You can view this tutorial as pdf by typing on your terminal:

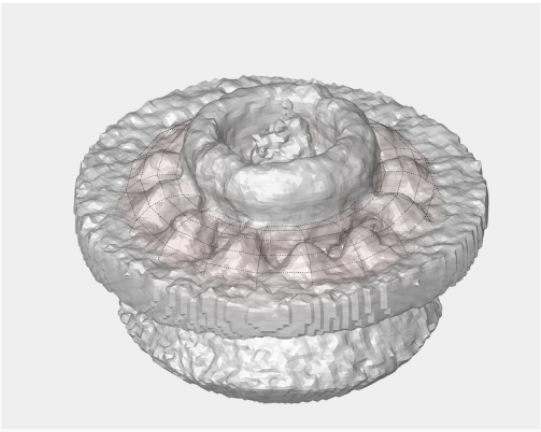
> cd Prac-4

>evince dynamo_tutorial.pdf

You can then use it to copy-paste commands, but always double-check before running as sometimes characters are not reproduced properly

The example data set

The data is a fraction of a tomogram. The full tomogram was used in "Cryo-electron tomography reveals novel features of a viral RNA replication compartment." (Ertel et al.), and represents several FHV viruses docked in the outer membrane of a mitochondrion.



Average for several hundreds of particles. The "crown" area is shown under a red shadow.

Change directory to Prac-4, the cropped tomogram file is there with name crop.rec.

>cd Prac-4

>ls

Now, open *Dynamo* by typing:

>dynamo

The *Dynamo* command line will open

Size check of a file

You're probably curious to see what's inside, so that let's write first:

```
dfile crop.rec
```

to let *Dynamo* check the dimensions of the file. The header of a `.rec` file is read as a regular `mrc`, yielding:

```
 filetype: volume
```

```
size: 1285 x 956 x 786
```

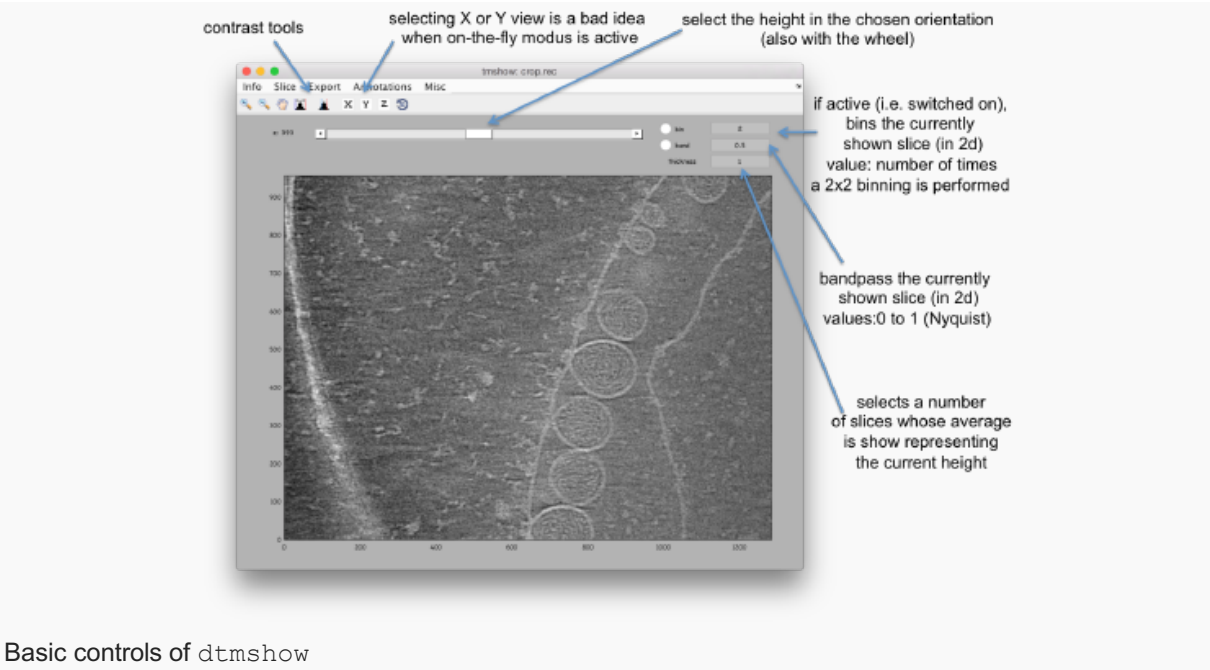
So, it's a tomogram.

Lightweight visualization

We can inspect quickly its contents with dtmshow

```
dtmshow -otf crop.rec
```

Hereby, the flag `-otf` means "on the fly", telling `dtmshow` to *not* preload the full tomogram, but to access in disk the individual slices that are needed when inspecting a particular area.



Basic controls of `dtmshow`

Go up and down. We want to select the locations were the vesicles intersect the mitochondrion membrane and average them together. For this, we need to *catalogue* the tomogram, so that our annotations are stored with a clear relationship with the tomogram.

Cataloguing the tomogram

We can create catalogues just to contain a single tomogram. They are useful to keep track of all annotations, and of the typical transforms (binning, cropping of fractions) that we usually perform on a larged size tomogram of interest. In this case, we can create the catalogue directly from the command line. Close the dtmshow window and type:

```
dcm -create fhv
```

where `dcm` is the short form of `dynamo_catalogue_manager` and `fhv` is just an arbitrary name. The just created catalogue is empty, and we can add our tomogram with:

```
dcm -c fhv -at crop.rec
```

We can check that the tomogram is in the catalogue by asking *Dynamo* to show the contents of the catalogue

```
dcm -c fhv -l tomograms
```

(the flag is the letter l, not the number 1)

or

```
dcm -c fhv -l t
```

The flag `-l` asks *Dynamo* to list items of a given category of catalogue contents, in this case `tomograms`

Prebinning the tomograms

We typically want to prebin the tomogram, i.e., have a version of |smaller size that is known to the catalogue. This version will be useful in some operations that require a full tomogram in memory, an operation that can consume much memory and need a long time. In this example, this is probably not necessary: a tomogram with a sidelength on x and y of ~1000 pixels shouldn't pose any visualization problem. Still, in the command line, we can write:

```
dynamo_catalogue_bin('fhv', 1, 'zchunk', 300);
```

where the parameter `zchunk` represents the maximum number of z slices that are kept simultaneously in the memory during the binning process. This parameter might be important for larger size tomograms.

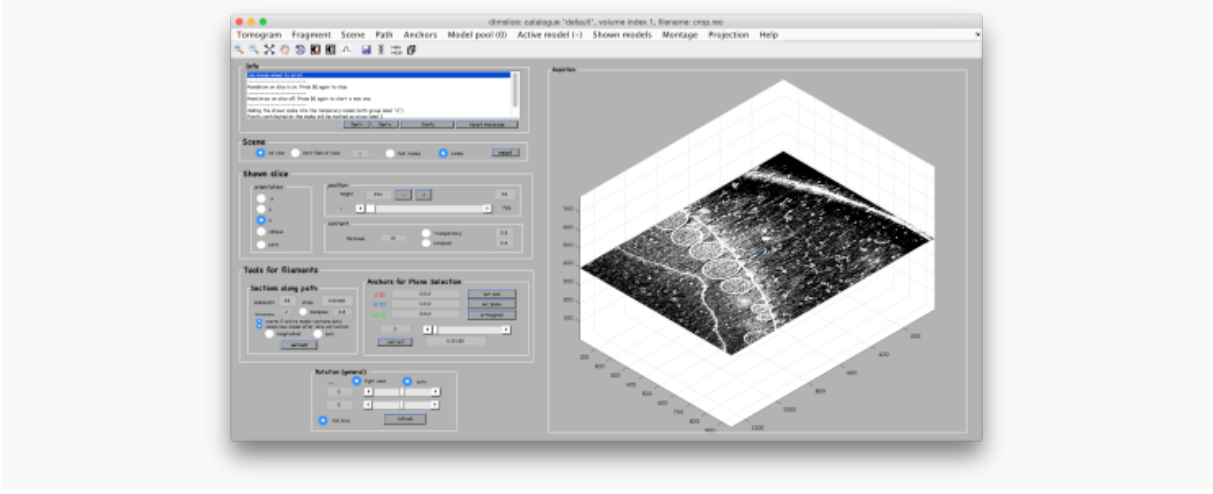
Operation with GUI

These steps could have been performed thorough the `dcm` GUI

Annotation of particle positions

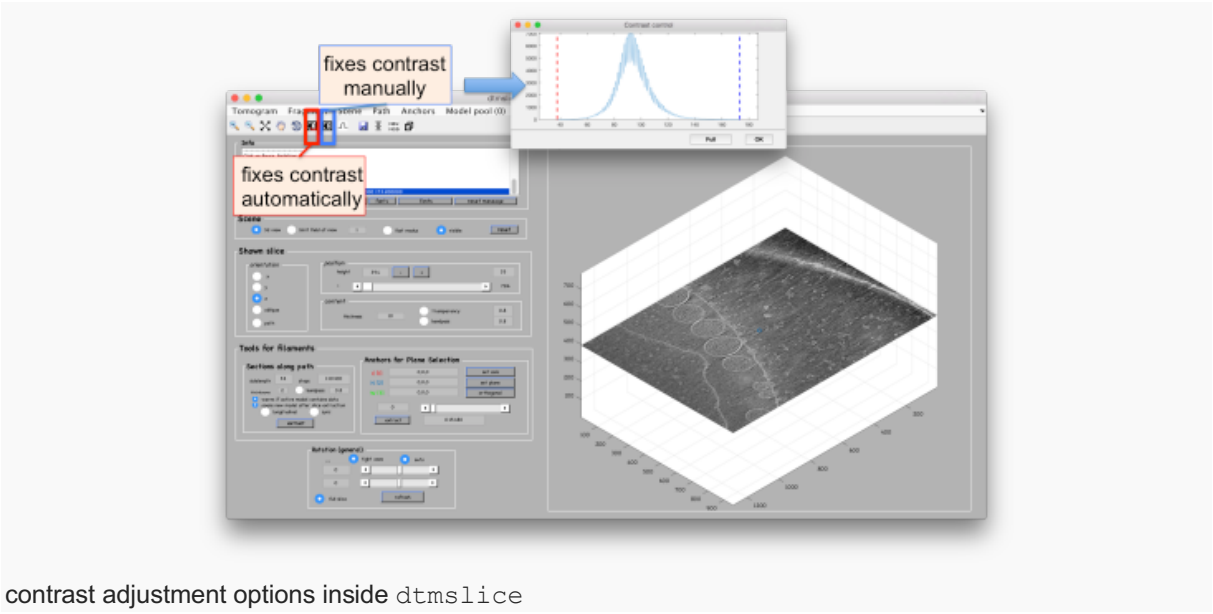
Now we can open the tomogram through the catalogue:

```
dtmslice crop.rec -c fhv  -prebinned 1
```



dtmslice opened on the FHV example tomogram`dtmslice`

Probably you don't like the initial contrast, change it with the button in the toolbar.



