CImg 库参考手册

（中文版 v.1.0.0）

英文版作者：David Tschumperle

E-mail：David.Tschumperle@greyc.ensicaen.fr

中文版作者：谢龙（云淡风轻）

E-mail：dragonxie1983@163.com

1. CImg库主页

这是CImg（C++ 图像处理模板）库的中文参考手册。本文英文版是使用doxygen工具生成的，此中文版由谢龙（云淡风轻）翻译。本文对CImg库中全部类和函数做了一个详细的描述。如果你已经下载了CImg包，你已经在目录CImg/documentation/reference/下有了这些页的本地拷贝。

可以使用上面的菜单来定位到这些文档页面。首先，你或许应该看看现有模块的列表。

为了离线阅读，同时*这里*（对应连接，待修改）还有一个这个参考文档的完整的 PDF 版本。

你或许也对用这个演示文档幻灯片（英文）来对CImg库的功能有个大体的了解感兴趣。

2. CImg库模块文档

2.1. CImg库概述

CImg 是为 C++程序员设计的一个图像处理库。它为加载/保存、显示和处理多种类型的图像，提供了有用的类和函数。

2.1.1. 库结构

CImg 库只包含了一个名为 CImg.h 的头文件。在 CImg.h 中提供了一组用于加载/保存，处理和显示图像或图像列表的 C++模板类。CImg 库有非常好的可移植性（Unix/X11、Windows、MacOS X、FreeBSD,……），同时 CImg 库非常的高效和易于使用。CImg 库是一个令人满意的在 C++中完成图像处理工作的工具箱。

头文件 CImg.h 包含了构成 CImg 库的全部类和函数。这是 CImg 库的一个新颖之处。这尤其意味着：

- 不需要对库的提前编译。因为，CImg 函数是和你自己的 C++代码同时编译的。
- 不用处理复杂的依赖关系：只要包含文件 CImg.h，你就可以使用 C++图像处理工具箱。
- 编译是按需完成的（done on the fly）：只要你程序中用到的 CImg 函数才被编译，并出现在编译后的可执行文件中。这样可以产生没有任何无用东西的，非常紧凑的代码。
- 类成员和函数都是内联的。这样程序在执行的时候会有更好的表现。

CImg 库的结构如下：

- 库中全部的类和函数都被定义在 cimg_library 名字空间（对应页，待修改）下。这个名字空间封装了这个库的全部功能，并且避免了在包含其他头文件时，可能发生的名字冲突问题。一般来说，可以把这个名字空间作为一个缺省名字空间来使用：

```
#include "CImg.h"
using namespace cimg_library;
...
```

- 在 **cimg_library::cimg** 名字空间（对应页，待修改）下定义了一组库所使用的*低级*函数和变量。这个名字空间下被本文档介绍的函数可以在你自己的程序中安全的使用。但是，不要使用 cimg_library::cimg 名字空间作为缺省空间，因为它包含的某些函数同 C/C++标准库中已经定义的函数同名。

- 类 **cimg_library::CImg**（对应页，待修改）<T>表示一个至多 4 维的，每个像素点的类型为 T 的图像。这也是本库的核心类。

- 类 **cimg_library::CImgList**（对应页，待修改）<T>表示 cimg_library::CImg<T>图像的列表。它可以被用于保存一个图像序列的不同帧。

- 类**cimg_library::CImgDisplay**（对应页，待修改）能够在图像显示窗口中显示图像或图像列表。你或许已经猜到了，这个类的代码是高度系统相关的。但是由于环境变量是被CImg库自动设置的（参见环境变量设置（P7）），这对程序员也是透明的了。

- 类 **cimg_library::CImgStats**（对应页，待修改）表示图像的统计资料。使用它来计算图像像素值中的最小值、最大值和方差，以及最小/最大像素点的位置。

- 类 **cimg_library::CImgException**（对应页，待修改）(和它的子类)被本库在发生错误时，用来抛出异常。这些异常能被块 try {…} catch (CImgException) {…}捕获。子类者明确定义了不同错误的类型。

知道了这五个类，就已经足够去享受 CImg 库的功能所带来的好处了。

2.1.2. CImg版的“Hello world”

下面这段非常简单代码是用来创建一幅“Hello World”图像的。这也向你展示了一个基本的 CImg 程序看起来是什么样子的。

```
#include "CImg.h"
using namespace cimg_library;
int main()
{
  // 定义一个每个颜色 8 位(bit)的 640x400 的彩色图像
   CImg<unsigned char> img(640,400,1,3);
  //将像素值设为 0（黑色）
   img.fill(0);
  // 定义一个紫色
   unsigned char purple[] = { 255,0,255 };

  // 在坐标(100, 100)处画一个紫色的"Hello world"
   img.draw_text("Hello World",100,100,purple);
  // 在一个标题为"My first CImg code"的窗口中显示这幅图像
   img.display("My first CImg code");

  return 0;
}
```

也可以用一种更紧凑的方式写：

```
#include "CImg.h"
using namespace cimg_library;
int main() {
  const unsigned char purple[] = { 255,0,255 };
  CImg<unsigned char>(640,400,1,3,0).draw_text("Hello World",100,100,purple).display("My first CImg code");

  return 0;
}
```

一般来说，你可以用非常小的代码来完成复杂的图像处理任务。CImg 库非常易于使用，并提供了很多图像操作方面很有意思的算法。

2.1.3. 如何编译

CImg 库是一个非常轻量级而且很用户友好的库：只使用了标准系统库。从而，避免了去处理复杂的依赖关系和由库的兼容性带来的问题。你所需要的唯一的一件事就是一个（现

代的）C++编译器：

- **Microsoft Visual C++ 6.0、Visual Studio.net 和 Visual Express Edition：**使用 CImg 库包中的工程文件和解决方案文件（‘compilation/’目录下）来看它是如何工作的。
- **Intel ICL compiler：**用下面的命令来使用 ICL 编译一个基于 CImg 的程序：

```
icl /Ox hello_world.cpp user32.lib gdi32.lib
```

- **g++（MingW windows 版）：**在 Windows 下，使用下面的命令用 g++编译一个基于 CImg 的程序：

```
g++ -o hello_word.exe hello_word.cpp -O2 -lgdi32
```

- **g++（Linux 版）：**在 Linux 下，使用下面的命令用 g++编译一个基于 CImg 的程序：

```
g++ -o hello_word.exe hello_world.cpp -O2 -L/usr/X11R6/lib -lm -lpthread -lX11
```

- **g++（Solaris 版）：**在 Solaris 下，使用下面的命令用 g++编译一个基于 CImg 的程序：

```
g++ -o hello_word.exe hello_world.cpp -O2 -lm -lpthread -R/usr/X11R6/lib -lrt -lnsl -lsocket
```

- **g++（Mac OS X 版）：**在 Mac OS X 下，使用下面的命令用 g++编译一个基于 CImg 的程序：

```
g++ -o hello_word.exe hello_world.cpp -O2 -lm -lpthread -L/usr/X11R6/lib -lm -lpthread -lX11
```

