SUSAN MANUAL

RICARDO MIGUEL SÁNCHEZ LOAYZA

Substack Analysis
v0.1

Departmet of Structural Biology
Max Planck Institute for Biophysics

July 2021

CONTENTS

1 INTRODUCTION 1
- 1.1 Design overview 1
  - 1.1.1 Software architecture 1
  - 1.1.2 Multi-thread Multi-GPU design 3
  - 1.1.3 Cluster support 3
- 1.2 Installation 3
  - 1.2.1 Getting the code and the dependencies 4
  - 1.2.2 Compiling the core modules 4
  - 1.2.3 Installing the Matlab package 5
  - 1.2.4 Installing the Python modules 5
- 1.3 A workflow example 6
  - 1.3.1 Tomograms Info and CTF estimation 7
  - 1.3.2 Basic subtomogram averaging on binned data 9
  - 1.3.3 Mid-complexity subtomogram averaging on unbinned data 11

1 INTRODUCTION

SUSAN is a computational framework that uses substacks to solve the subtomogram averaging problem. A substack is a compact representation of a subtomogram, where the three-dimensional volumes are replaced by sets of two-dimensional projections and their image formation information (tilt angles and CTF). Most operations that use subtomograms can be modified to use substacks with the benefit of lowering the computational complexity, and thus, increasing the computational efficiency of the system. With this in mind, SUSAN was designed to be simple, flexible, fast, and with lower computational requirements than equivalent systems. The main algorithms of SUSAN were coded in C++ with a minimal set of dependencies (EIGEN for basic mathematical operations, CUDA to use GPU accelaration, and OpenMPI to use computer clusters). These algorithms are used by a set of Matlab and Python scripts to perform the subtomogram averaging operations.

1.1 DESIGN OVERVIEW

The SUSAN framework is organized in two layers: the high-performance layer, that implements the computationally intenstive algorithms, and the scripting layer, that controls the processing workflow. The high-performance layer implements, mainly, the CTF estimation, 3D and 2D alignment, and the volume reconstruction. These modules were coded using C++ and the nVidia CUDA libraries for GPU accelaration, with the OpenMPI as an optional dependency needed for cluster support.

The scripting layer is built on top of the high-performance one and is designed to simplify its usage. The main implemantation of this layer is provided as a Matlab package, but fully functional Python3 modules are available. The Matlab package can be complemented with other Matlab based CryoET frameworks, like DYNAMO, and depends only on the basic Matlab installation (no additional packages). While the high-performance layer may have support for processing in a cluster, the Matlab package is implemented to work on a single node with one or more GPUs. On the other hand, the Python3 modules for SUSAN can be used to process data on multi-nodes GPU-accelerated clusters.

1.1.1 *Software architecture*

The operations implemented by the high-performance layer are executed over files and their specific behaviour are controlled by a set of input parameters. The scripting layer, besides controlling the processing workflow, provides an abstraction layer for each high-performance

module that simplifies their usage. Apart from the traditional files like 3D maps, masks, or stacks, SUSAN uses three types of files to store the information to process:

TOMOGRAMS INFO: stores the information of all the tomograms in the project. These information includes: tilt angles, CTF per projection, tomogram and stack dimensions, pixel size, and the stack files associated to the tomograms. The file extension must be *.tomostxt* and it is a text file (it can be viewed in any text editor).

PARTICLES INFO: stores the information of all the particles in the project. These information includes: location of the particle in ångströms, alignment angles, shifts and score, per-particle-per-projection CTF, etc. The file extension must be *.ptclsraw* and it is a binary file. Its contents can be modified using the respective Matlab or Python functions.

REFERENCE INFO: stores the information of the alignment references for the project. These information includes: reference maps, masks, and halfmaps. The file extension must be *.refstxt* and it is a text file (it can be viewed in any text editor).

It is important to note that the binning of the project is defined by the stack and pixel size configured on the `Tomograms Info` file. The `Particles Info` file stores the location and shifts of the particles in ångströms and references to the center of the tomogram. This way, the same `Particles Info` file can be used in projects with different binnings.