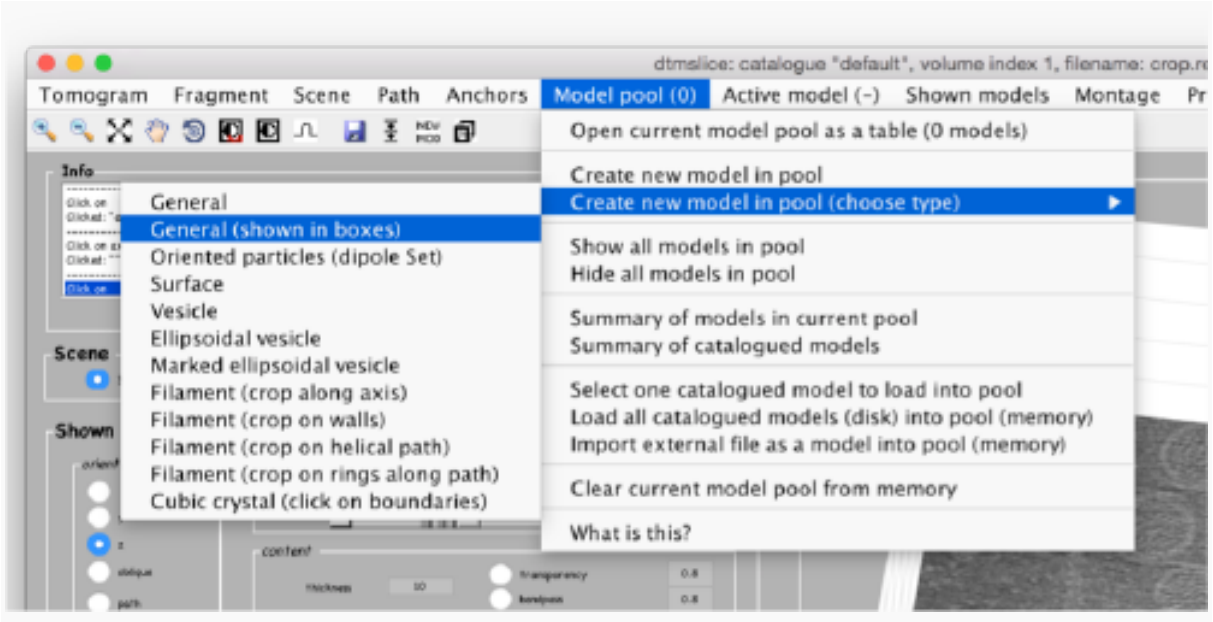
contrast adjustment options inside `dtmslice`

Navigating the tomogram

Use the bar to move the slice up and dow, or drag it with the cursor while keeping the main mouse button pressed. Other auxiliary tools are the keys x,y,z to change the slice orientation, the number of projected slices (called "thickness" in the GUI controls).

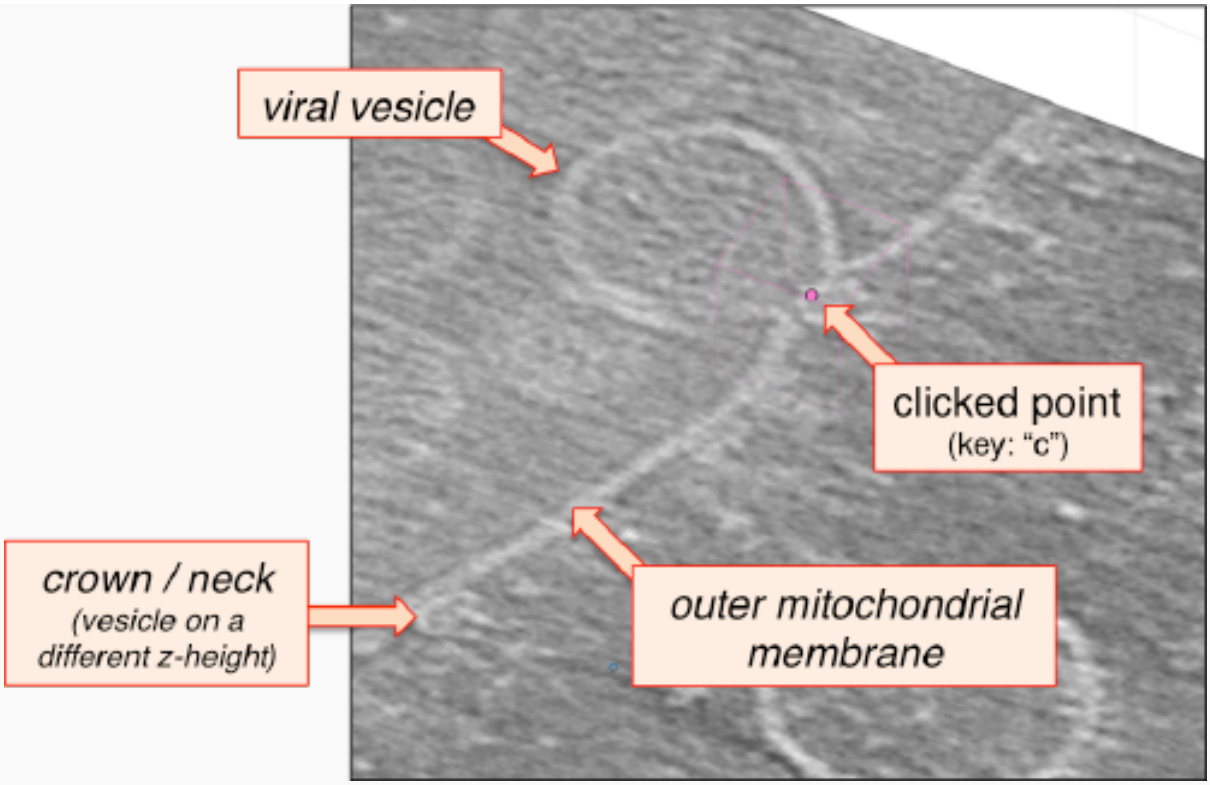
Creation of models to contain annotations

In this example we just want to manually pick some particles. This can be done creating a general or box model, which will reside in memory till we save it into the catalogue.



Creating a new model in the scene. Each annotation is represented by a box

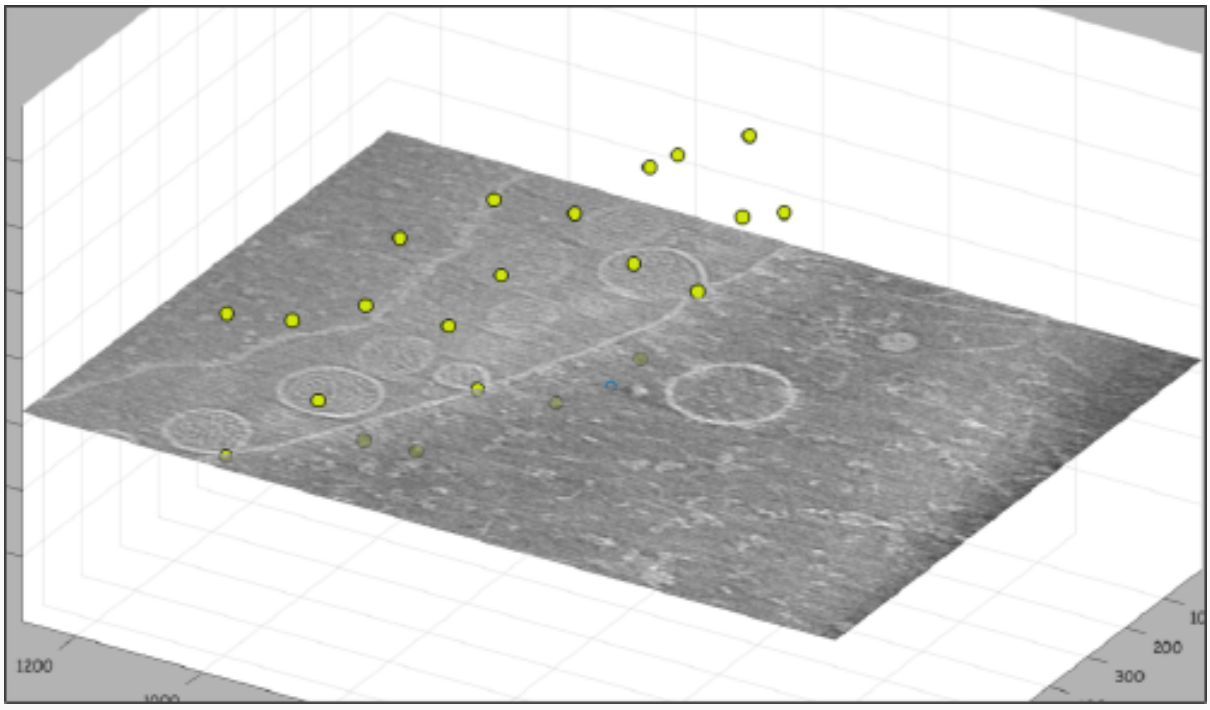
After creating the model, it will be only model currently active in the `dtmslice` scene. You can add new points pressing on [c]. The idea is to mark on the positions where you see the "neck" of a vesicle (what we called "crowns") in contact with the outer mitochondrial membrane.



Clicking the crowns on screen

The last marked point can be deleted by pressing [delete]. An arbitrary point can be deleted by clicking on it with the right mouse button. This will open a menu that includes the option of deleting the point (through Ctrl+X in Linux or Cmmd+X in Mac).

At this stage you probably want to change the transparency of the depicted slice, so that you can control which objects have been already clicked below the depicted slide.



