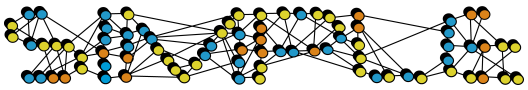
The  2.0 manual

"Keep It SIMPLE, Stupid!"

(Kelly Johnson, lead engineer at the Lockheed Skunk Works, coined the famous KISS principle stating that systems work best if they are kept simple rather than made complex, therefore simplicity should be a key goal in design and unnecessary complexity should be avoided.)

"Everything should be made as SIMPLE as possible, but no simpler"

(Albert Einstein)

"Complex theories do not work, SIMPLE algorithms do"

(Vladimir N. Vapnik, author of "The Nature of Statistical Learning Theory)

About SIMPLE

Single-particle IMage Processing Linux Engine (SIMPLE) does *ab initio* 3D reconstruction, heterogeneity analysis, and refinement. The SIMPLE back-end consists of an object-oriented numerical library with a single external dependency—the Fastest Fourier Transform in the West (FFTW) (Frigo and Johnson, 2005). The SIMPLE front-end consists of a few standalone, interoperable components developed according to the 'Unix toolkit philosophy'.

The SIMPLE codes prioritize computational efficiency, robustness, parameter-free, and empiric design over mathematical elegance. One of the SIMPLE development goals is to move cryo-EM image processing from the distributed computing environments to multicore desktop computers. SIMPLE is written in modern Fortran, which combines the high performance of compiled code with the ease of development via object oriented design paradigms. The modular design of the SIMPLE suite invites advanced users to develop new alignment and 3D reconstruction algorithms. Most of the SIMPLE algorithms are parallelized for shared-memory architectures using the OpenMP protocol. The most intensive parts of the calculations are parallelized for distributed systems according to a SMPD (Single Program Multiple Data) model.

What is new in release 2.0?

Compiler support for object-oriented Fortran development was limited when we released SIMPLE 1.0 in 2012. During that year, the open-source compiler gfortran http://gcc.gnu.org/wiki/GFortran provided full support for Fortran 2003. SIMPLE 1.0 was compiled using the Intel Fortran compiler, but all the code has been refactored to conform to the new 2003 standard and compiles using gfortran 4.7.1 or newer. It should be a walk in the park to compile SIMPLE 2.0 on a mac or linux machine. We also provide pre-compiled distributions (as described below). SIMPLE 2.0 was developed using test-driven development and comes with a set of automatic unit tests. We have removed our awkward *.fim file format and use Spider stacks and Spider volumes exclusively (http://www.wadsworth.org/spider_doc/spider/docs/formats.html). All major electron microscopy image-processing packages can convert to Spider files. We have replaced the common lines-based routines with a new probabilistic cryo-EM image alignment method (**simple_prime**, described below).

Installation of SIMPLE

```
>>> mv Downloads/simple2.tar.gz /myfolder

>>> gunzip simple2.tar.gz

>>> tar -xvf simple2.tar

>>> cd simple2

>>> ./simple_config.pl

Installation on local/biox3/delphi?
```

The precompiled programs in `/myfolder/simple2/bin` have been compiled on a 64-bit Linux AMD machine. If you want executable binaries for mac or other Linux/Unix architectures you'd have to compile the suite yourself (see **Compilation of SIMPLE**, below). Biox3 is the cluster that we use for distributed SIMPLE execution and delphi is the cluster used by the Grigorieff lab. Biox3 and delphi are both equipped with the portable batch system (PBS) for job scheduling. However, there are many different versions of the PBS and if you want a distributed SIMPLE version you will have to take a look at the `distr_simple.pl` and `distr_simple_prime.pl` scripts in the `apps` folder (see **Distributed Execution** below for details). Biox3 and delphi should be considered as templates to modify according to your cluster. We are happy to help anyone that is not comfortable with simple perl scripting, just use the contact form on the simple website to get in touch with us.

```
>>> local
```

The produced files `source_tcsh` and `source_bash` containing the lines you need to add to your .cshrc or .bashrc file, respectively, in order to have prompt access to the SIMPLE programs. If you are compiling SIMPLE from source, the compiler options used by the compile script `simple_compile.pl` are valid for the gfortran compiler, assumed to be executed from the prompt with the command `gfortran`. No other compilers are currently supported.

Compilation of SIMPLE

We provide pre-compiled binaries for linux but not for mac, so you will have to compile the suite yourself.