SUSAN implements three operations required to solve the subtomogram averaging problem:

CTF ESTIMATION: Estimates the CTF using the two-step algorithm and the location of the particles.

- *Input files:* `Tomograms Info` and `Particles Info`.
- *Output files:* (for each tomogram) a txt file with the defocus information and a estimation report in the form of MRC files.

ALIGNER: Aligns a substack against a set of references. The alignment can be in 3D (subtomogram averaging) or in 2D (projection refinement).

- *Input files:* `Tomograms Info`, `Particles Info` and `Reference Info`.
- *Output file:* `Particles Info`.

AVERAGER: Reconstruct volumes.

- *Input files:* `Tomograms Info` and `Particles Info`.
- *Output files:* A set of 3D maps (MRC files).

1.1.2 *Multi-thread Multi-GPU design*

SUSAN uses substacks that are cropped on-the-fly from the aligned stack and performs the CTF correction according to the selected operation (alignment or reconstruction). With this approach we decrease the computational resources needed in a typical subtomogram averaging pipeline, as we no longer need the CTF corrected stacks, the full tomogram reconstructions, or all the subtomograms in multiple binning stages. Additionally, SUSAN uses the *producer-consumer pattern* for concurrent multi-threaded computers to increase the computational performance. In this paradigm, the system uses a double buffer to share information between a producer thread and one or more consumer threads. The producer reads the data to be processed and stores it in one of the buffer while the consumer threads process the data available in the other buffer. Once the producer and consumer threads finish their tasks, the buffers are swapped and the threads use the new buffers. This way we can reduce the execution time by overlapping the reading times with the execution times.

The high-performance layer of SUSAN uses the *producer-consumer pattern* in two instances. The first one is when reading the stacks associated to the tomograms. The producer thread loads the full stack into system memory while multiple consumer threads, one for each GPU, crop substacks from the already available stack. The second usage of the pattern is on the consumer threads of the previous step, where each of them becomes a producer thread, that normalizes the cropped substacks and uploads them into the GPU, and creates a new consumer threads that process the substacks using the GPU.

1.1.3 *Cluster support*

SUSAN provides OpenMPI support for the `Aligner` and `Reconstruction` modules. For both cases, the associated `Particles Info` file is scattered among the processing nodes and each of them process the it locally. After the normal execution is finished, the information if gathered into the main node and saved. For the `Aligner` module, the gathering process involves the resulting `Particles Info`, while for the `Reconstruction` module, the gathered data are the partial reconstructed maps. It is important to consider that this scatter/gather operation increases the execution time according to the size of the data to transfer, and that data is not shared using storage system but using the gather/scatter functions of the OpenMPI libraries.

1.2 INSTALLATION

SUSAN is an Open Source project (GPLv3.0) that can be downloaded from the GitHub account from the Kudryashev lab.

For our examples we will install SUSAN in the following folder:

```
/home/user/Software/CryoET/
```

1.2.1 *Getting the code and the dependencies*

We use the `git` command to download the code:

```
mkdir -p ~/Software/CryoET
cd ~/Software/CryoET/
git clone https://github.com/KudryashevLab/SUSAN
```

As we mention before, SUSAN depends on two libraries (EIGEN and CUDA), needing OpenMPI only if cluster support is required. To install the dependencies:

EIGEN: Download the latest version of the projection from its Gitlab server (compilation is not needed):

```
cd ~/Software/CryoET/SUSAN/dependencies
git clone https://gitlab.com/libeigen/eigen.get eigen
```

CUDA: Install the CUDA libraries using the software package manager of your linux distribution. If you do not have administrator rights to install software, contact your system administrator and request the installation of the latest CUDA libraries. For more details follow the instructions on the CUDA toolkit documentation webpage:
https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html#package-manager-installation

OPENMPI: *[Optional]* Install the OpenMPI libraries using the software package manager of your linux distribution. If you do not have administrator rights to install software, contact your system administrator and request the installation of the latest OpenMPI libraries. Examples of installations for different linux distributions:

- Ubuntu/Debian:

  ```
  sudo apt-get install -y openmpi-bin
  ```

- CentOS/Fedora/RedHat:

  ```
  sudo yum install openmpi-devel
  ```

- ArchLinux:

  ```
  sudo pacman -S openmpi
  ```

1.2.2 *Compiling the core modules*

The compilation process uses the *GCC* compiler and the *CMake* software:

```
cd ~/Software/CryoET/SUSAN/+SUSAN
mkdir bin
cd bin
cmake ../
make -j
```

KNOWN ERROR: if when compiling SUSAN the error:

```
error #308: more than one instance of overloaded function "round"
     matches the argument list
```

is shown, or a similar one for functions like `min` or `max`, then the *GCC* version is too old. Use a newer version (tested version: 8.4).

KNOWN WARNINGS: Using EIGEN in a CUDA project makes the compiler to show messages like:

```
warning #2979-D: calling a __host__ function from a __host__ __
    device__ function is not allowed
```

or

```
warning #2976-D: calling a __host__ function(...) is not allowed
```

These messages are harmless and can be safely ignored. They can be disabled by editing the CMakeLists.txt file. Add the following lines:

```
string(APPEND CMAKE_CUDA_FLAGS " -Xcudafe --diag_suppress=2976")
string(APPEND CMAKE_CUDA_FLAGS " -Xcudafe --diag_suppress=2976")
```

before the first `add_executable` command. Note that the numbers on each added line match the warning numbers reported. These number can change in function of the CUDA version.

1.2.3 *Installing the Matlab package*

Some operations of the SUSAN Matlab package are implemented in C/C++ to speed up the execution time. These operation must be compiled. Assuming that Matlab is available as an executable on the path, the `mex` compilar should also be available. The compilation steps are:

```
cd ~/Software/CryoET/SUSAN/+SUSAN
make
```

To use SUSAN in Matlab we have to add the location of the package to its working path:

```
>> addpath ~/Software/CryoET/SUSAN/
```

To verify the correct installation of the SUSAN package in the current Matlab instance, check if the documentation can be accessed:

```
>> help SUSAN
```

1.2.4 *Installing the Python modules*

The SUSAN Python module depends on the numpy one. If it is not installed, use `pip` to install it. In a python console execute:

```
pip3 install --user numpy
```

To use SUSAN on a python script, or in a python project in general, we have to add the SUSAN path to the python instance along with the dependencies. After that we can import the SUSAN module and its submodules:

```
import sys,os,datetime,math
sys.path.append('~/Software/CryoET/SUSAN/+SUSAN')
import susan,susan,project,susan.data,susan.modules
```

1.3 A WORKFLOW EXAMPLE

This example uses the Matlab package of SUSAN to make the initial CTF estimation and execute two subtomogram averaging projects, one with binned data and the second one with unbinned data. The project folder is `/home/user/ribosome-example/` and it contains the stack files of four tomograms ($N = 4$) in the `data` folder. Each stack has 41 projections ($K = 41$) and the binned stacks are available. The folder structure, showing only the needed files, is:

```
/home/user/ribosome-example/data/
├── tomo1/
│   ├── stack_ali.mrc
│   ├── stack_ali_bin2.mrc
│   └── stack.tlt
├── tomo2/
│   ├── stack_ali.mrc
│   ├── stack_ali_bin2.mrc
│   └── stack.tlt
├── tomo3/
│   ├── stack_ali.mrc
│   ├── stack_ali_bin2.mrc
│   └── stack.tlt
└── tomo4/
    ├── stack_ali.mrc
    ├── stack_ali_bin2.mrc
    └── stack.tlt
```

The SUSAN projects will be located in the `susan` folder, and the initial files are: the initial bin2 reference, the masks for the bin2 and unbinned projects, the DYNAMO table with the location of the particles in bin2 and a valid *tomogram number* entry (column 20), and the matlab script that uses SUSAN:

```
/home/user/ribosome-example/susan/
├── mask_bin2.mrc
├── mask_unbin.mrc
├── picked_bin2.tbl
├── reference_bin2.mrc
└── workflow.m
```

Finally, the project will be executed on a workstation with 4 GPUs. In what follows we will explain the contents of `workflow.m` having */home/user/ribosome-example/susan/* as the working folder.