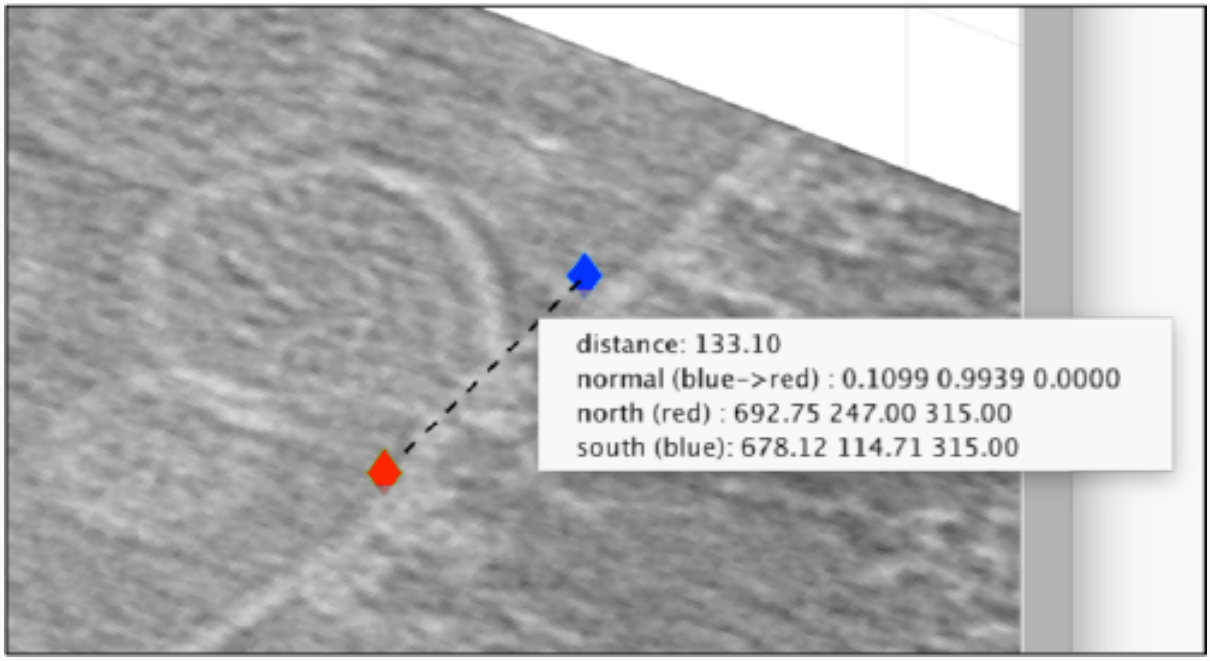
Selection of points. Transparency of slide was set to 0.8

.

When you are done, remember to save the model, clicking on the disk icon to save or Active model #1 > save active model into catalogue (disk).

Cropping particles

Now we want to use the positions that we have marked to extract the subtomograms and format them as a data folder. The first thing we need is an estimation of the sidelength in pixels of each of the subtomograms. The subtomogram box should be ~2 to 3 times the size of the particle. In dtmslice We can use the keys [1] and [2] to define two anchor points that appear as rombohedra (you might need to zoom in). Clicking (with the right button) onto the black dashed line that links the points will show on screen both coordinates and the distance between them. All distances are reported in pixels of the non-binned tomogram: even if you are using a binned version, *Dynamo* keeps track of it.



Measuring distances with [1] , [2] and right-click

.

We will thus choose to create a datafolder with a cubic sidelength of 128 pixels (remember that the particles will be cropped in the *unbinned* tomogram). This will ensure that the crowns fit comfortably inside the physical box, even if our manual picking imposes an error of several pixels. If you were using, say, a *thickness* parameter of 10 pixles in `dtmslice`, you have to count with at least this inaccuracy in the location of the particles.

Now, we check that the catalogued tomogram contains the model that we manually picked before:

close the GUI

```
>> dcmodels fhv
Passing to MATLAB…
Volume 1. Matching models: 1 (total: 1)
/i/embo2019/u/emboX/Prac-4/fhv/tomogram/volume_1/modules/mboxes.omd
```

(ignore the openmpi error)

Creating a table

We could just use the catalogue GUI to extract the particles, be it is also possible to proceed directly with the command line. We will use the `dtcrop` command, .which requires preparing a table with the information of the model.

```
m = dread('/i/embo2019/u/emboX/Prac-4/fhv/tomograms/volume_1/models/mboxes.omd');
t = m.grepTable();
```

Here, you read the file into a model object (which we arbitrarily choose to call `m`), and then you use the `grepTable` method on this object to extract a variable into your workspace. We arbitrarily call it `t`.

```
>> dtinfo(t);

        size                :  22 35
        NaNs                :  0

 COLUMN
[  2  ] marked for alignment:  22
[  3  ] included in average :  22
[ 4-6 ] shifts              :  all zero
[ 7-9 ] angles              :  all zero
[  10 ] cross correlation   :  min: 0.00  max: 0.00  mean: 0.00  std:
0.00
[  13 ] Fourier sampling    :  1 (single tilt around y)
[  13 ] fsampling types     :  all of the same type
[14-15] ytilt range         :  min:120.00   max:120.00
[16-17] xtilt range         :  min:120.00   max:120.00
[  20 ] linked volumes      :  total 1 (labels: [1])
[  21 ] regions inside tomograms :  total 1 (labels: [0])
[  22 ] user-defined classes:  total 1 (labels: [0])
[  23 ] annotation types    :  total 1 (labels: [0])
[24-26] spatial locations   :  initialized: 22
[  24 ]   *  x              :  min: 645.21  max: 1001.92  mean: 799.05
std: 109.44
[  25 ]   *  y              :  min: 23.78  max: 917.51  mean: 484.07
std: 271.28
[  26 ]   *  z              :  min: 198.00  max: 563.00  mean: 415.55
std: 114.03
[  31 ] original tags       :  total 1 (labels: [0])
[  32 ] compacted particles :  total 1 (labels: [1])
[  34 ] references          :  total 1 (labels: [0])
[  35 ] subreferences       :  total 1 (labels: [0])
[  36 ] apix                :  Warning: column not available in this
table
```

```
[  37 ] defocus             :  Warning: column not available in this
table
```

Using `dtcrop`

The simplest syntax of `dtcrop` requires passing the name of the tomogram from which we want to crop (syntax varies for cropping from multiple tomograms). We know that the file is `crop.rec`, and we could directly insert this name in the command. But a catalogued model already contains information about its source tomogram (inside its property `cvolume`), so that we can always track it back. We could then define a variable `tomogramFile` by accessing this information inside the model variable `m`:

```
tomogramFile = m.cvolume.file();
```

(ignore the warning message)

and launch the cropping order:

```
o = dtcrop(tomogramFile,t,'particlesData',128);
```

The last part of the final output into screen should look like this:

```
21 [read_subtomogram] Volume has size 1285 956 786
[read_subtomogram] Accessing subvolume x: 713:840; y: 339:466; z:
160:287  totalling ~ 16.0Mb
Elapsed time is 0.191014 seconds.
22
Total time invested in cropping: 7s
[table_crop] Done extracting 20 particles
             from tomogram      :"/d/embo2019/u/emboX/Prac-4/crop.rec"
             destination folder :"particlesData"
             excluded particles : 2

 [ok] table_crop
```

informing you that some of the particles where excluded, as they were probably too close to the boundary of the tomogram, given the sidelength we asked for. Inside the created data folder, you will find the table `particlesData/crop.tbl`, which only indexes the actually cropped particles.

Creating an average

The particles can now be averaged together. They have different orientations, but in this tomogram we only have a fraction of the membrane.

```
oa = daverage('particlesData','t','particlesData/crop.tbl');
dwrite(t,'raw.tbl');
dwrite(oa.average,'rawAverage.em');
```

You can now look at the average with chimera - open another terminal and type ‘chimera’

Using the membrane to impart an orientation

This section describes a method to compute:

- a rough orientation of the particles, and
- a rough first template

In this section we will use the membrane of the mithocondrion to assign a roughly orientation to each of the points in the table. This orientation will be defined by the normal of the closest point in the membrane. The membrane will be defined as a a triangulation, to be constructed based on a set of manually picked points.