The `simple_compile.pl` script takes care of compiling individual simple programs as well as the entire suite. It is assumed that gfortran 4.7.1 or newer is installed on your system http://gcc.gnu.org/wiki/GFortran. We have only tested compilation with up to gfortran version 4.8 and we can therefore not ensure that compilation will work with newer versions. If you get compiler bugs with newer versions of gfortran, please let us know. The environment variable `FFTWLIB` needs to point to the location where the FFTW (Fastest Fourier Transform in the West, http://fftw.org) library is installed. The typical mac bash directives look like

```
export FFTWLIB=/sw/lib
```

or

```
export FFTWLIB=/sw2/lib
```

If you use fink as a package manager on the mac, you install FFTW with the command

```
sudo fink install fftw3 fftw3-shlibs
```

Note that SIMPLE uses the single-precision FFTW library. We also provide the FFTW library for Linux together with SIMPLE, located in `<whatever path>/simple/fftw-3.3.2`. For compilation of the suite on a linux machine, cd to the SIMPLE directory and execute

```
>>> mkdir fftw

>>> cd fftw-3.3.2

>>> make clean

>>> ./configure prefix=<whatever path>/simple/fftw —-
enable-float —enable-threads
```

```
>>> make

>>> make install
```

Once you have FFTW installed and the environment variable set you can compile the entire suite with the command

```
>>> ./simple_compile.pl prg=all_fast mac=no
```

on a Linux machine, or the command

```
>>> ./simple_compile.pl prg=all_fast
```

on a mac. This will compile a library of all core functionality and link this to the individual program units. The executable binaries end up in the folder `<whatever path>/simple2/bin` . We have experienced problems creating the library on some machines. If you experience problems, use the slower option of compiling without making a library

```
>>> ./simple_compile.pl prg=all
```

If you want to compile only a single program, then give that program as input

```
>>> ./simple_compile.pl prg=simple_prime
```

If you want to clean up all intermediate files produced during compilation, execute

```
>>> ./simple_compile.pl prg=clean
```

You can turn off the compiler optimizations and turn on the debugging info by

```
>>> ./simple_compile.pl prg=simple_prime optimize=no
```

Parallelization of SIMPLE

There are two modes of parallelization used in SIMPLE. All programs are parallelized with OpenMP, using a shared-memory model (not for cluster environments). **Simple_prime** is additionally parallelized using a single program multiple data (SMPD) model (suitable for both cluster and workstation environments). The OpenMP model is applied by setting command line variable `nthr=X`, where `X` is the number of threads. OpenMP is used by default but you can turn it off in compilation by

```
>>> ./simple_compile.pl prg=simple_prime para=no
```

The SMPD model is executed by perl scripts `distr_simple.pl` and `distr_simple_prime.pl`. See **Distributed SIMPLE execution,** below for

details on how to use SIMPLE in distributed computing environments. There are significant performance gains to be made by using the SPMD model for **simple_prime** execution in a workstation environment.

Programs

SIMPLE is not a standalone suite for single-particle reconstruction, but complements available developments. It is assumed that the windowed single-particle images represent 2D projections, and therefore, that the projection slice theorem applies (Bracewell, 1956). This requires correction of the micrographs due to the contrast transfer function (CTF) of the electron microscope and particle windowing using other software. "CTF-corrected images" refers to images corrected using the simple heuristic of binary phase flipping, adopted in numerous image-processing packages (Ludtke et al., 1999; Tang et al., 2007; van Heel et al., 1996). CTF-correction by phase flipping only corrects the resolution-dependent CTF phase inversions, disregarding the damping of Fourier amplitudes with increasing resolution. Phase-flipped images are noisier than images corrected with more accurate approaches, such as the reciprocal space adaptive Wiener filter (Penczek, 2010). So far, we have been primarily concerned with the problem of reconstructing accurate low-resolution maps *ab initio* from very noisy images. More sophisticated approaches for dealing with the CTF will be implemented in future SIMPLE releases. In EMAN, for example, the envelope component of the CTF, describing the falloff of the signal with resolution, is parameterized and then used in the class averaging procedure in the refinement. Some implementations only phase-corrects the micrographs, ignoring the envelope function initially. Instead, the final refined maps are B-factor sharpened. SIMPLE has so far been used in combination with the latter approach, assuming the same resolution falloff for all micrographs. The windowed projections should be roughly centered in the box. Many program suites are available for dealing with these and other matters (Frank et al., 1996; Hohn et al., 2007; Ludtke et al., 1999; Sorzano et al., 2004). The SIMPLE workflow is divided into four phases:

(1) CTF correction and particle windowing.
(2) *Ab initio* reconstruction.
(3) Heterogeneity analysis.
(4) Refinement.

For detailed instructions, see http://simple.stanford.edu/workflow.html.

The SIMPLE program instructions are listed in the order the programs are normally executed during the reconstruction process. Each paragraph begins with an overview description of the algorithms used by the program, followed by a paragraph describing how to execute the program, and ending with a paragraph of comments describing alternative execution routes. We have tried to keep parameter tweaking to a minimum by design but there are some strategies for tuning that may be useful (as described below). All SIMPLE programs are executed using the command line. Input and output consists of Spider image stacks, Spider volumes,

Spider document files, .txt text files. The output is written directly to the working directory, giving you freedom to organize a directory structure that suits your needs. Remember that another round of execution in the same directory overwrites your old files. The notation <what> denotes input input parameter "what". <this|that> denotes alternative input parameters "this" and "that", [<noway>] denotes the optional parameter "noway", and {val} denotes the suggested or default value "val". Command line arguments are passed as *key*=<value>, where "*key*" is the key in the hash used to store the parameters. For example, the following instruction:

```
>>> program param=<what{20-40}> either=<this|that>
[itisanumber=<noway{0}>]
```

can be executed like this:

```
>>> program param=30 either=this
```

or like this:

```
>>> program param=35 either=that itisanumber=500
```

The command line interface is similar to the interface used by the *proc2d* and *proc3d* programs in EMAN, so you should feel right at home if you are familiar with EMAN. You are encouraged to contact the author of this manual and bark at him if the instructions contain errors (e-mail: hael@stanford.edu). Each program prints a short version of the description presented here when executed without command line arguments.

