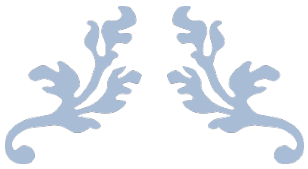


DIPPRO 函数接口文档

版本	修订日期	编写人
1.0	2015 年 3 月 10 日	Tony

GitHub：www.github.com/Tony-Tan/DIPpro

www.tcvpr.com

tony.sheng.tan@gmail.com

目录

版权声明

开发指南

开发者说明，本函数库使用 C 语言编写，调用 C 语言标准库函数，全部函数未针对机器做特殊优化，属于通用型函数库。

本函数库在 GitHub.com 中完全开放，属于开源软件，任何人和团体都可以基于此函数库继续开发，并且欢迎大家贡献代码。

后续版本中将陆续发布一些机器学习算法的函数供大家开发使用。

版本 1.0 中函数参数未经过验证合法性，且未对相关数据进行数据结构的封装，例如:图像以一维数组存储，并在函数调用时显式的传递图像的宽和高。

函数接口

ADABOOST

```
typedef struct TrainData_    TrainData;
struct TrainData_{
    double property;
    double w;
    char status;
    int label;
};
```

训练样本数据结构，其中
Property:为训练样本的特征值
w:为权重
status:为迭代中的分类状态（包括分类正确，和分类错误）
label:为样本的类别标签

double getBeta(double erro);

根据 adaboost 算法，通过误差值计算每次迭代的 Beta 值；
输入：erro 本次迭代的误差值
返回：Beta 值

double getAlpha(double beta);

根据 adaboost 算法计算每次迭代的 Alpha 值；
输入：
beta 由上面函数 getBeta 计算出的 Beta 值
返回：
函数返回 Alpha 值

void updataWi(TrainData *data,double beta);

更新样本的权重参数，正确分类的样本数据的权重将会降低，等价于错误分类样本权重增加。

输入：

Data：为训练样本组，

Beta：为误差更新权重比

void nomalization(TrainData *data);

归一化权重，使所有权重之和为 1。

输入:

Data：样本数据

void InitWi(TrainData *data);

初始化样本数据的权重。

输入:

Data：样本数据

void InitStatus(TrainData *data);

初始化样本数据的分类状态。

输入:

Data：样本数据

void Adaboost(TrainData *data,int T);

Adaboost 分类函数接口。

输入：

Data：样本数据

T：分类器数量

彩色基础

#define MIN3(x,y,z) (((x)>(y))?(((y)>(z))?(z):(y)):((x)>(z)?(z):(x)))

宏定义：找出 x,y,z 中最小的数据，返回最小值。

#define MAX3(x,y,z) (((x)<(y))?(((y)<(z))?(z):(y)):((x)<(z)?(z):(x)))

宏定义：找出 x,y,z 中最大的数据，返回最大值。

GMAX	255
COLOR_SPACE_RGB	0
COLOR_SPACE_CMY	1
COLOR_SPACE_HSI	2
COLOR_SPACE_HSV	3

以上定义色彩空间，灰度级定义为 0~255。

```
struct Chanel3_{
    double c1;
    double c2;
    double c3;
};
struct Chanel4_{
    double c1;
    double c2;
    double c3;
    double c4;
};
```

三通道和四通道数据结构，所有三通道色彩空间的成员全部按照上面定义的数据结构调用，例如

```
RGB Img;
Img.c1=100;
```

Img.c2=200;
Img.c3=150;

void Split(C3 *src ,double *dst1,double *dst2,double *dst3,int width,int height);

分离三通道数据为三个单通道图像矩阵。其输入可以是任意三通道结构数据。
输入
Src：三通道彩色空间数据。
Width：输入数据和输出数据的宽
Height：输入数据和输出数据的高
输出：
Dst1：c1 通道数据。
Dst2：c2 通道数据。
Dst3：c3 通道数据。

void Merge(double *src1,double *src2,double *src3,C3 *dst ,int width,int height);

合并三个单通道数据为一个整体的三通道数据。其输入可以是任意三通道数据的三个单独分量。
输入
src1：c1 通道数据。
src2：c2 通道数据。
src3：c3 通道数据。
Width：输入数据和输出数据的宽
Height：输入数据和输出数据的高
输出：
dst：三通道彩色空间数据。

void RGB2XYZ(RGB *src ,XYZ *dst,int width,int height);

色彩空间转换，从 RGB 到 XYZ 空间
输入：
Src：RGB 数据
Width：输入数据和输出数据的宽
Height：输入数据和输出数据的高
输出：
Dst：为 XYZ 空间数据。

void XYZ2RGB(XYZ *src ,RGB *dst,int width,int height);

色彩空间转换，从 XYZ 到 RGB 空间
输入：
Src：XYZ 数据
Width：输入数据和输出数据的宽
Height：输入数据和输出数据的高
输出：
Dst：为 RGB 空间数据。

void RGB2nRGB(RGB *src ,nRGB *dst,int width,int height);

色彩空间转换，从 RGB 到 nRGB 空间
输入：
Src：RGB 数据
Width：输入数据和输出数据的宽
Height：输入数据和输出数据的高
输出：
Dst：为 nRGB 空间数据。

void nRGB2RGB(nRGB *src ,RGB *dst,int width,int height);

色彩空间转换，从 nRGB 到 RGB 空间

输入：

Src：nRGB 数据

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

输出：

Dst：为 RGB 空间数据。

void RGB2CMY(RGB *src ,CMY *dst,int width,int height);

色彩空间转换，从 RGB 到 CMY 空间

输入：

Src：RGB 数据

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

输出：

Dst：为 CMY 空间数据。

void CMY2RGB(CMY *src ,RGB *dst,int width,int height);

色彩空间转换，从 CMY 到 RGB 空间

输入：

Src：CMY 数据

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

输出：

Dst：为 RGB 空间数据。

void RGB2sRGB(RGB *src ,sRGB *dst,int width,int height);

色彩空间转换，从 RGB 到 sRGB 空间

输入：

Src：RGB 数据

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

输出：

Dst：为 sRGB 空间数据。

void sRGB2RGB(sRGB *src ,RGB *dst,int width,int height);

色彩空间转换，从 sRGB 到 RGB 空间

输入：

Src：sRGB 数据

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

输出：

Dst：为 RGB 空间数据。

void sRGB2RGB(sRGB *src ,RGB *dst,int width,int height);

色彩空间转换，从 sRGB 到 RGB 空间

输入：

Src：sRGB 数据

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

输出：

Dst：为 RGB 空间数据。

void RGB2YIQ(RGB *src ,YIQ *dst,int width,int height);

色彩空间转换，从 RGB 到 YIQ 空间

输入：

Src：RGB 数据

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

输出：

Dst：为 YIQ 空间数据。

void YIQ2RGB(YIQ *src ,RGB *dst,int width,int height);

色彩空间转换，从 YIQ 到 RGB 空间
输入：
Src：YIQ 数据
Width：输入数据和输出数据的宽
Height：输入数据和输出数据的高
输出：
Dst：为 RGB 空间数据。

void RGB2YUV(RGB *src ,YUV *dst,int width,int height);

色彩空间转换，从 RGB 到 YUV 空间
输入：
Src：RGB 数据
Width：输入数据和输出数据的宽
Height：输入数据和输出数据的高
输出：
Dst：为 YUV 空间数据。

void YUV2RGB(YUV *src ,RGB *dst,int width,int height);

色彩空间转换，从 YUV 到 RGB 空间
输入：
Src：YUV 数据
Width：输入数据和输出数据的宽
Height：输入数据和输出数据的高
输出：
Dst：为 RGB 空间数据。

void RGB2YCbCr(RGB *src ,YCbCr *dst,int width,int height);

色彩空间转换，从 RGB 到 YCbCr 空间

输入：

Src：RGB 数据

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

输出：

Dst：为 YCbCr 空间数据。

void YCbCr2RGB(YCbCr *src ,RGB *dst,int width,int height);

色彩空间转换，从 YCbCr 到 RGB 空间

输入：

Src：YCbCr 数据

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

输出：

Dst：为 RGB 空间数据结构。

void RGB2HSV(RGB *src ,HSV *dst,int width,int height);

色彩空间转换，从 RGB 到 HSV 空间

输入：

Src：RGB 数据

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

输出：

Dst：为 HSV 空间数据。

void HSV2RGB(HSV *src ,RGB *dst,int width,int height);

色彩空间转换，从 HSV 到 RGB 空间

输入：

Src：HSV 数据

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高
输出：
Dst：为 RGB 空间数据。

void RGB2HSI(RGB *src ,HSI *dst,int width,int height);

色彩空间转换，从 RGB 到 HSI 空间
输入：
Src：RGB 数据
Width：输入数据和输出数据的宽
Height：输入数据和输出数据的高
输出：
Dst：为 HSI 空间数据。

void HSI2RGB(HSI *src ,RGB *dst,int width,int height);

色彩空间转换，从 HSI 到 RGB 空间
输入：
Src：HSI 数据
Width：输入数据和输出数据的宽
Height：输入数据和输出数据的高
输出：
Dst：为 RGB 空间数据。

void Complementary_Color(C3 *src,C3 *dst,int width,int height,int color_space_type);

反色操作，此功能输入色彩空间只能为 RGB，CMY，HSI，HSV
输入：
Src：RGB，CMY，HIS，HSV 色彩空间类型的三通道数据。
Width：输入数据和输出数据的宽
Height：输入数据和输出数据的高

Color_space_type：色彩空间类型，COLOR_SPACE_RGB 0， COLOR_SPACE_CMY 1， COLOR_SPACE_HSI 2， COLOR_SPACE_HSV 3

