OpenCV常用函数和应用实例

本章主要介绍 OpenCV 的常用函数、变量以及基本数据结构。在此基础上，给出部分应用示例，详细讲解反向、滤波、双目视觉测深度算法，为灵活使用 OpenCV 打下基础。

3.1 OpenCV 常用函数

本节主要针对 OpenCV 中最常用的，也是最核心的 Mat 类、imread 函数、imshow 函数以及 imwrite 函数进行介绍。

3.1.1 Mat 类

说到 Mat 类就必须要介绍一下 OpenCV 2 系列。在 OpenCV 1 系列的函数库中，函数都是基于 C 接口构建的，使用名为 IplImage 的 C 语言结构体在内存中存储图像，用法类似于 C 语言中的指针，在使用函数之前先为变量开辟内存，使用之后释放内存，在算法复杂度较高的情况下，指针的使用会引起内存混乱。而在 OpenCV 2 系列中，C++的出现带来了全新的 Mat 类，使 OpenCV 能执行自动的内存管理，也就是说，在使用 OpenCV 时不必再进行手动开辟内存、管理内存等烦琐的工作，大大减少了开发者的工作量。自动开辟指的是程序自动根据需要使用的内存空间，合理开辟内存，不会造成资源的浪费，当然如果必要，也可以手动开辟内存空间[1]。

1. Mat 类的定义

官方手册上 Mat 的定义是：Mat 类用于表示一个多维度的单通道或者多通道的稠密数组，能够用来保存实数或复数的向量、矩阵、灰度或彩色图像、立体元素、点云、张量以及直方图等[2]。

在使用的过程中只需要记住两点：

(1) Mat 类用来保存多维度的矩阵；

(2) Mat 类不需要手动开辟内存，也不需要手动释放内存。

2. Mat 的数据结构

Mat 的数据结构主要包括两个部分：Header 和 Pointer。其中 Header 主要包含矩阵

的大小、存储方式、存储地址等信息，而 Pointer 是存储指向像素值的指针。

Mat 类的常见属性如下：

(1) data。

data 是 uchar 型的指针，也就是上述指向矩阵内数据的指针。

(2) dims。

dims 指的是矩阵的维度，如：一维数组 dims=1；二维矩阵 dims=2 等。

(3) rows、cols。

rows 指矩阵的行数，cols 指矩阵的列数。需要注意的是，如果是多通道的彩色图像，rows 和 cols 分别表示的是图像中纵向的像素点数量和横向像素点数量，这与通道数没有关系。

(4) channels。

channels 指矩阵的通道数，如果读入的是图像，也指图像的通道数。通常单通道的图像为灰度图像，彩色图像有三通道的 RGB(Red、Green、Blue)图像和四通道的 RGBA(Red、Green、Blue、Alpha)图像。

(5) type。

type 表示矩阵中元素的类型以及矩阵的通道数，属于预定义的常量，其命名的规则为[3]：

CV_＋位数＋数据类型＋通道数

如：CV_8UC1 表示 8 位、unsigned integer 无符号型、单通道；

CV_16SC2 表示 16 位、singed integer 有符号整数、双通道；

CV_64FC4 表示 64 位、float 浮点型、四通道。

其具体的型号如表 3-1 所示。

表 3-1 type 常量

CV_8UC1	CV_8UC2	CV_8UC3	CV_8UC4
CV_8SC1	CV_8SC2	CV_8SC3	CV_8SC4
CV_16UC1	CV_16UC2	CV_16UC3	CV_16UC4
CV_16SC1	CV_16SC2	CV_16SC3	CV_16SC4
CV_32SC1	CV_32SC2	CV_32SC3	CV_32SC4
CV_32FC1	CV_32FC2	CV_32FC3	CV_32FC4
CV_64FC1	CV_64FC2	CV_64FC3	CV_64FC4

(6) depth。

depth 指矩阵中元素一个通道内的数据类型，这个和上面的 type 型类似，相比于 type 更加简单。depth 常用常量如表 3-2 所示。

表 3-2 depth 常量

CV_8U	CV_8S	CV_8F
CV_16U	CV_16S	CV_8F
CV_32U	CV_32S	CV_32F
CV_64U	CV_64S	CV_64F

（7）elemSize。

elemSize 是指矩阵中一个元素占用的字节数，其值为：

elemSize＝通道数×位数÷8

如：CV_64FC4；elemSize＝4×64÷8＝32bytes。

（8）elemSize1。

elemSize1 指矩阵元素一个通道占用的字节数，相比于 elemSize 仅少了通道数，如：CV_64FC4；elemSize1＝64÷8＝8bytes。

下面的程序详细示范了 Mat 属性的使用方法及效果：

```
#include<opencv2/opencv.hpp>
#include<iostream>
usingnamespace cv;
usingnamespace std;

void main()
{
    Mat img(3, 4, CV_16UC4, Scalar_<uchar>(1, 2, 3, 4));
    cout << img << endl;
    cout <<"dims:"<< img.dims << endl;
    cout <<"rows:"<< img.rows << endl;
    cout <<"cols:"<< img.cols << endl;
    cout <<"channels:"<< img.channels() << endl;
    cout <<"type:"<< img.type() << endl;
    cout <<"depth:"<< img.depth() << endl;
    cout <<"elemSize:"<< img.elemSize() << endl;
    cout <<"elemSize1:"<< img.elemSize1() << endl;
    getchar();
}
```

该程序定义了一个 3 行 4 列 4 通道 16 位 unchar 型矩阵，存放的数值依次为 1、2、3、4。运行结果如图 3-1 所示。

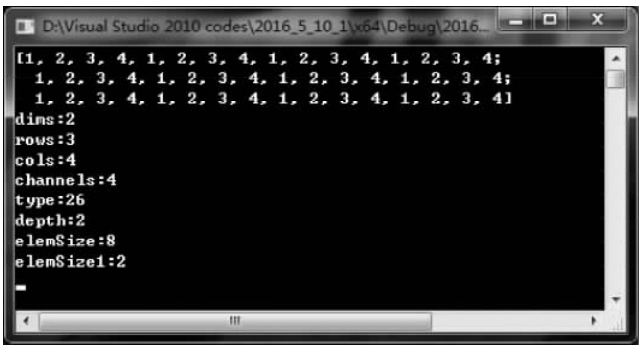

图 3-1　Mat 属性测试结果图

除以上介绍的几种 Mat 属性外，下面还要特别指出三种易混淆的 Mat 属性：step、step1 和 size。不过在此之前需要知道 OpenCV 内对于维度的定义。

OpenCV 对于维度的定义是：矩阵的深度为第一维，矩阵的高度为第二维，矩阵的宽度

为第三维。用一个例子来进行说明，如图 3-2 所示，该矩阵高度为 4，宽度为 5，深度为 3。矩阵的面就是其第一维，也就是矩阵深度，例子中该值为 3；矩阵中每个面上的每一行为第二维，也就是矩阵高度，例子中该值为 4；矩阵的每一行中的每一个点是第三维，也就是矩阵的宽度，例子中该值为 5，如图 3-3 所示。

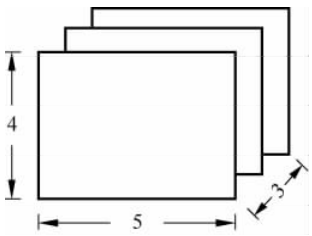

图 3-2　三维矩阵示意图

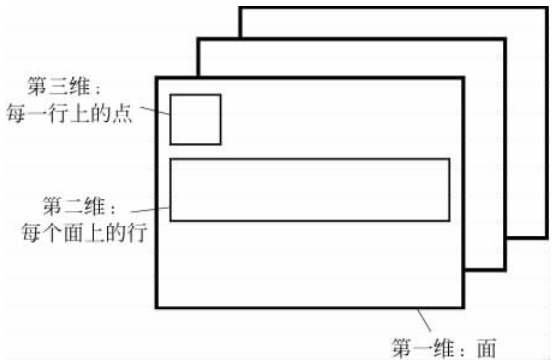

图 3-3　OpenCV 对三维矩阵的维度划分

(9) size。

size 表示每一维元素的个数，通常情况下会使用 size[0]、size[1]和 size[2]来分别表示矩阵的第一维、第二维和第三维。使用如图 3-2 所示的矩阵，对应的取值为 size[0]＝3，size[1]＝4，size[2]＝5。

(10) step

step 表示的是每一维的元素的大小，单位是字节。step 与 size 用法类似，也是用 step[0]、step[1]和 step[2]。step[0]表示第一维的元素的大小，即每个面的元素的总字节数；step[1]表示第二维的元素的大小，即面上的每一行的元素的总字节数；step[2]表示第三维元素的大小，即每一行中的一个元素的总字节数。

那么对于之前的矩阵，其 step 值为：

```
step[2] = 3 × 8 ÷ 8 = 3;
step[1] = step[2] × 5 = 3 × 8 ÷ 8 × 5 = 15;
step[0] = step[1] × 4 = 3 × 8 ÷ 8 × 5 × 4 = 60;
```

需要补充的是，如果矩阵是二维矩阵，那么 step[0]仍然表示一维元素的大小，但是这个时候，就不是面，而是单个面中的线，也就是一行元素所占的字节数量。同理，step[1]表示二维元素的大小，即矩阵中一个元素的大小，而 step[2]则没有意义。

(11) step1。

step1 表示每一维元素的通道总数，同样的 step1(0)是一维元素的通道总数，即每一个面上所有元素的通道数；step1(1)是二维元素的通道总数，即每一个面上的每一行的元素的通道数；step1(2)是三维元素的通道总数，即每一行上，每一个元素的通道数。还是用前面提到的三维矩阵来举例，那么有[4]：

```
step1(2) = 3;                    //三通道
step1(1) = 3 × 5 = 15;
step1(0) = 3 × 5 × 4 = 60;
```

通过以下程序来解释一下 size、step 和 step1 的区别，程序如下：

```
#include<opencv2/opencv.hpp>
#include<iostream>
usingnamespace cv;
usingnamespace std;

void main()
{
    int matSize[]={3,4,5};   //定义第一维为 3;第二维为 4;第三维为 5
    Mat img(3,matSize,CV_8UC3, Scalar::all(0));

    cout<<"------ size 的大小 ------"<<endl;
    cout<<"size[2]="<<img.size[2]<<endl;
    cout<<"size[1]="<<img.size[1]<<endl;
    cout<<"size[0]="<<img.size[0]<<endl;

    cout<<"\n------ step 的大小 ------"<<endl;
    cout<<"step[2]="<<img.step[2]<<endl;
    cout<<"step[1]="<<img.step[1]<<endl;
    cout<<"step[0]="<<img.step[0]<<endl;

    cout<<"\n------ step1 的大小 ------"<<endl;
    cout<<"step1(2)="<<img.step1(2)<<endl;
    cout<<"step1(1)="<<img.step1(1)<<endl;
    cout<<"step1(0)="<<img.step1(0)<<endl;

    getchar();
}
```

程序运行结果如图 3-4 所示。

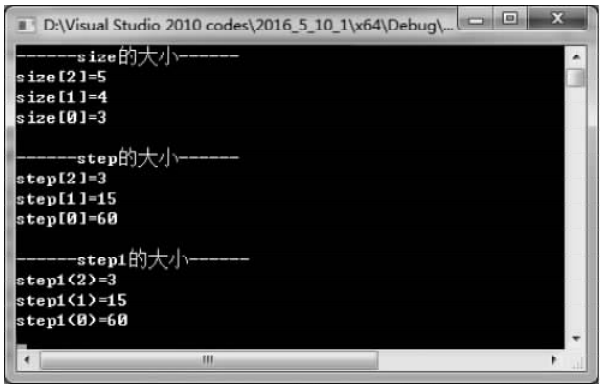

图 3-4　Mat 中 size、step 和 step1 的区别

3.1.2　imread 函数

imread 函数是 OpenCV 2 系列新推出的读入图像函数，其使用方式和 MATLAB 中的 imread 函数极为相似，使用方便，减少了编程时的难度。

imread 函数的定义为：

```
cv::Mat imread(const string & filename, int flag = 1);
```

(1) const string & filename。

这个参数是 const string 类型的 filename，需要在这里放入默认路径下的图像的名称，并且支持绝大多数的图像格式，如：JPEG 型中的.jpeg、.jpg 文件；Windows 位图型中的.bmp 文件；以及 PNG 格式下的.png 文件等。

(2) int flag=1。

这个参数是 int 类型的 flags，它作为一个图像读入的标识，指定了图像读入的颜色类型，也可以说成是指定读入通道的数量。若不指定 flag 参数，则其默认值为 1，代表读入三通道 RGB 彩色图像[5]。

其参数值有如下几种：

- flat=0——图像以单通道灰度图方式读入。
- flat=1——图像以三通道 RGB 方式读入。
- flat=2——图像深度若为 16 位或 32 位，则以相应的深度进行读入，如果不是深度为 16 位或 32 位，则会以 8 位图像读入。
- flat=其他——默认与 flat=1 对应的读入方式一致。在从前的版本中，flat=-1 表示读入 8 位有颜色图或灰度图，但是在新版本中已经被废置了，所以这里的负数部分无须太过深究。

imread()函数常用读取的例子如下：

```
Mat img = imread("Peashooter.jpg");                                  //RGB
Mat img = imread("Peashooter.jpg",0);                                //灰度图
Mat img = imread("E:\VisualStudio2012_code\Peashooter.jpg");         //RGB
Mat img = imread("E:\VisualStudio2012_code\Peashooter.jpg",2);       //RGBA
```

3.1.3 imshow 函数

imshow()函数可以在窗口中显示图像，与 MATLAB 中 imshow()函数十分类似，其原型如下：

```
void cv::imshow( const std::string & winname, cv:: InputArray mat)
```

头文件：highgui.hpp，命名空间：cv。

(1) const std::string & winname

winname 是 const string& 类型的参数，为显示图像的窗口名称。

(2) cv::InputArray mat

mat 是 InputArray 参数，这个位置放入 Mat 类的变量，即图像存储的变量。

在使用 imshow 函数时需要注意图像显示过程中可能会进行缩放，其具体缩放的程度取决于原始图像的深度，具体如下：

① 图像是 8 位无符号型，则输出图像不会发生任何变化；

② 图像是 16 位无符号型或 32 位整型，会将像素值除以 255 显示出来。

③ 图像是 32 位浮点型，像素值需要乘以 255 显示出来。

结合 Mat 类、imread()和 imshow()来看一下读入一张三通道图像后 Mat 类内部变量值的情况，对应的程序如下：

```
#include <iostream>
#include <opencv2/opencv.hpp>
using namespace std;
using namespace cv;

int main(int argc, char * argv[])
{
    Mat img = imread("Peashooter.jpg");
    imshow("image", img);
     cout << "图像参数如下:" << endl;
     cout << "dims:" << img.dims << endl;
     cout << "rows:" << img.rows << endl;
     cout << "cols:" << img.cols << endl;
     cout << "channels:" << img.channels() << endl;
     cout << "type:" << img.type() << endl;
     cout << "depth:" << img.depth() << endl;
     cout << "elemSize:" << img.elemSize() << endl;
     cout << "elemSize1:" << img.elemSize1() << endl;

    cout << "\n------ size 的大小 ------ " << endl;
    cout << "size[2] = " << img.size[2] << endl;
    cout << "size[1] = " << img.size[1] << endl;
    cout << "size[0] = " << img.size[0] << endl;

    cout << "\n------ step 的大小 ------ " << endl;
    cout << "step[2] = " << img.step[2] << endl;
    cout << "step[1] = " << img.step[1] << endl;
    cout << "step[0] = " << img.step[0] << endl;

    cout << "\n------ step1 的大小 ------ " << endl;
    cout << "step(2) = " << img.step1(2) << endl;
    cout << "step(1) = " << img.step1(1) << endl;
    cout << "step(0) = " << img.step1(0) << endl;

     waitKey(0);                //按任意键退出
     return 0;
}
```

程序运行结果如图 3-5 和图 3-6 所示。

以上程序需特别注意一点：图像是二维的，因此 size[2]、step[2]和 step1[2]会出现乱码，这三项没有参考价值。

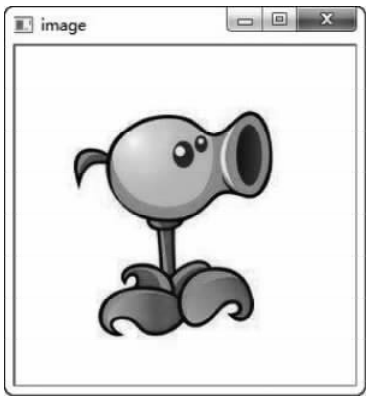

图 3-5　imshow 函数显示图像

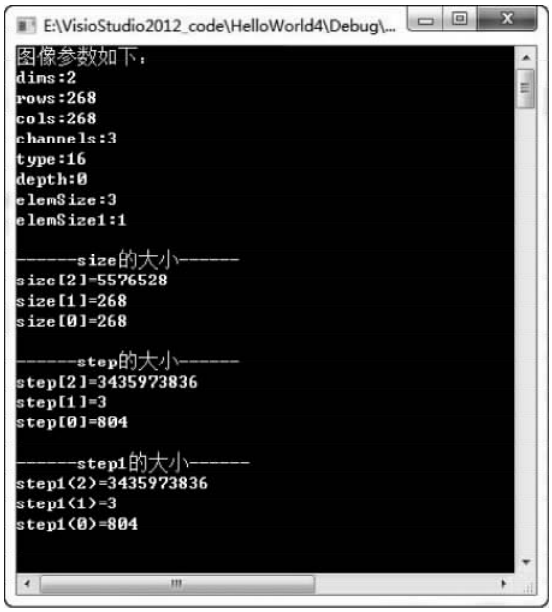

图 3-6　使用 imshow 得到图像参数

3.1.4　imwrite 函数

imwrite()函数通常用于存储处理后的图像，函数声明如下：

```
bool cv::imwrite(const std::string &filename, cv::InputArray img,const std::vector < int,
std::allocator < int >> &params = std::vector < int >())
```