Command line dictionary

acf	autocorrelation function
amsklp	automask low-pass limit (in Å)
bin	binarize (yes|no)
box	image size in pixels (image assumed to be *box*box* array)
boxpd	padded box size (default is *2*box*)
center	center(yes|no)
clsdoc	Spider format clustering document
cure	cure or not (yes|no), for curing NaN:s and normalize
cwd	current working directory
debug	debug mode (yes|no)
deterministic	deterministic search (yes|no)
discrete	discrete option (yes|no)
doalign	do alignment (yes|no)
dopca	do PCA (yes|no)
doprint	do print (yes|no)
e1	first Euler angle
e2	second Euler angle

e3	third Euler angle
edge	edge size for softening molecular envelope (in pixels)
eo	even-odd test (yes|no)
even	even (yes|no)
fbody	file body (<body>.ext)
filter	filter (yes|no)
frac	fraction [0,1]
fromp	from particle index
froms	from state index
gw	Gaussian half width (in pixels)
hp	high-pass limit (in Å)
kmeans	do kmeans (yes|no)
local	local refinement (yes|no)
lp	low-pass limit (in Å)
matched	use matched Wiener filter (yes|no)
maxits	maximum number of iterations
minp	minimum number of particle images (in a cluster)
moldiam	molecular diameter (in Å)
msk	circular or spherical mask radius (in pixels)
mskfile	external mask file (*.spi)
mul	multiplication (scaling) factor (for shifts)
mw	molecular weight (in kD)
navgs	number of averages
ncls	number of clusters
ndiscrete	number of discrete (orientations)
ndocs	number of documents
newbox	new box size
noise	noise (yes|no)
noris	number of orientations
norm	normalize (yes|no)
npart	number of partitions
npeaks	number of peaks (=number of nonzero orientation weights)
nptcls	number of particle images
nran	number of images in random sample
nspace	number of projection directions in search space
nstates	number of discrete state groups
nthr	number of openMP threads
nvars	number of eigenvectors or hidden variables
nvox	number of voxels
oritab	SIMPLE orientations file (*.txt)
outfile	SIMPLE output file (*.txt)
outstk	output Spider image stack (*.spi suffix required)
outvol	output Spider volume (*.spi suffix required)
part	partition number
pgrp	point-group symmetry (c1, c2, c3, …, or d1, d2, d3, …)
phrand	phase randomize (yes|no)

phranlp	phase randomization low-pass limit (in Å)
ppca	probabilistic pca (yes|no)
ring1	inner ring in polar image representation
ring2	outer ring in polar image representation
rnd	random (yes|no)
roalgn	rotational alignment (yes|no)
shalgn	shift alignment (yes|no)
smpd	sampling distance (in Å)
snr	signal-to-noise ratio
space	space (real|fourier)
startit	starting iteration (if different than 1)
state	discrete state group
stk	spider image stack name (*.spi suffix required)
stk2	second spider image stack name (*.spi suffix required)
stk3	third spider image stack name (*.spi suffix required)
tau	temperature parameter
time_per_image	per particle time indicator
top	to particle index
tos	to state index
tres	threshold
trs	origin shift search range parameter [-*trs*,*trs*]
trsstep	translation step size
utst	unit test number
var	variance
vol1	spider volume 1 name (*.spi suffix required)
wfun	weighting function or interpolation kernel
winsz	window size for interpolation
zero	zero (yes|no)

SIMPLE Utilities & Applications

We divide the SIMPLE programs into two groups: utilities and applications. Utility software focuses on *how* the system infrastructure operates, providing functionality required for running the applications. Utility software is often rather technical and targeted at users with advanced knowledge about the system. This contrasts application software, which allows users to execute the higher-level functionalities that motivated the design of the system. To make most effective use of a system, it is often necessary to master both the utility and the application software.

Utility Program: simple_make_cavgs

Simple_make_cavgs is a program for making class averages given a stack of images and their alignment. Some people like to make class averages and compare them with projections of their volume in corresponding orientations. This program makes the class averages and the volume can be projected in the outputted orientations *cavgoris.txt* using **simple_projvol**, described below.

Usage:

```
>>> simple_make_cavgs stk=stack.spi box=<image size(in
pixels)> smpd=<sampling distance(in A)> oritab=<alignment
doc> nspace=<nr of class averages> [pgrp=<cn|dn>]
```

Utility Program: simple_mkrndoris

Simple_mkrndoris is a program for making random orientations in various flavors.

Usage:

```
>>> simple_mkrndoris nptcls=<number of oris>
outfile=<output alignment doc> [trs=<origin shift(in
pixels){0}>] [nstates=<nr of states{1}>] [pgrp=<cn|dn>]
[even=<yes|no{no}>] [zero=<yes|no{no}>] [ndiscrete=<nr of
discrete orientations>]
```

Comments:

nptcls is the number of desired orientations, *outfile,* is the output file, *trs* is the one-sided size of the origin shift interval (2 means from -2 to +2), *nstates* is the number of states, *pgrp* is the point-group, if set to 'yes' *even* will given you even projection directions and random in-plane parameters, with *ndiscrete* set the program will sample projection directions randomly from a set of *ndiscrete* even ones and give you random in-plane parameters.