输出：

Dst：与输入相对应的色彩空间的反色输出。

彩色图像处理

void HistEqualRGB(RGB *src,RGB *dst,int width,int height);

对 RGB 类型图像进行亮度均衡，将 RGB 转化到 HSI，然后对 I 通道进行直方图均衡计算得到 I’，计算后合并 H，S 和 I’，并转换回 RGB 通道。

输入：

Src：输入 RGB 图像

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

输出：

Dst：亮度均衡后图像

void SmoothRGB(RGB *src,RGB *dst,int width,int height,int m_width,int m_height,double param1,double param2,int Smooth_type);

平滑RGB通道图像，平滑方法包括

```
SMOOTH_GAUSSIAN,SMOOTH_MEDIAN,SMOOTH_BILATERAL,SMOOTH_MEAN
```

算法操作 R，G，B 三个通道，分别进行平滑，然后将三通道合并。

输入：

Src：RGB 输入图像

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

m_width：平滑窗口宽

m_height：平滑窗口高

param1：平滑参数 1（高斯函数的标准差，双边滤波距离高斯函数的标准差）

param2：平滑参数 2（双边滤波函数的灰度高斯函数的标准差）

Smooth_type：平滑类型

输出：

Dst：平滑后 RGB 图像

void SmoothHSI(HSI *src,HSI *dst,int width,int height,int m_width,int m_height,double param1,double param2,int Smooth_type);

平滑HSI通道图像，平滑方法包括
SMOOTH_GAUSSIAN,SMOOTH_MEDIAN,SMOOTH_BILATERAL,SMOOTH_MEAN
算法操作 I 通道进行平滑，然后将三通道合并。
输入：
Src：HSI 输入图像
Width：输入数据和输出数据的宽
Height：输入数据和输出数据的高
m_width：平滑窗口宽
m_height：平滑窗口高
param1：平滑参数 1（高斯函数的标准差，双边滤波距离高斯函数的标准差）
param2：平滑参数 2（双边滤波函数的灰度高斯函数的标准差）
Smooth_type：平滑类型
输出：
Dst：平滑后 HSI 图像

void SharpenRGB(RGB *src,RGB *dst,int width,int height,double c,int Sharpen_type);

锐化RGB通道图像，锐化方法包括
SHARPEN_LAPLACE，SHARPEN_SOBEL，SHARPEN_ROBERT算法操作 R，G，B 三个通道，分别进行锐化，然后将三通道合并。
输入：
Src：RGB 输入图像
Width：输入数据和输出数据的宽
Height：输入数据和输出数据的高
C：细节加强系数（使用相关算子检测到细节后，加入到原图时的参数）。
Sharpen_type ：锐化类型。
输出：

Dst：锐化后 RGB 图像

void SharpenHSI(HSI *src,HSI *dst,int width,int height,double c,int Sharpen_type);

锐化HSI通道图像，锐化方法包括SHARPEN_LAPLACE，SHARPEN_SOBEL，SHARPEN_ROBERT算法操作 I 通道，进行锐化。

输入：

Src：HSI 输入图像

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

C：细节加强系数（使用相关算子检测到细节后，加入到原图时的参数）。

Sharpen_type ：锐化类型。

输出：

Dst：锐化后 HSI 图像

void SegmentRGB(RGB* src,RGB *dst,int width,int height,RGB * color_center,double threshold);

分割 RGB 通道图像，通过颜色阈值，在 RGB 三维空间中，以颜色中心点为球心，阈值为半径，在这个球内的颜色将被保留，超出此球的颜色将会置零。

输入：

Src：输入 RGB 三通道彩色图像

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

Color_center：将要分割的颜色中心（球心）

Threshold：分割阈值，球半径。

输出：

Dst：输出 RGB 三通道彩色图像。

void Cover_RGB(RGB *src,RGB *dst,RGB *cover,int width,int height);

彩色图像覆盖操作，cover 为覆盖模板，如果 cover 上的点（x，y）为（R=0，G=0，B=0）,则输出结果在（x，y）处显示 src 在点（x，y）处的颜色，否则显示为 cover 在点（x，y）处的颜色。

输入：

Src：输入 RGB 三通道彩色图像

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

Cover：彩色覆盖模板

输出：

Dst：输出 RGB 三通道彩色图像。

void Zero_RGB(RGB *src,int width,int height);

RGB 图像置零，将 RGB 图像置零

输入：

Src：RGB 图像

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

注意，此操作可以用于所有三通类型数据的置零操作。

void Copy_RGB(RGB *src,RGB *dst,int width,int height);

RGB 图像复制，将输入 RGB 图像复制到另外一个 RGB 数据中。

输入：

Src：RGB 图像

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

输出：

Dst：复制的输出

注意，此操作可以用于所有三通道类型数据的复制操作。

void Threshold_RGB(RGB *src,RGB *dst,RGB *threshold,int width,int height);

阈值分割 RGB 通道值，threshold 为阈值，为一个三通道的向量的指针而不是一个三通道数组。

输入：

Src：输入 RGB 数据

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

Threshold：阈值的指针

输出：

Dst：分割后的图像。

灰度变换

图像灰度变换，变换方式共如下方式：

灰度变换宏	变换对应整数编号	变换方式
CONTRASTFUNC0	-1	翻转
CONTRASTFUNC1	0	分段
CONTRASTFUNC2	1	对数
CONTRASTFUNC3	2	反对数
CONTRASTFUNC4	3	n 次幂
CONTRASTFUNC5	4	n 次根
CONTRASTGAMA	5	gama 变换

void ContrastStretch(double *src,double *dst,int width,int height,int method,double p0,double p1,int p2,int p3);

对灰度图像进行灰度变换，变换类型见上表。

输入：

Src：灰度图像

Width：输入数据和输出数据的宽

Height：输入数据和输出数据的高

Method：灰度变换方法（见上表）

P0：double 型变量

P1：double 型变量

P2：int 型变量

P3：int 型变量

注意：分段变换是（p0，p1）（p2，p3）位两个转折点，N 次根操作经过规定化，0 将映射到 0，255 将映射到 255，伽马变换中，p0 为 gama 值，p1 为变换参数，如果 p1 将 255 映射到超过 255 时，p1 无效。将自动转换为使 255 映射到 255 的系数。

输出：

Dst：变化后的灰度图像

矩阵卷积与相关

void RotateRealMatrix(double *matrix,double *dst,int width,int height)

实数矩阵 180°旋转。从一维存储结构来看等价于将一维数组倒序。

输入：

Matrix：实数矩阵

Width：矩阵宽。

Height：矩阵高。

输出：

Dst：旋转后的矩阵

void RotateComplexMatrix(Complex *matrix,int width,int height);

复数矩阵 180°旋转。从一维存储结构来看等价于将一维数组倒序。

输入：

Matrix：复数矩阵

Width：矩阵宽。

Height：矩阵高。

输出：

Matrix：复数矩阵旋转结果

void RealRelevant(double *src,double *dst,double *mask,int width,int height,int m_width,int m_height);

实数矩阵相关操作，相关操作具体可以参考 wiki 给出的定义。

输入：

Src：输入矩阵

Mask：卷积模板

Width：输入输出矩阵矩阵宽

Height：输入输出矩阵矩阵高

M_width：模板宽

M_height：模板高

输出：

Dst：相关结果输出（大小与 src 一致，宽 width，高 height）。

void ComplexRelevant(Complex* src,Complex *dst,Complex *mask,int width,int height,int m_width,int m_height);

复数矩阵相关操作，相关操作具体可以参考 wiki 给出的定义。

输入：

Src：输入矩阵

Mask：卷积模板

Width：输入输出矩阵矩阵宽

Height：输入输出矩阵矩阵高

M_width：模板宽

M_height：模板高

输出：

Dst：相关结果输出（大小与 src 一致，宽 width，高 height）。

void RealConvolution(double *src,double *dst,double *mask,int width,int height,int m_width,int m_height);

实数矩阵卷积操作，卷积操作具体可以参考 wiki 给出的定义。

输入：

Src：输入矩阵

Mask：卷积模板

Width：输入输出矩阵矩阵宽

Height：输入输出矩阵矩阵高

M_width：模板宽

M_height：模板高

输出：

Dst：卷积结果输出（大小与 src 一致，宽 width，高 height）。

void ComplexCovolution(Complex* src,Complex *dst,Complex *mask,int width,int height,int m_width,int m_height);

复数矩阵卷积操作，卷积操作具体可以参考 wiki 给出的定义。

输入：

Src：输入矩阵

Mask：卷积模板

Width：输入输出矩阵矩阵宽

Height：输入输出矩阵矩阵高

M_width：模板宽

M_height：模板高

输出：

Dst：卷积结果输出（大小与 src 一致，宽 width，高 height）。

滤波器

int ChangtoPower2(int size);

将数字转化为比它大的最近的 2 的幂次函数，例如输入 127 将得到 128，输入 128 将得到 128，输入 129，将得到 256；
输入：
Size：输入整数
输出：
返回大于输入的最接近的 2 的整数次幂的结果。

void ResizeMatrix4FFT(double *src,double **dst,int width,int height);

将矩阵进行大小变换，使其高和宽为 2 的整数幂，以便进行 FFT 计算。
输入：
Src：原始图像
Width：输入输出矩阵宽度
Height：输入输出矩阵高度
输出：
Dst：变换后图像，dst 在函数内建立内存空间，故需要输入指针的指针作为参数，并在使用后进行释放，避免内存泄露。

double FrequencyFiltering(double *src,int width,int height,double *dst,int filter_type,double param1,int param2,double param3,double param4,double param5,int isgetPower);