头文件：highgui.hpp，命名空间：cv。

(1) const std::string &filename。

filename 是 const string& 类型的，表示文件名及存储格式。

(2) cv::InputArray img。

img 是准备被保存起来图像的 Mat 类参数。

(3) const std::vector < int, std::allocator < int >> &params=std::vector < int >()。

params 是 const vector < int >类型的参数，表示特定格式残存的参数编码。其默认值为 vector < int >()。

3.2　反向算法

1. 反向算法的原理

反向算法是图像处理中最简单的算法之一，适合用于环境搭建测试，反向算法即将图像中所有的像素点的色彩度反色，算法原理如下面公式所示：

$$f(x) = 255 - g(x) \tag{3-1}$$

因书中图是黑白图，很难看到颜色变化，如图 3-7 所示为预期的效果图。

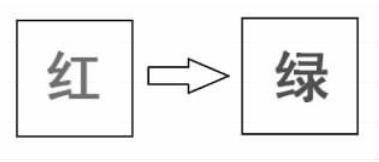

图 3-7　反向算法示意图

2. 反向算法实现

```
//------------------------------ 头文件部分 ------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <iostream>

using namespace cv;
using namespace std;


//------------------------------ 主函数部分 ------------------------------
int main(int argc, char * * argv)
{
    IplImage * srcImg = cvLoadImage("lena512.png",1);
//读入图像 lena512.png

//-------------------------- 计时函数:开始计时段 ----------------------------
    double timeSpent = (double)getTickCount();


//------------------------------ 提取图像信息 ------------------------------
    Int iii,jjj;
    int nHeight,nWidth,nChannels,nWidthStep;
    nHeight = srcImg->height;                    //使用 OpenCV 提取图像高度
    nWidth = srcImg->width;                      //使用 OpenCV 提取图像宽度
    nChannels = srcImg->nChannels;               //使用 OpenCV 提取图像通道数
    nWidthStep = srcImg->widthStep;              //使用 OpenCV 截取一行像素点宽度

//------------------------------ 算法实现部分 ------------------------------
uchar * data = (uchar * )srcImg->imageData;
        //三通道彩色图像,使用两套循环遍历所有像素点实现反向
    for(jjj = 0;jjj < nHeight;jjj++)
    {
        for(iii = 0;iii < nWidth * nChannels;iii++)
        {
    data[jjj * nWidthStep + iii] = 255 - srcImg->imageData[jjj * nWidthStep + iii];
        }
    }

//---------------------------- 计时函数:终止阶段 ----------------------------
```

```
    timeSpent = ((double)getTickCount() - timeSpent)/getTickFrequency();
    cout <<"Time spent in milliseconds: "<< timeSpent * 1000 << endl;
//在屏幕上输出使用的时间

    cvShowImage("反向图像",srcImg);                    //显示图像
    cout <<" Channels "<< nChannels << endl;           //显示图像通道数
    waitKey(0);                                        //循环等待退出
    return 0;
}
```

3. 程序运行结果

程序运行结果如图 3-8 和图 3-9 所示，左图为原始图像，右侧为处理后的图像。

图 3-8　反向算法的结果 1

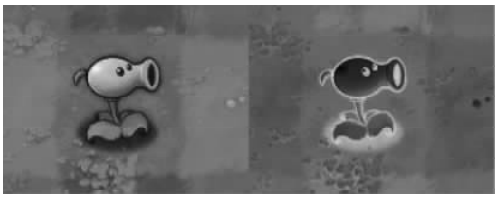

图 3-9　反向算法的结果 2

4. 程序简析

反向算法相对浅显，相信读者根据程序中的备注，可以看懂该算法。因此比较容易理解以反向算法为例介绍 OpenCV 编程时会涉及的常识以及需要注意的细节。

1）包含头文件部分

编写 OpenCV 程序时，经常会使用名为 opencv.hpp 的头文件，该头文件中有如下定义[6]：

```
#inlude <opencv2/opencv.hpp>
#ifndef __OPENCV_ALL_HPP__
#define __OPENCV_ALL_HPP__

#include "opencv2/core/core_c.h"
#include "opencv2/core/core.hpp"
#include "opencv2/flann/miniflann.hpp"
#include "opencv2/imgproc/imgproc_c.h"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/photo/photo.hpp"
#include "opencv2/video/video.hpp"
#include "opencv2/features2d/features2d.hpp"
```

```
#include "opencv2/objdetect/objdetect.hpp"
#include "opencv2/calib3d/calib3d.hpp"
#include "opencv2/ml/ml.hpp"
#include "opencv2/highgui/highgui_c.h"
#include "opencv2/contrib/contrib.hpp"
#endif
```

通过以上定义可以知道，opencv.hpp 头文件包含了绝大多数常见的 OpenCV 头文件。因此，编程时仅写 #inlude < opencv2/opencv.hpp >，在实现功能的同时，也可以实现程序的精简化。

2）头文件命名部分

```
using namespace cv;
using namespace std;
```

这两行程序说的是文件的命名空间在 cv 之内，通常对文件进行命名有两种方法：

第一种方法如前面程序所写，程序最开头，包含头文件的部分直接添加上 using namespace cv，这样后续所有的类均在 cv 之内。

第二种方法是在程序开头部分不加 cv 声明，但后面每一个 cv 类或函数，都要以“cv::”这种方式命名，这种命名方式虽然正确但是过于烦琐，且易出错，因此不推荐。

- 主函数部分：main()函数的写法。

例程中的 main()函数与 C 语言和 C++语言的 main()函数命名稍有不同，这里写法为：

```
int main(int argc, char * * argv)
```

其中 arg 指的是参数，argc 为整数，用来统计运行程序时送给 main()函数的命令行参数的个数。argv 同上会加上 * 和[ ]，成为 * argv[ ]，表示字符串数组，用来存放指向字符串参数的指针数组，每一个元素指向一个参数。

main()函数在使用过程中，不论写不写上 argc 和 argv 都是正确的，因此 main()函数通常有如下两种写法：

① int main(int argc, char * * argv) { }

② void main() { }

- 图像的读入：IplImage * srcImg = cvLoadImage("lena512.png",1)。

示例采用 opencv 1 系列中图像读入的方法，IplImage 函数使用了 C 语言作为接口进行读入。在 OpenCV 2.4.X 系列和之后的版本中，基本已经舍弃了这种读入方式，取而代之的是用 Mat 类进行读入。但本算法需要使用指针，所以没有把图像变成 Mat 类。

这两种读入方式用法如下：

```
IplImage * srcImg = cvLoadImage("lena512.png",1);
Mat srcImg = imread("lena512.png",1);
```

前者为 C 接口的指针型，后者为 C++接口的 Mat 类。

① 读入方式。

以上两种读入方式都支持的图像格式有：

```
Windows 位图: .bmp、.dib
```

```
JPEG 文件：.jpeg、.jpg、.jpe
JPEG2000 文件：.jp2
PNG 文件：.png
便携文件：.pbm、.pgm、.ppm
Sun rasters 光栅文件：.sr、.ras
TIFF 文件：.tiff、.tif
```

② 载入标识 flags。

载入标识指在图像读入过程中对有颜色图像加载的颜色类型，flags 不同对应读入计算机中的图像颜色也会有区别，通常不输入 flags 时，其默认值为 1。与前面介绍的 Mat 类的 flags 标识含义完全相同。

• 计时函数：getTickCount()。

计时开始函数：

```
double timeSpent = (double)getTickCount();
```

计时终止函数：

```
timeSpent = ((double)getTickCount() - timeSpent)/getTickFrequency();
```

getTickCount 函数的作用是返回(retrieve)从操作系统启动所经过(elapsed)的毫秒数(ms)，它的返回值是 DWORD。

使用方法如上所述，只要设定好一个变量，按照上面的格式去写就可以测试这两个变量中间的程序运行所消耗的时间[7]。

• 提取图像信息。

```
nHeight = srcImg->height;
nWidth = srcImg->width;
nChannels = srcImg->nChannels;
nWidthStep = srcImg->widthStep;
```

本算法使用 IplImage 读取图像的信息，其中 nHeight、nWidth、nChannels、nWidthStep 分别指存储图像的高度 height、图像的宽度 width、图像的通道数 nChannels 和每一行像素点存储所需要的字节数 nWidthStep，如图 3-10 所示。其中 height、width、nChannels 和 widthStep 是 OpenCV 自带的函数，专门用来进行图像的特征提取。

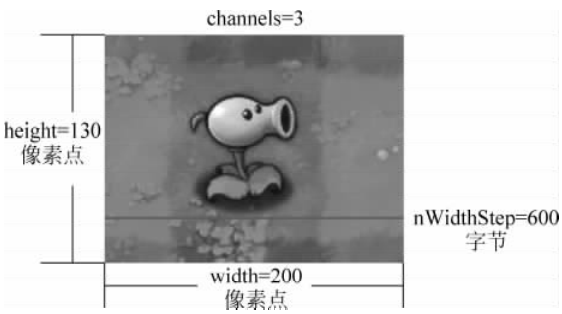

图 3-10　提取图像信息的变量

需要注意的是，nWidthStep 指存储一行图像的像素点所需要的字节数，其自身必须为 4 的倍数，以便实现字节对齐，这种方式会提高运算速度。一般情况下 nWidthStep＝width×nChannels，即每一行所需要的字节数等于每一行像素点的数量乘图像自身的通道数，但是如果图像自身没有令 nWidthStep 为 4 的倍数，那么这个计算方式就不成立，需要使用空字节来补齐。

使用上面的图像进行说明：

上述存储在计算机中的三通道的彩色图，其数据存储效果如图 3-11 所示，以 B、G、R 的形式进行存储。像素点 width＝200，通道数 nChannels＝3，所以刚好可以实现 nWidthStep＝600 是 4 的倍数。

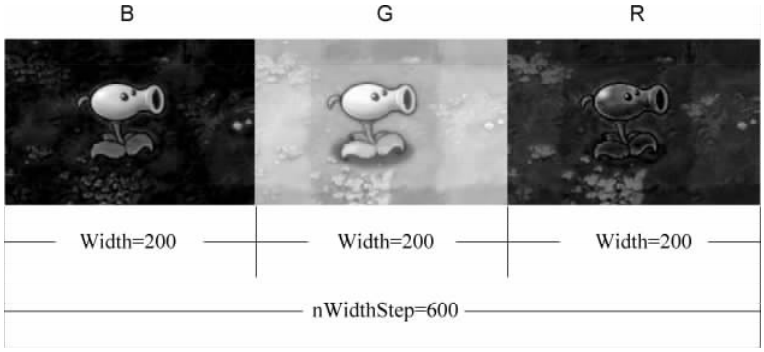

图 3-11　计算机中存储三通道图像

假设一行像素点为 199 个，即 width＝199，nChannels＝3，理论上存储在计算机中的 nWidthStep 应该是 199×3＝597，但是在实际应用的过程中 nWidthStep 的值会是 600，因为 597 不是 4 的倍数，遵照向上对齐的原则使读入的图像后边添加了三个空的字符，这三个字符仅仅是用来实现字符对齐的，没有其他任何作用，如图 3-12 所示。

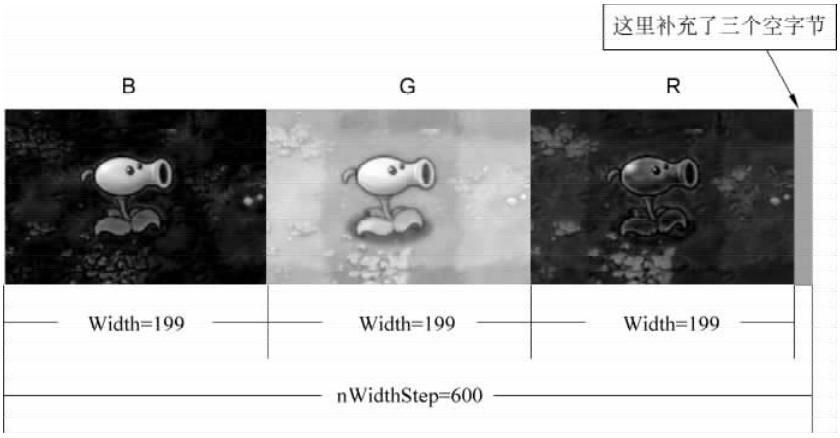

图 3-12　计算机中 nWidthStep 存储补足

这个对齐的模式可能会对一些特殊的图像处理算法造成影响，例如 NLM 算法在外围搜索的过程中会搜索到边缘没有意义的像素点。所以建议在处理图像时尽量选择标准大小的图像，如像素点为 64×64、128×128、256×256……或者图像的一行像素点为 4 的倍数也可以。

• 算法部分：

```
for(jjj = 0;jjj < nHeight;jjj++)
{
    for(iii = 0;iii < nWidth * nChannels;iii++)
    {
data[jjj * nWidthStep + iii] = 255 - srcImg -> imageData[jjj * nWidthStep + iii];
    }
}
```

程序使用了两套循环来实现图像的遍历。图 3-13 为三通道彩色图像存入计算机中显示效果图，OpenCV 和 MATLAB 显示的部分都一样，都是三通道三个矩阵的形式进行显示，图下方的方框示意为图像中像素点的具体数值。

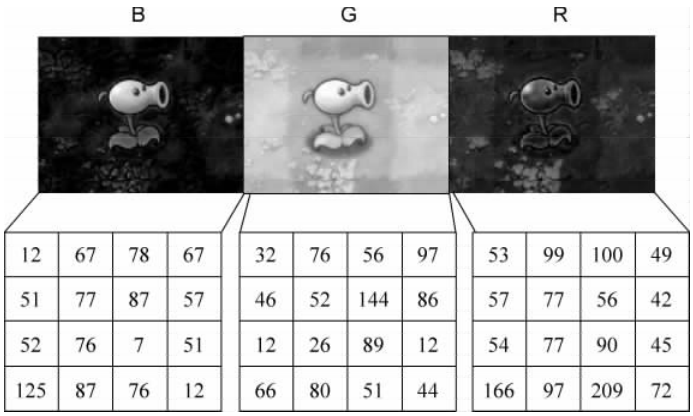

图 3-13　计算机中存储三通道图像示意图

在计算机存储的过程中，OpenCV 和 MATLAB 的存储方式有一些区别，MATLAB 会将单通道灰度图像存成二维平面矩阵，将三通道或者多通道彩色图像存成三维空间矩阵，例如现在想取 Green 层第二行第二列的像素点 46，MATLAB 就可以写成：

```
A = Data[1,1,1]
% A 为变量,Data 为图像
% 第一个 1 是指第二行,第二个 1 指第二列,第三个 1 为第二层 Green 层
```

OpenCV 的存储方式是将所有的 BGR 三色像素点以一个一维数组的方式进行存储的，如图 3-14 所示，第一行存储完成之后直接存储第二行，这也就是为什么 OpenCV 要用到 nWidthStep 函数。如果图像 width 不是 4 的倍数，导致 nWidthStep 需要补充时，补充的空白字节也将以这种方式存在这个一维数组中。

了解 OpenCV 的存储方式后，之前的遍历算法也就清楚了：

jjj(行)×nWidthStep(每一行所需的字节数)+iii(列)=像素点位置

• 图像显示：cvShowImage("反向图像",srcImg)。

图形显示也有两种方式：一种是例子中使用的 cvShowImage()函数，这个函数可以将指针形式存储的图像无损地显示出来。第一个参数是“窗口名称”，用双引号(一定注意是英文输入法下的双引号)将名称包括在中间即可；第二个参数是图像的指针名称。

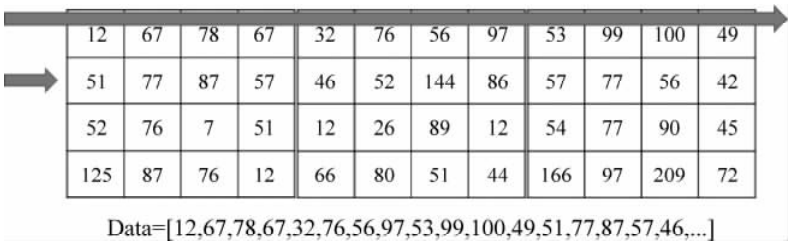

图 3-14　OpenCV 存储像素点的模式

第二种输出函数较为常用：imshow()函数。

imshow()函数可以输出 Mat 类读入的图像，使用起来也更方便，因此这种方式更为常用，使用方式如下：

```
imshow("图像显示",image);
```

第一个参数是在双引号内写出图像显示窗口的名称，后一个参数是用来存储图像的 Mat 类变量。

5. 附加

为了对比 OpenCV 函数和 MATLAB 函数的区别，这里使用 MATLAB 进行一次反向算法编写，程序如下：

```
clear all
f = imread('PeaShooter.png');                    % 图像读入
tic                                              % MATLAB 的计时函数
[M,N,Z] = size(f);                               % 图像高度、宽度、通道数测量
g = ones(M,N,Z);                                 % 建立一个全是 1 的矩阵
for i = 1:M                                      开始进行像素点遍历
for j = 1:N
for k = 1:Z
            g(i,j,k) = 255 - f(i,j,k);           % 实现反向
end
end
end
g = uint8(g);                                    % 将浮点型数据转化为整型
figure;                                          % 画图
subplot(121);                                    % 在图中左侧放上原图像，右侧放上反向后的图像
imshow(f,[]);
subplot(122);
imshow(g,[]);
toc
```

为了方便对比，这次实现仍使用了与之前一样的图像，处理的结果如图 3-15 所示，与使用 OpenCV 的处理结果一模一样，但是处理的时间却远超过 OpenCV。

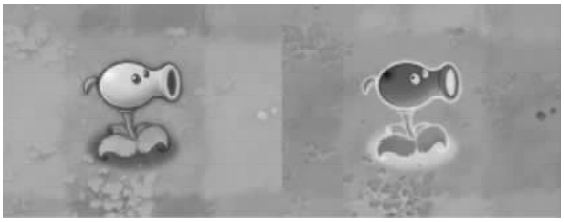

图 3-15　MATLAB 实现反向算法

3.3　图像融合

图像融合是图像处理中比较常用的一种处理手段，将两个或多个图像以一定的方式融合成一个图像。如图 3-16 所示，左侧为静物的左聚焦图像，图中间为静物的右聚焦图像，图右侧为两个图像中清晰部分融合起来的图像。

图 3-16　图像融合示例

本节将使用 OpenCV 自带函数实现简单的图像融合或者叫图像混合，通过这些例程可以加深对 OpenCV 函数的理解。

3.3.1　覆盖型图像融合

这种图像融合是将一个图像以马赛克的形式盖在另外一个图像的某个部位上。

（1）程序如下：

```
#include<opencv2/opencv.hpp>
#include<opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
using namespace cv;
using namespace std;

int main()
{
    Mat srcImage1=imread("1.jpg");              //读取图像 1 作为背景
    Mat srcImage2=imread("2.jpg");              //读取图像 2 作为覆盖面
    Mat imageROI=srcImage1(Rect(100,100,srcImage2.cols,srcImage2.rows));
    //ROI 部分
    Mat mask=imread("2.jpg",0);                 //读入图像
    srcImage2.copyTo(imageROI,mask);            //图像覆盖
    namedWindow("图像融合");                    //给窗口命名
    imshow("图像融合",srcImage1);               //图像输出
    waitKey();
    return 0;
}
```

（2）实验结果。

相信通过程序备注，读者可以理解程序的含义，程序运行的结果如图 3-17～图 3-19 所示，图 3-17 为图像的背景，图 3-18 为背景上添加的图像，图 3-19 为完成示意图。