Utility Program: simple_npeaks

Simple_npeaks is a program for calculating the size of the feasible subset of orientations, used in **simple_prime**, described below. The number of feasible orientations per particle image (or the number of nonzero orientation weights) is determined based on the selected resolution limit, the molecular radius, and the resolution of the discrete 3D orientation search space. This program checks how many copies of the same image are added (with weights) to the reconstruction for a given set of input parameters.

Usage:

```
>>> simple_npeaks box=<image size(in pixels)>
smpd=<sampling distance(in A)> [nspace=<nr of projection
directions>] [lp=<low-pass limit(A){20}>] [moldiam=
<molecular diameter(A)>] [pgrp=<cn|dn{c1}>]
```

Utility Program: simple_nspace

Simple_nspace is a program for checking how many projection directions (reference images) that you need to get to a certain resolution with projection matching, given the molecular diameter as input. The number is slightly overestimated, because the in-plane rotations are sampled finer than the projection directions, so the finer in-plane sampling can compensate for limited angular resolution of the distribution of projection directions.

Usage:

```
>>> simple_nspace moldiam=<molecular diameter (in A)>
```

Utility Program: simple_projvol

Simple_projvol is a program for projecting a 3D volume. Output is a spider stack of projection images.

Usage:

```
>>> simple_projvol vol1=invol.spi box=<image size(in
pixels)> smpd=<sampling distance(in A)> [nspace=<nr of
projs>] outstk=<ouput spider stack> [oritab=<SIMPLE
alignment doc>]
** less commonly used** [pgrp=<point group symmetry{c1}>]
[gw=<window function halfwidth{0.5}>] [winsz=<hard window
halfwidth{1}>] [wfun=<window function,
gau|gau2|sinc|lin|{gau}>] [rnd=<yes|no{no}>]
[space=<real|fourier{fourier}>] [msk=<mask radius (in
pixels)>]
```

Comments:

The *oritab* parameter allows you to input the orientations that you wish to have your volume projected in, *gw* controls the half-width of the Gaussian (or other kernel) used for interpolation, *winsz,* is the hard window half-width, *wfun* controls the window function used (two Gaussian variants: gau, gau2, sinc function, or linear interpolation), *rnd* set to 'yes' projects the volume in *nspace* random orientations, *space* controls the space used for interpolation (*real*: real-space projection or *fourier*: which extracts a central section from the Fourier volume and back transforms the 2D FT, which is MUCH faster), and finally, *msk* is used to control the radius within which the volume is projected if the *real* option is used.

Utility Program: simple_recvol

Simple_recvol is a program for reconstructing volumes from spider stacks (*.spi), given input orientations and state assignments (obtained by program **simple_prime**). The algorithm is based on direct Fourier inversion with a truncated Gaussian interpolation kernel. This window function reduces the real-space ripple

artifacts associated with direct moving window sinc interpolation. The feature sought when implementing this algorithm was to enable quick, reliable reconstruction from aligned individual particle images.

Usage:

```
>>> simple_recvol stk=projs.spi box=<box size(in pixels)>
smpd=<sampling distance(in A)> oritab=algndoc.txt
[frac=<fraction of ptcls to include{1.}>] [eo=<yes|no>]
[nthr=<nr of openMP threads{1}>]
**less commonly used** [mul=<shift multiplication
factor{1}>] [pgrp=<cn|dn{c1}>] [part=<partition number>]
[fromp=<start ptcl{1}>] [top=<stop ptcl{nptcls}>]
[state=<state to reconstruct{all}>]
```

Comments:

The optional parameter *eo* is used for generating even-odd reconstructions subjected to FSC analysis.

Utility Program: simple_volassemble

Simple_volassemble is a program that assembles a volume when the reconstruction program (**simple_recvol**) has been executed in distributed mode.

Usage:

```
>>> simple_volassemble npart=<number of partitions to
assemble> nstates=<nr of states> box=<image size(in
pixels)> smpd=<sampling distance(in A)> [nthr=<nr of openMP
threads{1}>] [eo=<yes|no>] [lp=<low-pass limit{20}>]
[matched=<yes|no{no}>]
```

Comments:

eo=yes is used when even-odd pairs has been reconstructed with distributed **simple_recvol**, with *lp* set the assembled volume is low-pass filtered to the given resolution (used by **distr_simple_prime.pl**, described below)

Utility Program: simple_rndrec

Simple_rndrec is a program for generating random initial models. We use this program to provide starting models when executing **simple_prime** in distributed parallelization mode.