频域滤波函数，支持理想低通，布特沃斯低通，高斯低通，理想高通，布特沃斯高通，高斯高通，同态滤波，高频提升滤波等滤波操作，参数 1~5 给出相关操作的参数，说明如下：
输入：
Src：输入图像矩阵

Width：图像矩阵宽度

Height：图像矩阵高度

Filter_type：滤波类型 ILPF，BLPF，GLPF，IHPF，BHPF，GHPF，HOMPF，HFEPF

Param1：对应于截止频率

Param2：布特沃斯滤波器参数 N，同态滤波 lamda_l，高频提升参数 k1

Param3：同态滤波 lamda_h，高频提升参数 k2

Param4：高频提升滤波器的类型选择（使用对应的整数，具体参数见《高频滤波器》）

Param5：高频提升滤波器中布特沃斯滤波器参数 N

isgetPower：是否计算能量

输出：

Dst：滤波后图像矩阵，如需要计算能量，函数返回能量值

double getPower(Complex *src,int totalsize);

计算能量值。

输入：

Src：复数矩阵

Totalsize：总数量

输出：

函数返回能量值

傅里叶变换

void DFT(double * src,Complex * dst,int size);

一维 DFT 计算，使用原始 DFT 定义公式计算，速度慢，但可以对任意大小的数组进行计算，输入为实数，输出为复数。
输入：
Src：一维实数信号
Size：信号大小
输出：
Dst：一维复数信号

void IDFT(Complex *src,Complex *dst,int size);

一维 IDFT 计算，使用原始 IDFT 定义公式计算，速度慢，但可以对任意大小的数组进行计算，输入为复数，输出为复数（有可能是实数，此时虚部为 0）。
输入：
Src：一维复数信号
Size：信号大小
输出：
Dst：一维复数信号

void RealFFT(double * src,Complex * dst,int size_n);

实数快速傅里叶变换。使用蝶形计算方法，如果数组大小非 2 的整数幂，程序退出，所以在输入前，应该将信号进行扩充运算。
输入：
Src：一维实数信号
Size_n：信号长度，必须为 2 的整数次幂
输出：
Dst：一维复数信号

void FFT(Complex * src,Complex * dst,int size_n);

标准 FFT 变换，输入信号为复数，信号大小应该为 2 的整数次幂，否者程序退出，故在信号输入前调整信号长度。
输入：
Src：一维复数信号
Size_n：一维信号长度，非 2 的整数次幂退出
输出：
Dst：快速傅里叶变换结果，一维复数信号

void IFFT(Complex * src,Complex * dst,int size_n);

逆快速傅里叶变换，输入为复数（傅里叶变换后的序列），size_n 信号长度，必须为 2 的整数次幂，否则程序退出。
输入：
Src：一维复数信号序列
Size_n：一维复数信号序列大小，必须为 2 的整数次幂，否则程序退出。
输出：
Dst：输出逆傅里叶变换结果，为复数。

int DFT2D(double *src,Complex *dst,int size_w,int size_h);

二维标准 DFT 变换，输入为实数矩阵。
输入：
Src：二维实数信号。
Size_w：信号矩阵宽度
Size_h：信号矩阵高度
输出：
Dst：输出复数矩阵。

int IDFT2D(Complex *src,Complex *dst,int size_w,int size_h);

二维标准 IDFT 变换，输入为复数矩阵。

输入：

Src：二维复数信号。

Size_w：信号矩阵宽度

Size_h：信号矩阵高度

输出：

Dst：输出复数矩阵。

int FFT2D(double *src,Complex *dst,int size_w,int size_h);

二维 FFT 变换，输入为实数矩阵，其长和宽必须为 2 的整数次幂，否则程序退出。

输入：

Src：二维实数信号。

Size_w：信号矩阵宽度，必须为 2 的整数次幂

Size_h：信号矩阵高度，必须为 2 的整数次幂

输出：

Dst：输出复数矩阵。

int IFFT2D(Complex *src,Complex *dst,int size_w,int size_h);

二维 IFFT 变换，输入为实数矩阵，其长和宽必须为 2 的整数次幂，否则程序退出。

输入：

Src：输入复数矩阵。

Size_w：信号矩阵宽度，必须为 2 的整数次幂

Size_h：信号矩阵高度，必须为 2 的整数次幂

输出：

Dst：输出复数矩阵。

灰度形态学

int isSmooth(double *src,int width,int height);

判断结构元是否平滑，如果全部为 0 或 255，认为结构元为 2 值化的，为平滑结构元，否则为非平滑结构元（非平滑结构元会破坏原图像的性质，不常用，但不排除有特殊用途）。
输入：
Src：实数矩阵
Width：矩阵宽度
Height：矩阵高度
输出：
如果是平滑结构元，函数返回 1 否则返回 0.

void G_Translation(double *src,double *dst,int width,int height,double SEvalue,Position *d,int istoFindMin);

位移操作，如果非平滑 SE 将加上 SEvalue 即结构元对应的灰度值，若超过 255，设置为 255，如果位移后矩阵元素在矩阵大小以外，则忽略这些元素。
输入：
Src：原始矩阵
Width：输入和输出矩阵宽度
Height：输入和输出矩阵高度
SEvalue：为结构元当前元素值。
*d：为位移向量，如果 d（x，y），则位移向量为（x，y）
istoFindMin：决定初始化时使用 0 还是 1 填充，如果 istoFindMin 设置为 1 则用 1 填充，否则使用 0 填充，也就是如果某些位置在变化后没有元素被放入，则用 1 或者 0 填充，使用该参数控制。
输出：
Dst，位置变换后的矩阵。

void Dilate_Gray(double *src,double *dst,int width,int height,double *se,int sewidth,int seheight,Position *center);

灰度图像膨胀操作。

输入：

Src：输入图像矩阵

Width：输入和输出矩阵宽度

Height：输入和输出矩阵高度

Se：结构元矩阵

Sewidth：结构元宽度

Seheight：结构元高度

Center：结构元中心位置

输出：

Dst：膨胀后的图像矩阵

void Erode_Gray(double *src,double *dst,int width,int height,double *se,int sewidth,int seheight,Position *center);

灰度图像腐蚀操作。

输入：

Src：输入图像矩阵

Width：输入和输出矩阵宽度

Height：输入和输出矩阵高度

Se：结构元矩阵

Sewidth：结构元宽度

Seheight：结构元高度

Center：结构元中心位置

输出：

Dst：腐蚀后的图像矩阵

void Open_Gray(double *src,double *dst,int width,int height,double *se,int sewidth,int seheight,Position *center);

灰度图像开操作，对灰度图像先进行腐蚀操作，然后对腐蚀的结果进行膨胀，两次操作使用同一结构元和结构元中心位置。

输入：

Src：输入图像矩阵

Width：输入和输出矩阵宽度

Height：输入和输出矩阵高度

Se：结构元矩阵

Sewidth：结构元宽度

Seheight：结构元高度

Center：结构元中心位置

输出：

Dst：开操作后的图像矩阵

void Close_Gray(double *src,double *dst,int width,int height,double *se,int sewidth,int seheight,Position *center);

灰度图像闭操作，对灰度图像先进行膨胀操作，然后对膨胀的结果进行腐蚀，两次操作使用同一结构元和结构元中心位置。

输入：

Src：输入图像矩阵

Width：输入和输出矩阵宽度

Height：输入和输出矩阵高度

Se：结构元矩阵

Sewidth：结构元宽度

Seheight：结构元高度

Center：结构元中心位置

输出：

Dst：闭操作后的图像矩阵

void Gard_Gray(double *src,double *dst,int width,int height,double *se,int sewidth,int seheight,Position *center);

灰度梯度形态学提取，此算法原理为先对原图像矩阵进行膨胀，然后对原图像进行腐蚀，两次结果做差，得出形态学梯度。

输入：

Src：输入图像矩阵

Width：输入和输出矩阵宽度

Height：输入和输出矩阵高度

Se：结构元矩阵

Sewidth：结构元宽度

Seheight：结构元高度

Center：结构元中心位置

输出：

Dst：灰度形态学梯度图像矩阵

void TopHat(double *src,double *dst,int width,int height,double *se,int sewidth,int seheight,Position *center);

顶帽操作，对灰度图像进行开操作，并用原图像矩阵减去开操作结果，得到所有被开操作削掉的峰顶，得出图像中相对较亮的部分

输入：

Src：输入图像矩阵

Width：输入和输出矩阵宽度

Height：输入和输出矩阵高度

Se：结构元矩阵

Sewidth：结构元宽度

Seheight：结构元高度

Center：结构元中心位置

输出：

Dst：顶帽操作结果的图像矩阵

void BottomHat(double *src,double *dst,int width,int height,double *se,int sewidth,int seheight,Position *center);

底帽操作，对灰度图像进行闭操作，并用闭操作结果减去原图像，得到所有被闭操作填充的谷底，得出图像中相对较暗的部分

输入：

Src：输入图像矩阵

Width：输入和输出矩阵宽度

Height：输入和输出矩阵高度

Se：结构元矩阵

Sewidth：结构元宽度

Seheight：结构元高度

Center：结构元中心位置

输出：

Dst：底帽操作结果的图像矩阵

void Erode_Gray_g(double *src,double *ground,double *dst,int width,int height,double *se,int sewidth,int seheight,Position *center);

灰度图像测地腐蚀，将腐蚀后的灰度图像矩阵与原图进行最大值处理（与二值图像中的“或”相同，即找出两幅图像中对应位置元素的较小的，作为结果）

输入：

Src：输入图像矩阵

Ground：输入的“地”

