Deconv Library User Manual

Author: Yuansheng Sun

Contact: yuansheng.sun@gmail.com

March, 2007

Department of Biomedical Engineering

University of Iowa

TABLE OF CONTENTS

INTRODUCTION TO THE DECONV LIBRARY

Deconvolution, a computationally intensive image processing technique, has become an established procedure for restoring quantitative information from sectioning images acquired using fluorescence microscopy and also for improving images' resolution and signal-to-noise ratio (SNR). The *Deconv* library was designed as open-source to provide constrained iterative deconvolution routines for 3-D wide-field fluorescence microscopy imaging and is compatible with the GPL (http://www.gnu.org/licenses/gpl.txt). This library was written in C++ and includes the classes designed for handling 3-D or 2-D data sets, calculating FFT, generating a 3-D PSF, and deconvolving a 3-D image data set acquired from a standard wide-field fluorescence microscope. The purpose of this manual is to explain the usage of the *Deconv* library based on the classes listed as follows:

Table 1: The classes in the *Deonv* library

Functionality	Classes	Files
Handle 3-D or 2-D data sets (Chapter 1)	CCube CSlice	CCube.h, MYcube.h, MYcube.cc CSlice.h, MYimage.h, MYpgm.h, MYpgm.cc
Calculate FFT (Chapter 2)	FFTW3_FFT	FFTW3fft.h, FFTW3fft.cc, SHIFTfft.h, SHIFTfft.cc
PSF (Chapter 3)	FluoPSF Fluo3DPSF FluoRZPSF	FluoPSF.h, FluoPSF.cc Fluo3DPSF.h, Fluo3DPSF.cc FluoRZPSF.h, FluoRZPSF.cc
Deconvolution (Chapter 4)	deconvolver LWCGdeconvolver LWdeconvolver CGdeconvolver EMdeconvolver	deconvolver.h, deconvolver.cc LWCGdeconvolver.h, LWCGdeconvolver.cc LWdeconvolver.h, LWdeconvolver.h CGdeconvolver.h, CGdeconvolver.cc EMdeconvolver.h, EMdeconvolver.cc

CHAPTER 1 - DATA

The "CCube" and "CSlice" template classes introduced in this chapter was designed for handling a 3-D data set (cube) and a 2-D data set (slice), respectively. The dimensions of a cube or a slice are given below:

Table 2: The dimensions of a cube (a 3-D data set) or a slice (a 2-D data set)

Template Class	Dimensions	Private Members
CCube for a 3-D data set	Dimension X – length, the fastest varying dimension Dimension Y – width Dimension Z – height, the slowest varying dimension	_length _width _height
CSlice for a 2-D data set	Dimension X – width, the fast varying dimension Dimension Y – height, the slow varying dimension	_width _height

The data of a cube or a slice is stored in a one-dimensional array as a single contiguous block in row-major order:

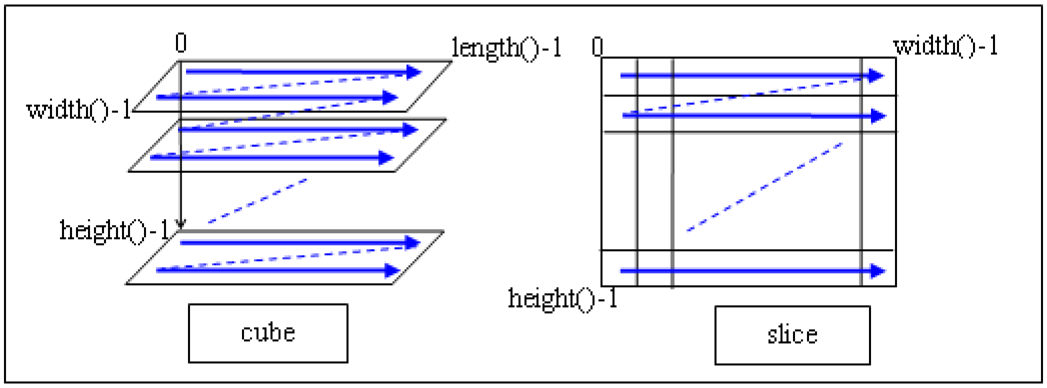

Fig. 1 – The data of a cube or a slice data is stored in a one-dimensional array in row-major order. "**width()**", "**length()**" and "**height()**" are the three public member functions in the CCube class and return the width, length and height of a cube, respectively. While, "**width()**" and "**height()**" are the two public member functions in the CSlice class and return the width and height of a slice.

CCube

The constructors, operators and public member functions provided in the "CCube" template class are listed below:

Table 3: The CCube class

Purpose	Public Member Functions
Initialize a cube	CCube (const CCube& cube) CCube (int length = 0, int width = 0, int height = 0, T* data = NULL) CCube& operator = (const CCube& cube) void init (int length = 0, int width = 0, int height = 0, T* data = NULL)
Compare cubes	bool operator == (const CCube& cube) const
Check a cube	bool Valid (bool throw_if_not = false)
Free a cube	void free ()
Get a cube's dimensions and data array	int length () const; int width () const; int height () const int size () const T* data () const
Get and set a cube's voxel	T& operator () (int x, int y, int z) void setvoxel (int x, int y, int z, T value) void getvoxel (int x, int y, int z, T& value)
Get and set a cube's slice	int getslice (int slice_index , int slice_plane , CSlice<T>& slice) int setslice (int slice_index , int slice_plane , CSlice<T> slice)
Set a cube	void fillcube (T value)
Draw objects in a cube	int draw_cylinder (int cx, int cy, int cz, int rx, int ry, int hz, T value) int draw_ellipse (int cx, int cy, int cz, int rx, int ry, int rz, T value)
Read and save a cube	int read (const char * filename) void write (const char * filehead)

Crop a cube	int length_crop (int index0, int index1) int width_crop (int index0, int index1) int height_crop (int index0, int index1)
Pad a cube	int length_pad (int num0, int num1, T value) int width_pad (int num0, int num1, T value) int height_pad (int num0, int num1, T value)
Get statistics of a cube	double cubemaxval (); double cubeminval (); double cubemeanval (); double cubevarval (); double cubesumval ()
Shift a cube	int shift ()

A cube is usually initialized using the constructor "**CCube(…)**" or the member function "**init(…)**" [Table 3], where *length*, *width* and *height* are the three dimensions of the cube (see Table 2) and *data* is the pointer that points to a one-dimensional array whose size is equal to the product of length, width, and height. An initialized cube can be destroyed using the member function "**free()**".

