**用python学习rgbd-slam**

最近开始学习ros，然后搜索slam教程，看到高翔大神写的《一起做rgbd-slam》系列，很有启发，也很佩服，但是高翔大神用的都是c++，本人比较喜欢python，所以想把文章中的代码改成python版本，也好记录一下自己的学习心得。

**原文链接：**

高博博客：[**http://www.cnblogs.com/gaoxiang12/tag/%E4%B8%80%E8%B5%B7%E5%81%9ARGB-D%20SLAM/**](http://www.cnblogs.com/gaoxiang12/tag/%E4%B8%80%E8%B5%B7%E5%81%9ARGB-D%20SLAM/)

ROSClub链接: <http://rosclub.cn/portal.php?mod=list&catid=9&page=7>

**时间规划：**

《一起做rgbd-slam》高博原文一共9篇，加上我个人水平有限，数学也不高，所以前期先做原文代码验证及其自己补充学习，有些可能写的不对，这个也在所难免，所以预计时间为一个月，大概从2017年2月10日-2017年3月30日，这里先立一个flag，万一呢。

**永久更新链接：**

ROSClub：<http://www.rosclub.cn>

相关代码及数据：<https://github.com/zsirui/slam-python.git>

**第一篇：从图像到点云（Python版）**

**原文链接：**

[**http://www.cnblogs.com/gaoxiang12/p/4652478.html**](http://www.cnblogs.com/gaoxiang12/p/4652478.html)

[**http://rosclub.cn/post-75.html**](http://rosclub.cn/post-75.html)

**0x00 准备**

**软件**：ubuntu14.04、ros-indigo-desktop-full、python2.7、pip等

**硬件**：asus xtion pro、电脑

**依赖库：**OpenCV**（强烈建议使用3.0以上版本，本教程所有代码使用的是OpenCV3.2.0版）**、Numpy、PCL

依赖库安装：

**pip**：**sudo apt-get install python-pip**

**OpenCV**：

（1）如果安装了ros，用自带的就可以

（2）编译安装OpenCV（推荐使用）

编译安装步骤（参考地址：<http://www.cnblogs.com/asmer-stone/p/5089764.html>）

1.依赖关系：

* GCC 4.4.x or later
* CMake 2.8.7 or higher
* Git
* GTK+2.x or higher, including headers (libgtk2.0-dev)
* pkg-config
* Python 2.6 or later and Numpy 1.5 or later with developer packages (python-dev, python-numpy)
* ffmpeg or libav development packages: libavcodec-dev, libavformat-dev, libswscale-dev
* [optional] libtbb2 libtbb-dev
* [optional] libdc1394 2.x
* [optional] libjpeg-dev, libpng-dev, libtiff-dev, libjasper-dev, libdc1394-22-dev

注：官方文档中虽然说其中一些依赖包是可选的，但是最好还是都装上，以防出问题。

以上依赖包可使用以下命令安装：

**sudo apt-get install build-essential**

**sudo apt-get install cmake git libgtk2.0-dev pkg-config libavcodec-dev libavformat-dev libswscale-dev**

**sudo apt-get install python-dev python-numpy libtbb2 libtbb-dev libjpeg-dev libpng-dev libtiff-dev libjasper-dev libdc1394-22-dev**

2.安装cmake-gui（用于生成OpenCV编译文件）：

**sudo apt-get install cmake-gui**

3.下载OpenCV源代码：

可以从OpenCV官网直接下载：<http://opencv.org/downloads.html>

也可以使用git命令行直接clone：

cd ~/<my\_working\_directory>//比如工作目录为opencv即，cd ~/opencv

**git clone** [**https://github.com/Itseez/opencv.git**](https://github.com/Itseez/opencv.git)

**git clone** [**https://github.com/Itseez/opencv\_contrib.git**](https://github.com/Itseez/opencv_contrib.git)

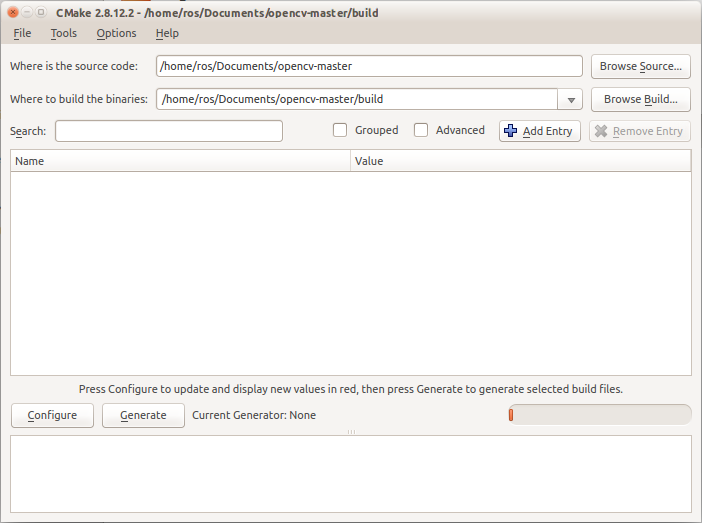
进入OpenCV源代码目录，创建build文件夹：

**cd ~/<my\_working\_directiory>/opencv**

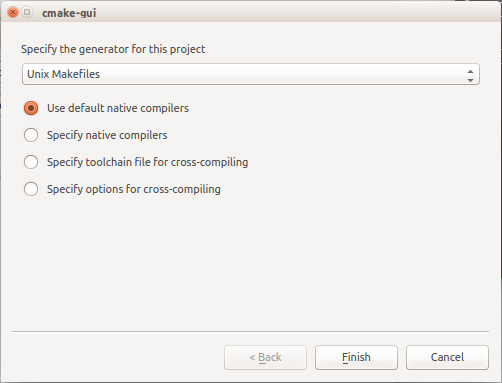
**mkdir build**

使用cmake-gui生成编译文件：

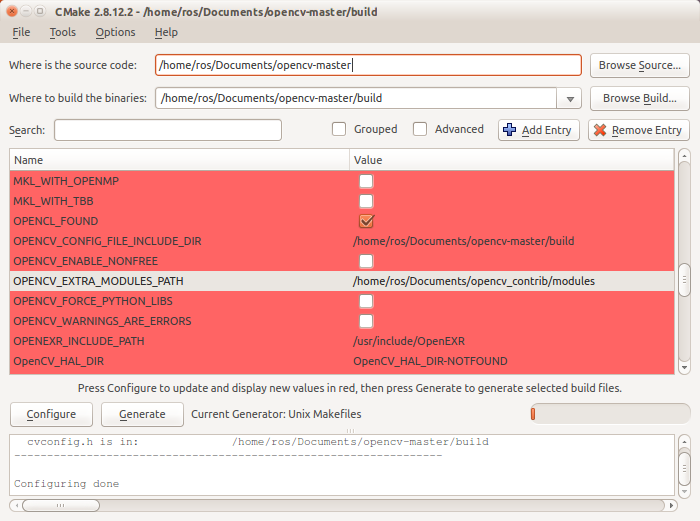
打开cmake-gui后，在source code栏选择OpenCV源代码所在目录（例如：/home/username/workplace/opencv），在build栏中填写刚才新建的build文件夹（例如：/home/username/workplace/opencv/build）如下图所示：



点击Configure按钮，会弹出如下图所示的提示框，点击Finish按钮即可：



接下来cmake会检查所有依赖库以及其他各可选部分版本，这个过程中cmake会下载一个名为ippicv的库，但经常会出现下载失败而导致configure失败，建议手动下载（下载地址：<https://raw.githubusercontent.com/Itseez/opencv_3rdparty/81a676001ca8075ada498583e4166079e5744668/ippicv/ippicv_linux_20151201.tgz>）后放入/home/username/workplace/opencv/3rdparty/ippicv/downloads/linux-808b791a6eac9ed78d32a7666804320e文件夹中。检查完毕后需要修改OPENCV\_EXTRA\_MODULES\_PATH内容，如下图所示：



修改完后再点击Configure按钮，如果依旧有红色高亮显示条目，则再次点击Configure按钮，等待所有条目均不再高亮显示后点击Generate按钮，退出cmake-gui程序，在命令行里进入/home/username/workplace/opencv/build目录，输入以下命令：

**make -j8**

**sudo make install**

**sudo ldconfig**

**[注] 建议勾选BUILD\_EXAMPLE选项，选用了不同的摄像头时需要先对摄像头进行标定，OpenCV示例中有提供标定程序。**

**Numpy**：**pip install numpy** (可能需要管理员权限)

**PCL**：先安装基本pcl库，再去git上下载python版pcl接口

步骤：（Ubuntu下，其他Linux发行版本安装办法参见：<http://pointclouds.org/downloads/linux.html>）

1. 通过PPA安装完整PCL库

**sudo add-apt-repository ppa:v-launchpad-jochen-sprickerhof-de/pcl**

**sudo apt-get update**

**sudo apt-get install libpcl-all**

1. 安装python-pcl

在github上下载python-pcl（地址：<https://github.com/strawlab/python-pcl>）

解压后进入python-pcl目录，输入命令：**sudo python setup.py install**即可，如果报错，可能缺少依赖库。使用命令**sudo pip install cython**安装python-pcl的依赖库后再执行安装命令即可。

在ROS中，我们输入以下命令可以让kinect运行：

**roslaunch openni\_launch openni.launch**

如果kinect连接在电脑上，可以通过输入下面命令：

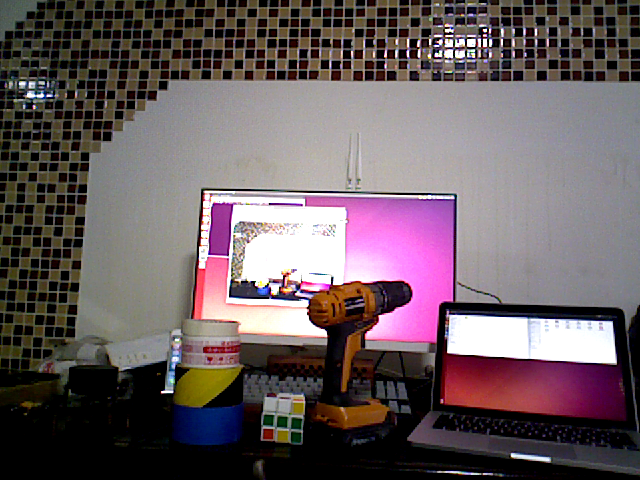
**rosrun image\_view image\_view image:=/camera/rgb/image\_color**

这样在启动起来的rviz界面中就可以看到kinect获取的图像信息

不过，如果手头没有kinect，可以使用下面提供的两张图片（来源：<http://www.cnblogs.com/gaoxiang12/p/4652478.html>）来作为测试数据使用，改组图片的相机内参矩阵为



或者使用自己手中深度摄像头获取的图片信息来作为测试数据使用（需要提前对摄像头进行标定）：

这里的两张图片中，左边是RGB数据图片，右边的是深度数据图片

接下来，我们将编写一段python脚本将这两张图片数据转换成3D点云数据。

**0x01 获取相机内参矩阵（标定相机）**

不同的深度摄像头具有不同的特征参数，在计算机视觉里面，将这组参数成为相机的内参矩阵***C***。格式为：

其中，***fx***，***fy***指相机在***x***轴和***y***轴上的焦距，***cx***，***cy***是相机的光圈中心，这组参数是摄像头生产制作之后就固定的。获取这组参数，只有通过相机标定。

首先，在一张A4大小的纸上打印86的黑白棋盘格作为标定板，使用摄像头拍摄不同角度的带有完整标定板的图片，数量在1020张之间，图片上一定要能够完整识别黑白格的边界交点。将OpenCV中编译好的示例程序中的cpp-example-calibration和cpp\_example-imagelist-creator的二进制程序（二进制程序保存在<opencv\_source>/build/bin下）复制到图片目录中，在终端中切换到图片目录，输入以下命令：

**./cpp\_example\_imagelist\_creator image\_list.xml \*.png**

**./cpp\_example\_calibration -w=7 -h=5 -n=12 -o=camera.yml -op -oe image\_list.xml**

第一条命令是用于生成标定程序所需的图片列表，图片信息保存在image\_list.xml中；第二条命令是根据图片列表进行标定，-w表示标定板横向有7个交点，-h表示纵向有5个交点，-n表示图片一共有12张，-o是输出文件保存在camera.yml中，-op和-oe是将检测到的点和相机外参一并输出到文件中。camera.yml文件内容如下（完整文件数据内容过长，省略部分数据）：

%YAML: 1.0

---

calibration\_time: "Wed 08 Mar 2017 09:35:01 AM CST"

nframes: 12

image\_width: 640

image\_height: 480

board\_width: 7

board\_height: 5

square\_size: 3.2869998931884766e+01

aspectRatio: 1.

