

第一章 作业

二 熟悉Linux

1. 如何在Ubuntu 中安装软件（命令行界面）？它们通常被安装在什么地方？

Ubuntu 可通过如下命令安装库

```
1 | sudo apt-get install package
```

通过apt方式安装的库一般被安装在目录 `usr/lib` 或者 `usr/include` 中。

补充：

下载的软件存放位置: `/var/cache/apt/archives`

安装后软件默认位置: `/usr/share`

可执行文件位置: `/usr/bin`

配置文件位置: `/etc`

lib文件位置: `/usr/lib`

2. linux 的环境变量是什么？我如何定义新的环境变量？

环境变量用来定义每个用户的操作环境。linux环境变量和Windows的环境变量一样，分系统环境变量和用户环境变量，系统环境变量对所有用户有效，而用户环境变量只对当前用户有效。

以路径 `$PATH:/opt/au1200_rm/build_tools/bin` 为例，定义环境变量的方法如下：

方法一：用**export**命令，

1. 输入 `"export PATH="$PATH:/opt/au1200_rm/build_tools/bin"`。

方法二：修改**profile**文件，

1. 输入 `sudo gedit /etc/profile`，回车；
2. 再输入 `"export PATH="$PATH:/opt/au1200_rm/build_tools/bin"`。

方法三：修改**.bashrc**文件，

1. 输入 `sudo gedit /root/.bashrc`，回车。
2. 再输入 `"export PATH="$PATH:/opt/au1200_rm/build_tools/bin"`。

3. linux 根目录下面的目录结构是什么样的？至少说出3 个目录的用途

Linux根目录结构及用途

1. **bin 目录**：此目录存放所有二进制命令（用户）
2. **boot目录**：Linux内核及引导系统程序所需的目录
3. **dev目录**：所有设备文件的目录（如声卡、磁盘、光驱）
4. **etc目录**：二进制安装包（yum，rpm）配置文件默认路径，服务启动命令存放目录
5. **lib目录**：库文件存放目录
6. **home目录**：普通用户的家目录默认数据存放目录
7. **opt目录**：自定义软件安装存放目录，用户自行安装的软件包存放目录。

8. **lost+found目录**: 在EXT3系统中, 当系统意外崩溃或意外关机时, 会产生一些碎片文件在这个目录下面, 系统启动fsck工具会检查这个目录, 并修复已损坏的文件系统。
9. **mnt目录**: 用于临时挂载存储设备, 通常情况下可以挂载LINUX ISO光盘进行无网条件下的安装其他软件包
10. **proc目录**: 进程及内核信息存放目录
11. **root目录**: 管理的家目录
12. **sbin目录**: 系统管理员命令存放的目录 (超级管理员使用的命令)
13. **tmp目录**: 临时文件目录, 程序运行时产生的临时文件存放目录
14. **usr目录**: 系统存放程序的目录 (命令和帮助文件)

4. 假设我要给a.sh 加上可执行权限, 该输入什么命令?

为一个文件添加可执行权限

```
1 | chmod +x a.sh
```

*: 为一个文件夹下的所有文件添加可执行权限

```
1 | chmod +x *
```

5. 假设我要将a.sh文件的所有者改成xiang:xiang, 该输入什么命令?

