QuantIm

C/C++ Library for Scientific image processing

Version 4.01β 20.10.2008

Legal Considerations

QuantIm is free software; which means that you are free to use it and to redistribute free and verbatim copies of the source code. QuantIm is not in the public domain, however, it is protected by copyright.

• You are free to modify the source of QuantIm for your own, private use.

• You may distribute modified, non-commercial versions which retain the name "QuantIm", provided that (i) you indicate these modifications clearly in the file "quantim4.h" under "modifications" as well as at the beginning of every single file that you modified or added, (ii) you distribute it under the same legal terms as QuantIm is distributed, and (iii) you include the copyright notice and the first paragraph of the file "quantim4.h" without any change.

• You may incorporate parts or all of QuantIm into your own non-commercial software and distribute it, provided that (i) you incorporate the QuantIm copyright notice at a prominent place in your software and documentation, and (ii) you distribute it under the same legal terms as QuantIm is distributed.

• You are not allowed to include QuantIm nor any part of it in any product that is sold commercially, i.e. whose cost exceeds that of media, shipping and handling, without prior written consent of Hans-Jörg Vogel.

QuantIm is licensed free of charge. Therefore all the files which in their entirety constitute QuantIm, are provided "as is", without warranty of any kind, whether expressed or implied. You are responsible for ascertaining the fitness of QuantIm for any specific use, and consequently you assume all the responsibilities and cost that may arise from using it.

Hans-Jörg Vogel
UFZ - Helmhotz Center for Environmental Research
Department Soil Physics
Theodor-Lieser-Str. 4
D-06120 Halle
GERMANY

e-mail: `hans-joerg.vogel@ufz.de`

1 Introduction and concepts

QuantIm was created for analyzing 2-dimensional and 3-dimensional images. When using commercial or free software, it is often hard to understand what is going on exactly when pushing buttons which is unacceptable from a scientific point of view and is why the QuantIm project was started. An other motivation for the ongoing development of QuantIm is that it can be used on any computer system, the only requirement is a C/C++ compiler. There is no fancy GUI so you will need some basic knowledge of C/C++ - which actually is not too difficult.
To communicate with the rest of the world, the input and output format of QuantIm is TIFF which is the most common bitmap format. This is established in QuantIm by using the standard library `libtiff` and the development files `libtiff-dev` which are available for any computer platform using a standard package manager or an external file server. It should be installed somewhere in the search path of your computer. 3D-images are handled by QuantIm as grey level or binary data using an own format but can be converted to vgi or dx (data explorer). The only restriction for the size of images is the RAM of your system. The few routines to produce eps-graphics are based on the excellent library `PSgraf3` of Kurt Roth which which is automatically installed alongside QuantIm.

The main features of QuantIm are:

- Algorithms for 2D/3D grey scale image processing for image filtering and segmentation.
- Algorithms for the quantification of 2D/3D binary images, including methods of mathematical morphology, topological analysis and image intersection.
- Tools to generate 2D/3D random fields with predefined properties.

Contributors: Thanks to Uli Weller and Steffen Schlüter for adding some highly useful functions.

2 Installation and Use in a LINUX/UNIX Environment

To use the functions of QuantIm in your own programs, you may directly include the corresponding source files. The preferable way, however, is to link the QuantIm library to your programs.

2.1 Creating the QuantIm Library

- Copy the directory `quantim4_distribution.tgz` from the distribution medium to a convenient place in your file system.
- Extract the archive using
  > `tar -xvzf quantim4_distribution.tgz`
- Make sure that the `libtiff` and `libtiff-dev` packages are installed. These are standard package that you can retrieve from your prefered package manager or directly in a terminal (Debian or Ubuntu) by:
  > `sudo apt-get install libtiff4 libtiff4-dev`
- Create the library by
  > `cd quantim_distribution`
  > `sh install.sh`
  This will create the file `libQuantim4.a` in your source directory and move the library `libQuantim4.a` and the header file `quantim4.h` to the standard search path directories `/usr/local/lib` and `/usr/local/include`. This step requires root privileges.
- Good luck!

2.2 Using QuantIm

To use QuantIm include the header `quantim4.h` into your program. Then, if the library `libQuantim4.a` has been copied to a place in the search path of the compiler, compile the program with the option `-lQuantim4 -ltiff -lPSgraf3 -lm`, for example

```
g++ -o program myProgram.c -lQuantim4 -ltiff -lPSgraf3 -lm
```

If the compiler cannot find the library and/or the header file, you must include the paths explicitly. Assuming that you have copied the library to `/a/b/c/d`, and the header file to `/x/y` the program can be compiled as

```
g++ myProgram.c -L/a/b/c/d -I/x/y -lQuantim4 -ltiff -lPSgraf3 -lm
```

2.3 New features of the actual QuantIm version

QuantIm4 has been modified substantially so that programs written for previous versions may not compile with this version of QuantIm. This has been sacrificed for a more intuitive structure of image variables, the possibility to handle 16-bit images and the usage of the same functions for different image types (2D, 3D, grey scale, color-rgb). Especially `GetImage` and `SaveImage` is now `LoadImage` and `StoreImage`, while most of the old function are still included.

3 Reference Manual

3.1 Basic image structure

All images are identified by a structure of variables defined in `quantim4.h`:

```
typedef struct
{
char name[127];
char itype[5];
unsigned char *pix;
unsigned int ndim;
unsigned int nchannel;
unsigned int *dim;
double *res;
unsigned int numbits;
unsigned int numbytes;
} image_cc;
```

`name`	Image name without any extension.
`itype`	Image type `gry`=grey scale, `gry16`=16bit/pixel grey scale, `rgb`=RGB-image, `ddd`= 3D grey scale, `ddd16`= 3D grey scale 16bit/pixel, `btd`= 3D binary, `doub`= double precision 2D (only for internal use), `doub3`= double precision 3D (only for internal use).
`*pix`	Pointer to the image data stored sequentially row by row.
`ndim`	Number of dimensions.
`nchannel`	Number of channels per pixel.
`*dim`	Array of `ndim` elements containing the size of the image in each dimension.
`*res`	Array of `ndim` elements containing the size of a pixel in each dimension.
`numbits`	Number of bits per pixel.
`numbytes`	Number of bytes per pixel.

Note that QuantIm can only handle data in little Endian byte order (Intel byte order), i.e. where the most significant bit arrives last. Moreover, QuantIm assumes an index order in which x changes fastest. 16-bit data has to be unsigned, i.e. the gray values are only allowed to be in the range [0;65535].

3.2 Image types

QuantIm handles the following types of images:

- **2D grey-scale *.tif** 8 bits/pixels or 16 bits/pixels
- **2D binary *.tif** also 8 bit/pixels but the pixel values are typically only 0 (black) or 255 (white).

- **2D rgb-images *.tif** 3 channels á 8 bit/pixels, byte1=red, byte2=green, byte3=blue.
- **3D grey scale *.ddd** 8 bit/voxel [0,255] (internal format of QuantIm).
- **3D grey scale *.ddd16** 16 bit/voxel [0,255] (internal format of QuantIm).
- **3D grey scale *.raw** 16 bit/voxel `unsigned int` with *.vgi or *.mhd description file.
- **3D grey scale *.vol** 32 bit/voxel `float` with *.vgi description file. This type is internally converted to 16-bit integer.
- **3D binary *.btd** 1 bit/voxel [0,1] (internal format of QuantIm)

3.3 Basic image handling (all types)

image_cc *LoadImage(char *buf)
Loads images of any type.

`buf`	Name of the image to be loaded (with or without extension).
return value:	pointer to the loaded image.

image_cc *LoadMetaImage(char *buf)
Loads 3D images in the standard file format of ITK, where *.mhd contains the meta data and *.raw contains the binary block of raw data.

`buf`	Name of the image to be loaded (with or without .mhd extension).
return value:	pointer to the loaded image.

image_cc *LoadRaw(char *buf, int cols, int rows, int layers, int offset, int nbyte)
Loads 3D raw images, where the meta information is given by arguments. Little Endian byte order and unsigned data required.

`buf`	Name of the image to be loaded (with extension).
`cols`	Number of columns (voxels in x dimension).
`rows`	Number of rows (voxels in y dimension).
`layers`	Number of layers (voxels in z dimension).
`offset`	Byte offset to the first voxel, e.g. 64 for *.ddd images.
`nbytes`	Number of bytes per voxel (1 for 8-bit, 2 for 16-bit).
return value:	pointer to the loaded image.

void StoreImage(image_cc *im, char *buf)
Saves the images of any type.

`im`	Pointer to the image.

`buf`	Name of the image to be saved.

void StoreMetaImage(image_cc *im, char *buf)
void StoreMetaImage(image_cc *im)
Saves a 3D image of any type in ITK MetaImage format (*.mhd and *.raw). Images with btd format (1-bit) are converted into ddd images (8-bit).

`im`	Pointer to the image.
`buf`	Name of the image to be saved. When function is used with *buf, im-¿name is used as file name instead.

void StoreRaw(image_cc *im, char *buf)
void StoreRaw(image_cc *im)
Saves a 3D image of any type in raw format (*.mhd and *.raw). Images with btd format (1-bit) are converted into ddd images (8-bit). Important meta information is stored in a textfile with the same name.

`im`	Pointer to the image.
`buf`	Name of the image to be saved. When function is used with *buf, im-¿name is used as file name instead.

void SavePaletteImage(image_cc *im, char *buf)
Saves an 8-bit image (only 2D grey scale) to a Palette Color TIFF-file.

`im`	Pointer to the image.
`buf`	Name of the tiff-image to be saved.

void SavePrinciplePlanes(image_cc *im, char * name)
Saves the central 2D planes (xy, xz, yz) of a 3D image using the given `name`.

`im`	Pointer to the 3D image.
`name`	Name of resulting 2D images: 'plane_[xy,xz,yz]_name'

image_cc *InitImage(int x, int y, int val, char *type)
image_cc *InitImage(int x, int y, int z, int val, char *type)
image_cc *InitImage(image_cc *im, int val)
image_cc *InitImage(image_cc *im)
Initiates a new image and allocates the required memory.

`x,y,(z)`	Number of pixels in x, y, and z direction.
`val`	Initial value of each pixels.
`im`	image whose structure (dimensions aso) is copied.
`type`	Type of image (optional): `gry`=grey scale, `gry16`=16 bits grey scale, `rgb`=RGB-image, `ddd`= 3D grey scale, , `xxd`= multi-dimensional grey scale, `btd`= 3D binary. The default type is 8-bit grey scale for 2D and 3D.
return value:	Pointer to the new image.

image_cc *InitRandImage(int col, int row, double rx, double ry, double cx, double cy, int mode, double *cdf)

image_cc *InitRandImage(int col, int row, int dep, double rx, double ry, double rz, double cx, double cy, double cz, int mode, double *cdf)

Returns a random 2D or 3D image optionally with defined grey distribution function (equal distribution, Gauss or predefined by a given `cdf`), defined correlation length and defined correlation model (Gauss, Lorentz, Exponential, von Karman).

`col, row, dep`	dimensions of the generated image
`rx, ry, rz`	size of pixel in x, y, z
`cx, cy, cz`	correlation lengths (number of pixel) in x, y, z
`mode`	0: equal distribution without any correlation (cx,cy,cz and cdf have no meaning here) 1: equal grey distribution with gaussian correlation 2: predefined grey distribution (cdf) and gaussian correlation 3: Gaussian correlation (cdf have no meaning here) 4: Lorentz-Correlation model (cdf have no meaning here) 5: Exponential-Correlation model (cdf have no meaning here) 6: von Karman-Correlation model (cdf have no meaning here)
`*cdf`	cdf of grey levels
return value:	pointer to the generated image

image_cc *CopyImage(image_cc *im)

Makes a copy of an image.

`*im`	Pointer to the original image.
return value:	Pointer to the copy.

void DeleteImage(image_cc *im)

Delete an image and deallocate memory.