Usage:

```
>>> simple_rndrec [stk=stack.spi] box=<image size(in
pixels)> smpd=<sampling distance(in A)> [nspace=<nr of
reference sections{1000}>] [lp=<low-pass limit{20}>]
```

```
[nstates=<nr of states{1}>] [nran=<size of random sample>]
[nthr=<nr of OpenMP threads{1}>]
**less commonly used** [pgrp=<cn|dn>] [npeaks=<number of
nonzero orientation weights{1}>] [noise=<yes|no>]
```

Comments:

Setting the *nstates* parameter > 1 partitions the data randomly into discrete state groups and calculates one blob per group. If the data set is large (>5000 images), generating a random model can be quite slow. To speedup, set *nran* to some smaller number, resulting in *nran* images selected randomly for reconstruction. If you want to override the automatically determined feasible region size (the number of nonzero orientation weights per image) you can set *npeaks* to any number that you wish. If you set *noise* equal to yes, the program will produce a pure noise volume (which works equally well for initialization of **simple_prime**).

Utility Program: simple_automask

Simple_automask is a program for solvent flattening of a spider volume. The algorithm for background removal is based on low-pass filtering and binarization. First, the volume is low-pass filtered to *amsklp*. A binary volume is then generated by assigning foreground pixels (=1) based on the volume calculated from the molecular weight, assuming a protein density of 1.43 g/mL. A real-space low-pass filter softens the edge of the resulting binary volume before multiplying it with the 'raw' input volume to generate the flattened map.

Usage:

```
>>> simple_automask vol1=invol1.spi [vol2=invol2.spi etc.]
box=<box size(in pixels)> smpd=<sampling distance(in Å)>
mw=<molecular weight(in kD)> [amsklp=<low-pass limit(in
Å){40}>]
**less commonly used** [edge=<edge size for softening of
the molecular envelope(in pixels)>]
```

Utility Program: simple_stackops

Simple_stackops is a program that provides standard single-particle image processing routines that are applied to spider stacks.

Usage:

```
>>> simple_stackops [stk=stack.spi] [stk2=stack2.spi]
box=<image size(in pixels)> smpd=<sampling distance(in A)>
outstk=<output_imgs.spi> [nthr=<nr of openMP threads{1}>]
[oritab=<SIMPLE alignment doc>] [hp=<high-pass limit(in
A)>] [lp=<low-pass limit(in A)>] [shalgn=<yes|no{no}>]
[mul=<shift multiplication factor{1}>] [trs=<origin
shift(in pixels)(Shaw et al.)>] [roalgn=<yes|no{no}>]
[ring1=<inner mask radius(in pixels){1}>] [ring2=<outer
```

mask radius(in pixels){*box*/2}>] [*state*=<state to extract>] [*frac*=<fraction of ptcls to extract{1}>] [*snr*=<signal2noise ratio>] [*msk*=<mask radius(in pixels){box/2}>] [*bin*=<binarize{no}>] [*acf*=<yes|no{no}>] [*phrand*=<yes|no{no}>] [*fromp*=<start ptcl>] [*top*=<stop ptcl>] [*nran*=<number of random images to select>] [*newbox*=<scaled box>] [*norm*=<yes|no{no}>] [*tres*=<binary treshold{0}>] [*rnd*=<yes|no{no}>]

Comments:

You can do many things with **simple_stackops**. Filtering is controlled by the *hp* and *lp* arguments. Two kinds of alignments are available: shift alignment and rotational alignment. If you input an alignment document (*oritab*) *shalgn*=yes will produce a shift aligned stack based on the inputted orientations, whereas if you do not input an alignment document the alignment will be done in a reference-free manner (remember to set *trs* to some nonzero value). *roalgn*=yes behaves in the same way, but note that if an alignment document is provided *roalgn*=yes will both shift and rotate the stack. The *ring* parameters control the inner and outer mask radius for the polar image representation used for rotational alignment. If you want to extract a particular state, give an alignment document (*oritab*) and set *state* to the state that you want to extract. If you want to select the fraction of best particles (according to the goal function), input an alignment doc (*oritab*) and set *frac* [0,1]. You can combine the *state* and *frac* options. If you want to apply noise to images, give the desired signal-to-noise ratio via *snr.* If you want to mask your images with a spherical mask with a soft falloff, set *msk* to the radius in pixels. If you want to binarize your images, set *bin*=yes. If you want to calculate the autocorrelation function of your images set *acf*=yes. If you want to randomize the phases of the Fourier transforms of your images, set *phrand*=yes and *lp* to the desired low-pass limit. If you want to extract a contiguous subset of particle images from the stack, set *fromp* and *top.* If you want to fish out a number of particle images from your stack at random, set *nran* to some nonzero number < *nptcls*. If you want to resize you images, set the desired box *newbox* < *box.* If you want to normalize your images, set *norm*=yes. If you want to use the threshold option for binarization (instead of k-means clustering), set *tres* [0,1]. If you want to generate 100 random images, set *rnd*=yes and *nptcls*=100.

Utility Program: simple_volops

Simple_volops is a program that provides standard single-particle image processing routines that are applied to spider volumes.