可利用如下命令:

```
1 | sudo chown -R username:groupname filename
```

解释

sudo: 管理员权限

chown : 修改文件所有者和组别

-R : 递归文件夹内部的所有文件及文件夹

username: 目标所有者名称

groupname: 组名

filename: 文件或文件夹名称

答案:

```
1 | sudo chown -R xiang:xiang a.sh
```

三 SLAM综述文献阅读

1. SLAM 会在哪些场合中用到? 至少列举三个方向

答: SLAM在以下方向可能会有非常大的用处

1. **增强现实应用:** AR通过电脑技术, 将虚拟的信息应用到真实世界, 真实的环境和虚拟的物体实时地叠加到了同一个画面或空间同时存在。这一画面的实现, 离不开SLAM技术的实时定位。虽然在AR行业有很多可代替技术, 但SLAM技术是最理想的定位导航技术。相较于SLAM在机器人、无人驾驶等领域的应用, 在AR行业的应用则有很多不同点。
 - 精度: AR一般更关注于局部精度, 要求恢复的相机运动避免出现漂移、抖动, 这样叠加的虚拟物体才能看起来与现实场景真实地融合在一起。但在机器人和无人驾驶领域则一般更关注全局精度, 需要恢复的整条运动轨迹误差累积不能太大, 循环回路要能闭合, 而在某个局部的漂移、抖动等问题往往对机器人应用来说影响不大。
 - 效率: AR需要在有限的计算资源下实时求解, 人眼的刷新率为24帧, 所以AR的计算效率通常需要到达30帧以上; 机器人本身运动就很慢, 可以把帧率降低, 所以对算法效率的要求相对较低。
 - 配置: AR对硬件的体积、功率、成本等问题比机器人更敏感, 比如机器人上可以配置鱼眼、双目或深度摄像头、高性能CPU等硬件来降低SLAM的难度, 而AR应用更倾向于采用更为高效、鲁棒的算法达到需求。
2. **无人机:** 无人机在飞行的过程中需要知道哪里有障碍物, 该怎么规避, 怎么重新规划路线。显然, 这是SLAM技术的应用。但无人机飞行的范围较大, 所以对精度的要求不高, 市面上其他的一些光流、超声波传感器可以作为辅助。
3. **自动驾驶:** 随着城市物联网和智能系统的完善, 无人驾驶必是大势所趋。无人驾驶利用激光雷达传感器 (Velodyne、IBEO等) 作为工具, 获取地图数据, 并构建地图, 规避路程中遇到的障碍物, 实现路径规划。跟SLAM技术在机器人领域的应用类似, 只是相比较于SLAM在机器人中的应用, 无人驾驶的雷达要求和成本要明显高于机器人。
4. **机器人:** 激光+SLAM是目前机器人自主定位导航所使用的主流技术。激光测距相比较于图像和超声波测距, 具有良好的指向性和高度聚焦性, 是目前最可靠、稳定的定位技术。激光雷达传感器获取地图信息, 构建地图, 实现路径规划与导航。机器人可以应用的地方很多, 比如配送机器人, 探索机器人等。
5. **智能家居:** 近年来, 智能家居的发展使得我们的居家生活变得更加充满乐趣与效率。对于智能清扫机器人这类室内移动型机器人而言, 机器人自主定位和对周边环境的识别是其高效工作的根本。利用激光或相机进行SLAM在智能家居机器人上有很大的应用空间。

2. SLAM 中定位与建图是什么关系? 为什么在定位的同时需要建图?

答: SLAM全称是simultaneous localization and mapping (即时定位与建图)。假设了机器人从未知环境中的未知地点出发, 在运动过程中通过重复观测到的地图特征 (比如, 墙角, 柱子等) 定位自身位置和姿态, 再根据自身位置增量式的构建地图, 从而达到同时定位和地图构建的目的。

定位与建图的关系

整个SLAM过程中, 移动机器人一方面要明白自身的状态 (即定位), 另一方面又需要了解外在的环境 (即建图)。两者紧密相关。建图的准确性依赖于定位精度, 而定位的实现又离不开精确的建图。

为什么定位的同时需要建图

答: SLAM强调在未知环境下进行整个过程, 如果在未知环境下进行定位, 首先需要能够识别并理解周围的环境。再利用环境中的外部信息作为定位的基准, 所以需要对所处的环境进行建图。

3. SLAM 发展历史如何? 我们可以将它划分成哪几个阶段?

答: SLAM发展历史

SLAM最早由Smith、Self和Cheeseman于1988年提出。由于其重要的理论与应用价值, 被很多学者认为是实现真正全自主移动机器人的关键。在文献2 (Past, Present, and Future of Simultaneous Localization And Mapping: Towards the Robust-Perception Age) 中, 根据Durrant-Whyte和Bailey在两篇综述中对SLAM的前20年历史进行的详尽回顾。SLAM的发展可以大致分为三个阶段:

1. **朴素时期: classical age** (1986-2004) 。classical age出现了SLAM最基本的概率公式, 包括基于扩展卡尔曼滤波器(EKF), Rao-Blackwellised粒子滤波器和最大似然估计的方法。此外, 它还剔除了与效率以及鲁棒数据等相关的基本问题, 这也是日后的发展方向。
2. **算法分析时期: algorithmic-analysis age** (2004-2015) , 这一时期研究了SLAM的基本特性, 包括可观察性, 收敛性, 一致性以及稀疏性对SLAM高效求解的关键作用, 主要的开源SLAM库在这个时期得到了开发。
3. **鲁棒性-预测性时代: robust-perception** (2015-至今) : 这一时期主要探索SLAM在位置环境中对鲁棒性、高级别的场景理解, 计算资源优化, 任务驱动的环境感知等。

视觉SLAM是在传统SLAM的基础上发展起来的, 早期的视觉SLAM多采用扩展卡尔曼滤波等手段来优化相机位姿的估计和地图构建的准确性, 后期随着计算能力的提升及算法的改进, BA优化、位姿优化等手段逐渐成为主流。随着人工智能技术的普及, 基于深度学习的SLAM越来越受到研究者的关注。

4. 列举三篇在SLAM 领域的经典文献。

0. On the Representation and Estimation of Spatial Uncertainty[0]: 公认的SLAM开山之作

1. MonoSLAM[1]: real-time single camera SLAM[1]: 第一个实时的单目视觉SLAM系统
2. ORB-SLAM2[2]: 当前应用最多基于优化的视觉SLAM方法, 系统框架非常完善
3. MSCKF[3]: 基于滤波的视觉SLAM方法, 由于优秀的计算量和精度得到了实际应用 (AR Kit)

[0] Randall, C, Smith,等. On the Representation and Estimation of Spatial Uncertainty[J]. The International Journal of Robotics Research, 1986.

[1] Davison A J , Reid I D , Molton N D , et al. MonoSLAM: real-time single camera SLAM[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2007, 29(6):1052-1067.

[2] Mur-Artal R , Tardos J D . ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras[J]. IEEE Transactions on Robotics, 2016, 33(5):1255-1262.

[3]Mourikis, Anastasios /I, Roumeliotis, Stergios /I. A Multi-State Constraint Kalman Filter for Vision-aided Inertial Navigation[C]// IEEE International Conference on Robotics & Automation. IEEE, 2007.

四. CMake 练习

书写一个由 cmake 组织的 C++ 工程, 要求如下:

1. include/hello.h 和 src/hello.c 构成了 libhello.so 库。hello.c 中提供一个函数 sayHello(), 调用此函数时往屏幕输出一行“Hello SLAM”。我们已经为你准备了 hello.h 和 hello.c 这两个文件, 见“code/”目录下。
2. 文件 useHello.c 中含有一个 main 函数, 它可以编译成一个可执行文件, 名为“sayhello”。
3. 默认用 Release 模式编译这个工程。
4. 如果用户使用 sudo make install, 那么将 hello.h 放至/usr/local/include/下, 将 libhello.so 放至/usr/local/lib/下。请按照上述要求组织源代码文件, 并书写CMakeLists.txt。

答: 编译运行截图如下:

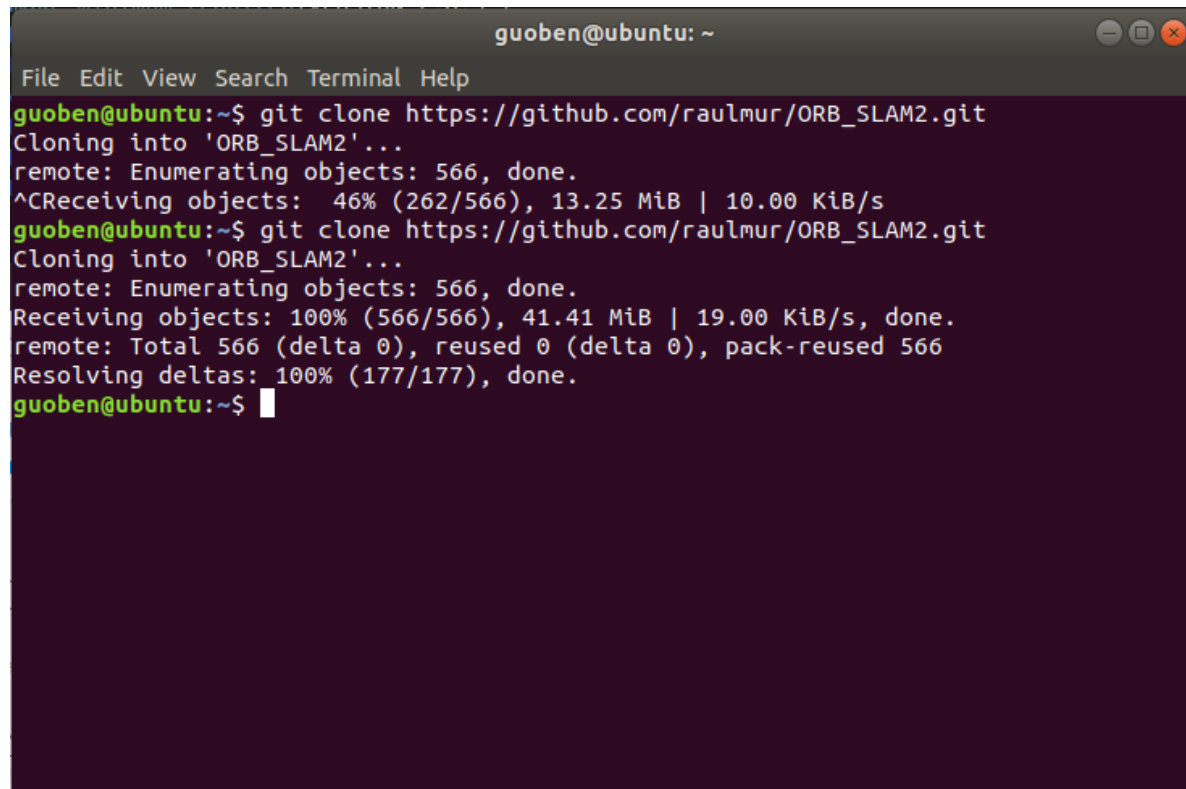
```
guoben@ubuntu:~/Project/Hello/build$ cmake ..
-- Configuring done
-- Generating done
-- Build files have been written to: /home/guoben/Project/Hello/build
guoben@ubuntu:~/Project/Hello/build$ make
[ 50%] Built target HELLO
[100%] Built target usehello
guoben@ubuntu:~/Project/Hello/build$ cd ../bin/
guoben@ubuntu:~/Project/Hello/bin$ ./usehello
Hello SLAM
```

五. 理解ORB-SLAM2 框架

1. 从github.com 下载ORB-SLAM2 的代码。

地址在: https://github.com/raulmur/ORB_SLAM2.

提示: 在安装git 之后, 可以用git clone https://github.com/raulmur/ORB_SLAM2 命令下载ORB-SLAM2。下载完成后, 请给出终端截图。



```
guoben@ubuntu: ~
File Edit View Search Terminal Help
guoben@ubuntu:~$ git clone https://github.com/raulmur/ORB_SLAM2.git
Cloning into 'ORB_SLAM2'...
remote: Enumerating objects: 566, done.
^CReceiving objects: 46% (262/566), 13.25 MiB | 10.00 KiB/s
guoben@ubuntu:~$ git clone https://github.com/raulmur/ORB_SLAM2.git
Cloning into 'ORB_SLAM2'...
remote: Enumerating objects: 566, done.
Receiving objects: 100% (566/566), 41.41 MiB | 19.00 KiB/s, done.
remote: Total 566 (delta 0), reused 0 (delta 0), pack-reused 566
Resolving deltas: 100% (177/177), done.
guoben@ubuntu:~$
```

2. 阅读ORB-SLAM2 代码目录下的CMakeLists.txt, 回答问题:

(a) ORB-SLAM2 将编译出什么结果? 有几个库文件和可执行文件?

```
1 project(ORB_SLAM2)
2 add_library(${PROJECT_NAME} SHARED
3   src/System.cc
4   src/Tracking.cc
5   src/LocalMapping.cc
6   src/LoopClosing.cc
7   src/ORBextractor.cc
8   src/ORBmatcher.cc
9   src/FrameDrawer.cc
10  src/Converter.cc
```

```

11 src/MapPoint.cc
12 src/KeyFrame.cc
13 src/Map.cc
14 src/MapDrawer.cc
15 src/Optimizer.cc
16 src/PnP solver.cc
17 src/Frame.cc
18 src/KeyFrameDatabase.cc
19 src/Sim3Solver.cc
20 src/Initializer.cc
21 src/Viewer.cc
22 )

```

根据以上CMakeLists.txt的描述，ORB_SLAM2编译将生成 `libORB_SLAM2.so` **动态链接库**

```

1  # Build examples
2
3  set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${PROJECT_SOURCE_DIR}/Examples/RGB-D)
4
5  add_executable(rgbd_tum
6  Examples/RGB-D/rgbd_tum.cc)
7  target_link_libraries(rgbd_tum ${PROJECT_NAME})
8
9  set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${PROJECT_SOURCE_DIR}/Examples/Stereo)
10
11 add_executable(stereo_kitti
12 Examples/Stereo/stereo_kitti.cc)
13 target_link_libraries(stereo_kitti ${PROJECT_NAME})
14
15 add_executable(stereo_euroc
16 Examples/Stereo/stereo_euroc.cc)
17 target_link_libraries(stereo_euroc ${PROJECT_NAME})
18
19
20 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
21 ${PROJECT_SOURCE_DIR}/Examples/Monocular)
22
23 add_executable(mono_tum
24 Examples/Monocular/mono_tum.cc)
25 target_link_libraries(mono_tum ${PROJECT_NAME})
26
27 add_executable(mono_kitti
28 Examples/Monocular/mono_kitti.cc)
29 target_link_libraries(mono_kitti ${PROJECT_NAME})
30
31 add_executable(mono_euroc
32 Examples/Monocular/mono_euroc.cc)
33 target_link_libraries(mono_euroc ${PROJECT_NAME})

```

根据以上部分CMakeLists.txt的描述，ORB_SLAM2还将生成 `rgbd_tum`, `stereo_kitti`, `stereo_euroc`, `mono_tum`, `mono_kitti`, `mono_euroc` 共6个可执行的**demo程序**。

库文件有19个：

```

1  add_library(${PROJECT_NAME} SHARED

```

```
2  src/System.cc
3  src/Tracking.cc
4  src/LocalMapping.cc
5  src/LoopClosing.cc
6  src/ORBextractor.cc
7  src/ORBmatcher.cc
8  src/FrameDrawer.cc
9  src/Converter.cc
10 src/MapPoint.cc
11 src/KeyFrame.cc
12 src/Map.cc
13 src/MapDrawer.cc
14 src/Optimizer.cc
15 src/PnPsolver.cc
16 src/Frame.cc
17 src/KeyFrameDatabase.cc
18 src/Sim3Solver.cc
19 src/Initializer.cc
20 src/Viewer.cc
21 )
```

可执行文件有如下6个：

1. `rgbdtum`
2. `stereo_kitti`
3. `stereo_euroc`
4. `mono_tum`
5. `mono_kitti`
6. `mono_euroc`

(b) ORB-SLAM2 中的include, src, Examples 三个文件夹中都含有什么内容？

- **include**：存放头文件
- **src**：用来存放 `.cc` 等库文件
- **Example**：存放运行例子的可执行文件，包含了针对不同数据集的运行文件，：
 - 基于TUM数据集的深度相机运行文件
 - 基于TUM数据集的单目相机运行例程
 - 基于EuRoc数据集的单目相机运行例程
 - 基于EuRoc数据集的双目相机运行例程
 - 基于KITTI数据集的单目相机运行例程
 - 基于KITTI数据集的双目相机运行例程

(c) ORB-SLAM2 中的可执行文件链接到了哪些库？它们的名字是什么？

ORB-SLAM2中的可执行文件链接了五个第三方库：`OpenCV`、`EIGEN3`、`Pangolin`、`DBow2`、`g2o` 和一个生成库 `libORB_SLAM2.so`。