flags: 2

camera\_matrix: !!opencv-matrix

rows: 3

cols: 3

dt: d

data: [ 6.0782982475382448e+02, 0., 3.6927587384670619e+02, 0.,

6.0782982475382448e+02, 2.0049685183455608e+02, 0., 0., 1. ]

distortion\_coefficients: !!opencv-matrix

rows: 5

cols: 1

dt: d

data: [ 2.6097424564484067e-01, -3.0826179906924500e-01,

-3.6561830159047480e-02, 2.8149494385205569e-02,

9.0272371325874165e-02 ]

avg\_reprojection\_error: 6.6092895845048028e-01

per\_view\_reprojection\_errors: !!opencv-matrix

rows: 12

cols: 1

dt: f

data: [ 1.12569833e+00, 3.25649649e-01, 3.52165848e-01,

2.45700657e-01, 2.78621674e-01, 5.25273621e-01, 7.99378633e-01,

1.01514637e+00, 6.92025602e-01, 5.77147245e-01, 8.56786847e-01,

3.39320183e-01 ]

extrinsic\_parameters: !!opencv-matrix

rows: 12

cols: 6

dt: d

data: [ -8.6337003636994508e-02, -1.1783451602046391e+00,

2.7303741738649485e+00, 7.2512051125174594e+01,

……

5.2296340497370135e-02, -3.3999347389601979e+02,

-1.0395036593678920e+02, 8.6211111700661570e+02 ]

image\_points: !!opencv-matrix

rows: 12

cols: 35

dt: "2f"

data: [ 4.20297913e+02, 1.73744476e+02, 3.97531860e+02,

1.79406143e+02, 3.75055542e+02, 1.85217300e+02, 3.52161591e+02,

……

2.22855743e+02, 2.41747681e+02, 2.25628357e+02, 2.64387665e+02,

2.28642029e+02 ]

这里，camera\_matrix就是我们所需要的相机内参。我们可以使用python的yaml（如果提示**ImportError: No module named yaml**，可以使用**sudo pip install yaml**安装）库对yml文件进行读取解析操作。代码如下：

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import yaml, sys

import numpy as np

def fixYamlFile(filename):

with open(filename, 'rb') as f:

lines = f.readlines()

if lines[0] != '%YAML 1.0\n':

lines[0] = '%YAML 1.0\n'

for line in lines:

if ' !!opencv-matrix' in line:

lines[lines.index(line)] = line.split(' !!opencv-matrix')[0] + '\n'

with open(filename, 'wb') as fw:

fw.writelines(lines)

def parseYamlFile(filename):

fixYamlFile(filename)

f = open(filename)

x = yaml.load(f)

f.close()

arr = np.array(x['camera\_matrix']['data'], dtype = np.float32)

return (arr.reshape(3, 3))

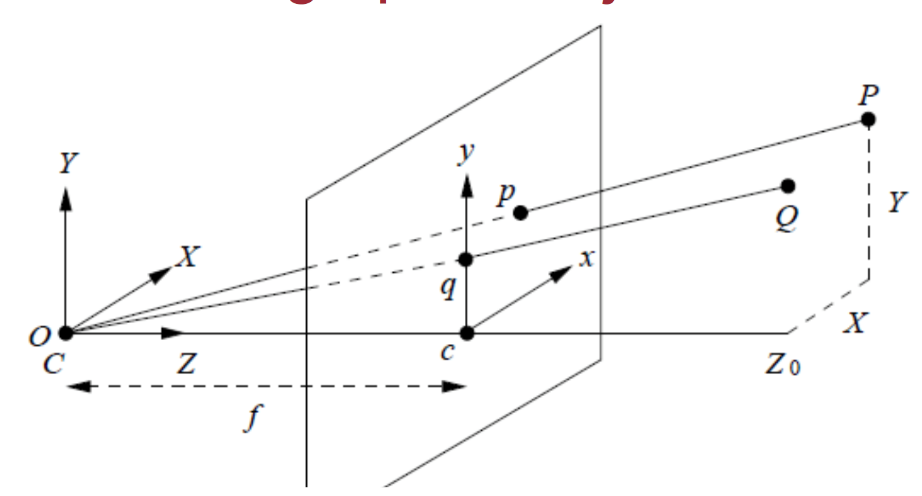
if \_\_name\_\_ == '\_\_main\_\_':

print(parseYamlFile(sys.argv[1]))

OpenCV生成的YAML文件会带有一些特殊标记，而python的yaml库会将这些标记认为文件错误，所以首先先将这些标记去除，然后在进行解析，最后使用numpy库将相机内参矩阵一维数组转换成33的矩阵。将脚本保存为readyaml.py，留作自定义库在后续代码中引入调用。

**0x02数学模型**

在计算机中，我们需要把相机转换为一种数学模型，这样我们把图片信息交给计算机才会被正确的识别出来。在SLAM中，相机被简化成针孔相机模型，如下图所示（图片来自<http://www.comp.nus.edu.sg/~cs4243/lecture/camera.pdf>）：



空间中的一个点**(*x, y, z*)**和它在图像中的坐标**(*u, v, d*)**的对应关系如下：

其中，***fx***，***fy***指相机在***x***轴和***y***轴上的焦距，***cx***，***cy***是相机的光圈中心，***s***是缩放因数。

由此公式可以推导出用**(*u, v, d*)**来表示**(*x, y, z*)**的关系公式：

这里，我们把***fx***，***fy***，***cx***，***cy***定义为相机内参矩阵参数***C***，是相机生产完成后就不会改变的参数。确定了相机内参之后，点云的每个点的空间位置可以使用下面这个矩阵模型来进行描述：

其中，R和t代表相机姿态，R是相机的旋转矩阵，t是相机的平移矢量。在本例中，我们使用单幅图片来进行点云转换，所以可以认为相机没有旋转也没有平移，那么R可以看作单位矩阵I，t可以看作0。s为缩放参数，一般设为1000。

**0x03 2D到3D（编程）：**

首先我们需要对相机内参加以封装，便于程序调用。在python中，一切都是为对象（object），那么我们就先定义个名为CameraIntrinsicParameters的类来对相机内参进行封装：

class CameraIntrinsicParameters(object):

def \_\_init\_\_(self, cx, cy, fx, fy, scale):

super(CameraIntrinsicParameters, self).\_\_init\_\_()

self.cx = cx

self.cy = cy

self.fx = fx

self.fy = fy

self.scale = scale

把相机内参封装好了之后，可以通过将标定好的相机内参数据传递进来保存。

CameraIntrinsicData = readyaml.parseYamlFile('./calibration\_data/asua/camera.yml')

camera = CameraIntrinsicParameters(CameraIntrinsicData[0][2], CameraIntrinsicData[1][2], CameraIntrinsicData[0][0], CameraIntrinsicData[1][1], 1000.0)

这样，指定的相机内参就保存在名为camera的类变量里面了。

根据之前的数学模型，我们需要读取两张图片内的信息，这里就需要借助OpenCV库来获取图片信息了：

import cv2

rgb = cv2.imread(“rgb.png”)

depth = cv2.imread( “depth.png”, -1 )

在python中，cv2.imread()函数会将图片数据存储为一个多维列表（list），rgb存储了图像的颜色信息，但cv2会将单个像素的颜色信息存储为[b, g, r]格式。有了图像信息，我们就可以开始转换点云了，这里需要借助名为PCL（点云库）的第三方库来生成点云。首先使用PCL库创建一个空白的点云：

import pcl

cloud = pcl.PointCloud()

获取图像的长度和宽度并新建一个名为pointcloud的空列表，以便后续程序遍历整个图像使用：

rows = len(depth)

cols = len(depth[0])

pointcloud = []

接下来遍历图像，根据之前推导出来的公式计算点云信息：

for m in range(0, rows):

for n in range(0, cols):

d = depth[m][n]

if d == 0:

pass

else:

z = float(d) / camera.scale

x = (n - camera.cx) \* z / camera.fx

y = (m - camera.cy) \* z / camera.fy

points = [x, y, z]

pointcloud.append(points)

由于pcl库并不会直接识别列表格式的点云数据，所以我们需要使用numpy库进行数据格式转换，并将点云保存在文本文件中：

import numpy as np

pointcloud = np.array(pointcloud, dtype = np.float32)

cloud.from\_array(pointcloud)

pcl.save(cloud, “cloud.pcd”, format = 'pcd')

这样，一个简单的将图像转换成点云的脚本就写好了。

**0x04优化：**

当我们执行完上述脚本后，通过pcl\_viewer打开pcd文件，会发现生成的点云并没有颜色信息。接下来我们将对代码进行优化，让生成的点云附带有rgb颜色信息。

要想让点云带有颜色信息，需要先来研究下pcd的文件格式。我们可以使用文本编辑器（如Ubuntu下的gedit，sublime text等）打开cloud.pcd。这时我们会发现这个文件的前几行起到了文件头的作用。通过查询网络，这些文件头有如下含义：

（以下内容来自：<http://www.pclcn.org/study/shownews.php?lang=cn&id=54>）

· VERSION – 指定PCD文件版本

· FIELDS – 指定一个点可以有的每一个维度和字段的名字。例如：

FIELDS x y z # XYZ data

FIELDS x y z rgb # XYZ + colors

FIELDS x y z normal\_x normal\_y normal\_z # XYZ + surface normals

FIELDS j1 j2 j3 # moment invariants

...

· SIZE – 用字节数指定每一个维度的大小。例如：

*unsigned char*/*char* has 1 byte

*unsigned short/short has 2 bytes*

*unsignedint/int/float has 4 bytes*

*double has 8 bytes*

· TYPE – 用一个字符指定每一个维度的类型。现在被接受的类型有：

**I** – 表示有符号类型int8（char）、int16（short）和int32（int）；

**U** – 表示无符号类型uint8（unsigned char）、uint16（unsigned short）和uint32（unsigned int）；

**F** – 表示浮点类型。

· COUNT – 指定每一个维度包含的元素数目。例如，x这个数据通常有一个元素，但是像VFH这样的特征描述子就有308个。实际上这是在给每一点引入n维直方图描述符的方法，把它们当做单个的连续存储块。默认情况下，如果没有COUNT，所有维度的数目被设置成1。

· WIDTH – 用点的数量表示点云数据集的宽度。根据是有序点云还是无序点云，WIDTH有两层解释：

1)它能确定无序数据集的点云中点的个数（和下面的POINTS一样）；

2)它能确定有序点云数据集的宽度（一行中点的数目）。

例如：

WIDTH 640       # 每行有640个点

· HEIGHT – 用点的数目表示点云数据集的高度。类似于WIDTH ，HEIGHT也有两层解释：

1)它表示有序点云数据集的高度（行的总数）；

2)对于无序数据集它被设置成1（被用来检查一个数据集是有序还是无序）。

有序点云例子：

WIDTH 640       # 像图像一样的有序结构，有640行和480列，

HEIGHT 480      # 这样该数据集中共有640\*480=307200个点

无序点云例子：

WIDTH 307200

HEIGHT 1        # 有307200个点的无序点云数据集

· VIEWPOINT – 指定数据集中点云的获取视点。VIEWPOINT有可能在不同坐标系之间转换的时候应用，在辅助获取其他特征时也比较有用，例如曲面法线，在判断方向一致性时，需要知道视点的方位，

视点信息被指定为平移（txtytz）+ 四元数（qwqxqyqz）。默认值是：

VIEWPOINT 0 0 0 1 0 0 0

· POINTS – 指定点云中点的总数。从0.7版本开始，该字段就有点多余了，因此有可能在将来的版本中将它移除。

例如：

POINTS 307200   #点云中点的总数为307200

· DATA – 指定存储点云数据的数据类型。从0.7版本开始，支持两种数据类型：ascii和二进制。

注意：文件头最后一行（DATA）的下一个字节就被看成是点云的数据部分了，它会被解释为点云数据。

根据这段文件头信息的解释，我们可以编写一段代码对生成的点云数据进行添加颜色的处理。

使用文本编辑器打开cloud.pcd文件，文件头信息如下：

# .PCD v0.7 - Point Cloud Data file format

VERSION 0.7

FIELDS x y z

SIZE 4 4 4

TYPE F F F

COUNT 1 1 1

WIDTH 204186

HEIGHT 1

VIEWPOINT 0 0 0 1 0 0 0

POINTS 204186

DATA ascii

这里面需要修改的只有第三、第四和第五行。

将第三行的信息修改为：FIELDS x y z rgb；

将第四行的信息修改为：SIZE 4 4 4 4

将第五行的信息修改为：TYPE F F F I

这些修改是为后续向数据添加颜色信息做准备。对于代码，首先我们要定义一个名为AddColorToPCDFile的函数，用来向生成后的pcd文件内添加颜色数据，首先先修改文件头信息并保存：

def AddColorToPCDFile(filename):

with open(filename, 'rb') as f:

lines = f.readlines()

lines[2] = lines[2].split('\n')[0] + ' rgb\n'

lines[3] = lines[3].split('\n')[0] + ' 4\n'

lines[4] = lines[4].split('\n')[0] + ' I\n'

lines[5] = lines[5].split('\n')[0] + ' 1\n

with open(filename, 'wb') as fw:

fw.writelines(lines)

之前我们提到过，rgb颜色信息被存在名为rgb的变量里，那么我们需要将每个像素的颜色信息分离出来，并将分离好的每个像素的颜色信息转换成十六进制格式后保存进一个列表中：

def ImageToPointCloud(RGBFilename, DepthFilename, CloudFilename, camera):

rgb = cv2.imread( RGBFilename )

depth = cv2.imread( DepthFilename, -1 )

cloud = pcl.PointCloud()

rows = len(depth)

cols = len(depth[0])

pointcloud = []

colors = []

for m in range(0, rows):

for n in range(0, cols):

d = depth[m][n]

if d == 0:

pass

else:

z = float(d) / camera.scale

x = (n - camera.cx) \* z / camera.fx

y = (m - camera.cy) \* z / camera.fy

points = [x, y, z]

pointcloud.append(points)

b = rgb[m][n][0]

g = rgb[m][n][1]

r = rgb[m][n][2]

color = (r << 16) | (g << 8) | b

colors.append(int(color))

pointcloud = np.array(pointcloud, dtype = np.float32)

cloud.from\_array(pointcloud)

pcl.save(cloud, CloudFilename, format = 'pcd')