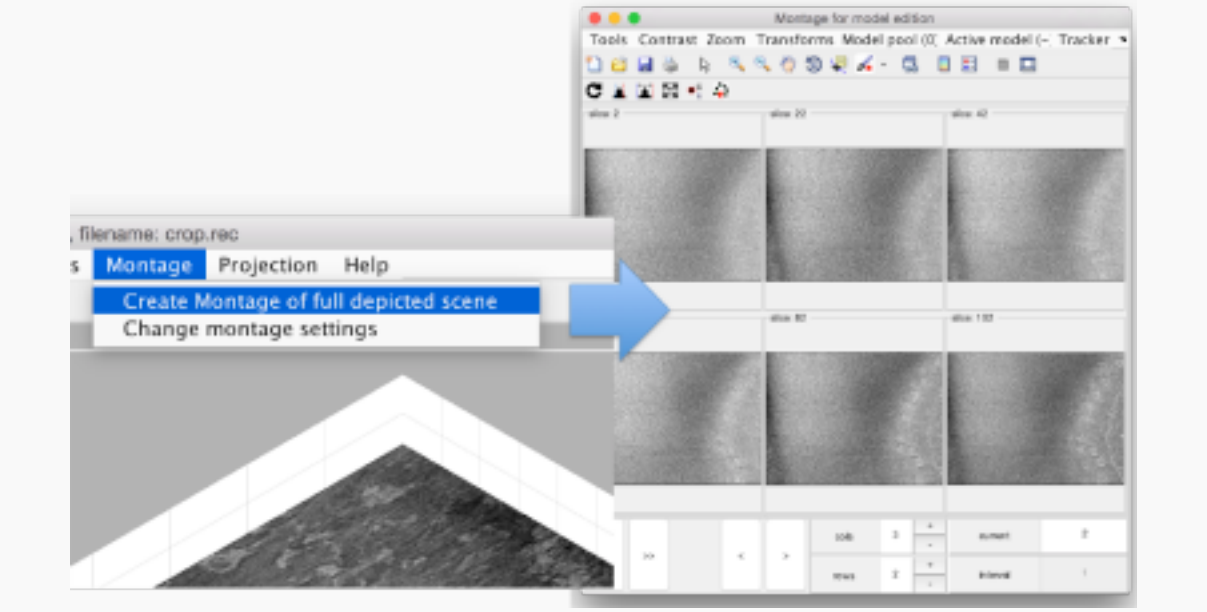
Pick membrane points

We open our tomogram in `dtmslice`

```
\dtmslice @{fhv}1 -prebin 1
```

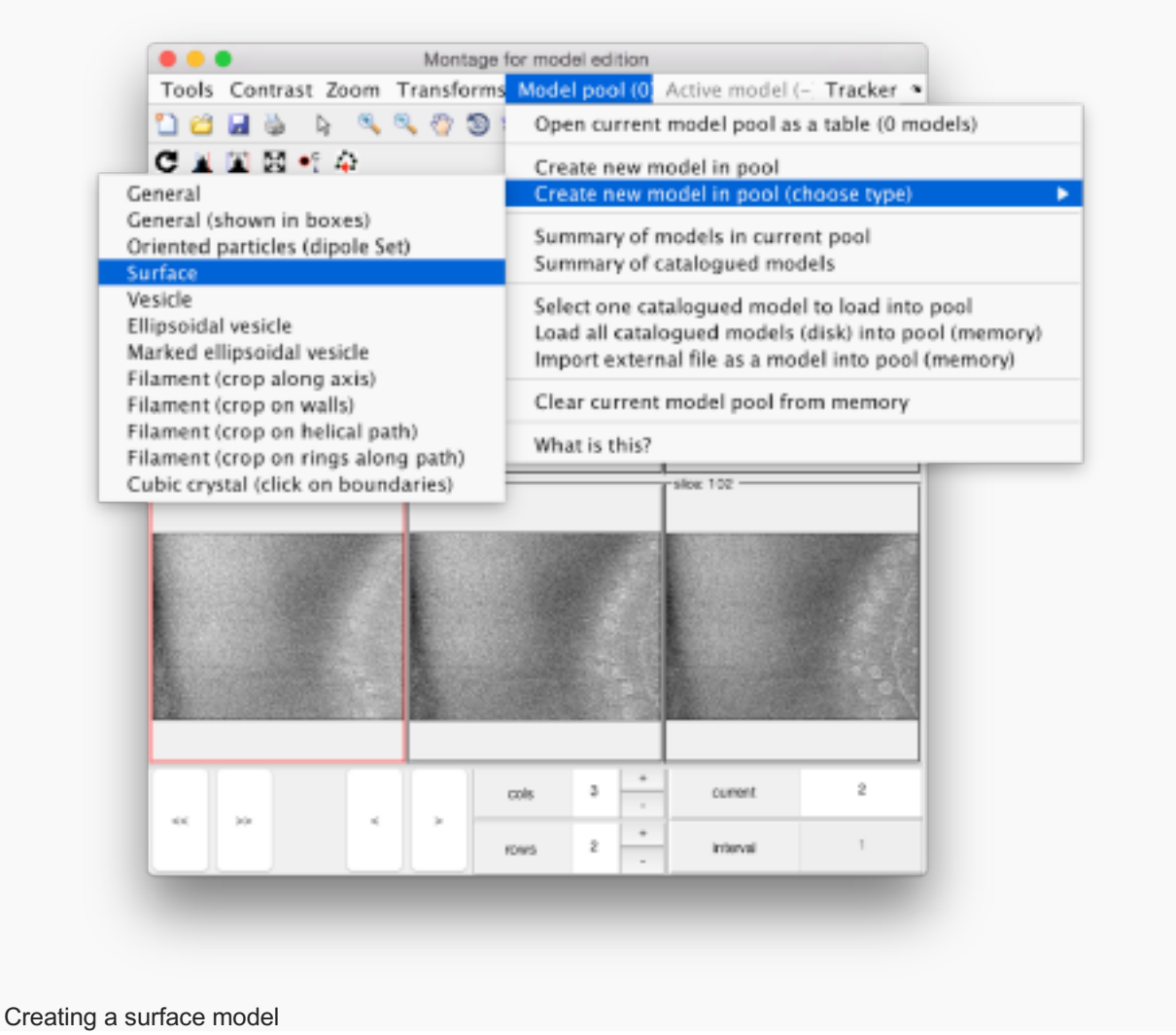
In this variant of the syntax of `dtmslice`, the string `@{fhv}1` just means "tomogram number 1 inside catalogue `fhv`. On the opened scene, we will use a montage to manually click on a set of membrane points.

It asks whether to keep or delete from memory the model pool: delete it.



Opening a montage view

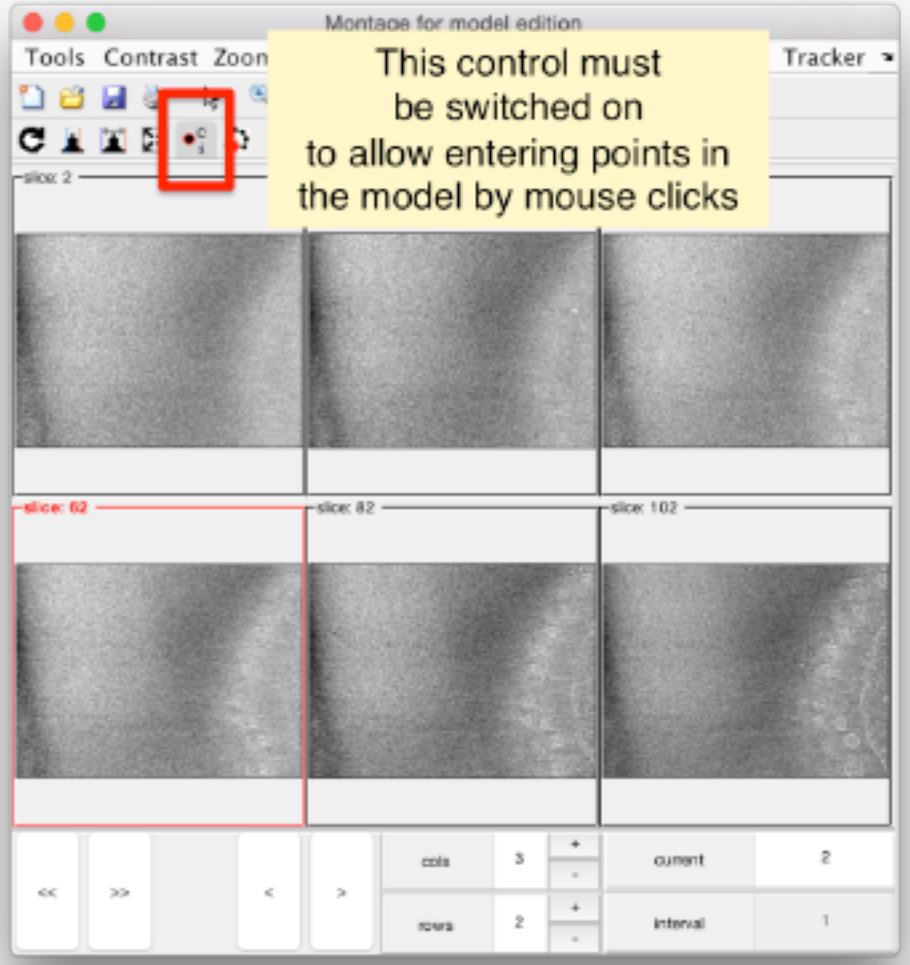
By default, this `montage` represents slices taken orthogonally to *z* every 20 pixels of the unbinned tomogram. This can be changed in the settings *before* opening the montage GUI. Now, we can create a model of type `Surface`, which we will use to store a set of points in the membrane to be picked manually (or semiautomatically). To do this, go to *Model pool> Create new model in pool (choose type) > surface*



Creating a surface model

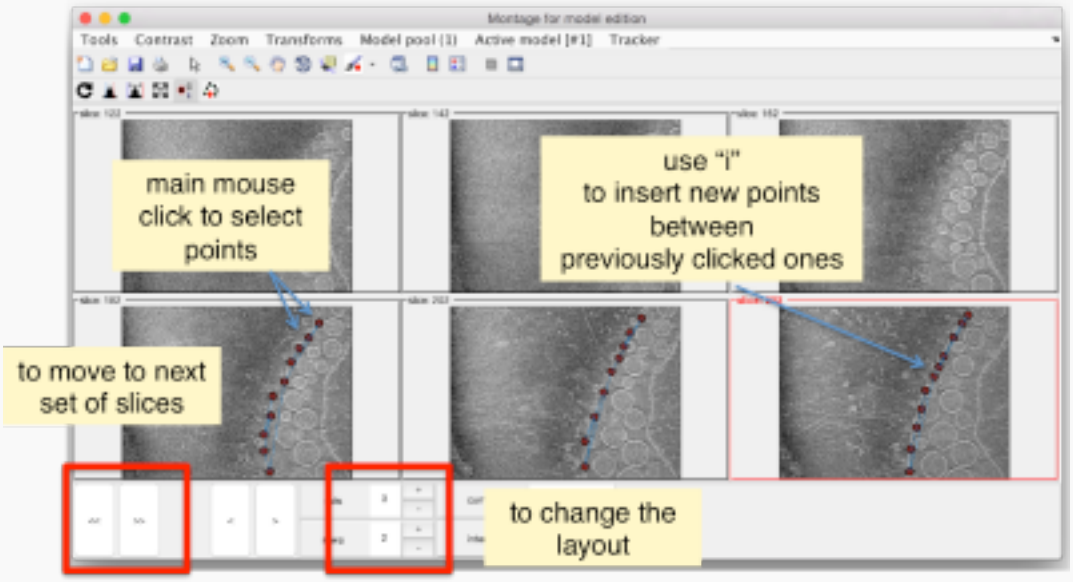
You need to click on the point switcher (the control in the toolbar with a *c* and an *i*) to allow entering points. The basic controls are:

- Mouse click to enter a point.
- `d` to delete a point.
- `i` to insert a point.



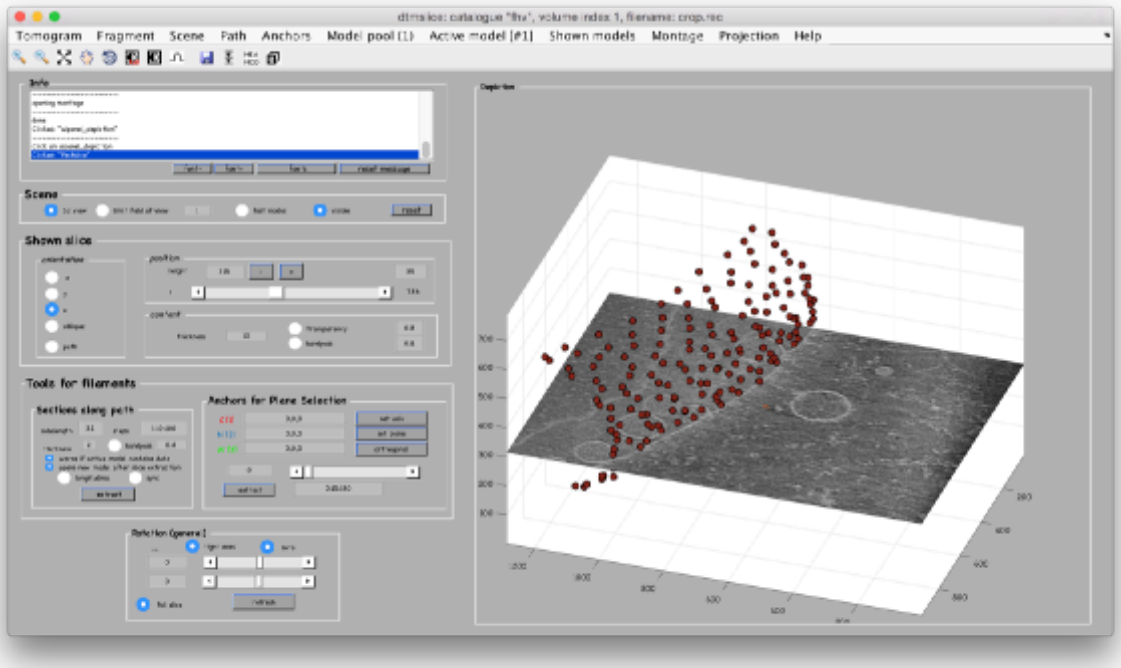
Point switcher must be toggle on to pick points into the model

Use the >> button in the bottom of the GUI to go to next set of orthoslices. You do not need to pick every slice, but every 4-5 or so.