KNOWN ISSUE: Naming the working folder as `SUSAN` will create conflicts with the SUSAN package module. We recommend using names like `ssa`, `SSA`, `susan` or `Susan`.

1.3.1 *Tomograms Info and CTF estimation*

The first step in the project is to add the SUSAN package to the Matlab's path, if it wasn't done before:

```
%% Add SUSAN to the path
addpath ~/Software/CryoET/SUSAN/
```

The next step is to create a basic `Tomograms Info` file and a `Particles Info` file to perform the CTF estimation. In our example, we have 4 tomograms ($N = 4$) and all the related stacks have the same number of projections ($K = 41$) and the same dimensions (same thickness). If the stacks have different number of projections, K must be the maximum number of projections of all the available stacks. If the tomograms have different thickness, they should be set up independently.

The CTF estimation module uses the location of the particles, which are not always known before hand, so the typical approach is to create a grid in the central plane of all the tomograms to have an intial CTF estimation:

```
%% Create the Tomograms Info
N = 4;
K = 41;
pix_size = 2.62;
tomo_size = [3710 3710 880]; % Tomograms of same size

tomos = SUSAN.Data.TomosInfo(N,K);

%% Programatically set up the needed files:
for n = 1:N
   tomos_base = ['../data/tomo' num2str(n) '/stack'];
   tomos.tomo_id(n) = n; % Same as the dynamo catalog/column 20.
   tomos.set_stack (n,[tomo_base '_ali.mrc']);
   tomos.set_angles(n,[tomo_base '.tlt']);
   tomos.pix_size (n,:) = pix_size;
   tomos.tomo_size(n,:) = tomo_size;
end
tomos.save('tomo_raw_b1.tomostxt');

%% Create a 2D grid to estimate the CTF:
sampling = 180; % spacing between particles, in pixels
grid_ctf = SUSAN.Data.ParticlesInfo.grid2D(sampling,tomos);
grid_ctf.save('grid_ctf.ptclsraw');
```

The SUSAN Matlab package uses object-oriented approach. As such, we have to create an instance of the class that performs the CTF estimation and configure it to use a patch size of 400 pixels.

```
%% Create CtfEstimator
ctf_est = SUSAN.Modules.CtfEstimator(400);
ctf_est.binnig = 0; % No binning (2^0).
ctf_est.gpu_list = [0 1 2 3]; % 4 GPUs available.
ctf_est.resolution.min = 30;  % angstroms
ctf_est.resolution.max = 8.5; % angstroms
ctf_est.defocus.min = 10000;  % angstroms
ctf_est.defocus.max = 50000;  % angstroms
```

```
tomos_ctf = ctf_est.estimate('ctf_grid','grid_ctf.ptclsraw',...
            'tomo_raw_b1.tomostxt');
tomos_ctf.save('tomo_ctf_b1.tomostxt');
```

As mentioned before, the CtfEstimator module recieves two inputs, `grid_ctf.ptclsraw` and `tomo_raw_b1.tomostxt`, and creates a set of folders and files with the results of the estimation. The number of files created depends on the requested verbosity level. In this example, we used the default verbosity, which creates the minimum number of files. Additionally, it returns a new `Tomograms Info` instance with the estimated values. The created files include the CTF estimation results and diagnostic files:

```
/home/user/ribosome-example/susan/ctf_grid/
├── Tomo001/
│   ├── ctf_fitting_result.mrc
│   └── defocus.txt
├── Tomo002/
│   ├── ctf_fitting_result.mrc
│   └── defocus.txt
├── Tomo003/
│   ├── ctf_fitting_result.mrc
│   └── defocus.txt
└── Tomo004/
    ├── ctf_fitting_result.mrc
    └── defocus.txt
```