由于保存颜色信息的序列顺序和生成点云的序列顺序完全一样，那么我们就可以遍历cloud.pcd的data区域，逐个添加颜色信息：

def AddColorToPCDFile(filename, colors):

with open(filename, 'rb') as f:

lines = f.readlines()

lines[2] = lines[2].split('\n')[0] + ' rgb\n'

lines[3] = lines[3].split('\n')[0] + ' 4\n'

lines[4] = lines[4].split('\n')[0] + ' I\n'

lines[5] = lines[5].split('\n')[0] + ' 1\n'

for i in range(11, len(colors) + 11):

lines[i] = lines[i].split('\n')[0] + ' ' + str(colors[i - 11]) + '\n'

with open(filename, 'wb') as fw:

fw.writelines(lines)

最后，在ImageToPointCloud函数结尾添加修改颜色的调用AddColorToPCDFile(CloudFilename, colors)即可。这样对生成的点云进行的添加颜色的优化就完成了。

参考资料：

[1] 一起做RGB-D SLAM (2) - 半闲居士 - 博客园 - 地址：<http://www.cnblogs.com/gaoxiang12/p/4652478.html>

[2] PCD（点云数据）文件格式 - 地址：<http://www.pclcn.org/study/shownews.php?lang=cn&id=54>

**0x05附件：**

**以下是优化后的完整代码：**

***文件名：slamBase.py***

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import cv2

import numpy as np

import pcl

import readyaml

class CameraIntrinsicParameters(object):

"""docstring for CameraIntrinsicParameters"""

def \_\_init\_\_(self, cx, cy, fx, fy, scale):

super(CameraIntrinsicParameters, self).\_\_init\_\_()

self.cx = cx

self.cy = cy

self.fx = fx

self.fy = fy

self.scale = scale

def addColorToPCDFile(filename, colors):

with open(filename, 'rb') as f:

lines = f.readlines()

lines[2] = lines[2].split('\n')[0] + ' rgb\n'

lines[3] = lines[3].split('\n')[0] + ' 4\n'

lines[4] = lines[4].split('\n')[0] + ' I\n'

lines[5] = lines[5].split('\n')[0] + ' 1\n'

for i in range(11, len(colors) + 11):

lines[i] = lines[i].split('\n')[0] + ' ' + str(colors[i - 11]) + '\n'

with open(filename, 'wb') as fw:

fw.writelines(lines)

def point2dTo3d(n, m, d, camera):

z = float(d) / camera.scale

x = (n - camera.cx) \* z / camera.fx

y = (m - camera.cy) \* z / camera.fy

point = np.array([x, y, z], dtype = np.float32)

return point

def imageToPointCloud(RGBFilename, DepthFilename, CloudFilename, camera):

rgb = cv2.imread( RGBFilename )

depth = cv2.imread( DepthFilename, cv2.COLOR\_BGR2GRAY )

# ROS中rqt保存的深度摄像头的图片是rgb格式，需要转换成单通道灰度格式

if len(depth[0][0]) == 3:

depth = cv2.cvtColor(depth, cv2.COLOR\_BGR2GRAY)

cloud = pcl.PointCloud()

rows = len(depth)

cols = len(depth[0])

pointcloud = []

colors = []

for m in range(0, rows):

for n in range(0, cols):

d = depth[m][n]

if d == 0:

pass

else:

point = point2dTo3d(n, m, d, camera)

pointcloud.append(point)

b = rgb[m][n][0]

g = rgb[m][n][1]

r = rgb[m][n][2]

color = (r << 16) | (g << 8) | b

colors.append(int(color))

pointcloud = np.array(pointcloud, dtype = np.float32)

cloud.from\_array(pointcloud)

pcl.save(cloud, CloudFilename, format = 'pcd')

addColorToPCDFile(CloudFilename, colors)

if \_\_name\_\_ == '\_\_main\_\_':

CameraIntrinsicData = readyaml.parseYamlFile('./calibration\_data/asua/camera.yml')

camera = CameraIntrinsicParameters(CameraIntrinsicData[0][2], CameraIntrinsicData[1][2], CameraIntrinsicData[0][0], CameraIntrinsicData[1][1], 1000.0)

imageToPointCloud('./data/asua/rgb.png', './data/asua/depth.png', './data/asua/cloud.pcd', camera)

***文件名：readyaml.py***

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import yaml, sys

import numpy as np

def fixYamlFile(filename):

with open(filename, 'rb') as f:

lines = f.readlines()

if lines[0] != '%YAML 1.0\n':

lines[0] = '%YAML 1.0\n'

for line in lines:

if ' !!opencv-matrix' in line:

lines[lines.index(line)] = line.split(' !!opencv-matrix')[0] + '\n'

with open(filename, 'wb') as fw:

fw.writelines(lines)

def parseYamlFile(filename):

fixYamlFile(filename)

f = open(filename)

x = yaml.load(f)

f.close()

arr = np.array(x['camera\_matrix']['data'], dtype = np.float32)

return (arr.reshape(3, 3))

if \_\_name\_\_ == '\_\_main\_\_':

print(parseYamlFile(sys.argv[1]))

***文件名：Checkerboard.py*（用于生成标定板棋盘格）**

#!/usr/bin/python

#-\*- coding:utf-8 -\*-

import cv2

import numpy as np

width = 1240

height = 1754

row = []

col = []

for rows in range(0, width):

for cols in range(0, height):

if (cols > 76) and (cols < 1677) and (rows > 19) and (rows < 1220):

if (((cols - 77) / 200) % 2 != 0) and (((rows - 20) / 200) % 2 != 0):

col.append([0, 0, 0])

if (((cols - 77) / 200) % 2 == 0) and (((rows - 20) / 200) % 2 == 0):

col.append([0, 0, 0])

if (((cols - 77) / 200) % 2 != 0) and (((rows - 20) / 200) % 2 == 0):

col.append([255, 255, 255])

if (((cols - 77) / 200) % 2 == 0) and (((rows - 20) / 200) % 2 != 0):

col.append([255, 255, 255])

else:

col.append([255, 255, 255])

row.append(col)

col = []

img = np.array(row, dtype = np.uint8)

cv2.imwrite( "Checkerboard.png", img )

**第二篇：特征提取与配准（Python版）**

**0x00 准备**

上一篇（从图像到点云）介绍了如何将平面图像转换为立体点云，在本篇中，我们将上一篇的代码进行封装，然后在本篇中调用封装好的函数接口。

**0x01 图像配准（数学部分）**

SLAM算法由定位（Localization）和建图（Mapping）构成。要求解机器人的运动问题，首先要解决如何从给定的两张图片中求出图像的运动关系。

假设我们有两帧图片，分别为和，并且我们获取了相应的特征点：

现在要通过这两组特征点来求出一个旋转矩阵和位移矢量，使得：

但是由于误差的存在，使得等号无法完全成立，所以使用一个最小误差法来求解,:

那么这个问题关键就是获取一组一一对应的空间点，这个可以通过图像的特征匹配来完成。

**0x02 图像配准（编程实现）**

首先我们先通过深度摄像头获取一组带有明显位移的图像（RGB图像和深度图像）数据，分别命名为rgb1.png，rgb2.png，depth1.png，depth2.png。编写函数读取图像信息：

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import cv2

import numpy as np

from slamBase import point2dTo3d, CameraIntrinsicParameters

import readyaml

def readImgFiles(RGBFilenames, DepthFilenames, paras):

rgbs = []

depths = []

for i in range(0, len(RGBFilenames)):

rgbs.append(cv2.imread(RGBFilenames[i]))

for j in range(0, len(DepthFilenames)):

depth = cv2.imread(DepthFilenames[i], paras)

# ROS中rqt保存的深度摄像头的图片是rgb格式，需要转换成单通道灰度格式

if len(depth[0][0]) == 3:

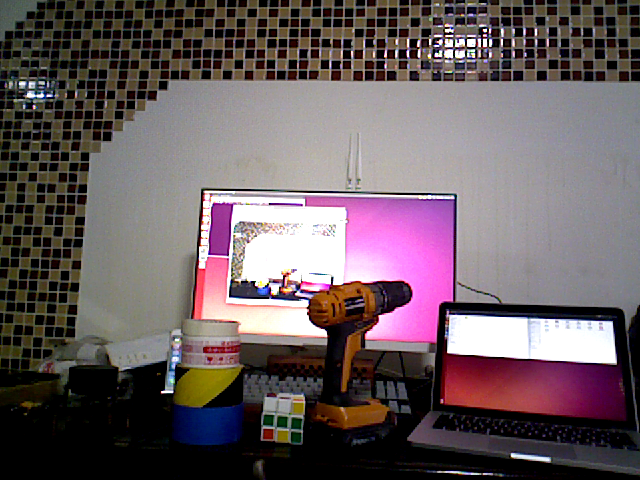
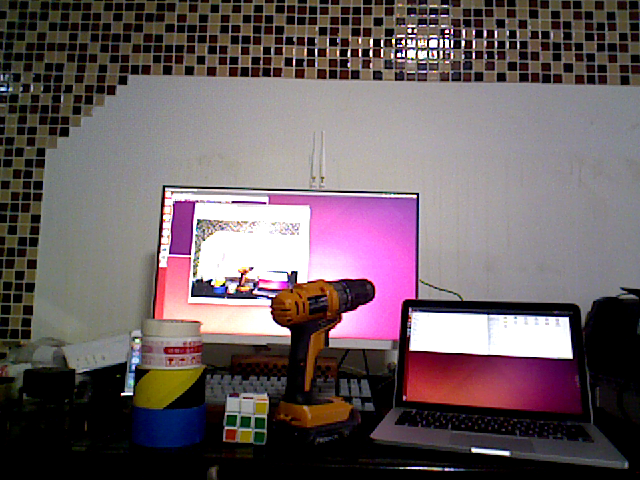
depths.append(cv2.cvtColor(depth, cv2.COLOR\_BGR2GRAY))

else:

depths.append(depth)

return rgbs, depths

这里定义了两个空列表rgbs和depths，按读取顺序将图像信息保存进去。需要注意的是，使用ROS的rqt获取的深度图像是RGB三通道格式，后续代码需要单通道灰度格式图像，这里通过OpenCV进行一次格式转换才能使数据在后续代码中继续使用。

在这里，我们会使用OpenCV中的SIFT算法来进行特征提取并计算描述子。在OpenCV3.0以后的版本中，SIFT算法被移动到了OpenCV\_Contrib库里面，需要在编译OpenCV的时候指定这个库的路径（OPENCV\_EXTRA\_MODULES\_PATH键值）后才能使用。

def computeMatches(rgb1, rgb2, depth1, depth2, CameraIntrinsicData, distCoeffs, camera):

sift = cv2.xfeatures2d.SIFT\_create()

kp1, des1 = sift.detectAndCompute(rgb1, None)

kp2, des2 = sift.detectAndCompute(rgb2, None)

print("Key points of two images: " + str(len(kp1)) + ", " + str(len(kp2)))

将提取出来的关键点（KeyPoints）在对应图片上标记后并展示出来：

imgShow = None

imgShow = cv2.drawKeypoints(rgb1, kp1, imgShow, flags = cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)

cv2.imshow( "keypoints\_1", imgShow )

cv2.imwrite( "./data/keypoints\_1.png", imgShow )

cv2.waitKey(0)

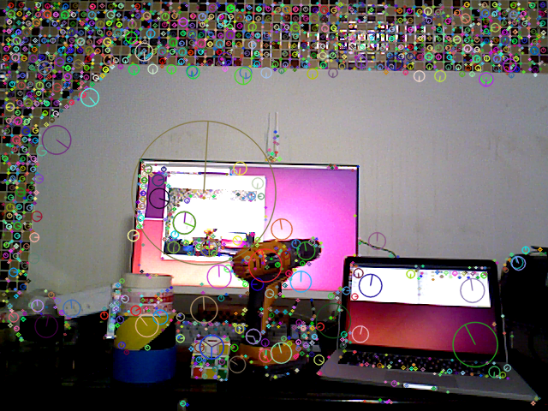
imgShow = None

imgShow = cv2.drawKeypoints(rgb2, kp2, imgShow, flags = cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)

cv2.imshow( "keypoints\_2", imgShow )

cv2.imwrite( "./data/keypoints\_2.png", imgShow )

cv2.waitKey(0)

关键点（KeyPoints）是OpenCV中的一种特殊的数据结构，拥有点坐标，半径，角度等成员。将关键点画在图像上就是一个个不同半径的圆。标记圈的半径长短和特征点所在尺度有关，那条半径是特征点的方向。

接下来使用FLANN算法对提取后的特征点的描述子进行特征匹配运算，并用drawMatches函数画出运算后的结果：

FLANN\_INDEX\_KDTREE = 0

index\_params = dict(algorithm = FLANN\_INDEX\_KDTREE, trees = 5)

search\_params = dict(checks = 50) # or pass empty dictionary

matcher = cv2.FlannBasedMatcher(index\_params, search\_params)

matches = matcher.match(des1, des2)

print("Find total " + str(len(matches)) + " matches.")