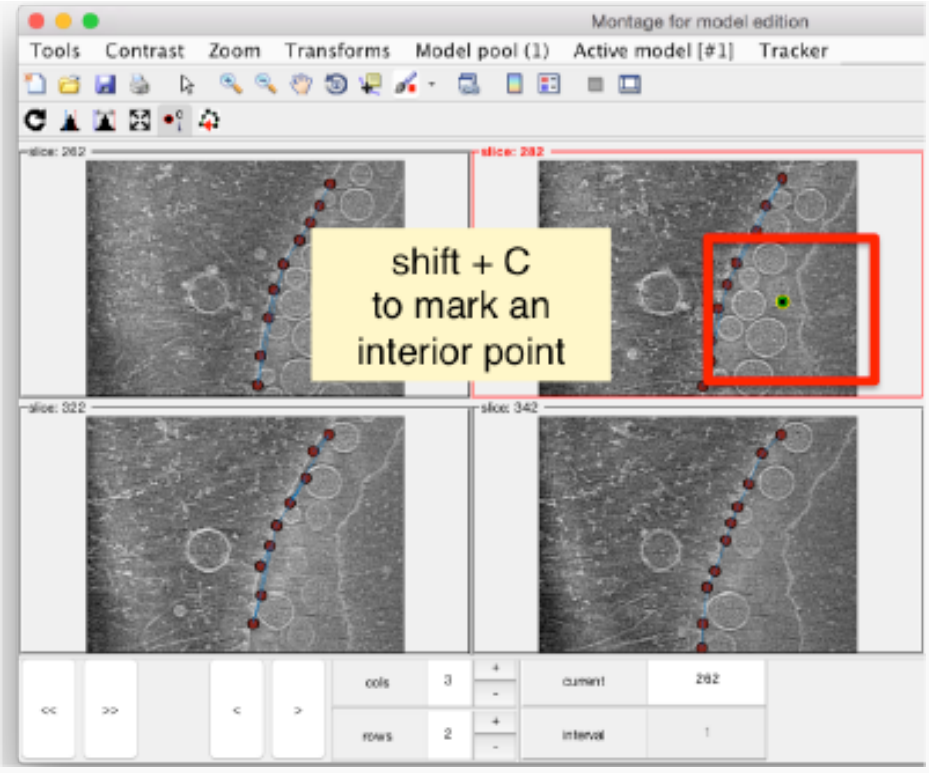
Controls of the montage GUI

Note that the points that you click in the Montage GUI are assigned to a model in the pool, and as such, they appear automatically in the dtmslice GUI



Points clicked in the montage GUI reflect in the `tmslice` GUI

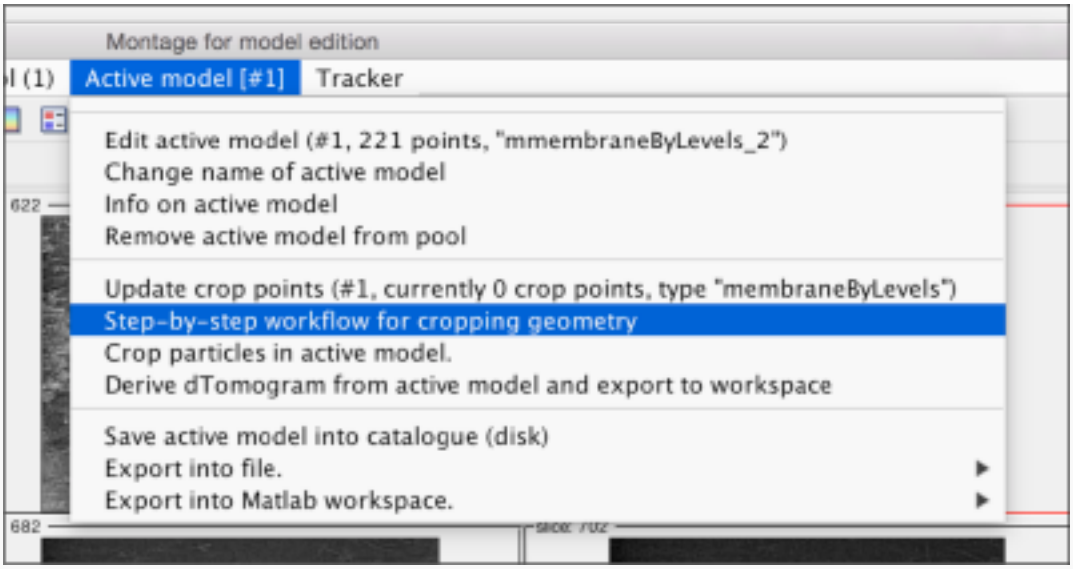
You also need `shift + C` in order to place a center inside the mithocondrion. This point is just used to tell *Dynamo* what is the inside and what is the outside face of the surface that we are building.



`shift + C` marks an arbitrary interior point

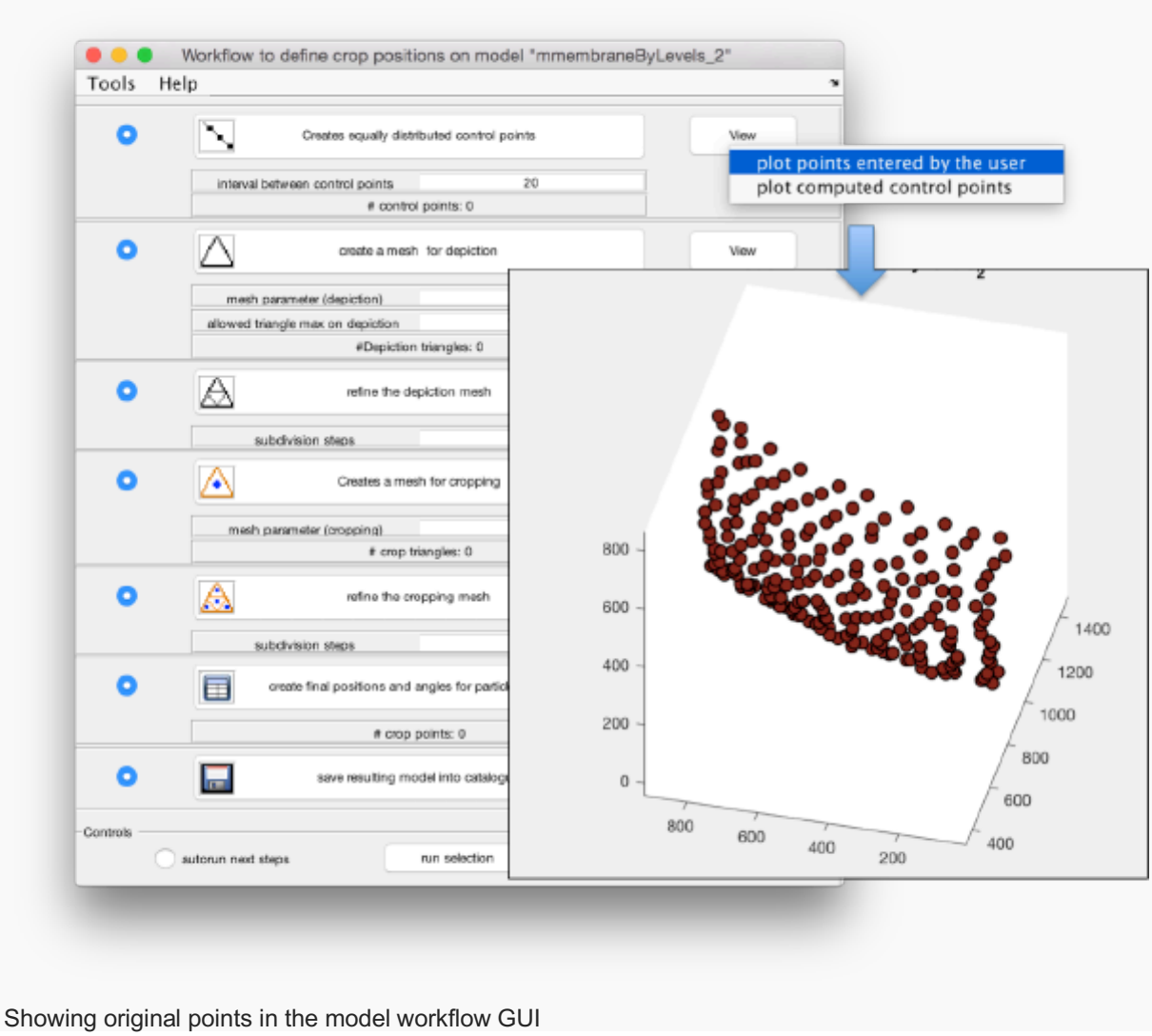
Create a surface

Now, we want to convert the points that we have introduced into a triangulation. This is a part of the workflow used to crop particles from membrane models, so we open the workflow GUI for this model in the *Active Model*menu tab.



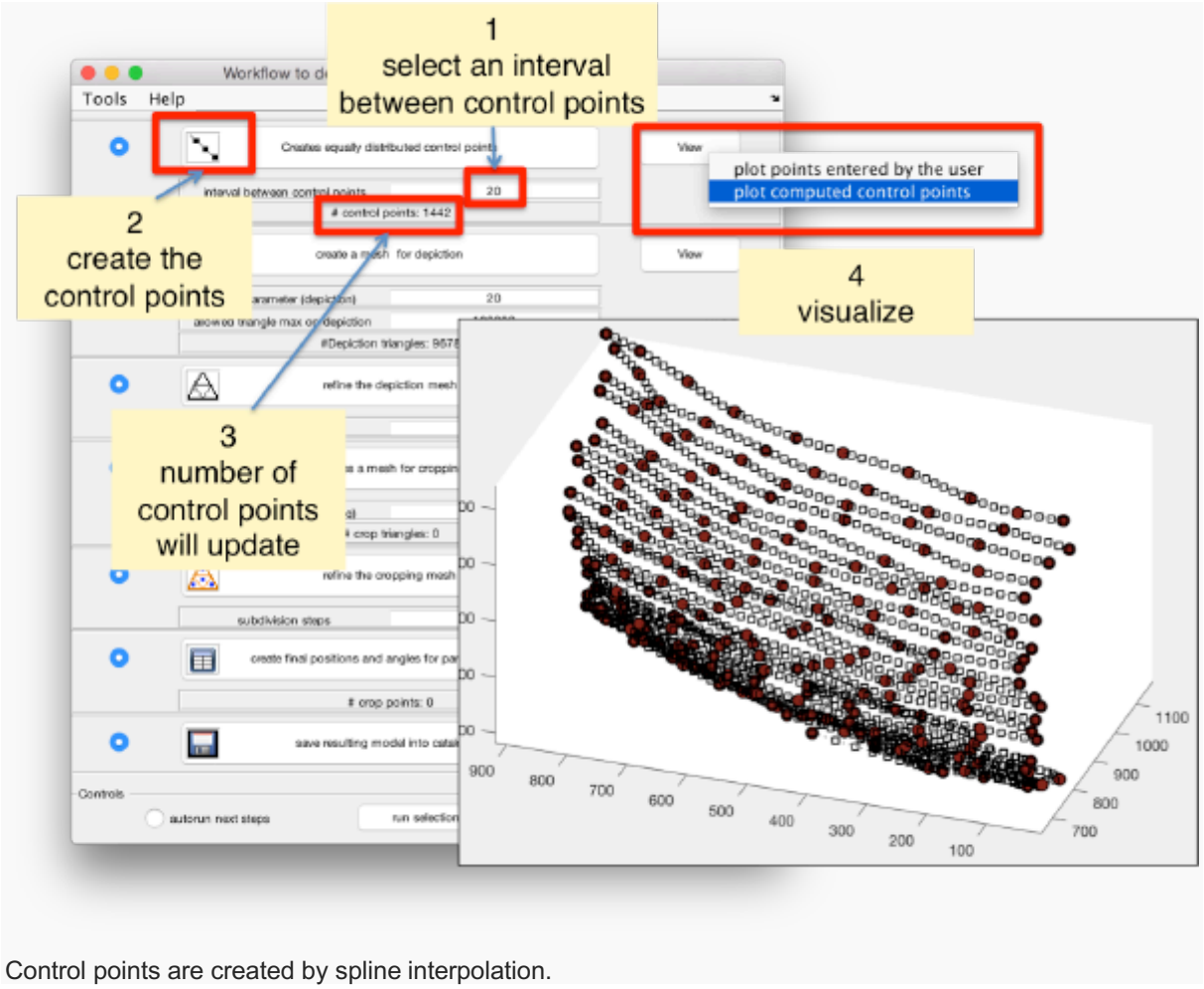
Invoking the workflow editor of active model

