次迭代的时候函数都是下降的（虽然从上式可以推导出来是下降方向，但是步长可能过长）。   1. LM方法(Levenberg-Marquardt)   本质是通过参数在最速下降法和Gauss-Newton法之间切换。   1. 李群李代数的应用   在优化迭代的过程中，我们求的值为，然后利用来更新的值。对于空间点的坐标平移没有问题，但是旋转矩阵而言是不存在加法的，应该使用 更新旋转。优化算法的迭代过程又不能是乘法，这就出现了矛盾。这里旋转矩阵及相关运算属于李群，此时应将旋转矩阵变换到其对应的李代数上进行计算，然后再变换回李群。 |

## 复数与二维旋转

|  |
| --- |
| 对于两个复数    相乘计算结果为    其中是复数的矩阵形式    该矩阵形式提出一个，而每个复数可以在复平面上表示为一个点：    将斜边与实数轴正方向的夹角记为 ，满足  , ,  复数的矩阵形式可以分解为左侧的缩放矩阵以及右侧的二维旋转矩阵    当模长为1时表示纯旋转  (二维旋转矩阵型)  反之，已知旋转角，可以构建一个复数与待变换向量所构建的复数相乘，计算旋转之后的向量  (二维旋转复数积型)  由欧拉公式可得    复数的极坐标形式为    当模长为1时表示纯旋转  (二维变换指数型)  旋转变换可以进行复合，有两个旋转变换      以及一个向量    两次变换的等效变换为      由于复数乘法具有交换律，可证多个二维旋转的结果与其施加顺序无关。 |

## 三维旋转

|  |
| --- |
| 角轴式：绕着指定轴旋转指定角度。默认旋转轴模长为1。    将待旋转向量分解为垂直旋转轴和平行旋转轴的两个向量    对这两个分量进行旋转之后再合并，得到原向量旋转之后的向量。平行于旋转轴的分量就是向量在旋转轴上的投影。    由可得    平行于旋转轴的分量相当于没有被旋转    垂直于旋转轴的分量旋转后    构造一个同时垂直于旋转轴和的向量，.    该向量模长与模长相等    将分解到和上    综上可得Rodrigues公式 |

## 四元数相关计算

### 基本形式

|  |
| --- |
| 四元数由一个实部和三个虚部组成    与复数类似，因为四元数其实就是对于基 的线性组合，四元数也可以写成向量的形式    也可以将四元数的实部与虚部分开，并用一个三维的向量来表示虚部，将它表示为标量和向量的有序对形式    如果一个四元数实部为零    称之为纯四元数 |

### 乘法

|  |
| --- |
| 四元数之间的乘法比较特殊，它们是不遵守交换律的，也就是说一般情况 下。这也就有了左乘和右乘的区别．如果是，称为 左乘以，如果是，称为右乘以。除了交换律之外，结合律和分配律在四元数内都是成立的。  对于两个四元数        对于有序对形式的两个四元数，的结果可以用向量的点乘及叉乘表示，称为Graßmann 积      两个纯四元数做乘法运算 |

### 逆与共轭

|  |
| --- |
| 逆运算满足    四元数 的共轭为  四元数与其共轭的乘法满足交换律  逆与共轭之间的关系 |

### 四元数与角轴式旋转

|  |
| --- |
| 向量绕旋转轴旋转      反之，已知旋转四元数时，可求得旋转轴及旋转角 |

### 双倍覆盖

|  |
| --- |
| 四元数与三维旋转并不是一对一的关系，对于一个单位四元数，是绕旋转轴旋转，是绕旋转轴旋转，和等价于同一个旋转。  单位四元数与 3D 旋转有一个2 对 1 满射同态(2-1 Surjective Homomorphism)，或者说单位四元数双倍覆盖 (Double Cover) 了 3D 旋转。 |