imgMatches = None

imgMatches = cv2.drawMatches( rgb1, kp1, rgb2, kp2, matches, imgMatches )

cv2.imshow( "matches", imgMatches )

cv2.imwrite( "./data/matches.png", imgMatches )

cv2.waitKey(0)



从匹配结果来看，大部分的匹配并不是我们想象中的那样，把许多不相似的地方匹配了起来，属于误匹配。因此，我们需要将这些误匹配筛选掉。例如：筛选掉大于最小距离4倍的匹配。

goodMatches = []

minDis = 9999.0

for i in range(0, len(matches)):

if matches[i].distance < minDis:

minDis = matches[i].distance

for i in range(0, len(matches)):

if matches[i].distance < (minDis \* 4):

goodMatches.append(matches[i])

print("good matches = " + str(len(goodMatches)))

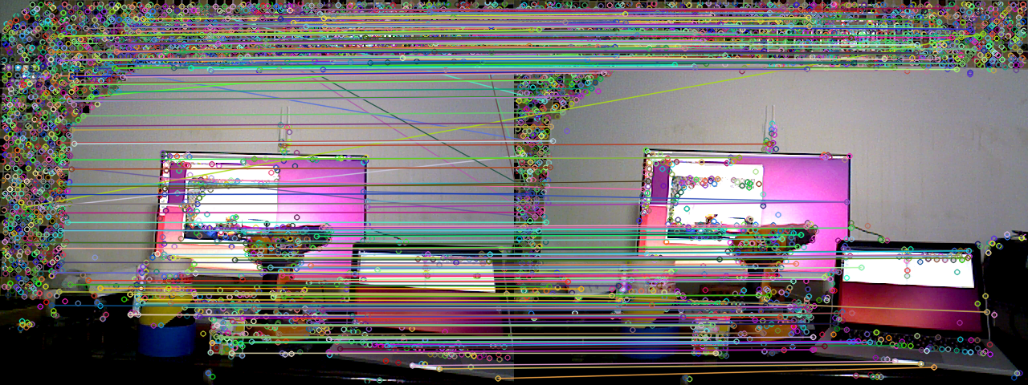
imgMatches = None

imgMatches = cv2.drawMatches( rgb1, kp1, rgb2, kp2, goodMatches, imgMatches )

cv2.imshow( "good\_matches", imgMatches )

cv2.imwrite( "./data/good\_matches.png", imgMatches )

cv2.waitKey(0)



筛选过后，误匹配的情况大大减少。对筛选过后的匹配点进行PnP求解，计算两张图片的旋转矩阵和位移矢量：

OpenCV的solvePnPRansac API（来自：[http://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera\_calibration\_and\_3d\_reconstruction.html?highlight=solvepnpransac#cv2.solvePnPRansac](http://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html?highlight=solvepnpransac%23cv2.solvePnPRansac)）：

**Python**: cv2.solvePnPRansac(objectPoints, imagePoints, cameraMatrix, distCoeffs[, rvec[, tvec[, useExtrinsicGuess[, iterationsCount[, reprojectionError[, minInliersCount[, inliers[, flags]]]]]]]]) → rvec, tvec, inliers

|  |  |
| --- | --- |
| **Parameters**: | **objectPoints** – Array of object points in the object coordinate space, 3xN/Nx3 1-channel or 1xN/Nx1 3-channel, where N is the number of points. vector<Point3f> can be also passed here.  **imagePoints** – Array of corresponding image points, 2xN/Nx2 1-channel or 1xN/Nx1 2-channel, where N is the number of points. vector<Point2f> can be also passed here.  **cameraMatrix** – Input camera matrix A = \vecthreethree{fx}{0}{cx}{0}{fy}{cy}{0}{0}{1} .  **distCoeffs** – Input vector of distortion coefficients(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6],[s_1, s_2, s_3, s_4]]) of 4, 5, 8 or 12 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.  **rvec** – Output rotation vector (see [Rodrigues()](http://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html?highlight=solvepnpransac#void Rodrigues(InputArray src, OutputArray dst, OutputArray jacobian)) ) that, together with tvec , brings points from the model coordinate system to the camera coordinate system.  **tvec** – Output translation vector.  **useExtrinsicGuess** – Parameter used for SOLVEPNP\_ITERATIVE. If true (1), the function uses the provided rvec and tvec values as initial approximations of the rotation and translation vectors, respectively, and further optimizes them.  **iterationsCount** – Number of iterations.  **reprojectionError** – Inlier threshold value used by the RANSAC procedure. The parameter value is the maximum allowed distance between the observed and computed point projections to consider it an inlier.  **confidence** – The probability that the algorithm produces a useful result.  **inliers** – Output vector that contains indices of inliers in objectPoints and imagePoints .  **flags** – Method for solving a PnP problem (see [solvePnP()](http://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html?highlight=solvepnpransac#bool solvePnP(InputArray objectPoints, InputArray imagePoints, InputArray cameraMatrix, InputArray distCoeffs, OutputArray rvec, OutputArray tvec, bool useExtrinsicGuess, int flags)) ). |

求解PnP需要输入一组匹配好的三维点**objectPoints**和一组二维图像点**imagePoints**，返回的结果是旋转向量rvec和平移向量tvec，cameraMatrix是相机内参矩阵，通过标定相机获得，distCoeffs是相机畸变参数，标定后会和相机内参矩阵同时生成写入文件中，我们可以使用readyaml.py来获取相机的畸变参数，reprojectionError参数在使用示例标定相机时最后会被打印在屏幕上 。接下来我们可以将筛选过后的goodmatches通过上一章的函数转变为三维点。

pts\_obj = []

pts\_img = []

for i in range(0, len(goodMatches)):

p = kp1[goodMatches[i].queryIdx].pt

d = depth1[int(p[1])][int(p[0])]

if d == 0:

pass

else:

pts\_img.append(kp2[goodMatches[i].trainIdx].pt)

pd = point2dTo3d(p[0], p[1], d, camera)

pts\_obj.append(pd)

pts\_obj = np.array(pts\_obj)

pts\_img = np.array(pts\_img)

cameraMatrix = np.matrix(CameraIntrinsicData)

rvec = None

tvec = None

inliers = None

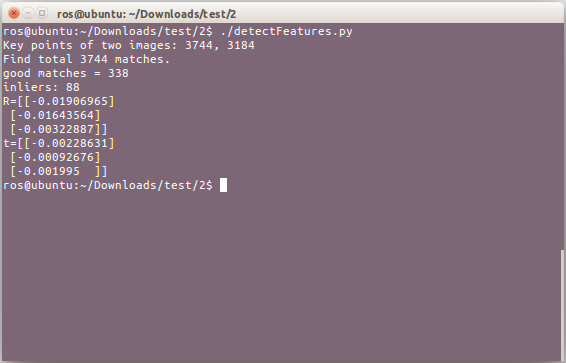
retval, rvec, tvec, inliers = cv2.solvePnPRansac( pts\_obj, pts\_img, cameraMatrix, distCoeffs, useExtrinsicGuess = False, iterationsCount = 100, reprojectionError = 0.66 )

print("inliers: " + str(len(inliers)))

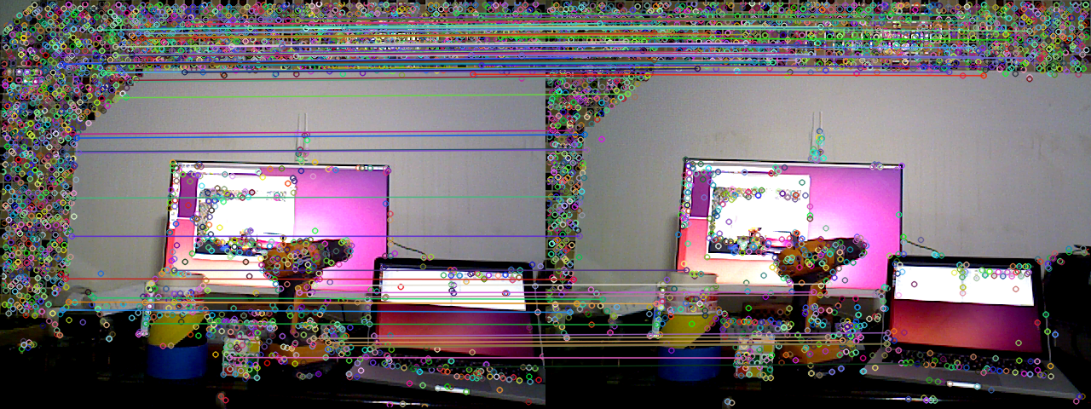
print("R=" + str(rvec))

print("t=" + str(tvec))

求解完成后，我们就有了rvec和tvec信息，这样我们就算出了两个图像之间的运动关系了：



尽管经过了筛选，匹配结果中还是存在一定的误匹配情况，在OpenCV中，会利用“随机采样一致性”，在现有的匹配结果中随机提取一部分，估计其运动，在本例中，这对图像的inlier是这样的：



本节中，我们介绍了如何提取、匹配图像的特征，并通过这些匹配，使用ransac方法估计图像的运动。下一节，我们将介绍如何使用刚得到的平移、旋转向量来拼接点云。至此，我们就完成了一个只有两帧的迷你SLAM程序

参考资料：

[1] 一起做RGB-D SLAM (3) - 半闲居士 - 博客园 - 地址：<http://www.cnblogs.com/gaoxiang12/p/4659805.html>

[2] OpenCV API Reference - Camera Calibration and 3D Reconstruction - 地址：[http://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera\_calibration\_and\_3d\_reconstruction.html?highlight=solvepnpransac#bool solvePnPRansac(InputArray objectPoints, InputArray imagePoints, InputArray cameraMatrix, InputArray distCoeffs, OutputArray rvec, OutputArray tvec, bool useExtrinsicGuess, int iterationsCount , float reprojectionError , double confidence , OutputArray inliers , int flags)](http://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html?highlight=solvepnpransac%23bool%20solvePnPRansac(InputArray%20objectPoints,%20InputArray%20imagePoints,%20InputArray%20cameraMatrix,%20InputArray%20distCoeffs,%20OutputArray%20rvec,%20OutputArray%20tvec,%20bool%20useExtrinsicGuess,%20int%20iterationsCount%20,%20float%20reprojectionError%20,%20double%20confidence%20,%20OutputArray%20inliers%20,%20int%20flags))

**0x03附件：**

**以下是完整代码：**

***文件名：slamBase.py***

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import cv2

import numpy as np

import pcl

import readyaml

class CameraIntrinsicParameters(object):

"""docstring for CameraIntrinsicParameters"""

def \_\_init\_\_(self, cx, cy, fx, fy, scale):

super(CameraIntrinsicParameters, self).\_\_init\_\_()

self.cx = cx

self.cy = cy

self.fx = fx

self.fy = fy

self.scale = scale

def addColorToPCDFile(filename, colors):

with open(filename, 'rb') as f:

lines = f.readlines()

lines[2] = lines[2].split('\n')[0] + ' rgb\n'

lines[3] = lines[3].split('\n')[0] + ' 4\n'

lines[4] = lines[4].split('\n')[0] + ' I\n'

lines[5] = lines[5].split('\n')[0] + ' 1\n'

for i in range(11, len(colors) + 11):

lines[i] = lines[i].split('\n')[0] + ' ' + str(colors[i - 11]) + '\n'

with open(filename, 'wb') as fw:

fw.writelines(lines)

def point2dTo3d(n, m, d, camera):

z = float(d) / camera.scale

x = (n - camera.cx) \* z / camera.fx

y = (m - camera.cy) \* z / camera.fy

point = np.array([x, y, z], dtype = np.float32)

return point

def imageToPointCloud(RGBFilename, DepthFilename, CloudFilename, camera):

rgb = cv2.imread( RGBFilename )

depth = cv2.imread( DepthFilename, cv2.COLOR\_BGR2GRAY )

# ROS中rqt保存的深度摄像头的图片是rgb格式，需要转换成单通道灰度格式

if len(depth[0][0]) == 3:

depth = cv2.cvtColor(depth, cv2.COLOR\_BGR2GRAY)

cloud = pcl.PointCloud()

rows = len(depth)

cols = len(depth[0])

pointcloud = []

colors = []

for m in range(0, rows):

for n in range(0, cols):

d = depth[m][n]

if d == 0:

pass

else:

point = point2dTo3d(n, m, d, camera)

pointcloud.append(point)

b = rgb[m][n][0]

g = rgb[m][n][1]

r = rgb[m][n][2]

color = (r << 16) | (g << 8) | b

colors.append(int(color))

pointcloud = np.array(pointcloud, dtype = np.float32)

cloud.from\_array(pointcloud)

pcl.save(cloud, CloudFilename, format = 'pcd')

addColorToPCDFile(CloudFilename, colors)

if \_\_name\_\_ == '\_\_main\_\_':

CameraIntrinsicData = readyaml.parseYamlFile('./calibration\_data/tianmao/camera.yml')

camera = CameraIntrinsicParameters(CameraIntrinsicData[0][2], CameraIntrinsicData[1][2], CameraIntrinsicData[0][0], CameraIntrinsicData[1][1], 1000.0)

imageToPointCloud('rgb.png', 'depth.png', 'cloud1.pcd', camera)