We can first check the points currently contained in the model, by right clicking on the first viewing option and choosing the *User points* option. They will be depicted in a graphic window which will update with the new graphical elements that we depict there.



Showing original points in the model workflow GUI

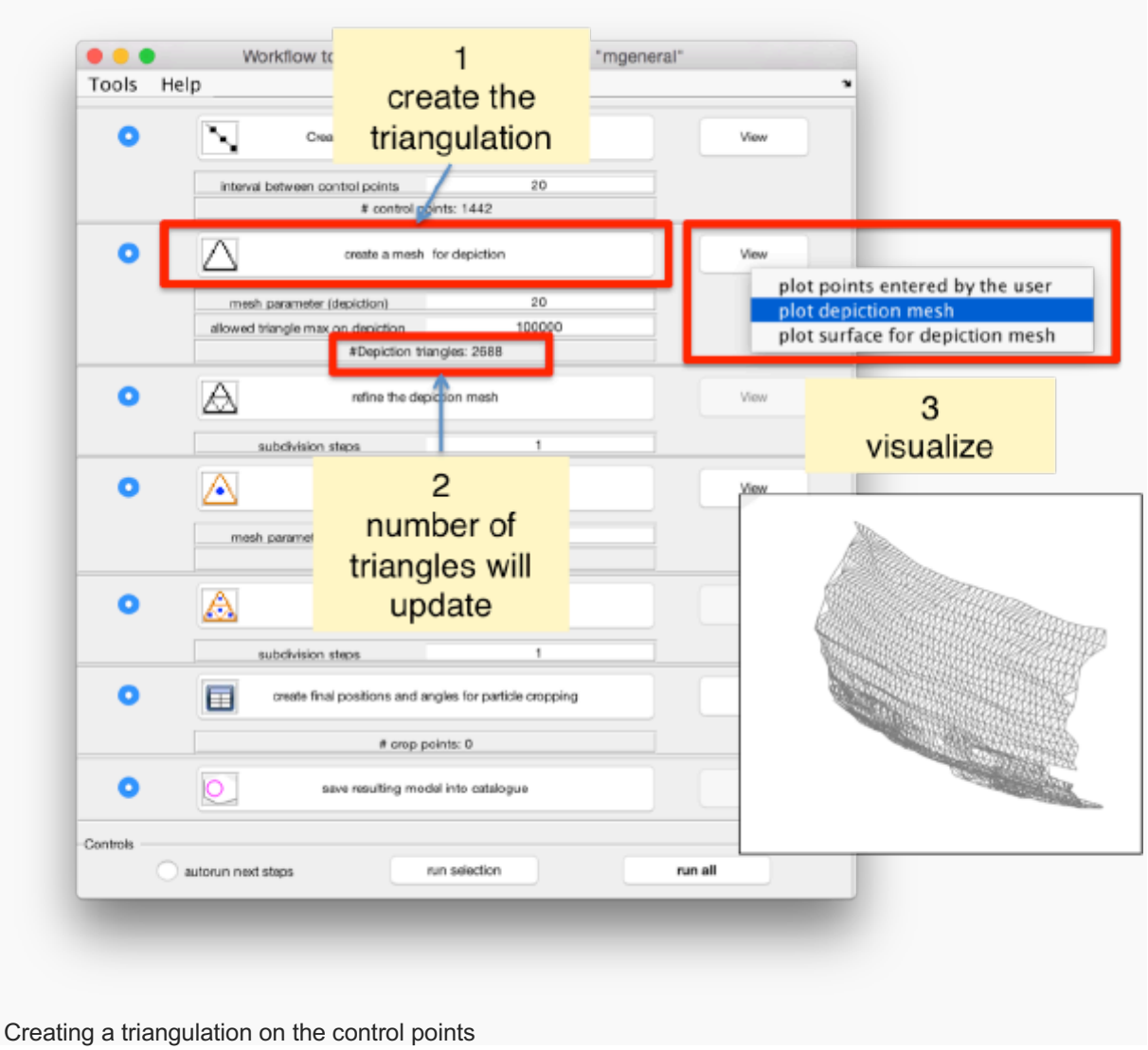
We first need to create a spline interpolation on each of the levels of our point cloud.



Control points are created by spline interpolation.

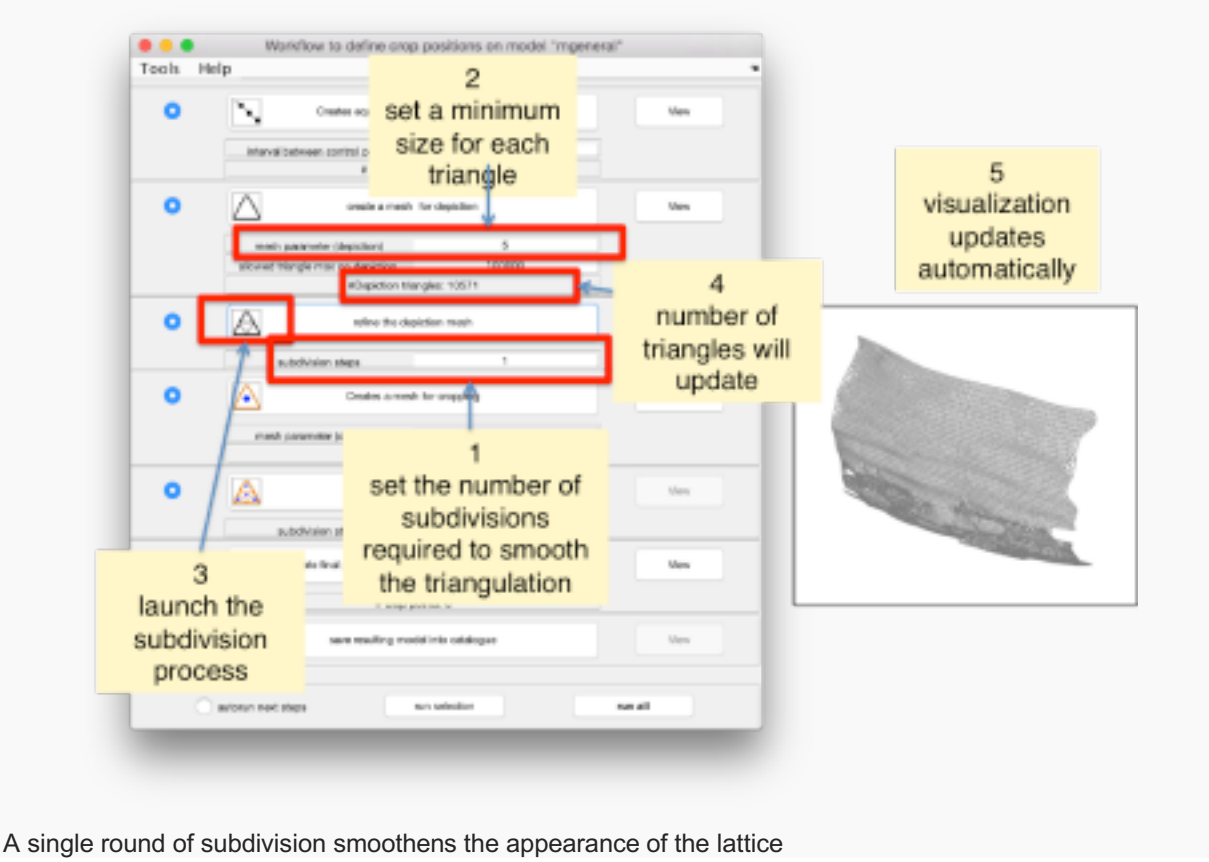
To rotate the view, right-click on the graph, select the yz plane view, and click the rotate3D button.

Now we define a triangulation on this control points.



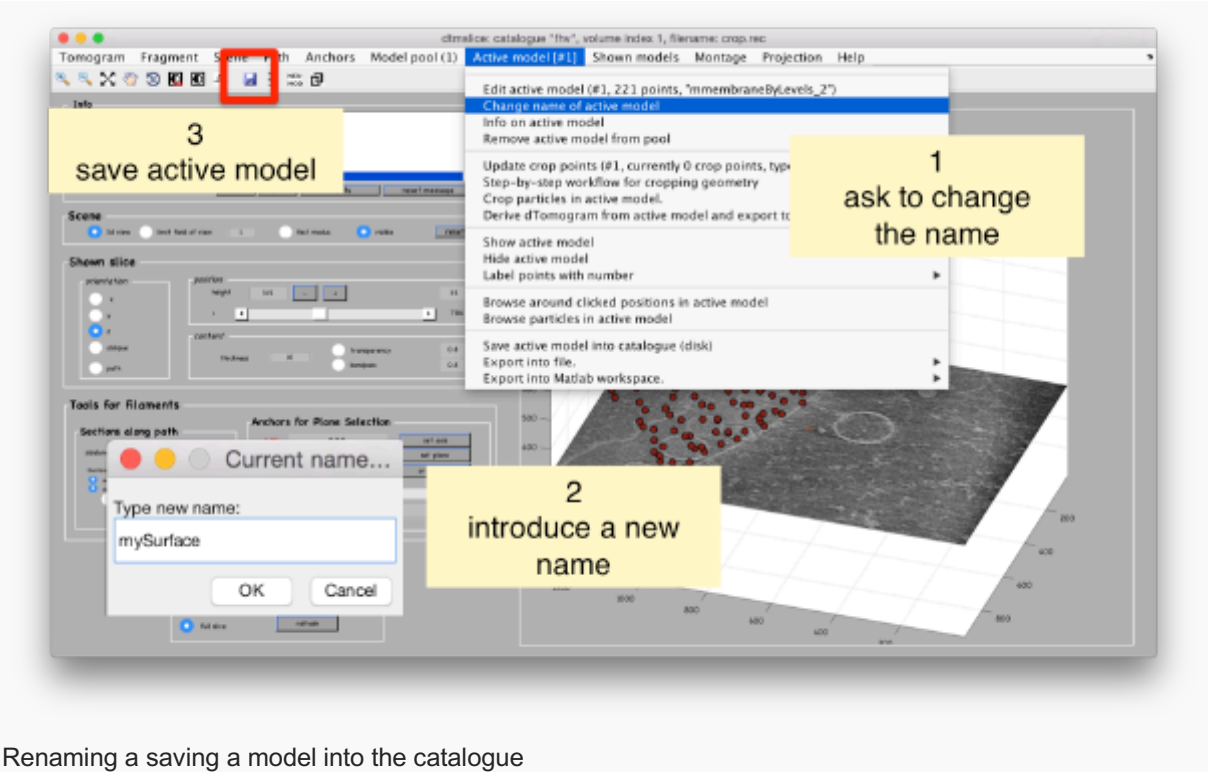
Creating a triangulation on the control points

Which can be subdivided to get a smoother appearance, and also a more accurate representation of the geometry.