图 3-17　背景

图 3-18　背景上的图像

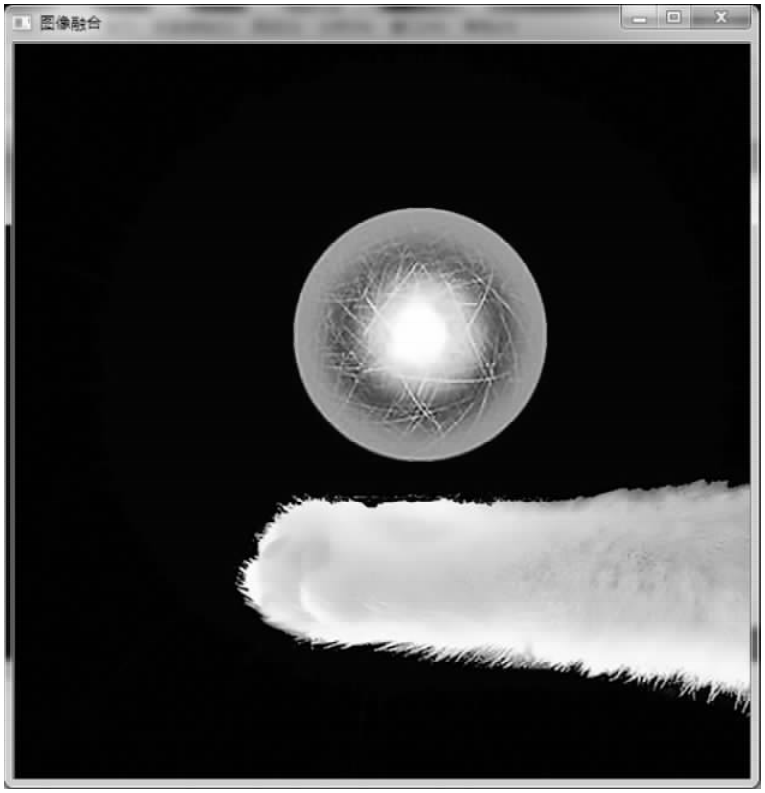

图 3-19　两幅图像融合后的结果

(3) 注意事项和程序简析。

ROI(Region Of Interest)指图像中感兴趣的部分,其实就是找到图像中想要进行图像处理的部分,通常使用 Rect 函数和 Range 函数,其中 Rect 函数更为常用,Rect 函数的使用方法如下:

```
X = A(Rect(B,C,D,E));
```

X是一个Mat类的变量，用来存放图像中的ROI；

A是原图像；

B、C用来确定ROI左上角点的坐标，其中B是原图像A中的第多少列，C是原图像A中的第多少行；

D和E是ROI的高度和宽度，D是从B、C确定的原点位置，向下D个像素点，作为ROI的高度；E是从B、C确定的原点位置，向右E个像素点，作为ROI的宽度。如图3-20所示。

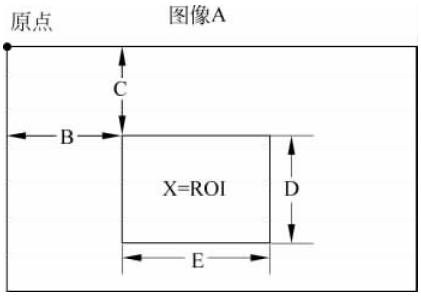

图3-20　ROI示意图

例如，在上述的程序中：

```
Mat imageROI = srcImage1(Rect(100,100,srcImage2.cols,srcImage2.rows));
```

imageROI是原图像的ROI部分；

srcImage1是原图像；

100,100是ROI左上角的原点坐标。

srcImage2.cols和srcImage2.rows是从srcImage2图像中获取的该图像的高度和宽度，其中.cols是图像的高度而.rows是图像的宽度。需要注意的是，ROI要和掩膜(mask)的大小一致。

3.3.2　线性图像混合

线性图像混合是将图像中的像素点进行叠加，应用的是addweight()函数，其公式如下：

$$g(x) = (1-a)f_1(x) + af_2(x) \tag{3-2}$$

将两幅像素点相同、尺寸相同的图像，根据权重叠加起来即为线性图像混合，通过这种权重叠加，不会出现像素值超出255的情况。

(1) 程序如下：

```
#include<opencv2/opencv.hpp>
#include<opencv2/highgui/highgui.hpp>
using namespace cv;

void main()
{

double a= 0.6;                                  //预设值
double b=1-a;
    Mat srcImage1, srcImage2, dstImage;

    srcImage1= imread("1.jpg");                 //读取图像1
    srcImage2= imread("2.jpg");                 //读取图像2

    namedWindow("原图像1");                     //显示图像1
    imshow("原图像1", srcImage1 );
    namedWindow("原图像2");                     //显示图像2
```

```
    imshow("原图像 2", srcImage2 );

    addWeighted(srcImage1, a, srcImage2, b, 0, dstImage);
                                          //混合图像 1 和图像 2 并存于 dstImage 中

    namedWindow("线性混合结果");          //输出混合图像 dstImage
    imshow("线性混合结果", dstImage );

    waitKey();

}
```

(2) 处理结果。

图 3-21 和图 3-22 为需要混合的两幅图像,图 3-23 为二者线性混合后的图像。

图 3-21 线性混合图像原图 1

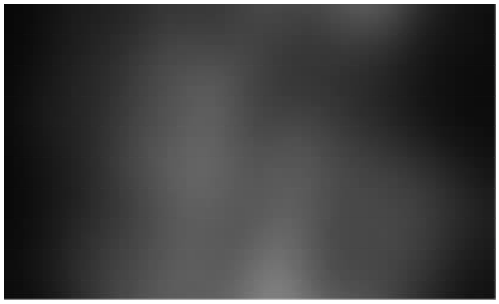

图 3-22 线性混合图像原图 2

图 3-23 线性混合结果图

(3) 程序简析。

本程序主要使用了 addWeight()函数,函数声明如下:

```
void addWeighted( InputArray src1, double alpha, InputArray src2, double beta, double gamma,
OutputArray dst, int dtype = -1);
```

- src1

第一个参数是 InputArray 类型的 src1,存放第一个图像,即要用来加权的第一个数组。

• alpha

第二个参数是 double 类型的 alpha，存放第一个加权数组的权重。

• src2

第三个参数是 InputArray 类型的 src2，存放第二个图像，即要用来加权的第二个数组。

• beta

第四个参数是 double 类型的 beta，存放第二个加权数组的权重。

• gamma

第五个参数是 double 类型的 gamma，存放另外一个加权到该权重上的标量值，可以用来调整图像整体的颜色深度等。

• dst

第六个参数是 OutputArray 类型的 dst，存放即将输出的数组，即作为输出的图像，该图像与原图像的大小和属性必须相同。经过 addWeighted()函数的处理，输出的 dst 像素点的公式如下：

$$dst = src1 \times alpha + src2 \times beta + gamma$$

• dtype

第七个参数是 int 类型的 dtype，存放输出数组 dst 的深度，其默认值为－1，表示与原本的输入数组 src1 和 src2 相同的图像深度。

(4) 注意事项。

addweight()函数仅仅适用于三通道和单通道的图像，如果是 16 位四通道或者是 32 位五通道的图像就不能使用这种方式去实现。

3.3.3 动画效果的线性混合

图像处理偶尔会出现需要图像动态显示的过程，本节仍然使用前面的线性混合函数 addWeighted()，并给出一个如何做简单动态效果的例程——将彩色图像渐变成灰度图像。

(1) 程序如下：

```
#include"stdafx.h"
#include<opencv2/core/core.hpp>
#include<opencv2/imgproc/imgproc.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<iostream>

using namespace cv;
using namespace std;

int main(int argc, char * * argv)
{
    Mat srcImg = imread("Jayce.jpg");
    namedWindow("原图像");
    imshow("原图像", srcImg);

/* ---------------------------计时函数终止端---------------------------- */
    double timeSpent = (double)getTickCount();
/* ------------------------------------------------------------------- */
```

```
    Mat tmpImg1;
    Mat tmpImg2;

    cvtColor(srcImg,tmpImg1,CV_RGB2GRAY);
//将彩色的原图像 srcImg 转换成单通道的灰度图像,并存储在 tmpImg1 中

    cvtColor(tmpImg1,tmpImg2,CV_GRAY2RGB);
//将单通道的灰度图像 tmpImg1 转换成三通道的彩色图像,并存储在 tmpImg2 中

    Mat dstImg;                                          //定义输出图像
    double a;                                            //定义权重
    for(int i = 0; i < 100; i++)
    {
        a = (double)i/100.0;
        addWeighted(srcImg,1 - a,tmpImg2,a,0,dstImg);

        //addWeighted(srcImg,a,tmp3Img,1 - a,0,dstImg);   //反过来的灰度变彩色的过程
        namedWindow("渐变图像");
        imshow("渐变图像",dstImg);
        waitKey(20);                                     //控制渐变速度
    }

/* ----------------------------计时函数终止端---------------------------- */
    timeSpent = ((double)getTickCount() - timeSpent)/getTickFrequency();
    cout <<"Time spent in milliseconds: "<< timeSpent * 1000 << endl;
/* ------------------------------------------------------------------------ */

    waitKey(0);
    return 0;
}
```

(2) 运行结果。

如图 3-24 所示为原始图像,之后会随着时间渐渐变为如图 3-25 所示的图像,最后渐变为如图 3-26 所示的灰度图像。

图 3-24　原始图像

图 3-25　渐变图像

图 3-26　变成灰度图像

（3）程序简析。

程序首先将一个三通道的彩色图像转变成三通道灰度图像，之后使用 for 循环，使两者不断地以不同的权重进行混合，并由新图像不断地覆盖旧图像，从而实现动态显示的过程，如图 3-27 所示。

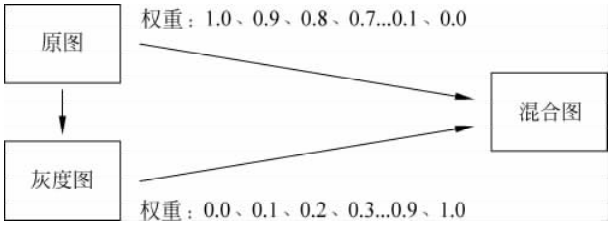

图 3-27　图像渐变原理图

头文件、计时函数、图像的读入输出部分不再重复，需要注意的是，使用 namedWindow() 函数可以实现图像的不断更新，如果直接使用 imshow() 函数则无法这样显示。

另外介绍一个图像变换函数 cvtColor()，该函数声明如下：

```
cvtColor(InputArray src, OutputArray dst, int code, int dstCn = 0);
```

- src

第一个参数 src 是 InputArray 类型的变量，是输入进来的要改变彩色空间的原始图像。

- dst

第二个参数 dst 是 OutputArrary 类型的变量，是将原图像做完彩色空间的转换之后，作为输出的图像。

- code

第三个参数 code 是 int 类型的变量，写入的是要转变的类型，例如程序中的第一个 code 为 CV_RGB2GRAY 代表将三通道 RGB 图像转换成单通道的灰度图像。程序中的第二个 code 为 CV_GRAY2RGB，是将单通道的灰度图像转换为三通道的 RGB 图像。

实现混合需要先得到一个三通道的灰度图像，才能和原图像进行线性混合，单通道的灰度图像不可以实现。但是空间转换中没有可以将三通道图像直接转换为三通道灰度图像的方法，所以采用的方案是先将三通道转换为单通道灰度，因为已经是灰度图，所以再次转换到三通道彩色图像时仍然不会带有任何颜色，通过这样的处理方式，就可以实现使用 addWeighted()函数进行线性混合。

空间转换命名的方式都比较简单，如刚刚说到的 CV_RGB2GRAY，就是将前半段的 RGB 图像转换成后半段中的 GRAY 图像，其他类型的命名也是同样道理。

这里将常用到颜色的空间转换总结如下：

RGB <--> BGR:

CV_BGR2BGRA、CV_RGB2BGRA、CV_BGRA2RGBA、CV_BGR2BGRA、CV_BGRA2BGR;

RGB <--> 5X5:

CV_BGR5652RGBA、CV_BGR2RGB555;

RGB <---> Gray:

CV_RGB2GRAY、CV_GRAY2RGB、CV_RGBA2GRAY、CV_GRAY2RGBA;

RGB <--> CIE XYZ:

CV_BGR2XYZ、CV_RGB2XYZ、CV_XYZ2BGR、CV_XYZ2RGB;

RGB <--> YCrCb(YUV) JPEG:

CV_RGB2YCrCb、CV_RGB2YCrCb、CV_YCrCb2BGR、CV_YCrCb2RGB、CV_RGB2YUV;

RGB <--> HSV:

CV_BGR2HSV、CV_RGB2HSV、CV_HSV2BGR、CV_HSV2RGB;

RGB <--> HLS:

CV_BGR2HLS、CV_RGB2HLS、CV_HLS2BGR、CV_HLS2RGB;

RGB <--> CIE L * a * b * :

CV_BGR2Lab、CV_RGB2Lab、CV_Lab2BGR、CV_Lab2RGB;

RGB <--> CIE L * u * v:

CV_BGR2Luv、CV_RGB2Luv、CV_Luv2BGR、CV_Luv2RGB;

RGB <--> Bayer:

CV_BayerBG2BGR、CV_BayerGB2BGR、CV_BayerRG2BGR、CV_BayerGR2BGR、CV_BayerBG2RGB、CV_BayerGB2RGB;

YUV420 <--> RGB:

CV_YUV420sp2BGR、CV_YUV420sp2RGB、CV_YUV420i2BGR、CV_YUV420i2RGB;

- dstCn

第四个参数 dstCn 是 int 型变量，如果不填则默认为 0，它是输出图像的通道数，若 dstCn＝0，则表示与原图像的通道数保持一致。

3.4 图像去噪

图像信号在获取、传输和存储过程中，不可避免地会受到噪声的干扰，噪声降低了图像的质量，淹没了图像的边缘和细节特征，给图像分析和后续处理带来困难，图像噪声的消除是图像处理中的一个重要研究内容，能否有效地滤除噪声直接影响着图像后续工作的进行，因此图像去噪工作尤为重要[8]。本节将主要介绍 OpenCV 自带的均值滤波、高斯滤波、方框滤波这三种常用的去噪算法，并介绍一种去噪效果更好的非局部均值滤波。

3.4.1 均值滤波

1. 均值滤波算法的原理

均值滤波也称为线性滤波，其采用的主要方法为邻域平均法。线性滤波的基本原理是用均值代替原图像中的各个像素值，即为待处理的当前像素点(x,y)选择一个模板，该模板由其邻近的若干像素组成，求模板中所有像素的均值，再把该均值赋予当前像素点(x,y)，作为处理后图像在该点上的灰度值 $g(x,y)$，即

$$g(x,y)=\frac{\sum f(x,y)}{m} \tag{3-3}$$

式子中的 m 为该模板中包含当前像素在内的像素总个数。

均值滤波能够有效滤除图像中的加性噪声，但均值滤波本身存在着固有的缺陷，即它不能很好地保护图像细节，在图像去噪的同时也破坏了图像的细节部分，从而使图像变得模糊[9]。

2. 均值滤波的算法实现

```
#include<opencv2/opencv.hpp>
#include<opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
usingnamespace cv;
usingnamespace std;

int main()
{
    Mat image=imread("1.jpg");

    namedWindow("滤波前");
    namedWindow("滤波后");
    imshow("滤波前",image);

    Mat out;
    blur(image,out,Size(5,5));                    //均值滤波
```

```
    imshow("滤波后",out);

    waitKey();
    return 0;
}
```

3. 程序运行结果

图 3-28 为原图像，图 3-29 和图 3-30 分别是均值滤波内核为 3×3 和 5×5 的滤波后的图像。

图 3-28　用于均值滤波的原始图像

图 3-29　内核为 3×3 的均值滤波结果

图 3-30　内核为 5×5 的均值滤波结果

4. 程序简析

上述程序调用了 OpenCV 内自带的均值滤波函数 blur()，该函数声明如下：

```
C++: void blur(InputArray src, OutputArray dst, Size ksize, Point anchor = Point(-1,-1), int
borderType = BORDER_DEFAULT )
```

- src

第一个参数 src 是 InputArray 类型的变量，用于存放输入图像，使用 Mat 类的对象即可。该函数对通道独立处理，且可以处理任意通道数的图片。

• dst

第二个参数 dst 是 OutputArray 类型的变量，即目标图像，需要和原图像有一样的尺寸和类型。

• ksize

第三个参数 ksize 是 Size 类型的变量，即为内核的大小。通常的写法是 Size(w,h)表示内核的大小(w 为像素宽度，h 为像素高度)。Size(3,3)就表示 3×3 的核；同样，Size(5,5)就表示 5×5 的内核大小。

• anchor

第四个参数 anchor 是 Point 类型的变量，用来表示锚点(即被平滑的像素点)，注意其默认值为 Point(－1,－1)。如果点坐标是负值，就表示取核的中心为锚点，所以默认值 Point(－1,－1)表示这个锚点在方框核的中心。通常情况下，这个点都是核的中心点，而 anchor 这个参数也不需要填写。

• borderType

第五个参数 borderType 是 int 类型的变量，用于推断图像外部像素的某种边界模式。其默认值为 BORDER_DEFAULT，一般不需要去填写。

5. 程序优化

在图像处理尤其是图像去噪过程中，通常更改取值参数会比较麻烦，每一次都需要更改程序中的参数再运行查看效果。因此，可以通过创建一个轨迹条函数来解决这个问题，本节将简单说明如何使用轨迹条函数来方便查看均值滤波的效果。

程序如下：

```
#include<opencv2/opencv.hpp>
#include<opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>
#include<iostream>
using namespace cv;
using namespace std;

Mat srcImage,dstImage;                                          //定义全局变量的输入和输出
int nBlur=3;                                                    //起始的内核默认值

staticvoid MeanBlur(int,void*);

int main()
{

srcImage=imread("1.jpg");
namedWindow("原始图像");                                        //创建窗口
imshow("原始图像",srcImage);
namedWindow("均值滤波");
createTrackbar("内核值","均值滤波",&nBlur,20,MeanBlur);         //创建轨迹条
MeanBlur(nBlur,0);                                              //进行图像处理

waitKey();
```

```
return 0;
}

staticvoid MeanBlur(int,void * )
{
blur(srcImage,dstImage,Size(nBlur + 1,nBlur + 1));                    //防止窗口为 0 崩溃
imshow("均值滤波",dstImage);
}
```

6. 实验结果

图 3-31 为输入的原图像；图 3-32 是内核大小为 1×1 的均值滤波后图像，即在不滤波情况下的输出图像；图 3-33 是内核大小为 3×3 的均值滤波后图像，图 3-34 是内核大小为 10×10 的均值滤波后图像。

图 3-31　均值滤波原始图像

图 3-32　内核为 1×1 的均值滤波结果

图 3-33　内核为 3×3 的均值滤波结果

图 3-34　内核为 10×10 的均值滤波结果

7. 程序简析

这段程序还是将图像进行均值滤波，唯一的区别在于增加了一个轨迹条函数：

createTrackbar()。

createTrackbar()声明如下：

```
int createTrackbar(conststring& trackbarname, conststring& winname, int * value, int count,
TrackbarCallbackonChange = 0, void * userdata = 0);
```