- **Dev-C++：**使用 CImg 库包中提供的工程文件来看它是如何工作的。

因为提供兼容性是CImg库的一个特征，所以，如果你使用其他的编译器，并遇到了问题，请E-mail作者（英文）或译者（中文）。不过，那些不遵守C++标准的编译器将不支持CImg库。

2.1.4. 下面要做什么？

如果你已经做好了了解更多的准备，并且想用CImg写更认真的程序的话，请到小节“**指南：正式开始**（- 10 -）”。

2.2. FAQ：常见问题

2.2.1. FAQ摘要

- 一般信息和可用性
  - CImg库是什么？
  - 支持什么平台？
  - CImg是如何分发的？

  - 什么样的人会对CImg感兴趣？
  - CeCILL许可的有什么特点？
  - 谁在CImg背后？
- C++相关问题
  - C++知识达到什么程度才能使用CImg？
  - 如何在自己的C++程序中使用CImg？

2.2.2. 1. 一般信息和可用性

2.2.2.1. 1.1. CImg 库是什么？

CImg 库是一个*开源的 C++图像处理工具箱*。

它包含在一个（大的）单独的头文件 CImg.h 中。提供一组可以在你自己的源代码中使用的 C++类和函数。这些类和函数提供了加载/保存，处理和显示图像的功能。它实际上是一个非常简单而易于使用的，用于在 C++中完成图像处理任务的工具箱：仅需要包含头文件 *CImg.h*，你就可以在你的 C++程序中处理图像。

2.2.2.2. 1.2. 支持什么平台？

可移植性在 CImg 的设计中一直被关注。CImg 已经在不同的体系结构和编译器上彻底的测试过了，并且可以工作在任何带有一个近期 C++编译器的正式操作系统上工作。在每次发布之前，CImg 库都在下面这些的不同的配置下编译过：

- 带有 g++的 32 位 Linux PC 机。
- 带有 Visual C++ 6.0 的 32 位 Windows PC。
- 带有 Visual C++ Express Edition 的 32 位 Windows PC。
- 带有 g++的 32 位 Solaris Sun SPARC。
- 带有 g++的 OS X Mac PPC。

CImg 有极少数的依赖。在它的最小版本中，它可以只使用 C++标准头文件编译。同时，它还拥有有趣的扩展能力，可以使用外部库来更高效的完成一些特殊任务（使用 **FFTW** 来完成傅立叶变换（Fourier Transform））。

2.2.2.3. 1.3. CImg 是如何分发的？

CImg 库是以一个完整.zip 包的形式在 *Sourceforge servers* 上自由分发的。

这个包是在 CeCILL 许可下分发的。

这个包包括：

- 库的主文件 CImg.h（C++头文件）。
- 一些展示*如何使用 CImg* 的 C++源文件。
- 一份库的完整文档，HTML 格式和 PDF 格式。
- 附加的库插件，这些插件可以为特殊应用扩展库的能力。

CImg 库是一个相当轻量级的库，这样它就容易被控制（得益于它独特的结构），并且拥有一个快速的发布周期。差不多每三个月就会有 CImg 包的新版本被发布。

2.2.2.4. 1.4. 什么样的人会对 CImg 感兴趣？

CImg 库是一个图像处理库，主要是为工作在图像处理或计算机视觉领域，并有一些 C++基础的计算机科学家或学生设计的。由于这个库的便捷和易于使用，它也适用于任何临时需要在 C++中处理图像工具的程序员，因为 C++中还没有这个用于这个目的的标准库。

2.2.2.5. 1.5. CeCILL 许可的有什么特点？

CeCILL 许可管理着对 CImg 库的使用。这是一个*开源*协议给了你在特定条件下获取、使用、修改和重新分发源代码的权利。在 CImg 中使用了 CeCILL 许可的两个不同变体（叫做 CeCILL 和 CeCILL-C，都是开源的），对应于对源文件的不同约束：

- CeCILL-C 许可接近于 *GUN LGPL 许可*（对应连接，待修改），是最宽松的一个许可。仅适用于*库的主文件* CImg.h。主要的，这个许可允许在一个闭源的产品中使用 CImg.h，而不强迫你重新分发整个软件的源代码。但是，如果某人修改了 CImg.h 源文件，则必须在相同的 CeCILL-C 许可下重新分发修改后的版本。
- CeCILL 许可接近于（甚至完全兼容）*GNU GPL 许可*（对应连接，待修改），适用于 CImg 库包中的所有其他文件（例子源代码，插件和文档）。它不允许在闭源产品中使用这些文件。

我们建议你在发布一个基于 CImg 库的软件前阅读 CeCILL-C 和 CeCILL 许可的完整描述。

2.2.2.6. 1.6. 谁在 CImg 背后？

CImg 始于 1999 年 10 月的 David Tschumperl 的博士论文的开始。他仍然是这个项目的主要协调者。自从在 Sourceforge 上第一次发布以来，出现了越来越多的贡献者。得益于这个库简单而紧凑的形式，提交一个贡献非常容易，并能快速的集成到提供的发布中。

2.2.3. 2. C++相关问题

2.2.3.1. 2.1. C++知识达到什么程度才能使用 CImg？

CImg 库被设计为使用 C++模板和面向对象编程技术，但在一个非常易于了解的水平上。

在 CImg 库中只有没有任何继承的公共类（就像 C 的结构体）和每个 CImg 类至多有一个模板参数（用来定义图像像素的类型）。CImg 的设计简单而清晰，使得这个库对非专业 C++程序员也是易于理解的，同时又为 C++专家提供了强大的扩展能力。

2.2.3.2. 2.2. 如何在自己的 C++程序中使用 CImg？

基本上，为了能够使用 CImg 图像，你只需要在你的 C++源代码中加入这两行就可以了：

```
#include "CImg.h"
using namespace cimg_library;
```

2.3. 环境变量设置

CImg 库是一个多平台库，工作在大量不同的系统上。这就暗示着存在一些必须被正确定义的*环境变量*，它们依赖于你的当前系统。大部分时候，CImg 库自动定义这些变量（对于流行的系统）。总之，如果你的系统没有被识别，你就必须手动设置这些环境变量。这有一个对这些环境变量的快速说明。

使用 define 关键字来设置这些环境变量。这些设置必须在你的源代码中包含文件 CImg.h 之前完成。例如，定义环境变量 cimg_display_type 将和这个类似：

```
#define cimg_display_type 0
#include "CImg.h"
...
```