A single round of subdivision smoothens the appearance of the lattice

Then, we can change the name of the model for clarity, and then save it into the catalogue (with the disk icon in `tmslice` or the *Save into catalogue* options under the *Active model* menu). Use mySurface as your new name (do not use 'membrane').



Renaming a saving a model into the catalogue

Use a surface to impart orientation

We bring the model into our workspace. We search for it in the catalogue:

```
dcmodels fhv -nc mySurface -ws output
```

which will prompt the response

```
Volume 1. Matching models: 1 (total: 1)
/i/embo2019/u/emboX/Prac-4/fhv/tomograms/volume_1/models/mySurface.omd
```

now, we can read the model file into our memory space to operate with it:

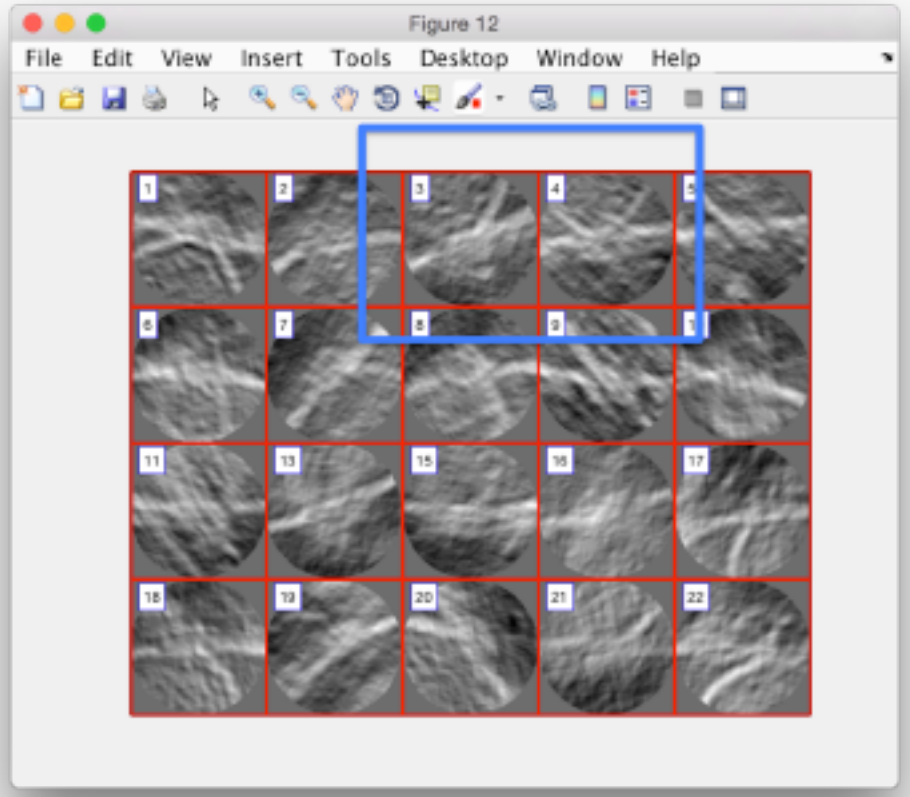
```
\m = dread(output.files{1});
```

Now the model `m` is in our space of memory and can be used to impart an orientation to the points in our table.

```
tOrientedBySurface = dpktbl.triangulation.fillTable(m,'raw.tbl');
```

In this table, orientations are orthogonal to the membrane (or more precisely, orthogonal to the closest triangle in the mesh that represents the membrane)

```
 figure;
dslices('particlesData','t', tOrientedBySurface,'projy','c20','align',1,'labels','on');
```

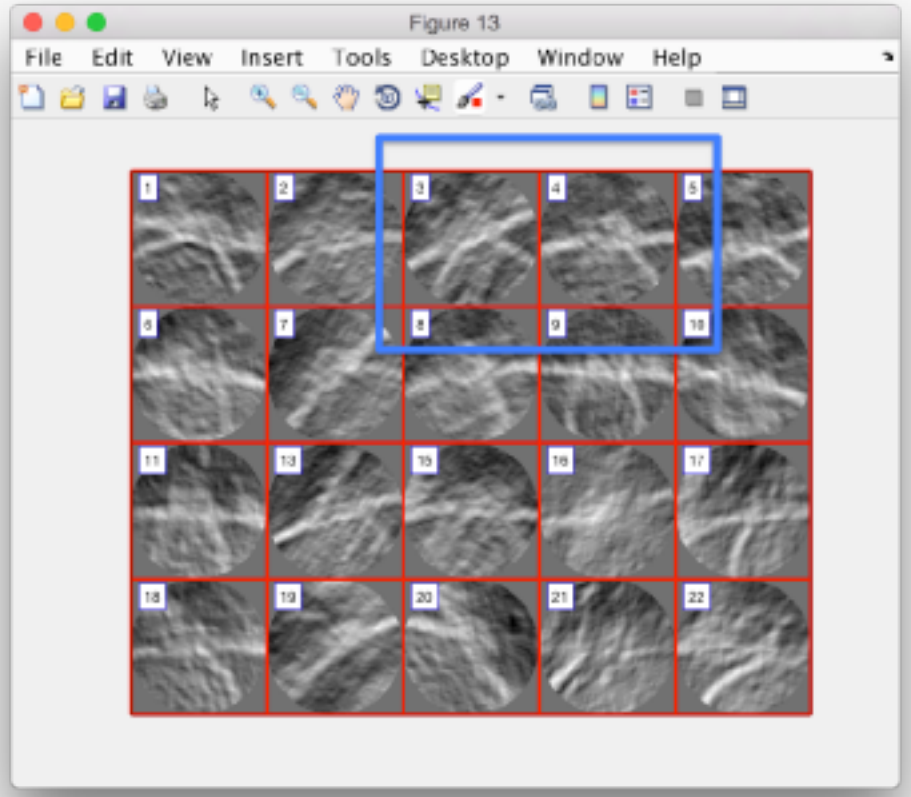


Slices along the xz plane as seen by the table

```
tConsistent =
dynamo_table_flip_normals(tOrientedBySurface,'center',m.center)
```

We can check the effect of this table on the particles through:

```
figure;dslices('particlesData','t',
tConsistent,'projy','c20','align',1,'labels','on');
```

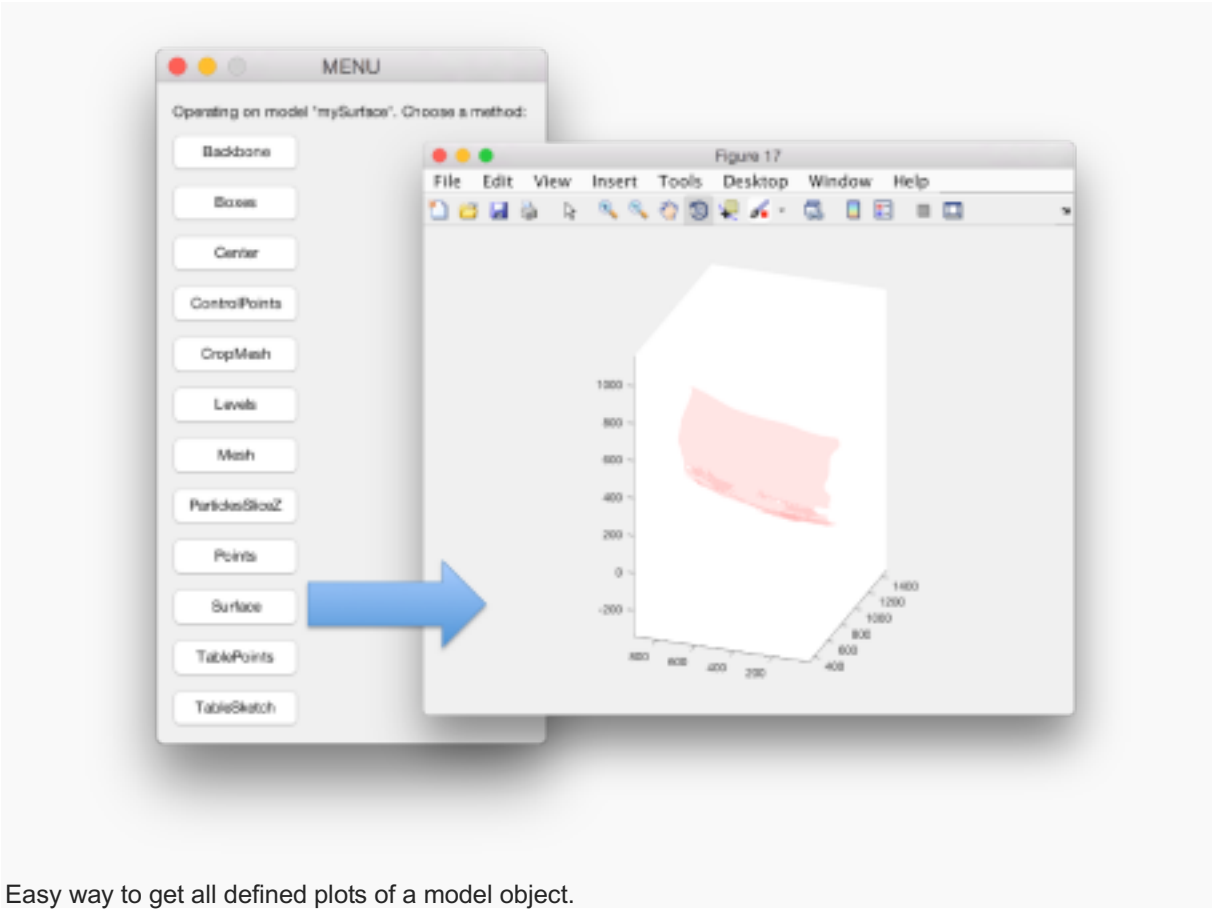


Slices along the xz plane as seen by the table, after using the center point to establish the interior side of the surface

