EagleEye 图像图形处理框架介绍

1. 数据封装

所有操作均是基于矩阵作为基本数据单元完成。矩阵操作方式类似matlab。

|  |  |
| --- | --- |
| 矩阵操作类型 | 表达形式 |
| 任意数据类型 | Matrix<float>; Matrix<int>;  Matrix<Array<unsigned char,3>> |
| 子矩阵提取 | Matrix<float> small\_matrix=big\_matrix(Range(0,10),Range(0,10)); |
| 矩阵相乘 | m\*n |
| … |  |

元数据类型

|  |  |
| --- | --- |
|  |  |
| 基本数据类型 | int; unsigned char; float … |
| 多通道数据类型 | Array<unsigned char, 3> 三通道数据（RGB）  Array<unsigned char,4> 四通道数据（RGBA） |

所有元数据类型均支持基本操作 加、减、乘、除

支持多数据类型的算法设计方案：

通过AtomicTypeTrait来萃取通过模板传递进来的元数据类型

|  |  |
| --- | --- |
| 操作 | 表达式 |
| 萃取数据类型 | AtomicTypeTrait<PixelType>::pixel\_type |
| 萃取通道数 | AtomicTypeTrait<PixelType>::size |
| 萃取多通道元数据类型中的元数据 | AtomicTypeTrait<PixelType>::AtomicType |
| 萃取元数据的最大及最小值 | AtomicTypeTrait<PixelType>::minval()  AtomicTypeTrait<PixelType>::maxval() |

通过萃取机制，便可以实现支持多数据类型的算法。如何设计见样板算法：FilterNode。

常用矩阵操作例子：

1. 矩阵间加、减、乘

Matrix<float> a(10,10,float(0));

Matrix<float> b(10,10,float(0));

加：a+b

减: a-b

乘: a\*b

1. 矩阵与标量数据间加、减、乘、除

Matrix<float> a(10,10,float(0));

加：a+5;

减：a-5

乘：a\*5

除：a/5

1. 转换当前矩阵到任意类型矩阵

强制类型转换

从Matrix<float> （单通道float矩阵类型矩阵）转换到 Matrix<Array<float,3>>(三通道float类型矩阵)

Matrix<float> old(10,10);

Matrix<Array<float,3>> nm=old.transform<Array<float,3>>();

从Matrix<Array<unsigned char,3>> （三通道unsigned char矩阵类型）转换到 Matrix<float> （单通道float矩阵类型）

Matrix<Array<unsigned char,3>> old(10,10);

Matrix<float> nm=old.transform<float>();

从Matrix<unsigned char> （单通道unsigned char 矩阵类型）转换到 Matrix<float> （单通道 float矩阵类型）

Matrix<unsigned char> old(10,10);

Matrix<float> nm=old.transform<float>();

自定义转化算子转换

暂略。

1. 典型操作说明

等号操作仅使同一片内存的引用计数增加

Matrix<float> src(10,10);

//src 和 target 共用同一片内存

Matrix<float> target=src;

如果想要使当前矩阵的内存独立则需要调用

target.clone();

如果想要复制来自其它矩阵的数据

Matrix<float> target(10,10);

target.copy(src);

提取子矩阵

Matrix<float> sub\_src=src(Range(row\_low,row\_high),Range(col\_low,col\_high));

注意此时sub\_src 与 src 依然是同一片内存。sub\_src仅仅是通过处理偏移来实现子矩阵操作。

如果想要使子矩阵的内存独立，使用sub\_src.clone();

对子矩阵的所有运算方式与一般矩阵一致。

1. 提取矩阵中指定位置处元素

Matrix<float> src(10,10,float(0));

提取(1,1)位置处的元素

float pos\_val=src(1,1);

float pos\_val=src(1\*10+1);

float pos\_val=src.at(1,1);

float pos\_val=src.at(1\*10+1);

1. 提取任意位置指针

Matrix<float> src(10,10);

提取第10个位置处的指针

src.anyptr(10)；

1. 提取矩阵所拥有的内存的起始位置

Matrix<float> src(10,10);