### 四元数插值

|  |
| --- |
| 假设有两个旋转变换和 ，中间变换使得初始变换平滑的过渡到。 |

## IMU预积分

|  |
| --- |
|  |

## 因子图

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **图**（Graph）是一种复杂的非线性结构，在图结构中，每个元素都可以有零个或多个前驱，也可以有零个或多个后继，也就是说，元素之间的关系是任意的。   |  |  | | --- | --- | | **术语** | **含义** | | 顶点 | 图中的某个结点 | | 边 | 顶点之间连线 | | 相邻顶点 | 由同一条边连接在一起的顶点 | | 度 | 一个顶点的相邻顶点个数 | | 简单路径 | 由一个顶点到另一个顶点的路线，且没有重复经过顶点 | | 回路 | 出发点和结束点都是同一个顶点 | | 无向图 | 图中所有的边都没有方向 | | 有向图 | 图中所有的边都有方向 | | 无权图 | 图中的边没有权重值 | | 有权图 | 图中的边带有一定的权重值 |   图的结构很简单，就是由顶点 V 集和边 E 集构成，因此图可以表示成 G = (V,E)。 |

## Vins-fusion

|  |
| --- |
| Ceres只能求解最小二乘优化，有些情况下需要使得马氏距离最小，如    这种情况需要对协方差矩阵P进行LLT分解      在计算残差时左乘一个    Vins-fusion中imu\_factor.cpp中IMUFactor::Evaluate()使用到这种处理方法。 |

## Orb-slam

## SVO

|  |
| --- |
|  |

# 其他

## RTK

|  |
| --- |
| $GNGGA,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,M,<10>,M,<11>,<12>\*hh  <1> 023229.000：UTC时间，hhmmss.fff（时分秒.毫秒）格式  <2> 3640.6001：纬度ddmm.mmmm（度分）格式（前面的0也将被传输）  <3> N：纬度半球N（北半球）或S（南半球）  <4> 11707.8562：经度dddmm.mmmm（度分）格式（前面的0也将被传输）  <5> E：经度半球E（东经）或W（西经）  <6> 2: GPS状态：0=未定位，1=非差分定位，2=差分定位，4=固定解，5=浮点解，6=正在估算  <7> 10：正在使用解算位置的卫星数量（00~12）（前面的0也将被传输）  <8> 1.16：HDOP水平精度因子（0.5~99.9）  <9> 79.5：海拔高度（-9999.9~99999.9）  <10> -2.4 ：地球椭球面相对大地水准面的高度  <11> 差分时间（从最近一次接收到差分信号开始的秒数，如果不是差分定位将为空）  单点解：表示未收到差分数据，精度最低，一般为 10 米以内；  伪距解：表示接收到基准站差分，经过伪距解算得到的解，或接收 SBAS 信号得到的解状态，精度较低，一般为 3 米以内；  浮点解：表示接收到基准站差分，经载波相位差分数据解算得到的初步解，精度较高，一般在 0.5 米以内；  固定解：表示接收到基准站差分，经载波相位差分数据解算得到的最终解，精度较最高，一般在 0.02 米以内，进行高精度 GPS 测量时，需要达到固定解状态才能记录数据； |

## 投影坐标系

WGS84 / UTM zone 50N：基于WGS84坐标系的投影坐标系，50为条带号，N为北半球。转换为经纬高可使用以下网站：

|  |
| --- |
| https://epsg.io/transform#s\_srs=32650&t\_srs=4326&x=NaN&y=NaN  选择对应的坐标系，填入坐标，点击transform。（经纬高坐标系为WGS84 world） |

## 标定IMU

|  |
| --- |
| 录制两小时以上的IMU静置数据,修改launch文件  roslaunch imu\_utils xxx.launch  rosbag play imu.bag -r 200 |