Usage:

>>> **simple_volops** *vol1*=invol.spi [*vol2*=invol2.spi] *box*=<image size(in pixels)> *smpd*=<sampling distance(in A)> [*outvol*=<outvol.spi>] [*winsz*=<convolutional window size>] [*stepsz*=<convolutional stepsize{1}>] [*nthr*=<nr of openMP

threads{1}>] [*phrand*=<yes|no{no}>] [*msk*=<mask radius(in pixels)>] [*lp*=<low-pass limit{20}>] [*snr*=<signal-to-noise ratio>] [*oritab*=<previous rounds alignment doc>] [*center*=<yes|no{no}>]

Comments:

center=yes centers the inputted volume (*vol1*) (remember to give an output volume filename *outvol*) by lowpass filtering *vol1* to *lp,* using k-means to separate foreground pixels from background pixels, determining the center of mass, and shifting the inputted volume accordingly. *phrand*=yes randomizes the phases of the FT of *vol1* from resolution *lp.* If two volumes are inputted, the *FSC* is calculated and the resolution at both *FSC*=0.5 and *FSC*=0.143 is printed to STDOUT. If you give a signal-to-noise ratio via *snr,* noise is applied to the input volume accordingly. If defined, *winsz* will result in denoising of *vol1* by convolutional principal component analysis. *lp* is for low-pass filtering. The *msk* parameter (radius of the mask) is used by both the centering routine and the *snr* option.

Utility Program: SIMPLE_UNIT_TESTS

Simple_unit_tests is a program that executes all the module unit tests in the library. Since we are using test-driven development we have unit tests associated with many of the classes. This program tests:

```
simple_defs
enfors_ftiter
simple_ori
enfors_image
simple_oris
enfors_spider
simple_polarimg
simple_roalgn
simple_shalgn
simple_hac
simple_kmeans
simple_sa_opt
simple_align_pair
simple_convol_imgs
simple_args
```

Usage:

```
>>> simple_unit_tests
```

Application Program: simple_cluster

Simple_cluster is a program for image clustering based on reference-free in-plane alignment (Penczek et al., 1992) and probabilistic principal component analysis (PCA) for generation of feature vectors (Tipping and Bishop, 1999). Agglomerative

hierarchical clustering (HAC) is used for grouping of feature vectors (Murtagh, 1983). Refinement of the clustering solution is done with the center-based k-means clustering. **Simple_cluster** in-plane aligns the input image stack. Bicubic interpolation is used for shifting and rotating the stack before extraction of the pixels within the circular mask defined by mask radius *msk*. Next, the probabilistic PCA method generates feature vectors from the vectors of extracted pixels. The minimum cluster population (*minp*) prevents clusters below population *minp* to be represented by an output average.

Usage:

```
>>> simple_cluster stk=stack.spi box=<box size(in pixels)>
smpd=<sampling distance(in Å)> [msk=<mask radius(in
pixels){box/2}>] [ncls=<nr clusters{500}>] [minp=<minimum
nr ptcls cluster{10}>] [nran=<size of random
sample{nptcls}>] [oritab=<SIMPLE alignment doc>] [nthr=<nr
openMP threads{1}>]
**less commonly used** [nvars=<nr eigenvectors{30/60}>]
[clsdoc=<Spider clustering doc>] [kmeans=<yes|no{yes}>]
[dopca=<yes|no] [doalign=<yes|no{no}>] [rnd=<yes|no{no}>]
[utst=<unit test nr(1-4){0}>]
```

Comments:

The setup allows for quick testing of the number of clusters. One pass produces the file 'pdfile.bin' containing the matrix of all pair-wise feature vector distances. Using the optional parameter *dopca* set to 'no' in a second round of execution from the same directory will make the program read the previously generated distances and re-do the clustering using whatever settings inputted for parameters *ncls* & *minp*. The optional parameter *oritab* is used to provide in-plane parameters for the clustering (provided by program **simple_prime**, described below). This option is used for generating class averages that are going to be subjected to heterogeneity. The default setting uses 30 eigenvectors if you are not inputting in-plane parameters via optional parameter *oritab,* and 60 eigenvectors if you do input in-plane parameters. Note that the distance matrix is kept in RAM, so for large data sets you need LOTS of internal memory. This quirk can be addressed by using a random sample of the data for initial clustering by HAC. This is done by setting *nran* to some number < *nptcls*. In this setting, the HAC centers generated from the random sample are used to extend the clustering to the entire data set with k-means. This overcomes the well-known initialization problem of k-means and enables clustering of many hundreds of thousands of particle images. SIMPLE has been used to cluster 300,000 images with a box size of 100 using a random subset of 60,000 images on a machine with 96 GB RAM.

Application Program: simple_prime

Simple_prime is an *ab inito* reconstruction/low-resolution refinement program based on probabilistic projection matching. **PRIME** stands for **PR**obabilistic **I**nitial

3D **M**odel Generation for Single-Particle Cryo-**E**lectron Microscopy. There should be no impediment to using **PRIME** also for high-resolution refinement and reconstruction of discrete state groups. We are currently playing around a lot with the **PRIME** algorithm to understand it better and to find the right combination of goal function and weighting scheme for the different problems that we want to apply it to. In the end of this year (2013) we plan to release a new SIMPLE version that will implement heterogeneity analysis with **PRIME**. The code can, as it stands, be used for *ab initio* 3D reconstruction by initialization with a random starting model (generated automatically if no volume is inputted or by program **simple_rndrec**, described above). There are many ways of using (and probably also abusing) **simple_prime**. We will walk you through a few examples (see **Using simple_prime,** below). For data sets of realistic size you would not execute **simple_prime** in the shared-memory parallelization setting described here, but use **distr_simple_prime.pl**, described below.