`im`	Pointer to the image.

int rPixel(unsigned long i, image_cc *im)
int rPixel(int x, int y, image_cc *im)
int rPixel(int x, int y, image_cc *im, unsigned char *value, unsigned char *value, unsigned char *value)
int rPixel(int x, int y, image_cc *im, unsigned char *value)
int rPixel(int x, int y, int z, image_cc *im)

Reads the value of a pixel.

`i`	offset of the pixel within the stored 1-D array of image data (for 2d and 3d images).
`x, y, z`	Coordinates of the pixel.
`im`	Pointer to the image.

`*value`	Address to which the pixel values are written. For RGB images a pointer to a 3-element array containing the RGB values is also possible.
return value:	Value of the pixel, or 1/3(R+G+B) for rgb-images.

void wPixel(unsigned long i, image_cc *im, int value)
void wPixel(int x, int y, image_cc *im, int value)
void wPixel(int x, int y, image_cc *im, int value, int value, int value)
void wPixel(int x, int y, image_cc *im, int *value)
void wPixel(int x, int y, int z, image_cc *im, int value)
Write a value to a pixel.

`i`	offset of the pixel within the stored 1-D array of image data (for 2d and 3d images).
`x, y, z`	Coordinates of the pixel.
`im`	Pointer to the image.
`value`	value(s) to be written, number of given values corresponds to the number of channels per pixel
`*value`	For RGB images a pointer to a 3 element array containing the RGB values.

void InvertImage(image_cc *im)
Inverts an image.

`im`	Pointer to the image.

void FlipImage(image_cc *im, int mode)
Flips an image (only 2D).

`im`	Pointer to the image.
`mode`	0=horizontal, 1= vertical flip.

void RotateImage(image_cc *im, int mode)
Rotates an image (only 2D) by 90° or 180°.

`im`	Pointer to the image.
`mode`	-1= 90° counter clock wise, 1= 90° clock wise, 0=180°.

void TurnImage(image_cc *im, double grad)
Rotates an image (only 2D) by a given angle.

`im`	Pointer to the image.
`grad`	angle in degrees.

image_cc *ImageSegment(image_cc *image, int ulx, int uly, int dx, int dy)
image_cc *ImageSegment(image_cc *image, int ulx, int uly, int ulz, int dx, int dy, int dz)
Cut a rectangular segment out of an image.

`image`	Pointer to the image.

`ulx,uly,(ulz)`	x,y,z coordinates of the upper left corner of the segment.
`dx,dy,(dz)`	size of the segment in x,y,z direction (number of pixels).
return value:	Pointer to the segment.

image_cc *GetPlane(image_cc *im, int plane, int mode)
Extracts a 2D plane from a 3D image

`im`	Pointer to the 3D image.
`plane`	x,y or z coordinate of the plane.
`mode`	direction of the plane 0=xy, 1=yz, 2=xz.
return value:	Pointer to the 2D image.

void SetFrame(image_cc *im, int value)
Writes the edges of an image to a certain value.

`*im`	Pointer to the image.
`value`	Value to be written at the edges

image_cc *ChangeResolution(image_cc *image, int newx, int newy)
Rescales an image to the new dimensions `newx` and `newy`. (Only for 2D images, rgb and grey).

`*image`	Pointer to the original image
`newx, newy`	new dimensions in x and y direction
return value:	pointer to a rescaled image

image_cc *Diff(image_cc *wnd1,image_cc *wnd2)
Returns an image of the absolute differences between two images.

`*wnd1,*wnd2`	Pointer to the original images of any type.
return value:	Pointer to the resulting image.

3.4 Image conversions

image_cc *bit16to8(image_cc *image)
Converts a 16-bit grey scale image to a 8-bit image

`*image`	Pointer to the original image.
return value:	Pointer to the resulting image.

image_cc *Btd2Ddd(image_cc *image)
Converts a btd-image (1 bit/voxel) to a ddd-image (1 byte/voxel).

`*image`	Pointer to the binary image
return value:	Pointer to the corresponding ddd-image

image_cc *RGBtoGray(image_cc *image)
Converts a RGB-3byte-color image to a 8bit-grey scale image.

`image`	Pointer to the RGB-image
return value:	pointer to the converted grey scale image.

void DDD2Dx(image_cc *image, char *buf);

Generates a description file for DX named toto.general. The file name (toto) which is provided through `buf` must be the same name as was used to save the ddd image using StoreImage(). Note that the image resolution should be set correctly.

`*image`	Pointer to the ddd-image
`*buf`	name of the ddd images (without extension)
return value:	no return value, a dx description file *.general is produced.

image_cc *Ddd2Btd(image_cc *im);

Converts a ddd-image to a btd-image. All non-zero voxels are set to 1 the others stay at 0.

`*image`	Pointer to the ddd-image
return value:	pointer to the resulting btd-image.

3.5 Image filtering

void Mean(image_cc *im, int size)
Mean filter using a squared window

`im`	Pointer to the image.
`size`	sidelength of squared filter: 2·size+1 pixel.

image_cc *MeanVar(image_cc *im, int size)
Same as Mean but the variance within the squared window is returned as a new image

`im`	Pointer to the image.
`size`	sidelength of squared filter: 2·size+1 pixel.
return value:	Pointer to the image of variances

void Gauss(image_cc *im, int size, double sig)
Gauss filter using a squared window with side length 2·`size`+1 pixel.

`im`	Pointer to the image.
`size`	Sidelength of squared filter: 2·size+1 pixel.
`sig`	Variance of Gauss filter.

void DiffGauss(image_cc *im, int size, double sig, double sig2)
Difference of Gaussian (DoG) filter using a squared window with side length 2·`size`+1 pixel. At the same time the image is smoothed by a Gauss filter with low `sig` while edges are enhanced by adding the difference to a strongly smoothed image `sig2` at each location (The difference is highest at edges).

`im`	Pointer to the image.
`size`	Sidelength of squared filter: 2·size+1 pixel.
`sig`	Small variance of Gauss filter.
`sig2`	Big variance of Gauss filter.

void UnsharpMask(image_cc *im, int size, double sig)
Edge enhancement filter using a squared window with side length 2·`size`+1 pixel. Same rationale as a Difference of Gaussian filter (see above) but here the local grey value difference due to convolution with Gaussian kernel of `sig` is directly added to the image. This is faster but may enhance noise at the same time.

`im`	Pointer to the image.
`size`	Sidelength of squared filter: 2·size+1 pixel.
`sig`	Variance of Gauss filter.

void MinMax(image_cc *im, double size, int mode)
Minimum-Maximum filter for grey scale images with a circular/spherical kernel where `size` is in length units of the image according to im− >res[0]. This filter corresponds to grey scale erosion/dilation

`im`	Pointer to the image.
`size`	Radius of the circular/sherical filter

`mode`	0 is minimum filter, else maximum.

void Luul(image_cc *im, double size)

Combined Minimum-Maximum filter: lower-upper-upper-lower for grey scale images with a circular/spherical kernel where `size` is in length units of the image according to im− >res[0]. This filter corresponds to grey-scale opening, the result is similar to the median but Luul is more efficient.

`im`	Pointer to the image.
`size`	Radius of the circular/sherical filter in pixel

void Ullu(image_cc *im, double size)

Combined Minimum-Maximum filter: upper-lower-lower-upper for grey scale images with a circular/spherical kernel where `size` is in length units of the image according to im− >res[0]. This filter corresponds to grey-scale closing, the result is similar to the median but Ullu is more efficient (The result of the Median filter is between Luul and Ullu).

`im`	Pointer to the image.
`size`	Radius of the circular/sherical filter in pixel

void pseudoMad(image_cc *orig, double size)
void pseudoMad(image_cc *orig, image_cc **pMed, double size)

pseudo Median absolute deviation The median absolute deviation is the median of the absolute differences towards the median. For reason of speed the median is replaced by the luul filter.

`orig`	pointer to original image
`pMed`	pointer to pointer to image containing the luul filtered image. If NULL, this will be created (and the pointer set to the new image).
`size`	radius of spherical window to use.

image_cc *MajorityFilter(image_cc *im, int nr, int wnd, double maj, double rel)
image_cc *MajorityFilter(image_cc *im, image_cc *roi, int nr, int wnd, double maj, double rel)

The current label is replaced by the most representative label among all neighbors in a cubic kernel, if (i) the most representative label exceeds a certain majority and (ii) the number exceeds that of the current label at the central voxel by a certain percentage

`im`	Pointer to the image.
`roi`	Pointer to the binary image that represents the region of interest (optional).
`nr`	Number of labels in the image
`wnd`	sidelength of cubic window: 2·wnd+1 pixel.
`maj`	majority [0-1]
`rel`	relative majority with respect to current label

return value:	Pointer to the filtered image

void FastMedian(image_cc *im, int size)

Median filter using a cubic window with fast updating of entries when the window is moved by one position. The running median method is adapted from Ashelly and distributed under the MIT License (MIT) (`https://gist.github.com/ashelly/5665911`)

`im`	Pointer to the image.
`size`	sidelength of cubic filter: 2·size+1 pixel.

void TotVarFilter(image_cc *im, double timestep, double lambda, int maxstep, int interval, int function)^{3D}

This filter minimizes the total variation while keeping a maximal fidelity to the original image (only 3D). The strength of fidelity is given by `lambda`. Method is according to Rudin, Fatemi, Osher (1992): Physica A,60,259-268.

`im`	Pointer to the image.
`timestep`	timestep for each filtering. If negative, will be handled dynamically. -0.1 normally works well.
`lambda`	fidelity parameter. 5 is a good start.
`maxstep`	maximal number of filtering steps.
`interval`	number of steps after which the image in memory will be updated.
`function`	kernel functions for filtering. 0: classical Total variation, 1: Entropy, 2: Enhanced entropy

void AnisoDiffFilter(image_cc *im, int maxstep, double threshold, double sigma)^{3D}

Anisotropic (or non-linear) diffusion filter according to Catté et. al (1992) SIAM Journal on Numerical Analysis, 29(1), 82-193.

`maxstep`	maximal number of filtering steps.
`threshold`	diffusion stop criterion - gradients larger than this are conserved
`sigma`	standard deviation of the Gaussian kernel for smoothing prior to edge evaluation

void Bin(image_c *im, int thresh)
void Bin(image_cc *im, int thresh, int lower, int upper)

Binarization of an image according to a threshold value. Grey values ≤ `thresh` are set to `lower` the others to `upper`. If no explicite values for `lower`and `upper` are given, this is equivalent to `lower`=0 and `upper`= maximum grey level. If the value of `upper` is larger than the maximum possible grey value, the original grey values above `thresh` are maintained.

`im`	Pointer to the image.
`thresh`	Threshold value
`lower`	Value written to pixel ≤ `thresh`

`upper`	Value written to pixel > `thresh`

.

void BinBand(image_cc *im, int lthresh, int uthresh)

Binarization of an image according to 2 threshold values. Values ≥ `lthresh` and ≤ `uthresh` are set to 0, the others to the maximum grey level.

`im`	Pointer to the image.
`lthresh`	Lower threshold value
`uthresh`	Upper threshold value

void BinBilevel(image_cc *im, int lower, int upper)

Segmentation of an image according to 2 thresholds which are regarded to be the limits of a fuzzy region within which the 'true' threshold is expected. The threshold is chosen locally according to the values of the neighboring pixel values. All pixel of a grey level ≤ `lower` are written to 0 as well as all pixel ≤ `upper` having at least one direct neighbor ≤ `lower`. The other pixel keep their original values. This rule is applied iteratively until no pixel has to be changed anymore.

`im`	Pointer to the image.
`lower`	Lower threshold.
`upper`	Upper threshold.

image_cc *DddBin(image_cc *im, int thresh)

Binarization of a ddd-image according to a single threshold values. The result is converted to a btd-image voxels are set to 0 for values ≤ `thresh` and t 1 else.