Accessing a voxel in a cube can be made by using either the operator "(…)" or the member functions "**setvoxel(…)**" and "**getvoxel(…)**" [Table 3], where *x*, *y* and *z* are the indices of the voxel along the cube's length, width and height, respectively. The difference lies in that the boundary check, that made is to verify if *x*, *y* or *z* is out of its range, will be performed when the member functions are used. However, this check will not be carried out by using the operator.

To assign all voxels in a cube with a same value, one can use the member function "**fillcube(…)**" [Table 3], where *value* is the assigned value.

Accessing a slice in a cube can be made using the member functions "**getslice(…)**" and "**setslice(…)**" [Table 3], where *slice_plane* indicates which dimension the slice is along (1–length, 2-width and 3-height); *slice_index* is the index number of the slice along the chosen dimension; and *slice* is a CSlice object (the CSlice template class will be explained below) that is used here as the container for the slice.

The data of a cube can be saved as two files with a same basename and different extensions in a computer disk using the member function "**write(…)**" [Table 3], where *filehead* is the basename. One is the header file whose extension is "hdr" and stores the three dimensions of the cube in texts. The other is the data file storing the cube's data in a binary stream of row-major order and its extension depends on the data type: "u8" for 8-bit unsigned char, "i16" for 16-bit unsigned integer, "f32" for 32-bit floating real and "f64" for 64-bit floating real. The data of a cube saved by the header and data files can also be read as a cube into a program using the member function "**read(…)**", where *filehead* is the basename of the two files.

The following utilities are also provided by the CCube class: 1 - Draw a homogeneous elliptic cylinder object or a homogeneous ellipse object in a cube [Fig. 1]. 2 - Crop or pad a cube [Fig. 2]. 3 - Shift a PSF from its original shape to a shape that is required for a deconvolution process [Fig. 3].

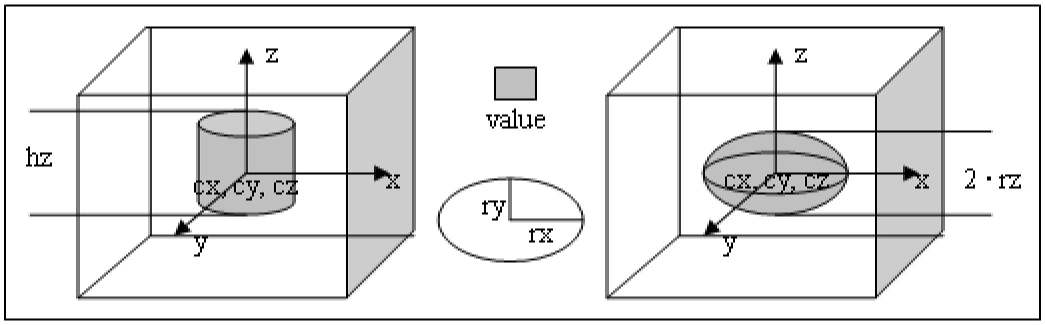

Fig. 2 – A homogeneous elliptic cylinder (left) object is drawn in a cube using the member function "**draw_cylinder(…)**" [Table 3], where *cx, cy,* and *cz* are the indices along the cube's length, width and height and the point at (*cx, cy, cz*) is the geometric center of the cylinder. *hz* is the cylinder's height, and *rx* and *ry* are the radii of its cross section along the cube's length and width, respectively. A homogeneous ellipse object (right) is drawn in a cube using the member function "**draw_ellipse(…)**" [Table 3], where *cx*, *cy*, and *cz* are the indices along the cube's length, width and height and the point at (*cx, cy, cz*) is the geometric center of the ellipse. *rx*, *ry* and *ry* are the radii of the ellipse along the cube's length, width and height, respectively. *value* is the filled value within the ellipse object or the elliptic cylinder object.

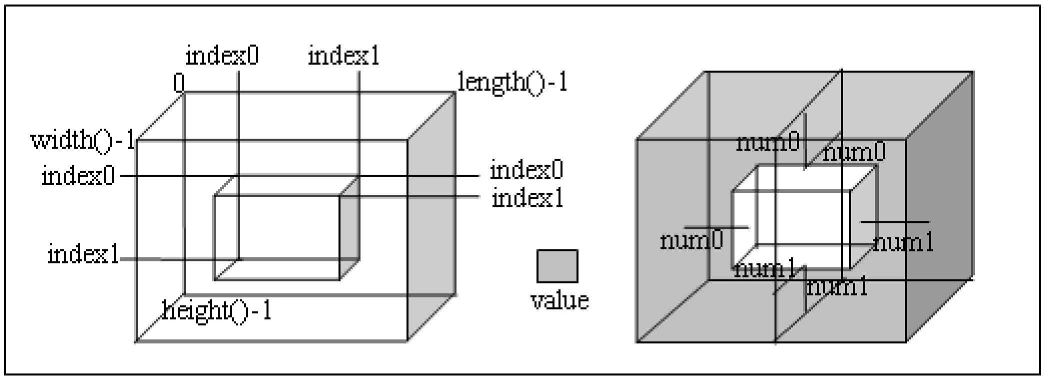

Fig. 3 – A cube can be cropped along its length, width or height using the member functions "**length_crop(…)**", "**width_crop(…)**", or "**height_crop(…)**" [Table 3], where *index0* and *index1* are the two indices along a dimension and the parts before *index0* and after *index1* along the dimension will be removed. A cube can also be padded with a same value along its length, width, or height using the member functions "**length_pad(…)**", "**width_pad(…)**", or "**height_pad(…)**" [Table 3], where *num0* and *num1* slices will be added before the first index and after the last index along a dimension and *value* is the filled value in the added slices.

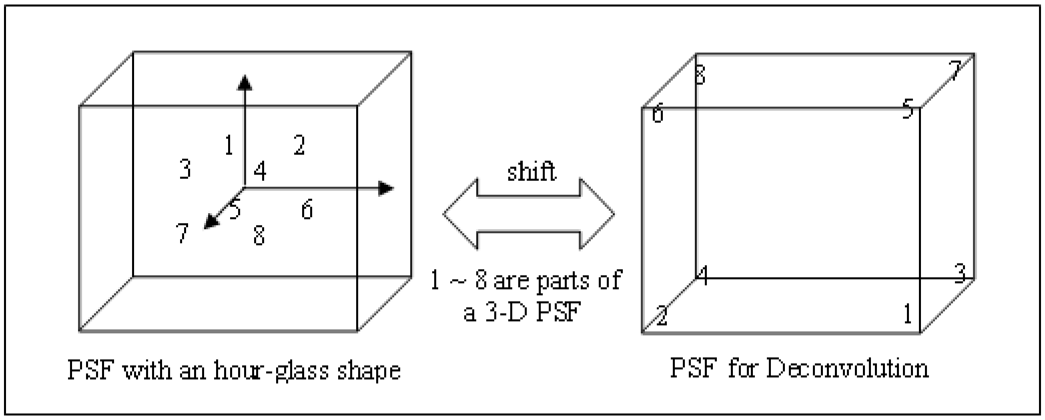

Fig. 4 – The values of a 3-D PSF that naturally has an hour-glass shape, distribute in the center of a cube (left) and these values must be shifted into the eight corners of the cube (right) for the deconvolution process using the member function "**shift()**" [Table 3]. The 3-D PSF in its natural shape is divided into 8 parts (1~8) which are then shifted into the eight corners of the cube.

