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/

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://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

git clone https://github.com/Itseez/opencv_contrib.git

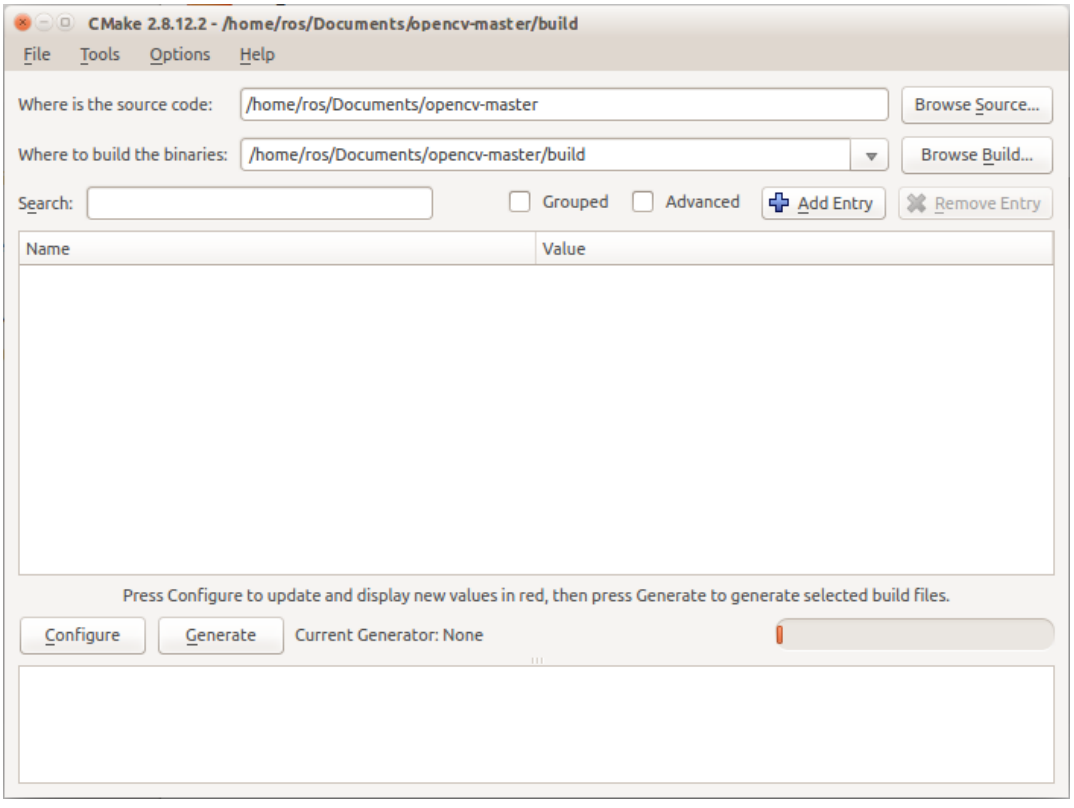
进入 OpenCV 源代码目录，创建 build 文件夹：

cd ~/<my_working_directiory>/opencv

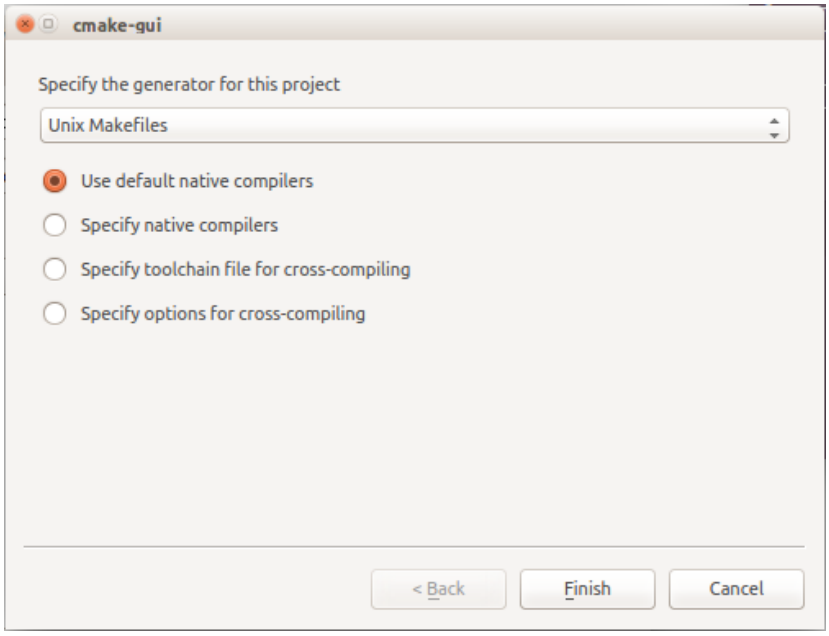