这是被 CImg 库所使用的不同的环境变量：

- cimg_OS：这个变量定义了你的操作系统的类型。它可以被是设置为 1（*Unix*）、2（*Windows*），或者 0（*其他配置*）。实际上它应该被 CImg 库自动决定。如果不是（cimg_OS = 0）的情况，你将可能要调剂下面描述的环境变量。
- cimg_display_type：这个变量定义了用于在窗口上显示图像的图形库的类型。它可以被设置为 0（没有有效的显示库）、1（基于 X11 显示）或者 2（Windows-GDI 显示）。如果你正在一个即没有 X11 又没有 Windows-GDI 能力的系统上运行时。请把这个变量设为 0.这将关闭对显示的支持，因为 CImg 库没有包含在 X11 或 Windows GDI 之外的其他系统上显示图像的必要代码。
- cimg_color_terminal:这个参数告诉库，系统终端是否拥有 VT100 能力。它可以*被定义*或*不被定义*。当使用 CImg 库时，定义这个变量来在你的终端上得到彩色输出。
- cimg_debug：这个变量定义了在运行期将被 CImg 库显示的调试信息的级别。它可以被设置为 0（无调试信息）、1（在标准错误提示上显示普通调试信息）、2（在模式窗口中显示调试信息，这也是缺省值）、3（高级调试信息）。将这个值设置为 3 会减慢你的程序，这是因为库要做更多的调试测试（尤其是检查对像素的访问是否超出了图像的范围）。参见 CImgException（对应连接，待修改）来更好的理解调试信息是如何工作的。

- cimg_graphicsmagick_path：这个变量告诉 CImg，**Graphicsmagick** 的 *gm* 工具在什么地方。如果 Graphicsmagick 安装在标准目录中或 *gm* 在你系统的 PATH 变量中，就没有必要设置这个变量。这个宏应当仅在 Graphicsmagick 的 *gm* 工作没有呗自动找到，而且还尝试加载/保存压缩图像格式时被设置。参见 cimg_library::CImg::get_load_convert()（对应连接，待修改）和 cimg_library::CImg::save_convert()（对应连接，待修改）获取更多信息。
- cimg_imagemagick_path：这个变量告诉 CImg，**ImageMagick** 的 *convert* 工具在什么地方。如果 ImageMagick 安装在标准目录中或 *convert* 在你系统的 PATH 变量中，就没有必要设置这个变量。这个宏应当仅在 ImageMagick 的 *convert* 工具没有被自动找到，而且还尝试读取压缩的图像格式（GIF、PNG……）时被设置。参见 cimg_library::CImg::get_load_convert()（对应连接，待修改）和 cimg_library::CImg::save_convert()（对应连接，待修改）获取更多信息。
- cimg_use_png：这个变量只在你的程序想提供对 png 文件的本地支持的时候才应被定义。在此变量被定义后，你需要在链接你的代码时同 zlib/png 库进行链接。
- cimg_use_jpeg：这个变量只在你的程序想提供对 jpeg 文件的本地支持的时候才应被定义。在此变量被定义后，你需要在链接你的代码时同 jpeg 库进行链接。
- cimg_use_tiff：这个变量只在你的程序想提供对 tiff 文件的本地支持的时候才应被定义。在此变量被定义后，你需要在链接你的代码时同 tiff 库进行链接。
- cimg_use_magick：这个变量只在你的程序想提供本地的 magick 功能的时候才应被定义。在此变量被定义后，你需要在链接你的代码时同 magick++库进行链接。
- cimg_use_fftw3：这个变量只在你的程序想提供本地的 FFTW 功能的时候才应被定义。在此变量被定义后，你需要在链接你的代码时同 ffwt 库进行链接。
- cimg_use_lapack：这个变量只在你的程序想提供本地的 lapack 线性运算功能的时候才应被定义。在此变量被定义后，你需要在链接你的代码时同 lapack 库进行链接。
- cimg_temporary_path：这个变量告诉 CImg 在哪里它能找到一个存放临时文的目录。如果你在一个标准系统上运行，就没有必要设置这个参数。这个宏仅当在尝试读取压缩图像格式（GIF、PNG……）时遇到麻烦的时候被定义。参见 cimg_library::CImg::get_load_convert()（对应连接，待修改）和 cimg_library::CImg::save_convert()（对应连接，待修改）获取更多信息。
- cimg_plugin：这个变量告诉库使用插件文件向类 CImg<T>中添加特征。如果你想向类 CImg<T>中添加成员函数，你不必直接修改“CImg.h”文件，只要用你的插件文件的路径来定义它就可以了。在类 CImg<T>中完成了对插件文件的包含。如果 cimg_plugin 没有被指定（缺省），则没有文件被包含。
- cimglist_plugin：和 cimg_plugin 相同，只是是向类 CImgList<T>中添加特征。
- cimgdisplay_plugin：和 cimg_plugin 相同，只是是向类 CImgDisplay 中添加特征。
- cimgstats_plugin：和 cimg_plugin 相同，只是是向类 CImgStats 中添加特征。

所以这些变量都是可以使用函数 cimg_library::cimg::info() 检测的。

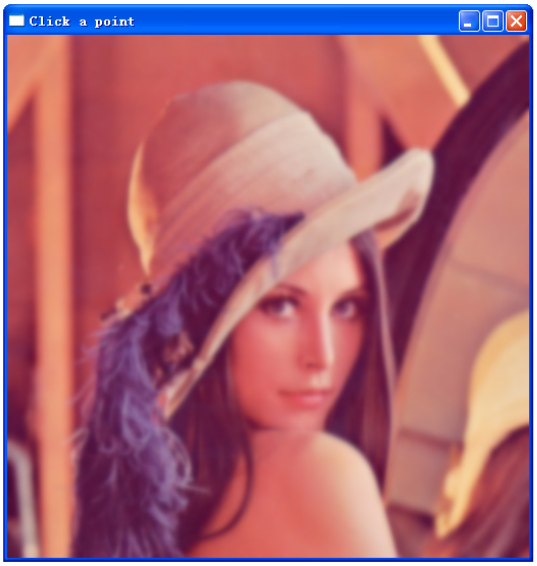
cimg_library::cimg::info()在标准错误输出上显示一个由不同配置变量和它们的值组成的列表。