附加题六 使用摄像头或视频运行ORB-SLAM2

1. 编译ORB-SLAM2，请给出它编译完成的截图。

```
guoben@ubuntu: ~/Project/ORB_SLAM2
File Edit View Search Terminal Help
CMakeLists.txt:47 (FIND_PACKAGE)
This warning is for project developers. Use -Wno-dev to suppress it.

-- Configuring done
-- Generating done
-- Build files have been written to: /home/guoben/Project/ORB_SLAM2/Thirdparty/g2o/build
[100%] Built target g2o
Uncompress vocabulary ...
Configuring and building ORB_SLAM2 ...
mkdir: cannot create directory 'build': File exists
Build type: Release
-- Using flag -std=c++11.
-- Configuring done
-- Generating done
-- Build files have been written to: /home/guoben/Project/ORB_SLAM2/build
[ 62%] Built target ORB_SLAM2
[ 68%] Built target stereo_kitti
[ 93%] Built target stereo_euroc
[ 93%] Built target rgbd_tum
[ 93%] Built target mono_euroc
[ 93%] Built target mono_tum
[100%] Built target mono_kitti
guoben@ubuntu:~/Project/ORB_SLAM2$
```

2. 修改工程

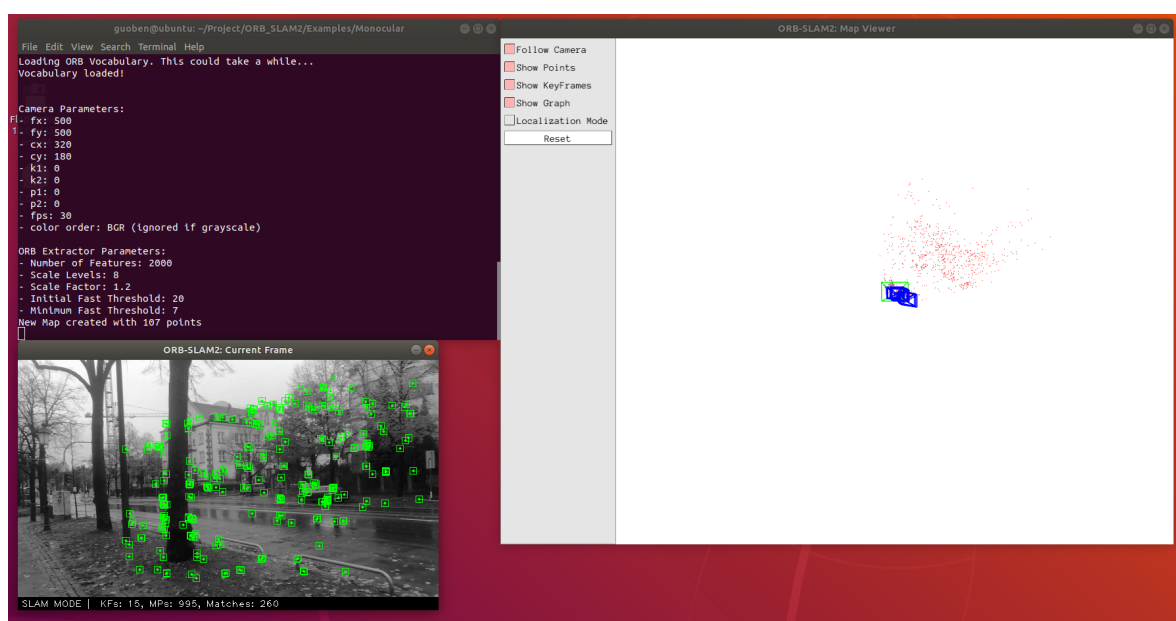
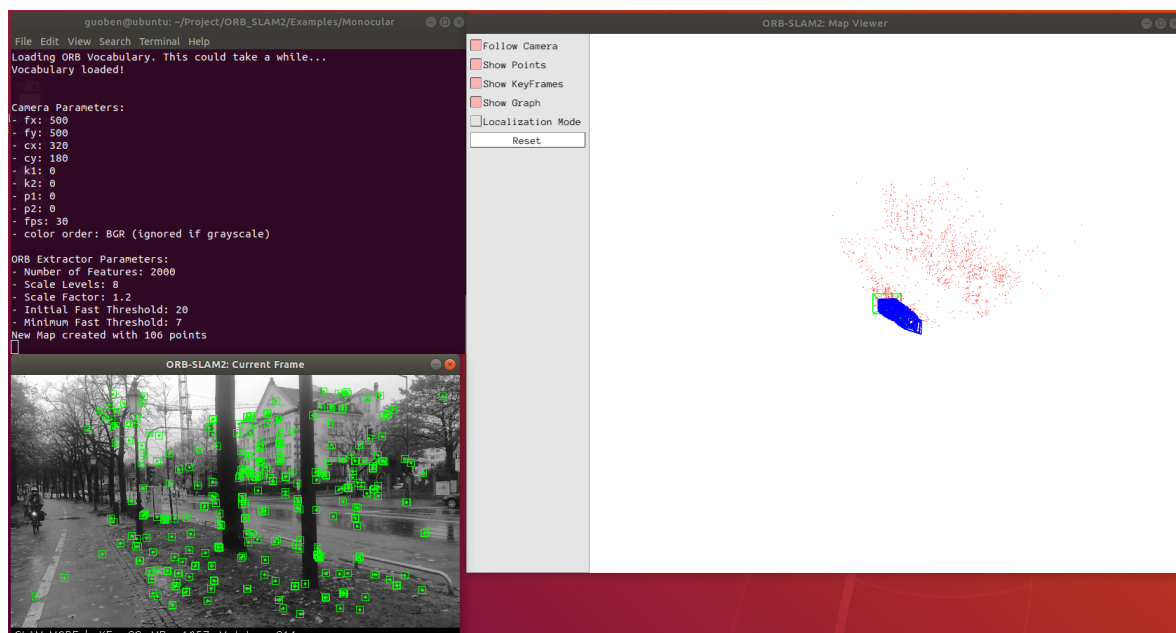
注意到，ORB-SLAM2 提供了若干数据集中的运行示例，这可以作为我们运行自己摄像头程序的参考，因为它们很相似。对于数据集上的示例，ORB-SLAM2 会首先读取数据集中的图像，再放到SLAM 中处理。那么对于我们自己的摄像头，同样可以这样处理。所以最方便的方案是直接将我们的程序作为一个新的可执行程序，加入到ORB-SLAM2 工程中。那么请问，如何将myslam.cpp或myvideo.cpp 加入到ORB-SLAM2 工程中？请给出你的CMakeLists.txt 修改方案。