- trackbarname

第一个参数 trackbarname 是 conststring& 类型的变量，表示滑动线条部分的名称。

- winname

第二个参数 winname 是 conststring& 类型的变量，表示图像显示过程中的名称。如果前边有对应的 namedWindow 创建的图像窗口，那么这个名称就会在相应的图像上进行显示。

- value

第三个参数 value 是 int * 类型的变量，表示在滑动条中，起始时刻滑块的位置。

- count

第四个参数 count 是 int 类型的变量，表示滑动条中的最大值，即滑动条的上限。补充一点，在使用轨迹函数时，最小的滑动条位置是 0，这个是默认存在的，没有办法修改。

本程序中，功能函数为 blur(srcImage, dstImage, Size(nBlur + 1, nBlur + 1))，因为 count 的值最小可以取值到 0，但是在 Size(a,b)中，最小值必须大于 0，因此在这里使用了 +1 的方式，即滑块的位置 +1 即为内核的大小。

- onChange

第五个参数 onChange 是 TrackbarCallback 类型的变量，其自身的默认值为 0。这是一个指向回调函数的指针，每次滑块位置改变时，这个函数都会进行回调。并且这个函数的原型必须为 void XXXX(int, void *)。第一个参数是轨迹条的位置，第二个参数是用户数据。如果回调是 NULL 指针，表示没有使用回调函数，仅第三个参数 value 有变化。

- userdata

第六个参数 userdata 是 void * 型的变量，其自身的默认值为 0，表示用户传给回调函数的数据，用来处理轨迹函数。通常情况下不会更改这个参数，而是直接不填写这个参数，使用其默认值。

3.4.2 高斯滤波

1. 高斯滤波算法的原理

高斯滤波器(Gaussian Filter)是一种时频宽积最小的理想滤波器，有优良的特性。与传统的巴特沃思(Butterworth Filter)等滤波器有一整套成熟的设计理论和方法不同，高斯滤波器尚无完善的设计理论。不过高斯滤波克服了传统滤波相移和设计复杂的缺陷，因此在图像处理、计算机视觉、通信技术、计量测试、时频分析、小波变换等众多领域得到了广泛的应用[10]。

高斯滤波器的脉冲响应函数为：

$$h(x) = \frac{1}{\alpha\lambda_c}\exp\left[-\pi\left(\frac{x}{\alpha\lambda_c}\right)^2\right] \tag{3-4}$$

式子中 $\alpha=\sqrt{\frac{\ln 2}{\pi}}$；$\lambda_c$ 为滤波器的截止波长。

通过一次卷积运算可以将原始信号 $z(x)$分离成为低频信号 $W(x)$和高频信号 $R(x)$两个部分：

$$W(x)=\int_{-\infty}^{+\infty}h(x-\varepsilon)\cdot z(x)\mathrm{d}\varepsilon \tag{3-5}$$

$$R(x)=z(x)-W(x) \tag{3-6}$$

2. 算法实现

```
#include<opencv2/opencv.hpp>
#include<opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>
#include<iostream>
usingnamespace cv;
usingnamespace std;

Mat srcImage,dstImage;
int nGaussian=3;

staticvoid Gaussian(int,void*);

int main()
{
    srcImage=imread("1.jpg");
    namedWindow("原始图像");
    imshow("原始图像",srcImage);

    namedWindow("高斯滤波");
    createTrackbar("内核值:","高斯滤波",&nGaussian,20,Gaussian);
    Gaussian(nGaussian,0);

    waitKey();
    return 0;
}

staticvoid Gaussian(int,void*)
{
    GaussianBlur(srcImage,dstImage,Size(nGaussian*2+1,nGaussian*2+1),0,0);
    imshow("高斯滤波",dstImage);
}
```

3. 实验结果

图 3-35 为原图像，图 3-36 为滤波后参数值取 3 的图像，图 3-37 为图像滤波后参数值为 7 的图像。

4. 程序简析

本程序与前面的均值滤波程序很相似，只是换了一个图像处理函数。除了高斯滤波，OpenCV 中还有很多去噪函数，例如方框滤波、中值滤波等，都是这种应用方式，仅仅是换了一个函数而已，此处不再赘述。

图 3-35　用于高斯滤波的原始图像

图 3-36　高斯滤波内核为 3 的图像

图 3-37　高斯滤波内核为 7 的图像

本次使用的是 GaussianBlur()函数，其声明如下：

```
void GaussianBlur( InputArray src, OutputArray dst, Size ksize, double sigmaX, double sigmaY = 0,
int borderType = BORDER_DEFAULT );
```

- src

第一个参数 src 是 InputArray 类型的变量，其用处是存储 Mat 类的需要进行高斯滤波的原始图像图像数据。

- dst

第二个参数 dst 是 OutputArray 类型的变量，其用处是存储 Mat 类的高斯滤波之后的图像数据。

- ksize

第三个参数 ksize 是 Size 型的变量，表示高斯滤波器的模板大小。

- sigmaX、sigmaY

第四个参数 sigmaX 和第五个参数 sigmaY 都是 double 类型的变量，两者分别表示高斯滤波在横向和纵向的滤波系数。

- borderType

第六个参数是 int 型的变量，表示边缘检测点的插值类型。

3.4.3　非局部均值滤波

1. 非局部均值算法的原理

非局部均值算法是一种图像去噪算法，而领域平均去噪是非局部均值算法的理论基础，

接下来先介绍邻域平均去噪算法的原理，再介绍非局部均值算法的理论。

1）邻域平均去噪算法[11]

假设图像受到加性高斯白噪声的干扰，那么可以得到被干扰后的图像：

$$g(x) = f(x) + \varphi(x) \tag{3-7}$$

其中 $f(x)$表示原本的纯净图像，$g(x)$表示受到干扰之后的输出的图像，$\varphi(x)$表示均值为零的高斯白噪声。

邻域平均的思想是较早提出的一种去噪理念，其利用邻域像素的均值估计中心像素点。该方法是基于这样一种前提假设：噪声在图像局部区域内服从相同的分布，并且像素点的灰度值在非常小的范围内是缓慢变化的，即一定程度上是相似的。因此，可以用邻域像素估计中心像素的值。对于一个给定的像素点 i，设 $N(i)$为所选取的用于平均计算的邻域，则像素 $f(i)$的估计值$\hat{f}(i)$为：

$$\hat{f}(i) = \frac{1}{N}\sum_{j\in N(i)}(f(j)+\varphi(j))N(i) = \frac{1}{N}\sum_{j\in N(i)}f(j) + \frac{1}{N}\sum_{j\in N(i)}\varphi(j) \tag{3-8}$$

式中 N 表示 $N(i)$内像素点的个数，因 $E[\varphi(j)]=0$，若 $f(i)=f(j)$，则有$\hat{f}(i)=\hat{f}(j)$。用 VA 表示像素点 i 去噪后的方差，σ^2 为噪声信号 $\varphi(j)$的方差，则有：

$$\mathrm{VA} = \mathrm{Var}\left\{\frac{1}{N}\sum_{j\in N(i)}\varphi(j)\right\} = \frac{1}{N^2}\sum_{j\in N(i)}\mathrm{Var}[\varphi(j)] = \frac{1}{N^2}N\sigma^2 = \frac{1}{N}\sigma^2 \tag{3-9}$$

由式(3-9)可知，N 值越大，滤波后的像素 i 处的噪声方差就会越小，仅为原来的$\frac{1}{N}$。

在实际去噪过程中，由于噪声的污染，在噪声图像中寻找 $f(i)$与真实值完全相同的像素比较困难，并且在噪声较大时去噪能力有限。虽然完全相同的像素较少，但是在一定邻域内，相似的像素却有很多。为了能够充分利用这一特性，学者们提出了加权的思想。

加权平均的思想利用图像中的自相似信息，根据像素之间的相似程度设置权值的大小。基于加权平均思想的邻域平均去噪算法取得了非常好的去噪效果。此类算法的关键在于如何度量像素之间的相似性或者构造权值函数，这就是非局部均值算法的前身。

2）非局部均值算法

局部去噪和变换域去噪算法在去除噪声的同时，能够恢复图像的主要几何结构信息，但在精细结构、细节信息和纹理的保留上明显不足。图像中的任何一个像素都不是孤立的，而是与其周围的像素点结合在一起共同构成图形的几何结构。以某一个像素点为中心的窗口邻域，可以很好地描述像素点的结构特征。针对任何一个像素点图像块的所有集合可以看作是图像的一种过完备表示。它采用的结构相似性定义像素间的差异，并对像素周围整个区域的灰度分布做整体对比，根据图像中灰度分布的相似性决定权值的大小，如图 3-38 所示，假设 p、q_1、q_2、q_3 具有完全相同的灰度值，那么 q_1、q_2、q_3 三者都会根据与 p 点不同的欧氏距离而得到相应的权值，但 q_1 和 q_2 的邻域灰度分布与 p 更接近，因此贡献更大的权值，q_3 则对 p 贡献较小的权值[12]。

图 3-38　NLM 算法原理

假设一幅含噪图像 $z=\{z(i)\mid i\in I\}$，其定义在有界域

$I \in N^2$。在这幅图像中，对于某个像素点 i，非局部均值滤波算法利用所有的像素的加权平均来得到该点的估计值 $NL(z)(i)$，即：

$$NL(z)(i) = \sum_{j \in I} w(i,j) z(j) \tag{3-10}$$

其中，权值$\{w(i,j)\}_j$ 依赖于像素 i 与像素 j 之间的相似性，且满足如下条件：$0 \leqslant w(i,j) \leqslant 1$ 且 $\sum_j w(i,j) = 1$。

图像域 I 上的邻域系统 $N = \{N_i\}_{i \in I}$是图像域 I 的子集，使得对于所有的像素点 $i \in I$ 都必须满足以下两个条件：

- $i \in N_i$；
- $j \in N_i \Rightarrow i \in N_j$；

其中，N_i 是像素 i 的窗口邻域。

为了能够更好地适应图像不同区域的特征，可以将相似性窗口取不同的形状和大小。为了方便起见，这里使用固定大小的方形窗口。相似性窗口 N_i 内的灰度值向量可以表示如下：

$$z(N_i) = (z(j), j \in N_i) \tag{3-11}$$

灰度值向量 $z(N_i)$和 $z(N_j)$之间的相似性可以用来决定像素点 i 和像素点 j 之间的相似性，即在加权平均时，那些与 $z(N_i)$具有相似灰度值向量的像素点将被分配较大的权值，反之则被分配到较小的权值。为了能够定量地计算 $z(N_i)$和 $z(N_j)$之间的相似性，可以采用高斯加权的欧氏距离 $\| z(N_i) - z(N_j) \|_{2,a}^2$。在含噪声的图像与滤波后的图像对应位置的窗口内，灰度值向量之间的欧氏距离满足如下关系：

$$E \| z(N_i) - z(N_j) \|_{2,a}^2 = \| y(N_i) - y(N_j) \|_{2,a}^2 + 2\sigma^2 \tag{3-12}$$

其中，z 与 y 分别表示带有噪声图像与滤波后的图像，σ^2 是噪声的方差。

基于以上的式子，可以得到像素点 i 和像素点 j 之间的权值 $w(i,j)$：

$$w(i,j) = \frac{1}{Z(i)} \exp\left(\frac{- \| z(N_i) - z(N_j) \|_{2,a}^2}{h^2} \right) \tag{3-13}$$

其中 $Z(i) = \sum_j \exp\left(\frac{- \| z(N_i) - z(N_j) \|_{2,a}^2}{h^2} \right)$ 是归一化常数；$\| z(N_i) - z(N_j) \|_{2,a}^2$是指 i 块和 j 块的加权欧式距离的平方，用 $d(i,j)$来表示，$a(a>0)$是指高斯核的标准差，由选定像素邻域的窗口大小决定。参数 h 控制指数函数的衰减速度，同时影响着权值的衰减速度，$h = c \times \sigma$。c 是用于调整的系数，Buades 将其范围规定在 1～10 之间。

最终可以将非局部均值滤波的算法整理为如下 3 个算式：

$$d(i,j) = \| z(N_i) - z(N_j) \|_{2,a}^2 \tag{3-14}$$

$$w(i,j) = \exp\left(- \frac{d(i,j)}{h^2} \right) \tag{3-15}$$

$$z(i) = \frac{\sum_{j \in I} w(i,j) z(j)}{\sum_{j \in I} w(i,j)} \tag{3-16}$$

图 3-39 显示了非局部均值滤波算法的执行的过程。在算法执行的过程中，需要设置两个窗口的大小：一个是像素邻域窗口尺寸 $K \times K$，一个是像素邻域窗口搜索范围的窗口尺寸 $L \times L$，

即在 $L\times L$ 大小的窗口内选择像素的邻域大小为 $K\times K$ 执行非局部均值滤波算法，$K\times K$ 的窗口在 $L\times L$ 的区域内滑动，根据区域的相似性确定区域中心像素灰度的贡献权值，而在这里的图像处理实现中，窗口尺寸为 $K=7$，滑动范围为 $L=17$。

在相对稳定的条件下（即图像有足够大的尺寸时），对于图像内部的各种细节都能找到足够多的相似区域。

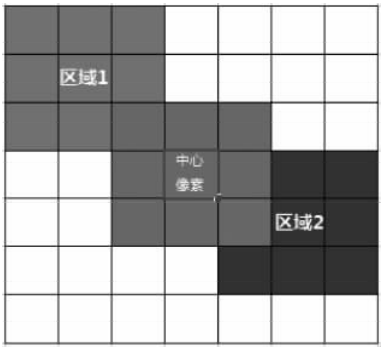

图 3-39　NLM 算法执行示意图

2. NLM 算法实现

```
#include<opencv2/opencv.hpp>
#include<iostream>
usingnamespace cv;
usingnamespace std;

void addNoiseSoltPepperMono(Mat& src, Mat& dest,double per)
{
    cv::RNG rng;
#pragma omp parallel for
for(int j=0;j<src.rows;j++)
    {
        uchar* s=src.ptr(j);
        uchar* d=dest.ptr(j);
for(int i=0;i<src.cols;i++)
        {
double a1 = rng.uniform((double)0, (double)1);

if(a1>per)
                d[i]=s[i];
else
            {
double a2 = rng.uniform((double)0, (double)1);
if(a2>0.5)d[i]=0;
else d[i]=255;
            }
        }
    }
}
void addNoiseMono(Mat& src, Mat& dest,double sigma)
{
    Mat s;
    src.convertTo(s,CV_16S);
    Mat n(s.size(),CV_16S);
    randn(n,0,sigma);
    Mat temp = s+n;
    temp.convertTo(dest,CV_8U);
}
```

```
void addNoise(Mat&src, Mat& dest, double sigma,double sprate = 0.0)
{
if(src.channels() == 1)
    {
        addNoiseMono(src,dest,sigma);
if(sprate!= 0)addNoiseSoltPepperMono(dest,dest,sprate);
return;
    }
else
    {
        vector < Mat > s;
        vector < Mat > d(src.channels());
        split(src,s);
for(int i = 0;i < src.channels();i++)
        {
            addNoiseMono(s[i],d[i],sigma);
if(sprate!= 0)addNoiseSoltPepperMono(d[i],d[i],sprate);
        }
        cv::merge(d,dest);
    }
}

staticdouble getPSNR(Mat& src, Mat& dest)
{
int i,j;
double sse,mse,psnr;
    sse = 0.0;
for(j = 0;j < src.rows;j++)
    {
        uchar * d = dest.ptr(j);
        uchar * s = src.ptr(j);
for(i = 0;i < src.cols;i++)
        {
            sse += ((d[i] - s[i]) * (d[i] - s[i]));
        }
    }
if(sse == 0.0)
    {
return 0;
    }
else
    {
        mse = sse /(double)(src.cols * src.rows);
        psnr = 10.0 * log10((255 * 255)/mse);
return psnr;
    }
}

double calcPSNR(Mat& src, Mat& dest)
{
    Mat ssrc;
```

```
    Mat ddest;
if(src.channels() == 1)
    {
        src.copyTo(ssrc);
        dest.copyTo(ddest);
    }
else
    {
        cvtColor(src,ssrc,CV_BGR2YUV);
        cvtColor(dest,ddest,CV_BGR2YUV);
    }
double sn = getPSNR(ssrc,ddest);
return sn;
}

void nonlocalMeansFilter(Mat& src, Mat& dest, int templeteWindowSize, int searchWindowSize,
double h, double sigma = 0.0)
{
if(dest.empty())dest = Mat::zeros(src.size(),src.type());
    constint tr = templeteWindowSize >> 1;
constint sr = searchWindowSize >> 1;
constint bb = sr + tr;
    constint D = searchWindowSize * searchWindowSize;
    constint H = D/2 + 1;
constdouble div = 1.0/(double)D;
constint tD = templeteWindowSize * templeteWindowSize;
constdouble tdiv = 1.0/(double)(tD);
    Mat im;
    copyMakeBorder(src,im,bb,bb,bb,bb,cv::BORDER_DEFAULT);
    vector<double> weight(256 * 256 * src.channels());
    double * w = &weight[0];
constdouble gauss_sd = (sigma == 0.0) ? h :sigma;
double gauss_color_coeff = -(1.0/(double)(src.channels())) * (1.0/(h * h));
for(int i = 0; i < 256 * 256 * src.channels(); i++)
    {
double v = std::exp( max(i - 2.0 * gauss_sd * gauss_sd,0.0) * gauss_color_coeff);
        w[i] = v;
    }

constint cstep = im.step - templeteWindowSize * 3;
constint csstep = im.step - searchWindowSize * 3;
for(int j = 0;j < src.rows;j++)
        {
            uchar * d = dest.ptr(j);
int * ww = newint[D];
double * nw = newdouble[D];
for(int i = 0;i < src.cols;i++)
            {
double tweight = 0.0;
                uchar * tprt = im.data + im.step * (sr + j) + 3 * (sr + i);
                uchar * sptr2 = im.data + im.step * j + 3 * i;
```

```
                for(int l = searchWindowSize,count = D - 1;l -- ;)
                    {
                        uchar * sptr = sptr2 + im.step * (l);
for (int k = searchWindowSize;k -- ;)
                        {

int e = 0;
                            uchar * t = tprt;
                            uchar * s = sptr + 3 * k;
for(int n = templeteWindowSize;n -- ;)
                            {
for(int m = templeteWindowSize;m -- ;)
                                {
        e += (s[0] - t[0]) * (s[0] - t[0]) + (s[1] - t[1]) * (s[1] - t[1]) + (s[2] - t[2]) * (s[2] - t[2]);
                                    s += 3,t += 3;
                                }
                                t += cstep;
                                s += cstep;
                            }
constint ediv = e * tdiv;
                            ww[count -- ] = ediv;
                            tweight += w[ediv];
                        }
                    }
if(tweight == 0.0)
                    {
for(int z = 0;z < D;z++) nw[z] = 0;
                        nw[H] = 1;
                    }
else
                    {
double itweight = 1.0/(double)tweight;
for(int z = 0;z < D;z++) nw[z] = w[ww[z]] * itweight;
double r = 0.0,g = 0.0,b = 0.0;
                    uchar * s = im.ptr(j + tr); s += 3 * (tr + i);
for(int l = searchWindowSize,count = 0;l -- ;)
                    {
for(int k = searchWindowSize;k -- ;)
                        {
                            r += s[0] * nw[count];
                            g += s[1] * nw[count];
                            b += s[2] * nw[count++];
                            s += 3;
                        }
                        s += csstep;
                    }
                    d[0] = saturate_cast < uchar >(r);
                    d[1] = saturate_cast < uchar >(g);
                    d[2] = saturate_cast < uchar >(b);
                    d += 3;
                    }
```

```
            }
delete[] ww;
delete[] nw;
        }

}

int main(int argc, char * * argv)
{
constdouble noise_sigma = 15.0;

    Mat src = imread("lena512.jpg",1);

    Mat snoise;
    Mat dest;
    addNoise(src,snoise,noise_sigma);
    int64 pre = getTickCount();
    pre = getTickCount();
    nonlocalMeansFilter(snoise,dest,3,7,noise_sigma,noise_sigma);

    cout <<"time: "<< 1000.0 * (getTickCount() - pre)/(getTickFrequency())<<" ms"<< endl;
    cout <<"nonlocal: "<< calcPSNR(src,dest)<< endl << endl;
    imwrite("nonlocal.png",dest);
    imshow("noise", snoise);
imshow("Non - local Means Filter", dest);
    waitKey();
return 0;
}
```