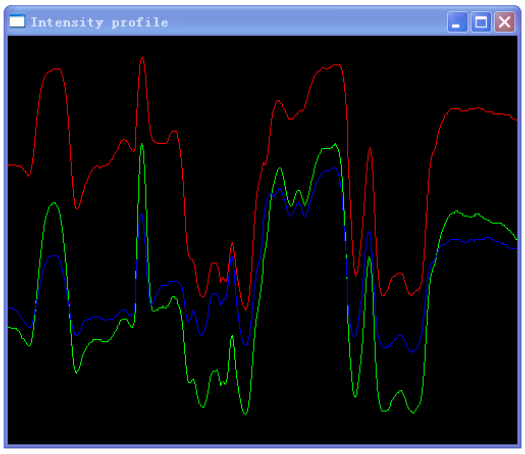
2.4. 指南：正式开始

让我们以书写我们的第一个程序来感受 CImg 的魅力。这段代码将为你展示如何加载和创建图像，以及如何处理图像显示和鼠标事件。假设我们想加载一幅名为 lena.jpg 的彩色图像，对它进行平滑处理，然后在窗口中显示它，并且进入一个点击图像中某点，就（在另一个窗口中）画出图像中相应行的（R，G，B）强度概况（intensity profile）的事件循环。是的，对于第一段代码这听起来很复杂，但是不用担心，使用 CImg 库这将变得十分简单！好的，看看下面完成这项任务的代码：

```
#define cimg_use_jpeg CIMG_USE_JPEG
#include "CImg.h"

using namespace cimg_library;

int main(int argc, char *argv[])
{
  CImg<unsigned char> image("lena.jpg"), visu(500, 400, 1, 3, 0);
  const unsigned char red[] = { 255, 0, 0 }, green[] = { 0, 255, 0 }, blue[] = { 0, 0, 255 };

  image.blur(2.5);

  CImgDisplay main_disp(image, "Click a point"), draw_disp(visu, "Intensity profile");

  while (!main_disp.is_closed && !draw_disp.is_closed) {
    main_disp.wait();

    if (main_disp.button && main_disp.mouse_y >= 0) {
      const int y = main_disp.mouse_y;

      visu.fill(0).draw_graph(image.get_crop(0, y, 0, 0, image.dimx() - 1, y, 0, 0), red, 0, 256, 0);
      visu.draw_graph(image.get_crop(0, y, 0, 1, image.dimx() - 1, y, 0, 1), green, 0, 256, 0);
      visu.draw_graph(image.get_crop(0, y, 0, 2, image.dimx() - 1, y, 0, 2), blue, 0, 256, 0).display(draw_disp);
    }
  }

  return 0;
}
```

下面是一张这个程序运行时的屏幕截图：





接下来，我们来详细解释源码的每一行：

```
#define cimg_use_jpeg CIMG_USE_JPEG
```

这行定义环境变量cimg_use_jpeg，你可以把cimg_use_jpeg定义为任何你喜欢的值。

（注：如果你使用Graphicsmagick的gm或ImageMagick的convert的话，则你不需要这行，但请注意环境变量cimg_graphicsmagick_path或cimg_imagemagick_path应被正确设置，详细信息请参见**环境变量设置**一节。）

```
#include "CImg.h"
```

引入 CImg 库的主文件同时也是唯一的头文件。

```
using namespace cimg_library;
```

使用 CImg 库的名字空间来简化以后的声明。

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

定义主函数。

```
CImg<unsigned char> image("lena.jpg"), visu(500, 400, 1, 3, 0);
```

创建两个使用 unsigned char 型像素的图像的实例。第一个图像 image 通过从磁盘读取一幅图像文件来初始化。此处，lena.jpg 必须同当前程序在同一目录中。第二个图像 visu 被初始化为一幅黑色的彩色图，它的大小是 dx = 500（长），dy = 400（宽），dz = 1（深）（这里，它是一幅 2D 图像，而不是一幅 3D 图像），并且 dv = 3（每个像素拥有 3 个‘向量’通道 R，G，B）。在构造函数中的最后一个参数定义了像素值的缺省值（这里是 0，也就是说 visu 将被初始化为黑色）。

```
const unsigned char red[] = { 255, 0, 0 }, green[] = { 0, 255, 0 }, blue[] = { 0, 0, 255 };
```

定义三种不同的颜色，每种颜色都是一个 const unsigned char 数组。它们将在后面被用来画不同颜色的曲线。

```
image.blur(2.5);
```

柔化图像。这里使用的是标准差为 2.5 的高斯模糊。注意 CImg 中的大部分函数都有两个版本：一个是就地完成（就是 blur 这种情况），而另一种则把结果作为一个新的图像返回（这些函数的名字都以 get_开始）。在样，我们也可以这样写：

```
image = image.get_blur(2.5);
```

（代价更大，因为它需要一次额外的拷贝操作）。

```
CImgDisplay main_disp(image, "Click a point"), draw_disp(visu, "Intensity profile");
```

为输入的图像 image，和将用来显示强度概况的的图像 visu 创建两个显示窗口（各一个）。缺省情况下，CImgDisplay 处理（来自鼠标、键盘，……）的事件。在 Windows 上，有一个创建全屏显示的方法。

```
while (!main_disp.is_closed && !draw_disp.is_closed) {
```

进入事件循环，但某个显示窗口被关闭的时候就将退出代码。

```
main_disp.wait();
```

等待在显示窗口 main_disp 上的某个（鼠标、键盘，……）事件发生。

```
 if (main_disp.button && main_disp.mouse_y >= 0) {
```

检测鼠标按键是否在图像区域上点击了。或许有区别 3 个不同鼠标按键的需求，但是在这里这是不需要的。

```
const int y = main_disp.mouse_y;
```

获取图像中同被点击点所在的 y 轴的那一行。

```
visu.fill(0).draw_graph(image.get_crop(0, y, 0, 0, image.dimx() - 1, y, 0, 0), red, 0, 256, 0);
```

这一行展现了大多数 CImg 类功能的管道（pipeline）特性。第一个函数 fill(0)简单的把全部像素点的值设为 0（也就是，清空图像 visu）。有趣的事是它返回 visu 的一个引用，这个引用可以被函数 draw_graph()管道化使用。draw_graph()在图像 visu 上画一条曲线。曲线的数据由另一幅图像给出（draw_graph()的第一个参数）。这个例子中，所给定的图像是原始图像第 y 行的红色分量。这个分量由，返回图像 image 的一个子图像的函数 get_crop()，所定位到。记住图像的坐标是 4D（x，y，z，v），而对于彩色图像，R，G，B 三个通道分别通过 v = 0、v = 1、v = 2 给出。

```
visu.draw_graph(image.get_crop(0, y, 0, 1, image.dimx() - 1, y, 0, 1), green, 0, 256, 0);
```

画被点击行的绿色通道的强度概况曲线。

```
visu.draw_graph(image.get_crop(0, y, 0, 2, image.dimx() - 1, y, 0, 2), blue, 0, 256, 0).display(draw_disp);
```

对蓝色通道做同样的事情。注意（返回 visu 的一个引用的）函数是如何同函数 display()管道化工作的。display()只是在相应的窗口中画出图像 visu。

```
……直到最后
```

我认为你不需要更多的说明！

就像你已经注意到的，CImg 库允许写非常短小而直观的代码。也要注意到这个源代码将在 Unix 和 Windows 系统上都能完美的工作。请也看看 CImg 包中提供的例子（目录 examples/下）。它将向你展示基于 CImg 的代码是会如何令人惊讶的短小的。另外，也有一个同你想做的东西非常相近的例子。看看文件 CImg.cpp 将会是个好的开始。CImg.cpp 中包含了短小而多样的，你可以用 CImg 库完成的功能的例子。在这个源代码中使用了 CImg 的全部类，而且可以简单的修改代码来看看将发生什么。

2.5. 使用绘图函数

2.5.1. 使用绘图函数

这一小节讲解 CImg 图像的更多绘画特性。绘图函数列表可以在 CImg 函数列表（**绘图**函数小节）中找到，并且都是在同一个基础上定义的。下面是在使用绘图函数前需要理解的重点：

