Accurate, Dense, and Robust Multi-View Stereopsis论文分析与代码实现（四）

稀疏重建代码优化

本文对上一个文档编写的程序进行若干优化：

1. 优化类结构，以符合面向对象的思想。
2. 优化程序结构，将计算特征点、计算匹配和计算轨迹拆分为3个独立的过程。

第1项优化的目的纯粹是本人爱好，看着结构不好的代码，整个人都不好啦。第2项优化是一个常规操作。因为计算特征点、计算匹配和计算轨迹每一项所耗费的时间都很长，模块独立之后，可以针对某一项的算法进行优化，而不会影响到其他部分。采用这种方法，计算特征点和计算匹配的结果可以保存为文件，计算轨迹时就直接读取这些文件，这样在优化轨迹计算算法时就不用每次都去计算特征点和匹配啦。

1.程序的整体结构

优化后的程序用命令行参数将计算特征点、计算匹配和计算轨迹拆分为3个独立的过程（也可以采用独立的程序，但鄙人认为过于复杂）。`sparse.cpp`是程序执行的入口代码如下：

```
int main(int argc,char *argv[])
{
    if(argc<2)
    {
        cout<<"使用说明:"<<endl;
        cout<<"计算特征点: sparse sift [图像目录] [参数文件] [特征点文件
]"<<endl;
        cout<<"计算匹配: sparse match [特征点文件] [匹配文件]"<<endl;
        cout<<"计算轨迹: sparse track [图像目录] [参数文件] [特征点文件
] [匹配文件] [轨迹文件]"<<endl;
        cout<<"计算全部: sparse all [图像目录] [参数文件] [特征点文件]
[匹配文件] [轨迹文件]"<<endl;
        exit(0);
    }
    string command=argv[1];

    if(command=="sift")
    {
        //特征点计算
        string imageDir=argv[2];
        string parFileName=argv[3];
        string keypointsFileName=argv[4];
        ImageSet imageset(imageDir,parFileName);
        KeyPoints keypoints(imageset.images);
        keypoints.saveTo(keypointsFileName);
        cout<<"保存特征点到文件"<<keypointsFileName<<endl;

    }else if(command=="match")
    {
        //特征点匹配
        string keypointsFileName=argv[2];
        string matchesFileName=argv[3];
        KeyPoints keypoints(keypointsFileName);
        Matches matches(keypoints);
        matches.saveTo(matchesFileName);
        cout<<"保存匹配到文件"<<matchesFileName<<endl;

    }else if(command=="track")
    {
        //轨迹计算
        string imageDir=argv[2];
        string parFileName=argv[3];
        string keypointsFileName=argv[4];
```

```
            string matchesFileName=argv[5];
            string tracksFileName=argv[6];
            ImageSet imageset(imageDir,parFileName);
            KeyPoints keypoints(keypointsFileName);
            Matches matches(matchesFileName);
            TrackList trackList(keypoints, matches);
            trackList.triangulate(imageset.kpmats);
            trackList.saveTo(tracksFileName);
            cout<<trackList<<endl;
            cout<<"保存轨迹到文件"<<tracksFileName<<endl;
        }else if(command=="all")
        {
            //特征点计算、特征点匹配、轨迹计算
            string imageDir=argv[2];
            string parFileName=argv[3];
            string keypointsFileName=argv[4];
            string matchesFileName=argv[5];
            string tracksFileName=argv[6];
            ImageSet imageset(imageDir,parFileName);
            KeyPoints keypoints(imageset.images);
            keypoints.saveTo(keypointsFileName);
            cout<<"保存特征点到文件"<<keypointsFileName<<endl;
            Matches matches(keypoints);
            matches.saveTo(matchesFileName);
            cout<<"保存匹配到文件"<<matchesFileName<<endl;
            TrackList trackList(keypoints, matches);
            trackList.triangulate(imageset.kpmats);
            trackList.saveTo(tracksFileName);
            cout<<"保存轨迹到文件"<<tracksFileName<<endl;

        }
        return 0;
    }
```

代码不用注释也应该能够看懂吧。程序执行时如果不输入任何命令行参数，则打印使用说明：

```
LiuJi-MacBook-Pro:code liuji$ ./build/sparse
使用说明：
计算特征点：sparse sift [图像目录] [参数文件] [特征点文件]
计算匹配：sparse match [特征点文件] [匹配文件]
计算轨迹：sparse track [图像目录] [参数文件] [特征点文件] [匹配文件] [轨迹文件]
计算全部：sparse all [图像目录] [参数文件] [特征点文件] [匹配文件] [轨迹文件]
```

以技术特征点为例，输入`sparse sift ../images/templeSparseRing/templeSR_par.txt keypoints.txt`（注意`/`是必须的）意味着对`../images/templeSparseRing`目录下的`templeSR_par.txt`文件进行解析，并读取该目录下的图像文件，提取sift特征，并将特征点保存在`keypoints.txt`文件中。

接下来输入`sparse match keypoints.txt matches.txt`就执行特征匹配过程，并将结果保存在`matches.txt`文件中。

最后输入`sparse track ../images/templeSparseRing/ templeSR_par.txt keypoints.txt matches.txt tracks.txt`执行计算轨迹的过程，结果保存在`tracks.txt`。