Usage:

```
>>> simple_prime stk=stack.spi [vol1=invol.spi]
[vol2=<refvol_2.spi> etc.] box=<image size(in pixels)>
smpd=<sampling distance(in A)> [ring2=<outer mask radius(in
pixels){box/2}>] [trs=<origin shift(in pixels){0}>]
[trsstep=<origin shift stepsize{1}>] [lp=<low-pass
limit{20}>] [dynlp=<yes|no{no}>] [nstates=nstates to
reconstruct>] [frac=<fraction of ptcls to include{1}>]
[mw=<molecular weight (in kD)>] [oritab=<previous rounds
alignment doc>] [nthr=<nr of OpenMP threads{1}>]
 **less commonly used** [nspace=<nr of reference
sections{1000}>] [amsklp=<automask low-pass limit(in
A){40}>] [maxits=<nr of iterations{50}>] [pgrp=<cn|dn{c1}>]
[ring1=<inner mask radius(in pixels){1}>] [edge=<edge size
for softening molecular envelope(in pixels){3}>] [tau=<for
sharpening the weight distribution(0.05-1){1}>]
[find=<Fourier index>] [fstep=<Fourier step size>]
[local=<yes|no{no}>] [deterministic=<yes|no{no}>]
[noise=<yes|no>] [time_per_image=<{60}>] [lphigh=<stay at
this low-pass limit(in A)>]
```

Comments:

You could in principle get away with a command like

```
>>> simple_prime stk=stack.spi dynlp=yes box=100 smpd=2.33
```

Note that the first round of prime should always be run with dynamic resolution stepping (*dynlp*=yes) whereas later runs can be executed with (*lp*=x) for low-pass limit x Å or without any lp-argument, in which case a Wiener filter will be used (the Wiener filter is presently buggy, we are working on it). Since the search is probabilistic, we figured that an elegant convergence criterion could be formulated based on the variance of the distribution of orientations assigned to each image.

This works well for asymmetrical reconstructions, but for symmetrical reconstructions the variance increases towards the end of the run when the shape most consistent with the point group is being established. Note that we do not assume any point-group symmetry in the initial runs. However, the **simple_symsrch** program (described below) can be used to align the reconstruction to its symmetry axis so that the search can be restricted to the asymmetric unit in refinement (see **Reconstruction of symmetrical structures**, below). We have not characterized the algorithm for super high-resolution reconstruction; our primary focus has been *ab initio* reconstruction. The code is being benchmarked on data sets known to refine to high resolution. Less commonly used and less obvious input parameters are *nspace*, which controls the number of reference projections, *amsklp,* which controls the low-pass limit used in the automask routine, *maxits,* which controls the maximum number of iterations executed, *pgrp,* which controls the point-group symmetry, assuming that the starting volume is aligned to its principal symmetry axis, *ring1,* which controls the inner mask radius in the polar image representation, *edge,* which controls the size of the softening edge in the automask routine. *local*=yes indicates that the search should be local around the previous best orientation, intended for higher-resolution refinement using a larger number of reference projections, say 10,000 (experimental), *deterministic,* which removes the stochastic component of the search and evaluates the entire neighborhood in every round (experimental), *noise,* which produces a random noise starting volume instead of a random blob if no input volume is given.

Application Program: simple_symsrch

Simple_symsrch is a program for searching for the principal symmetry axis of a volume reconstructed without assuming any point-group symmetry. Our philosophy is to start off without assuming any symmetry and analyze the reconstructed volume to identify the correct point-group symmetry. **Simple_symsrch** can be used for this purpose. The program takes as input the asymmetrical reconstruction (obtained with **simple_prime**, described above), the alignment document for all the particle images that have gone into the reconstruction, and the desired point-group symmetry. It then projects the reconstruction in 20 quasi-even directions, applies common lines-based optimization to identify the principal symmetry axis, and applies the rotational transformation to the inputted orientations, and produces a new alignment document. Input this document to **simple_recvol** together with the images and the point-group symmetry to generate a symmetrized map. If you are unsure about the point-group, you can of course test many different point-groups and formulate a statistical test that compares the asymmetric map with the symmetrized maps for different point-groups.