Width：输入和输出矩阵宽度

Height：输入和输出矩阵高度

Se：结构元矩阵

Sewidth：结构元宽度

Seheight：结构元高度

Center：结构元中心位置

输出：

Dst：测地腐蚀操作结果的图像矩阵

void Dilate_Gray_g(double *src,double *ground,double *dst,int width,int height,double *se,int sewidth,int seheight,Position *center);

灰度图像测地膨胀，将膨胀后的灰度图像矩阵与原图进行最小值处理（与二值图像中的“与”相同，即找出两幅图像中对应位置元素的较小的，作为结果）

输入：

Src：输入图像矩阵

Ground：输入的“地”

Width：输入和输出矩阵宽度

Height：输入和输出矩阵高度

Se：结构元矩阵

Sewidth：结构元宽度

Seheight：结构元高度

Center：结构元中心位置

输出：

Dst：测地膨胀操作结果的图像矩阵

高通滤波器

IHPF	4
BHPF	5
GHPF	6

void IdealHPFilter(double *Filter,int width,int height,double cut_off_frequency);

生成理想高通滤波器，其滤波结果振铃效应较大，不建议使用。滤波器值域为[0,1]
注意，为保持处理后图像灰度不至于太低，将直流分量（0,0）处设为 1.0；
Filter：滤波器
Width：滤波器宽
Height：滤波器高
Cut_off_frequency：截止频率

void ButterworthHPFilter(double *Filter,int width,int height,double cut_off_frequency,int n);

$$H(u,v)=\frac{1}{1+\left[D(u,v)/D_0\right]^{2n}}$$

生成布特沃斯高通滤波器，其滤波结果振铃效应根据参数 n 而定，n 越大，滤波器边缘越陡峭，振铃效果越明显。滤波器值域为[0,1]
注意，为保持处理后图像灰度不至于太低，将直流分量（0,0）处设为 1.0；
Filter：滤波器
Width：滤波器宽
Height：滤波器高
Cut_off_frequency：截止频率
N：滤波器参数

void GaussianHPFilter(double *Filter,int width,int height,double cut_off_frequency);

生成高斯高通滤波器，无振铃效应。滤波器值域为[0,1]
注意，为保持处理后图像灰度不至于太低，将直流分量（0,0）处设为 1.0；
Filter：滤波器
Width：滤波器宽
Height：滤波器高
Cut_off_frequency：截止频率

void HighFrequencyEmphasisFilter(double *Filter,int width,int height,int filter_type,double cut_off_frequency,double k1,double k2,double param1);

高频强调滤波器，即在使用“理想高通，布特沃斯高通，高斯高通滤波器”的时候对滤波器进行增强，相应的增强方式 k1+k2*filter(x,y)，在 k1 的基础上对滤波器进行拉伸或压缩（取决于 k2 的数值）。
Filter：滤波器
Width：滤波器宽
Height：滤波器高
filter_type：滤波器类型，包括：
IHPF 理想高通， BHPF 布特沃斯高通， GHPF高斯高通
Cut_off_frequency：截止频率
K1：参数 1
K2：参数 2
param1：布特沃斯滤波器参数 N

直方图

void HistogramEqualization(double *src,double *dst,int width,int height);

灰度图像直方图均衡
输入：
Src：原始图像矩阵
Width：输入和输出图像矩阵宽度
Height：输入和输出图像矩阵高度
输出：
Dst：直方图均衡后的图像矩阵

void HistogramSpecification(double *src,double *dst,int* hist,int width,int height);

灰度图像直方图规定化，将原始图像的直方图朝向提供的直方图的模式进行变换。当提供的直方图为恒定数值的时候，等效于直方图均衡。
输入：
Src：原始图像矩阵
Hist：目标直方图
Width：输入和输出图像矩阵宽度
Height：输入和输出图像矩阵高度
输出：
Dst：直方图规定化后的图像矩阵

void setHistogram(double *src,int *hist,int width,int height);

获取图像矩阵的直方图，直方图的维度为灰度级数（256），直方图为整型数组。
输入：
Src：输入图像矩阵

Width：输入图像矩阵宽度

Height：输入图像矩阵高度

输出：

Hist：输出直方图

void InitHistogram(int *hist);

初始化直方图，将所有项初始化为 0

输入：

Hist：直方图数组，大小为灰度级相应大小（256）

int findHistogramMax(int *hist);

通过直方图查找图像最大灰度值

输入：

Hist：直方图

输出：

返回最大值

int findHistogramMin(int *hist);

通过直方图查找图像最小灰度值

输入：

Hist：直方图

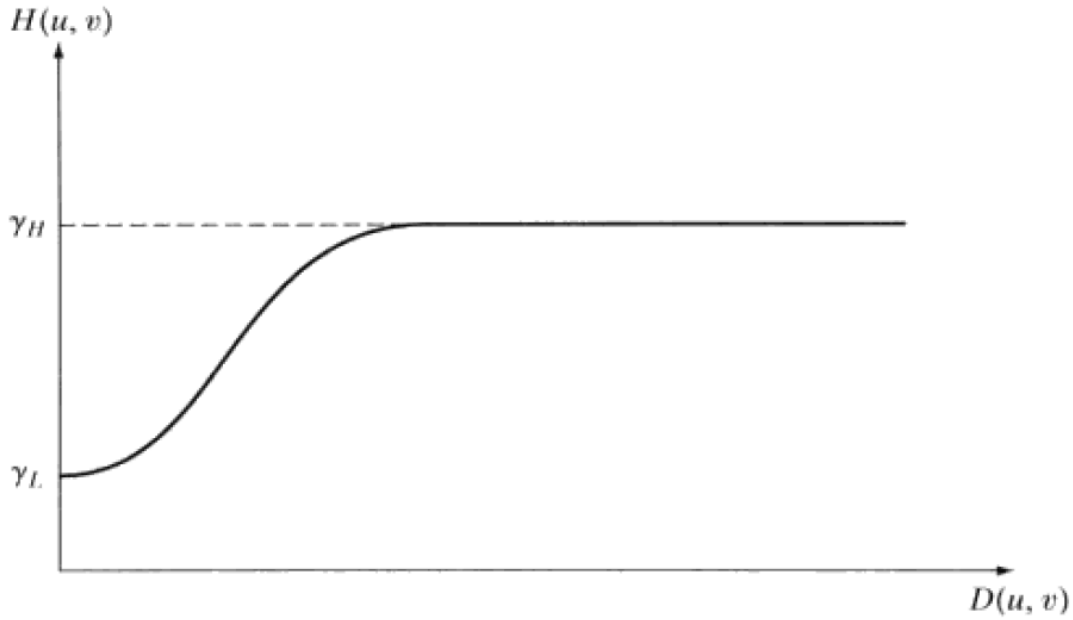
输出：

返回最小值

同态滤波

void HomorphicFilter(double *filter,int width,int height,double cut_off_frequency,double lambda_l,double lambda_h);

同态滤波，通过对原始图像进行同态变换，并根据成像原理，将高频部分视为反射部分，低频部分视为光照部分。滤波器形状大致如下。



输入：

Filter：同态滤波器矩阵

Width：滤波器宽

Height：滤波器高

Cut_off_frequency：滤波器截止频率

lambda_l：最低频率系数

lambda_h：最高频率系数

输出：

同态滤波器

霍夫变换

void HoughLine(double *src,double *dst,int width,int height,int lineLength);

霍夫变换，检测直线，对于输入的二值图像，检测其中的超过 linelength 长度的直线并在 dst 中绘制该直线。

输入：

Src：输入二值图像矩阵

Width：输入和输出图像矩阵宽度

Height：输入和输出图像矩阵高度

Linelength：检测到的直线长度的阈值

输出：

Dst：绘制超过阈值长度的直线

图像傅里叶变换

void FFT_Shift(double * src,int size_w,int size_h);

移位操作，根据傅里叶变换性质，将 FFT 变换的低频部分转移到图像中心
输入：
Src：数据矩阵
Size_w：矩阵宽度
Size_h：矩阵高度

void ImageFFT(double * src,int width,int height,Complex * dst);

图像快速傅里叶变换，将得到与图像大小相等的复数矩阵
输入：
Src：输入图像矩阵
Width：输入图像矩阵，输出复数矩阵的宽
Height：输入图像矩阵，输出复数矩阵的高
输出：
Dst：傅里叶变换结果复数矩阵

void Nomalsize(double *src,double *dst,int size_w,int size_h);

规则化矩阵内元素大小到 0~255 之内用于显示矩阵，多用于对频谱的显示工作。
输入：
Src：输入矩阵，值域范围不做规定
Size_w：输入输出矩阵宽度
Size_h：输入输出矩阵高度
输出：
Dst：输出矩阵，值域范围 0~255

void getAmplitudespectrum(Complex * src,double *dst,int size_w,int size_h);

计算法傅里叶谱的幅度，用于显示频谱（幅度）图像。
输入：
Src：复数矩阵
Size_w：输入输出矩阵的宽
Size_h：输入输出矩阵的高
输出：
Dst：输出复数矩阵的幅度矩阵

void ImageIFFT(Complex *src,double *dst,int size_w,int size_h);

图像快速逆傅里叶变换，将得到与频谱大小相等的实数矩阵
输入：
Src：输入频谱复数矩阵
Width：输入图像矩阵，输出复数矩阵的宽
Height：输入图像矩阵，输出复数矩阵的高
输出：
Dst：逆傅里叶变换结果实数矩阵

低通滤波器

ILPF	1
BLPF	2
GLPF	3

低通滤波器，定义理想低通 1，布特沃斯低通 2，高斯低通 3

void IdealLPFilter(double *Filter,int width,int height,double cut_off_frequency);