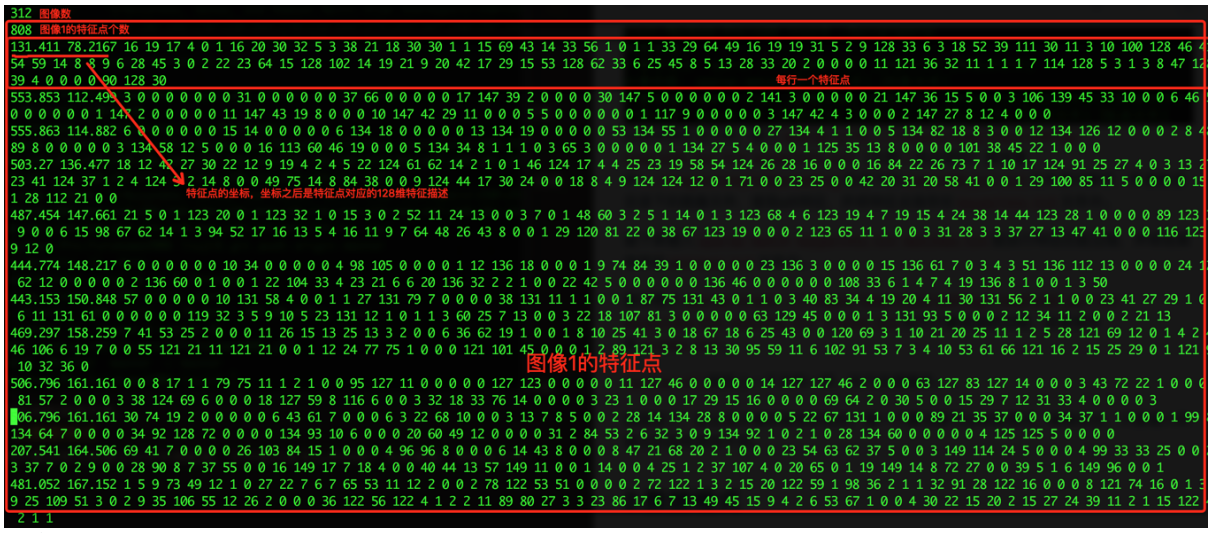
`sparse all`就和上一个文档一样，执行全部操作。

前面说了，这么划分模块是有好处的，每一个算法都可以单独调试，单独优化，只要输入和输出的结构保持不变。

2.数据格式

不同计算过程之间通过文件交换数据，文件格式是怎么样的呢?

2.1特征点文件



文件格式是:
[图像数]
[图像1的特征点数目]
[图像1的特征点]
[图像2的特征点数目]
[图像2的特征点]
[图像3的特征点数目]
[图像3的特征点]
……
其中`特征点`的格式如下:

[特征点的x坐标][特征点的y坐标][特征描述128维向量]

2.2 匹配文件

文件格式是：

[图像数量]

[图像1和图像2的匹配]

[图像1和图像3的匹配]

[图像1和图像4的匹配]

……

[图像2和图像3的匹配]

[图像2和图像4的匹配]

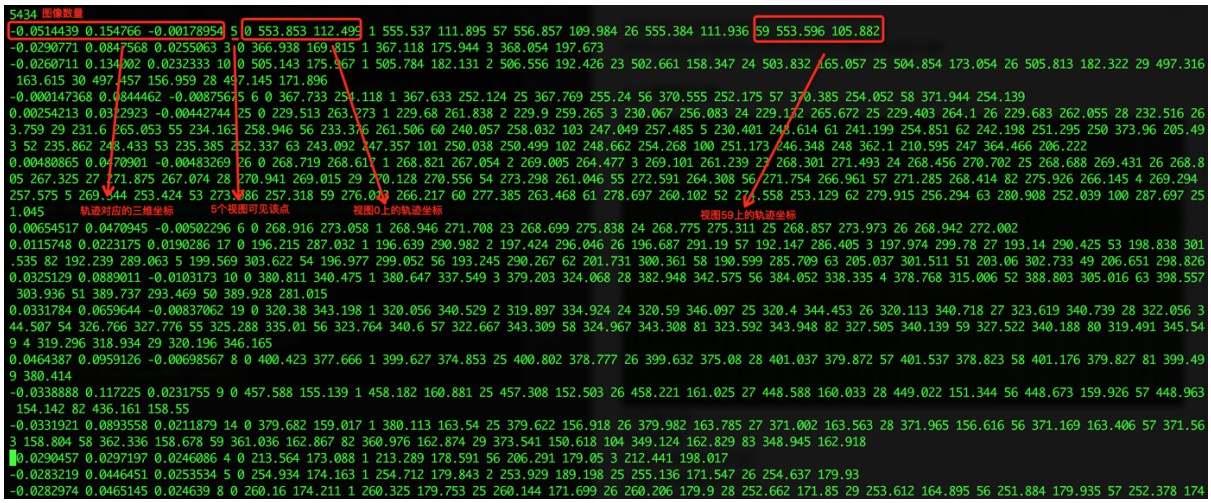
[图像2和图像5的匹配]