`im`	Pointer to the ddd-image.
`thresh`	threshold value
return value:	Pointer to the binary btd-image

image_cc *Sobel(image_cc *im)
image_cc *Sobel(image_cc *im, image_cc *maskim)

Sobel filter (first derivative of local grey levels). The image border is set to 0. If an additional binary image `maskim` of same dimensions is provided, the evaluation is only done for regions where `maskim` is non-zero.

`*im`	Pointer to the image.
return value:	Pointer to the filtered image

image_cc *Laplace(image_cc *im, int mode)
image_cc *Laplace(image_cc *im, int mode,image_cc *maskim)

Laplace filter (second derivative of local grey levels). The image border is set to 0. For `mode`=1 only positive Laplacians are stored, for `mode`=2 only negative and for `mode`=3 both. If an additional binary image `maskim` of same dimensions is provided, the evaluation is only done for regions where `maskim` is non-zero.

`*im`	Pointer to the image.

`mode`	indicates which sign is to be considered (bit1=positive, bit2=negative).
return value:	Pointer to the filtered image

3.6 Grey scale images (8 and 16 bit)^{8,16} (2D and 3D)^{2D,3D}

double *Histo(image_cc *im)^{8} ^{2D,3D}
double *Histo(image_cc *im, image_cc *mask)
Returns the histogram (pdf) of an image (16-bit images are converted to 8-bit prior to the evaluation). Optionally, a histogram is only calculated for locations where the mask image is white.

`*image`	Pointer to the grey image.
`*mask`	Pointer to the mask image
return value:	Pointer to the histogram (256 element array).

double *GreyCdf(image_cc *im)^{8} ^{2D,3D}
double *GreyCdf(image_cc *im, image_cc *mask)
Returns a 256 element array containing the cdf (cumulative histogram) of grey levels. Optionally, a cdf is only calculated for locations where the mask image is white.

`*im`	Pointer to the grey image
`*mask`	Pointer to the mask image
return value:	cdf array (256 elements)

void StretchHisto(image_cc *image, int Low, int High)
void StretchHisto(image_cc *image, double th)
void StretchHisto(image_cc *image)
Scales the histogram of a grey scale images (2D and 3D, 8-bit and 16-bit). Grey values between `Low` and `High` are mapped on a grey scale between 0 and the maximum value 2^8 or 2^{16} for 8-bit or 16-bit images respectively. Values lower than Low are set to 0, values higher than `High` are set to the maximum value. If the parameters `Low` and `High` are replaced by `th` the limits are set to lower and upper `th` percent of existing values. If no additional argument is given, the histogram ist rescaled according to the minimum and maximum value of the original image (corresponding to `th`=0). To match the grey values to a smooth histogram a suitable random value is added to rescaled values.

`*image`	Pointer to the image
`Low`	lower threshold
`High`	upper threshold

void HistoMatch(double *cdf, image_cc *image)^{8}
Transforms the histogram of an image according to a predefined cdf, typically optained from a source image using the function `GreyCdf()`.

`*cdf`	predefined cdf of grey levels
`*image`	Pointer to the 2D grey scale image
return value:	no return value

int UnimThresh(double *histo, int mode);

Threshold detection in a unimodal histogram typically obtained by `Histo` via triangulation according to Rosin(2001):Pattern Recognition,34,2083-2096. The threshold is located at the characteristic knee of the histogram, which has the largest perpendicular distance to an imaginary line connecting the mode with the brightest grey value (`mode`=1) or the darkest grey value (`mode`=0).

`*histo`	8bit grey value frequency distribution
`mode`	0=right tail, 1=left tail
return value:	threshold value

int KMeansThresh(double *histo, int nr);

Iterative Threshold Selection Method according to Ridler & Calvard (1978): IEEE Transactions on Systems, Man, and Cybernetics,8(8),630-632. Starting from an arbitrary set of thresholds, the arithmetic mean of adjacent class means is iteratively set as a new threshold until all thresholds converge to a stable values.

`*cdf`	8bit grey value frequency distribution
return value:	threshold value

int *ShapeThresh(double *cdf, int nr, double tau)
int *ShapeThresh(double *cdf, int nr, double tau, double perc)

Multilevel thresholding by local minima search according to Tsai (1995): Pattern Recognition ,16(6),653-666. Local maxima of histogram curvature are detected in addition, if there are less peaks than the specified number of classes.

`*cdf`	8bit cumulative grey value frequency distribution
`nr`	number of classes
`perc`	percentile that sets a fuzzy region around the optimal grey threshold
return value:	pointer to vector of nr-1 thresholds or 3(n-1) thresholds if `perc` is set

int *FuzzyCMeansThresh(double *cdf, int nr, double tau)
int *FuzzyCMeansThresh(double *cdf, int nr, double tau, double perc)

Multilevel thresholding by fuzzy c-means clustering according to Jawahar et al. (1997): Pattern Recognition ,30(10),1605-1613.

`*cdf`	8bit cumulative grey value frequency distribution
`nr`	number of classes
`tau`	fuzzyness index $[1, \infty)$. Method equals k-means clustering if $\tau = 1$.
`perc`	percentile that sets a fuzzy region around the optimal grey threshold
return value:	pointer to vector of nr-1 thresholds or 3(n-1) thresholds if `perc` is set

int *MaxVarThresh(double *cdf, int nr);
int *MaxVarThresh(double *cdf, int nr, double perc)

Multilevel thresholding by maximizing between-class variance (Otsu method) according to Liao et al. (2001):Journal of Information science and Engineering,17,713-727.

`*cdf`	8bit cumulative grey value frequency distribution
`nr`	number of classes
`perc`	percentile that sets a fuzzy region around the optimal grey threshold
return value:	pointer to vector of nr-1 thresholds or 3(n-1) thresholds if `perc` is set

int *MaxEntroThresh(double *cdf, int nr)
int *MaxEntroThresh(double *cdf, int nr, double perc)

Multilevel thresholding by maximizing the sum of histogram entropy of each class according to Kapur et al. (1985):Computer Vision, Graphics and Image Processing,29,273-285.

`*cdf`	8bit cumulative grey value frequency distribution
`nr`	number of classes
`perc`	percentile that sets a fuzzy region around the optimal grey threshold
return value:	pointer to vector of nr-1 thresholds or 3(n-1) thresholds if `perc` is set

int *MinErrThresh(double *cdf, int nr)
int *MinErrThresh(double *cdf, int nr, double perc)

Multilevel thresholding by minimizing the overlap error of fitted Gaussians according to Kittler & Illingworth (1986): Pattern Recognition ,19(1),41-47.

`*cdf`	8bit cumulative grey value frequency distribution
`nr`	number of classes
`perc`	percentile that sets a fuzzy region around the optimal grey threshold
return value:	pointer to vector of nr-1 thresholds or 3(n-1) thresholds if `perc` is set

int *GradMaskThresh(image_cc *image, double alpha)
int *GradMaskThresh(image_cc *image, double alpha, image_cc *mask)

Thresholding with gradient masks according to Schlüter et al. (2010): Computers & Geosciences,36,1246-51. Only locations along edges are taken into consideration for detection of an upper threshold. The lower threshold is calculated subsequently from simple histogram statistics.

`*image`	Pointer to the grey value image

`alpha`	defines the width of a fuzzy threshold range. Usually set between one and two.
`*mask`	Pointer to the mask image (optional)
return value:	pointer to vector of two thresholds (lower and upper threshold)

image_cc *WaterShed(image_cc *image, int conmode)[8] ^{2D,3D}

Calculates the watershed lines for a grey image (only 8-bit images). The different basins separated by the watershed lines are marked by different grey values (2D) or by a single grey value (3D) , the watershed line is 0. This may be applied to a distance map of a binary image to separate overlapping grains.

`*image`	Pointer to the grey image
`conmode`	Connectivity mode: either 4 or 8 (2D) and 6 or 26 (3D).
return value:	Pointer to the image containing the watershed and the basins

double *Acov(image_cc *im, double *corl, int lag, int mode)[8] ^{2D,3D}

Returns the autocovariance function of a 2D grey image

`*im`	Pointer to the image
`*corl`	address to write the correlation length as result
`lag`	maximum distance to evaluate (# pixels)
`mode`	indicates which directions are to be considered (bit1=x, bit2=y, bit3=z)
return value:	pointer to an array of dimension `lag` where the autocovariance function is stored

double *SemiVar(image_cc *im, int lag, int mode)[8] ^{2D,3D}

Returns the semi-variance function of a grey scale image.

`*im`	Pointer to the image
`*corl`	address to write the correlation length as result
`lag`	maximum distance to evaluate (# pixels)
`mode`	indicates which directions are to be considered (bit1=x, bit2=y, bit3=z)
return value:	pointer to an array of dimension `lag` where the semi-variance function is stored

long *GreyConFunc(image_cc *image, int mode)[8] ^{2D,3D}

Returns the connectivity function of a grey scale image. The image is binarized for all possible thresholds [0,255] and the corresponding Euler number is dermined which is returned as a vector of 255 elements. The Euler numbers are not normalized by the size of the image.

`*image`	Pointer to the grey image
`mode`	Connectivity mode for values < grey threshold: 4 or 8 (2D), 6 or 26 (3D)

return value:	array of 255 Euler numbers

image_cc *CircMask(image_cc *im, int xmid, int ymid, int rad)[8,16] ^{2D,3D}

Cuts out a circular (2D) or cylindrical (3D) image with center `xmid, ymid` and radius `rad`. All pixels outside the circle are set to the maximum grey level.

`*image`	Pointer to the grey image
`xmid, ymid`	center coordinates of the circle
`rad`	radius of the circle
return value:	Pointer to the resulting image

3.7 Binary images

image_cc *ErodeCirc(image_cc *image, double rad, int mode)
image_cc *ErodeDist(image_cc *image, double rad, int mode)

Performs an erosion or dilation of a binary image using a circular/spherical structuring element. The radius of this element is given in number of pixels in x-direction (this might be relevant for unisotropic pixel geometry). The function `ErodeDist` is based on the entire distance map of the image and is more efficient for large structuring elements, while `ErodeCirc` evaluates only individual structuring elements which is more efficient for small `rad`

`*image`	Pointer to the image.
`step`	radius of the structuring element **(in x-pixel)**
`mode`	0 = erosion, 1 = dilation of the black phase
return value:	Pointer to the resulting image

image_cc *OpenCirc(image_cc *im, double rad, int mode)
image_cc *OpenDist(image_cc *im, double rad, int mode)

Performs a morphological opening or closing of a binary image using a circular/spherical structuring element. The radius of this element is given in number of pixels in x-direction (this might be relevant for unisotropic pixel geometry). The function `OpenDist` is based on the entire distance map of the image and is more efficient for large structuring elements, while `OpenCirc` evaluates only individual structuring elements which is more efficient for small `rad`

`*image`	Pointer to the image.
`step`	radius of the structuring element
`mode`	0 = opening of the black phase, 1 = closing of the black phase
return value:	Pointer to the resulting image

image_cc *GetDistMap(image_cc *image)

Generates the distance map of a binary image, which is a grey scale image where the grey value of each pixel indicates the orthogonal distance to the black-white interface. The resulting image is in 16bit grey scale. The Euclidian distance is calculated in number of pixels in x-direction and is rounded up to the next integer x. Hence the distance map has values of 32768+/-x while the value 32768 representing the black-white interface does not exist. This function can also be used for anisotropic pixel geometry (different resolution in x,y,z). In this case the calculated distances are always in units of number of pixel in x-direction.