生成理想低通滤波器，其振铃效果与理想高通滤波器类似，相对严重，顾应用中很少使用。

输入：

Filter：滤波器矩阵

Width：滤波器宽度

Height：滤波器高度

cut_off_frequency：滤波器截止频率。

输出：

Filter：理想低通滤波器。

void ButterworthLPFilter(double *Filter,int width,int height,double cut_off_frequency,int n);

生成布特沃斯低通滤波器，其滤波结果振铃效应根据参数 n 而定，n 越大，滤波器边缘越陡峭，振铃效果越明显。滤波器值域为[0,1]。

输入：

Filter：滤波器

Width：滤波器宽

Height：滤波器高

Cut_off_frequency：截止频率

N：滤波器参数

输出：

Filter：布特沃斯低通滤波器

void GaussianLPFilter(double *Filter,int width,int height,double cut_off_frequency);

生成高斯低通滤波器，无振铃效应。滤波器值域为[0,1]

注意，为保持处理后图像灰度不至于太低，将直流分量（0,0）处设为 1.0；

输入：

Filter：滤波器

Width：滤波器宽

Height：滤波器高

Cut_off_frequency：截止频率

输出：

Filter：高斯低通滤波器

基本数学运算库

M_PI	3.14159265358979323846
W_PI	57.29577951308233

定义 π 为 M_Pi，弧度和角度的转换系数 W_Pi

GRAY_LEVEL	256
isEVEN(x)	(!(x%2))

判断整数是否为偶数，是返回 1 否则返回 0；

GETINT(x)　(((x)-((double)((int)(x)))>=0.5)?((int)(x)+1):((int)(x)))

对小数进行舍入取整，大于 0.5 的进 1.否则舍去小数

```
struct Position_{
    int x;
    int y;
};
```

定义位置数据结构，整型，像素级别位置信息。

```
struct Position_DBL_{
    double x;
    double y;
};
```

定义位置数据结构，双浮点型，亚像素级别位置信息。

```
struct Complex_{
    double real;
    double imagin;
};
typedef struct Complex_ Complex;
```

复数数据结构，x 为实部，y 为虚部

int isBase2(int size_n);

检测 size_n 是否是 2 的整数次幂

输入：

Size_n:整数输入

输出：

如果 size_n 是 2 的整数次幂则返回 1 否则返回 0

void Add_Complex(Complex * src1,Complex *src2,Complex *dst);

复数加法

输入：

Src1：输入复数 1

Src2：输入复数 2

输出：

Dst：输出结果复数

void Sub_Complex(Complex * src1,Complex *src2,Complex *dst);

复数减法

输入：

Src1：输入复数 1

Src2：输入复数 2

输出：

Dst：输出结果复数

void Multy_Complex(Complex * src1,Complex *src2,Complex *dst);

复数乘法

输入：

Src1：输入复数 1

Src2：输入复数 2

输出：

Dst：输出结果复数

void Copy_Complex(Complex * src,Complex *dst);

复制复数，赋值操作
输入：
Src：输入复数
输出：
Dst：输出复数

void Show_Complex(Complex * src,int size_n)

显示复数，打印复数矩阵
输入：
Src：复数矩阵
Size_n：矩阵大小
输出：
屏幕打印复数矩阵。

double Distance(double x,double y,double c_x,double c_y);

计算距离，欧氏距离，亚像素点间的直线距离。
输入：
X：目标点坐标 x
Y：目标点坐标 y
C_x：原点坐标 x
C_y：原点坐标 y
输出：
返回欧式距离，双精浮点型

void matrixAdd(double *src1,double *src2,double *dst,int width,int height);

矩阵加法

输入：

Src1：输入矩阵 1

Src2：输入矩阵 2

Width：输入输出矩阵宽度

Height：输入输出矩阵高度

输出：

Dst：矩阵加法结果

void matrixSub(double *src1,double *src2,double *dst,int width,int height);

矩阵减法，src1-src2

输入：

Src1：输入矩阵 1

Src2：输入矩阵 2

Width：输入输出矩阵宽度

Height：输入输出矩阵高度

输出：

Dst：矩阵减法结果

void matrixCopy(double *src,double *dst,int width,int height);

矩阵复制，将 src 复制到等大小的 dst 中。

输入：

Src：矩阵

Width：输入输出矩阵宽度

Height：输入输出矩阵高度

输出：

Dst：复制结果矩阵

int matrixCopyLocal(double *src,double *dst,int width,int height,int w_width,int w_height,Position *lefttoppoint);

矩阵局部复制，将 src 矩阵指定位置开始的指定大小的区域，复制到目标矩阵中。
输入：
Src：原始数据矩阵
Width：原始矩阵宽度
Height：原始矩阵高度
W_wdith：目标矩阵宽度
W_height：目标矩阵高度
Lefttoppoint：左上方坐标点，即起始坐标点，向右下方扩展
输出：
Dst：从原图像矩阵中复制出来的部分。

void matrixMultreal(double *src,double *dst,double k,int width,int height);

矩阵乘以实数 k，矩阵中每个元素都乘以实数 k。
输入：
Src：矩阵
Width：输入输出矩阵宽度
Height：输入输出矩阵高度
K：参数
输出：
Dst：计算结果矩阵

void matrixMul_matrix(double *src1,double *src2,double *dst,int width,int height);

矩阵点乘矩阵，两个矩阵中每个对位元素相乘。
输入：

Src1：矩阵 1
Src2：矩阵 2
Width：输入输出矩阵宽度
Height：输入输出矩阵高度
输出：
Dst：计算结果矩阵

void matrixDBL2INT(double *src,int *dst,int width,int height);

矩阵数据类型变换，从双精浮点型到整形的转换。
输入：
Src：输入浮点型矩阵
Width：输入和输出矩阵的宽度
Height：输入和输出矩阵的高度
输出：
Dst：整形矩阵

double matrixMean(double *mat,int width,int height);

矩阵所有元素的均值
输入：
Mat：输入浮点型矩阵
Width：输入和输出矩阵的宽度
Height：输入和输出矩阵的高度
输出：
返回矩阵元素的均值。

void matrixRotation(double *src,double *dst,int s_width,int s_height,int d_width,int d_height,double theta,Position_DBL* center);

矩阵旋转按照角度 theta（0°~359°）进行旋转，旋转中心为亚像素级别的 center，目标矩阵和原始矩阵大小可以不同，进行缩放和旋转差值的时候使用双线性差值，

注意旋转坐标系为自然右手坐标系，而不是图像的坐标系。

输入：

Src：输入图像矩阵

S_width：输入图像矩阵宽度

S_height：输入图像矩阵高度

D_width：目标图像矩阵宽度

D_height：目标图像矩阵高度

Theta：旋转角度（0°~359°）

Center：旋转中心，亚像素级坐标点

输出：

Dst：旋转后的矩阵

void matrixIntegral(double * src,double *dst,int width,int height);

图像积分，积分结果 dst（x0，y0）处的值为 src 中 x<=x0 且 y<=y0 的所有像素值的和。

输入：

Src：输入图像矩阵

Width：输入输出图像矩阵的宽度

Height：输入输出图像矩阵的高度

输出：

Dst：图像矩阵的积分结果

void matrixE(double *src,int width,int height);

长宽相等的图像矩阵主对角线置 1，即线性代数中的矩阵 E，其他位置元素为 0，多用于矩阵初始化。

输入：

Src：输入矩阵

Width：矩阵宽度

Height：矩阵高度

输出：

Src：输出 E

void matrixEigen_Jacobi(double * src,double *EigenValue,double *EigenVector,double threshold,int width,int height);

对称矩阵雅可比算法求矩阵的特征值和特征向量，用 threshold 来控制计算结果的精确度。

输入：

Src：输入对称矩阵

Width：输入矩阵和特征向量矩阵的宽

Height：输入矩阵和特征向量矩阵的高，Height 和 Width 相等

Threshold：结果精确度阈值。

输出：

EigenValue：特征值的向量形式，有 Width 个

EigenVector：特征值对应的特征向量，如果为 NULL 则不计算次结果

void matrixMulmatrix(double *src1,double *src2,double *dst,int width,int height);

矩阵乘矩阵。

输入：

Src1：矩阵 1

Src2：矩阵 2

Width：输入输出矩阵宽度

Height：输入输出矩阵高度

输出：

Dst：乘法结果

void matrixTranspose(double *src,double *dst,int width,int height);

伴随矩阵，输入矩阵的宽等于输出矩阵的高，输入矩阵的高等于输出矩阵的宽。

输入：

Src：输入图像矩阵

Width：输入矩阵宽，输出矩阵高
Height：输入矩阵高，输出矩阵宽
输出：
Dst：转置结果

void matrixCovariance(double *src,double *dst,int width,int height);

协方差矩阵，输入矩阵，求其协方差矩阵，用于 PCA 等算法中，列为特征值，行为样本组，求每组的均值，后计算协方差。
输入：
Src：输入矩阵
Width：输入矩阵的宽
Height：输入矩阵的高，输出矩阵的宽和高
输出：
Dst：输出协方差矩阵。

double findMatrixMax(double *src,int width,int height,Position *max_posion);

寻找矩阵最大值，及其位置。
输入：
Src：输入图像矩阵
Width：输入矩阵的宽
Height：输入矩阵的高
输出：
函数返回最大值
Max_posion：最大值位置点坐标。

double findMatrixMin(double *src,int width,int height,Position *min_posion);

寻找矩阵最小值，及其位置。

输入：

Src：输入图像矩阵

Width：输入矩阵的宽

Height：输入矩阵的高

输出：

函数返回最小值

Max_posion：最小值位置点坐标。

void MaxPix(double *src1 ,double *src2,double *dst,int width,int height);