Usage:

```
>>> simple_symsrch vol1=invol.spi box=<image size(in
pixels)> smpd=<sampling distance(in A)> oritab=<previous
rounds alignment doc> pgrp=<cn|dn{c1}> outfile=<output
```

```
alignment doc> [lp=<low-pass limit{20}>] [nthr=<nr of
OpenMP threads{1}>]
**less commonly used** [state=<state 2 rotate>]
```

Distributed SIMPLE execution

Application Program: distr_simple_prime.pl

Distr_simple_prime.pl distributes **simple_prime** on multicore workstations or clusters. The program requires two environment variables to be set: SIMPLEBIN and SIMPLESYS. You should have set these variables when installing simple (see Installation of SIMPLE, above). We use SIMPLESYS=BIOX3 for our cluster executions. BIOX3 is a cluster that uses the portable batch system (PBS) for job scheduling. If you are using PBS, there is a chance that the distributed execution will work painlessly, but there is no insurance (because there are many different versions of the PBS). The code that takes care of the job distribution is, however, extremely simple and you need only to modify the `exec_prime_para` subroutine in the `<whatever folder>/simple/apps/distr_simple_prime.pl` script. **Distr_simple_prime.pl** accepts the same arguments as **simple_prime** with the exception that *nthr* (the number of OpenMP threads) have been replaced with *npart* (number of partitions). Checkout the workflow on the SIMPLE website for links to tutorials that describe the exact execution routes. The distributed prime version is executed as `nohup distr_simple_prime.pl <command line args> > PRIME_OUT &` and progress is monitored by `tail -50f PRIME_OUT`

Application Program: distr_simple.pl

Distr_simple.pl distributes any SIMPLE program that accepts the command line arguments *fromp, top,* and *part.* We are using it exclusively for volume reconstruction when the data sets are large. A strategy that we have found useful is to do deterministic reconstruction from the most likely converged orientations obtained by **simple_prime.** This is done, for example, by

```
>>> distr_simple.pl prg=simple_recvol stk=ptcls.spi box=100
smpd=2.33 oritab=merged.txt npart=100
```

which splits the reconstruction job into 100 partitions, generating 200 output files containing Fourier and interpolation kernel matrices. The *merged.txt* file is the merged orientation file from distributed **simple_prime** exection. The reconstruction is assembled by

```
>>> simple_volassemble npart=100 nstates=1 box=100
smpd=2.33 nthr=4
```

Heterogeneity analysis

Is underway.

High-resolution refinement

Is underway.

References

Acciari, V.A., Aliu, E., Arlen, T., Bautista, M., Beilicke, M., Benbow, W., Bradbury, S.M., Buckley, J.H., Bugaev, V., Butt, Y.*, et al.* (2009). Radio imaging of the very-high-energy gamma-ray emission region in the central engine of a radio galaxy. Science *325*, 444-448.

Bracewell, R.N. (1956). Strip integration in radio astronomy. Aust J Phys *9*, 198-217.

Frank, J., Radermacher, M., Penczek, P., Zhu, J., Li, Y.H., Ladjadj, M., and Leith, A. (1996). SPIDER and WEB: Processing and visualization of images in 3D electron microscopy and related fields. Journal of Structural Biology *116*, 190-199.

Frigo, M., and Johnson, S.G. (2005). The design and implementation of FFTW3. Proceedings of the IEEE *93*, 216-231.

Hohn, M., Tang, G., Goodyear, G., Baldwin, P.R., Huang, Z., Penczek, P.A., Yang, C., Glaeser, R.M., Adams, P.D., and Ludtke, S.J. (2007). SPARX, a new environment for Cryo-EM image processing. J Struct Biol *157*, 47-55.

Ludtke, S.J., Baldwin, P.R., and Chiu, W. (1999). EMAN: Semiautomated software for high-resolution single-particle reconstructions. J Struct Biol *128*, 82-97.

Mindell, J.A., and Grigorieff, N. (2003). Accurate determination of local defocus and specimen tilt in electron microscopy. Journal of Structural Biology *142*, 334-347.

Murtagh, F. (1983). A surevy of recent advances in hierarchical clustering algorithms. Computer Journal *26*, 354-359.

Penczek, P., Radermacher, M., and Frank, J. (1992). 3-Dimensional Reconstruction of Single Particles Embedded in Ice. Ultramicroscopy *40*, 33-53.

Penczek, P.A. (2010). Image restoration in cryo-electron microscopy. Methods enzymol *482*, 35-72.

Pettersen, E.F., Goddard, T.D., Huang, C.C., Couch, G.S., Greenblatt, D.M., Meng, E.C., and Ferrin, T.E. (2004). UCSF chimera - A visualization system for exploratory research and analysis. J Comput Chem *25*, 1605-1612.

Shaw, P.A., Sahasrabuddhe, C.G., Hodo, H.G., 3rd, and Saunders, G.F. (1978). Transcription of nucleosomes from human chromatin. Nucleic Acids Research *5*, 2999-3012.

Sorzano, C.O.S., Marabini, R., Velazquez-Muriel, J., Bilbao-Castro, J.R., Scheres, S.H.W., Carazo, J.M., and Pascual-Montano, A. (2004). XMIPP: a new generation of an open-source image processing package for electron microscopy. Journal of Structural Biology *148*, 194-204.

Tang, G., Peng, L., Baldwin, P.R., Mann, D.S., Jiang, W., Rees, I., and Ludtke, S.J. (2007). EMAN2: an extensible image processing suite for electron microscopy. J Struct Biol *157*, 38-46.

Tipping, M.E., and Bishop, C.M. (1999). Probabilistic principal component analysis. Journal of the Royal Statistical Society Series B-Statistical Methodology *61*, 611-622.

van Heel, M., Harauz, G., Orlova, E.V., Schmidt, R., and Schatz, M. (1996). A new generation of the IMAGIC image processing system. J Struct Biol *116*, 17-24.