`*image`	Pointer to the binary image.
return value:	Pointer to the resulting image (16bit grey)

image_cc *GetOpenMap(image_cc *distmap, int smax)

Generates the 'opening map' for the black phase of a binary image. This is a grey scale image where the grey value of each pixel indicates the maximum size of a circle (2D) or a sphere (3D) which is a complete subset of the black phase. The size of the circle/sphere is in number of pixels in x-direction. As input the distance map is used which is generated by `GetDistMap()`. The generated opening map can be used to calculate the 'opening size distribution' using the function `MinkowskiOpenFunctions`

`*distmap`	Pointer to the distance map (16bit grey).
`smax`	number of opening steps
return value:	Pointer to the resulting opening map

void MinkowskiFunctions(image_cc *image, char *outfile)
Calculates all Minkowski-Functions, i.e. Minkowski functionals in dependency of the grey threshold. All possible thresholds are evaluated. This function is typically used for the evaluation of distance maps generated by `GetDistMap`. The results are written to the file `outfile`.

`*image`	Pointer to the input image.
`outfile`	file name where the results are written to

void MinkowskiOpenFunctions(image_cc *image, char *outfile)
void MinkowskiOpenFunctions(image_cc *image, image_cc *maskim, char *outfile)
Calculates all Minkowski-Functions for an opening map generated by `GetOpenMap`, i.e. Minkowski functionals in dependency of the opening size. If an additional binary image `maskim` of same dimensions is provided, the evaluation is only done for regions where `maskim` is non-zero. The results are written to the file `outfile`.

`*image`	Pointer to the input image.
`gstep`	scaling of the grey levels of the opening maps
`pixstep`	evaluated sizes in unit of pixel
`outfile`	file name where the results are written to

long int *m2Quant(image_cc *image)
long int *m2Quant(image_cc *image, image_cc *maskim)
Calculates the frequency distribution of 16 different pixel configurations in a 2x2 square. The returned pointer is input for the routines to calculate Minkowski functionals: volume density, surface density, length density, and Euler number. If another binary 'mask' is provided (`maskim`) having the same dimensions as `image`, the evaluation is done only for regions where `maskim` is not zero.

`*im`	Pointer to the binary 2D-image.
`*maskim`	Pointer to the image containing a mask.
return value:	Pointer to the 16-element array containing frequencies of pixel configurations

long int *m3Quant(image_cc *image)

long int *m3Quant(image_cc *image, image_cc *maskim)

Calculates the frequency distribution of 256 different voxel configurations in a 2x2x2 cube. The returned pointer is input for the routines to calculate Minkowski functionals: volume density, surface density, curvature density and Euler number. If another binary 'mask' is provided (`maskim`) having the same dimensions as `image`, the evaluation is done only for regions where `maskim` is not zero. NOTE THAT 3-dimenional Minkowski functionals treat the white phase (btd values = 1) as foreground which is in contrast to the 2-dimensional version (just to keep you flexible).

`*im`	Pointer to the binary 3D-image in btd-format.
return value:	Pointer to the 256-element array containing frequencies of voxel configurations

double m2areadens(image_cc *image, long int *h);

Returns the area density of a 2D binary structure (black phase) which corresponds to the volume density of a 3D structure as estimated from the 2-dimensional section.

`*h`	Pointer to the array of pixel configurations obtained by m2Quant()
return value:	Volume density $[L^2/L^2]$

double m2lengthdens(image_cc *image, long int *h);

Returns the length density B_A $[L/L^2]$ of the boundary per area of a 2D binary structure. The units of L are given in the units of image resolution (image− >res[]).

`*h`	Pointer to the array of voxel configurations obtained by m2Quant()
`*image`	Pointer to a 2D binary image (0=black, 255=white)
return value:	Length density $[L/L^2]$

double m2surfdens(image_cc *image, long int *h);

Returns the surface density S_V $[L^2/L^3]$ of the black-white interface of a 3D binary structure. The evaluated image is considered to be a 2-dimensional section through that 3D structure. The units of L are given in the units of image resolution (image− >res[]).

`*h`	Pointer to the array of pixel configurations obtained by m2Quant()
`*image`	Pointer to a 2D binary image (0=black, 255=white)
return value:	Surface density $[L^2/L^3]$

double m2euler4(image_cc *image, long int *h);

Returns the Euler number χ_A $[1/L^2]$ of a 2D binary structure considering 4-connectivity of the black phase. The units of L are given in the units of image resolution (image− >res[]).

`*h`	Pointer to the array of pixel configurations obtained by m2Quant()
`*image`	Pointer to a 2D binary image (0=black, 255=white)

return value:	Euler number [$1/L^2$]

double m2euler8(image_cc *image, long int *h);

Returns the Euler number χ_A [$1/L^2$] of a 2D binary structure considering 8-connectivity of the black phase. The units of L are given in the units of image resolution (image− >res[]).

`*h`	Pointer to the array of pixel configurations obtained by m2Quant()
`*image`	Pointer to a 2D binary image (0=black, 255=white)
return value:	Euler number [$1/L^2$]

double m3voldens(long int *h);

Returns the volume density of a 3D binary structure (white phase).

`*h`	Pointer to the array of voxel configurations obtained by m3Quant()
`*image`	Pointer to a 3D binary image in btd-format (0=black, 1=white)
return value:	Volume density [L^3/L^3]

double m3surfdens(image_cc *image, long int *h);

Returns the surface density S_V [L^2/L^3] of the black-white interface of a 3D binary structure. The units of L are given in the units of image resolution (image− >res[i]).

`*h`	Pointer to the array of pixel configurations obtained by m3Quant()
`*image`	Pointer to a 3D binary image in btd-format (0=black, 1=white)
return value:	Surface density [L^2/L^3]

double m3meancurv(image_cc *image, long int *h);

Returns the mean curvature density C_V [L/L^3] of the black-white interface of a 3D binary structure. The units of L are given in the units of image resolution (image− >res[]).

`*h`	Pointer to the array of pixel configurations obtained by m3Quant()
`*image`	Pointer to a 3D binary image in btd-format (0=black, 1=white)
return value:	Curvature density [L/L^3]

double m3euler6(image_cc *image, long int *h);

Returns the Euler number χ_V [$1/L^3$] of a 3D binary structure considering 6-connectivity of the white phase. The units of L are given in the units of image resolution (image− >res[]).

`*h`	Pointer to the array of voxel configurations obtained by m3Quant()
`*image`	Pointer to a 3D binary image in btd-format (0=black, 1=white)
return value:	Euler number [$1/L^3$]

double m3euler26(image_cc *image, long int *h);

Returns the Euler number χ_V [$1/L^3$] of a 3D binary structure considering 26-connectivity of the white phase. The units of L are given in the units of image resolution (image− >res[]).

`*h`	Pointer to the array of voxel configurations obtained by m3Quant()
`*image`	Pointer to a 3D binary image in btd-format (0=black, 1=white)
return value:	Euler number [$1/L^3$]

image_cc *LogAnd(image_cc *wnd1, image_cc *wnd2)
Logical AND relation (intersection) of two binary images.

`wnd1`	Pointer to the first image description structure.
`wnd2`	Pointer to the second image description structure.
return value:	Pointer to the resulting image

image_cc *LogOr(image_cc *wnd1, image_cc *wnd2)
Logical OR relation (unification) of two binary images.

`wnd1`	Pointer to the first image description structure.
`wnd2`	Pointer to the second image description structure.
return value:	Pointer to the resulting image

image_cc *Intersection(image_cc *wnd1, image_cc *wnd2, int gv1, int gv2)
Intersection of two binary images. The black phase of the two images is set to `gv1` and `gv2` respectively, their intersection is set to black.

`wnd1`	Pointer to the first image description structure.
`wnd2`	Pointer to the second image description structure.
`gv1`	grey level to be set for phase in wnd1.
`gv2`	grey level to be set for phase in wnd2.
return value:	Pointer to the resulting image

void bThinning(image_cc *im);
Thinning of 0-phase of a binary image, considering 4-connectivity.

`*im`	Pointer to the image.

void bThinning8(image_cc *im);
Thinning of 0-phase of a binary image, considering 8-connectivity .

`*im`	Pointer to the image.

void bConCom(image_cc *im,int xv,int yv,int val)
Marking of th connected component which includes the seed point `xv yv` in a binary image.

`*im`	Pointer to the image.
`xv yv`	Seed point of the component.
`val`	Grey value to mark the conected component.

int bConCom2(image_cc *image,int xv,int yv);

Determines the size of the connected object identified by greylevel 0 around point xv/yv considering 8-connectivity. Return value is the number of pixels attributed to the object. The original image is not changed.

`*image`	Pointer to the binary image
`xv, yv`	coordinates for seed point of the connected component
return value:	number of pixels attributed to the object considering 8-connectivity

int isPercol(image_cc *image, int backbone, int mode);

returns 1 if the 0-values of a binary structure percolate in the directions indicated by `mode` (bit1=x, bit2=y, bit3=z). if `backbone` is not 0 the backbone (or in case of no percolation the continuous part with respect to plane x/y/z=0 indicated by `mode`) is stored as additional image named 'backbone' if `backbone` is 0 the calculation is interupted as soon as percolation was detected (which is faster).

`*image`	Pointer to the binary image
`backbone`	if not 0 an image of the continuous part is stored (file name 'backbone')
`mode`	direction in which percolation is tested
return value:	1 if the structure percolates, 0 else

image_cc *Cluster(image_cc *image, int *nr)

Fast object detection with the Hoshen-Kopelman algorithm, as implemented by Tobin Fricke and distributed under GNU public license (`http://www.ocf.berkeley.edu/~fricke/projects/hoshenkopelman/hoshenkopelman.html`). Each isolated white objects gets a different grey value label and the label image is returned.

`*image`	Pointer to the binary image
`*nr`	Pointer to integer where number of objects is written
return value:	image with object labels

void RemoveObjects(image_cc *image, int size, int mode)
void RemoveObjects(image_cc *image, image_cc *mask, int size, int mode)

Removes objects or fills holes smaller than a certain size limit. Objects are internally labeled with the Hoshen-Kopelman algorithm.

`*image`	Pointer to the binary image
`*mask`	Pointer to the binary mask for region of interest (optional)
`size`	size limit in number of voxels
`mode`	fill holes [0] or remove objects[1]

double ConLength(image_cc *image, int phase, int every, char *outfile);

Calculates the pair connectivity as a function of distance considering 8-connectivity. Return value is the mean connectivity length. The step size has be higher than one, especially in 3D, because the evaluation of voxel pairs is rather slow.

`*image`	Pointer to the binary image
`phase`	black [0] or white[1]
`every`	step size
`*outfile`	file name where the pair connectivity data is written to
return value:	mean connectivity length

3.8 RGB color images

unsigned char Red(image_cc *image, int x, int y);
Reads the red value at given coordinates.

`*image`	Pointer to the image
`x, y`	coordinates
return value:	red value

unsigned char Green(image_cc *image, int x, int y);
Reads the green value at given coordinates.

`*image`	Pointer to the image
`x, y`	coordinates
return value:	green value

unsigned char Blue(image_cc *image, int x, int y);
Reads the blue value at given coordinates.

`*image`	Pointer to the image
`x, y`	coordinates
return value:	blue value

void WRed(image_cc *in, int x, int y, unsigned char val);
Writes the red value `val` at given coordinates.

`*image`	Pointer to the image
`x, y`	coordinates
`val`	written red value
return value:	no return

void WGreen(image_cc *in, int x, int y, unsigned char val);
Writes the green value `val` at given coordinates.

`*image`	Pointer to the image
`x, y`	coordinates
`val`	written green value
return value:	no return

void WBlue(image_cc *in, int x, int y, unsigned char val);
Writes the blue value `val` at given coordinates.

`*image`	Pointer to the image
`x, y`	coordinates
`val`	written blue value
return value:	no return

void StretchRGBHisto(image_cc *in, int Rlow, int Rhigh, int Glow, int Ghigh, int Blow, int Bhigh)

Stretches the color histogram according the given limits. The source-image is overwritten.