找出两图像矩阵中同一位置中相对较大的像素值

输入：

Src1：输入矩阵 1

Src2：输入矩阵 2

Width：输入输出矩阵宽

Height：输入输出矩阵的高

输出：

Dst：src1 和 src2 对应位置处较大值形成的矩阵

void MinPix(double *src1 ,double *src2,double *dst,int width,int height);

找出两图像矩阵中同一位置中相对较小的像素值

输入：

Src1：输入矩阵 1

Src2：输入矩阵 2

Width：输入输出矩阵宽

Height：输入输出矩阵的高

输出：

Dst：src1 和 src2 对应位置处较小值形成的矩阵

void One(double *src,int width,int height);

矩阵全部元素置 1
输入：
Src：输入矩阵
Width：矩阵宽度
Height：矩阵高度
输出：
Src：矩阵元素全为 1

void Zero(double *src,int width,int height);

矩阵全部元素置 0
输入：
Src：输入矩阵
Width：矩阵宽度
Height：矩阵高度
输出：
Src：矩阵元素全为 0

void Mask(double *src,double *dst,double *mask,int width,int height);

矩阵掩模，mask 为模板，当 mask 为 255 时输出图像灰度为 src 对应位置的灰度，否则为 0；
输入：
Src：输入矩阵
Mask：输入模板矩阵
Width：输入输出矩阵的宽
Height：输入输出矩阵的高
输出：
Dst：输出 Mask 后的矩阵

void matrixOrdinaryDiff(double *src,double *range,double* angle,int width,int height);

矩阵一阶差分，返回梯度幅值和梯度方向。

输入：

Src：输入矩阵

Width：输入矩阵和输出矩阵的宽

Height：输入矩阵和输出矩阵的高

输出：

Rang：输出梯度幅度值

Angle：输出梯度角度（0°~360°）

二值图像形态学

void Translation(double *src,double *dst,int width,int height,MoveDirection *direction);

位移操作，将图像矩阵按照 direction 向量进行整体移动，如果超出边界的像素将会舍去。
输入：
Src：原始图像矩阵
Width：输入图像和输出图像矩阵的宽
Height：输入图像和输出图像矩阵的高
Direction：位移向量，像素级位置
输出：
Dst：位移后的图像矩阵

void Zoom(double *src,int s_width,int s_height,double *dst,int d_width,int d_height);

将小的图像嵌入比它大的黑色图像中间，等价于给图像加黑色边框。
输入：
Src：输入图像矩阵
S_width：输入图像矩阵的宽
S_height：输入图像矩阵的高
D_width：输出图像矩阵的宽
D_height：输出图像矩阵的高
输出：
Dst：嵌入处理后的图像

void And(double *src0,double *src1,double *dst,int width,int height);

逻辑“与”操作，矩阵元素 255 为真，其他为假。
输入：
Src0：输入矩阵 1
Src1：输入矩阵 2
Width：输入输出矩阵宽
Height：输入输出矩阵高
输出：
Dst：输出逻辑操作结果。

void Or(double *src0,double *src1,double *dst,int width,int height);

逻辑“或”操作，矩阵元素 255 为真，其他为假。
输入：
Src0：输入矩阵 1
Src1：输入矩阵 2
Width：输入输出矩阵宽
Height：输入输出矩阵高
输出：
Dst：输出逻辑操作结果。

void Not(double *src,double *dst,int width,int height);

取反操作，各元素等于 255 减去其本身。
输入：
Src：输入矩阵
Width：输入输出矩阵的宽
Height：输入输出矩阵的高
输出：
Dst：取反操作结果

void G_One(double *src,int width,int height);

将所有元素设为 255
输入：
Src：原始图像矩阵
Width：图像矩阵宽
Height：图像矩阵高

void Dilate(double *src,int s_width,int s_height,double *dst,int d_width,int d_height,double *se,int se_width,int se_height,Position *center);

二值图像形态学膨胀操作。
输入：
Src：输入图像矩阵
S_width 输入图像矩阵宽
S_height：输入图像矩阵高
D_width：输出图像矩阵宽
D_height：输出图像矩阵高
Se：结构元
Se_width：结构元宽
Se_height：结构元高
Center：膨胀结构元的中心位置坐标，像素级
输出：
Dst：膨胀结果图像矩阵

void Erode(double *src,int s_width,int s_height,double *dst,int d_width,int d_height,double *se,int se_width,int se_height,Position *center);

二值图像形态学腐蚀操作。
输入：
Src：输入图像矩阵

S_width 输入图像矩阵宽
S_height：输入图像矩阵高
D_width：输出图像矩阵宽
D_height：输出图像矩阵高
Se：结构元
Se_width：结构元宽
Se_height：结构元高
Center：腐蚀结构元的中心位置坐标，像素级
输出：
Dst：腐蚀结果图像矩阵

void Open(double *src,int width,int height,double *dst,double *se,int se_width,int se_height,Position *center);

二值图像开操作，先腐蚀后膨胀
输入：
Src：输入图像矩阵
Width：输入图像矩阵宽度
Height：输入图像矩阵高度
Se：结构元
Se_width：结构元宽
Se_height：结构元高
Center：结构元中心坐标，像素级别
输出：
Dst：开操作结果

void Close(double *src,int width,int height,double *dst,double *se,int se_width,int se_height,Position *center);

二值图像闭操作，先膨胀后腐蚀
输入：
Src：输入图像矩阵

Width：输入图像矩阵宽度
Height：输入图像矩阵高度
Se：结构元
Se_width：结构元宽
Se_height：结构元高
Center：结构元中心坐标，像素级别
输出：
Dst：闭操作结果

void HitorMiss(double *src,int width,int height,double *se1,int se1_width,int se1_height,double *se2,int se2_width,int se2_height,double *dst,Position *se1center,Position *se2center);

二值图像击中操作，用于检测特定的连通结构，先对结构元 1 进行腐蚀，然后对输入矩阵取反，再对结构元 2 进行腐蚀，两次结果取逻辑与。
输入：
Src：输入图像矩阵
Width：输入图像矩阵宽
Height：输入图像矩阵高
Se1：结构元 1,
Se1_width：结构元 1 的宽
Se1_height：结构元 1 的高
Se2:结构元 2
Se2_width：结构元 2 的宽
Se2_height：结构元 2 的高
Se1center：结构元 1 的中心位置，像素级
Se2center：结构元 2 的中心位置，像素级
输出：
Dst：输出集中结果矩阵

void BinaryEdge(double *src,int width,int height,double* dst);

二值图像边缘检测，先对图像进行腐蚀，然后用原图像矩阵减去腐蚀后结果，得到图像结构内边界

输入：

Src：输入图像矩阵

Width：输入输出图像矩阵宽

Height：输入输出图像矩阵高

输出：

Dst：边缘检测结果矩阵

void FillHole(double *src,double *dst,int width,int height,Position *seed);

孔洞填充操作，以种子点位置开始进行测地膨胀，用原图取反作为参考地。

输入：

Src：输入图像矩阵

Width：输入输出图像矩阵宽度

Height：输入输出图像矩阵高度

Seed：种子点，像素级别

输出

Dst：填充结果矩阵

void GetConCompG_Onent(double *src,double *dst,int width,int height,Position *seed);

在二值图像矩阵中，获取包含种子点的连通分量。

Src：输入图像矩阵

Width：输入输出图像矩阵宽度

Height：输入输出图像矩阵高度

Seed：种子点，像素级

输出

Dst：包含种子点的连通分量

void FrameWork(double *src,double *dst,int width,int height,double *se,int se_width,int se_height);

获取二值图像的骨架，注意，此方法获得的骨架宽度不一定为 1，可以经过细化操作获取宽度为 1 的骨架，具体方法：腐蚀后图像 A 减去腐蚀后再开操作（A 的开操作）的结果。

输入：

Src：输入图像矩阵

Width：输入输出图像矩阵的宽度

Height：输入输出图像矩阵的高度

输出：

Dst：输出骨架结构

void Convexhull(double *src,double *dst,int width,int height);

凸壳操作，将填充二值图像中的凹陷部分形成凸壳结构

输入：

Src：输入图像矩阵

Width：输入输出图像矩阵的宽度

Height：输入输出图像矩阵的高度

输出：

Dst：输出凸壳结构

void Thinning(double *src,double *dst,int width,int height);

细化操作，使用预设定的结构元对二值图像进行细化，最后得到不改变连通结构但宽度为 1 的结构

输入：

Src：输入二值图像矩阵

Width：输入输出二值图像矩阵宽度
Height：输入输出二值图像矩阵高度
输出：
Dst：输出细化后的二值图像。

void reBuildOpen(double *src,double *dst,double *ground,int width,int height,double *dilateSE,int dse_width,int dse_height,double *erodeSE,int ese_width,int ese_height,int eroden);

二值图像重建开操作，先使用指定模式的结构元 1 进行指定次数的腐蚀，然后再使用结构元 2 对结果进行测地（原图）膨胀得到含有结构元 1 模式的部分
输入：
Src：输入二值图像矩阵
Ground：输入参考地（多为 src，也有特殊情况）
Width：二值图像矩阵宽度
Height：而知图像矩阵高度
DilateSE：膨胀结构元
Dse_width：膨胀结构元宽度
Dse_height：膨胀结构元高度
ErodSE：腐蚀结构元
Ese_width：腐蚀结构元宽度
Ese_height：腐蚀结构元高度
Eroden：腐蚀次数
输出：
Dst：输出重建开操作结果

主成分分析（开发中）

void PCA(double *src,int width,int height,double *dst,int feature_num);

主成分析，开发完善中

输入：

Src：特征矩阵

Width：特征矩阵宽度

Height：特征矩阵高度

Feature_num：主特征数目

输出：

Dst：主成分分析结果

伪彩色处理

GRAY2HSV	1.23046875
HSVMAX	315.0
HIGHVALUE_EQU_RED	1
LOWVALUE_EQU_RED	0

void Gray2Color(double *src,RGB* dst,int width,int height,int type);