mkdir build

使用 cmake-gui 生成编译文件：

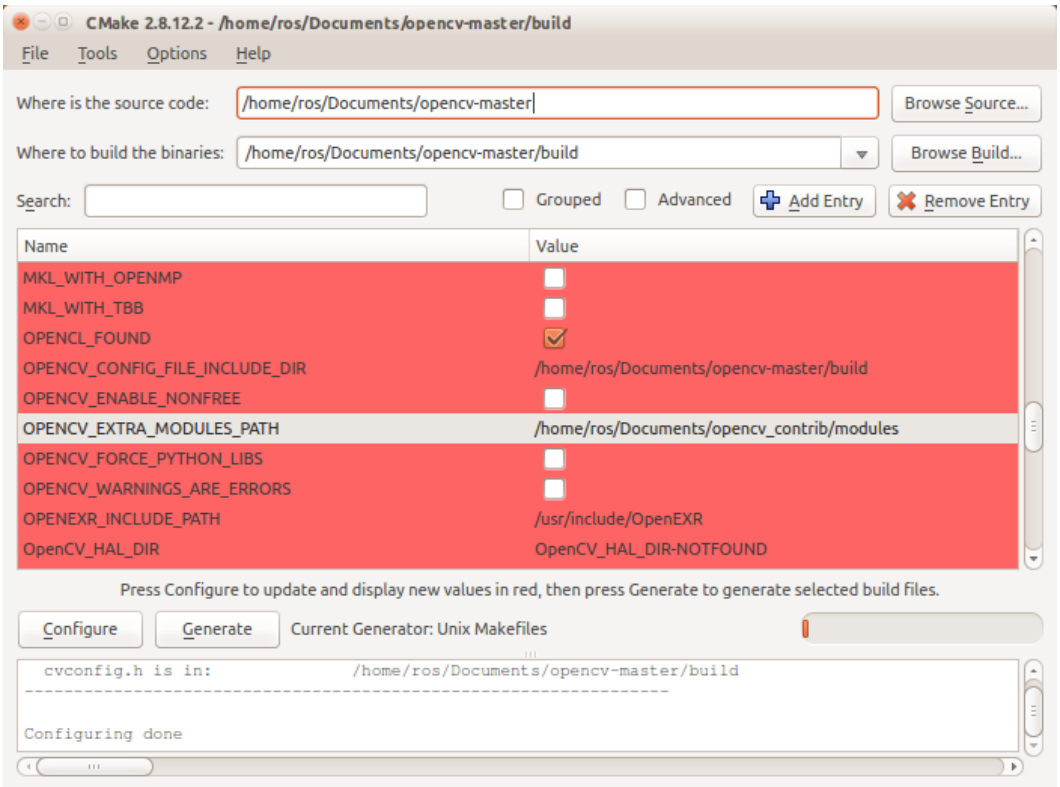
打开 cmake-gui 后，在 source code 栏选择 OpenCV 源代码所在目录（例如：/home/username/workplace/opencv），在 build 栏中填写刚才新建的 build 文件夹（例如：/home/username/workplace/opencv/build）如下图所示：



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
```

2） 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
```