## Eigen坐标变换常用类型

|  |
| --- |
| AngleAxisd rotation\_vector(Pi,Vector3d(0,0,1));  //绕z轴旋转180°的旋转向量  Matrix3d rotation\_matrix = Martix3d::Identity();  rotation\_matrix = rotation\_vector.toRotationMatrix();  //从旋转向量获得旋转矩阵  Vector3d euler\_angles = rotation\_matrix.eulerAngles(2,1,0);  //从旋转矩阵获得欧拉角，按照ZYX顺序，即yaw-pitch-roll顺序  Isometry3d T = Isometry3d::Identity();  T.rotate(rotation\_vector);  T.pretranslate(Vector3d(1,3,4));  //通过旋转向量和平移向量获取变换矩阵  //名称中为3d，实质是4\*4的矩阵  Quaterniond q= Quaterniond(rotation\_vector);  q= Quaterniond(rotation\_matrix);  std::cout<<“Quaterniond is ”<<q.coeffs().transpose()<<std::endl;  //从旋转向量或旋转矩阵获取四元数，coeffs顺序是(x,y,z,w) |

## Eigen中四元数初始化

|  |
| --- |
| Eigen::Quaterniond q1(w,x,y,z);  Eigen::Quaterniond q2(Vector4d(x,y,z,w));  Eigen::Quaterniond q3(Matrix3d(R)); |

## eigen中的欧拉角

|  |
| --- |
| eulerAngles()函数说明  Eigen中的eulerAngles()函数，可以将旋转矩阵转换为欧拉角的形式，同一个旋转矩阵因为设置参数的不同，可以得到不同数值的欧拉角，这与旋转的顺序及类型有关。  旋转的类型分为：定轴（外旋）和动轴（内旋），定轴运动是左乘运算，动轴运动是右乘运算。  Eigen::vector3d eulerAngles(Index a0, Index a1, Index a2)，其中参数a0、a1、a2 用 0、1、2表示旋转轴，其中0表示X轴，1表示Y轴，2表示Z轴。此外，a0 表示首先选择的轴，a1 表示其次旋转的轴，a2 表示最后旋转的轴。因此，输出结果的顺序与轴旋转的顺序相同。  例如：  Eigen::Vector3d eulerAngle = rot.eulerAngles(0,1,2);  这里， eulerAngle的存储顺序为：绕x轴的角度、绕y轴的角度、绕z轴的角度  旋转的顺序为：先绕x轴、再绕y轴、最后绕z轴（动轴）  计算旋转矩阵的连乘顺序为：R\_matrix = Rx \* Ry \* Rz（右乘）  Eigen::Vector3d eulerAngle = rot.eulerAngles(2,1,0);  这里，eulerAngle的存储顺序为：绕z轴的角度、绕y轴的角度、绕x轴的角度  旋转的顺序为：先绕z轴、再绕y轴、最后绕x轴（动轴）  计算旋转矩阵的连乘顺序为：R\_matrix = Rz \* Ry \* Rx（右乘） |

## ros中的欧拉角

|  |
| --- |
| ros中坐标系为前左上坐标系，绕x轴为roll角，绕y轴为pitch角，绕z轴为yaw角。给定一个tf::Matrix3x3类型的旋转矩阵R。R.getRPY(roll,pitch,yaw)获取欧拉角。    经过验证  (左乘) |

## 四元数运算

|  |
| --- |
| 取逆： |

## 协方差矩阵

|  |
| --- |
| 以运动员身高、体重、年龄数据为例，求协方差矩阵。 |

## 标定鱼眼相机

|  |
| --- |
| 一、matlab  使用matlab标定工具箱CameraCalibrator,导入拍摄照片。  Camera model选择fisheye模型；  设置棋盘格尺寸；  勾选option下的estimate alignment；  开始标定。  二、opencv源码标定  三、kalibr标定  kalibr\_calibrate\_cameras --target ~/cal/apriltags.yaml --bag ~/cal/cameras\_calibration.bag --models omni-radtan omni-radtan --topics /fisheye1 /fisheye2  四、basalt标定  basalt\_calibrate --dataset-path ~/cal/cameras\_calibration.bag --dataset-type bag --aprilgrid ~/cal/apriltags.json --result-path ~/cal/ --cam-types ds ds ds ds  五、ros在线标定  rosrun camera\_calibration cameracalibrator.py --size 9x6 –sqare 0.021 image:=/camera/left/image\_raw  六、OCAMCALIB  AMD一篇论文(VIPS-Odom: Visual-Inertial Odometry Tightly-coupled with Parking Slots for Autonomous Parking)中使用该工具箱标定鱼眼相机。    网址：<https://sites.google.com/site/scarabotix/ocamcalib-omnidirectional-camera-calibration-toolbox-for-matlab>  论文：A Toolbox for Easily Calibrating Omnidirectional Cameras |