`in`	Pointer to the RGB-image.
`Rlow, Rhigh`	Lower and upper limit for the red channel.
`Glow, Ghigh`	Lower and upper limit for the green channel.
`Blow, Bhigh`	Lower and upper limit for the blue channel.

void StretchRGBBright(image_cc *in, int Low, int High)

Stretches the brightness of a RGB-color image between `Low and High` $\in [0, 255]$. The source-image is overwritten.

`Low,High`	Lower und upper limit of brightness

3.9 Graphics

The following routines use the library `PS_graf` to generate nice postscript graphics.

void psPlot(char *fname, int n, double *xdat, double *ydat, int mode)
Draws a x-y graphic and stores it to `fname`.eps.

`*fname`	Name of output file.
`n`	Number of data points.
`*xdat, *ydat`	Arrays of x- and y-data.
`mode`	Draws symbols for `mode`=0 and lines else.

void psPlotTit(char *fname,char *xtitle,char *ytitle, int n, double *xdat, double *ydat, int mode)
Draws a x-y graphic and stores it to `fname`.eps.

`*fname`	Name of output file.
`*xtitle`	title for x axis.
`*ytitle`	title for y-axis.
`n`	Number of data points.
`*xdat, *ydat`	Arrays of x- and y-data.
`mode`	Draws symbols for `mode`=0 and lines else.

void psMultiPlot(char *fname, int *ndat, int nplot, double **xdat, double **ydat, int mode)
Draws a x-y graphic of multiple data sets and stores it to `fname`.eps.

`*fname`	Name of output file.
`*ndat`	Array of size `nplot` containing the number of data points of each data set.
`nplot`	Number of data sets.
`**xdat, **ydat`	Arrays of size `nplot` of pointers to the x- and y-data sets.
`mode`	Draws symbols for `mode`=0 and lines else.

void psMultiPlotTit(char *fname,char *xtitle,char *ytitle, int *ndat, int nplot, double **xdat, double **ydat, int mode)
Draws a x-y graphic of multiple data sets and stores it to `fname`.eps.

`*fname`	Name of output file.
`*xtitle`	title for x axis.
`*ytitle`	title for y-axis.
`*ndat`	Array of size `nplot` containing the number of data points of each data set.
`nplot`	Number of data sets.
`**xdat, **ydat`	Arrays of size `nplot` of pointers to the x- and y-data sets.
`mode`	Draws symbols for `mode`=0 and lines else.

void psDddCircHisto(image_cc *im, char *buf, int prec);
Draws the histogram of a 3D greylevel image considering only the central cylinder of the image.

`*im`	Pointer to the image.
`*buf`	Name of output file.
`*prec`	Only a fraction (1/`prec`) of the total number of voxels are considered.

void psHisto(image_cc *im, char *buf, int prec);
Draws the histogram of a 2D greylevel image.

`*im`	Pointer to the image.
`*buf`	Name of output file (eps format).
`*prec`	Only a fraction (1/`prec`) of the total number of voxels are considered.

void ps3Dview(char *fname, image_cc *image, double min, double max);
Draws a 3D colored view of a ddd-image and write an eps-file.

`*fname`	name of the resulting eps-file
`*image`	Pointer to the ddd-image.
`min, max`	miniumum and maximum greylevel inbetween which the color is scaled.

3.10 Special routines

void DLine(int x1,int y1,int x2,int y2,image_cc *image, int val)
Draws a line from x1/y1 to x2/y2.

`x1,y1`	Coordinates of one end of the line ...
`x2,y2`	... and the other.
`image`	Pointer to the image.
`val`	Pixel value of the line $\in [0, 255]$.
return value:	p

ointer to the new image.

image_cc *GetVoronoiTes(int Xdim, int Ydim , int Nump)[8]
Generates a Voronoi tesselation based on `NumP` random seed point. In the resulting image the Voronoi-cells are marked by different grey levels [1-254], the edges between the cells are white [255].

`Xdim`	x-size of resulting image
`Ydim`	y-size of resulting image
`Nump`	number of seed points
return value:	pointer to the resulting image

image_cc *GetPercolClus(int width, int height, double lamx, double lamy, int mode)[8]
Returnes a percolation cluster based on the excursion set of a random greyscale image

`width, height`	size of image
`lamx, lamy`	Correlation length in x and y
`mode`	0: random (correlation lengths have no meaning)
	1: gaussian covariance with correlation lengths
return value:	pointer to the binary image of resulting percolation cluster

void bObjects(image_cc *image,unsigned long *o,unsigned long *l)
Counts the number of objects (disconnected parts) and the number of loops (holes within the objects for the dark phase (0) of a binary image.

`image`	Pointer to the image description structure.
`o`	Pointer to the number of objects.
`l`	Pointer to the number of loops.

The edges of the image should be set to non phase (255)

int bContour(int x, int y, unsigned char mark, image_cc *image)
Marks the edge of an object of the dark phase (0) of a binary image and determines if it is the outer edge of an object or the edge of a hole within the object.

`x`	x coordinate at the edge of an object.
`y`	y coordinate at the edge of an object.

`mark`	value to mark the edge 0 <mark< 255.
`image`	Pointer to the image description structure.
return value:	n

egativ if the marked edge is a hole, positiv else.

int bContourCent(int x, int y, int mark, image_cc *image, int *xx, int *yy)

Determines the geometrical center of an object or a hole within an object, where an object is defined by a values $<$ `mark`. The value of the center is set to `mark+1`, the pixels at the borders to `mark`.

`x,y`	Coordinates at the border of an object.
`mark`	Threshold defining the objects.
`xx, yy`	Adresses to which the coordinates of the object are written.
return value:	>0 for real objects, $<=0$ if the contoured border line describes a hole in an object.

double pDisector(image_cc *wnd1, image_cc *wnd2)

Calculates an unbiased estimate for the volumetric 3D Euler number [L^{-3}] of the dark phase (0) from a pair of parallel binary images (a disector). Note that the resolution of the images must be set. The separation of the parallel images should be smaller than the objects considered.

`wnd1`	Pointer to the first image description structure.
`wnd2`	Pointer to the image description structure of a parallel image.
return value:	v

olumetric Euler number.

double BtdDiffusionZ(image_cc *in, FILE *dif, unsigned long max, unsigned long min, int fluxstep, double sens);

Calculates diffusion through phase [1] in z-direction. The concentration at one side is kept fixed, $C(z=0)=$const, while the opposite side is fixed at zero $C(z=z_{max})=0$. Diffusive flow across the plane z_{max} is calculated iteratively using explicite finite differences.

`*in`	Pointer to the 3D binary image
`*dif`	Pointer to the file where the diffusive flow is stored in intervals indicated by `fluxstep`
`max`	Maximum number of iterations (depends on sample size, may be 50.000 to 1.000.000)
`min`	Minimum number of iterations
`fluxstep`	Interval of iterations to store the diffusive flux at z=zmax (e.g. 100)
`sens`	Interupt criteria: stop iterations if flux[j]-flux[j-1] $<$ flux[j]*sens
return value:	Relative apparent diffusion coefficient D_s/D_0

void BtdSkelet(image_cc *im, int mode, int deadends);
Transform the image to its 3D skeleton.

`*im`	Pointer to the 3D binary image.
`mode`	0 = 6 neighbors are considered to be connected, 26 else
`deadends`	0 = the minimum skeleton without any dead ends is calculated, else, dead ends are preserved

void BtdContinuity(image_cc *im,int mode)
Filter for the continuous part of the phase coded by [1] within a binary 3D-image. The bit-sequenze of `mode` determins the faces of the 3D-image to which the phase must be connected. (mode=1: face at x=0; mode=63: any face).

`im`	Pointer to the 3D image
`mode`	bitposition:face 1:x=0, 2:x=xmax, 3:y=0, 4:y=ymax, 5:z=0, 6:z=zmax

void BtdDrawSphere(image_cc *image, int xmid, int ymid, int zmid, int rad, int val);
Draws a sphere at center `xmid/xmid/zmid` with radius `rad` and value 0 if `val`= 0 and 1 else.

`*image`	Pointer to the 3d binary image
`xmid, xmid, zmid`	coordinates of sphere center
`rad`	radius of sphere
`val`	value to be written for the sphere

3.11 Useful stuff

... typically for internal use

image_cc *LoadRaw(char *buf, int cols, int rows, int layers, int offset, int nbyte)

Loads grey images of 8-bit or unsigned 16bit. Byte order has to be little Endian and data index is assumed to be x-fastest.

`buf`	Name of the image to be loaded (with extension).
`cols`	Number of voxels in x-direction.
`rows`	Number of voxels in y-direction.
`layers`	Number of voxels in z-direction.
`offset`	Header size in bytes.
`nbyte`	Number of bytes per voxel - 1 for `char`, 2 for `unsigned int`.
return value:	pointer to the loaded image.

void StoreRaw(image_cc *im)

Saves a 3D grey image to a `raw` file and writes file information to a `txt` file.

`im`	Pointer to the image.

char *GetCircElement(int rad, double rx, double ry)

Generates a 2D circular structuring element with radius `rad` [pixel]. Returns a pointer to an array, a (char), containing the structuring element: $a[0] =$ radius in x-direction [pixel], $a[1] =$ radius in y-direction. $a[2...]$ contains all pixel of the smallest square containing the structuring element with dimension $(2a[0] + 1) \times (2a[1] + 1)$. Each pixel is coded by 1 bit which has the value 1 if it is part of the stucturing element and 0 else. The position of a pixel (x,y) is identified by $pos = y * (2a[0] + 1) + x$. For a given pos you find its value by $*(a + 2 + pos/8)$ & $1 << pos\%8$.

`rad`	radius of the structuring element [voxels]
`rx, ry`	Resolution (pixel size) in different dimensions [L]
return value:	pointer to the structuring element

char *GetSphereElement(int rad, double rx, double ry, double rz)

Generates a spherical structuring element with radius `rad` [voxels].

`rad`	radius of the structuring element [voxels]
`rx, ry, rz`	Resolution in different dimensions [L]
return value:	pointer to an array, a (char), containing the structuring element: $a[0] =$ radius in x-direction [voxel], $a[1] =$ radius in y-direction, $a[2] =$ radius in z-direction. $a[3...]$ contains all voxels of the smallest cube containing the structuring element with dimension $(2a[0] + 1) \times (2a[1] + 1) \times (2a[2] + 1)$. Each voxel is coded by 1 bit which has the value 1 if it is part of the stucturing element and 0 else. The position of a voxel (x,y,z) is identified by

$pos = z * (2a[0]+1) * (2a[1]+1) + y * (2a[0]+1) + x$. For a given pos you find its value by $*(a+3+pos/8)$ & $1 << pos\%8$.

char *GetSphereElementDouble(double rad, double rx, double ry, double rz, int *vol)

Same as GetSphereElement but with non-integer radius `rad` [voxels].

`rad`	radius of the structuring element [voxels]
`rx, ry, rz`	Resolution in different dimensions [L]
`vol`	the size of the structuring element [number of voxels] is written to this address
return value:	same as GetSphereElement.

long int *bQuantMask(image_cc *image, int xm, int ym, char *mask)

Calculates the frequency distribution of 2x2 pixel configurations for a region identified by `mask`. The format of `mask` corresponds to that returned by GetCircElement. The returned pointer is input for the routines to calculate the Minkowski functionals.

`*im`	Pointer to the binary 2D-image.
`int xm, ym`	Coordinates of the center of the region described by `mask`.
`*mask`	Mask for the region to be analyzed.
return value:	Pointer to the 16-element array containing the frequencies of pixel configurations

float ran3(long *idum);

Genetrates a random number ∈[0,1].

`*idum`	Pointer to the initialization value.
return value:	Random number.

void GetMinMax(image_cc *image, int *max, int *min);

Determines the maximum and the minimum value of an image.

`*image`	Pointer to the image.
`*min, *max`	The resulting minimum and maximum values.

int GetMaxGrey(image_cc *im)

Determines the maximum grey value of an image not considering the absolut possible maximum of the image type.