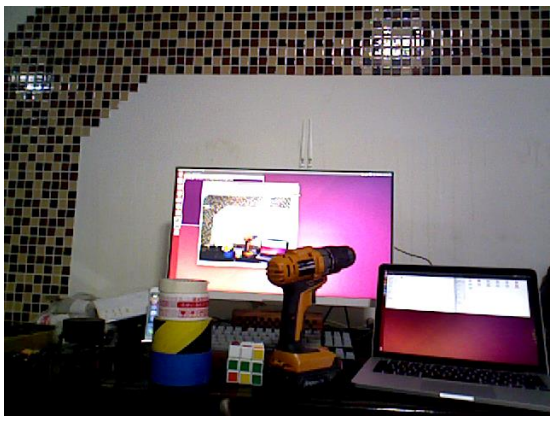
如果 kinect 连接在电脑上，可以通过输入下面命令：

```
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$\boldsymbol{C} = [[518.0, 0, 325.0], [0, 519.0, 253.5], [0, 0, 1]]$

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：

$$C = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

其中，f_x，f_y指相机在 x 轴和 y 轴上的焦距，c_x，c_y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 8×6 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 10~20 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 ]
```

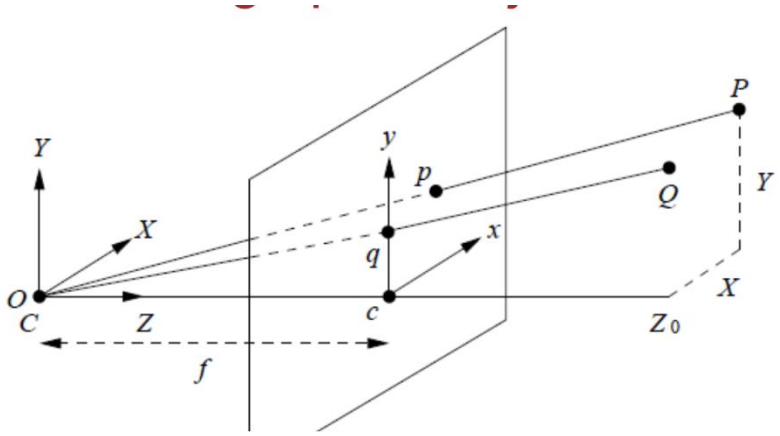
这里，camera_matrix 就是我们所需要的相机内参。我们可以使用 python 的 yaml（如果提示 **ImportError: No module named yaml**，可以使用 **sudo pip install yaml** 安装）库对 yml 文件进行读取解析操作。代码如下：

```
#!/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 3×3 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：

$$u = \frac{x \cdot f_x}{z} + c_x$$

$$v = \frac{y \cdot f_y}{z} + c_y$$

$$d = z \cdot s$$

其中，f_x，f_y指相机在x轴和y轴上的焦距，c_x，c_y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：

$$z = d / s$$

$$x = (u - c_x) \cdot z / f_x$$

$$y = (v - c_y) \cdot z / f_y$$

这里，我们把f_x，f_y，c_x，c_y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：