## 分布式训练

|  |
| --- |
| 模型并行  模型并行主要应用于模型相比显存来说更大，一块 GPU 无法加载的场景，通过把模型切割为几个部分，分别加载到不同的 GPU 上，来进行训练  数据并行  这个是日常会应用的比较多的情况。即每个 GPU 复制一份模型，将一批样本分为多份分发到各个GPU模型并行计算。因为求导以及加和都是线性的，数据并行在数学上也有效。采用数据并行相当于加大了batch\_size，得到更准确的梯度或者加速训练。  常用的 API 有两个：  torch.nn.DataParallel(DP)  torch.nn.DistributedDataParallel(DDP)  DP 相比 DDP 使用起来更友好（代码少），但是 DDP 支持多机多卡，训练速度更快，而且负载相对要均衡一些。常优先选用 DDP 。  原文链接：https://blog.csdn.net/ytusdc/article/details/122091284 |

## TCP和UDP传输协议

|  |
| --- |
| TCP:  建立连接过程：    三次握手为了防止出现失效的连接请求报文段被服务端接收的情况，从而产生错误。  断开连接过程：    Receiver没有接受到ACK时会重发FIN，因此Initiator需要延时等待。  特点：  **面向连接：**面向连接，是指发送数据之前必须在两端建立连接。建立连接的方法是“三次握手”，这样能建立可靠的连接。建立连接，是为数据的可靠传输打下了基础。  **仅支持单播传输**：每条TCP传输连接只能有两个端点，只能进行点对点的数据传输，不支持多播和广播传输方式。  **面向字节流**：TCP不像UDP一样那样一个个报文独立地传输，而是在不保留报文边界的情况下以字节流方式进行传输。  **可靠传输：**对于可靠传输，判断丢包，误码靠的是TCP的段编号以及确认号。TCP为了保证报文传输的可靠，就给每个包一个序号，同时序号也保证了传送到接收端实体的包的按序接收。然后接收端实体对已成功收到的字节发回一个相应的确认(ACK)；如果发送端实体在合理的往返时延(RTT)内未收到确认，那么对应的数据（假设丢失了）将会被重传。  **提供拥塞控制**：当网络出现拥塞的时候，TCP能够减小向网络注入数据的速率和数量，缓解拥塞  UDP：  **面向无连接：**首先 UDP 是不需要和 TCP一样在发送数据前进行三次握手建立连接的，想发数据就可以开始发送了。并且也只是数据报文的搬运工，不会对数据报文进行任何拆分和拼接操作。性能损耗少、资源占用少。  具体来说就是：在发送端，应用层将数据传递给传输层的 UDP 协议，UDP 只会给数据增加一个 UDP 头标识下是 UDP 协议，然后就传递给网络层了  在接收端，网络层将数据传递给传输层，UDP 只去除 IP 报文头就传递给应用层，不会任何拼接操作  **有单播，多播，广播的功能：**UDP 不止支持一对一的传输方式，同样支持一对多，多对多，多对一的方式，也就是说 UDP 提供了单播，多播，广播的功能。  **UDP是面向报文的：**发送方的UDP对应用程序交下来的报文，在添加首部后就向下交付IP层。UDP对应用层交下来的报文，既不合并，也不拆分，而是保留这些报文的边界。因此，应用程序必须选择合适大小的报文  **不可靠性：**首先不可靠性体现在无连接上，通信都不需要建立连接，想发就发，这样的情况肯定不可靠。          并且收到什么数据就传递什么数据，并且也不会备份数据，发送数据也不会关心对方是否已经正确接收到数据了。          网络环境时好时坏，但是 UDP 因为没有拥塞控制，一直会以恒定的速度发送数据。即使网络条件不好，也不会对发送速率进行调整。这样实现的弊端就是在网络条件不好的情况下可能会导致丢包，但是优点也很明显，在某些实时性要求高的场景（比如电话会议）就需要使用 UDP 而不是 TCP。 |