The statistics of the data in a cube, such as its maximum, minimum, mean, standard deviation and summation can be given by the member functions "**cubemaxval()**", "**cubeminval()**", "**cubemeanval()**", "**cubevarval()**", and "**cubesumval()**" [Table 3], respectively.

CSlice

The constructors, operators and public member functions provided in the "CSlice" template class are listed below:

Table 4: The CSlice class

Purpose	Public Member Functions
Initialize a slice	CSlice (const CSlice& slice) CSlice (int width = 0, int height = 0, T* data = NULL) CSlice& operator = (const CSlice& slice) void init (int width = 0, int height = 0, T* data = NULL)
Compare slices	bool operator == (const CSlice& slice) const
Check a slice	bool Valid (bool throw_if_not = false)
Free a slice	void free()
Get a slice's dimensions and data array	int width () const; int height () const int size () const T* data () const
Get and set a slice's pixel	T& operator() (int x, int y) void setpixel (int x, int y, T value) void getpixel (int x, int y, T value)
Set a slice	void fillslice(T value)
Read and save a slice	int read (const char * filename) void write (const char * filehead)
Crop or pad a slice	int crop (int newx, int newy, int cx = 0, int cy = 0) int pad (int newx, int newy, T value)
Get a slice's statistics	double slicemaxval(); double sliceminval(); double slicemeanval(); double slicevarval(); double slicesumval()

A slice is usually initialized using the constructor "**CSlice(…)**" or the member function "**init(…)**" [Table 4], where *width* and *height* are the two dimensions of the slice (see Table 2) and *data* is the pointer that points to a one-dimensional array whose size is equal to the product of width and height. An initialized slice can be destroyed using the member function "**free()**".

Accessing a pixel in a slice can be made by using either the operator "(…)" or the member functions "**setpixel(…)**" and "**getpixel(…)**" [Table 4], where *x* and *y* are the indices of the pixel along the slice's width and height, respectively. The difference lies in that the boundary check, that made is to verify if *x* or *y* is out of its range, will be performed when the member functions are used. However, this check will not be carried out by using the operator.

To assign all pixels in a slice with a same value, one can use the member function "**fillslice(…)**" [Table 4], where *value* is the assigned value.

The data of a slice can be saved as a PGM image [Table 5] in a computer disk using the member function "**write(…)**" [Table 4], where *filehead* is the basename. A PGM image can also be read as a slice into a program using the member function "**read(…)**", where *filename* is the full name of the PGM image.

Table 5: The format of the 8-bit or 16-bit PGM

Greylevel	Format
8-bit	P5 ImageWidth ImageHeight Greylevel // no more than 255 unsigned char data in a binary stream
16-bit	P5 ImageWidth ImageHeight Greylevel //between 255 and 65535 unsigned integer data in a binary stream

The utility of cropping or padding a slice is provided by the CSlice class [Fig. 5].

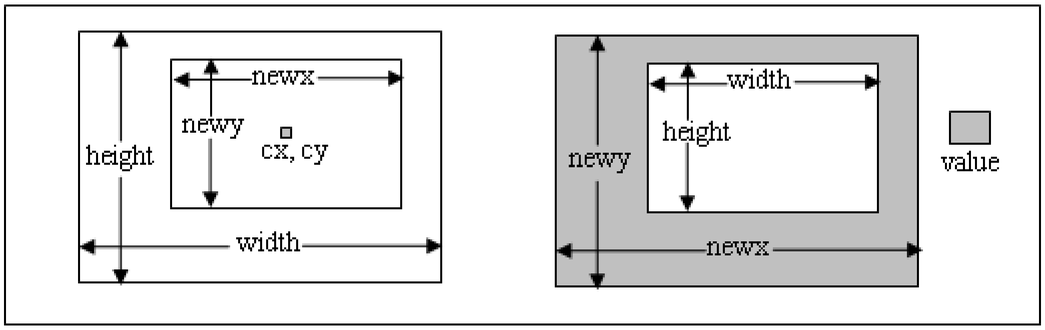

Figure 5 – A slice can be cropped along its width or height using the member function “**crop(…)**” [Table 4], where *newx* and *newy* are the width and height of the new slice after cropping, and *cx* and *cy* are the two indices along the slice’s width and height and the point at (*cx*, *cy*) is center of the new slice after cropping. A slice can also be padded with a same value along its width or height using the member function “**pad(…)**” [Table 4], where *newx* and *newy* are the width and height of the new slice after padding, and *value* is the filled value in the added slices.

The statistics of the data in a slice, such as its maximum, minimum, mean, standard deviation and summation can be given by the member functions “**slicemaxval()**”, “**sliceminval()**”, “**slicemeanval()**”, “**vslicevarval()**”, and “**slicesumval()**” [Table 4], respectively.

CHAPTER 2 - FFT

The routines used for calculating a fast Fourier transform (FFT) in the *Deconv* library were programmed using the FFTW3 library (www.fftw3.org).