- 绘图是在图像实例上完成的。绘图函数——参数被定义为 const 变量，返回的是当前实例的一个引用（*this），因此绘图函数可以支持管道化用法（见下面的例子）。绘图通常是以带有向量形式值的维和任何可能像素类型的 2D 彩色图像的形式完成的，但是也能以 3D 图像的方式完成。
- 在一幅图像的绘画特性中通常需要一个颜色参数。颜色必须被定义为一个 C 风格的数组，并且它的维数至少是 1 维的。

2.6. 使用图像循环

CImg 库提供了一组在图像上的有用的迭代循环的不同宏。基本上，它不但可以用来代替一个或多个 for(..)指令，更为经典循环提供了有趣的扩展。下面是现存的全部循环宏的一个列表，它们被归为四个不同的类：

- **像素缓冲区上的循环**（- 13 -）
- **图像维度上的循环**（- 14 -）
- **内部区域和边界上的循环**（- 15 -）
- **使用邻域的循环**（- 16 -）

2.6.1. 像素缓冲区上的循环

像素缓冲区上的循环真的是非常基础的循环了。它在一个 **cimg_library::CImg**（对应页，待修改）图像的像素数据缓冲区上迭代一个指针。为了达到这个目的定义了两个宏：

- **cimg_for(img, ptr, T)**：这个宏在图像 img 的像素数据缓冲区上，使用指针 T* ptr 进行循环，循环是从缓冲区的结尾（最后一个像素）开始到缓冲区的开始（第一个像素）结束的。

- img 必须是一个拥有 T 类型像素的（非空）**cimg_library::CImg<T>**（对应页，待修改）图像。
- ptr 是一个 T*类型的指针。这个类型的循环不应该大量出现在你自己的源代码中，因为这是一个低级的循环，并且有很多 CImg 的函数可以完成同样的工作。

这里是个使用该宏的例子：

```
CImg<float> img(320, 200);
cimg_for(img, ptr, float) { *ptr = 0; } // Equivalent to 'img.fill(0);'
```

- **cimg_foroff(img, off)**：这个宏使用一个偏移量在图像 img 的像素数据缓冲区上循环，从缓冲区的开端（第一个像素，off = 0）开始直到缓冲区的结尾（最后一个像素值，off = img.size()-1）结束。
  - img 必须是一个带有 T 类型像素的（非空的）**cimg_library::CImg<T>**图像。
  - off 是一个内循环变量，仅在循环域内部定义。

这是一个使用该宏的例子：

```
CImg<float> img(320,200);
cimg_foroff(img, off) { img[off] = 0; } // Equivalent to 'img.fill(0);'
```

2.6.2. 图像维度上的循环

下面这些循环可能是在图像处理程序中用的最多的循环。它们提供了在图像的一个或多个维度上的实现光栅扫描过程的循环。这是在单维上该类宏循环的一个列表：

- **cimg_forX(img, x)**：等价于：for (int x=0; x<img.dimx(); x++)。
- **cimg_forY(img, y)**：等价于：for (int y=0; y<img.dimy(); y++)。
- **cimg_forZ(img, z)**：等价于：for (int z=0; z<img.dimz(); z++)。
- **cimg_forV(img, v)**：等价于：for (int v=0; v<img.dimv(); v++)。

这些宏的组会也被定义成了其他的循环宏，允许直接在 2D、3D 或 4D 图像上进行循环：

- **cimg_forXY(img, x, y)**：等价于：cimg_forY(img, y) cimg_forX(img, x)。
- **cimg_forXZ(img, x, z)**：等价于：cimg_forZ(img, z) cimg_forX(img, x)。
- **cimg_forYZ(img, y, z)**：等价于：cimg_forZ(img, z) cimg_forY(img, y)。
- **cimg_forXV(img, x, v)**：等价于：cimg_forV(img, v) cimg_forX(img, x)。
- **cimg_forYV(img, y, v)**：等价于：cimg_forV(img, v) cimg_forY(img, y)。
- **cimg_forZV(img, z, v)**：等价于：cimg_forV(img, v) cimg_forZ(img, z)。
- **cimg_forXYZ(img, x, y, z)**：等价于：cimg_forZ(img, z) cimg_forXY(img, x, y)。
- **cimg_forXYV(img, x, y, v)**：等价于：cimg_forV(img, v) cimg_forXY(img, x, y)。
- **cimg_forXZV(img, x, z, v)**：等价于：cimg_forV(img, v) cimg_forXZ(img, x, z)。

- **cimg_forYZV(img, y, z, v)**：等价于：cimg_forV(img, v) cimg_forYZ(img, y, z)。
- **cimg_forXYZV(img, x, y, z, v)**：等价于：cimg_forV(img, v) cimg_forXYZ(img, x, y, z)。
  - 对于这些循环，x、y、z 和 v 都是在循环域内部定义的并只能在域内可见的变量。不需要在调用这些宏之前定义它们。
  - img 必须是一个（非空的）**cimg_library::CImg<T>**（对应页，待修改）图像。

这是使用它们创建一个带有平滑梯度颜色图像的例子：

```
CImg<unsigned char> img(256, 256, 1, 3); // Define a 256x256 color image
cimg_forXYV(img, x, y, v) { img(x, y, v) = (x + y) * (v + 1)/6; }
img.display("Color gradient");
```

2.6.3. 内部区域和边界上的循环

仅在图像的边界或内部区域（不包含边界）上也定义了相似的宏。边界可以是几个像素宽：

- **cimg_for_insideX(img, x, n)**：沿x轴循环，但不包括在宽为 n 个像素的边界内的像素点。
- **cimg_for_insideY(img, y, n)**：沿y轴循环，但不包括在宽为 n 个像素的边界内的像素点。
- **cimg_for_insideZ(img, z, n)**：沿z轴循环，但不包括在宽为 n 个像素的边界内的像素点。
- **cimg_for_insideV(img, v, n)**：沿v轴循环，但不包括在宽为 n 个像素的边界内的像素点。
- **cimg_for_insideXY(img, x, y, n)**：沿(x, y)轴循环，但不包括在宽为 n 个像素的边界内的像素点。
- **cimg_for_insideXYZ(img, x, y, z, n)**：沿(x, y, z)轴循环，但不包括在宽为 n 个像素的边界内的像素点。

同时：

- **cimg_for_borderX(img, x, n)**：沿x轴循环，但仅包括在宽为 n 像素的边界内的像素点。
- **cimg_for_borderY(img, y, n)**：沿y轴循环，但仅包括在宽为 n 像素的边界内的像素点。
- **cimg_for_borderZ(img, z, n)**：沿z轴循环，但仅包括在宽为 n 像素的边界内的像素点。
- **cimg_for_borderV(img, v, n)**：沿v轴循环，但仅包括在宽为 n 像素的边界内的像素点。
- **cimg_for_borderXY(img, x, y, n)**：沿(x, y)轴循环，但仅包括在宽为 n 像素的边界内的像素点。
- **cimg_for_borderXYZ(img, x, y, z, n)**：沿(x, y, z)轴循环，但仅包括在宽为 n 像素的边界内的像素点。
  - 对于所有这些循环，x, y, z 和 v 都是内部定义变量，仅在循环域内可见。不需要在调用这些宏之前定义他们。
  - img 必须是一个（非空的）**cimg_library::CImg<T>**对应页，待修改)图像。