Matrix<float> sub\_src=src(Range(2,3),Range(5,9));

float\* src\_data=src.dataptr();

float\* sub\_src\_data=sub\_src.dataptr();

注：由于sub\_src是src的子矩阵，因而sub\_src与src共用一块内存。因而src\_data与sub\_src\_data相等。

1. 提取某一行的起始指针

Matrix<float> src(10,10);

提取第5行的起始指针

float\* src\_data=src.row(5);

1. 得到矩阵的行数和列数

Matrix<float> src(10,10);

int rows=src.rows();

int cols=src.cols();

1. 矩阵转置

Matrix<float> src(10,10);

Matrix<float> src\_t=src.t();

注：转置后的矩阵与原矩阵拥有两片内存。

1. 转换矩阵形状

Matrix<float> src(10,10);

Matrix<float> target=src.reshape(5,20);

将10\*10矩阵转换为5\*20矩阵，可选择按行排序或按列排序。

注：转换后的矩阵和原矩阵拥有两块不同的内存。

2. 算法管道模型

基本思想最大程度降低调用复杂性，建立一套方便的黑盒系统。

IO Node

(input)

Process Node 1

Process Node 2

Process Node 3

Process Node 4

IO Node

(output)

算法结构

IONode\* input\_node=new IONode;

ProcessNode1\* p\_node1=new ProcessNode1;

ProcessNode2\* p\_node2=new ProcessNode2;

ProcessNode3\* p\_node3=new ProcessNode3;

ProcessNode4\* p\_node4=new ProcessNode4;

IONode\* output\_node=new IONode;

//建立算法链接

p\_node1->setInputPort(input\_node->getOutputPort());

p\_node2->setInputPort(input\_node->getOutputPort());

p\_node3->setInputPort(p\_node1->getOutputPort());

p\_node4->setInputPort(p\_node2->getOutputPort());

output\_node->setInputPort(p\_node3->getOutputPort(),0);

output\_node->setInputPort(p\_node4->getOutputPort(),1);

//算法执行

output\_node->start();

//得到一些执行结果

1. 算法管道模型建立细节

算法之间端口链接依靠AnySignal 对象完成。由于当前处理的图像，因而我们使用从AnySignal派生的ImageSignal对象。

|  |
| --- |
| template<class T>  class ImageSignal:public AnySignal  {  public:  typedef T MetaType;  ImageSignal(Matrix<T> m=Matrix<T>(),char\* name="",char\* info="");  virtual ~ImageSignal(){};  /\*\*  \* @brief copy info  \*/  virtual void copyInfo(AnySignal\* sig);  Matrix<T> img;  std::string img\_name;  std::string ext\_info;  const EagleEyePixel pixel\_type;  const int channels;  }; |

所有的图像处理算法都需要封装到派生于AnyNode的类中。

需要由算法设计者补充的函数一共有三个（按照管道执行顺序排列）

selfcheck()

在这个函数里，我们可以实现一些判断语句。例如，是否当前算法所需要的所有条件均已经满足。比如在ImageReadNode中的这个函数里我们将会写下面这些代码：

if (m\_file\_path=="")

{

EAGLEEYE\_ERROR("file path is empty\n");

return false;

}

core::FileManager::FileHandle file\_handle=core::FileManager::FileFactory(m\_file\_path);

if (!file\_handle.get())

{

EAGLEEYE\_ERROR("sorry, I couldn't support this file type ()\n");

return false;

}

return true;

如果这个自检函数不能通过，整个算法管道就会停止（当然这个已经在底层自动实现）。

passonNodeInfo()

执行到这个函数时，说明这个算法处理节点已经通过了自检，所有参数已经齐备。关于数据的一些基本结构信息已经从上层处理节点得到，例如上层算法处理后图像的大小等信息。如果有需要我们可以利用这些信息来开辟一些内存等操作。（注意，此时只是一些结构信息，具体算法仍为执行）

executeNodeInfo()

执行具体算法。

1. 示例

见 FilterNode 滤波算法处理节点。