FFT class for deconvolution

In the *Deconv* library, the “FFTW3_FFT” class is dedicated for the usage in the deconvolver classes [Table 1], and it only supports the 3-D real-to-complex/forward or complex-to-real/backward transforms. The constructor and the public member functions provided by this class are listed as follows:

Table 6: The FFTW3_FFT class

Purpose	Public Member Functions
Initialize a 3-D FFT plan	FFTW3_FFT (int DimX, int DimY, int DimZ, bool IsForward, bool IsDouble, int status)
Execute a 3-D FFT as planed	void execute (double * buf1, double * buf2, double * buf3) void execute (float * buf1, float * buf2, float * buf3)
Get FFT’s dimensions	int DimX (); int DimY (); int DimZ () int FFTsize ()
Get FFT’s plan information	int status (); bool IsDouble (); bool IsForward ()

Two steps are required for calculating a 3-D FFT using the “FFTW3_FFT “class: 1, create a plan and 2, perform the FFT on a given data set as planned. A plan is generated using the member function “**FFTW3_FFT(…)**” [Tabel 6], where *DimX* (the fastest varying dimension), *DimY*, and *DimZ* (the slowest varying dimension) are the three dimensions of a given data set. *IsDouble* indicates whether the input data is double (true) or float (false). *IsForward* tells whether the Fourier transform is forward/complex-to-real (true) or backward/real-to-complex (false). *status* indicates the requirements for the data arrays given to the member function "**execute(…)**" [Table 6] in step 2, where *buf1*, *buf2* and *buf3* are the one-dimensional double or float arrays storing the input and output data in raw-major order. The requirements are explained by the two following tables:

Table 7: Requirements of the data arrays by a forward FFTW3_FFT plan

status	*buf1* Input Real	*buf2* Output Real	*buf3* Output Imaginary	Overlap
1	Fullsize	Fullsize	Fullsize	NO
2	Fullsize	Fullsize	Fullsize	Buf1 and buf2
3	Fullsize	FFTsize	FFTsize	NO

Table 8: Requirements of the data arrays by a backward FFTW3_FFT plan

status	*buf1* Input Real	*buf2* Input Imaginary	*buf3* Output real	Overlap
1	Fullsize	Fullsize	Fullsize	NO
2	Fullsize	Fullsize	Fullsize	Buf1 and buf3
3	FFTsize	FFTsize	Fullsize	NO

"Fullsize" and "FFTsize" indicate the size of an array. "Fullsize" is equal to the product of *DimX*, *DimY* and *DimZ*. Taking the advantage of the symmetry of the real-to-complex or the complex-to-real transform, the space of "FFTsize" is enough for holding the complex data. "FFTsize" is equal to "$DimX \cdot DimY \cdot (DimZ/2+1)$" for *DimZ* is even and it is equal to "$DimX \cdot DimY \cdot (DimZ+1)/2$" for *DimZ* is odd. Overlap of the two arrays means that the input array will be overwritten by the output data.

FFT functions for other usages

To calculate a single FFT using the *Deconv* library for other purposes rather in a deconvolution process, it is recommended to use the C functions “**fft1d(…)**”, “**fft2d(…)**” and “**fft3d(…)**” listed in the following table:

Table 9: FFT functions

Dimension	Functions
3D	fft3d (int DimX, int DimY, int DimZ, bool IsForward, bool IsShift, double * in_re, double * in_im, double * out_re, double * out_im) fft3d (int DimX, int DimY, int DimZ, bool IsForward, bool IsShift, float * in_re, float * in_im, float * out_re, float * out_im)
2D	fft2d (int DimX, int DimY, bool IsForward, bool IsShift, double * in_re, double * in_im, double * out_re, double * out_im) fft2d (int DimX, int DimY, bool IsForward, bool IsShift, float * in_re, float * in_im, float * out_re, float * out_im)
1D	fft1d (int DimX, bool IsForward, bool IsShift, double * in_re, double * in_im, double * out_re, double * out_im) fft1d (int DimX, bool IsForward, bool IsShift, float * in_re, float * in_im, float * out_re, float * out_im)

DimX, *DimY*, *DimZ*, and *IsForward* have the same meanings as described in the “FFTW3_FFT” class. *IsShift* indicates whether to shift (true) the DC to the center in the frequency domain or not (false). *in_im*, *out_re*, *out_im*, and *in_re* with an equivalent size are the one-dimensional arrays storing the input and output data in row-major order. This size is equal to *DimX* in “**fft1d(…)**”, and it is equal to $DimX \cdot DimY$ in “**fft2d(…)**”, and it is equal to $DimX \cdot DimY \cdot DimZ$ in “**fft3d(…)**”. *in_re* and *out_re* are the arrays storing the input and output real data and can be overlapped by passing a same array. *in_im* and *out_im* are the arrays storing the input and output imaginary data and can be overlapped by passing a same array.

CHAPTER 3 - PSF

PSF routines in the *Deconv* library were programmed based on a computational PSF model derived by Gibson and her advisor Lanni [1,2,3]. Their model was set up based on geometrical- and physical-optics theory and was used to predict the image of a point-source object under a variety of optical conditions. There are three classes associated with the PSF routines in the *Deonv* library. Both the “Fluo3DPSF” class and the “FluoRZPSF” class are derived from the “FluoPSF” class [Table 1].

Physical properties of the objective lens coupled to a microscope imaging system are important to simulate a 3-D PSF using the PSF routines. An example of reading these properties from an objective lens is given in Fig. 6, where 1.40 is the numerical aperture (NA) of the objective lens and 0.21 is its working distance (WD). Usually, the manufacturer of an objective lens suggests a type of coverslip with the specific thickness (0.17 mm in Fig. 6) and a type of immersion medium with the specific refractive index (1.40 in Fig. 6) for optimal imaging through the objective lens. These properties together with the properties of the actually used coverslip and immersion medium can be given to the PSF routines as parameters and will influence the simulation of the 3-D PSF.

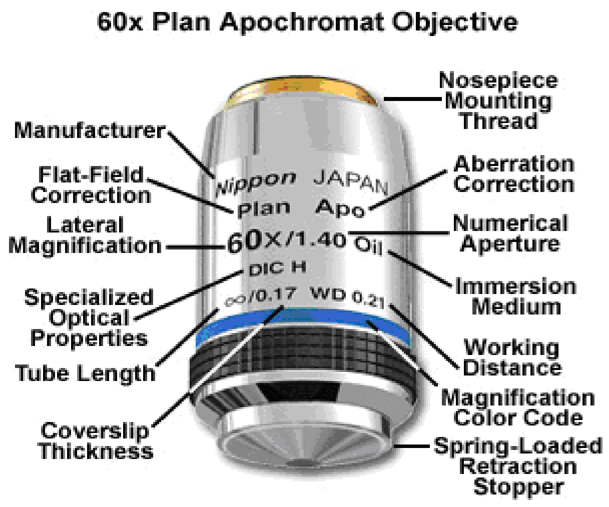

Fig. 6 – Shown is an example of reading the physical properties of an objective lens (courtesy from http://www.micro.magnet.fsu.edu/primer/anatomy/specifications.html).