答：修改方案：修改ORB_SLAM2的目录下CMakeLists

```
1 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${PROJECT_SOURCE_DIR}/Examples/myvideo)
2 add_executable(myvideo
3   Examples/myvideo/myvideo.cpp)
4 target_link_libraries(myvideo ${PROJECT_NAME})
```

3. 运行ORB_SLAM

现在你的程序应该可以编译出结果了。但是我们现在还没有谈相机标定，所以你还没办法标定你的摄像头。我们可以用一个不那么好的标定参数，先来试一试效果（所幸 ORB-SLAM2对标定参数不太敏感）。我为你提供了一个myslam.yaml（myvideo.yaml），这个文件是我们假想的标定参数。现在，用这个文件让 ORB-SLAM2 运行起来，看看 ORB-SLAM2 的实际效果吧。请给出运行截图，并谈谈你在运行过程中的体会。



体会

实验过程中，我使用的是虚拟机，运行内存为3G，ORB-SLAM2在运行myvideo的过程中，能够保持24~40的关键帧，同时追踪200个特征点，可见其能在计算量有限的情况下仍能追踪到特征点，非常强大。

对于一个不那么准确的标定文件，依然能展现出一定的结果，表明ORB-SLAM具有对于标定过程具有一定的鲁棒性。我猜想这可能是由于单目相机的单一性，不需要考虑多个传感器之间的协同。在我测试VINS的过程中则发现运行效果对标定的依赖性非常大。