***文件名：readyaml.py***

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import yaml, sys

import numpy as np

def fixYamlFile(filename):

with open(filename, 'rb') as f:

lines = f.readlines()

if lines[0] != '%YAML 1.0\n':

lines[0] = '%YAML 1.0\n'

for line in lines:

if ' !!opencv-matrix' in line:

lines[lines.index(line)] = line.split(' !!opencv-matrix')[0] + '\n'

with open(filename, 'wb') as fw:

fw.writelines(lines)

def parseYamlFile(filename):

fixYamlFile(filename)

f = open(filename)

x = yaml.load(f)

f.close()

CameraIntrinsicData = np.array(x['camera\_matrix']['data'], dtype = np.float32)

DistortionCoefficients = np.array(x['distortion\_coefficients']['data'], dtype = np.float32)

return (CameraIntrinsicData.reshape(3, 3), DistortionCoefficients)

if \_\_name\_\_ == '\_\_main\_\_':

print(parseYamlFile(sys.argv[1]))

***文件名：detectFeatures.py***

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import cv2

import numpy as np

from slamBase import point2dTo3d, CameraIntrinsicParameters

import readyaml

def readImgFiles(RGBFilenames, DepthFilenames, paras):

rgbs = []

depths = []

for i in range(0, len(RGBFilenames)):

rgbs.append(cv2.imread(RGBFilenames[i]))

for j in range(0, len(DepthFilenames)):

depth = cv2.imread(DepthFilenames[i], paras)

# ROS中rqt保存的深度摄像头的图片是rgb格式，需要转换成单通道灰度格式

if len(depth[0][0]) == 3:

depths.append(cv2.cvtColor(depth, cv2.COLOR\_BGR2GRAY))

else:

depths.append(depth)

return rgbs, depths

def computeMatches(rgb1, rgb2, depth1, depth2, CameraIntrinsicData, distCoeffs, camera):

sift = cv2.xfeatures2d.SIFT\_create()

kp1, des1 = sift.detectAndCompute(rgb1, None)

kp2, des2 = sift.detectAndCompute(rgb2, None)

print("Key points of two images: " + str(len(kp1)) + ", " + str(len(kp2)))

imgShow = None

imgShow = cv2.drawKeypoints(rgb1, kp1, imgShow, flags = cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)

cv2.imshow( "keypoints", imgShow )

cv2.imwrite( "./data/keypoints.png", imgShow )

cv2.waitKey(0)

FLANN\_INDEX\_KDTREE = 0

index\_params = dict(algorithm = FLANN\_INDEX\_KDTREE, trees = 5)

search\_params = dict(checks = 50) # or pass empty dictionary

matcher = cv2.FlannBasedMatcher(index\_params, search\_params)

matches = matcher.match(des1, des2)

print("Find total " + str(len(matches)) + " matches.")

imgMatches = None

imgMatches = cv2.drawMatches( rgb1, kp1, rgb2, kp2, matches, imgMatches )

cv2.imshow( "matches", imgMatches )

cv2.imwrite( "./data/matches.png", imgMatches )

cv2.waitKey(0)

goodMatches = []

minDis = 9999.0

for i in range(0, len(matches)):

if matches[i].distance < minDis:

minDis = matches[i].distance

for i in range(0, len(matches)):

if matches[i].distance < (minDis \* 4):

goodMatches.append(matches[i])

print("good matches = " + str(len(goodMatches)))

imgMatches = None

imgMatches = cv2.drawMatches( rgb1, kp1, rgb2, kp2, goodMatches, imgMatches )

cv2.imshow( "good\_matches", imgMatches )

cv2.imwrite( "./data/good\_matches.png", imgMatches )

cv2.waitKey(0)

pts\_obj = []

pts\_img = []

for i in range(0, len(goodMatches)):

p = kp1[goodMatches[i].queryIdx].pt

d = depth1[int(p[1])][int(p[0])]

if d == 0:

pass

else:

pts\_img.append(kp2[goodMatches[i].trainIdx].pt)

pd = point2dTo3d(p[0], p[1], d, camera)

pts\_obj.append(pd)

pts\_obj = np.array(pts\_obj)

pts\_img = np.array(pts\_img)

cameraMatrix = np.matrix(CameraIntrinsicData)

rvec = None

tvec = None

inliers = None

retval, rvec, tvec, inliers = cv2.solvePnPRansac( pts\_obj, pts\_img, cameraMatrix, distCoeffs, useExtrinsicGuess = False, iterationsCount = 100, reprojectionError = 0.66 )

print("inliers: " + str(len(inliers)))

print("R=" + str(rvec))

print("t=" + str(tvec))

matchesShow = []

for i in range(0, len(inliers)):

matchesShow.append( goodMatches[inliers[i][0]] )

imgMatches = None

imgMatches = cv2.drawMatches( rgb1, kp1, rgb2, kp2, matchesShow, imgMatches )

cv2.imshow( "inlier matches", imgMatches )

cv2.imwrite( "./data/inliers.png", imgMatches )

cv2.waitKey(0)

def main():

rgbfilenames = ('./data/rgb1.png', './data/rgb2.png')

depthfilenames = ('./data/depth1.png', './data/depth2.png')

CameraIntrinsicData, DistortionCoefficients = readyaml.parseYamlFile('./calibration\_data/tianmao/camera.yml')

camera = CameraIntrinsicParameters(CameraIntrinsicData[0][2], CameraIntrinsicData[1][2], CameraIntrinsicData[0][0], CameraIntrinsicData[1][1], 1000.0)

rgbs, depths = readImgFiles(rgbfilenames, depthfilenames, cv2.COLOR\_BGR2GRAY)

computeMatches(rgbs[0], rgbs[1], depths[0], depths[1], CameraIntrinsicData, DistortionCoefficients, camera)

if \_\_name\_\_ == '\_\_main\_\_':

main()

**第三篇：点云拼接（Python版）**

**0x00 准备**

在上一篇中，我们利用比对两张图片的特征点，估计相机运动，最后得出了旋转向量和位移向量。在本篇中，我们将使用这两个向量，将两张图像的点云拼接起来，组合成一个更大的点云。

首先，先将上一篇估计相机运动的内容进行封装，这里我们定义了一个名为SolvePnP的类（**Python没有C/C++中结构体的概念**）：

class SolvePnP(object):

def \_\_init\_\_(self, RGBFileNameList, DepthFileNameList, distCoeffs, CameraIntrinsicData, camera):

super(SolvePnP, self).\_\_init\_\_()

self.RGBFileNameList = RGBFileNameList

self.DepthFileNameList = DepthFileNameList

self.distCoeffs = distCoeffs

self.CameraIntrinsicData = CameraIntrinsicData

self.camera = camera

在本篇中还是使用上一篇的两张图片作为数据，为了简化操作，在类的初始化时，输入RGB图像文件名列表和深度图像的文件名列表以及通过标定获得的摄像头畸变参数与摄像头内参矩阵。camera是从相机内参矩阵中分离出来的对应参数，在之前图像转点云中使用这些参数进行运算（本篇中也会使用）。在这个类中，定义一个readImgFiles的方法，将输入进来的文件名列表按顺序使用OpenCV读取图片信息，返回的是RGB图像信息列表和深度图像信息列表（**与C++不同的是，Python的return可以返回多个变量**）：

def readImgFiles(self, RGBFilenames, DepthFilenames, paras):

rgbs = []

depths = []

for i in range(0, len(RGBFilenames)):

rgbs.append(cv2.imread(RGBFilenames[i]))

for j in range(0, len(DepthFilenames)):

depth = cv2.imread(DepthFilenames[i], paras)

# ROS中rqt保存的深度摄像头的图片是rgb格式，需要转换成单通道灰度格式

if len(depth[0][0]) == 3:

depths.append(cv2.cvtColor(depth, cv2.COLOR\_BGR2GRAY))

else:

depths.append(depth)

return rgbs, depths

将上一篇实现的计算图像特征点来估算相机运动的代码封装进这个类中，并返回旋转向量，平移向量和inliers：

def ResultOfPnP(self, rgb1, rgb2, depth, distCoeffs, CameraIntrinsicData, camera):

sift = cv2.xfeatures2d.SIFT\_create()

kp1, des1 = sift.detectAndCompute(rgb1, None)

kp2, des2 = sift.detectAndCompute(rgb2, None)

FLANN\_INDEX\_KDTREE = 0

index\_params = dict(algorithm = FLANN\_INDEX\_KDTREE, trees = 5)

search\_params = dict(checks = 50) # or pass empty dictionary

matcher = cv2.FlannBasedMatcher(index\_params, search\_params)

matches = matcher.match(des1, des2)

print("Find total " + str(len(matches)) + " matches.")

goodMatches = []

minDis = 9999.0

for i in range(0, len(matches)):

if matches[i].distance < minDis:

minDis = matches[i].distance

for i in range(0, len(matches)):

if matches[i].distance < (minDis \* 4):

goodMatches.append(matches[i])

print("good matches = " + str(len(goodMatches)))

pts\_obj = []

pts\_img = []

for i in range(0, len(goodMatches)):

p = kp1[goodMatches[i].queryIdx].pt

d = depth[0][int(p[1])][int(p[0])]

if d == 0:

pass

else:

pts\_img.append(kp2[goodMatches[i].trainIdx].pt)

pd = point2dTo3d(p[0], p[1], d, camera)

pts\_obj.append(pd)

pts\_obj = np.array(pts\_obj)

pts\_img = np.array(pts\_img)

cameraMatrix = np.matrix(CameraIntrinsicData)

rvec = None

tvec = None

inliers = None

retval, rvec, tvec, inliers = cv2.solvePnPRansac( pts\_obj, pts\_img, cameraMatrix, distCoeffs, useExtrinsicGuess = False, iterationsCount = 100, reprojectionError = 1.76 )

return rvec, tvec, inliers

**0x01 拼接点云（数学部分）**

点云的拼接实际上是点云的变换过程，这个过程需要用到变换矩阵(**Transform Matrix**)：

左上角的是旋转矩阵，右上角的是位移矢量（求解PnP得到的tvec就是这个），左下角的是缩放矢量，在SLAM中一般全部取0，因为环境不可能忽然变大变小，右下角是1，这样就可以对其他东西进行齐次变换。：

**0x02 拼接点云（编程实现）**

在高翔的博客中，使用PCL自带的点云变换函数进行点云旋转与平移，但是Python中的PCL库并没有这个函数的接口，所以我们得自己编程实现：

def transformPointCloud(src, T):

pointcloud = []

for item in src:

a = list(item)

a.append(1)

a = np.matrix(a)

a = a.reshape((-1, 1))

temp = T \* a

temp = temp.reshape((1, -1))

temp = np.array(temp)

temp = list(temp[0])

pointcloud.append(temp[0:3])

return pointcloud

其中src是点坐标列表（list类型），T是变换矩阵。根据之前的公式，先给列表中的每个元素后面补充1，然后使用numpy库将横向列表（）变为纵向列表（），在进行其次变换，变换完后再转换回横向列表并将补充的1去掉，就是我们想要得到的每个点的点云坐标了。

对于变换矩阵T，我们可以使用OpenCV自带的罗德里格斯变换（Rodrigues）将旋转向量和平移向量组装成变换矩阵。我们可以在类SolvePnP中定义方法transformMatrix来实现：

class SolvePnP(object):

……

def transformMatrix(self, rvec, tvec):

temp = np.matrix([0, 0, 0, 1])

r = np.matrix(rvec)

t = np.matrix(tvec)

dst, jacobian = cv2.Rodrigues(r)

c = np.hstack((dst, t))

T = np.vstack((c, temp))

return T

其中，rvec是旋转向量，tvec是平移向量。OpenCV中的罗德里格斯变换会返回一个旋转矩阵和雅可比矩阵（这里并不使用，但是没有接收变量Python会报错），最后使用numpy库对矩阵进行合并。在SolvePnP类的构造函数中添加矩阵变换和求解PnP的调用：

class SolvePnP(object):

def \_\_init\_\_(self, RGBFileNameList, DepthFileNameList, distCoeffs, CameraIntrinsicData, camera):

super(SolvePnP, self).\_\_init\_\_()

……

self.rvec, self.tvec, self.inliers = self.ResultOfPnP(self.frame1.kps, self.frame2.kps, self.frame1.des, self.frame2.des, self.frame1.depth, self.distCoeffs, self.CameraIntrinsicData, self.camera)

self.T = self.transformMatrix(self.rvec, self.tvec)

最后编写程序入口joinPointCloud.py：

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import slamBase

from pcl import save

import readyaml

RGBFileNameList = ['./data/rgb1.png', './data/rgb2.png']

DepthFileNameList = ['./data/depth1.png', './data/depth2.png']

CalibrationDataFile = './calibration/asus/camera.yml'

CloudFilename = './data/cloud.pcd'

def main():