Simulate a 3-D PSF using Fluo3DPSF

A 3-D PSF is calculated based on the following equations:

$$PSF(i,j,k) = \left| \int_0^{NA^2} J_0\left\{ WN \cdot \sqrt{\eta} \cdot \sqrt{x(i)^2 + y(j)^2} \right\} \cdot \exp\left\{ \sqrt{-1} \cdot OPD(\eta,k) \right\} \cdot d\eta \right|^2$$

$$OPD(\eta,k) = z(k) \cdot \sqrt{n_{imm}{}^2 - \eta} - WD \cdot \left\{ \sqrt{n_{imm^*}{}^2 - \eta} - \frac{n_{imm}}{n_{imm^*}} \sqrt{n_{imm}{}^2 - \eta} \right\} +$$

$$t_g \cdot \left\{ \sqrt{n_g{}^2 - \eta} - \frac{n_{imm}}{n_g} \sqrt{n_{imm}{}^2 - \eta} \right\} - t_{g^*} \cdot \left\{ \sqrt{n_{g^*}{}^2 - \eta} - \frac{n_{imm}}{n_{g^*}} \sqrt{n_{imm}{}^2 - \eta} \right\} \qquad (1)$$

$$x(i) = (i - N_x/2 - 1) \cdot \delta_x,\ 0 \le i \le N_x\ \text{-}1;$$
$$y(j) = (j - N_y/2 - 1) \cdot \delta_y,\ 0 \le j \le N_y\ \text{-}1;$$
$$z(k) = (k - N_z/2 - 1) \cdot \delta_z,\ 0 \le k \le N_z\ \text{-}1.$$

$$\text{Sum PSF}:\ PSF_W = \sum_{k=0}^{N_z-1} \left\{ \sum_{j=0}^{N_y-1} \left\{ \sum_{i=0}^{N_x-1} PSF(i,j,k) \right\} \right\} \qquad (2)$$

$$\text{Weight PSF}: PSF(i,j,k) = PSF(i,j,k) / PSF_W \qquad (3)$$

η is derived from the normalized radius of the rear pupil of the objective lens.

i, j and ***k*** are indices of a PSF voxel along x, y and z (the optical axis), respectively.

N_x, N_y and N_z are the PSF's dimensions along x, y and z (the optical axis), respectively.

δ_x and δ_y are x- and y-calibrations (μm/pixel), respectively.

δ_z is the sectioning interval which is the distance (in μm) between two adjacent sections.

WN* = 2*π*/*λ is the wave number and ***λ*** is the wavelength (in μm) of the emitted light.

NA is the numerical aperture (NA) of the objective lens.

WD is the working distance (WD) of the objective lens.

OPD is the optical path difference (OPD) function and is set up based on the optics used.

n_{imm} or n_{imm^*} is the refractive index of the immersion medium actually used for the objective or suggested by the manufacturer of the objective lens, respectively.

n_g, t_g or n_{g^*}, t_{g^*} are the refractive index and the thickness of the actually used coverslip or the one suggested by the manufacturer of the objective lens, respectively.

Initialize a 3-D PSF using

- Fluo3DPSF (float *NA*, float λ, float n_{imm}, float δ_x, float δ_y, float δ_z)
- void init (float *NA*, float λ, float n_{imm}, float δ_x, float δ_y, float δ_z)
- void setImmersionMedium_Mismatch (float n_{imm*}, float n_{imm}, float *WD*)
- void setCoverSlip_Mismatch (float n_{g*}, float t_{g*}, float n_g, float t_g)

where the variables used above were explained in Eqs. 1~3. The member function "**setImmersionMedium_Mismatch(…)**" is used only if the immersion medium for the objective lens is improperly used. While, the usage of the member function "**setCoverSlip_Mismatch(…)**" is needed if the coverslip is improperly used.

Create a 3-D PSF using

- void create (int N_x, int N_y, int N_z, float * *psf*, bool *Check* = true)
- void create (int N_x, int N_y, int N_z, double * *psf*, bool *Check* = true)

where the variables N_x, N_y, and N_z were explained in Eqs. 1~3 and *psf* is a one-dimensional array whose size is $N_x \cdot N_y \cdot N_z$ and this array stores the data of the generated 3-D PSF in the manner for a deconvolution process (See Fig. 4) in row-major order. *Check* is used to indicate whether to print (true) the progress of the PSF generation in the terminal or not (false).

Export the profile of the generated 3-D PSF profile using

- exportProfile (const char * *filename*)

where *filename* is the fullname of the output file that saves the information about the generated 3-D PSF in texts.

Generate a 2-D RZ PSF using FluoRZPSF

Since a 3-D PSF is cylindrically symmetric, its x- and y-dimensions can be combined together and represented by one variable r such that $r^2 = x^2 + y^2$. The dimension where the variable r lies in is called the radial dimension. The use of this radial dimension (r) and the axial dimension (z) make a new coordinate system – the cylindrical coordinate system used for representing a 3-D PSF. The PSF calculated based on this 2-D cylindrical coordinate system is named as the RZ_PSF. The simulation of a 3-D PSF can be achieved by first generating a RZ_PSF table in a 2-D cylindrical coordinate system (Eq. 4) and then calculating the 3-D PSF in a 3-D (x, y, z) geometrical coordinate system employing the RZ_PSF table lookups and the spline interpolations (Eq. 5).

$$\begin{aligned}
&RZ\_PSF(i,j) = \left| \int_0^{NA^2} J_0\left\{WN \cdot \sqrt{\eta} \cdot r(i)\right\} \cdot \exp\left\{\sqrt{-1} \cdot OPD(\eta, j)\right\} \cdot d\eta \right|^2 \\
&OPD(\eta, j) = z(j) \cdot \sqrt{n_{imm}{}^2 - \eta} - WD \cdot \left\{ \sqrt{n_{imm*}{}^2 - \eta} - \frac{n_{imm}}{n_{imm*}} \sqrt{n_{imm}{}^2 - \eta} \right\} + \\
&t_g \cdot \left\{ \sqrt{n_g{}^2 - \eta} - \frac{n_{imm}}{n_g} \sqrt{n_{imm}{}^2 - \eta} \right\} - t_{g*} \cdot \left\{ \sqrt{n_{g*}{}^2 - \eta} - \frac{n_{imm}}{n_{g*}} \sqrt{n_{imm}{}^2 - \eta} \right\} \\
&r(i) = (i - M_r/2 - 1) \cdot \kappa_r,\ 0 \le i \le M_r\text{ -1};\ z(j) = (j - M_z/2 - 1) \cdot \kappa_z,\ 0 \le j \le M_z\text{ -1.}
\end{aligned} \tag{4}$$

$$\begin{aligned}
&PSF(i,j,k) = spline\left\{RZ\_PSF\left(r = 0 : (M_r - 1) \cdot \kappa_r, z(k)/\kappa_z\right), r = \sqrt{x^2(i) + y^2(j)}\right\} \\
&x(i) = (i - N_x/2 - 1) \cdot \delta_x, 0 \le i \le N_x\text{ -1};\ y(j) = (j - N_y/2 - 1) \cdot \delta_y, 0 \le j \le N_y\text{ -1;} \\
&z(k) = (k - N_z/2 - 1) \cdot \delta_z,\ 0 \le k \le N_z\text{ -1.}
\end{aligned} \tag{5}$$

i and ***j*** are the indices of a point in the 2-D RZ_PSF table along its radial and axial dimension, respectively.