`*image`	Pointer to the image.
return value:	maximum grey value.

int GetMinGrey(image_cc *im)

Determines the minimum grey value of an image not considering the value zero.

`*image`	Pointer to the image.

return value:	minimum grey value.

void MinMaxf(double *ar, int n, double *min, double *max);
calculates the minimum and maximum value of a double array

`*ar`	Pointer to the array
`n`	dimension of the array
`*min`	addresse to store minimum
`*max`	addresse to store maximum
return value:	Euler number [1/L^3] in units of image-¿rcol

double **dmatrix(int n, int m);
allocates a 2D double array with dimensions n x m

`n, m`	dimensions of the array
return value:	pointer to the allocated memory

int **imatrix(int n, int m);
allocates a 2D int array with dimensions n x m

`n, m`	dimensions of the array
return value:	pointer to the allocated memory

3.12 Functions of previous versions (still active)

void gBin(int LOW, int value1,int value2,image_cc *image)
Segmentation of a 8-bit grey scale images according to a threshold. The grey levels smaller or equal to `LOW` are written to `value1` the others are written to `value2`. If `value2` is larger than 255 the original values are maintained for grey levels > `LOW`.

`LOW`	Threshold
`value1`	grey level [0-255]
`value2`	grey level [0-255]
`image`	Pointer to the image

void gBibin(int LOW,int HIGH,image_cc *image)
Segmentation of a 8-bit grey scale image according to 2 thresholds which are regarded to be the limits of a fuzzy region of the grey scale histogram where the true threshold is expected. The threshold is chosen locally according to the values of the neighboring pixel values (Conditional Dilation).

`LOW`	lower threshold.
`HIGH`	upper threshold.
`image`	Pointer to the image.

All pixel of a grey level smaller or equal than `LOW` are written to 0 as well as all pixel having values smaller than `HIGH` and at least one direct neighbour smaller than `LOW`. This algorithm is repeated iteratively until no pixel has to be changed anymore. The other pixel keep their original value.

int gBilevel(int *th_high,int *th_low,image_cc *image)
Calculates the limits of a fuzzy region on the grey scale of an 8-bit grey level image as required by gBibin for thresholding. The image should have a bimodal grey level histogram where the different modes are not clearly separated (a typical case).

`th_high`	pointer to the upper limit.
`th_low`	pointer to lower limit.
`image`	Pointer to the image.
return value:	-

1 if the histogram is not bimodal, 1 if sucessful

The bimodal grey level histogram $h(x)$ is analysed to get the lower and upper maxima $max1$ and $max2$ as well as the minimum min in between. Then a Gaussian distribution $\hat{h}(x)$ is fitted to the upper mode. The lower limit is calculated as $(max1 + min)/2$, the upper limit as $(p + min)/2$ where p is the location on the grey scale where $\hat{h}(p) = h(min)$. The resulting values `th_high` and `th_low` may be used in the function `gBibin` for conditional dilation.

image_cc *GetDddSegment(image_cc *im, int ulx, int uly, int ulz, int nx, int ny, int nz)
Cuts out a segment of a 3D grey level image.

`*im`	pointer to the original image.
`ulx, uly, ulz`	coordinates of the upper left corner of the Segment.

`nx, ny, nz`	number of voxels of the segment in different directions.
return value:	pointer to the segment.

double *GetGreyCdf(image_cc *im)[8]

Returnes a 255 element array containing the cdf of grey levels

`*im`	Pointer to the 2D grey image
return value:	cdf array (255 elements)

image_cc *gWaterShed(image_cc *image, int conmode)[8];

Calculates the watershed lines for a grey image (only 8-bit images). The different basins separated by the watershed lines are marked by different grey values, the watershed line is 0. This may be applied to a distance map of a binary image to separate overlapping grains.

`*image`	Pointer to the grey image
`conmode`	Connectivity mode: either 4 or 8.
return value:	Pointer to the image containing the watershed and the basins

double *GetAcov(image_cc *im, double *corl, int lag, int mode)[8]

Returnes the autocovariance function of a 2D grey image

`*im`	Pointer to the image
`*corl`	address to write the correlation length as result
`lag`	maximum distance to evaluate (# pixels)
`mode`	indicates which directions are to be considered (bit1=x, bit2=y)
return value:	pointer to an array of dimension `lag` where the autocovariance function is stored

double *GetSemiVar(image_cc *im, int lag, int mode)[8]

Returnes the semivariogram of a 2D grey image

`*im`	Pointer to the image
`lag`	maximum distance to evaluate (# pixels)
`mode`	indicates which directions are to be considered (bit1=x, bit2=y)
return value:	pointer to an array of dimension `lag` where the semivariogram is stored

void gErode(image_cc *image,int step, int mode)

Minimum/maximum filter for 8-bit grey scale images.

`image`	Pointer to the image.
`step`	radius of the considered environment
`mode`	0 = minimum, 1 = maximum filter

Each pixel is set to the minimum/maximum value of its direct neighbours. This is iteratively done `step` times considering the 4 and 8 nearest neighbours alternant starting with 4. The edges of the image are written to 255.

image_cc *gMean2(image_cc *image, int mode)

As Mean but returns an image of the variance at each pixel within the defined window of size `mode` × `mode`

`image`	Pointer to the image.
`mode`	side length of the squared operating window (number of pixel)
return value:	Pointer to the image of variances

image_cc *gSobel(image_cc *im, int mode)
Sobel filter (first derivative of local grey levels). The image border is set to 0. For `mode`=1 the histogram of the resulting image is rescaled to the entire grey scale [0 - max-grey]. (only for 2D images).

`*im`	Pointer to the image.
return value:	Pointer to the filtered image

image_cc *gLaplace(image_cc *im, int mode)
Laplace filter (second derivative of local grey levels). The image border is set to 0. For `mode`=1 the histogram of the resulting image is rescaled to the entire grey scale [0 - max-grey]. (only for 2D images).

`*im`	Pointer to the image.
`mode`	Rescaling of the result if non-zero.
return value:	Pointer to the filtered image

double *gHisto(image_cc *im, int precision);
Calculates the grey level histogram (pdf).

`*im`	Pointer to the image.
`precision`	only a fraction (1/precision) of pixels is considered
return value:	Pointer to a 255-element array containing the histogram

image_cc *GetRandImage(int col, int row, double rx, double ry, double cx, double cy, int mode)[8]
Returnes a random 2D image with defined correlation lengths in x and y direction

`*cdf`	cdf of grey levels
`col, row`	dimensions of resulting image
`rx, ry`	size of pixel in x and y
`cx, cy`	correlation lengths (# pixel) in x ynd y
`mode`	0: Gaussian correlation 1: Lorentz-Correlation model 2: Exponential-Correlation model 3: von Karman-Correlation model
return value:	pointer to the created image

image_cc *GetCorImage(double *cdf, int col, int row, double rx, double ry, double cx, double cy, int mode)[8]
Returnes a random 2D image optionally with defined grey distribution function (equal, Gauss or predefined `cdf`), defined correlation length and defined correlation model (Gauss, Lorentz, Exponentila, von Karman).

`*cdf`	cdf of grey levels

`col, row`	dimensions of reulting image
`rx, ry`	size of pixel in x and y
`cx, cy`	correlation lengths (# pixel) in x ynd y
`mode`	0: equal distribution without any correlation (cx,cy,cz and cdf have no meaning here) 1: equal grey distribution with gaussian correlation 2: predefined grey distribution (cdf) distribution and gaussian correlation 3: Gaussian correlation (cdf have no meaning here) 4: Lorentz-Correlation model (cdf have no meaning here) 5: Exponential-Correlation model (cdf have no meaning here) 6: von Karman-Correlation model (cdf have no meaning here) `**acov` covariance in x and y direction (acov[2][lag])
return value:	pointer to the created image

long *gConfunc(image_cc *image)[8]

Returns the connectivity function of a 2D grey image. The image is binarized for all possible thresholds [0,255] and the corresponding Euler number is dermined which is returned as a vector of 255 elements. The Euler numbers are not normalized by the size of the image.

`*image`	Pointer to the grey image
return value:	array of 255 Euler numbers

image_cc *gCircMask(image_cc *im, int xmid, int ymid, int rad, int val)[8,16]

Cuts out a circular image with center `xmid, ymid` and radius `rad`. All pixels outside the circle are set to grey level `val`. The size of the returned image is reduced to 2*`rad`+1.

`*image`	Pointer to the grey image
`xmid, ymid`	center coordinates of the circle
`rad`	radius of the circle
`val`	grey level written to the background
return value:	Pointer to the resulting image

void gHistoMatch(double *cdf, image_cc *image)[8]

Transforms the histogram of an image according to a predefined cdf, typically optained from a source image using the function `GetGreyCdf()`.

`*cdf`	predefined cdf of grey levels
`*image`	Pointer to the 2D grey scale image
return value:	no return value

void bRemObjects(image_cc *image, int size);

Removes all objects (grey level=0) smaller than 'size' (number of pixels)

`*image`	Pointer to the binary image
`size`	maximum size of objects (number of pixels) to be removed
return value:	no return value, the original image is changed

long int *bQuant(image_cc *image);
Calculates the frequency distribution of 16 different pixel configurations in a 2x2 square. The returned pointer is input for the routines to calculate volume density, surface density, length density, curvature and Euler number.

`*im`	Pointer to the binary 2D-image.
return value:	Pointer to the 16-element array containing frequencies of pixel configurations

double bVoldens(image_cc *image, long int *h);
Returnes the Area density of a 2D binary structure (0 phase) which corresponds to the Volume density of a 3D structure as estimated from the 2-dimensional section

`*h`	Pointer to the array of pixel configurations obtained by bQuant()
return value:	Volume density [-]

double bSurfdens(image_cc *image, long int *h);
Returnes the Surface density S_V [cm^2/cm^3] of the boundary of a 3D binary structure, as estimated from a 2-dimensional binary section.

`*h`	Pointer to the array of pixel configurations obtained by bQuant()
return value:	Surface density [L^2/L^3] in units of image-¿rcol

double bEuler4(image_cc *image, long int *h);
Returnes the Euler number of a 2D binary structure considering 4-connectivity of phase [1]

`*h`	Pointer to the array of voxel configurations obtained by bQuant()
return value:	Euler number [$1/L^2$] in units of image-¿rcol

double bEuler8(image_cc *image, long int *h);
Returnes the Euler number of a 2D binary structure considering 8-connectivity of phase [1]

`*h`	Pointer to the array of voxel configurations obtained by bQuant()
return value:	Euler number [$1/L^2$] in units of image-¿rcol

double bLengthdens(image_cc *image, long int *h);
Returnes the Lengthdensity B_A [cm/cm^2] of the boundary of a 2D binary structure

`*h`	Pointer to the array of voxel configurations obtained by bQuant()
return value:	Lengthdensity [L/L^2] in units of image-¿rcol

double b2DEuler(image_cc *image)
Calculates the 2D Euler number [-] of the dark phase (0) of a binary images.

`image`	Pointer to the first image description structure.
return value:	d

imensionless Euler number.

void bErode(image_cc *image,int step, int mode)
Performs an erosion or dilation of a binary image (0/255).

`image`	Pointer to the image description structure.
`step`	radius of the hexagonal structuring element
`mode`	0 = erosion of the dark phase (0), 1 = dilation

Erosion or dilation is iteratively performed `step` times considering the 4 and 8 nearest neighbours alternant starting with 4. Note that the original image is replaced and the outer shell (1 pixel) is written to 255.

void bErodeMir(image_cc *image,int step, int mode)
as bErode but the outer shell of width `step` is mirrored so that the complete image is treated.

`image`	Pointer to the image description structure.
`step`	radius of the hexagonal structuring element
`mode`	0 = erosion of the dark phase (0), 1 = dilation

image_cc *bErodeMirCirc(image_cc *image,int step, int mode)
as bErodeMir but optimal circles are used as structuring elements for erosion. The original image is not replaced, the pointer to the resulting image is returned.