3. 程序运行结果

如图 3-40 所示，左图为原图像，中间图为加噪后图像，右边为 NLM 算法去噪后的图像。

图 3-40 NLM 算法结果

4. NLM 算法的 MATLAB 实现

MATLAB 程序相比于 C++更好理解，接下来将给出一套 NLM 算法使用 MATLAB 语言实现的程序，但该程序只能处理单通道的灰度图像，程序如下：

```
clear all
```

```
close all
n = 3;
sigma = 8;
M = 8;
T = sigma ^ 2;
y00 = imread('lena64.jpg');
y0 = y00(1:64,1:64);
y0 = double(y0);
N = size(y0,1);
noise = randn(N,N);
y = y0 + sigma * noise;
tic//计时代码 tic toc
figure(1);
clf;
image([y0,y]);
colormap(gray(256));
axis image;
axis off;
drawnow;
yout = zeros(N,N);
h = waitbar(0,'NLM filtering ...');
y = [y(:,M + n: - 1:1),y,y(:,end - M - n + 1:end)];
y = [y(M + n: - 1:1,:); y; y(end - M - n + 1:end,:)];
for i = n + M + 1:1:N + M + n
    waitbar((i - M - n)/N);
for j = n + M + 1:1:N + M + n
        Center = y(i - n:i + n,j - n:j + n);
Weights = zeros(2 * M + 1,2 * M + 1);
for p = - M:M
for q = - M:M
                Patch = y(i + p - n:i + p + n,j + q - n:j + q + n);
                dist2 = mean((Patch(:) - Center(:)).^2);
                Weights(p + M + 1,q + M + 1) = exp( - dist2/T);
end;
end;
        Weights = Weights/sum(Weights(:));
        yout(i - n - M,j - n - M) = sum(sum(y(i - M:i + M,j - M:j + M). * Weights));
end;
end;
close(h)
y = y(M + n + 1:M + n + N,M + n + 1:M + n + N);
figure(2);
clf;
image([y0,y,yout]);
colormap(gray(256));
axis image;
axis off;
drawnow;
toc
```

3.5　双目视觉测量物体深度

在机器视觉领域中，双目视觉是一种应用很广泛的手段。双目视觉利用两台摄像机同时对物体进行拍摄，根据景物点在左右摄像机图像上位置关系，可以计算出景物点的三维坐标，从而可以实现三维测量和恢复。双目视觉测量系统因其结构简单、操作方便、成本低，以及具有在线、实时测量的潜力，而被广泛应用于机器人指导、工业生产现场以及航空等诸多领域[13]。

图 3-41　双目摄像头

如图 3-41 所示为常见的双目摄像头。

3.5.1　双目视觉原理

双目视觉测量的系统模型如图 3-42 所示，设左侧摄像机坐标为 $o_1x_1y_1z_1$，右侧摄像机坐标为 $o_2x_2y_2z_2$，选取左侧摄像机坐标系为世界坐标系，左侧理想图像坐标系为 $O_1X_1Y_1$，右侧理想图像坐标为 $O_2X_2Y_2$，f_1 和 f_2 分别为左右摄像机的焦距，像元尺寸为 w_1 和 w_2，则由空间几何关系可以得到空间点 P 在测量坐标系下的三维坐标为：

$$\begin{cases} X_w = \dfrac{B\cot(\omega_1+\alpha_1)}{\cot(\omega_1+\alpha_1)+\cot(\omega_2+\alpha_2)} \\ Y_w = Y_1\dfrac{z\sin\omega_1}{f_1\sin(\omega_1+\alpha_1)} = Y_2\dfrac{z\sin\omega_2}{f_2\sin(\omega_2+\alpha_2)} \\ Z_w = \dfrac{B}{\cot(\omega_1+\alpha_1)+\cot(\omega_2+\alpha_2)} \end{cases} \tag{3-17}$$

式子中：$\omega_1=\arctan(X_1/f_1)$，$\omega_2=\arctan(X_2/f_2)$，$z$ 为物距，B 为系统基线距[14]。

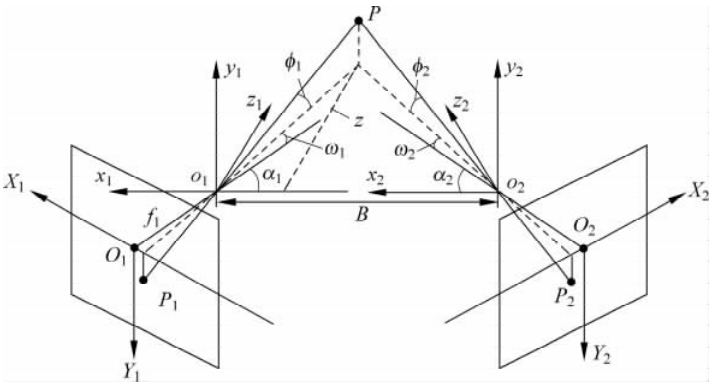

图 3-42　双目视觉原理图

3.5.2　双目视觉标定

双目视觉可以确定物体的空间坐标，还可以通过这种方式来测量物体距离，也就是物体和摄像头之间的距离，这也被称作测量物体的深度，实际测量中需要注意很多现实的问题。

1. 预处理

在理论上，双目摄像机的两个摄像头即便焦距不同也可以实现双目视觉测量物体的深度，但是实际上在操作时务必使用两个焦距相同的摄像头。如果是使用两个单目摄像头拼装成的双目摄像头，请尽量固定好两个摄像头之间的距离，并且保证两个单目摄像头不会有高度差，尽可能地保持水平等。通常如果使用了集成好的双目摄像头，这样拍出的两个图像会集成在一个图片上，这种图像则需要进行一次图像分割预处理。如果是两个单独的摄像头，或者是双目摄像头拍出的两个图像是分开进行存储的，一定要注意每次拍摄好的图像要分别重命名，如 left1 和 right1，否则很容易在处理之前发生两个图像不配套的情况。

2. 摄像机标定

在预处理结束之后，就需要对摄像机进行标定，为后续的双目摄像头测深度做好预处理。标定相当于使用一些手段获取到双目摄像头的焦距、双目摄像头之间的距离等信息。因此不论是使用 MATLAB 工具箱进行标定，还是使用 OpenCV 标定甚至是手动输入都是可以的，不过 MATLAB 工具箱标定的精度更高。本节将介绍两种使用 MATLAB 工具箱进行标定的方法，OpenCV 标定则在后面的程序实现中进行介绍。

方法一：calib 工具箱

calib 工具箱不是 MATLAB 自身集成进去的工具箱，但是其标定的效果通常比其他标定方法好，因此这种标定方式还在普遍使用中，下面将介绍如何使用 calib 工具箱进行标定[16]。

(1) MATLAB 和 calib 的下载。

MATLAB 的版本没有限制，后续的标定将以 MATLAB2016b 来作示例，calib 工具箱全称是 TOOLBOX_calib。

下载地址：http://www.vision.caltech.edu/bouguetj/calib_doc/

CSDN 下载地址：http://download.csdn.net/download/kevinfrankchen/9454023

百度网盘：链接：http://pan.baidu.com/s/1c2Eurc 密码：l9x8

(2) 解压并安装。

下载完成后需要将其解压，将文件夹名称修改为 calib，并放在 MATLAB 的默认路径下或者是放在 MATLAB 的 toolbox 文件夹中。示例中的路径为：

```
E:\MATLAB2015B\toolbox
```

解压后的效果如图 3-43 所示。

(3) 在 MATLAB 中添加 calib 工具箱。

将 calib 放进指定的路径之后，就将 MATLAB 打开，在“主页”界面中，找到“设置路径”选项，并打开，如图 3-44 所示。

进入添加路径界面之后单击“添加并包含子文件夹…”按钮，添加路径和对应路径下的子文件夹，如图 3-45 所示。

找到之前放置的 calib 文件夹，单击“选择文件夹”将其添加至路径中，如图 3-47 所示。最后回到如图 3-46 所示的界面中，单击“保存”按钮即可。至此，calib 标定工具箱添加完成。

图 3-43　calib 安放位置

图 3-44　MATLAB 添加工具箱第一步

图 3-45　MATLAB 添加工具箱第二步

图 3-46　MATLAB 添加工具箱第三步

(4) 在 MATLAB 中添加图像路径。

首先将左边相机拍到的图像按照拍照的先后顺序排列好，并依次命名为“left_＋数字”的形式，如 left_1. bmp、left_2. bmp。将右摄像头拍到的图像依次命名为“right_＋数字”的形式。最后将两个摄像头拍到的图像分别放在两个文件夹中，路径文件尽量不包含中文，虽然高版本的 MATLAB 可以兼容中文路径，但是仍然可能出现错误。如图 3-47 和图 3-48 所示为双目摄像头分别拍摄到的图像。为了演示方便，将左摄像头取出前十张图像命名为 left_1. bmp、left_2. bmp、…、left_10. bmp。同样，右摄像头取出与左摄像头对应的十张图像也分别命名为 right_1. bmp、right_2. bmp、…、right_10. bmp。

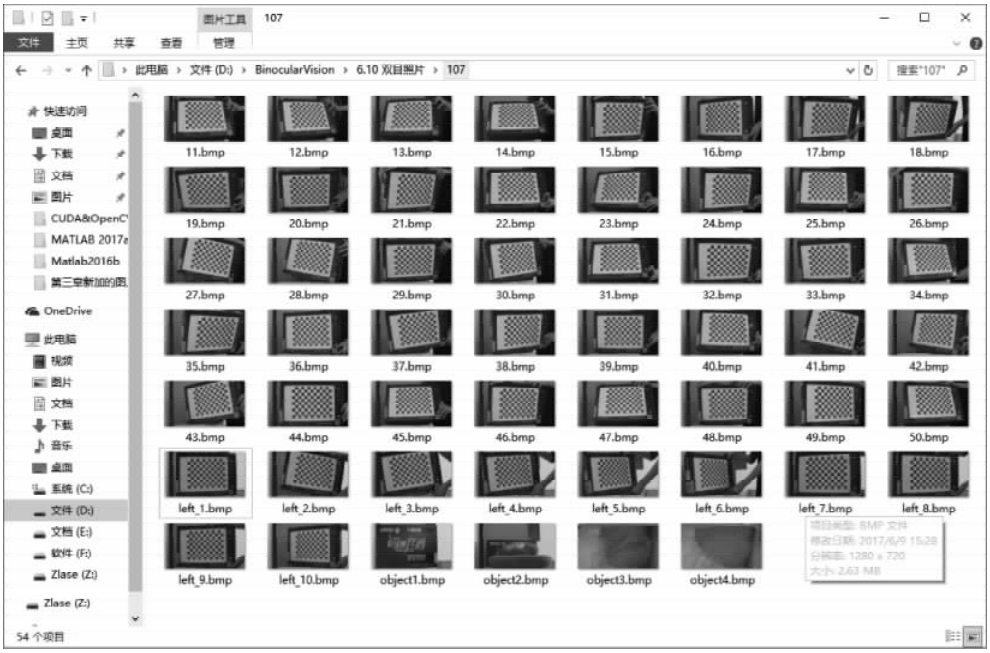

图 3-47　左摄像头拍到的图像

两个文件夹的路径分别为：

```
D:\BinocularVision\6.10_双目照片\107
D:\BinocularVision\6.10_双目照片\108
```

图 3-48　右摄像头拍到的图像

之后打开 MATLAB 中的“设置路径”界面，并单击“添加并包含子文件夹…”按钮，将两个文件夹添加到 MATLAB 的 path 中，如图 3-49 所示。

图 3-49　MATLAB 中添加图像路径第一步

将两个路径添加进去之后会在“设置路径”界面中出现两个路径，单击“保存”按钮之后关闭“设置路径”界面即可，如图 3-50 所示。

(5) 单个摄像头标定。

所有需要用到的路径添加完成后，在 MATLAB 命令窗口中输入 calib_gui，会出现如

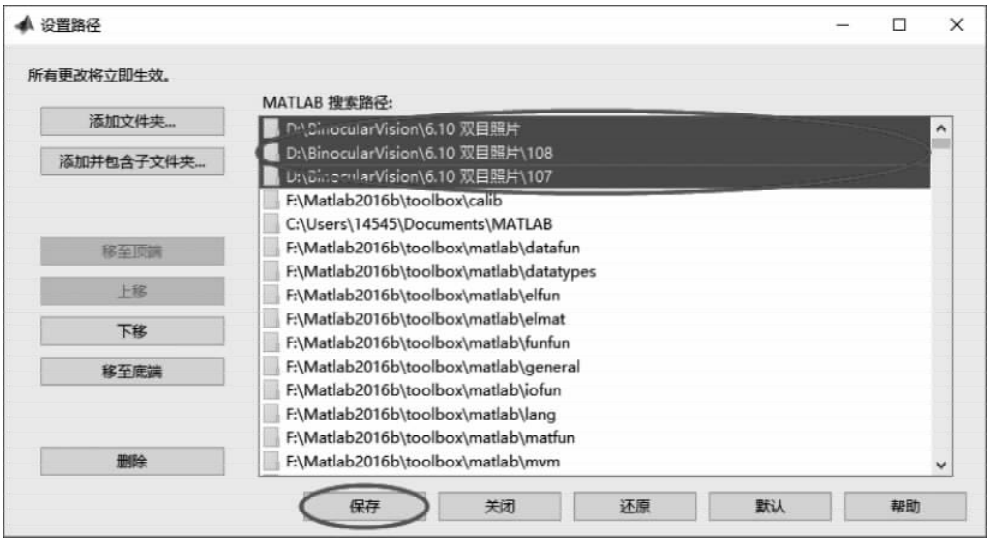

图 3-50　MATLAB 中添加图像路径第二步

图 3-51 所示的窗口，单击第一个选项 Standard(all the images are stored in memory)，会出现如图 3-52 所示的菜单栏。

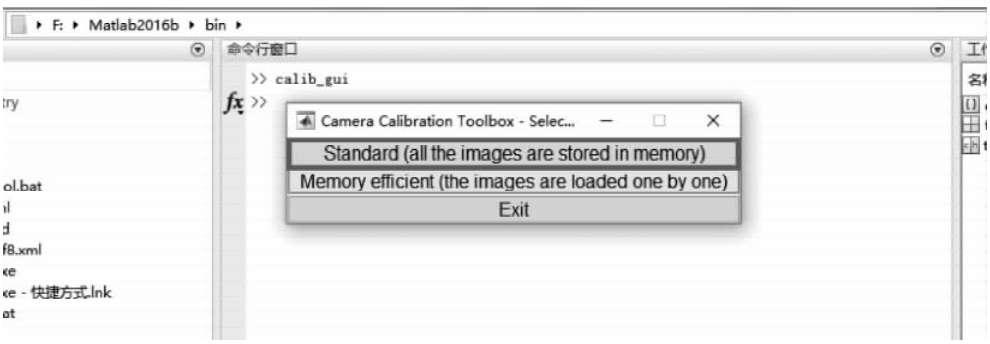

图 3-51　调用标定函数

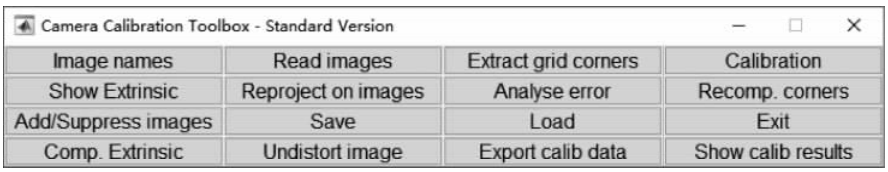

图 3-52　标定菜单栏

回到 MATLAB 的命令行窗口中，将路径改为左图像存储的路径，如图 3-53 所示。

单击图 3-52 中的 Image names，进行图像读入。如图 3-54 所示为读入图像后的画面，图上方为文件夹中所有图像，在下方的第一行输入 left_表示从上述图像中仅提取文件名以 left 开始的图像。