$\boldsymbol{M_r}$, and $\boldsymbol{M_z}$ are the RZ_PSF's radial and axial dimensions.

$\boldsymbol{\kappa_r}$ and $\boldsymbol{\kappa_z}$ are the RZ_PSF's radial sampling rate and optical sectioning rate.

The other variables have the same meanings as they were described in Eqs. 1~3.

Initialize a 2-D RZ_PSF using

- FluoRZPSF (float NA, float λ, float n_{imm}, float κ_r, float κ_z)
- void init (float NA, float λ, float n_{imm}, float κ_r, float κ_z)
- void setImmersionMedium_Mismatch (float n_{imm*}, float n_{imm}, float WD)
- void setCoverSlip_Mismatch (float n_{g*}, float t_{g*}, float n_g, float t_g)

where the variables used above were explained in Eqs. 1~5.

Create a 2-D RZ_PSF using

- void create (bool *Check* = true)

where *Check* is to indicate whether to print (true) the generation progress in the terminal or not (false). The data of the generated RZ_PSF will be stored in a one-dimensional double array, which is a private member of the "FluoRZPSF" class, and this array can be obtained using the following member function in the "FluoRZPSF" class.

- double * RZpsfData( )

Save and read a 2-D RZ_PSF using

- void save(const char * *filehead*)
- void read(const char * *filehead*)

where *filehead* is the basename of the two files used to save the data of a RZ_PSF. One with the extension "rzh" is the header file and stores the information about the RZ_PSF in texts. The other with the extension "rzd" stores the actual data of the RZ_PSF in a binary strean of row-major order.

Create a 3-D PSF from a 2-D RZ-PSF using

- int get3Dpsf (int N_x, int N_y, int N_z, float δ_x, float δ_y, float δ_z, float * *psf*,
  const char * *file* = NULL, int *Points2Sum* = 5)
- int get3Dpsf (int N_x, int N_y, int N_z, float δ_x, float δ_y, float δ_z, double * psf,
  const char * *file* = NULL, int *Points2Sum* = 5)

where the variables except *file* and *Points2Sum* were explained in Eqs. 1~5. If not null, *file* is the full name of a text file exported by the program to save the information about the RZ_PSF and the generated 3-D PSF. *Points2Sum* is a compensation parameter and must be an odd integer. If *Points2Sum* is larger than 1, the PSF values at an extended plane are first calculated from the RZ_PSF for x = 0 ~ $N_x \cdot Points2Sum$ and y = 0 ~ $N_y \cdot Points2Sum$, and the PSF plane of the size "N_x x N_y" is then generated by summing the $Points2Sum^2$ pixels on the expanded plane. According to Eq. 5, the following relationships between the generated 3-D PSF and the RZ_PSF must be satisfied: $N_x < M_r \cdot \kappa_r / \delta_x / 2$, $N_y < M_r \cdot \kappa_r / \delta_y / 2$, $N_z < M_z \cdot \kappa_z / \delta_z$, $\delta_x > \kappa_r \cdot Points2Sum$, $\delta_y > \kappa_r \cdot Points2Sum$, and δ_z / κ_z must be an integer.

Get maximal dimensions of a 3-D PSF that can be generated from a RZ_PSF using:

- int maxDimensionX (float δ_x)
- int maxDimensionY (float δ_y)
- int maxSections (float δ_z)

Get minimal calibration of a 3-D PSF that can be generated from a RZ_PSF using:

- float minCalibration(int Points2Sum)

It is recommended to create a 3-D PSF from a RZ_PSF rather than directly using the "Fluo3DPSF" class when the actual sampling rates (δ_x and δ_y) used to generate a 3-D PSF are larger than the Nyquist sampling rate ($\lambda / NA / 4.0$) calculated from the objective numerical aperture (NA) and emission light wavelength (λ). In this case, it is better to first generate a RZ_PSF using a much smaller radial sampling rate (δ_r) than the Nyquist sampling rate to avoid the under-sampling problem, and then to calculate the 3-D PSF for the actual sampling rates from the RZ_PSF.

The numerical integration and spline routines are implemented by using the 61-point Gauss-Kronrod QAG adaptive integration and cubic spline interpolation in the GNU Scientific Library (GSL, http://www.gnu.org/software/gsl/).

CHAPTER 4 - DECONVOLUTION

Deconvolution routines in the *Deconv* library were programmed based on three constrained iterative deconvolution algorithms: the maximum likelihood-Lanweber iterative deconvolution (ML-LWID), the maximum likelihood-conjugate gradient iterative deconvolution (ML-CGID), and the maximum likelihood-expectation maximization iterative deconvolution (ML-EMID)[4]. Five classes in the *Deconv* library are associated with these three deconvolution routines: the “LWdeconvolver” class based on the ML-LWID algorithm, the “CGdeconvolver” class based on the ML-CGID algorithm, the “EMdeconvolver” class based on the ML-EMID algorithm, and the “LWCGdeconvolver” class were all derived from a base class named “deconvolver”. The “LWCGdeconvolver” class is a base class for both the “LWdeconvolver” and “CGdeconvolver” classes [Table 1].

Common control flags used in a deconvolution process:

- *IsApplyNorm* is to indicate whether to apply (true) normalization on the input image and the iteratively estimated object in a deconvolution process or not (false, default). This flag should be set up when a deconvolution plan is initialized.
- *IsTrackLike* is to indicate whether to track (true) the likelihood value iteratively in a deconvolution process or not (false, default), and this flag should be set up when a deconvolution plan is initialized. Be aware that the deconvolution process will be slowed if *IsTrackLike* is set to be true. If the likelihood values are tracked, they can be exported to a double array (*vtrack* is a pointer that points to the double array) after the deconvolution process using
    - unsigned int exportLikelihoodTrack (double * *vtrack*)

- *IsTrackMax* is to indicate whether to track (true) the max intensity value of an iteratively estimated object in a deconvolution process or not (false, default), and this flag should be set up when a deconvolution plan is initialized. Be aware that *IsTrackMax* will be automatically set to be true if the flag *IsApplyNorm* is true. If the max values are tracked, they can be exported to a double array (*vtrack* is a pointer that points to the double array) after the deconvolution process using
  - unsigned int exportObjectMaxTrack (double * vtrack)
- *CheckStatus* is to indicate whether to print (true) the progress of a deconvolution process in the terminal or not (false, default), and this flag should be set up when a deconvolution plan is initialized.