$$s \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = C \cdot \left(R \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} + t \right)$$

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$\boldsymbol{F_1}$和$\boldsymbol{F_2}$，并且我们获取了相应的特征点：

$$\boldsymbol{P} = \{\boldsymbol{P_1}, \boldsymbol{P_2}, \dots, \boldsymbol{P_n}\} \in \boldsymbol{F_1}$$

$$\boldsymbol{Q} = \{\boldsymbol{Q_1}, \boldsymbol{Q_2}, \dots, \boldsymbol{Q_n}\} \in \boldsymbol{F_2}$$

现在要通过这两组特征点来求出一个旋转矩阵$\boldsymbol{R}$和位移矢量$\boldsymbol{t}$，使得：

$$\forall \boldsymbol{i}, \boldsymbol{P_i} = \boldsymbol{R_{Q_i}} + \boldsymbol{t}$$

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$\boldsymbol{R}$,$\boldsymbol{t}$:

$$\sum_{\boldsymbol{i=1}}^{\boldsymbol{N}} \boldsymbol{\min_{R,t}} \left\| \boldsymbol{P_i} - (\boldsymbol{R_{Q_i}} + \boldsymbol{t}) \right\|_{\boldsymbol{2}}$$

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。

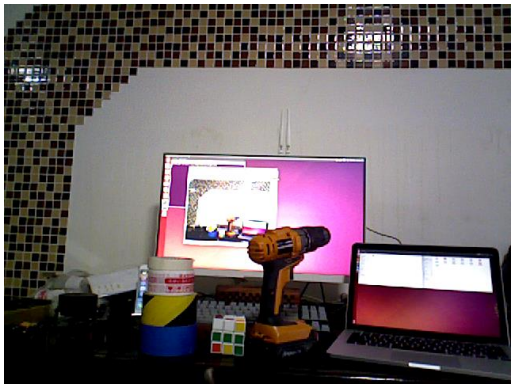
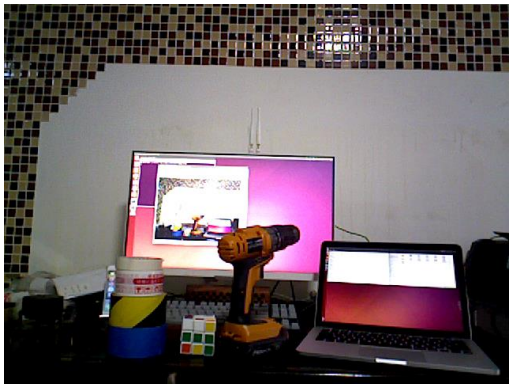
0x02 图像配准（编程实现）

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。

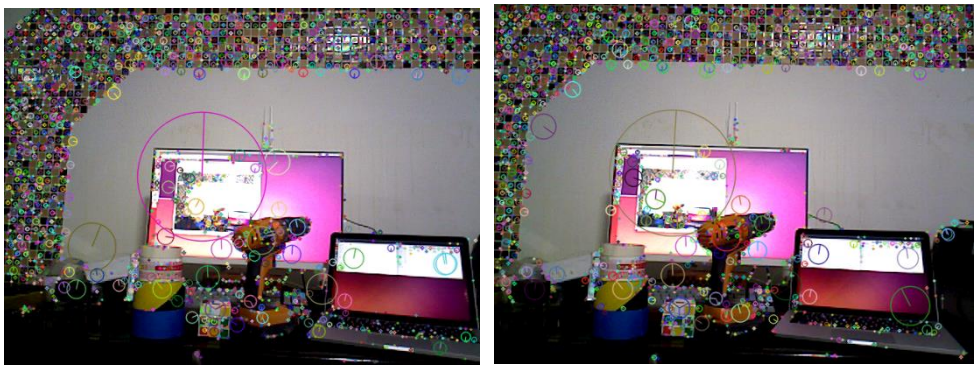



在这里，我们会使用 OpenCV 中的 SIFT 算法来进行特征提取并计算描述子。在 OpenCV3.0 以后的版本中，SIFT 算法被移动到了 OpenCV_Contrib 库里面，需要在编译 OpenCV 的时候指定这个库的路径（OPENCV_EXTRA_MODULES_PATH 键值）后才能使用。