CameraIntrinsicData, DistortionCoefficients = readyaml.parseYamlFile(CalibrationDataFile)

camera = slamBase.CameraIntrinsicParameters(CameraIntrinsicData[0][2], CameraIntrinsicData[1][2], CameraIntrinsicData[0][0], CameraIntrinsicData[1][1], 1000.0)

frame1 = slamBase.Frame(RGBFileNameList[0], DepthFileNameList[0])

frame2 = slamBase.Frame(RGBFileNameList[1], DepthFileNameList[1])

pnp = slamBase.SolvePnP(DistortionCoefficients, CameraIntrinsicData, camera, frame1, frame2)

p0, c0 = slamBase.imageToPointCloud(RGBFileNameList[0], DepthFileNameList[0], camera)

p1, c1 = slamBase.imageToPointCloud(RGBFileNameList[1], DepthFileNameList[1], camera)

colors = c0 + c1

p = slamBase.transformPointCloud(p0, pnp.T)

pointcloud = slamBase.addPointCloud(p, p1)

save(pointcloud, CloudFilename, format = 'pcd')

slamBase.addColorToPCDFile(CloudFilename, colors)

if \_\_name\_\_ == '\_\_main\_\_':

main()

这样，我们就实现了一个只有两帧的SLAM程序，这已经是一个视觉里程计的雏形，只要把后续新的数据与之前的数据不断进行比对，就可以得到完整的点云数据和地图数据了。

**0x03 优化：**

定义一个名为Frame的类（一个类能解决的事，就不浪费那么多变量和方法了），将读取图片信息和计算关键点（KeyPoints）和描述子（Descriptors）方法封装进去：

class Frame(object):

def \_\_init\_\_(self, RGBFilename, DepthFilename):

super(Frame, self).\_\_init\_\_()

self.rgb = self.ReadImg(RGBFilename)

self.depth = self.ReadImg(DepthFilename)

self.kps, self.des = self.ComputeKPointsAndDescriptors(self.rgb)

def ReadImg(self, filename):

if 'depth' in filename:

img = cv2.imread(filename)

return cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

else:

return cv2.imread(filename)

def ComputeKPointsAndDescriptors(self, rgb):

sift = cv2.xfeatures2d.SIFT\_create()

kps, des = sift.detectAndCompute(rgb, None)

return kps, des

这样我们可以通过frame = Frame(RGBFilename, DepthFilename)实现获取图像内容以及关键点和描述子的操作。

参考资料：

[1] 乾坤有数的博客 - OpenCV学习笔记（一）——旋转向量与旋转矩阵相互转化<http://blog.sina.com.cn/s/blog_5fb3f125010100hp.html>

[2] 一起做RGB-D SLAM (4) - 半闲居士 - 博客园 <http://www.cnblogs.com/gaoxiang12/p/4669490.html>

**0x04附件：**

**以下是完整代码：**

***文件名：slamBase.py***

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import cv2

import numpy as np

import pcl

class Frame(object):

"""docstring for Frame"""

def \_\_init\_\_(self, RGBFilename, DepthFilename):

super(Frame, self).\_\_init\_\_()

self.rgb = self.ReadImg(RGBFilename)

self.depth = self.ReadImg(DepthFilename)

self.kps, self.des = self.ComputeKPointsAndDescriptors(self.rgb)

def ReadImg(self, filename):

if 'depth' in filename:

img = cv2.imread(filename)

return cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

else:

return cv2.imread(filename)

def ComputeKPointsAndDescriptors(self, rgb):

sift = cv2.xfeatures2d.SIFT\_create()

kps, des = sift.detectAndCompute(rgb, None)

return kps, des

class CameraIntrinsicParameters(object):

"""docstring for CameraIntrinsicParameters"""

def \_\_init\_\_(self, cx, cy, fx, fy, scale):

super(CameraIntrinsicParameters, self).\_\_init\_\_()

self.cx = cx

self.cy = cy

self.fx = fx

self.fy = fy

self.scale = scale

class SolvePnP(object):

"""docstring for SolvePnP"""

def \_\_init\_\_(self, distCoeffs, CameraIntrinsicData, camera, frame1, frame2):

super(SolvePnP, self).\_\_init\_\_()

self.distCoeffs = distCoeffs

self.CameraIntrinsicData = CameraIntrinsicData

self.camera = camera

self.frame1 = frame1

self.frame2 = frame2

self.rvec, self.tvec, self.inliers = self.ResultOfPnP(self.frame1.kps, self.frame2.kps, self.frame1.des, self.frame2.des, self.frame1.depth, self.distCoeffs, self.CameraIntrinsicData, self.camera)

self.T = self.transformMatrix(self.rvec, self.tvec)

def transformMatrix(self, rvec, tvec):

temp = np.matrix([0, 0, 0, 1])

r = np.matrix(rvec)

t = np.matrix(tvec)

dst, jacobian = cv2.Rodrigues(r)

c = np.hstack((dst, t))

T = np.vstack((c, temp))

return T

def ResultOfPnP(self, kp1, kp2, des1, des2, depth, distCoeffs, CameraIntrinsicData, camera):

FLANN\_INDEX\_KDTREE = 0

index\_params = dict(algorithm = FLANN\_INDEX\_KDTREE, trees = 5)

search\_params = dict(checks = 50) # or pass empty dictionary

matcher = cv2.FlannBasedMatcher(index\_params, search\_params)

matches = matcher.match(des1, des2)

print("Find total " + str(len(matches)) + " matches.")

goodMatches = []

minDis = 9999.0

for i in range(0, len(matches)):

if matches[i].distance < minDis:

minDis = matches[i].distance

for i in range(0, len(matches)):

if matches[i].distance < (minDis \* 4):

goodMatches.append(matches[i])

print("good matches = " + str(len(goodMatches)))

pts\_obj = []

pts\_img = []

for i in range(0, len(goodMatches)):

p = kp1[goodMatches[i].queryIdx].pt

d = depth[int(p[1])][int(p[0])]

if d == 0:

pass

else:

pts\_img.append(kp2[goodMatches[i].trainIdx].pt)

pd = point2dTo3d(p[0], p[1], d, camera)

pts\_obj.append(pd)

pts\_obj = np.array(pts\_obj)

pts\_img = np.array(pts\_img)

cameraMatrix = np.matrix(CameraIntrinsicData)

rvec = None

tvec = None

inliers = None

retval, rvec, tvec, inliers = cv2.solvePnPRansac( pts\_obj, pts\_img, cameraMatrix, distCoeffs, useExtrinsicGuess = False, iterationsCount = 100, reprojectionError = 1.76 )

return rvec, tvec, inliers

def addColorToPCDFile(filename, colors):

with open(filename, 'rb') as f:

lines = f.readlines()

lines[2] = lines[2].split('\n')[0] + ' rgb\n'

lines[3] = lines[3].split('\n')[0] + ' 4\n'

lines[4] = lines[4].split('\n')[0] + ' I\n'

lines[5] = lines[5].split('\n')[0] + ' 1\n'

for i in range(11, len(colors) + 11):

lines[i] = lines[i].split('\n')[0] + ' ' + str(colors[i - 11]) + '\n'

with open(filename, 'wb') as fw:

fw.writelines(lines)

def transformPointCloud(src, T):

pointcloud = []

for item in src:

a = list(item)

a.append(1)

a = np.matrix(a)

a = a.reshape((-1, 1))

temp = T \* a

temp = temp.reshape((1, -1))

temp = np.array(temp)

temp = list(temp[0])

pointcloud.append(temp[0:3])

return pointcloud

def addPointCloud(cloud1, cloud2):

cloud = cloud1 + cloud2

cloud = np.array(cloud, dtype = np.float32)

out = pcl.PointCloud()

out.from\_array(cloud)

return out

def point2dTo3d(n, m, d, camera):

z = float(d) / camera.scale

x = (n - camera.cx) \* z / camera.fx

y = (m - camera.cy) \* z / camera.fy

point = np.array([x, y, z], dtype = np.float32)

return point

def imageToPointCloud(RGBFilename, DepthFilename, camera):

rgb = cv2.imread( RGBFilename )

depth = cv2.imread( DepthFilename, cv2.COLOR\_BGR2GRAY )

# ROS中rqt保存的深度摄像头的图片是rgb格式，需要转换成单通道灰度格式

if len(depth[0][0]) == 3:

depth = cv2.cvtColor(depth, cv2.COLOR\_BGR2GRAY)

rows = len(depth)

cols = len(depth[0])

pointcloud = []

colors = []

for m in range(0, rows):

for n in range(0, cols):

d = depth[m][n]

if d == 0:

pass

else:

point = point2dTo3d(n, m, d, camera)

pointcloud.append(point)

b = rgb[m][n][0]

g = rgb[m][n][1]

r = rgb[m][n][2]

color = (r << 16) | (g << 8) | b

colors.append(int(color))

return pointcloud, colors

***文件名：readyaml.py***

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import yaml, sys

import numpy as np

def fixYamlFile(filename):

with open(filename, 'rb') as f:

lines = f.readlines()

if lines[0] != '%YAML 1.0\n':

lines[0] = '%YAML 1.0\n'

for line in lines:

if ' !!opencv-matrix' in line:

lines[lines.index(line)] = line.split(' !!opencv-matrix')[0] + '\n'

with open(filename, 'wb') as fw:

fw.writelines(lines)

def parseYamlFile(filename):

fixYamlFile(filename)

f = open(filename)

x = yaml.load(f)

f.close()

CameraIntrinsicData = np.array(x['camera\_matrix']['data'], dtype = np.float32)

DistortionCoefficients = np.array(x['distortion\_coefficients']['data'], dtype = np.float32)

return (CameraIntrinsicData.reshape(3, 3), DistortionCoefficients)

***文件名：joinPointCloud.py***

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import slamBase

from pcl import save

import readyaml

RGBFileNameList = ['./data/rgb1.png', './data/rgb2.png']

DepthFileNameList = ['./data/depth1.png', './data/depth2.png']

CalibrationDataFile = './calibration/asus/camera.yml'

CloudFilename = './data/cloud.pcd'

def main():

CameraIntrinsicData, DistortionCoefficients = readyaml.parseYamlFile(CalibrationDataFile)

camera = slamBase.CameraIntrinsicParameters(CameraIntrinsicData[0][2], CameraIntrinsicData[1][2], CameraIntrinsicData[0][0], CameraIntrinsicData[1][1], 1000.0)

frame1 = slamBase.Frame(RGBFileNameList[0], DepthFileNameList[0])

frame2 = slamBase.Frame(RGBFileNameList[1], DepthFileNameList[1])

pnp = slamBase.SolvePnP(DistortionCoefficients, CameraIntrinsicData, camera, frame1, frame2)

p0, c0 = slamBase.imageToPointCloud(RGBFileNameList[0], DepthFileNameList[0], camera)

p1, c1 = slamBase.imageToPointCloud(RGBFileNameList[1], DepthFileNameList[1], camera)

colors = c0 + c1

p = slamBase.transformPointCloud(p0, pnp.T)

pointcloud = slamBase.addPointCloud(p, p1)

save(pointcloud, CloudFilename, format = 'pcd')

slamBase.addColorToPCDFile(CloudFilename, colors)

if \_\_name\_\_ == '\_\_main\_\_':

main()

**第四篇：视觉里程计（Python版）**

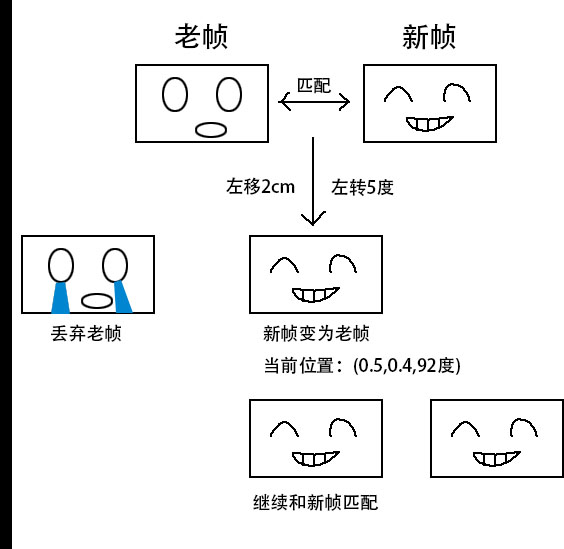
**0x00 准备：**

上一篇我们已经成功的实现了一个只有两帧的简单slam程序，但是真正的slam程序是不会只有两帧画面，本篇将会使用更多的数据，同时制作一个简易的视觉里程计。

本篇所使用的数据来自nyuv2数据集

（<http://cs.nyu.edu/~silberman/datasets/nyu_depth_v2.html>）。完整数据一共700多张图片，在本篇中只使用其中的20张作为测试样例（图片越多，程序需要的时间越久），具体数据参见github（<https://github.com/zsirui/slam-python>）。

视觉里程计，简单来说，就是用新帧数据和上一帧数据对比匹配，通过匹配运算，估算摄像头的运动轨迹，然后一遍遍的重复将源源不断的新数据与老数据匹配计算得到的运动数据累加起来，得到的就是相机的连续运动。整个过程的示意图如下图所示：（图片来自：<http://www.cnblogs.com/gaoxiang12/p/4719156.html>）



在上一篇中，我们已经将拼接点云的代码封装为了slamBase.py的自定义库，我们将使用这个自定义库来实现一个简易的Visual Odometry（视觉里程计）程序。

**0x01 代码实现：**

首先我们将上一篇中的joinPointCloud.py进行改写，增加部分常量定义：

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import slamBase

import pcl

import readyaml

from numpy import array, float32, pi

from os import path

from cv2 import norm

RGBFileNamePath = './data/rgb\_png'