在第二栏中输入 b 表示仅输入.bmp 格式的文件，导入成功后如图 3-55 和图 3-56 所示。

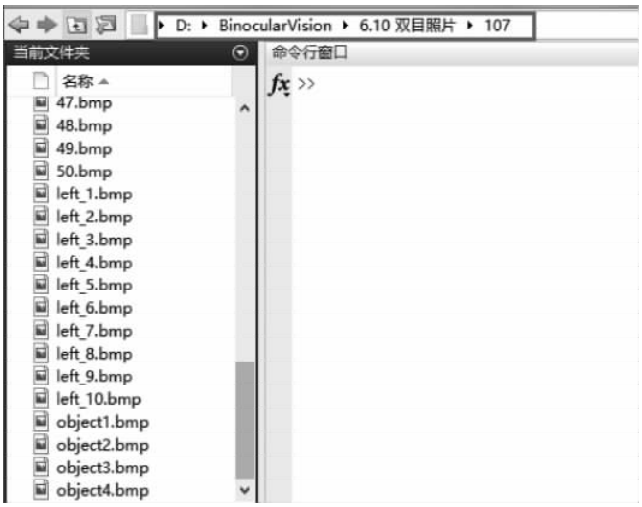

图 3-53　修改 Matlab 的路径

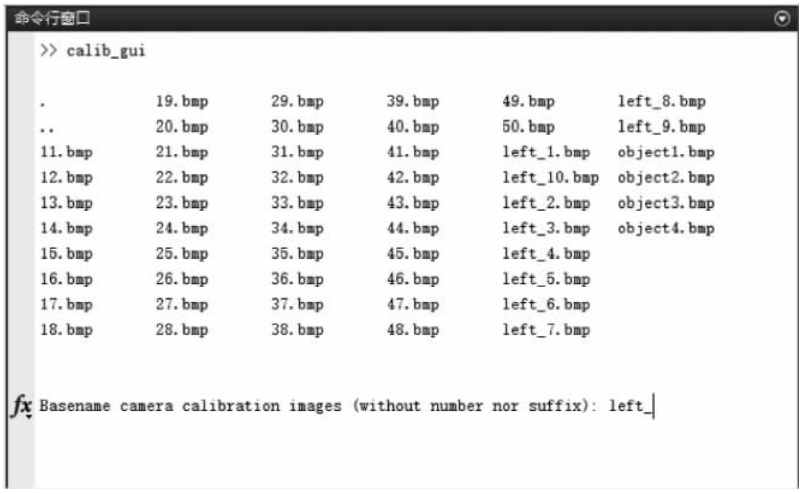

图 3-54　标定函数图像读入

```
Basename camera calibration images (without number nor suffix): left_
Image format: ([]='r'='ras', 'b'='bmp', 't'='tif', 'p'='pgm', 'j'='jpg', 'm'='ppm') b
Loading image 1...2...3...4...5...6...7...8...9...10...
done
```

图 3-55　标定函数成功读入图像 1

回到主控界面，单击 Extract grid corners 提取每幅图像的角点。单击完成之后，命令行会出现如图 3-57 所示的提示。这几项都可以直接按 Enter 键跳过，它们表示以多大的窗口去搜索棋盘窗，较大的窗口会更方便提取，即便点有适当偏离也能找到。

在按 Enter 键之后会跳出第一个棋盘窗，按照顺时针的顺序，依次选中棋盘中外侧第二圈的角点（不要单击最外侧的角点），如图 3-58 所示。

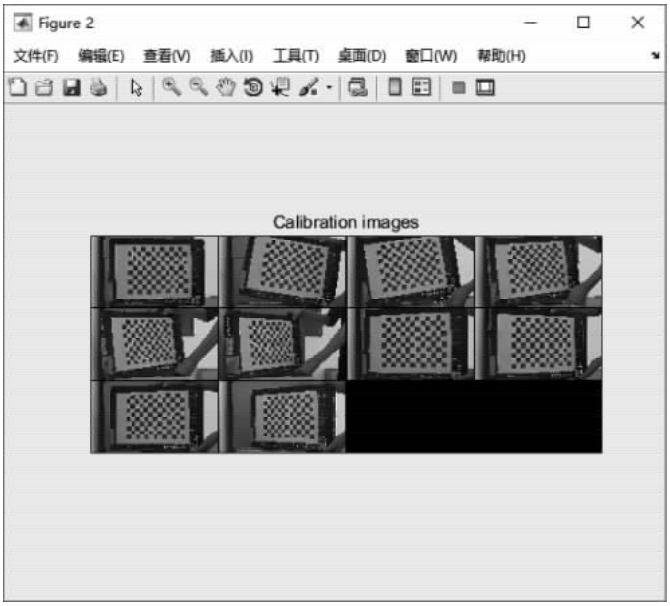

图 3-56　标定函数成功读入图像 2

```
Extraction of the grid corners on the images
Number(s) of image(s) to process ([] = all images) =
Window size for corner finder (wintx and winty):
wintx ([] = 10) =
winty ([] = 10) =
Window size = 21x21
Do you want to use the automatic square counting mechanism (0=[]=default)
  or do you always want to enter the number of squares manually (1,other)? 1
```

图 3-57　角点提取

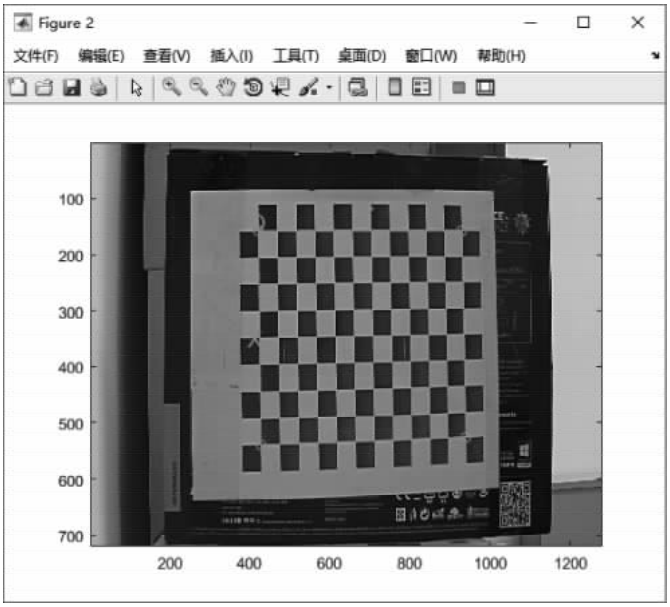

图 3-58　提取角点

单击之后需要输入一些棋盘的参数信息，首先输入图像中每一个方块的长和宽，单位是毫米，如图 3-59 所示。

```
Processing image 1...
Using (wintx,winty)=(10,10) - Window size = 21x21      (Note: To reset the window size, run script clearwin)
Click on the four extreme corners of the rectangular complete pattern (the first clicked corner is the origin)...
Size dX of each square along the X direction ([]=100mm) = 18
Size dY of each square along the Y direction ([]=100mm) = 18
If the guessed grid corners (red crosses on the image) are not close to the actual corners,
it is necessary to enter an initial guess for the radial distortion factor kc (useful for subpixel detection)
Need of an initial guess for distortion? ([]=no, other=yes)
```

图 3-59　输入标定参数

接下来按 Enter 键，依次标定剩下 9 张图像的角点，最后回到菜单界面。在菜单界面单击 Calibration 按钮进行标定。

标定完成后，即可得到标定的结果，单击 Show Extrinsic 可视化标定的结果，如图 3-60 所示。

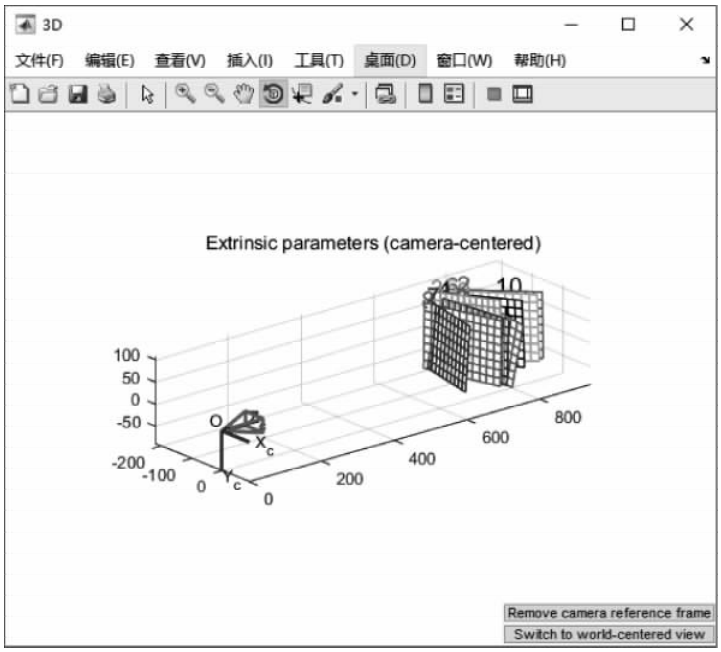

图 3-60　标定结果可视化

单击 Analyse Error 可以查看标定的误差情况，如图 3-61 所示。

标定过一个摄像头后，单击 Save 按钮保存结果。将保存的 Calib_Results. mat 文件的名称改为 Left_ Calib_Results. mat，并放在另一个文件夹中。

完成左摄像头标定后，对右摄像头的图像使用同样的方法进行标定，将保存的结果 Calib_Results. mat 名称改为 Right_Calib_Results. mat 并放在之前存放 Left_ Calib_Results. mat 的文件夹中。

左、右摄像头都标定完成后，在 MATLAB 的命令行窗口中输入 stereo_gui 启动立体标定板，如图 3-62 所示。

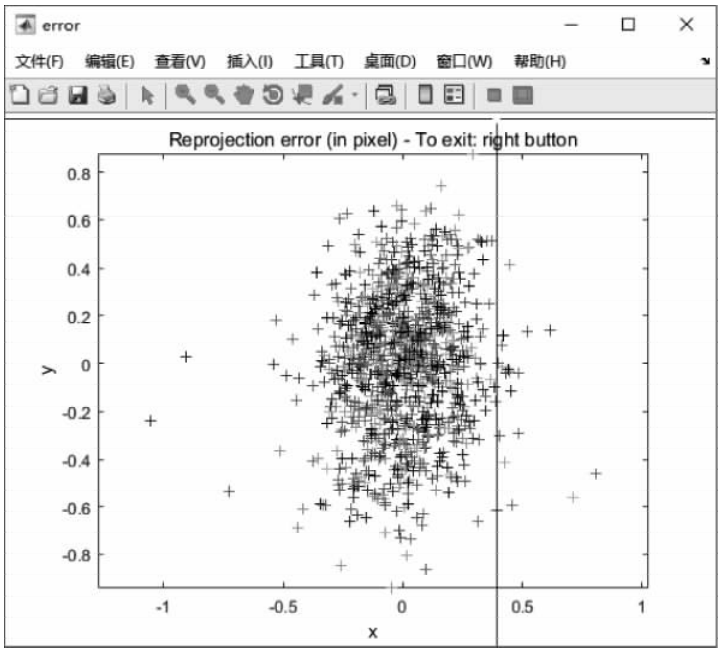

图 3-61　标定误差分析

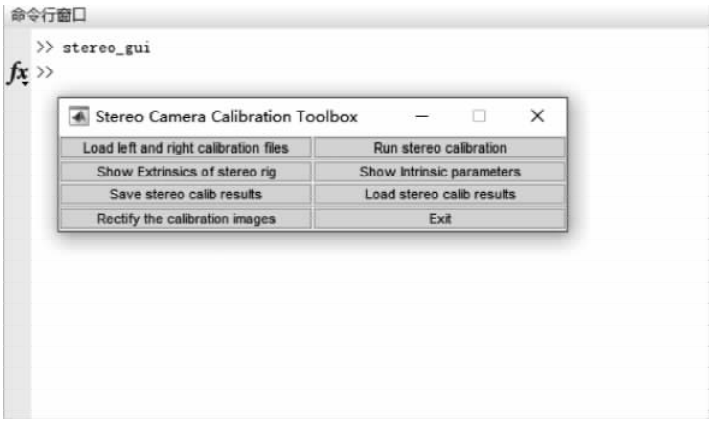

图 3-62　启动立体标定板

将 MATLAB 的默认路径改为存放两个.mat 文件的文件夹，如图 3-63 所示。

单击 Load left and right calibration files，在命令行窗口中分别填写左、右标定好的结果，如图 3-64 所示。

输入完成之后单击 Run stereo calibration 对左右参数进行修正，之后再单击 Show Extrinsics of stereo rig 即可将结果可视化，如图 3-65 和图 3-66 所示。

最后单击 Save stereo calib results 即可保存标定后的结果。

方案二：扩展标定 APP

(1) 在版本比较新的 MATLAB 中，可以在一侧找到附加的 APP，如图 3-67 所示，可以在其中找到一个名为 Stereo Camera Calibrator 的扩展 APP，其功能就是双目立体视觉的标定。

图 3-63　立体标定_修改路径

```
Loading of the individual left and right camera calibration files
Name of the left camera calibration file ([]=Calib_Results_left.mat): Left_Calib_Results.mat
Name of the right camera calibration file ([]=Calib_Results_right.mat): Right_Calib_Results.mat
Loading the left camera calibration result file Left_Calib_Results.mat...
Loading the right camera calibration result file Right_Calib_Results.mat...
```

图 3-64　输入两个.mat 文件

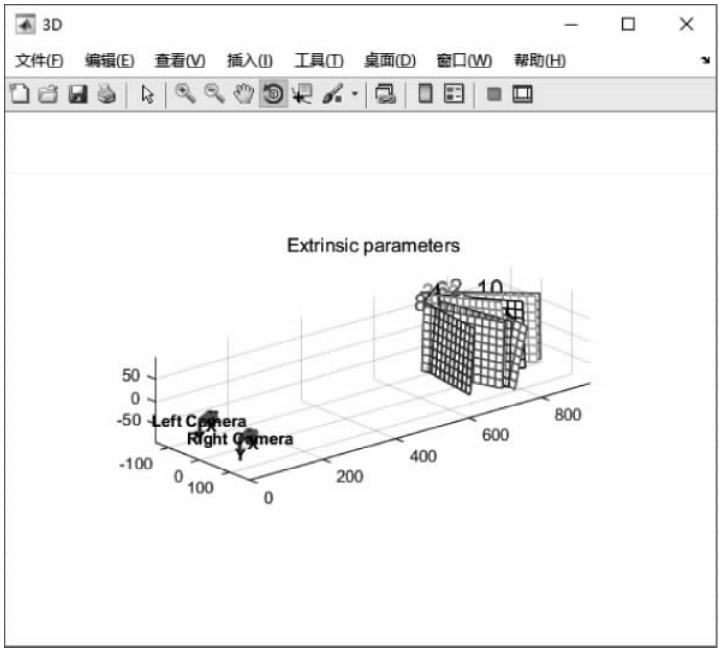

图 3-65　立体标定可视化主视图

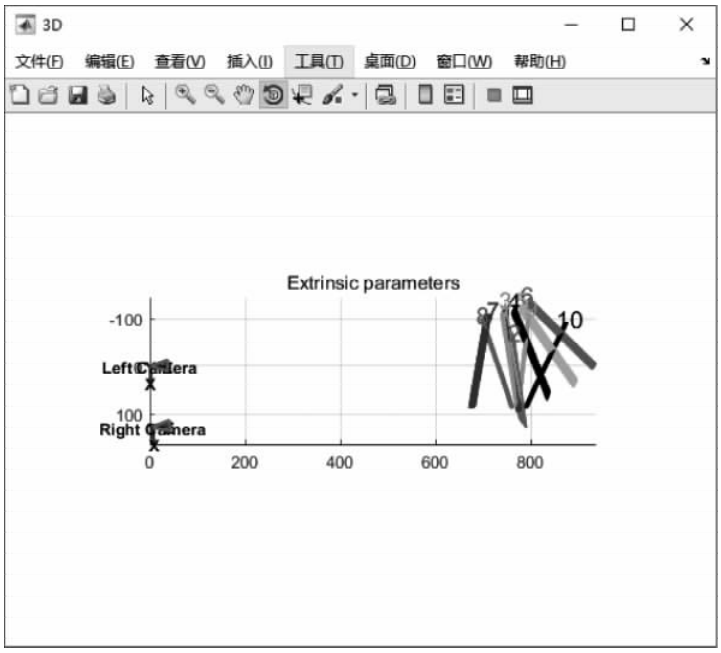

图 3-66　立体标定可视化俯视图

图 3-67　MATLAB 标定第一步

（2）安装好 APP 后，如图 3-68 所示，单击左上角的 Add Images 添加左、右摄像头的图像。

（3）在添加图片时，需要在上边的一栏中添加用于存储左摄像头拍摄到的图片的文件夹，下边则需要添加用于存储右摄像头拍摄到的图片的文件夹，如图 3-69 所示，最下边需要填好每一个标定格的宽度。

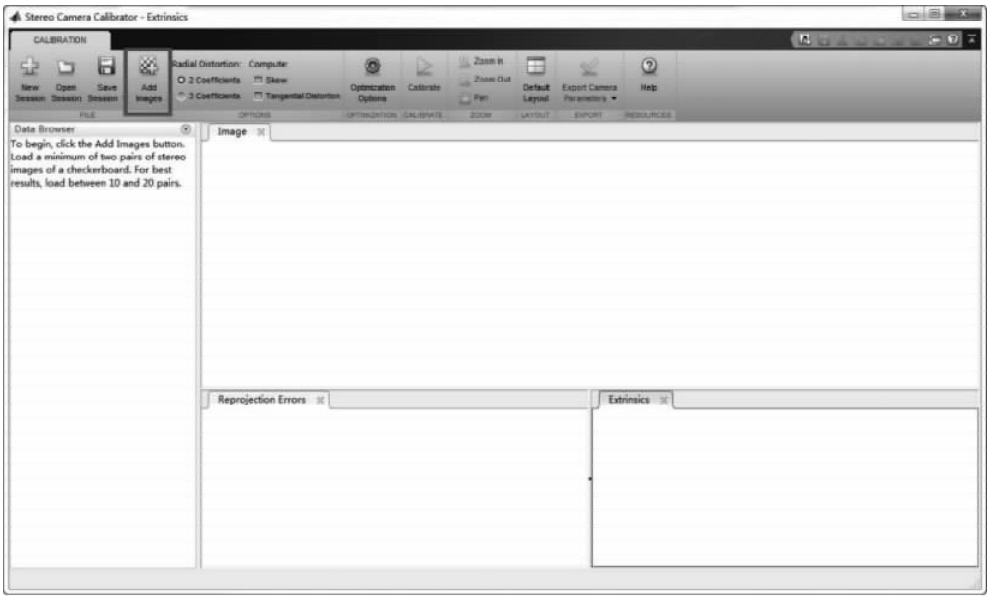

图 3-68　MATLAB 标定第二步

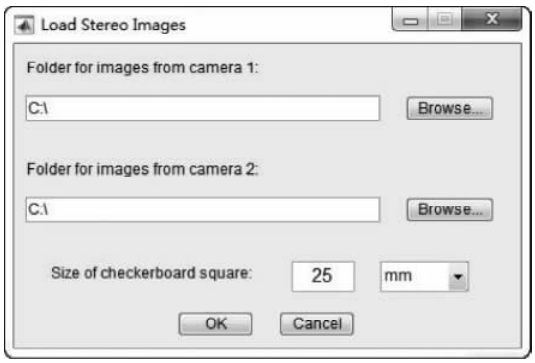

图 3-69　MATLAB 标定第三步

需要注意的是，虽然对文件夹中的图片名称没有明确要求，但是为了避免不必要的麻烦，提高匹配的成功率，一定要把拍摄到的图片按照对应的顺序排好，即按照标定板每个角度拍摄的顺序进行排序，如图 3-70 所示为示例文件夹存储的图片。