```
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：

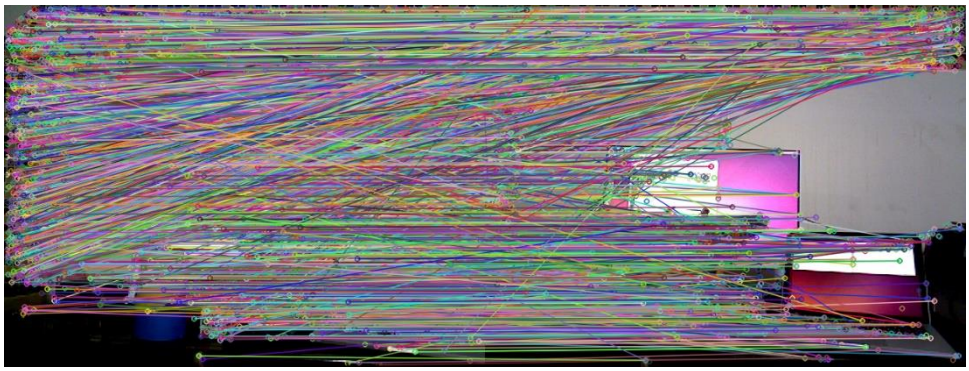
```
    imgShow = None
    imgShow = cv2.drawKeypoints(rgb1, kp1, imgShow, flags =
cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    cv2.imshow( "keypoints_1", imgShow )
    cv2.imwrite( "./data/keypoints_1.png", imgShow )
    cv2.waitKey(0)

    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：

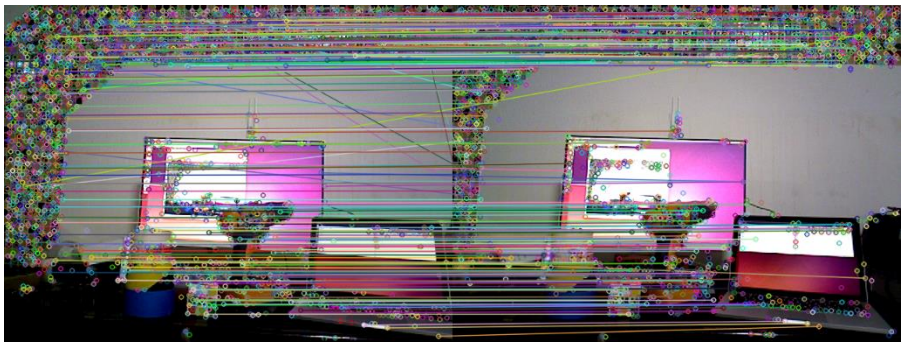
```
FLANN_INDEX_KDTREE = 0
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_params = dict(checks = 50)      # or pass empty dictionary
matcher = cv2.FlannBasedMatcher(index_params, search_params)
matches = matcher.match(des1, des2)
print("Find total " + str(len(matches)) + " matches.")

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$\boldsymbol{R}$和位移矢量$\boldsymbol{t}$：

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）：

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 $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$.

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()) that, together with tvec , brings points from the model coordinate system to the camera coordinate system.

tvec – Output translation vector.

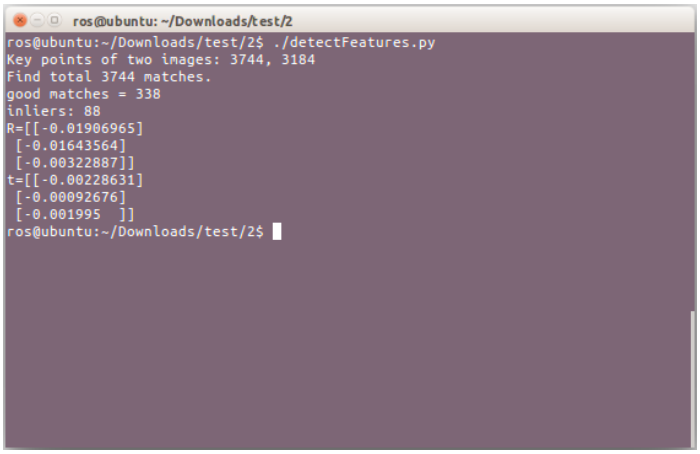
useExtrinsicGuess – Parameter used for SOLVEPNP_ITERATIVE. If true (1), the

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()).