DepthFileNamePath = './data/depth\_png'

CalibrationDataFile = './calibration/camera.yml'

CloudFilename = './data/cloud.pcd'

# 点云分辨率

GRIDSIZE = 0.02

# 起始与终止索引

START\_INDEX = 1

END\_INDEX = 20

# 最小匹配数量

MIN\_GOOD\_MATCH = 10

# 最小内点

MIN\_INLIERS = 5

# 最大运动误差

MAX\_NORM = 0.3

C = array([[518.0,0,325.0],[0,519.0,253.5],[0,0,1]])

这里数据一共20组（初期测试可以选择较少数据，数据量越大，程序所需时间越长，本人在性能较好的计算机上700组数据依旧需要将近**四小时半**的时间），每组数据分别包含一张RGB图像和深度图像，分别保存在/data/rgb\_png和/data/depth\_png路径下，相机标定文件存放在/calibration/路径下，文件名为camera.yml，（当使用自己数据的时候，这个文件才会用到）GRIDSIZE为滤波器使用的点云分辨率，START\_INDEX和END\_INDEX为数据索引起始，MIN\_GOOD\_MATCH为最小匹配数量，MIN\_INLIERS为最小内点，MAX\_NORM为最大运动误差。修改这些常量可以得到不同的点云结果，这里不做细致研究。定义normofTransform函数，用于度量运动的大小：

def normofTransform(rvec, tvec):

return abs(min(norm(rvec), pi \* 2 - norm(rvec))) + abs(norm(tvec))

定义joinPointCloud函数，将新帧数据和上一帧数据进行匹配计算并合并点云：

def joinPointCloud(pointCloud, color, RGBFileName, DepthFileName, lastFrame, CameraIntrinsicData, DistortionCoefficients, camera):

currentFrame = slamBase.Frame(RGBFileName, DepthFileName)

pnp = slamBase.SolvePnP(DistortionCoefficients, CameraIntrinsicData, camera, lastFrame, currentFrame)

try:

len(pnp.inliers)

except:

pass

else:

if len(pnp.inliers) < MIN\_INLIERS:

pass

else:

Norm = normofTransform(pnp.rvec, pnp.tvec)

print('norm = ' + str(Norm))

if Norm >= MAX\_NORM:

pass

else:

p0, c0 = slamBase.imageToPointCloud(RGBFileName, DepthFileName, camera)

colors = color + c0

p = slamBase.transformPointCloud(p0, pnp.T)

pointCloud = slamBase.addPointCloud(pointCloud.to\_list(), p)

voxel = pointCloud.make\_voxel\_grid\_filter()

voxel.set\_leaf\_size( GRIDSIZE, GRIDSIZE, GRIDSIZE )

pointCloud = voxel.filter()

lastFrame = currentFrame

return pointCloud, lastFrame, colors

这里需要注意一点，python版的pcl库的Voxel滤波器和C++版的代码有所不同，python版中，Voxel滤波器的初始化被内置到了PointCloud这个类中，通过PointCloud.make\_voxel\_grid\_filter()实现滤波器初始化。部分方法名和C++代码完全不同，遇到不知道的地方可以在python命令行中输入help命令来查看（第三方库需要先用import引入后再用help命令查看）。

接下来是最关键的VisualOdometry实现：

def visualOdometry():

CameraIntrinsicData, DistortionCoefficients = readyaml.parseYamlFile(CalibrationDataFile)

DistortionCoefficients = array([0,0,0,0], dtype = float32)

camera = slamBase.CameraIntrinsicParameters(C[0][2], C[1][2], C[0][0], C[1][1], 1000.0)

frame = slamBase.Frame(path.join(RGBFileNamePath, str(START\_INDEX) + '.png'), path.join(DepthFileNamePath, str(START\_INDEX) + '.png'))

p, colors = slamBase.imageToPointCloud(path.join(RGBFileNamePath, str(START\_INDEX) + '.png'), path.join(DepthFileNamePath, str(START\_INDEX) + '.png'), camera)

pcd = pcl.PointCloud()

pcd.from\_array(array(p, dtype = float32))

for i in range(START\_INDEX, END\_INDEX):

try:

pcd, frame, colors = joinPointCloud(pcd, colors, path.join(RGBFileNamePath, str(i + 1) + '.png'), path.join(DepthFileNamePath, str(i + 1) + '.png'), frame, C, DistortionCoefficients, camera)

except:

pass

pcl.save(pcd, CloudFilename, format = 'pcd')

这样简单的一个VisualOdometry（视觉里程计）程序就基本实现了。然而，查看了点云后就会发现，点云并没有颜色。这是python版pcl接口的问题，python版pcl中对PointCloud封装的时候，并没有将颜色（RGB）封装进去，这样通过PointCloud产生的点云，点云的每个点仅仅只有x, y, z坐标，再通过Voxel滤波器滤波后，更无法将点云的颜色添加回去。考虑到后续代码所用到的g2o库并没有python接口，所以打算抛弃python版的pcl接口，自行编写一个C++代码，并为python代码留出相应接口以便调用。然而由于作者本人知识所限，暂时无法对pcl库进行大面积的接口改造。而后续的g2o库扩展接口更为复杂，故后续不再介绍使用g2o库对视觉里程计的优化。

目前C++和python的混合编程一般有以下几种方式：python自带的ctypes库，需要对C++代码编写C风格接口；SWIG，一个可以分析代码并自动生成python接口的库，需要自行编写特殊的解释文档；Boost.Python是Boost库中的一套用于C++和python交互的接口，需要针对C++源码编写一段接口转换代码。现在简单介绍一下使用Boost.Python库对pcl库进行接口改造，首先是头文件和命名空间及类型定义：

#include <iostream>

//PCL

#include <pcl/io/io.h>

#include <pcl/io/pcd\_io.h>

#include <pcl/point\_types.h>

#include <pcl/PCLHeader.h>

#include <pcl/common/transforms.h>

//boost

#include <boost/python.hpp>

#include <boost/python/list.hpp>

//Eigen

#include <Eigen/StdVector>

#include <Eigen/Geometry>

using namespace std;

using namespace boost::python;

// 类型定义

typedef pcl::PointXYZRGBA PointT;

typedef pcl::PointCloud<PointT> PointCloud;

typedef boost::shared\_ptr < pcl::PointXYZRGBA > PointXYZRGBA\_ptr;

typedef boost::shared\_ptr < PointCloud::Ptr > PointCloud\_ptr;

typedef std::vector<PointT, Eigen::aligned\_allocator<PointT> > VectorType;

Boost库中的Python模块可以自动将C++的数据类型转换成Python可识别的类型，不过需要编写一些简单的转换语句。例如现在我们要将PointCloud这个类进行接口转换：

BOOST\_PYTHON\_MODULE(lib\_pcl)

{

register\_ptr\_to\_python <PointXYZRGBA\_ptr>();

register\_ptr\_to\_python <PointCloud\_ptr>();

def("showList", showList);

class\_<pcl::PCLHeader>("PCLHeader")

.def\_readwrite("seq", &pcl::PCLHeader::seq)

.def\_readwrite("stamp", &pcl::PCLHeader::stamp)

.def\_readwrite("frame\_id", &pcl::PCLHeader::frame\_id);

class\_<pcl::PointXYZRGBA>("PointXYZRGBA")

.def\_readwrite("x", &pcl::PointXYZRGBA::x)

.def\_readwrite("y", &pcl::PointXYZRGBA::y)

.def\_readwrite("z", &pcl::PointXYZRGBA::z)

.def\_readwrite("r", &pcl::PointXYZRGBA::r)

.def\_readwrite("g", &pcl::PointXYZRGBA::g)

.def\_readwrite("b", &pcl::PointXYZRGBA::b)

.def\_readwrite("a", &pcl::PointXYZRGBA::a)

.def\_readwrite("rgba", &pcl::PointXYZRGBA::rgba);

class\_<VectorType>("VectorType", init<>());

class\_<PointCloud>("PointCloud", init<>())

.def\_readwrite("width", &PointCloud::width)

.def\_readwrite("height", &PointCloud::height)

.def\_readwrite("is\_dense", &PointCloud::is\_dense)

.def\_readwrite("header", &PointCloud::header)

.def\_readwrite("points", &PointCloud::points)

.def(self + PointCloud())

.def("size", &PointCloud::size)

.def("Ptr", &PointCloud::makeShared)

.def("push\_back", &PointCloud::push\_back)

.def("clear", &PointCloud::clear)

.def("resize", &PointCloud::resize);

}

通过查询pcl源码，我们可以得到PointCloud类的详细定义，这里我们只需要将源码中public中公开的变量和方法接口通过Boost.Python进行转换即可。class\_<T>()就是将C++的类/结构体转换为Python的类的方法。这里，T对应的是类型。.def\_readwrite用来向Python类中添加Public变量，.def\_readonly用来添加Private变量，.def用来添加类中的方法。

BOOST\_PYTHON\_MODULE对应自定义的Python模块的名字，这里我们定义模块名为lib\_pcl，showList是一个测试函数，传递的参数是一个Python的list，通过toPointXYZRGBA函数，可以将对应的数据在C++中存为pcl:: PointXYZRGBA类型并打印输出，showList的函数内容如下：

void showList(const boost::python::list& pyList)

{

PointT p;

p = toPointXYZRGBA(boost::python::extract<long double>(pyList[0]), boost::python::extract<long double>(pyList[1]), boost::python::extract<long double>(pyList[2]), boost::python::extract<uint8\_t>(pyList[3]), boost::python::extract<uint8\_t>(pyList[4]), boost::python::extract<uint8\_t>(pyList[5]));

cout<<"p.x "<<p.x<<endl;

cout<<"p.y "<<p.y<<endl;

cout<<"p.z "<<p.z<<endl;

cout<<"p.rgba "<<p.rgba<<endl;

}

PointT toPointXYZRGBA(long double x, long double y, long double z, uint8\_t r, uint8\_t g, uint8\_t b)

{

PointT p;

p.x = x;

p.y = y;

p.z = z;

p.r = r;

p.g = g;

p.b = b;

return p;

}

最后将代码保存为lib\_pcl.cpp（文件名一定要和代码中的模块名一致），在命令行中输入以下命令：

g++ -c -fPIC lib\_pcl.cpp -o lib\_pcl.o

g++ -shared -Wl,-soname,lib\_pcl.so -o lib\_pcl.so lib\_pcl.o -lpython2.7 -lboost\_python -lboost\_system

最后会在目录中生成lib\_pcl.so文件，在python命令行中输入import lib\_pcl就可以将编写好的C++库引入到python中了：

Python 2.7.6 (default, Oct 26 2016, 20:30:19)

[GCC 4.8.4] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>> import lib\_pcl

>>> PointT = lib\_pcl.PointXYZRGBA()

>>> PointT.x = 1

>>> PointT.y = 2

>>> PointT.z = 3

>>> PointT.r = 128

>>> PointT.g = 255

>>> PointT.b = 127

>>> PointT.rgba

4286644095

>>> PointT.a

255

>>> lib\_pcl.showList([1,2,3,128,255,127])

p.x 1

p.y 2

p.z 3

p.rgba 4286644095

**0x03 附件**

**以下是完整代码：**

***文件名：slamBase.py***

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import cv2

import numpy as np

import pcl, lib\_pcl

class Frame(object):

"""docstring for Frame"""

def \_\_init\_\_(self, RGBFilename, DepthFilename):

super(Frame, self).\_\_init\_\_()

self.rgb = self.ReadImg(RGBFilename)

self.depth = self.ReadImg(DepthFilename)

self.kps, self.des = self.ComputeKPointsAndDescriptors(self.rgb)

def ReadImg(self, filename):

if 'depth' in filename:

img = cv2.imread(filename)

return cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

else:

return cv2.imread(filename)

def ComputeKPointsAndDescriptors(self, rgb):

sift = cv2.xfeatures2d.SIFT\_create()

kps, des = sift.detectAndCompute(rgb, None)

return kps, des

class CameraIntrinsicParameters(object):

"""docstring for CameraIntrinsicParameters"""

def \_\_init\_\_(self, cx, cy, fx, fy, scale):

super(CameraIntrinsicParameters, self).\_\_init\_\_()

self.cx = cx

self.cy = cy

self.fx = fx

self.fy = fy

self.scale = scale

class SolvePnP(object):

"""docstring for SolvePnP"""

def \_\_init\_\_(self, distCoeffs, CameraIntrinsicData, camera, frame1, frame2):

super(SolvePnP, self).\_\_init\_\_()

self.distCoeffs = distCoeffs

self.CameraIntrinsicData = CameraIntrinsicData

self.camera = camera

self.frame1 = frame1

self.frame2 = frame2

self.rvec, self.tvec, self.inliers = self.ResultOfPnP(self.frame1.kps, self.frame2.kps, self.frame1.des, self.frame2.des, self.frame1.depth, self.distCoeffs, self.CameraIntrinsicData, self.camera)

self.T = self.transformMatrix(self.rvec, self.tvec)

def transformMatrix(self, rvec, tvec):

temp = np.matrix([0, 0, 0, 1])

r = np.matrix(rvec)

t = np.matrix(tvec)

dst, jacobian = cv2.Rodrigues(r)

c = np.hstack((dst, t))