`image`	Pointer to the image description structure.
`step`	radius of the hexagonal structuring element
`mode`	0 = erosion of the dark phase (0), 1 = dilation
return value:	Pointer to the eroded image

image_cc *bGetDistMap(image_cc *image, int *n, int gval, int gstep);
Converts a binary image to its distance map: Each pixel in the white phase (255) is written to a grey value which corresponds to the distance of that pixel to phase 0. The closest distance gets grey value
tt gval which increases ba steps `gstep` with distance. The phase 0 is not changed. The total number of distance classes is written to `n`. Note that it is a good idea to chose the parameters such that n
$cdotgstep > 255 - gval$. The resulting distance map can be used as input to a watershed segmentation e.g. to separate sintered grains or to calculate Minkowski functions.

`*image`	Pointer to the binary image
`*n`	Number of detected distance classes
`gval`	Grey level for the first distance class
`gstep`	Grey level step between adjacent distance classes
return value:	Pointer to the image containing the distance map

image_cc *bGetFullDistMap(image_cc *image, int *nblack, int *nwhite, int gstruc, int gval, int gstep);
Converts a binary image to its full distance map: Each pixel in the white phase (255) is written to a grey value which corresponds to the distance of that pixel to phase 0 and each pixel in the black phase (0) is written to a grey value which corresponds to the distance of that pixel to the white phase (255). The boundary of the black phase is set to `gstruc`, the closest distance gets grey value `gstruc+/-gval` where the sign depends on whether the distance is in the white or in the black phase. It

is increased by steps `gstep` with distance. The total number of distance classes is written to `nblack` and `nwhite` respectively. Note that it is a good idea to chose the parameters such that nwhite·gstep < 255 -(gstruc+gval) and nblack·gstep<gstruc-gval. The maximum number of classes is limited by MAXDILAT=100 (parameter can be changed in `quantim4.h`). The resulting distance map can be used to calculate Minkowski functions.

`*image`	Pointer to the binary image
`*nblack`	Number of detected distance classes in the black phase
`*nwhite`	Number of detected distance classes in the white phase
`gstruc`	Grey level written to the boundary of the black phase
`gval`	Grey level for the first distance class
`gstep`	Grey level step between adjacent distance classes
return value:	Pointer to the image containing the distance map

int bGetDistOpenMap(image_cc *image, image_cc *distance, image_cc *opened, int gval, int gstep)

As bGetDistMap(), this function converts a binaryimage to its distance map. Additionally a 'granulometry map' is calculated where each pixel in phase 1 (255) is written to a grey value which corresponds to the diameter of the maximum circle that can be placed inside phase 1 at that location (corresponding to the 'opening size'). The smallest circle is marked by the grey value
tt gval which increases by steps
tt gstep with the size of the ball.

`*image`	Pointer to the binary image
`*distance`	Must initially be a copy of tt image and contains the resulting distance map after execution
`*opened`	Must initially be a copy of tt image and contains the resulting granulometry map after execution
`gval`	Grey level for the smallest size class
`gstep`	Grey level step between subsequent size classes
return value:	Number of detected size classes

image_cc *GetDistOpenCloseMap(image_cc *im, double gstep)
image_cc *GetDistOpenCloseMap(image_cc *im, double gstep, int maxdist)

Generates an opening-map (black phase) and a closing-map (white phase) of a binary image, which is a greyscale image where the grey value of each pixel indicates the 'opening size' ('closing-size') of the black (white) phase. `gstep` is the difference in grey level for the opening/closing of 1 pixel. This function is valid for isotropc pixel geometry. This function requires considerable computation time, so the maximum size for openings(closings) can be limited by `maxdist`.

`*image`	Pointer to the binary image.
`gstep`	scaling of the grey levels of the opening/closing map
`maxdist`	maximum opening size to be considered
return value:	Pointer to the resulting image

unsigned long bErodeMark(image_cc *image, int mark, int step);

Performs a dilation (0-phase) of a binary image with a circular structuring element of radius `step`. The dilated area is marked with the greylevel `mark`. The number of marked pixel is returned.

`*image`	Pointer to the binary image
`step`	Radius of structuring element
`mark`	Greylevel to mark dilated pixel
return value:	Number of marked pixel

void bLogAnd(image_cc *wnd1, image_cc *wnd2)

Logical AND relation (intersection) of two binary images.

`wnd1`	Pointer to the first image description structure.
`wnd2`	Pointer to the second image description structure.

The result is written to `wnd1`

void bLogOr(image_cc *wnd1, image_cc *wnd2)

Logical OR relation (unification) of two binary images.

`wnd1`	Pointer to the first image description structure.
`wnd2`	Pointer to the second image description structure.

The result is written to `wnd1`

void bAddition(image_cc *wnd1, image_cc *wnd2, int gv1, int gv2)

Addition of two images.

`wnd1`	Pointer to the first image description structure.
`wnd2`	Pointer to the second image description structure.
`gv1`	grey level to be set for phase in wnd1.
`gv2`	grey level to be set for phase in wnd2.

the 0-values of the two images are set to `gv1` and `gv2` respectively, their intersection is set to 0. The result is written to `wnd1`

void bHitMiss(image_cc *im, int Mx, int My, long MP, long MNP)

Hit or Miss Transform of a binary image. Structuring elemnts are still restricted to a size $<= 5 \times 5$ pixel (Mx, My $\in$ [1,2]).

`*im`	Pointer to the image.
`Mx My`	Operating window of the structuring element in x and y direction, $(2Mx+1)\times(2My+1)$
`MP`	Each nonzero bit of `MP` indicates membership to the structuring element at position x = bitpos/(2Mx+1) and y = bitpos modulo (2Mx+1) .
`MNP`	Each nonzero bit of `MNP` indicates explicit non-membership to the structuring elemnt at position x = bitpos/(2Mx+1) and y = bitpos modulo (2Mx+1). Coordinates which are neither described by `MP` nor `MNP` (the corresponding bits are zero in both variables) are not significant.

long int *bQuantRecMask(image_cc *image, int xdim,int ydim, int xul, int yul);

Calculates the frequency distribution of 16 different pixel configurations within a 2x2 square for a rectangular region. The returned pointer is input for the routines to calculate volume density, surface density, length density, curvature and Euler number.

`*im`	Pointer to the binary 2D-image.
`int xdim, ydimm`	Extension of the rectangular region [pixel].
`int xul, int yul`	Coordinates of the upper left corner.
return value:	Pointer to the 16-element array containing frequencies of pixel configurations

void SaveRGBImageSeg(image_cc *image, char *buf, int ulx, int uly, int drx, int dry)

Saves a rectangular segment of a RGB image.

`image`	Pointer to the image description structure.
`buf`	Name of the tif-image to be saved without extension.
`ulx`	x coordinate of the upper left corner of the segment
`uly`	y coordinate of the upper left corner
`drx`	x coordinate of the lower right corner
`dry`	y coordinate of the lower right corner

ddd stuff **ddd images**

void SetDddShell(image_cc *im, int thick, int val);

Writes the complete shell of thickness `thick` to `val`

`thick`	Thickness of the shell
`val`	Value written to the shell
return value:	no return value

image_cc *DddResRed(image_cc *image, int mode)

Reduces image size and herewith resolution by averaging over regions of size $(2\text{mode}+1)^2$.

`*im`	pointer to the original image.
`mode`	mode of reduction.
return value:	pointer to the resulting image.

image_cc *GetRandDDDImage(int col, int row, int dep, double rx,, double ry, double rz)

Creates a 3D grey scale image in which the values of voxels are set randomly.

`col,row,dep`	Size of the image (x,y,z).
`rx,ry,rz`	Resolution (size of voxels) in different dimensions (x,y,z).
return value:	pointer to the created image.

void DddMinMax(image_cc *im,int mode);

Minimum/Maximum filter for a 3D grey scale image which operates within a 3x3x3 window.

`*im`	pointer to the image.
`mode`	if 0 minimum, else maximum.

image_cc *DddBin(image_cc *im, unsigned char thresh);

Converts a 3D grey scale image to a 3D binary image (btd-format) according to a threshold. Values <=`thresh` are coded by 0 and by 1 else.

`im`	Pointer to the 3D grey scale image.
`thresh`	Threshold on the grey scale.
return value:	pointer to the 3D binary image (btd-format).

void DddBibin(int LOW,int HIGH,image_cc *im);

Segments an 3D grey scale image according to 2 thresholds which are regarded to be the limits of a fuzzy region of the grey scale histogram of the image (Conditional Dilation). All pixel of a grey level smaller or equal than `LOW` are written to 0 as well as all pixel having values smaller than `HIGH` and at least one direct neighbour smaller than `LOW`. This algorithm is repeated iteratively until no pixel is to be changed. The other pixel keep their original value. This function overwrites the original image.

`LOW`	lower threshold.
`HIGH`	upper threshold.
`image`	Pointer to the image.

image_cc *DddWaterShed(image_cc *image, int conmode);

Calculates the watershed lines for a 3D grey image. The different basins separated by the watershed lines are marked by different grey values, the watershed is written to 0. This may be applied to a distance map of a binary image to separate overlapping grains. The distance map is obtained by BtdGetDistOpenMap() or BtdGetDistMap().

`*image`	Pointer to the grey image
`conmode`	Connectivity mode: either 6 or 26.
return value:	Pointer to the image containing the watershed and the basins

void DddClas(image_cc *im, int Nclas, unsigned char *th, unsigned char *gval);

Transforms a 3D grey scale image by dividing the grey scale into a number (`Nclas`) of discrete classes. The upper limits of the grey values of the different classes are provided by `*th` and the values to be written for each class by `gval`.

`im`	Pointer to the 3D grey scale image.
`Nclas`	Number of different classes.
`*th`	Array of `Nclas` thresholds ∈[0,255] starting with lower values at `th[0]`.
`*gval`	Array of `Nclas` greylevels ∈[0,255] to be written for the corresponding classes.

double *DddHisto(image_cc *im, int precision);

Returns the grey-histogram of a 3D grey scale image.

`*im`	Pointer to the image.
`precision`	Only a fraction (1/`precision`) of the total number of voxels are considered.
return value:	Array with 255 elements containing the relative frequency of the corresponding grey value.

double *DddCircHisto(image_cc *im, int precision, int rad);
Same as `DddHisto` but only consideres a central cylinder of radius `rad`.

`*im`	Pointer to the image.
`precision`	Only a fraction (1/`precision`) of the total number of voxels are considered.
`rad`	Radius of the central cylinder to be considered (number of pixel).
return value:	Array with 255 elements containing the relative frequency of the corresponding grey value.

double *GetDddGreyCdf(image_cc *im)
Returns the cdf of grey levels for a 3D-grey scale image. This cdf can be used by `GetCorDDDImage` to generate a random structure accordingly.

`*im`	Pointer to the image.
return value:	Array with 256 elements describing the cdf

void DddEulerFunc(image_cc *im, int *num, double **xdat, double **ydat, int prec)
Calculates the connectivity function of a 3D grey scale image.(Take care that resolution of the image is set correctly!)

`*im`	Pointer to the image.
`*num`	Adress where the number of function value are written to.
`**xdat, **ydat`	Pointer to the adress where the function values are written to (`xdat[i]`=threshold of greylevel, `ydat[i]`= Euler number).
`prec`	Step of grey thresholds for which the Euler number is calculated. Number of function values: 255/`prec`.

image_cc *GetCorDDDImage(double *cdf, int col,int row,int dep,double rx,double ry, double rz,double cx,double cy, double cz, int mode)
Generates a random 3D greylevel structure with predefined grey-histogram, correlation length and correlation model. The maximum size is restricted to 64^3. The structure is periodic only for this maximum size. UNDER CONSTRUCTION!! NOT EVERYTHING WORKS!! USE WITH CARE!!