……

其中 图像i和图像j的匹配 格式如下：

[匹配点数][图像i的特征点序号][图像j的特征点序号]……[图像i的特征点序号][图像j的特征点序号]……

2.3 轨迹文件



文件格式是：

[图像数量]

[轨迹1]

[轨迹2]

[轨迹3]

……

其中`轨迹i`的格式如下：

[x坐标][y坐标][z坐标][可见轨迹的视图数量][视图序号][视图上的x坐标][视图上的y坐标]……[视图序号][视图上的x坐标][视图上的y坐标]……

3.类结构

程序在上一个文档的基础上增加了`ImageSet`、`KeyPoints`和`Matches`三个类。其定义分别为：

```
class ImageSet{
public:
    ImageSet(const string&imageDir,const string&parFileName);
    vector<Mat> images;
    vector<string> imageNames;
    vector<Mat> kmats;
    vector<Mat> pmats;
    vector<Mat> kpmats;

};

class KeyPoints{
private:
    vector<vector<KeyPoint> > keypointsVec;
    vector<Mat> descriptorsVec;
public:
```

```
    KeyPoints(const vector<Mat>&images);
    KeyPoints(const string&fileName);
    int getFrameNum() const;
    int getKeyPointNum(int i) const;
    KeyPoint getKeyPoint(int i,int j) const;
    const vector<KeyPoint>& getKeyPoints(int i) const;
    const Mat& getDescriptors(int i) const;
    void saveTo(string fileName);
};

class Matches{
private:
    void pairwiseMatch(const vector<KeyPoint>&keypoints1,const ve
ctor<KeyPoint>&keypoints2,const Mat&descriptors1,const
Mat&descriptors2,vector<DMatch>& matches);
    vector<vector<vector<DMatch> > > matchesTable;
public:
    Matches(const KeyPoints&keyPoints);
    Matches(const string&fileName);
    const vector<DMatch>& getMatches(int i,int j) const;
    void saveTo(const string&fileName);
};
```

`KeyPoint`和`Matches`都有两个构造函数，其中一个用数据构造，另一个则直接读取文件。两者都有一个`saveTo`方法用于保存结果。`TrackList`也增加了对应的方法：

```
class TrackList{
private:
    vector<Track> tracks;
public:
    TrackList(const KeyPoints&keypoints,const Matches&matches);
    TrackList(const string&fileName);
    //三角化
    void triangulate(const vector<Mat>&pmats);
    void getColor(const vector<Mat>&images);
    //保存到ply文件
    void save2ply(const string&fileName);
    void saveTo(const string&fileName);
    friend ostream&operator<<(ostream&os,const
TrackList&trackList);
};
```

下面贴一下`KeyPoint`的构造函数以及`saveTo`方法，其他类类似，各位可以直接阅读代码。

```
```

```
KeyPoints::KeyPoints(const vector<Mat>&images):keypointsVec(image
s.size(),vector<KeyPoint>()),descriptorsVec(images.size(),Mat())
{
    SIFT sift;
    for (int i=0; i<images.size(); i++) {
        cout<<"计算视图"<<i<<"的特征点，";
        sift(images[i],Mat(),keypointsVec[i],descriptorsVec[i]);
        cout<<"特征点共有"<<keypointsVec[i].size()<<"个"<<endl;
    }
}
KeyPoints::KeyPoints(const string&fileName)
{
    ifstream ifs(fileName.c_str());
    int nImages;
    ifs>>nImages;
    keypointsVec.resize(nImages,vector<KeyPoint>());
    descriptorsVec.resize(nImages);
    for(int i=0;i<nImages;i++)
    {
        int nPoints;
        ifs>>nPoints;
        descriptorsVec[i]=Mat(nPoints, 128, CV_32F);
        for (int j=0; j<nPoints; j++) {
            KeyPoint point;
            Mat descriptor=descriptorsVec[i].row(j);
            ifs>>point.pt.x>>point.pt.y;
            for (int k=0; k<descriptor.cols; k++) {
                ifs>>descriptor.at<float>(0,k);
            }
            keypointsVec[i].push_back(point);
        }
    }
}
void KeyPoints::saveTo(string fileName)
{
    ofstream ofs(fileName.c_str());
    ofs<<keypointsVec.size()<<endl;
    for (int i=0; i<keypointsVec.size(); i++) {
        ofs<<keypointsVec[i].size()<<endl;
        for (int j=0; j<keypointsVec[i].size(); j++) {
            ofs<<keypointsVec[i][j].pt.x<<" "<<keypointsVec[i][j]
.pt.y<<" ";
            Mat descriptor=descriptorsVec[i].row(j);
            for (int k=0; k<descriptor.cols; k++) {
                if(k==descriptor.cols-1)
                    ofs<<descriptor.at<float>(0,k)<<endl;
                else
                    ofs<<descriptor.at<float>(0,k)<<" ";
            }
        }
    }
    ofs.close();
}
```

4.程序说明

在程序目录下执行`make`即可完成编译，编译后程序在`build`目录下，之后在目录下执行`build/sparse`运行程序。