- 常量 n 代表边界的尺寸。

这是一个使用这些宏创建一幅拥有两个不同亮度梯度的 2D 灰度图像的例子：

```
CImg<> img(256,256);
cimg_for_insideXY(img,x,y,50) img(x,y) = x+y;
cimg_for_borderXY(img,x,y,50) img(x,y) = x-y;
img.display();
```

2.6.4. 使用邻域的循环

在图像循环中，得到循环区域中当前像素的邻域像素值经常是有用的。CImg 库为这个目的提供了一个非常巧妙而快速的机制，该机制定义了一些记录某像素的邻域像素值的循环宏。使用这些宏可以高度优化你的代码，也可以简化你的程序。

2.6.4.1.2D 图像上的基于邻域的循环

对于 2D 图像，基于邻域的循环宏是：

- **cimg_for2x2(img, x, y, z, v, I)**：使用一个居中的 2x2 的邻域沿(x, y)轴循环。
- **cimg_for3x3(img, x, y, z, v, I)**：使用一个居中的 3x3 的邻域沿(x, y)轴循环。
- **cimg_for4x4(img, x, y, z, v, I)**：使用一个居中的 4x4 的邻域沿(x, y)轴循环。
- **cimg_for5x5(img, x, y, z, v, I)**：使用一个居中的 5x5 的邻域沿(x, y)轴循环。

对于所有这些循环，x和y都是内部定义的变量，仅在循环域内可见。不需要在调用这些宏之前定义它们。img是一个非空的**CImg<T>**图像。z和v是常量（对于 2D灰度图像通常都是 0），定义了循环必须被应用在哪个图像切片和向量通道上。最后，I是将被更新的循环中当前像素的 2x2、3x3、4x4 或 5x5 邻域。（**邻域定义见**（- 16 -））。

2.6.4.2.3D 图像上的基于邻域的循环

对于 3D 图像，基于邻域的循环宏是：

- **cimg_for2x2x2(img, x, y, z, v, I)**：使用一个居中的 2x2x2 的邻域沿(x, y, z)轴循环。
- **cimg_for3x3x3(img, x, y, z, v, I)**：使用一个居中的 3x3x3 的邻域沿(x, y, z)轴循环。

对于所有这些循环，x、y和v都是内部定义的变量，仅在循环域内可见。不需要在调用这些宏之前定义它们。img是一个非空的**CImg<T>**图像。z是常量（对于 2D灰度图像通常是 0），定义了循环必须被应用在哪个图像通道上。最后，I是将被更新的循环中当前像素的 2x2x2、3x3x3 邻域。（**邻域定义见**（- 16 -））。

2.6.4.3.邻域定义

CImg 库使用如下特殊的 CImg 宏将邻域定义为一个由命名的*变量*或*引用*组成的集合：

- **CImg_2x2(I, type)**：定义一个名为 I 类型为 type 的 2x2 邻域。
- **CImg_3x3(I, type)**：定义一个名为 I 类型为 type 的 3x3 邻域。
- **CImg_4x4(I, type)**：定义一个名为 I 类型为 type 的 4x4 邻域。
- **CImg_5x5(I, type)**：定义一个名为 I 类型为 type 的 5x5 邻域。
- **CImg_2x2x2(I, type)**：定义一个名为 I 类型为 type 的 2x2x2 邻域。
- **CImg_3x3x3(I, type)**：定义一个名为 I 类型为 type 的 3x3x3 邻域。

实际上，I 只是邻域的一个*泛称*（*generic name*）。事实上，这些宏声明了一组新变量。例如，定义一个 3x3 的邻域 CImg_3x3(I, float)就声明了 9 个不同的 float 型变量：Ipp、Icp、

Inp、Ipc、Icc、Inc、Ipn、Icn 和 Inn，它们对应于 3x3 邻域中的每个像素值。变量索引 p、c 和 n 分别代表“*前一个(**p**revious)*”、“*中(**c**urrent)*”和“*后一个(**n**ext)*”。第一个索引表示x轴，第二个索引表示y轴。因此，变量名同其对应的像素点在邻域中的位置直接相关。对于 3D 邻域，第三个索引表示z轴。这样，你在邻域循环内部有如下等式：

- Ipp = img(x-1, y-1)
- Icn = img(x, y+1)
- Inp = img(x+1, y-1)
- Inpc = img(x+1, y-1, z)
- Ippn = img(x-1, y-1, z+1)
- 等等……

对于更大的邻域，如 4x4 或 5x5 的邻域，我们引入了两个附加索引：a（表示“*后(**a**fter)*”）和 b（表示“*前(**b**efore)*”）,因此：

- Ibb = img(x-2, y-2)
- Ina = img(x+1, y+2)
- 等等……

邻域中的像素如果在图像范围之外（图像边境问题），它的值被自动设置成于它最近的有效像素的值（这也被称为*诺依曼边界条件(Neumann border condition)*）。

作为引用的邻域

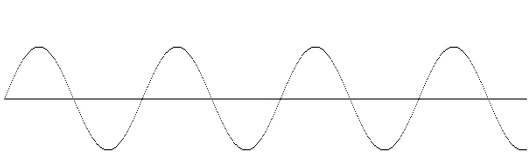
我们同样可以将邻域变量定义为经典 C 数组或 CImg<T>图像的引用，而不是分配新的变量。这是通过在用做邻域定义的宏名字后附加上_ref 来实现的：

- **CImg_2x2_ref(I, type, tab)**：定义一个名为 I，类型为 type 的 2x2 邻域作为 tab 的一个引用。
- **CImg_3x3_ref(I, type, tab)**：定义一个名为 I，类型为 type 的 3x3 邻域作为 tab 的一个引用。
- **CImg_4x4_ref(I, type, tab)**：定义一个名为 I，类型为 type 的 4x4 邻域作为 tab 的一个引用。
- **CImg_5x5_ref(I, type, tab)**：定义一个名为 I，类型为 type 的 5x5 邻域作为 tab 的一个引用。
- **CImg_2x2x2_ref(I, type, tab)**：定义一个名为 I，类型为 type 的 2x2x2 邻域作为 tab 的一个引用。
- **CImg_3x3x3_ref(I, type, tab)**：定义一个名为 I，类型为 type 的 3x3x3 邻域作为 tab 的一个引用。