T = np.vstack((c, temp))

return T

def ResultOfPnP(self, kp1, kp2, des1, des2, depth, distCoeffs, CameraIntrinsicData, camera):

FLANN\_INDEX\_KDTREE = 0

index\_params = dict(algorithm = FLANN\_INDEX\_KDTREE, trees = 5)

search\_params = dict(checks = 50) # or pass empty dictionary

matcher = cv2.FlannBasedMatcher(index\_params, search\_params)

matches = matcher.match(des1, des2)

print("Find total " + str(len(matches)) + " matches.")

goodMatches = []

minDis = 9999.0

for i in range(0, len(matches)):

if matches[i].distance < minDis:

minDis = matches[i].distance

for i in range(0, len(matches)):

if matches[i].distance < (minDis \* 4):

goodMatches.append(matches[i])

print("good matches = " + str(len(goodMatches)))

pts\_obj = []

pts\_img = []

for i in range(0, len(goodMatches)):

p = kp1[goodMatches[i].queryIdx].pt

d = depth[int(p[1])][int(p[0])]

if d == 0:

pass

else:

pts\_img.append(kp2[goodMatches[i].trainIdx].pt)

pd = point2dTo3d(p[0], p[1], d, camera)

pts\_obj.append(pd)

pts\_obj = np.array(pts\_obj)

pts\_img = np.array(pts\_img)

cameraMatrix = np.matrix(CameraIntrinsicData)

rvec = None

tvec = None

inliers = None

retval, rvec, tvec, inliers = cv2.solvePnPRansac( pts\_obj, pts\_img, cameraMatrix, distCoeffs, useExtrinsicGuess = False, iterationsCount = 100, reprojectionError = 1 )

return rvec, tvec, inliers

def addColorToPCDFile(filename, colors):

with open(filename, 'rb') as f:

lines = f.readlines()

lines[2] = lines[2].split('\n')[0] + ' rgb\n'

lines[3] = lines[3].split('\n')[0] + ' 4\n'

lines[4] = lines[4].split('\n')[0] + ' I\n'

lines[5] = lines[5].split('\n')[0] + ' 1\n'

for i in range(11, len(colors) + 11):

lines[i] = lines[i].split('\n')[0] + ' ' + str(colors[i - 11]) + '\n'

with open(filename, 'wb') as fw:

fw.writelines(lines)

def transformPointCloud(src, T):

pointcloud = []

cloud = pcl.PointCloud()

for item in src:

a = list(item)

a.append(1)

a = np.matrix(a)

a = a.reshape((-1, 1))

temp = T \* a

temp = temp.reshape((1, -1))

temp = np.array(temp)

temp = list(temp[0])

pointcloud.append(temp[0:3])

return pointcloud

def addPointCloud(cloud1, cloud2):

cloud = cloud1 + cloud2

cloud = np.array(cloud, dtype = np.float32)

out = pcl.PointCloud()

out.from\_array(cloud)

return out

def point2dTo3d(n, m, d, camera):

z = float(d) / camera.scale

x = (n - camera.cx) \* z / camera.fx

y = (m - camera.cy) \* z / camera.fy

point = np.array([x, y, z], dtype = np.float32)

return point

def point2dTo3dwithColors(n, m, d, r, g, b, camera):

point = lib\_pcl.PointXYZRGBA()

point.z = float(d) / camera.scale

point.x = (n - camera.cx) \* z / camera.fx

point.y = (m - camera.cy) \* z / camera.fy

point.r, point.g, point.b = r, g, b

return point

def imageToPointCloud(RGBFilename, DepthFilename, camera):

rgb = cv2.imread( RGBFilename )

depth = cv2.imread( DepthFilename, cv2.COLOR\_BGR2GRAY )

# ROS中rqt保存的深度摄像头的图片是rgb格式，需要转换成单通道灰度格式

try:

len(depth[0][0])

except:

pass

else:

depth = cv2.cvtColor(depth, cv2.COLOR\_BGR2GRAY)

rows = len(depth)

cols = len(depth[0])

pointcloud = []

colors = []

for m in range(0, rows):

for n in range(0, cols):

d = depth[m][n]

if d == 0:

pass

else:

point = point2dTo3d(n, m, d, camera)

pointcloud.append(point)

b = rgb[m][n][0]

g = rgb[m][n][1]

r = rgb[m][n][2]

color = (r << 16) | (g << 8) | b

colors.append(int(color))

pointcloud.append(point)

return pointcloud, colors

def imageToPointCloudwithColors(RGBFilename, DepthFilename, camera):

rgb = cv2.imread( RGBFilename )

depth = cv2.imread( DepthFilename, cv2.COLOR\_BGR2GRAY )

# ROS中rqt保存的深度摄像头的图片是rgb格式，需要转换成单通道灰度格式

try:

len(depth[0][0])

except:

pass

else:

depth = cv2.cvtColor(depth, cv2.COLOR\_BGR2GRAY)

rows = len(depth)

cols = len(depth[0])

pointcloud = lib\_pcl.PointCloud()

for m in range(0, rows):

for n in range(0, cols):

d = depth[m][n]

if d == 0:

pass

else:

b = rgb[m][n][0]

g = rgb[m][n][1]

r = rgb[m][n][2]

color = (r << 16) | (g << 8) | b

point = point2dTo3dwithColors(n, m, d, int(color), camera)

pointcloud.push\_back(point)

return pointcloud

***文件名：readyaml.py***

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import yaml, sys

import numpy as np

def fixYamlFile(filename):

with open(filename, 'rb') as f:

lines = f.readlines()

if lines[0] != '%YAML 1.0\n':

lines[0] = '%YAML 1.0\n'

for line in lines:

if ' !!opencv-matrix' in line:

lines[lines.index(line)] = line.split(' !!opencv-matrix')[0] + '\n'

with open(filename, 'wb') as fw:

fw.writelines(lines)

def parseYamlFile(filename):

fixYamlFile(filename)

f = open(filename)

x = yaml.load(f)

f.close()

CameraIntrinsicData = np.array(x['camera\_matrix']['data'], dtype = np.float32)

DistortionCoefficients = np.array(x['distortion\_coefficients']['data'], dtype = np.float32)

return (CameraIntrinsicData.reshape(3, 3), DistortionCoefficients)

***文件名：joinPointCloud.py***

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import slamBase

import pcl

import readyaml

from numpy import array, float32, pi

from os import path

from cv2 import norm

RGBFileNamePath = './data/rgb\_png'

DepthFileNamePath = './data/depth\_png'

CalibrationDataFile = './calibration/asus/camera.yml'

CloudFilename = './data/cloud.pcd'

# 点云分辨率

GRIDSIZE = 0.02

# 起始与终止索引

START\_INDEX = 1

END\_INDEX = 700

# 最小匹配数量

MIN\_GOOD\_MATCH = 10

# 最小内点

MIN\_INLIERS = 5

# 最大运动误差

MAX\_NORM = 0.3

C = array([[518.0,0,325.0],[0,519.0,253.5],[0,0,1]])

def normofTransform(rvec, tvec):

return abs(min(norm(rvec), pi \* 2 - norm(rvec))) + abs(norm(tvec))

def joinPointCloud(pointCloud, color, RGBFileName, DepthFileName, lastFrame, CameraIntrinsicData, DistortionCoefficients, camera):

currentFrame = slamBase.Frame(RGBFileName, DepthFileName)

pnp = slamBase.SolvePnP(DistortionCoefficients, CameraIntrinsicData, camera, lastFrame, currentFrame)

try:

len(pnp.inliers)

except:

pass

else:

if len(pnp.inliers) < MIN\_INLIERS:

pass

else:

Norm = normofTransform(pnp.rvec, pnp.tvec)

print('norm = ' + str(Norm))

if Norm >= MAX\_NORM:

pass

else:

p0, c0 = slamBase.imageToPointCloud(RGBFileName, DepthFileName, camera)

colors = color + c0

p = slamBase.transformPointCloud(p0, pnp.T)

pointCloud = slamBase.addPointCloud(pointCloud.to\_list(), p)

voxel = pointCloud.make\_voxel\_grid\_filter()

voxel.set\_leaf\_size( GRIDSIZE, GRIDSIZE, GRIDSIZE )

pointCloud = voxel.filter()

lastFrame = currentFrame

return pointCloud, lastFrame, colors

def visualOdometry():

CameraIntrinsicData, DistortionCoefficients = readyaml.parseYamlFile(CalibrationDataFile)

DistortionCoefficients = array([0,0,0,0], dtype = float32)

camera = slamBase.CameraIntrinsicParameters(C[0][2], C[1][2], C[0][0], C[1][1], 1000.0)

frame = slamBase.Frame(path.join(RGBFileNamePath, str(START\_INDEX) + '.png'), path.join(DepthFileNamePath, str(START\_INDEX) + '.png'))

p, colors = slamBase.imageToPointCloud(path.join(RGBFileNamePath, str(START\_INDEX) + '.png'), path.join(DepthFileNamePath, str(START\_INDEX) + '.png'), camera)

pcd = pcl.PointCloud()

pcd.from\_array(array(p, dtype = float32))

for i in range(START\_INDEX, END\_INDEX):

try:

pcd, frame, colors = joinPointCloud(pcd, colors, path.join(RGBFileNamePath, str(i + 1) + '.png'), path.join(DepthFileNamePath, str(i + 1) + '.png'), frame, C, DistortionCoefficients, camera)

except:

pass

pcl.save(pcd, CloudFilename, format = 'pcd')

# slamBase.addColorToPCDFile(CloudFilename, colors)

if \_\_name\_\_ == '\_\_main\_\_':

visualOdometry()

***文件名：lib\_pcl.cpp***

#include <iostream>

//PCL

#include <pcl/io/io.h>

#include <pcl/io/pcd\_io.h>

#include <pcl/point\_types.h>

#include <pcl/PCLHeader.h>

#include <pcl/common/transforms.h>

#include <pcl/visualization/cloud\_viewer.h>

//boost

#include <boost/python.hpp>

#include <boost/python/list.hpp>

//Eigen

#include <Eigen/StdVector>

#include <Eigen/Geometry>

using namespace std;

using namespace boost::python;

// 类型定义

typedef pcl::PointXYZRGBA PointT;

typedef pcl::PointCloud<PointT> PointCloud;

typedef boost::shared\_ptr < pcl::PointXYZRGBA > PointXYZRGBA\_ptr;

typedef boost::shared\_ptr < PointCloud::Ptr > PointCloud\_ptr;

typedef std::vector<PointT, Eigen::aligned\_allocator<PointT> > VectorType;

PointT toPointXYZRGBA(long double x, long double y, long double z, uint8\_t r, uint8\_t g, uint8\_t b);

void showList(const boost::python::list& pyList);

PointT toPointXYZRGBA(long double x, long double y, long double z, uint8\_t r, uint8\_t g, uint8\_t b)

{

PointT p;

p.x = x;

p.y = y;

p.z = z;

p.r = r;

p.g = g;

p.b = b;

return p;

}

void showList(const boost::python::list& pyList)

{

PointT p;

p = toPointXYZRGBA(boost::python::extract<long double>(pyList[0]), boost::python::extract<long double>(pyList[1]), boost::python::extract<long double>(pyList[2]), boost::python::extract<uint8\_t>(pyList[3]), boost::python::extract<uint8\_t>(pyList[4]), boost::python::extract<uint8\_t>(pyList[5]));

cout<<"p.x "<<p.x<<endl;

cout<<"p.y "<<p.y<<endl;

cout<<"p.z "<<p.z<<endl;

cout<<"p.rgba "<<p.rgba<<endl;

}

BOOST\_PYTHON\_MODULE(lib\_pcl)

{

register\_ptr\_to\_python <PointXYZRGBA\_ptr>();

register\_ptr\_to\_python <PointCloud\_ptr>();

def("showList", showList);

class\_<pcl::PCLHeader>("PCLHeader")

.def\_readwrite("seq", &pcl::PCLHeader::seq)

.def\_readwrite("stamp", &pcl::PCLHeader::stamp)

.def\_readwrite("frame\_id", &pcl::PCLHeader::frame\_id);

class\_<pcl::PointXYZRGBA>("PointXYZRGBA")

.def\_readwrite("x", &pcl::PointXYZRGBA::x)

.def\_readwrite("y", &pcl::PointXYZRGBA::y)

.def\_readwrite("z", &pcl::PointXYZRGBA::z)

.def\_readwrite("r", &pcl::PointXYZRGBA::r)

.def\_readwrite("g", &pcl::PointXYZRGBA::g)

.def\_readwrite("b", &pcl::PointXYZRGBA::b)

.def\_readwrite("a", &pcl::PointXYZRGBA::a)

.def\_readwrite("rgba", &pcl::PointXYZRGBA::rgba);

class\_<VectorType>("VectorType", init<>());

class\_<PointCloud>("PointCloud", init<>())

.def\_readwrite("width", &PointCloud::width)

.def\_readwrite("height", &PointCloud::height)

.def\_readwrite("is\_dense", &PointCloud::is\_dense)

.def\_readwrite("header", &PointCloud::header)

.def\_readwrite("points", &PointCloud::points)

.def(self + PointCloud())

.def("size", &PointCloud::size)

.def("Ptr", &PointCloud::makeShared)

.def("push\_back", &PointCloud::push\_back)

.def("clear", &PointCloud::clear)

.def("resize", &PointCloud::resize);

}