灰度图像伪彩化，将灰度图像[0,255] 映射到 HSV 空间中 V 通道的 0°~270°，并设置 H 通道和 S 通道为最大值 1.0

输入：

Src：输入灰度图像

Width：输入和输出图像矩阵宽度

Height：输入和输出图像矩阵的高度

Type：

紫色对应高灰度值—0

红色对应高灰度值—1

输出：

Dst：伪彩图像。

区域分割

void RegionGrow(double *src,double *dst,Position * position,int p_size,int width,int height,double param);

区域生长，从种子（可以是一个或多个）点开始，符合条件的点将向外生长，直到所有符合条件的点都包含到区域之中。此算法的“条件”是种子点的差的绝对值小于 param。

输入：

Src：原始图像矩阵

Position：种子点位置数组

P_size：种子点数组大小

Width：输入输出矩阵宽

Height：输入输出矩阵高

输出：

Dst：分割图像矩阵。

void RegionSplit(double *src,double *dst,int width,int height,double mean_param1,double mean_param2,double variance_param1,double variance_param2);

区域分裂算法，将图像按照区域分裂，并判断当前区域是否满足一定的条件 A，满足设为目标，否则将区域继续分割，递归至最小值分割值。

输入：

Src：输入图像矩阵

Width：输入输出矩阵宽

Height：输入输出矩阵高

Mean_param1：矩阵区域均值参数下限

Mean_param2：矩阵区域均值参数上限

Variance_param1：矩阵区域方差参数下限

Variance_param2：矩阵区域方差参数上限

输出：

Dst：分割结果矩阵

图像缩放

void Resize(double *src,int s_width,int s_height,double *dst,int d_width,int d_height);

图像矩阵缩放，使用双线性插值。

输入：

Src：输入图像矩阵

S_width：输入图像矩阵宽

S_height：输入图像矩阵高

D_width：输出图像矩阵宽

D_height：输出图像矩阵高

输出：

Dst：输出图像矩阵

SIFT 特征提取

```
struct SIFT_Feature_{
    double x;
    double y;
    double scale;
    double orientation;
    int des_vector[128];
    struct SIFT_Feature_ *next;
};
```

SIFT 特征点存储结构，x，y 为特征点位置坐标，亚像素级别，scale 特征点所在尺度，orientation 特征点方向，des_vector 特征点 128 维描述子

void ScaleSpace(double *src,double * dst[],double *delta_arry,int width,int height,double delta_max,int k);

尺度空间变换，将输入图像按照所给定的尺度进行高斯处理，得到一个矩阵数组，其数量由 k 决定，delta_max 决定最大的尺度，其他尺度按照此最大尺度开 n 次方决定（n<k）。

输入：

Src：输入图像矩阵

Width：输入输出图像矩阵宽

Height：输入输出图像矩阵高

Delta_max：输入图像最高层尺度

K：层数

输出：

Dst[]：指针数组，每个指针指向对应层的矩阵

Delta_arry：每层对应的 delta 值

void ReleaseMatArr(double * dst[],int k);

释放尺度空间矩阵数组（包含多个矩阵的数组，数组为指针数组，每个指针指向一个矩阵）
输入：
Dst[]：指向多层矩阵的指针数组
K：层数

void DOG_Scale(double *src[],double * dst[],int width,int height,int k);

计算尺度空间差分，DoG，结果层数比尺度空间少一层。
输入：
Src[]：多层尺度空间
Width：输入输出矩阵宽度
Height：输入输出矩阵高度
K：尺度空间层数（DoG 层数为 k-1）
输出：
Dst[]：输出尺度空间差分结果（k-1 层）

void SIFT(double *src,SIFT_Feature **dst,int width,int height,int scale_k,int octave);

SIFT 特征提取函数，设定每组尺度空间层数 scale_k，设定对应金字塔层数 octave，返回得到特征点数据链表。
输入：
Src：输入图像矩阵
Width：图像矩阵宽度
Height：图像矩阵高度
Scale_k：尺度空间层数
Octave：金字塔层数
输出：
Dst：特征点链表

void ReleaseSIFTlist(SIFT_Feature * head);

释放 SIFT 特征点存储链表

输入：

Head：SIFT 特征链表首节点。

分割

SHARPEN_LAP_0	0
SHARPEN_LAP_1	1
SHARPEN_LAP_2	2
SHARPEN_LAP_3	3
LAPLACE_MASK_SIZE	3
SOBEL_MASK_SIZE	3
ROBERT_MASK_SIZE	3
EDGE_DETECTOR_ROBERT	1
EDGE_DETECTOR_PREWITT	2
EDGE_DETECTOR_KIRSCH	3
EDGE_DETECTOR_SOBEL	4
EDGE_DETECTOR_LOG	5
EDGE_DETECTOR_DOG	6
EDGE_DETECTOR_LAPLACE	7
EDGE_DETECTOR_CANNY	8

double Sobel(double *src,double *dst,double *edgedriction,int width,int height,int sobel_size);

Sobel 算子，使用 Sobel 算子与图像矩阵卷积，得到梯度幅值和梯度方向。

输入：

Src：输入图像矩阵

Width：输入输出图像矩阵宽度

Height：输入输出图像矩阵高度

Sobel_size：Sobel 算子尺寸，3,5 或 7

输出：

Dst：输出梯度幅值

Edgedriction：图像梯度方向矩阵

double Scharr(double *src,double *dst,double *edgedriction,int width,int height);

Scharr 算子，使用 Scharr 算子与图像矩阵卷积，得到梯度幅值和梯度方向。
输入：
Src：输入图像矩阵
Width：输入输出图像矩阵宽度
Height：输入输出图像矩阵高度
输出：
Dst：输出梯度幅值
Edgedriction：图像梯度方向矩阵

double Robert(double *src,double *dst,double *edgedriction,int width,int height);

Robert 算子，使用 Robert 算子与图像矩阵卷积，得到梯度幅值和梯度方向。
输入：
Src：输入图像矩阵
Width：输入输出图像矩阵宽度
Height：输入输出图像矩阵高度
输出：
Dst：输出梯度幅值
Edgedriction：图像梯度方向矩阵

double Prewitt(double *src,double *dst,int width,int height);

Prewitt 算子，使用 Prewitt 算子与图像矩阵卷积，得到梯度幅值。
输入：
Src：输入图像矩阵
Width：输入输出图像矩阵宽度
Height：输入输出图像矩阵高度

输出：
Dst：输出梯度幅值

double Kirsch(double *src,double *dst,int width,int height);

Kirsch 算子，使用 Kirsch 算子与图像矩阵卷积，得到梯度幅值，返回方向，只有八个方向。
输入：
Src：输入图像矩阵
Width：输入输出图像矩阵宽度
Height：输入输出图像矩阵高度
输出：
Dst：输出梯度幅值
返回方向代码

double Laplace(double *src,double *dst,int width,int height);

拉普拉斯算子，计算图像矩阵的二阶差分
输入：
Src：输入图像矩阵
Width：输入输出图像矩阵宽度
Height：输入输出图像矩阵高度
输出：
Dst：二阶差分结果

void Canny(double *src,double *dst,int width,int height,int sobel_size,double threshold1,double threshold2);

Canny 算法边缘检测
输入：
Src：输入图像矩阵
Width：输入图像矩阵宽度

Height：输入图像矩阵高度
Sobel_size：sobel 算子大小（3,5 或 7）
Threshold1：低阈值
Threshold2：高阈值
输出：
Dst：二值化边缘结果

double LoG(double *src,double *dst,int width,int height,int m_width,int m_height,double deta,double threshold);

LoG 算子，即对高斯模板进行拉普拉斯差分后得到的 LoG 模板，对图像矩阵进行卷积。阈值为对零交叉处的检验，由于噪声影响，应设定合适的阈值来拒绝噪声的 LoG 相应。
输入：
Src：输入图像矩阵
Width：图像矩阵的宽
Height：图像矩阵的高
M_width：高斯窗口宽度
M_height：高斯窗口高度
Deta：高斯模板标准差
Threshold：零交叉阈值
输出
Dst：输出边缘二值图像

double DoG(double *src,double *dst,int width,int height,int m_width,int m_height,double delta,double threshold);

高斯差分算子 DoG 通过调整两个高斯算子的标准差比例来近似 log 算子，对图像进行卷积，提取细节
输入：
Src：输入图像矩阵
Width：图像矩阵的宽

Height：图像矩阵的高
M_width：高斯窗口宽度
M_height：高斯窗口高度
Deta：高斯模板标准差
Threshold：零交叉阈值
输出
Dst：输出边缘二值图像

void EdgeDetection(double *src,double *dst,int width,int height,int detector,double threshold,int m_width,int m_height,double deta);

边缘检测综合函数接口，可供调用多种方法。
输入：
Src：输入图像矩阵
Width：输入图像矩阵宽度
Height：输入图像矩阵高度
Detector：有一下算子供选择
EDGE_DETECTOR_ROBERT，EDGE_DETECTOR_PREWITT，EDGE_DETECTOR_SOBEL，EDGE_DETECTOR_KIRSCH，EDGE_DETECTOR_LOG，EDGE_DETECTOR_LAPLACE:
Threshold：经过上述算子检测后进行阈值处理，低于阈值的边缘将被放弃
M_width：算子模板宽度
M_height：算子模板高度
Deta：部分用到高斯函数的标准差
输出：
Dst：输出边缘检测结果二值图像

void getEdgeAngle(double *src_x,double *src_y,double *edgeAngle,int width,int height);