Criterion defined for stopping a deconvolution process:

- *Criterion* is a value set up before running a deconvolution process and is used to determine when to stop the deconvolution process. The deconvolution process will be stopped when the average value of the update between two successive estimated objects over 10 past iterations is less than *Criterion* that can be defined before running a deconvolution process using
  - void setCriterion (double *Criterion* = 1.0e-7)

  The update value between two successive estimated objects at each iteration can be exported to a double array (*vtrack* is a pointer that points to the double array) after a deconvolution process using
  - unsigned int exportUpdateTrack (double * vtrack)
- *MaxRunIteration* is the number of maximally allowed iterations in a deconvoluiton process. The deconvolution process will be stopped after *MaxRunIteration* iteraions even if *Criterion* is not achieved. *MaxRunIteration* (default = 1000) can be defined before running a deconvolution process using
  - void setMaxRunIteration (unsigned int *MaxRunIteration* = 1000)

Pre-conditioning for ML-LWID or ML-CGID:

A pre-conditioning process is required by ML-LWID or ML-CGID and the conditioning (or regularization) parameter applied in the pre-conditioning process can be either automatically determined from the input data using a modified golden search method or manually specified by a user to skip the searching process. It is recommended to apply the searching method for the users who have little prior knowledge about conditioning the input data even though the searching process may take a certain amount of time. The users will likely find that the conditioning parameters automatically determined from different input data sets devonvolved using a same PSF are very close. In this case, it is better to specify the conditioning parameter manually to skip the searching process. This is especially valuable for applying deconvolution on many 3-D data sets acquired at different time points.

- *ConditioningIteration* is the number of the iterations used in the modified golden search method to find the conditioning parameter. One would expect the larger *ConditioningIteration* will yield the more accurate conditioning parameter and the better pre-conditioning results, but more time will be consumed as well. In order to apply the automatically searching method to find the conditioning parameter, one has to set *ConditioningIteration* (default = 5) at least 1 using
  - void setConditioningIteration (unsigned int *ConditioningIteration* = 5)
- *ConditioningTolerance* is the tolerance value used in the modified golden searching method to determine whether the searched conditioning parameter is satisfied or not. Thus the smaller *ConditioningTolerance* will give the more accurate conditioning parameter and the better pre-conditioning results, but more time will be consumed as well. *ConditioningTolerance* (default = 0.1) can be given using:
  - void setConditioningTolerance (double *ConditioningTolerance* = 0.1)
- *ConditioningValue* represents the conditioning parameter as explained above. When the *ConditioningIteration* is set to be zero, *ConditioningValue* (default = 1.0e-8) can

be manually specified before running a deconvolution process using

- void setConditioningValue (double *ConditioningValue* = 1.0e-8)

NOTE: To use a manually specified conditioning parameter in ML-LWID or ML-CGID and skip the automatically searching process for determining the conditioning parameter, *ConditioningIteration* has to be set to ZERO.

Intensity regularization for ML-CGID:

- *IsApplyIR* is to indicate whether to apply (true, default) the intensity regularization on the estimated object iteratively in a ML-CGID process or not (false). This flag should be set up when a ML-CGID plan is initialized.

Newton acceleration for ML-EMID:

- *IsAccelerate* is to indicate whether to apply (true) acceleration using Newton method iteratively in a ML-EMID process or not (false, default). This flag should be set up when an ML-EMID plan is initialized.

Intensity regularization for ML-EMID:

- *EMIRiteration* is the number of the iterations every which the intensity regularization will be applied on the estimated object in a ML-EMID process. *EMIRiteration* cannot be negative and the intensity regularization will not be applied at all if it is set to be 0. *EMIRiteration* (default = 50) can be set before running the ML-EMID process using
  - void setEMIRiteration(unsigned int *EMIRiteration* = 50)

- *EMIRpenalty* is the intensity penalty parameter used for the intensity regularization in a ML-EMID process. *EMIRpenalty* can be initialized before running the ML-EMID process using
  - void setEMIRpenalty(double *EMIRpenalty*)

  If *EMIRpenalty* is not set or set to be smaller than 0 when *EMIRiteration* is initialized as a positive number, *EMIRpenalty* will be calculated as follows [5]:
  - the_max_value_in_the_input_PSF / the_max_intensity_in_the_input_image

Start deconvolution in two steps:

It is very straightforward to start a deconvolution process using the *Deconv* library: Step 1 - Initialize a deconvolution plan.

Initialize a ML-LWID plan

- LWdeconvolver() { init() ; }
- void init (bool IsApplyNorm = false, bool IsTrackLike = false,
  bool IsTrackMax = false, bool IsCheckStatus = true)
- void setCriterion (double *Criterion* = 1.0e-7)
- void setMaxRunIteration (unsigned int *MaxRunIteration* = 1000)
- void setConditioningIteration (unsigned int *ConditioningIteration* = 5)
- void setConditioningTolerance (double *ConditioningTolerance* = 0.1)

Initialize a ML-CGID plan

- CGdeconvolver() { init() ; }
- void init(bool IsApplyIR = true, bool IsApplyNorm = false,
  bool IsTrackLike = false, bool IsTrackMax = false, bool IsCheckStatus = true)
- void setCriterion (double *Criterion* = 1.0e-7)
- void setMaxRunIteration (unsigned int *MaxRunIteration* = 1000)
- void setConditioningIteration (unsigned int *ConditioningIteration* = 5)
- void setConditioningTolerance (double *ConditioningTolerance* = 0.1)

Initialize a ML-EMID plan

- EMdeconvolver() { init() ; }
- void init(bool IsAccelerate = false, bool IsApplyNorm = false,
  bool IsTrackLike = false, bool IsTrackMax = false, bool IsCheckStatus = true)
- void setCriterion (double *Criterion* = 1.0e-7)
- void setMaxRunIteration (unsigned int *MaxRunIteration* = 1000)
- void setEMIRiteration(unsigned int *EMIRiteration* = 50)

where “**LWdeconvolver()**”, “**CGdeconvolver()**”, and “**EMdeconvolver()**” are the default constructors. It is always recommended to call the member function “**init(…)**” to initialize an explicit deconvolution plan. The control flags used in “**init(…)**” were explained above and so were the other member functions. Refer to the earlier texts for using the member function “**setConditioningValue(…)**” to initialize a ML-LWID or ML-CGID plan or “**setEMIRpenalty(…)**” to initialize a ML-EMID plan.