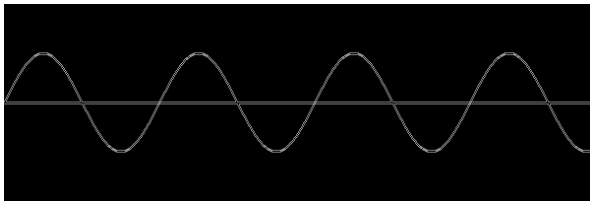
tab 可以是一个一维的 C 风格数组，或是一个非空的 CImg<T>图像。但这两个对象都必须同相应的邻域拥有同样的大小。

2.6.4.4.示例代码

不用过多的讨论，下面的例子将演示如何用 cimg_fro3x3x3()循环宏来计算一个 3D 量的梯度模（gradient norm）：



原图像



梯度模图像

```
//加载图片 2_1.bmp 作为数据源
CImg<float> volume("2_1.bmp");
//定义一个 3x3x3 的邻域
CImg_3x3x3(I,float);
//创建一幅同"volume"大小相同的图像
CImg<float> gradnorm(volume);

//使用邻域 I 在 volume 上循环
cimg_for3x3x3(volume, x, y, z, 0, I) {
  //计算 x 轴向上的差分
   const float ix = 0.5f * (Incc - Ipcc);
   //计算 y 轴向上的差分
   const float iy = 0.5f * (Icnc - Icpc);
   //计算 z 轴向上的差分
   const float iz = 0.5f * (Iccn - Iccp);

  // 在目标图像上设置梯度模
   gradnorm(x, y, z) = std::sqrt(ix*ix + iy*iy + iz*iz);
}

gradnorm.display("Gradient norm");
```

而下面的例子展示了如何通过计算 5x5 邻域的像素平均值使用邻域引用来柔化一幅彩色图像。

```
//定义图像
CImg<unsigned char> src("2_1.bmp"), dest(src,false), neighbor(5,5);

//避免在宏 CImg_5x5_ref 的第二个参数中出现空格
typedef unsigned char uchar;
//定义一个 5x5 的邻域作为图像 5x5 邻居的引用
CImg_5x5_ref(N,uchar,neighbor);

//色彩通道上的标准循环
cimg_forV(src,k)
  //5x5 邻域上的循环
   cimg_for5x5(src,x,y,0,k,N)
     //对像素求平均来滤波彩色图形
      dest(x,y,k) = neighbor.sum()/(5*5);

CImgList<unsigned char> visu(src,dest);
//同时显示原始图像和柔化后图像
visu.display("Original + Filtered");
```

www.cfannet.com C饭网，云淡风轻

原图像　　柔化后的图像

注意，在这个例子中，我们没有直接使用变量 Nbb、Nbp、……、Ncc、……因为只有领域图像的引用 neighbor 。并且我们使用了 neighbor 的一个成员函数。
就像你看到的，解释 CImg 邻域宏的作用要比使用它们难的多！

2.7. 使用显示窗口

当打开一个显示窗口时，你可以在像素值被显示到屏幕前选择规范化它们的方法。屏幕显示仅支持$[0,255]$之间的颜色值（原文不全）。
当使用 CImgDisplay::display()将图像显示在显示窗口内时，图像像素值由于可视化的目的最终被线性规范化到$[0,255]$之间。例如，在显示像素值在$[0,1]$之间的 CImg<double>图像时，这是很有用的。该规范化行为决定于 normalize 的取值，normalize 的值可以取 0、1 或 2：

- 0：在显示图像时不进行像素规范化。这是最快的方法，但是你必须保证你所要显示的图像，其像素值必须在区间$[0,255]$内。
- 1：对每个要显示的新图形都进行像素值规范化。图像的像素本身没有被修改，只是显示的像素被规范化了。
- 2：对第一个显示的图像进行像素值规范化，然后规范化参数被保留并被用来规范化后面所以显示的图像。

2.8. CImg中像素数据的保存

待添加

2.9. CImg中的文件IO

CImg 库本身就可以处理下列文件格式：

- **RAW**：先是一个非常简单的头部（ASCII 形式的），然后是图形数据。
- **ASC**（ASCII）
- **HDR**（Analyze 7.5）
- **INR**（Inrimage）
- **PPM/PGM**（**P**ortable **P**ixmap）
- **BMP**（未压缩）
- **PAN**（Pandore-5）
- **DLM**（Matlab ASCII）

如果安装了 ImageMagick，CImg 库把图像保存为 ImageMagick 可以处理的格式：**JPG**、**GIF**、**PNG**、**TIF**……

2.10. 检索命令行参数

CImg 库为基于控制台的程序从命令行中检索参数提供了相关工具。从命令行中检索参数通常是个必须的操作。为了此目的定义了三个宏：cimg_usage()、cimg_help()和 cimg_option()。使用这些宏使从命令行检索选项值变得容易。在调用编译得到的可执行文件时在要求的选项列表后附带参数-h 或-help 将自动显示程序的使用情况。

2.10.1. 宏cimg_usage()

宏 cimg_usage(usage)可以被用来描述程序的目的和使用情况。它通常被插入到 int main(int argc, char **argv)的定义之后。

参数：

usage：	描述程序目的和使用情况的字符串。

前置条件：

使用 cimg_usage()的函数必须正确的定义了变量 argc 和 argv。

2.10.2. 宏cimg_help()

宏 cimg_help(str)仅在运行程序时带有-h 或-help 选项时显示字符串 str。

2.10.3. 宏cimg_option()

宏 cimg_option(name, default, usage)用来从命令行检索一个选项值。

参数：

name:	将从命令行检索的选项的名字
default:	在运行程序时如果选项 name 没有被指定，此宏返回的缺省值
usage:	选项的一个简要说明。如果 usage == 0，当带有参数-h 或-help 运行程序时，此选项不会出现在选项列表中（隐藏此选项）

返回值：

cimg_option()返回一个同缺省值 default 有相同类型的对象。返回的值同在命令行中所指定的那个相同。如果某选项没有被指定，返回的值等于缺省值 default。**警告**：这在某些情况会引起混乱（见下一小节的结尾部分）。

前置条件：

使用 cimg_option()的函数必须正确的定义了变量 argc 和 argv。

2.10.4. 例程

下面的代码使用宏 cimg_usage()和 cimg_option()。这段代码加载一幅图像、平滑它然后使用一个指定的数值量化它。

```
#define cimg_use_jpeg CIMG_USE_JPEG
#include "CImg.h"

using namespace cimg_library;

int main(int argc, char *argv[])
{
  cimg_usage("Retrieve command line arguments");

  const char*    filename = cimg_option("-i", "test.bmp", "Input image file");
  const char*    output = cimg_option("-o", (char*) 0, "Output image file");
  const double   sigma = cimg_option("-s", 1.0, "Standard variation of the gaussian smoothing");
  const int      nblevels = cimg_option("-n", 16, "Number of quantification levels");
  const bool     hidden = cimg_option("-hidden", false, 0); // 这是一个隐藏选项

  CImg<unsigned char> img(filename);

  img.blur(sigma).quantize(nblevels);

  if(output){
    img.save(output);
  }
  else{
    img.display("Output image");
  }
  if(hidden){
    std::fprintf(stderr,"You found me !\n");
  }

  return EXIT_SUCCESS;
}
```