`*cdf`	Pointer to the 256-element array containing the cdf of grey levels.
`col, row, dep`	Size of the image (number of voxels in x, y and z).
`rx, ry, ry`	Size of the voxels in x, y and z.
`cx, cy, cz`	Correlation lengths (number of voxels in x, y and z).
`mode`	Correlation model: 0 = completely random without correlation (cx, cy, cz are ignored); 1 = Gaussian model; 2 = Mirrored gaussian (just try and you will get an idea or check the source to dig

	out that strange guy); 3 = Lorentz; 4 = Exponential, 5 = von Karman.
return value:	pointer to the generated image.

image_cc *GetCorDDDImage2(double *cdf, int col, int row, int dep, double rx, double ry, double rz, double cx, double cy, double cz, int mode);

Returns a random 3D grey image with predefined grey histogram and/or correlation lengths. The image is periodic only for dimensions = power of 2 UNDER CONSTRUCTION!! NOT EVERYTHING WORKS!! USE WITH CARE!!

`*cdf`	Pointer to the 256-element array containing the cumulative density function of grey levels.
`col, row, dep`	Size of the image (number of voxels in x, y and z).
`rx, ry, ry`	Size of the voxels in x, y and z.
`cx, cy, cz`	Correlation lengths (number of voxels in x, y and z).
`mode`	Correlation model: 0 = equal grey distribution without any correlation (cx,cy,cz and cdf have no meaning here) 1 = equal grey distribution with correlation (correlation lengths of resulting image will be different from cx, cy, cz!) 2 = predefined greylevel (cdf) distribution and gaussian correlation (correlation lengths of resulting image will be different from cx, cy, cz!) 3 = Gaussian covariance (cdf have no meaning here and in the following modes) 4 = Lorentz covariance 4 = Exponential covariance 5 = von Karman covariance
return value:	pointer to the generated image.

double *GetDddAcov(image_cc *im, double *corl, int lag, int mode)

Calculates the autocovariance function and the correlation length of a 3D greylevel image the correlation length is written to `*corl`. The first 3 bits of `mode` indicate which directions (x,y,z) are to be considered.

`*im`	Pointer to the image.
`*corl`	Adress where the correlation length is written to.
`lag`	Maximum distance considered (number of pixel).
`mode`	indicates which directions are to be considered (bit1=x, bit2=y, bit3=z)
return value:	pointer to an arry of dimension `lag` where the autocovariance function is stored.

void DddDrawCylinder(image_cc *image, int xmid, int ymid, int rad, int len, int val);

Draws a cylinder at center tt xmid/xmid with length `len` and radius `rad` and value `val`.

`*image`	Pointer to the 3d binary image

`xmid, xmid`	coordinates of cylinder center
`len`	length of cylinder
`rad`	radius of cylinder
`val`	value to be written for the cylinder

btd stuff **btd images**

void BtdErodeFilter(image_cc *im, int step, int mode)
Performs an erosion of the phase `mode` by a spherical structuring element of radius `step`. Note that an erosion of phase 1 corresponds to a dilation of phase 0. The outer shell of the eroded image (where the structuring element cannot be placed entirely into the image volume) is set to 0. This function is a filter, meaning the original image is lost after this operation (see *BtdErode()).

`im`	Pointer to the 3D image
`step`	radius of the structuring element [voxels]
`mode`	phase to be eroded [0,1]

image_cc *BtdErode(image_cc *im, int step, int mode)
Performs an erosion of the phase `mode` by a spherical structuring element of radius `step`. Note that an erosion of phase 1 corresponds to a dilation of phase 0. The outer shell of the eroded image (where the structuring element cannot be placed entirely into the image volume) is cut off.

`im`	Pointer to the 3D image
`step`	radius of the structuring element [voxels]
`mode`	phase to be eroded [0,1]
return value:	p

ointer to the 3D image containing erroded subvolume.

image_cc *BtdErodeMir(image_cc *im, int step, int mode)
Same as BtdErode except that the shell is not cut off. To calculate the erosion at the border of the image, it is enlarged by mirroring the structure at the boundaries.

`*im`	Pointer to the 3D image
`step`	Radius of the structuring element[voxels]
`mode`	Phase to be eroded [0,1]
return value:	Pointer to the image containing the eroded structure

image_cc *BtdErodeMirDouble(image_cc *im, double step, int mode)
Same as BtdErodeMir except that the radius of the structuring element is of type `double` (typically 0.5).

`*im`	Pointer to the 3D image
`step`	Radius of the structuring element[voxels]
`mode`	Phase to be eroded [0,1]
return value:	Pointer to the image containing the eroded structure

image_cc *BtdOpen(image_cc *im, int step, int mode)

Performs an opening (erosion followed by dilation) or closing (dilation followed by erosion) of a binary 3D image using a spherical structuring element of radius `step`. The outer shell of the eroded image (where the structuring element cannot be placed entirely into the image volume) is cut off.

`im`	Pointer to the 3D image
`step`	radius of the structuring element [voxels]
`mode`	opening or closing [0,1]
return value:	p

ointer to the 3D image containing opened (closed) 3D-image.

image_cc *BtdGetDistMap(image_cc *image, int *n, int gval, int gstep);

Converts a binary 3d image (btd-format) to its distance map (ddd-format): Each voxel in phase 1 is written to a grey value which corresponds to the distance of that voxel to phase 0. The closest distance gets grey value `gval` which increases ba steps `gstep` with distance. The phase 0 is not changed. The total number of distance classes is written to `n`. Note that it is a good idea to chose the parameters such that $n \cdot gstep < 255 - gval$. The resulting distance map can be used as input to a watershed segmentation e.g. to separate sintered grains.

`*image`	Pointer to the 3D binary image (btd-format)
`*n`	Number of detected distance classes
`gval`	Grey level for the first distance class
`gstaep`	Grey level step between adjacent distance classes
return value:	Pointer to the ddd-image containing the distance map

int BtdGetDistOpenMap(image_cc *image, image_cc *distance, image_cc *opened, int gval, int gstep)

As BtdGetDistMap(), this function converts a binary 3d image (btd-format) to its distance map (ddd-format). Additionally a 'granulometry map' is calculated where each voxel in phase 1 is written to a grey value which corresponds to the diameter of the maximum ball that can be placed inside phase 1 at that location (corresponding to the 'opening size'). The smallest ball is marked by the grey value `gval` which increases by steps `gstep` with the size of the ball.

`*image`	Pointer to the 3D binary image (btd-format)
`*distance`	Must be a copy of `image` in ddd-format and contains the resulting distance map after execution
`*opened`	Must be a copy of `image` in ddd-format and contains the resulting distance map after execution
`gval`	Grey level for the smallest size class
`gstep`	Grey level step between subsequent size classes
return value:	Number of detected size classes

int BtdGetDistOpenMapDouble(image_cc *image, image_cc *distance, image_cc *opened, int gval, int gstep, double step)

As BtdGetDistOpenMap(), but the diameter of the spherical structuring element is incremented by steps of 0.5 to get a better resolution of the size distribution. This

function converts a binary 3d image (btd-format) to its distance map (ddd-format). Additionally a 'granulometry map' is calculated where each voxel in phase 1 is written to a grey value which corresponds to the diameter of the maximum ball that can be placed inside phase 1 at that location (corresponding to the 'opening size'). The smallest ball is marked by the grey value
tt gval which increases by steps
tt gstep with the size of the ball.

`*image`	Pointer to the 3D binary image (btd-format)
`*distance`	Must be a copy of tt image in ddd-format and contains the resulting distance map after execution
`*opened`	Must be a copy of tt image in ddd-format and contains the resulting distance map after execution
`gval`	Grey level for the smallest size class
`gstep`	Grey level step between subsequent size classes
`step`	increment for structuring element (typically 0.5)
return value:	Number of detected size classes

void SetBtdShell(image_cc *im, int dx , int dy, int dz, int mode)

Writes the shell of a 3D image with the thickness of `dx, dy, dz` [voxels] to the value `mode`.

`im`	Pointer to the 3D image
`dx, dy, dz`	thickness of the shell in different dimensions [voxels]
`mode`	Value to be written to the shell [0,1]

double BtdEuler(image_cc *im);

Claculates the volumetric 3D Euler number. Note that the resolutions im− >ncol, im− >nrow and im− >nbits must be set to a meaningful value. The result is given in the corresponding unit [$1/L^3$].

`*im`	Pointer to the 3D binary image.
return value:	Volumetric Euler number L^{-3})

double BtdVolSurf(image_cc *im, double *vv, double *sv);

Claculates the volume density (vv) and surface density (sv).

`*im`	Pointer to the 3D binary image.
`*vv, *sv`	Addresses where the results are written to
return value:	Volume of the sample in units of im-¿rcol

long int *BtdQuant(image_cc *image);

Calculates the frequency distribution of 255 different voxel configurations in a 2x2x2 cube. The returned pointer is input for the routines to calculate volume density, surface density, mean curvature and Euler number.

`*im`	Pointer to the binary 3D-image.

return value:	Pointer to the 255-element array containing frequencies of voxel configurations

long int *BtdQuantMask(image_cc *image, int xm, int ym, int zm, char *mask);

Calculates the frequency distribution of the 255 different voxel configurations within a 2x2x2 cube for a region identified by `mask`. The format of `mask` corresponds to that returned by GetSphereElement. The returned pointer is input for the routines to calculate volume density, surface density, mean curvature and Euler number.

`*im`	Pointer to the binary 3D-image.
`int xm, ym, zm`	Coordinates of the center of the region described by `mask`.
`*mask`	description of the region to be analyzed.
return value:	Pointer to the 255-element array containing frequencies of voxel configurations

double BtdVoldens(long int *h);

Returns the volume density of the phase coded by [1]

`*h`	Pointer to the array of voxel configurations obtained by BtdQuant()
return value:	volume density [-]

double BtdSurfdens(image_cc *image, long int *h);

Returns the surface density of a binary structure

`*h`	Pointer to the array of voxel configurations obtained by BtdQuant()
return value:	surface density [L^2/L^3] in units of image-¿rcol

double BtdMeancurv(image_cc *image, long int *h);

Returns the mean curvature of a binary structure

`*h`	Pointer to the array of voxel configurations obtained by BtdQuant()
return value:	mean curvature in units of image-¿rcol

double BtdEuler6(image_cc *image, long int *h);

Returns the volumetric Euler number of a 3D binary structure considering 6-connectivity of phase [1]

`*h`	Pointer to the array of voxel configurations obtained by BtdQuant()
return value:	Euler number [$1/L^3$] in units of image-¿rcol

double BtdEuler26(image_cc *image, long int *h);

Returns the volumetric Euler number of a 3D binary structure considering 26-connectivity of phase [1]

`*h`	Pointer to the array of voxel configurations obtained by BtdQuant()
return value:	Euler number [$1/L^3$] in units of image-¿rcol

RGB stuff **RGB images**

image_cc *ChangeResolutionRGB(image_cc *image, int newx, int newy);
Returns the pointer to a rescaled RGB-image. The new dimensions are `newx` and `newy` in x and y direction respectively.

`*image`	Pointer to the original image
`newx, newy`	new dimensions in x and y direction
return value:	pointer to the rescaled image

image_cc *TurnRGBImage(image_cc *im, double grad)
rotates a RGB image by the angle `grad`

`*image`	Pointer to the original image
`grad`	angle to be turned (degree)
return value:	pointer to the rescaled image

image_cc *FlipRGBImage(image_cc *im);
Flip the RGB-image in x direction.

`*image`	Pointer to the original image
return value:	pointer to the fliped image

void WRGBPixel(image_cc *in, int x,int y,unsigned char rval,unsigned char gval,unsigned char bval);
Writes the RGB values at given coordinates.

`*image`	Pointer to the image
`x, y`	coordinates
`rval, gval, bval`	written rgb values
return value:	no return