(4) 之后进入标定，如图 3-71 所示，在这个过程中耐心等待即可。

标定完成后如图 3-72 所示，可以删除标定效果不好的图片，以提高标定的精度。本示例中使用的图片较少，在实际标定过程中使用的图片越多越好。

(5) 在完成整个标定流程后，可以单击“保存”按钮存储并关闭，将摄像机的参数以 MATLAB 特有的.mat 文件进行存储，为后续测深度提供摄像头参数，如图 3-73 所示。

3.5.3　OpenCV 实现

在本示例的测量物体深度程序中，采用了 OpenCV 进行摄像机标定。因为使用的是双目摄像头，所以在读入时需要先做简单的图像分割，之后再进行 SIFT 特征提取匹配生成视差图，最后测得物体深度。

图 3-70　MATLAB 标定第四步

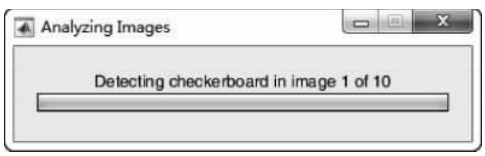

图 3-71　MATLAB 标定第五步

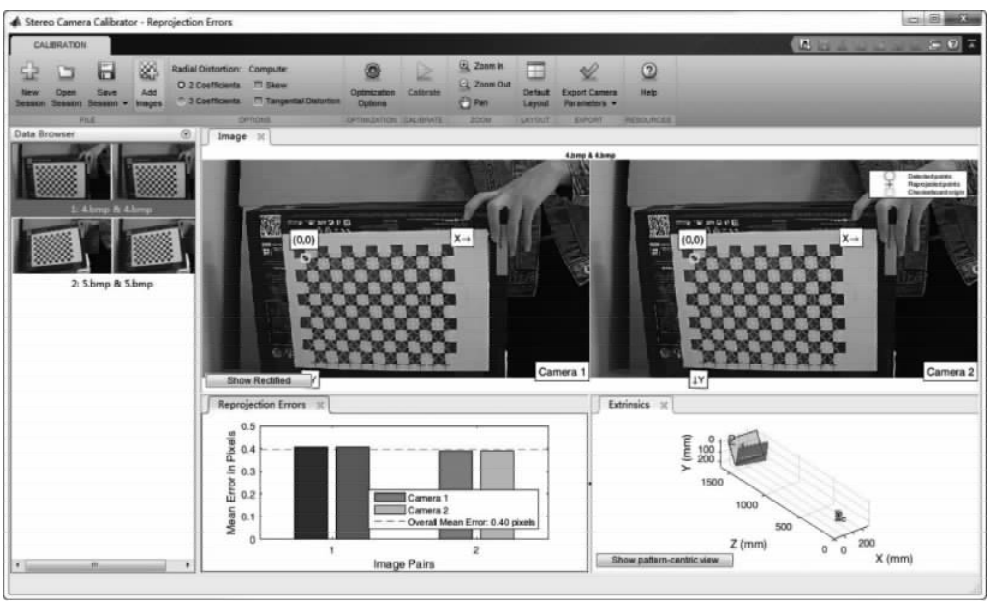

图 3-72　实际标定结果

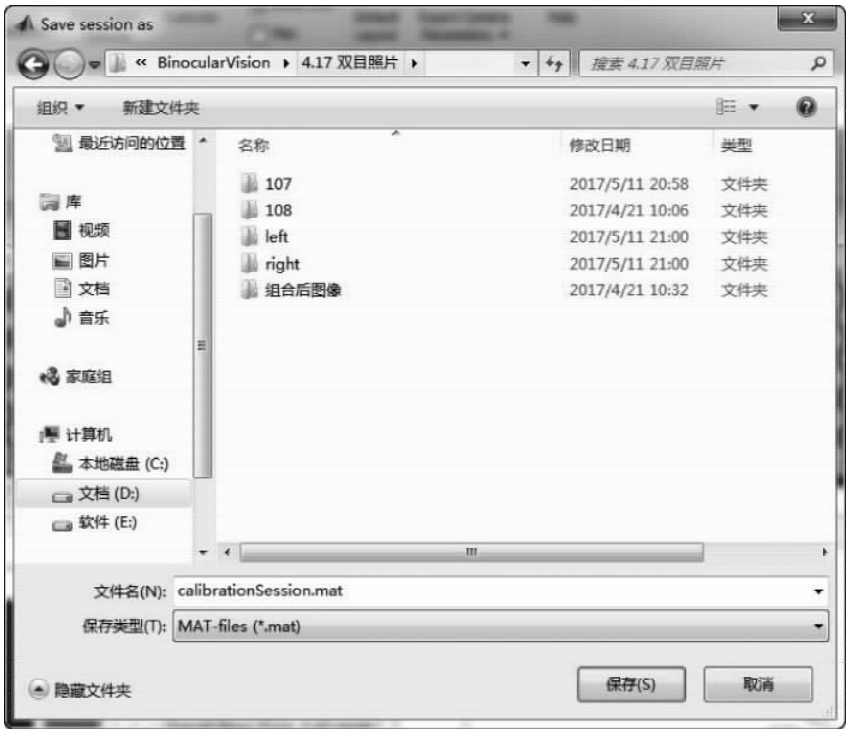

图 3-73　存储为.mat 文件

1. 程序读入的图像

(1) 程序运行前需要输入多组标定图像和实际场景的图像,如图 3-74 所示为其中一组标定图像。

图 3-74　OpenCV 标定

(2) 如图 3-75 所示为需要测定深度的图像。

图 3-75　被测图像

2. 程序实现

```
#include <opencv2/opencv.hpp>
#include <opencv2\imgproc\imgproc.hpp>
#include <opencv2\core\core.hpp>
#include <opencv2\highgui\highgui.hpp>
#include <opencv2\calib3d\calib3d.hpp>
#include <opencv2\features2d\features2d.hpp>
#include <opencv2\legacy\legacy.hpp>
#include <iostream>
#include <fstream>
#include <vector>
#include <list>
#include <algorithm>
#include <iterator>
#include <cstdio>
#include <string>

using namespace cv;
using namespace std;

typedef unsigned int uint;

Size imgSize(1280, 720);
Size patSize(12,9);                          //内半圈的脚点个数 12 * 9
const double patLen = 18.0f;                 //unit: mm 标定板每个格的宽度(金属标定板)
double imgScale = 1.0;                       //图像缩放的比例因子


//将要读取的图片路径存储在 fileList 中
vector<string> fileList;
void initFileList(string dir, int first, int last){
    fileList.clear();
    for(int cur = first; cur <= last; cur++){
        string str_file = dir + "/" + to_string(cur) + ".jpg";
        fileList.push_back(str_file);
    }
}


//生成点云坐标后保存
static void saveXYZ(string filename, const Mat& mat)
{
     const double max_z = 1.0e4;
     ofstream fp(filename);
    if (!fp.is_open())
     {
          std::cout<<"打开点云文件失败"<< endl;
fp.close();
        return;
     }
```

```
    //遍历写入
for(int y = 0; y < mat.rows; y++)
     {
for(int x = 0; x < mat.cols; x++)
          {
              Vec3f point = mat.at<Vec3f>(y, x);     //三通道浮点型
              if(fabs(point[2] - max_z) < FLT_EPSILON || fabs(point[2]) > max_z)
                continue;
              fp << point[0]<<" "<< point[1]<<" "<< point[2]<< endl;
          }
     }
    fp.close();
}


//存储视差数据
void saveDisp(const string filename, const Mat& mat)
{
    ofstream fp(filename, ios::out);
    fp << mat.rows << endl;
    fp << mat.cols << endl;
    for(int y = 0; y < mat.rows; y++)
    {
        for(int x = 0; x < mat.cols; x++)
        {
            double disp = mat.at<short>(y, x);      //这里视差矩阵是 CV_16S 格式的,故用
                                                      short 类型读取
            fp << disp << endl;                     //若视差矩阵是 CV_32F 格式,则用 float
                                                      类型读取
        }
    }
    fp.close();
}


void F_Gray2Color(Mat gray_mat, Mat& color_mat)
{
    color_mat = Mat::zeros(gray_mat.size(), CV_8UC3);
    int rows = color_mat.rows, cols = color_mat.cols;

    Mat red = Mat(gray_mat.rows, gray_mat.cols, CV_8U);
    Mat green = Mat(gray_mat.rows, gray_mat.cols, CV_8U);
    Mat blue = Mat(gray_mat.rows, gray_mat.cols, CV_8U);
    Mat mask = Mat(gray_mat.rows, gray_mat.cols, CV_8U);

    subtract(gray_mat, Scalar(255), blue);   //blue(I) = 255 - gray(I)
    red = gray_mat.clone();                  //red(I) = gray(I)
    green = gray_mat.clone();                //green(I) = gray(I),if gray(I) < 128

    compare(green, 128, mask, CMP_GE);       //green(I) = 255 - gray(I), if gray(I) >= 128
    subtract(green, Scalar(255), green, mask);
```

```
    convertScaleAbs(green, green, 2.0, 2.0);

    vector<Mat> vec;
    vec.push_back(red);
    vec.push_back(green);
    vec.push_back(blue);
    cv::merge(vec, color_mat);
}

Mat F_mergeImg(Mat img1, Mat disp8){
    Mat color_mat = Mat::zeros(img1.size(), CV_8UC3);

    Mat red = img1.clone();
    Mat green = disp8.clone();
    Mat blue = Mat::zeros(img1.size(), CV_8UC1);

    vector<Mat> vec;
    vec.push_back(red);
    vec.push_back(blue);
    vec.push_back(green);
    cv::merge(vec, color_mat);

    return color_mat;
}


//双目立体标定
int stereoCalibrate(string intrinsic_filename = "intrinsics.yml", string extrinsic_filename =
"extrinsics.yml")
{
    vector<int> idx;
    //左侧相机的角点坐标和右侧相机的角点坐标

    vector<vector<Point2f>> imagePoints[2];

    //vector<vector<Point2f>> leftPtsList(fileList.size());
    //vector<vector<Point2f>> rightPtsList(fileList.size());

    for(uint i = 0; i<fileList.size();++i)
    {
        vector<Point2f> leftPts, rightPts;      //存储左、右相机的角点位置
        Mat rawImg = imread(fileList[i]);       //原始图像
        if(rawImg.empty()){
            std::cout<<"the Image is empty..."<<fileList[i]<<endl;
            continue;
        }
        //截取左右图片
        Rect leftRect(0, 0, imgSize.width, imgSize.height);
        Rect rightRect(imgSize.width, 0, imgSize.width, imgSize.height);

        Mat leftRawImg = rawImg(leftRect);      //切分得到的左原始图像
```

```
        Mat rightRawImg = rawImg(rightRect);    //切分得到的右原始图像

        imwrite("left.jpg", leftRawImg);
        imwrite("right.jpg", rightRawImg);
        //std::cout <<"左侧图像:宽度"<< leftRawImg.size().width <<" 高度"<< rightRawImg.
size().height << endl;
        //std::cout <<"右侧图像:宽度"<< rightRawImg.size().width <<" 高度"<< rightRawImg.
size().height << endl;

        Mat leftImg, rightImg, leftSimg, rightSimg, leftCimg, rightCimg, leftMask, rightMask;
        //BGT -> GRAY
        if(leftRawImg.type() == CV_8UC3)
            cvtColor(leftRawImg, leftImg, CV_BGR2GRAY);  //转为灰度图
         else
           leftImg = leftRawImg.clone();
        if(rightRawImg.type() == CV_8UC3)
            cvtColor(rightRawImg, rightImg, CV_BGR2GRAY);
        else
            rightImg = rightRawImg.clone();

        imgSize = leftImg.size();

        //图像滤波预处理
        resize(leftImg, leftMask, Size(200, 200));        //resize 对原图像 img 重新调整大小
                                                            生成 mask 图像大小为 200 * 200
        resize(rightImg, rightMask, Size(200, 200));
GaussianBlur(leftMask, leftMask, Size(13, 13), 7);
        GaussianBlur(rightMask, rightMask, Size(13, 13), 7);
        resize(leftMask, leftMask, imgSize);
        resize(rightMask, rightMask, imgSize);
          medianBlur(leftMask, leftMask, 9);              //中值滤波
        medianBlur(rightMask, rightMask, 9);

          for (int v = 0; v < imgSize.height; v++) {
              for (int u = 0; u < imgSize.width; u++) {
                  int leftX = ((int)leftImg.at<uchar>(v, u) - (int)leftMask.at
<uchar>(v, u)) * 2 + 128;
                int rightX = ((int)rightImg.at<uchar>(v, u) - (int)rightMask.at
<uchar>(v, u)) * 2 + 128;
                leftImg.at<uchar>(v, u) = max(min(leftX, 255), 0);
                rightImg.at<uchar>(v, u) = max(min(rightX, 255), 0);
              }
          }

        //寻找角点,图像缩放
        resize(leftImg, leftSimg, Size(), imgScale, imgScale);
                                          //图像以 0.5 的比例缩放
        resize(rightImg, rightSimg, Size(), imgScale, imgScale);
          cvtColor(leftSimg, leftCimg, CV_GRAY2BGR);
                                          //转为 BGR 图像,cimg 和 simg 是 800×600 的图像
        cvtColor(rightSimg, rightCimg, CV_GRAY2BGR);
```

```
        //寻找棋盘角点
         bool leftFound = findChessboardCorners(leftCimg, patSize, leftPts, CV_CALIB_CB_
ADAPTIVE_THRESH|CV_CALIB_CB_FILTER_QUADS);
        bool rightFound = findChessboardCorners(rightCimg, patSize, rightPts, CV_CALIB_CB_
ADAPTIVE_THRESH|CV_CALIB_CB_FILTER_QUADS);

        if(leftFound)
            cornerSubPix(leftSimg, leftPts, Size(11, 11), Size(-1,-1 ),
                TermCriteria(CV_TERMCRIT_ITER + CV_TERMCRIT_EPS, 300, 0.01));
        if(rightFound)
            cornerSubPix(rightSimg, rightPts, Size(11, 11), Size(-1, -1),
                TermCriteria(CV_TERMCRIT_ITER + CV_TERMCRIT_EPS, 300, 0.01));
                                                    //亚像素

        //放大为原来的尺度
        for(uint j = 0;j < leftPts.size();j++)      //该幅图像共 132 个角点,坐标乘以 2,
                                                      还原角点位置
            leftPts[j] *= 1./imgScale;
        for(uint j = 0;j < rightPts.size();j++)
            rightPts[j] *= 1./imgScale;

        //显示
        string leftWindowName = "Left Corner Pic", rightWindowName = "Right Corner Pic";

        Mat leftPtsTmp = Mat(leftPts) * imgScale;   //再次乘以 imgScale
        Mat rightPtsTmp = Mat(rightPts) * imgScale;

        drawChessboardCorners(leftCimg, patSize, leftPtsTmp, leftFound);
                                                    //绘制角点坐标并显示
        imshow(leftWindowName, leftCimg);
        imwrite("输出/DrawChessBoard/" + to_string(i) + "_left.jpg", leftCimg);
        waitKey(200);

        drawChessboardCorners(rightCimg, patSize, rightPtsTmp, rightFound);
                                                    //绘制角点坐标并显示
        imshow(rightWindowName, rightCimg);
        imwrite("输出/DrawChessBoard/" + to_string(i) + "_right.jpg", rightCimg);
        waitKey(200);

        cv::destroyAllWindows();

        //保存角点坐标
        if(leftFound && rightFound)
        {
            imagePoints[0].push_back(leftPts);
            imagePoints[1].push_back(rightPts);     //保存角点坐标
            std::cout <<"图片 "<< i <<" 处理成功!"<< endl;
            idx.push_back(i);
        }
    }
    cv::destroyAllWindows();
```

```
    imagePoints[0].resize(idx.size());
    imagePoints[1].resize(idx.size());
    std::cout<<"成功标定的标定板个数为"<< idx.size()<<" 序号分别为: ";
    for(unsigned int i = 0;i < idx.size();++i)
        std::cout << idx[i]<<" ";

    //生成物点坐标
    vector < vector < Point3f >> objPts(idx.size());        //idx.size 代表成功检测的图像的个数
    for (int y = 0; y < patSize.height; y++) {
        for (int x = 0; x < patSize.width; x++) {
            objPts[0].push_back(Point3f((float)x, (float)y, 0) * patLen);
        }
    }
    for (uint i = 1; i < objPts.size(); i++) {
        objPts[i] = objPts[0];
    }

    //双目立体标定
    Mat cameraMatrix[2], distCoeffs[2];
    vector < Mat > rvecs[2], tvecs[2];
cameraMatrix[0] = Mat::eye(3, 3, CV_64F);
cameraMatrix[1] = Mat::eye(3, 3, CV_64F);
    Mat R, T, E, F;

    cv::calibrateCamera(objPts, imagePoints[0], imgSize, cameraMatrix[0],
distCoeffs[0], rvecs[0], tvecs[0], CV_CALIB_FIX_K3);

    cv::calibrateCamera(objPts, imagePoints[1], imgSize, cameraMatrix[1],
distCoeffs[1], rvecs[1], tvecs[1], CV_CALIB_FIX_K3);

    std::cout << endl <<"Left Camera Matrix: "<< endl << cameraMatrix[0]<< endl;
    std::cout << endl <<"Right Camera Matrix:"<< endl << cameraMatrix[1]<< endl;
    std::cout << endl <<"Left Camera DistCoeffs: "<< endl << distCoeffs[0]<< endl;
    std::cout << endl <<"Right Camera DistCoeffs: "<< endl << distCoeffs[1]<< endl;

    double rms = stereoCalibrate(objPts, imagePoints[0], imagePoints[1],
cameraMatrix[0], distCoeffs[0],
cameraMatrix[1], distCoeffs[1],
                        imgSize, R, T, E, F,
TermCriteria(CV_TERMCRIT_ITER + CV_TERMCRIT_EPS, 100, 1e - 5));
                        //CV_CALIB_USE_INTRINSIC_GUESS);


    std::cout << endl << endl <<"立体标定完成! "<< endl <<"done with RMS error = " << rms << endl;
//反向投影误差
    std::cout << endl <<"Left Camera Matrix: "<< endl << cameraMatrix[0]<< endl;
    std::cout << endl <<"Right Camera Matrix:"<< endl << cameraMatrix[1]<< endl;
    std::cout << endl <<"Left Camera DistCoeffs: "<< endl << distCoeffs[0]<< endl;
    std::cout << endl <<"Right Camera DistCoeffs: "<< endl << distCoeffs[1]<< endl;
```

```
    //标定精度检测
    //通过检查图像上点与另一幅图像的极线的距离来评价标定的精度.为了实现这个目的,使用
undistortPoints 来对原始点做去畸变的处理
    //随后使用 computeCorrespondEpilines 来计算极线,计算点和线的点积.累计的绝对误差形成
了误差
    std::cout << endl <<" 极线计算... 误差计算... ";
    double err = 0;
     int npoints = 0;
     vector < Vec3f > lines[2];
for(unsigned int i = 0; i < idx.size(); i++)
     {
         int npt = (int)imagePoints[0][i].size();                    //角点个数
         Mat imgpt[2];
for(int k = 0; k < 2; k++)
         {
             imgpt[k] = Mat(imagePoints[k][i]);
              undistortPoints(imgpt[k], imgpt[k], cameraMatrix[k], distCoeffs[k], Mat(),
cameraMatrix[k]);                                                    //畸变
             computeCorrespondEpilines(imgpt[k], k + 1, F, lines[k]);  //计算极线
         }
for(int j = 0; j < npt; j++)
         {
             double errij = fabs(imagePoints[0][i][j].x * lines[1][j][0] +
                                 imagePoints[0][i][j].y * lines[1][j][1] + lines[1][j][2]) +
                                 fabs(imagePoints[1][i][j].x * lines[0][j][0] +
                                 imagePoints[1][i][j].y * lines[0][j][1] + lines[0][j][2]);
             err += errij;                                           //累计误差
         }
         npoints += npt;
     }
     std::cout << " 平均误差 average reprojection err = " << err/npoints << endl;
                                                                     //平均误差

     //相机内参数和畸变系数写入文件
     FileStorage fs(intrinsic_filename, CV_STORAGE_WRITE);
     if(fs.isOpened())
     {
         fs << "M1" << cameraMatrix[0] << "D1" << distCoeffs[0] <<
             "M2" << cameraMatrix[1] << "D2" << distCoeffs[1];
fs.release();
     }
     else
         std::cout << "Error: can not save the intrinsic parameters\n";

    //立体矫正 BOUGUET'S METHOD
     Mat R1, R2, P1, P2, Q;
     Rect validRoi[2];

cv::stereoRectify(cameraMatrix[0], distCoeffs[0],
cameraMatrix[1], distCoeffs[1],
                    imgSize, R, T, R1, R2, P1, P2, Q,
```

```
                    CALIB_ZERO_DISPARITY, 1, imgSize, &validRoi[0], &validRoi[1]);

fs.open(extrinsic_filename, CV_STORAGE_WRITE);
if( fs.isOpened() )
     {
          fs << "R" << R << "T" << T << "R1" << R1 << "R2" << R2 << "P1" << P1 << "P2" << P2 << "Q" << Q;
fs.release();
     }
     else
          std::cout << "Error: can not save the intrinsic parameters\n";

    std::cout <<"双目标定完成..."<< endl;
    printf("\n输入任意字母进行下一步\n");
    getchar();getchar();
    return 0;
}


//--------------------------------------------------------------
//--------------------------------------------------------------
//双目立体匹配和测量
int stereoMatch(int picNum,
                string intrinsic_filename = "intrinsics.yml",
                string extrinsic_filename = "extrinsics.yml",
                bool no_display = false,
                string point_cloud_filename = "输出/point3D.txt"
                )
{
    //获取待处理的左、右相机图像
    int color_mode = 0;
    Mat rawImg = imread(fileList[picNum], color_mode);  //待处理图像 grayScale
    if(rawImg.empty()){
        std::cout <<"In Function stereoMatch, the Image is empty..."<< endl;
        return 0;
    }
    //截取
    Rect leftRect(0, 0, imgSize.width, imgSize.height);
    Rect rightRect(imgSize.width, 0, imgSize.width, imgSize.height);
    Mat img1 = rawImg(leftRect);                          //切分得到的左原始图像
    Mat img2 = rawImg(rightRect);                         //切分得到的右原始图像
    //图像根据比例缩放
if(imgScale != 1.f){
         Mat temp1, temp2;
         int method = imgScale < 1 ? INTER_AREA : INTER_CUBIC;
resize(img1, temp1, Size(), imgScale, imgScale, method);
         img1 = temp1;
resize(img2, temp2, Size(), imgScale, imgScale, method);
         img2 = temp2;
     }
    imwrite("输出/原始左图像.jpg", img1);
    imwrite("输出/原始右图像.jpg", img2);
```

```
    Size img_size = img1.size();

    //reading intrinsic parameters
    FileStorage fs(intrinsic_filename, CV_STORAGE_READ);
    if(!fs.isOpened())
    {
        std::cout<<"Failed to open file "<<intrinsic_filename<<endl;
        return -1;
    }
    Mat M1, D1, M2, D2;                                    //左、右相机的内参数矩阵和畸变系数
    fs["M1"]>>M1;
    fs["D1"]>>D1;
    fs["M2"]>>M2;
    fs["D2"]>>D2;

    M1 *= imgScale;
    M2 *= imgScale;

    //读取双目相机的立体矫正参数
fs.open(extrinsic_filename, CV_STORAGE_READ);
    if(!fs.isOpened())
    {
std::cout<<"Failed to open file "<<extrinsic_filename<<endl;
        return -1;
    }

    //立体矫正
    Rect roi1, roi2;
    Mat Q;
    Mat R, T, R1, P1, R2, P2;
    fs["R"]>>R;
    fs["T"]>>T;

    //Alpha 取值为 -1 时,OpenCV 自动进行缩放和平移
    cv::stereoRectify( M1, D1, M2, D2, img_size, R, T, R1, R2, P1, P2, Q, CALIB_ZERO_
DISPARITY, -1, img_size, &roi1, &roi2 );

    //获取两相机的矫正映射
    Mat map11, map12, map21, map22;
    initUndistortRectifyMap(M1, D1, R1, P1, img_size, CV_16SC2, map11, map12);
initUndistortRectifyMap(M2, D2, R2, P2, img_size, CV_16SC2, map21, map22);

    //矫正原始图像
    Mat img1r, img2r;
remap(img1, img1r, map11, map12, INTER_LINEAR);
remap(img2, img2r, map21, map22, INTER_LINEAR);
    img1 = img1r;
    img2 = img2r;

    //初始化 stereoBMstate 结构体
    StereoBM bm;

    int unitDisparity = 15;                          //40
    int numberOfDisparities = unitDisparity * 16;
```

```
    bm.state->roi1 = roi1;
     bm.state->roi2 = roi2;
     bm.state->preFilterCap = 13;
     bm.state->SADWindowSize = 19;                    //窗口大小
     bm.state->minDisparity = 0;                      //确定匹配搜索从哪里开始,默认值是 0
     bm.state->numberOfDisparities = numberOfDisparities;
                                                      //在该数值确定的视差范围内进行搜索
     bm.state->textureThreshold = 1000; //10          //保证有足够的纹理以克服噪声
     bm.state->uniquenessRatio = 1; //10              //!!使用匹配功能模式
     bm.state->speckleWindowSize = 200; //13          //检查视差连通区域变化度的窗口大小,值
                                                        为 0 时取消 speckle 检查
     bm.state->speckleRange = 32; //32               //视差变化阈值,当窗口内视差变化大于阈
                                                        值时,该窗口内的视差清零,int 型
     bm.state->disp12MaxDiff = -1;

    //计算
    Mat disp, disp8;
     int64 t = getTickCount();
bm(img1, img2, disp);
     t = getTickCount() - t;
     printf("立体匹配耗时: %fms\n", t*1000/getTickFrequency());

    //将 16 位带符号的整型视差矩阵转换为 8 位无符号的整型矩阵
disp.convertTo(disp8, CV_8U, 255/(numberOfDisparities*16.));

    //视差图转为彩色图
    Mat vdispRGB = disp8;
    F_Gray2Color(disp8, vdispRGB);
    //将左侧矫正图像与视差图融合
    Mat merge_mat = F_mergeImg(img1, disp8);

    saveDisp("输出/视差数据.txt", disp);

    //显示
     if(!no_display){
        imshow("左侧矫正图像", img1);
        imwrite("输出/left_undistortRectify.jpg", img1);
        imshow("右侧矫正图像", img2);
        imwrite("输出/right_undistortRectify.jpg", img2);
        imshow("视差图", disp8);
        imwrite("输出/视差图.jpg", disp8);
        imshow("视差图_彩色.jpg", vdispRGB);
        imwrite("输出/视差图_彩色.jpg", vdispRGB);
        imshow("左矫正图像与视差图合并图像", merge_mat);
        imwrite("输出/左矫正图像与视差图合并图像.jpg", merge_mat);
cv::waitKey(10000);
        printf("\n输入任意数字以继续\n");
        getchar();
         std::cout<<endl;
     }
//cv::destroyAllWindows();

    //视差图转为深度图
    cout<<endl<<"计算深度映射... "<<endl;
     Mat xyz;
     reprojectImageTo3D(disp, xyz, Q, true);          //获得深度图 disp: 720*1280
    cv::destroyAllWindows();
```

```
    cout << endl <<"保存点云坐标... "<< endl;
    saveXYZ(point_cloud_filename, xyz);

     cout << endl << endl <<"结束"<< endl <<"Press any key to end... ";

    getchar();
    return 0;
}


int main()
{
    string intrinsic_filename = "intrinsics.yml";
     string extrinsic_filename = "extrinsics.yml";
    string point_cloud_filename = "输出/point3D.txt";

    /* 立体标定运行一次即可 */
//initFileList("calib_pic", 1, 11);
//stereoCalibrate(intrinsic_filename, extrinsic_filename);

    /* 立体匹配 */
    initFileList("test_pic", 1, 2);
    stereoMatch(0, intrinsic_filename, extrinsic_filename, false, point_cloud_filename);

    return 0;
}
```