## 不同三维变换的矩阵形式

|  |
| --- |
| 1、欧式变换  相当于对一个刚体进行移动和旋转，而不改变其自身的形状。  2、相似变换  比欧式变换多了一个自由度，对物体进行均匀的缩放。  3、仿射变换  仿射投影也叫正交投影。立方体经仿射变换后不再是立方体，但是各个面仍然是平行四边形。  4、射影变换  真实世界到相机照片的变换是一个射影变换，如果相机的焦距为无穷远，那么这个变换为仿射变换。 |

## 视觉slam中常见的矩阵

|  |
| --- |
| 1. 本质矩阵   本质矩阵描述的是两个相机坐标系下匹配点的关系  （1）3\*3且自由度为5的矩阵  （2）因为只包含R，t共有6个自由度，又因为尺度等价去掉一个自由度  （3）本质矩阵E的奇异值必定为的形式   1. 基础矩阵   基础矩阵描述的是两个图像坐标系下匹配点的关系    为世界坐标系中的点，和分别为世界坐标系中的点在两个相机归一化平面的投影。和为两个相机归一化平面上的极点，和为两个相机归一化平面上的极线。为基线。  在上可得其中直线的表达式为，  由点和极点确定，有  其中符号为叉乘的矩阵形式  由于直线方程左右同乘以一个系数仍表示同一条直线，这里不用“=”而使用“~”。    可知另一时刻相机归一化平面上的极线      即     1. 单应矩阵   **单应矩阵**是射影几何中的一个术语，又称之为射影变换。本质上是一个数学概念，一般所说的单应矩阵是平面上的单应性矩阵，主要用来解决两个问题：  （1）表述真实世界中一个平面与他对应图像的透视变换  （2）通过透视变换实现图像从一个视图变换到另一个视图的转换。 |

## 李群与李代数

|  |
| --- |
| 在复平面上，对一个复数左乘一个单位复数，相当于对这个复数表示的点进行旋转。      特殊正交群SO(n)，旋转矩阵群，SO(2)和SO(3)最常见。  特殊欧式群SE(n)，n维欧式变换，SE(2)和SE(3)最常见。  李群是指具有连续(光滑)性质的群。每个李群都有对应的李代数，描述了李群在局部的性质。    李代数中的运算是李括号，三维向量的叉乘是一种李括号。g=(R3, R, ×)构成了一个李代数。  特殊正交群SO(3)：旋转矩阵R  对应的李代数：so(3)=  指数映射：    特殊欧式群SE(3)：  对应的李代数：se(3)  指数映射： |

## Octomap

|  |
| --- |
| 八叉树：    八叉树的叶子节点n状态更新，依据三个因素：   1. 先验概率，假设为0.5 2. 先前估计 3. 当前估计   占据概率更新函数    使用对数表达可写为    其中    移动机器人经常面临环境中的临时或永久变化，因此引入上下界，将占用概率约束在指定范围内    更新后的占用概率大于上限则取上限，低于下限则取下限，位于二者之间取计算值。  数据结构如图： |

## Ceres优化

|  |
| --- |
| Ceres提供了三种求导方法，分别是：解析求导、自动求导、数值求导。  自动求导需要重载括号运算符。  在SLAM中，使用的一般都是解析求导，需要自己填入雅克比函数。  数值求导（Numeric derivatives）核心思想是，当增量很小时，可以采用 近似求导，由此我们不需要实际计算导数的表达式，也可以从数值上进行近似。  数值求导方法NumericDiffMethodType又具体分为：前向求导、中心求导和Ridders方法，对应的精度和耗时依次增加。官方建议在不考虑时间约束时采用Ridders方法，中心求导是一个均衡的选择。  ceres::AutoDiffCostFunction<xxxError,6,4,3,4,3>尖括号中的数字依次为残差维数，参数一维数，参数二维数… |