使用如下参数调用所得到的相应的可执行文件

-h -hidden -n 20 -i lena.jpg -o lena-b.bmp

将得到如下信息：

```
C:\CImg_example\2.11.4>cimg_ex -h -hidden -n 20 -i lena.jpg -o lena_b.bmp

 cimg_ex : Retrieve command line arguments (Feb 18 2008, 14:47:19)

    -i        = lena.jpg      : Input image file
    -o        = lena_b.bmp    : Output image file
    -s        = 1             : Standard variation of the gaussian smoothing
    -n        = 20            : Number of quantification levels
You found me !
```

警告：

由于宏 cimg_option(name, default, usage)所返回对象的类型是由 default 的类型所定义的，在写代码的时候可能会出现不期望的类型转换，例如：

```
const double sigma = cimg_option("-val", 0, "A floating point value");
```

这种情况下，sigma 将一直被赋予一个整数值（因为缺省值 0 是个整数）。当重命令行传递一个浮点值的时候，将会进行一个从浮点型到整形的转换，将所给定的参数截取为一个整数值（这当然不是一个所期望的行为）。在这种情况下你必须指定 0.0 为缺省值。

2.10.5. 如何学习更多关于命令行选项的知识？

你应该看一下在 CImg 库包中所提供的例子 examples/inrcast.cpp。这是一个集中使用宏 cimg_option()和 cimg_usage()检索命令行参数的基于命令行的图像转换程序。

3. CImg库名字空间文档

3.1. cimg_library名字空间参考手册

该名字空间包含 CImg 库的全部类和函数。

类

- struct **CImgException**
  当在 CImg 库函数调用中有错误发生时被抛出的类。
- struct **CImgStats**
  用来计算一幅 ***CImg***（对应页，待修改）*图像的像素值基本统计资料的类。*
- struct **CImgDisplay**
  这个类代表一个能用来显示 ***CImg***（对应页，待修改）*图像并能处理鼠标和键盘事件的窗口。*
- struct **CImg**
  表每个像素是类型 **T** *的一幅（最多 4 维）的图像。*
- struct **CImgList**
  表示一个由 ***CImg<T>****图像构成的列表。*

名字空间

- 名字空间 **cimg**
  包括了 CImg 库的低级函数和变量。

3.1.1. 详细描述

包含了 CImg 库的全部类和函数的名字空间。
定义这个名字空间是为了避免类名字冲突的，这可能在引入其他 c++头文件时发生。但是，这是不会经常发生的，你可以是你的大部分程序用如下代码开头

```
#include "CImg.h"
using namespace cimg_library;
```

以简化其后 CImg 库对象变量的声明。

3.2. cimg_library::cimg名字空间参考手册

该名字空间包括 CImg 库中的*低级*函数和变量。

函数

- void **info()**
  输出有关 CImg 环境变量的信息。
- template <typename tfunc, typename tp, typename tf> void **marching_cubes** (const tfunc &func, const float isovalue, const float x0, const float y0, const float z0, const float x1, const float y1, const float z1, const float resx, const float resy, const float resz, CImgList<tp> &points, CImgList<tf> &primitives, const bool invert_faces)
  多边形化(Polygonize)一个隐式函数。
- template<typename tfunc, typename tp, typename tf> void **marching_squares** (const tfunc &func, const float isovalue, const float x0, const float y0, const float x1, const float y1, const float resx, const float resy, CImgList<tp> &points, CImgList<tf> &primitives)
  多边形化(Polygonize)一个使用 marching squares 算法的隐式 2D 函数。
- const char * **imagemagick_path** ()
  返回 ImageMagick 的 convert 工具的路径。
- const char * **graphicsmagick_path** ()
  返回 GraphicsMagick 的 gmt 工具的路径。
- const char * **medcon_path** ()
  返回 XMedcon 工具的路径。
- const char * **temporary_path** ()
  返回保存临时文件的路径。
- bool **endian** ()
  对小端 CPU（Intel）返回 false，对大端 CPU（Motorola）返回 true。
- unsigned long **time** ()
  获得系统时钟的一个毫秒级精度值。
- void **sleep** (const unsigned int milliseconds)
  睡眠确定毫秒数时间。
- unsigned int **wait** (const unsigned int milliseconds)
  在最后一次调用后等待确定毫秒数目时间。
- template<typename T> T **abs** (const T &a)
  返回 a 的绝对值。
- template<typename T> const T & **min** (const T &a, const T &b)
  返回 a、b 中较小的那一个。
- template<typename T> const T & **min** (const T &a, const T &b, const T &c)
  返回 a、b 和 c 中最小的那一个。
- template<typename T> const T & **min** (const T &a, const T &b, const T &c, const T &d)
  返回 a、b、c 以及 d 中最小的那一个。
- template<typename T> const T & **max** (const T &a, const T &b)
  返回 a、b 中较大的那一个。
- template<typename T> const T & **max** (const T &a, const T &b, const T &c)
  返回 a、b 和 c 中最大的那一个。

- template<typename T> const T & **max** (const T &a, const T &b, const T &c, const T &d)
  返回 a、b、c 以及 d 中最大的那一个。
- template<typename T> T **sign** (const T &x)
  返回 x 的符号。
- template<typename T> unsigned long **nearest_pow2** (const T &x)
  返回比 x 大的且最近的 2 的幂。
- template<typename T> T **mod** (const T &x, const T &m)
  返回 x 模 m 的值（一般模运算）。
- template<typename T> T **minmod** (const T &a, const T &b)
  返回 minmod(a, b)。
- double **rand** ()
  返回一个满足 $[0, 1]$ *间均匀分布的随机变量。*
- double **crand** ()
  返回一个满足 $[-1, 1]$ *间均匀分布的随机变量。*
- double **grand** ()
  返回一个满足标准差为 1 的高斯分布的随机变量。
- double **round** (const double x, const double y, const unsigned int round_type=0)
  返回一个圆整后的数字。
- template<typename t> int **dialog** (const char *title, const char *msg, const char *button1_txt, const char *button2_txt, const char *button3_txt, const char *button4_txt, const char *button5_txt, const char *button6_txt, const CImg<t> &logo, const bool centering=false)
  显示一个用户能点击标准按钮的对话框。

变量

- const double **PI** = 3.14159265358979323846
  定义数学常量 PI。

3.2.1. 详细描述

包括 CImg 库中的*低级*函数和变量的名字空间。
这个名字空间里的大部分函数和变量都是被这个库用来完成低级处理的。虽然如此，这个名字空间中被文档说明的变量和函数也可以在你自己的源代码中安全使用。

警告：

永远不要在你的源代码中使用名字空间**cimg_library :: cimg**（- 23 -），因为**cimg ::**名字空间中的很多函数同定义在全局名字空间**::**中的标准C函数有相似的原型。

3.2.2. 函数文档

3.2.2.1.void info()

输出有关 CImg 环境变量的信息。
输出在标准错误输出上完成。