3. 运行结果

运行结果如图 3-76 所示，图中有相机标定参数，也有标定过程中出现的误差，当误差过大时，建议重新拍摄图像进行标定。图 3-77 为标定过程，图 3-78 为图像的视差图。通过视差图可以得到图像的深度，而最后图像的深度数据将会存储在如图 3-79 所示的.txt 文件中，前两列是点云的坐标，第三列表示深度，即物体每个点距摄像机的距离。

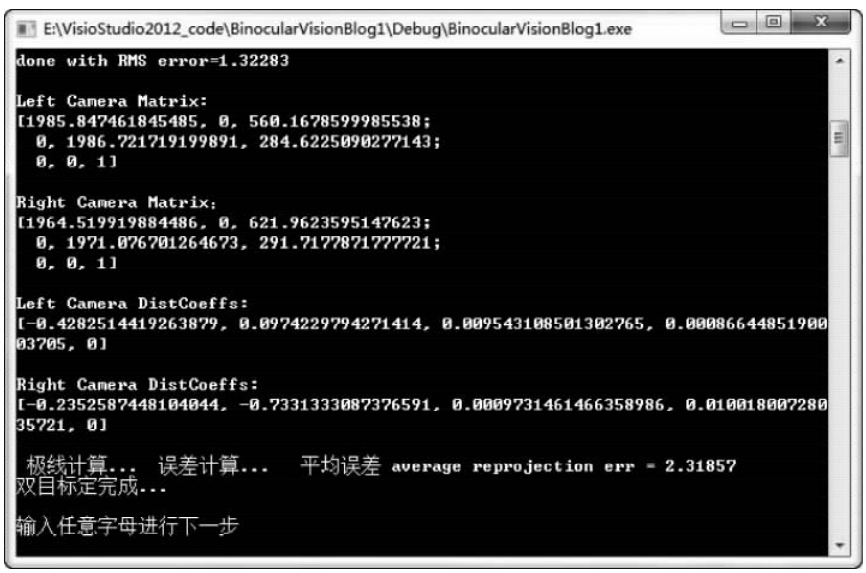

图 3-76 标定结果

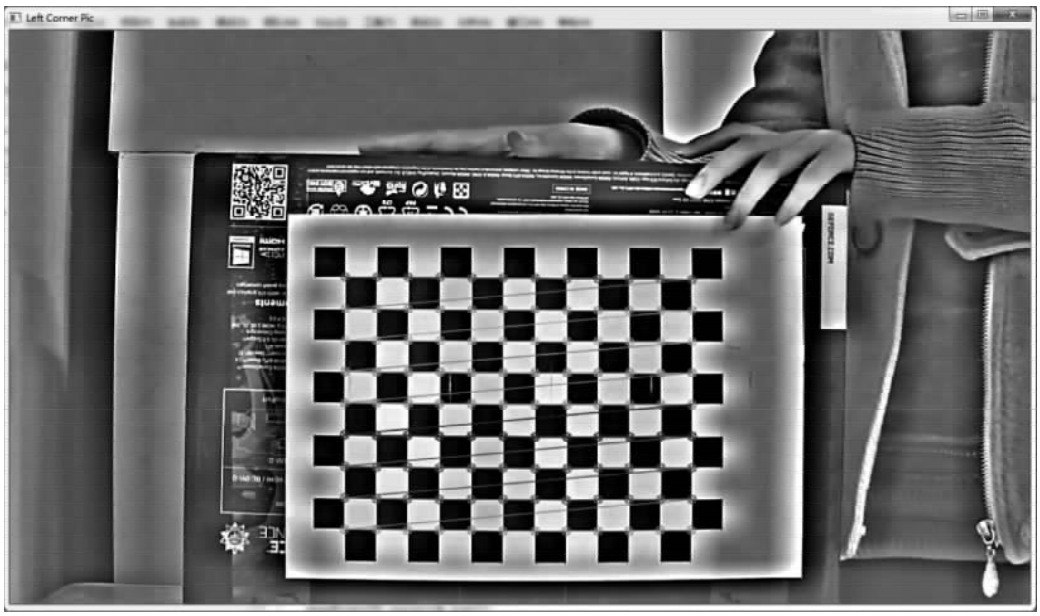

图 3-77　标定过程

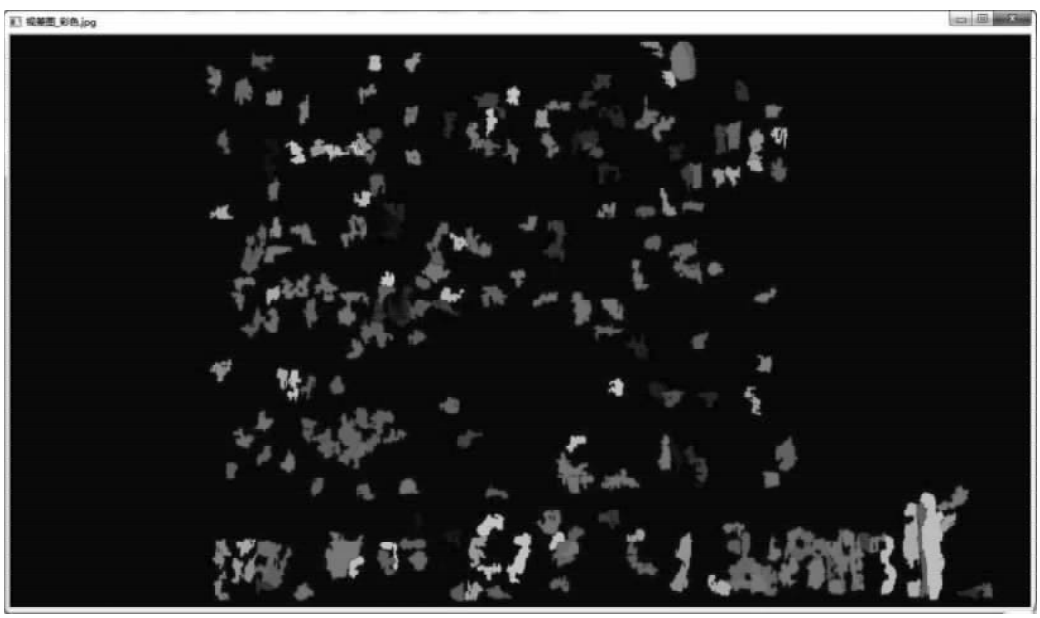

图 3-78　视差图

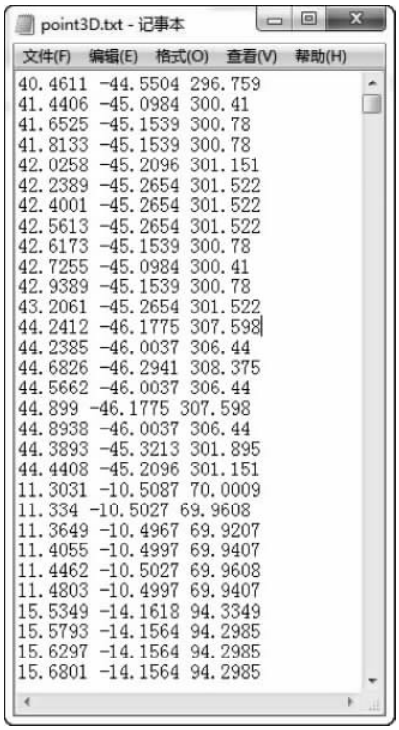

图 3-79 深度数据

3.6 本章小结

本章主要介绍了几个常用的图像处理算法并通过 OpenCV 进行了实现。3.1 节主要介绍了 OpenCV 最常用的 Mat 类和基础的输入输出函数。3.2 节介绍了反向算法，并通过反向算法对 OpenCV 常用的函数和处理方式进行了解析，为后续的学习扫清障碍。3.3 节主要介绍了 OpenCV 处理图像时常用的 addweight()函数。3.4 节介绍了三种常见的图像去噪算法，其中均值滤波和高斯滤波采用了 OpenCV 自带的函数。3.5 节主要介绍了机器视觉中比较热门的双目视觉技术，首先介绍了双目视觉的基本原理，之后介绍了如何使用 OpenCV 实现双目摄像机的标定，最后使用 OpenCV 实现了双目视觉测试物体的深度。

3.7 参考文献

[1] http://lib.csdn.net/article/opencv/24583.
[2] http://blog.csdn.net/giantchen547792075/article/details/7061391/.
[3] http://blog.sina.com.cn/s/blog_6a0e04380100r289.html.
[4] http://blog.csdn.net/qianqing13579/article/details/45318279.
[5] http://blog.csdn.net/poem_qianmo/article/details/20537737.
[6] http://www.bubuko.com/infodetail-1179988.html.
[7] http://baike.baidu.com/view/1485502.htm.

[8] 王智文,李绍滋.基于多元统计模型的分形小波自适应图像去噪[J].计算机学报,2014,37(6):1380-1389.
[9] 王科俊,熊新炎,任桢.高效均值滤波算法[J].计算机应用研究,2010,27(2):434-438.
[10] 许景波.高斯滤波器逼近理论与应用研究[D].哈尔滨:哈尔滨工业大学,2007.
[11] 王振雷.基于图像 N-邻域的协同滤波去噪方法的研究[D].西安:西安电子科技大学,2014.
[12] 郭红涛,王小伟,章勇勤.广义非局部均值算法的图像去噪[J].计算机应用研究,2015,32(7):2218-2221.
[13] 王新然.基于计算机视觉的物体体积测量系统[D].大连:大连理工大学,2012.
[14] Harwerth R S, Boltz R L. Behavioral measures of stereopsis in monkeys using random dot stereograms[J]. Physiology & Behavior, 1979, 22(2): 229-234.