Next, we will create the `Tomograms Info` for the binned stacks and use it to create a `Particles Info` from the `picked_bin2.tbl` file:

```
%% Create the bin2 Tomograms Info
N = 4;
K = 41;
pix_size = 2.62*2;
tomo_size = [3710 3710 880]/2;

tomos = SUSAN.Data.TomosInfo(N,K);

%% Programatically set up the needed files:
for n = 1:N
   tomos_base = ['../data/tomo' num2str(n) '/stack'];
   ctf_file = sprintf('ctf_grid/Tomo%03d/defocus.txt',n);
   tomos.tomo_id(n) = n; % Same as the dynamo catalog/column 20.

   % Changed to bin2 data
   tomos.set_stack  (n,[tomo_base '_ali_bin2.mrc']);
   tomos.set_angles (n,[tomo_base '.tlt']);

   % Load DefU, DefV and the angles.
   tomos.set_defocus(n,tomos_base,'Basic');
   tomos.pix_size (n,:) = pix_size;
   tomos.tomo_size(n,:) = tomo_size;
end
tomos.save('tomo_ctf_b2.tomostxt');

%% Create a Particles Info class from a table:
tbl = SUSAN.read('picked_bin2.tbl');
```

```
ptcls = SUSAN.Data.ParticlesInfo(tbl,tomos);
ptcls.save('project_001.ptclsraw');
```

1.3.2 *Basic subtomogram averaging on binned data*

To execute a subtomogram averaging project we need the `Tomograms Info`, `Particles Info` and `Reference Info` files. The first two are available from the previous step, we have to create the last one:

```
%% Create the bin2 Reference Info with only one reference
refs = SUSAN.Data.ReferenceInfo.create(1);
refs(1).map  = 'reference_bin2.mrc';
refs(1).mask = 'mask_bin2.mrc';
refs.save(refs,'project_001.refstxt');
```

The box size for the binned project is 192 voxels. We create the project and configure it for our box size, the GPUs available and the initial files:

```
%% Create the Project 001: Bin2
mngr = SUSAN.Project.Manager('project_001',192);
mngr.initial_reference = 'project_001.refstxt';
mngr.initial_particles = 'project_001.ptclsraw';
mngr.tomogram_file     = 'tomo_ctf_b2.tomostxt';
mngr.gpu_list          = [0 1 2 3];
```

SUSAN can be used in a similar way as DYNAMO, with various round of multiple iterations each. For this example we will use three rounds, with one, three and five iterations each. The first round estimates is a pure translation alignment from the center of the box. The second and third rounds include angular search with a smaller search in the third one, and with the offsets calculated from the particles location estimated in the previous iteration. In all iteration we use 90% of the particles for the reconstruction:

```
%% For all iterations;
mngr.cc_threshold = 0.90;

%% Round 1 with 1 Iteration:
% Iterations: 1.
mngr.aligner.drift = false;
mngr.aligner.set_angular_search(0,1,0,1);
mngr.aligner.set_offset_ellipsoid(20);
mngr.aligner.bandpass.lowpass = 20;
mngr.execute_iteration(1);

%% Round 2 with 3 Iterations:
% Iterations: from 2 to 4.
mngr.aligner.drift = true;
for i = 2:4
   mngr.aligner.set_angular_search(36,4,36,4);
   mngr.aligner.set_angular_refinement(1,2);
   mngr.aligner.set_offset_ellipsoid(6);
   mngr.aligner.bandpass.lowpass = 30;
   mngr.execute_iteration(i);
```

```
end

%% Round 3 with 5 Iterations:
% Iterations: from 5 to 9.
mngr.aligner.drift = true;
for i = 5:9
   mngr.aligner.set_angular_search(8,2,8,2);
   mngr.aligner.set_angular_refinement(3,2);
   mngr.aligner.set_offset_ellipsoid(6);
   mngr.aligner.bandpass.lowpass = 40;
   mngr.execute_iteration(i);
end
```

The `mngr.aligner.drift` controls the reference for the translation of the particles. When disabled, the reference will be the location of the particle as stated in the `position` property of the `Particles Info` class (equivalent to DYNAMO's area search modus 1). When enabled, the reference will be the `position` plus the shifts (property `ali_t` of the `Particles Info` class) from the previous iteration (equivalent to DYNAMO's area search modus 2). The angular search values are similar to the ones used by DYNAMO. The arguments for the `mngr.aligner.set_angular_search` method are cone range, cone step, inplane range, and inplane step, in that order. The arguments for the `mngr.aligner.set_angular_refinement` method are, in order, refinement level and refinement factor.