计算差分后的图像矩阵梯度角度。
Src_x：x 向差分矩阵
Src_y：y 向差分矩阵

Width：输入图像宽度

Height：输入图像高度

输出：

edgeAngel：输出边缘梯度方向

void getV_HBoundary(double *src,double *dst,int width,int height,double v_threshold,double a_threshold,int isVertical);

检测垂直或水平边缘。

输入：

Src：输入图像矩阵

Width：输入输出图像矩阵宽度

Height：输入输出图像矩阵高度

V_threshold：幅度阈值，小于此阈值的幅值将被舍去。

A_threshold：角度阈值，在该范围内的角度误差将被接受，超过次阈值将被舍弃

isVertical：选择垂直和水平方向，1 表示垂直，其他为水平

锐化

SHARPEN_LAPLACE	0
SHARPEN_SOBEL	1
SHARPEN_ROBERT	2

void LaplaceSharpen(double *src,double *dst,int width,int height,double c);

拉普拉斯锐化模型，将图像通过拉普拉算子进行检测得到的细节矩阵加权后与原图像相加

输入：

Src：输入图像矩阵

Width：输入图像矩阵宽度

Height：输入图像矩阵高度

C：检测细节后加权系数

输出：

Dst：输出锐化后图像

void UnsharpMasking(double *src,double *dst,int width,int height,int smooth_type,int smooth_mask_width,int smooth_mask_height,double gaussian_deta,double k);

非锐化掩蔽，首先对图像进行平滑，通过原图像减去平滑后的图像，得到细节，并将细节矩阵加权后加入原图，完成锐化效果

Src：输入图像矩阵

Width：输入图像矩阵宽度

Height：输入图像矩阵高度

Smooth_type：平滑方式 SMOOTH_GAUSSIAN，SMOOTH_MEAN

Smooth_mask_width：平滑模板宽度

Smooth_mask_height：平滑模板高度

gaussian_deta：高斯模板标准差

输出：

Dst：输出锐化后图像

void SobelSharpen(double *src,double *dst,int width,int height,double c);

Sobel 锐化模型，将图像通过 Sobel 算子进行检测得到的细节矩阵加权后与原图像相加

输入：

Src：输入图像矩阵

Width：输入图像矩阵宽度

Height：输入图像矩阵高度

C：检测细节后加权系数

输出：

Dst：输出锐化后图像

void RobertSharpen(double *src,double *dst,int width,int height,double c);

罗伯特锐化模型，将图像通过罗伯特算子进行检测得到的细节矩阵加权后与原图像相加

输入：

Src：输入图像矩阵

Width：输入图像矩阵宽度

Height：输入图像矩阵高度

C：检测细节后加权系数

输出：

Dst：输出锐化后图像

平滑

SMOOTH_GAUSSIAN	0
SMOOTH_MEDIAN	1
SMOOTH_BILATERAL	2
SMOOTH_MEAN	3
NLMF_FUN_A	0
NLMF_FUN_H	1
NLMF_FUN_G	2
NLMF_MASK_M	0
NLMF_MASK_G	1

void MedianFilter(double *src,double *dst,int width,int height,int m_width,int m_height);

中值滤波，主要用于减少椒盐噪声

输入：

Src：输入图像矩阵

Width：输入图像矩阵宽度

Height：输入图像矩阵高度

M_width：滤波器窗宽

M_height：滤波器窗高

输出：

Dst：平滑后结果

void GaussianMask(double *mask,int width,int height,double deta);

生成高斯模板

输入：

Mask：模板矩阵

Width：矩阵宽度

Height：矩阵高度

Deta：高斯标准差参数

void GaussianFilter(double *src,double *dst,int width,int height,int m_width,int m_height,double deta);

高斯滤波
输入：
Src：输入图像矩阵
Width：输入图像矩阵宽度
Height：输入图像矩阵高度
M_width：滤波器窗宽
M_height：滤波器窗高
Deta：高斯函数标准差
输出：
Dst：平滑后结果

void MeanFilter(double *src,double *dst,int width,int height,int m_width,int m_height);

均值滤波
输入：
Src：输入图像矩阵
Width：输入图像矩阵宽度
Height：输入图像矩阵高度
M_width：滤波器窗宽
M_height：滤波器窗高
输出：
Dst：平滑后结果

void BilateralFilter(double *src,double *dst,int width,int height,int m_width,int m_height,double deta_d,double deta_r);

双边滤波器，能够在平滑图像的同时较好的保存细节
输入：
Src：输入图像矩阵
Width：输入图像矩阵宽度
Height：输入图像矩阵高度
M_width：滤波器窗宽
M_height：滤波器窗高
Deta_d：距离高斯函数标准差
Deta_r：幅值高斯函数的标准差
输出：
Dst：平滑后结果

void MeanMask(double *mask,int width,int height);

生成均值模板
输入：
Mask：模板矩阵
Width：矩阵宽度
Height：矩阵高度

void MeanFilter(double *src,double *dst,int width,int height,int m_width,int m_height);

均值滤波
输入：
Src：输入图像矩阵
Width：输入图像矩阵宽度
Height：输入图像矩阵高度

M_width：滤波器窗宽

M_height：滤波器窗高

输出：

Dst：平滑后结果

阈值处理

阈值处理类型表

THRESHOLD_TYPE1	dst(x,y)=src(x,y)>T?src(x,y):Minvalue;
THRESHOLD_TYPE2	dst(x,y)=src(x,y)>T?Maxvalue:src(x,y);
THRESHOLD_TYPE3	dst(x,y)=src(x,y)>T?Maxvalue:Minvalue;
THRESHOLD_TYPE4	dst(x,y)=src(x,y)>T?Minvalue:Maxvalue;

void Threshold(double *src,double *dst,int width,int height,double threshold,int type);

阈值处理
输入：
Src：输入图像矩阵
Width：图像矩阵宽度
Height：图像矩阵高度
Threshold：阈值
Type：阈值处理类型，见本章开头处阈值处理类型表
输出：
Dst：阈值处理后图像矩阵

void MeanThreshold(double *src,double *dst,int width,int height,int type);

平均阈值处理，阈值通过所有矩阵元素的均值产生
输入：
Src：输入图像矩阵
Width：图像矩阵宽度
Height：图像矩阵高度
Type：阈值处理类型，见本章开头处阈值处理类型表

输出：

Dst：阈值处理后图像矩阵

void PtileThreshold(double *src,double *dst,double p_value,int width,int height,int type);

平均阈值处理，阈值通过所有矩阵元素的 p 分位数产生

输入：

Src：输入图像矩阵

P_value：p 分位数（0,1）

Width：图像矩阵宽度

Height：图像矩阵高度

Type：阈值处理类型，见本章开头处阈值处理类型表

输出：

Dst：阈值处理后图像矩阵

void IterativeThreshold(double *src,double *dst,double deta_t,int width,int height,int type);

迭代法求阈值，初始化一个阈值将直方图分为两部分，求出两部分的均值，这两个均值的均值为新的阈值，迭代这些步骤，deta_t 精确度，当迭代n次以后阈值tn与第n-1次迭代结果tn-1相差小于deta_t时，迭代停止。

输入：

Src：输入图像矩阵

Deta_t：精确度

Width：图像矩阵宽度

Height：图像矩阵高度

Type：阈值处理类型，见本章开头处阈值处理类型表

输出：

Dst：阈值处理后图像矩阵

void ValleyBottomThreshold(double *src,double *dst,int width,int height,int type);

双峰型直方图，经过直方图平滑后呈现出双峰后找出谷底，以此值为阈值划分灰度值，平滑直方图采用1/4[1 2 1]的模板，判断是否是双峰采用1阶微分，判断正负性，直方图非双峰不能使用该方法。
输入：
Src：输入图像矩阵
Width：图像矩阵宽度
Height：图像矩阵高度
Type：阈值处理类型，见本章开头处阈值处理类型表
输出：
Dst：阈值处理后图像矩阵

void Hist_int2double(int *hist,double *hist_d);

直方图从整型变换为双精度浮点型，为平滑直方图准备数据
输入
Hist：整型直方图
输出
Hist_d：浮点型直方图

void MeanDoubleHumpThreshold(double *src,double *dst,int width,int height,int type);

与谷底类似，针对双峰直方图，求直方图峰顶位置的均值得到分割阈值，进行阈值处理。
输入：
Src：输入图像矩阵
Width：图像矩阵宽度
Height：图像矩阵高度

Type：阈值处理类型，见本章开头处阈值处理类型表
输出：
Dst：阈值处理后图像矩阵

void OTSUThreshold(double *src,double *dst,int width,int height,int type);

OTSU 法求最佳阈值，算法使用贝叶斯分类原理得到最好分类，通过阈值将灰度分为两类，计算均值和方差，通过给定计算方式评估分类好还，找出最优阈值。
输入：
Src：输入图像矩阵
Width：图像矩阵宽度
Height：图像矩阵高度
Type：阈值处理类型，见本章开头处阈值处理类型表
输出：
Dst：阈值处理后图像矩阵

void SobelThreshold(double *src,double *dst,int width,int height,double threshold,int type);

对于小物体大背景，或者大物体小背景，使用边缘作为掩膜得到小面积直方图，计算阈值
输入：
Src：输入图像矩阵
Width：图像矩阵宽度
Height：图像矩阵高度
Type：阈值处理类型，见本章开头处阈值处理类型表
输出：
Dst：阈值处理后图像矩阵

分水岭算法

void MeyerWatershed(double *src,double *dst,int width,int height);

Meyer 分水岭分割算法
输入
Src：输入图像矩阵
Width：图像矩阵宽度
Height：图像矩阵高度
输出：
Dst：分割结果图像矩阵

部分算法说明