## 各种距离公式

|  |
| --- |
| a) 欧式距离 “euclidean”  b) 曼哈顿距离 “manhattan”  c) 切比雪夫距离“chebyshev”  d) 闵可夫斯基距离 “minkowski”  e) 带权重闵可夫斯基距离 “wminkowski”  f) 标准化欧式距离 “seuclidean”: 即对于各特征维度做了归一化以后的欧式距离。此时各样本特征维度的均值为0，方差为1.  g) 马氏距离“mahalanobis”：当样本分布独立时，马氏距离等同于欧式距离。 |

## CAN总线协议

|  |
| --- |
| CAN总线，全称为Controller Area Network，即控制器局域网，CAN 总线最初是由德国电气商博世公司开发，其最初动机就是为了解决现代汽车中庞大的电子控制系统之间的通讯，减少不断增加的信号线。因此设计了一个单一的网络总线，所有的外围器件可以被挂接在该总线上。    CAN总线通常为双绞线，两端有防信号反射和信号衰减的120Ω终端电阻。CAN总线传输的电平信号是差分信号，它使用两条线（CAN\_H和CAN\_L）来传输数据。总线电平分为显性电平和隐性电平。显性电平为逻辑 0，隐性电平为逻辑 1。差分信号的一个主要优点是它对外部电磁干扰具有较好的抗干扰能力。CAN为半双工，同一时间只可以单向通信。  终端电阻的阻值取决于线缆的阻抗。任何一根线缆的特征阻抗都可以通过实验的方式得出。线缆的一端接方波发生器，另一端接一个可调电阻，并通过示波器观察电阻上的波形。调整电阻阻值的大小，直到电阻上的信号是一个良好的无振铃的方波，此时的电阻值可以认为与线缆的特征阻抗一致。  在CAN总线中使用的典型线缆，将它们扭制成双绞线，就可根据上述方法得到特征阻抗大约为120Ω，这也是CAN标准推荐的终端电阻阻值，所以这个120Ω是测出来的，不是算出来的，都是根据实际的线束特性进行计算得到的。  **多主方式：**可以多主方式工作，网络上任意一个节点均可以在任意时刻主动地向网络上的其他节点发送信息，而不分主从，通信方式灵活。  **系统柔软性：**由于CAN总线的设计允许节点自主决定何时发送数据，这使得系统能够灵活地适应不同的通信需求和条件。  **速度快，距离远：**CAN总线支持较高的数据传输速率，并且可以在较长的距离上稳定传输数据，这得益于其高效的通信协议和抗干扰能力。  **具有错误检测、错误通知和错误恢复功能：**CAN总线使用循环冗余校验（CRC）等机制来检测数据在传输过程中是否出现错误。当检测到错误时，节点会发送错误帧，以通知网络上的其他节点。在检测到错误后，系统可以采取相应的措施，如重传数据，以确保数据的正确传输。  **故障封闭功能：**CAN总线能够将故障节点隔离，防止故障扩散到整个网络，从而保证系统的稳定性。  **连接节点多：**CAN总线理论上可以支持多达110个节点，这使得它非常适合大规模的网络通信。 |

## I2C总线协议

|  |
| --- |
| CAN优点就是充分考虑了抗干扰设计，常用于不同大模块之间的单片机通信。IIC(Inter-Integrated Circuit)一般用于同一个电路板上面不同简单传感器和模块的通信，优点是布线简单硬件开发成本低。  IIC一共有只有两个总线： 一条是双向的串行数据线ＳＤＡ，一条是串行时钟线ＳＣＬ。SDA(Serial data)是数据线，D代表Data也就是数据，Send Data 也就是用来传输数据的。SCL(Serial clock line)是时钟线，C代表Clock 也就是时钟，用于控制数据发送时序。IIC与CAN一样也是半双工。 |