SUSAN creates a folder for the project where the information of each iteration is stored (resulting `Particles Info`, `References Info`, halfmaps and final averages, and log files). For example, for the last iteration of the project:

```
/home/user/ribosome-example/susan/project_001/
├── info.prjtxt
└── ite_0009/
    ├── log.txt
    ├── map_class001_half1.mrc
    ├── map_class001_half2.mrc
    ├── map_class001.mrc
    ├── particles.ptclsraw
    ├── reference.refstxt
    ├── stdout.alignment
    └── stdout.reconstruction
```

To check the results of the project, we display the FSC curve of the last iteration and then we show the bandpassed reconstructed map. For the last operation we use a couple of function of the DYNAMO framework:

```
%% Show FSC of iteration 9
figure;
mngr.show_fsc(9);

%% Display recosntruction of iteration 9
dtmshow( dbandpass(mngr.get_map(9),[0 45 5]) );
```

For the next project we can use the resulting `Particles Info` file of the last iteration (`project_001/ite_0009/particles.ptclsraw`), or create a local copy of it while updating the location of the particles:

```
%% Load partciles of iteration 9 and save them
ptcls = mngr.get_ptcls(9);
ptcls.position = ptcls.position + ptcls.ali_t;
ptcls.ali_t(:) = 0;
ptcls.save('project_002.ptclsraw');
```

LOAD AN EXISTING PROJECT: To access an existing project (old project or from another MATLAB session) we can execute the following command:

```
%% Load Project 001: Bin2
mngr = SUSAN.Project.Manager('project_001');
```

With this we obtain access to functions like `mngr.get_map` or `mngr.show_fsc`. It is important to note that the project will not be configured, and will need configuration in order to execute iterations from this loaded instance.

1.3.3 *Mid-complexity subtomogram averaging on unbinned data*

While the initial reference was available for the binned data project, we have to create one for the unbinned data project. For that we use the `project_002.ptclsraw` file and the stand-alone Averager module to reconstruct a $384 \times 384 \times 384$ volume from the unbinned `Tomograms Info` file (`tomo_ctf_b1.tomostxt`):

```
%% Create and configure the Averager module:
avgr = SUSAN.Modules.Averager;
avgr.gpu_list = [0 1 2 3];

%% Execute the reconstruction:
avgr.reconstruct('project_002','tomo_ctf_b1.tomostxt','
    project_002.ptclsraw',384);
```

It is important to note that, by design, the changes needed to use a different binning level are the box size of the volumes and the `Tomograms Info` files. The reconstruction procedure creates the `project_002_class001.mrc` file that we will use as intial reference.

Before creating the subtomogram averaging project, we have to create the `Reference Info` file:

```
%% Create the unbinned Reference Info
refs = SUSAN.Data.ReferenceInfo.create(1);
refs(1).map  = 'project_002_class001.mrc';
refs(1).mask = 'mask_unbin.mrc';
refs.save(refs,'project_002.refstxt');
```

The box size for the unbinned project is 384 voxels. We create the project and configure it for our box size, the GPUs available and the initial files, and to use the best 90% of the particles:

```
%% Create the Project 002: unbinned
mngr = SUSAN.Project.Manager('project_002',384);
mngr.initial_reference = 'project_002.refstxt';
mngr.initial_particles = 'project_002.ptclsraw';
mngr.tomogram_file     = 'tomo_ctf_b1.tomostxt';
mngr.gpu_list          = [0 1 2 3];

%% For all iterations;
mngr.cc_threshold = 0.90;
```

For this example we will also use three rounds of iterations. The first round has one iteration without angular search. This is recommended after changing of binning of a project to fix any additional offsets:

```
%% Round 1 with 1 Iteration:
mngr.aligner.drift = false;
mngr.aligner.set_angular_search(0,1,0,1);
mngr.aligner.set_offset_ellipsoid(10);
mngr.aligner.bandpass.lowpass = 40;
mngr.execute_iteration(1);
```