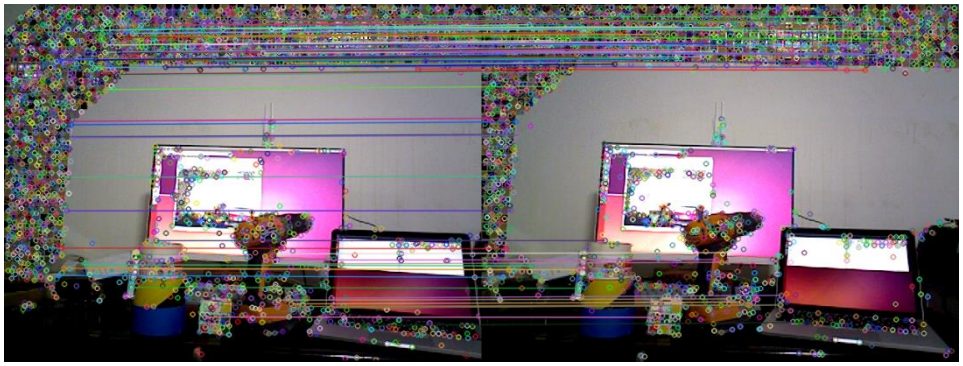
求解 PnP 需要输入一组匹配好的三维点 **objectPoints** 和一组二维图像点 **imagePoints**，返回的结果是旋转向量 rvec 和平移向量 tvec，cameraMatrix 是相机内参矩阵，通过标定相机获得，distCoeffs 是相机畸变参数，标定后会和相机内参矩阵同时生成写入文件中，我们可以使用 readyaml.py 来获取相机的畸变参数，reprojectionError 参数在使用示例标定相机时最后会被打印在屏幕上 。接下来我们可以将筛选过后的 goodmatches 通过上一章的函数转变为三维点。

```
    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)

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**)：

$$\boldsymbol{T} = \begin{bmatrix} \boldsymbol{R_{3\times3}} & \boldsymbol{t_{3\times1}} \\ \boldsymbol{O_{1\times3}} & \boldsymbol{1} \end{bmatrix} \in \boldsymbol{R^{4\times4}}$$

左上角的$\boldsymbol{R_{3\times3}}$是旋转矩阵，右上角的$\boldsymbol{t_{3\times1}}$是位移矢量（求解 PnP 得到的 tvec 就是这个），左下角的$\boldsymbol{O_{1\times3}}$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。：

$$\begin{bmatrix} \boldsymbol{y_1} \\ \boldsymbol{y_2} \\ \boldsymbol{y_3} \\ \boldsymbol{1} \end{bmatrix} = \boldsymbol{T} \cdot \begin{bmatrix} \boldsymbol{x_1} \\ \boldsymbol{x_2} \\ \boldsymbol{x_3} \\ \boldsymbol{1} \end{bmatrix}$$

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（$[\boldsymbol{x} \quad \boldsymbol{y} \quad \boldsymbol{z} \quad \boldsymbol{1}]$）变为纵向列表（$\begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{z} \\ \boldsymbol{1} \end{bmatrix}$），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

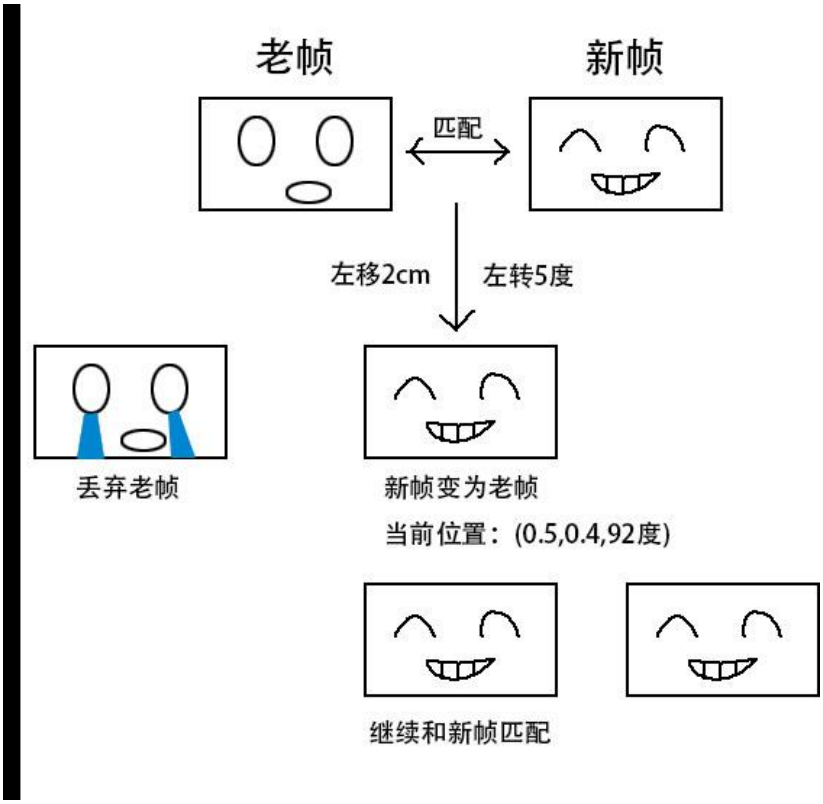
```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

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([0,0,0,0,0], 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);

}
```