Step 2 - Run a deconvolution process.

Run a ML-LWID process

-
  ```
  void run( int DimX, int DimY, int DimZ,
            double * image, double * psf, double * object, LWdws & ws,
            unsigned char * SpacialSupport = NULL,
            unsigned char * FrequencySupport = NULL )
  ```

-
  ```
  void run( int DimX, int DimY, int DimZ,
            float * image, float * psf, float * object, LWsws & ws,
            unsigned char * SpacialSupport = NULL,
            unsigned char * FrequencySupport = NULL )
  ```

Run a ML-CGID process

-
  ```
  void run( int DimX, int DimY, int DimZ,
            double * image, double * psf, double * object, CGdws & ws,
            unsigned char * SpacialSupport = NULL,
            unsigned char * FrequencySupport = NULL )
  ```

-
  ```
  void run( int DimX, int DimY, int DimZ,
            float * image, float * psf, float * object, CGsws & ws,
            unsigned char * SpacialSupport = NULL,
            unsigned char * FrequencySupport = NULL )
  ```

Run an ML-EMID process

-
  ```
  void run( int DimX, int DimY, int DimZ,
            double * image, double * psf, double * object, EMdws & ws,
            unsigned char * SpacialSupport = NULL,
            unsigned char * FrequencySupport = NULL )
  ```

-
  ```
  void run( int DimX, int DimY, int DimZ,
            float * image, float * psf, float * object, EMsws & ws,
            unsigned char * SpacialSupport = NULL,
            unsigned char * FrequencySupport = NULL )
  ```

The input arguments required by running each types of deconvolution are same except that each of them has a special work space. “LWdws” or “LWsws” is the work space for the double or float data given to a ML-LWID process, respectively. “CGdws” or “CGsws” is the work space for the double or float data given to a ML-LWID process, respectively. “EMdws” or “EMsws” is the work space for the double or float data given to a ML-EMID process, respectively. The memory required by a work space for deconvolving a 128x128x128 image data set is given in the following table:

Table 10: The required memory size for a deconvolution process

Work Space	LWdws	LWsws	CGdws	CGsws	EMdws	EMsws
Memory Cost (Mbytes)	~60	~30	~80	~40	~70	~35

image is a pointer that points to a one-dimensional array loading the 3-D image data set in row-major order. The image data set has the dimensions of *DimX* (the fastest varying dimension), *DimY* and *DimZ* (the slowest varying dimension). Each dimension must be a power of 2. Ideally, the input image data set should have a black background which intensity is 0 and this can easily achieved through applying a background subtraction on the input image data set.

psf is a pointer that points to a one-dimensional array loading the input 3-D PSF data in row-major order. This PSF must be stored in the manner for a deconvolution process (See Fig. 4) and have the same dimensions as the input image data set.

Be aware that both arrays that are pointed to by *image* and *psf*, will be rewritten in the deconvolution process.

object is a pointer that points to a one-dimensional array loading the data of the first estimated object in row-major order. The first estimated object can be identical to the input image data set and in this case, one can pass the same array to *image* and *object*. Or it could be initialized as a deconvolved object to continue a deconvolution process on a previously deconvovled image data set. The array that *object* points to, stores the data of the finally deconvolved object.

Spatial or Frequency support is not applied by default in a deconvolution process, but can be iteratively applied on the estimated object in the spatial domain or on the PSF in the Fourier domain by passing a one-dimensional unsigned char array to the pointer *SpacialSupport* or *FrequencySuppor*, respectively. The array must have the same dimensions as the input image data set and each value in the array must be either (unsigned char) 0 or (unsigned char) 1.

Export the profile of a deconvolution process

- Export an LW deconvolution profile using
  - void exportLW(const char * filename)
- Export a CG deconvolution profile using
  - void exportCG(const char * filename)
- Export an EM deconvolution profile using
  - void exportEM(const char * filename)

Examples of using deconvolution routines

- Load the input image data set and the 3-D PSF

```
CCube<float>    image, psf, object;
image.read ( “deconvolved_image.u8” );
psf.read ( “psf_for_deconvolved_image.f32” );
object = image;
```

- An example of starting a ML-LWID process

```
LWdeconvolver    lw;
LWsws    ws;
lw.init ( false, false, false, true );
lw.run ( image.length(), image.width(), image.height(), image, psf., object, ws );
lw.exportLW ( “LWdeconvolution.txt” );
```

- An example of starting a ML-CGID process

```
CGdeconvolver    cg;
CGsws    ws;
cg.init ( true, false, false, false, true );
cg.setMaxRunIteration ( 500 );
cg.setCriterion ( 1.0E-8 );
cg.setConditioningIteration ( 10 );
cg.run (image.length(), image.width(), image.height(), image, psf., object, ws );
cg.exportCG ( “CGdeconvolution.txt” );
```

- An example of starting a ML-EMID process

```
CGdeconvolver    em;
EMsws    ws;
em.init ( false, false, false, false, true );
em.setCriterion ( 1.0E-6 );
em.setEMIRiteration ( 30 );
em.run (image.length(), image.width(), image.height(), image, psf., object, ws );
em.exportEM ( “CGdeconvolution.txt” );
```

REFERENCES

1. Sarah Frishen Gibson and Frederick Lanni. 1989. Diffraction by a circular aperture as a model for three-dimensional optical microscopy. J. Opt. Soc. Am. A 6 (Septemper): 1357-1367.

2. Sarah Frishen Gibson and Frederick Lanni. 1991. Experimental test of an analytical model of aberration in an oil-immersion objective lens used in three-dimensional light microscopy. J. Opt. Soc. Am. A 8 (October): 1601-1613.

3. Sarah Frishen Gibson. 1990. Modeling the 3-D imaging properties of the fluorescence light microscope. Ph.D dissertation. Carnegie Mellon University, Pittsburgh, PA.

4. Yuansheng Sun, Paul Davis, Elizabeth A. Kosmacek, Fiorenza Ianzini and Michael A. Mackey. An Open-source Deconvolution Software Package For 3-D Quantitative Fluorescence Microscopy Imaging. J. Microscopy. Under Review.

5. José-Angel Conchello and James G. McNally. 1996. Fast regularization technique for expectation maximization algorithm for optical sectioning microscopy. Proceedings of SPIE 2655 (April): 199-208