For the second round we will use a bandpass sweep over 6 iterations. That means, we will increase the lowpass filter on each iteration. Additionally, we will calculate the angular search step size accoring to the lowpass value used. This behaviour can be implemented in two ways. The first one is to have an array with the lowpass values to use:

```
%% Round 2 (6 iter.) with a lowpass array:
mngr.aligner.drift = true;
low_pass_array = linspace(45,55,6); % or 45:2:55
for i = 2:7
   low_pass = low_pass_array(i-1); % adjust index offset
   as = atan2d(1,low_pass)  % angular sampling
   mngr.aligner.set_angular_search(4*as,as,4*as,as);
   mngr.aligner.set_angular_refinement(1,2);
   mngr.aligner.set_offset_ellipsoid(6);
   mngr.aligner.bandpass.lowpass = low_pass;
   mngr.execute_iteration(i);
end
```

The second way of implementing the same behaviour is to increase the lowpass on each iteration:

```
%% Round 2 (6 iter.) increasing the lowpass:
mngr.aligner.drift = true;
low_pass = 45;
for i = 2:7
   as = atan2d(1,low_pass)  % angular sampling
   mngr.aligner.set_angular_search(4*as,as,4*as,as);
   mngr.aligner.set_angular_refinement(1,2);
   mngr.aligner.set_offset_ellipsoid(6);
   mngr.aligner.bandpass.lowpass = low_pass;
   mngr.execute_iteration(i);
   low_pass = low_pass + 2;
end
```

For the third round we will use ten iterations in a closed-loop. That means, we will use the resolution estimation of the previous iteration to calculate the lowpass filter and a angular search step size. The method `mngr.execute_iteration(i)` returns a list of the resolutions estimated for each class on the project using the FSC and the 0.143 threshold. In our case we have only on class, so, the round can be implemented as follows:

```
%% Round 3 (10 iter.) closed-loop:
mngr.aligner.drift = true;
low_pass = 52;
for i = 8:17
   as = atan2d(1,low_pass)  % angular sampling
   mngr.aligner.set_angular_search(4*as,as,4*as,as);
   mngr.aligner.set_angular_refinement(1,2);
   mngr.aligner.set_offset_ellipsoid(6);
   mngr.aligner.bandpass.lowpass = low_pass;
   low_pass = mngr.execute_iteration(i);
end
```

In some cases, the estimated resolution can increase rapidly and can result in overfitted maps. To prevent this issue, the growth rate of the lowpass filter value can be limited. Additionally, the FSC threshold value can be configured from the default 0.143 to a more conservative 0.5. With this changes, we can rewrite the thrd round as:

```
%% Round 3 (10 iter.) conservative closed-loop:
mngr.fsc_threshold = 0.5;
mngr.aligner.drift = true;
low_pass = 52;
for i = 8:17
   as = atan2d(1,low_pass)  % angular sampling
   mngr.aligner.set_angular_search(4*as,as,4*as,as);
   mngr.aligner.set_angular_refinement(1,2);
   mngr.aligner.set_offset_ellipsoid(6);
   mngr.aligner.bandpass.lowpass = low_pass;
   low_pass_in = mngr.execute_iteration(i);
   low_pass = min(low_pass_in,low_pass+2);
end
```

Finally, the resulting maps, including the halfmaps, can be saved with the contrast inverted. This resulting maps can be used in any postprocessing software:

```
%% Get the filenames of the resulting maps:
m = mngr.get_name_map(17);
[h1,h2] = mngr.get_name_halves(17);

%% load the files, read the pixel size only for one map.
[map,pix_size] = SUSAN.IO.read_mrc(m);
map_h1 = SUSAN.IO.read_mrc(h1);
map_h2 = SUSAN.IO.read_mrc(h2);

%% save the maps inverting the contrast:
SUSAN.IO.write_mrc(-map,'rslt_raw.mrc',pix_size);
SUSAN.IO.write_mrc(-map_h1,'rslt_h1.mrc',pix_size);
SUSAN.IO.write_mrc(-map_h2,'rslt_h2.mrc',pix_size);
```