To get a better graphical impression, we can depict our surface (contained in the model `m`) with:

```
m.ezplot
```

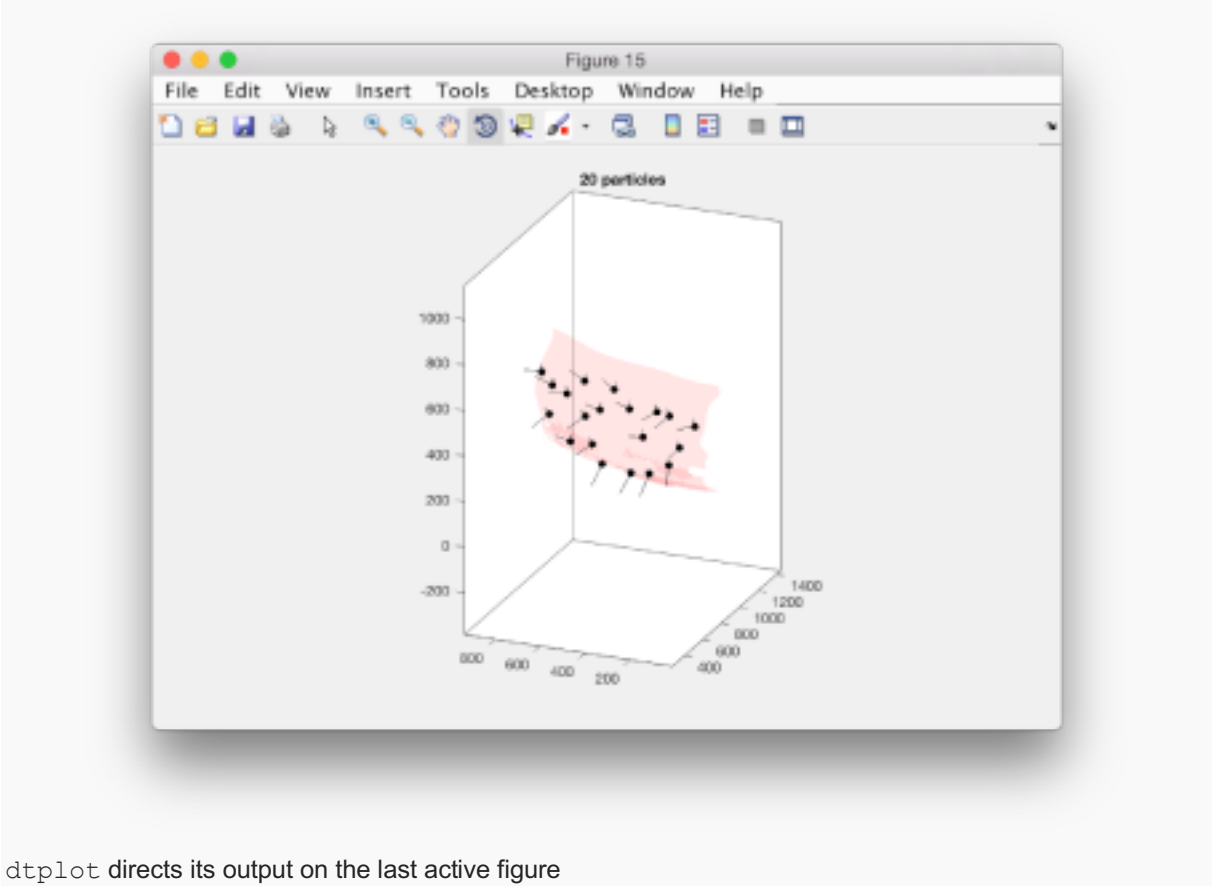
and choose the `Surface`option



Easy way to get all defined plots of a model object.

Then, we just plot the positions of the particles on the same graphical figure:

```
dtplot(tConsistent,'m','sketch','sketch_length',100,'sm',30);
```



`dtplot` directs its output on the last active figure

Management of the missing wedge

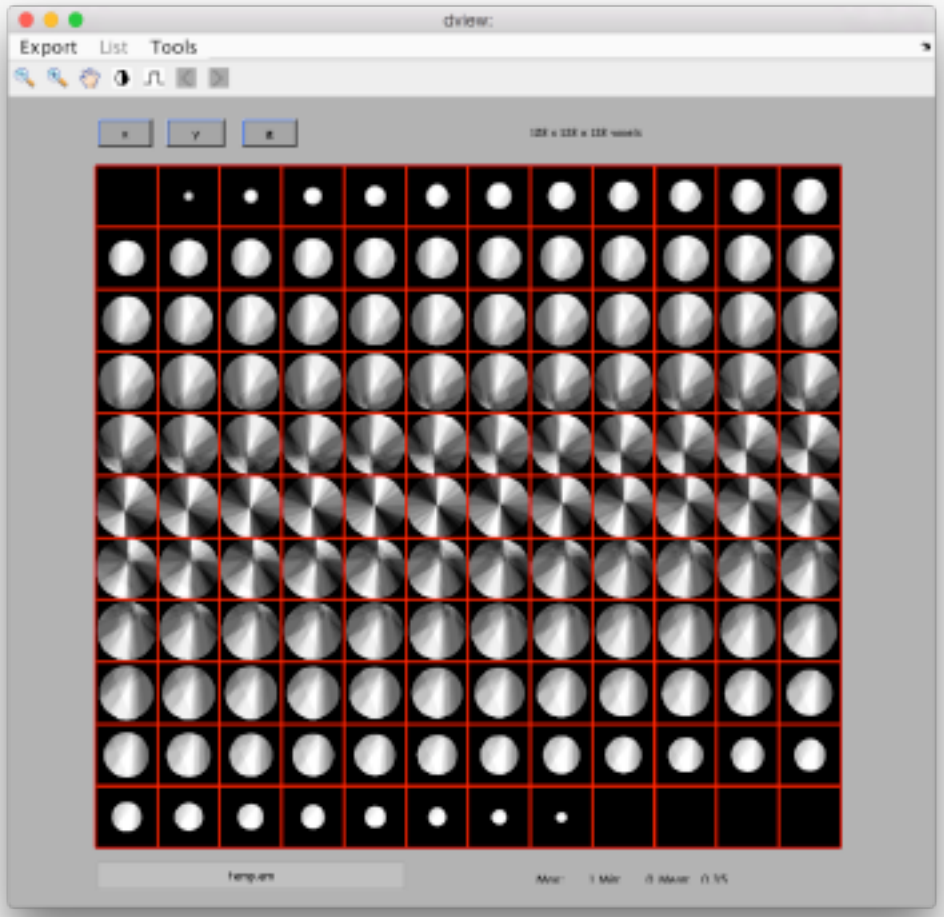
When we create a table considering only the orientations of the points with relation to a membrane, there is no particular preference for rotations of the particles about the normal direction (i.e., azimuthal directions). The `narot` angle in this table is initialized to zero

Thus, take into account that an average created on these particles will have a strong missing wedge, especially in our case, where we have a preferential direction. This can be checked by averaging the particles against this table:

```
oa=daverage('particlesData','t', tConsistent,'fc',1);
```

Here, we ask (`'fc',1`) to run a Fourier compensation step, so that we can check in the output `oa` the property `fweight`, which contains the number of times a Fourier component is represented in the average. Bright values mean that the Fourier component is present in many particles (yielding thus an average of better quality for the involved frequencies).

```
dview(oa.fweight);
```



Fourier component presence without azimuthal randomization

We can atenuate this effect by randomizing the rotational angle:

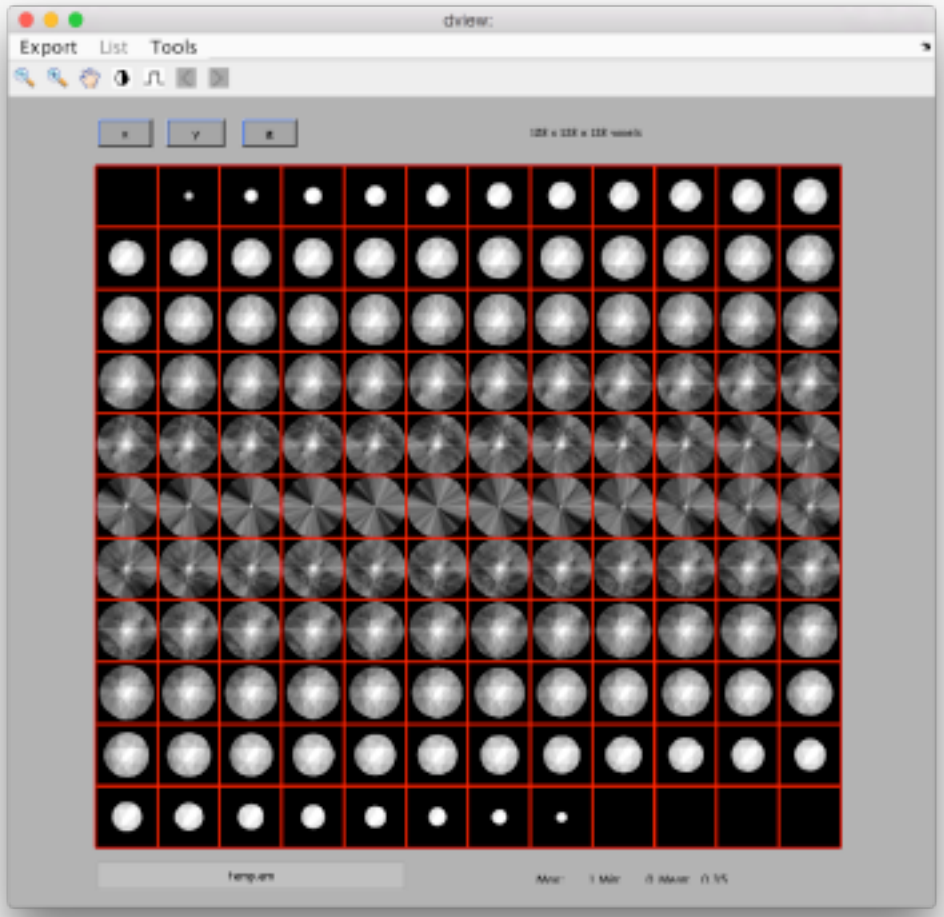
```
tConsistentRandomized = dynamo_table_randomize_azimuth( tConsistent);
```

and averaging again.

```
oaRandomized=daverage('particlesData','t', tConsistentRandomized,'fc',1);
```

Now, the `fweight` map doesn't show such a strongly preferential orientation:

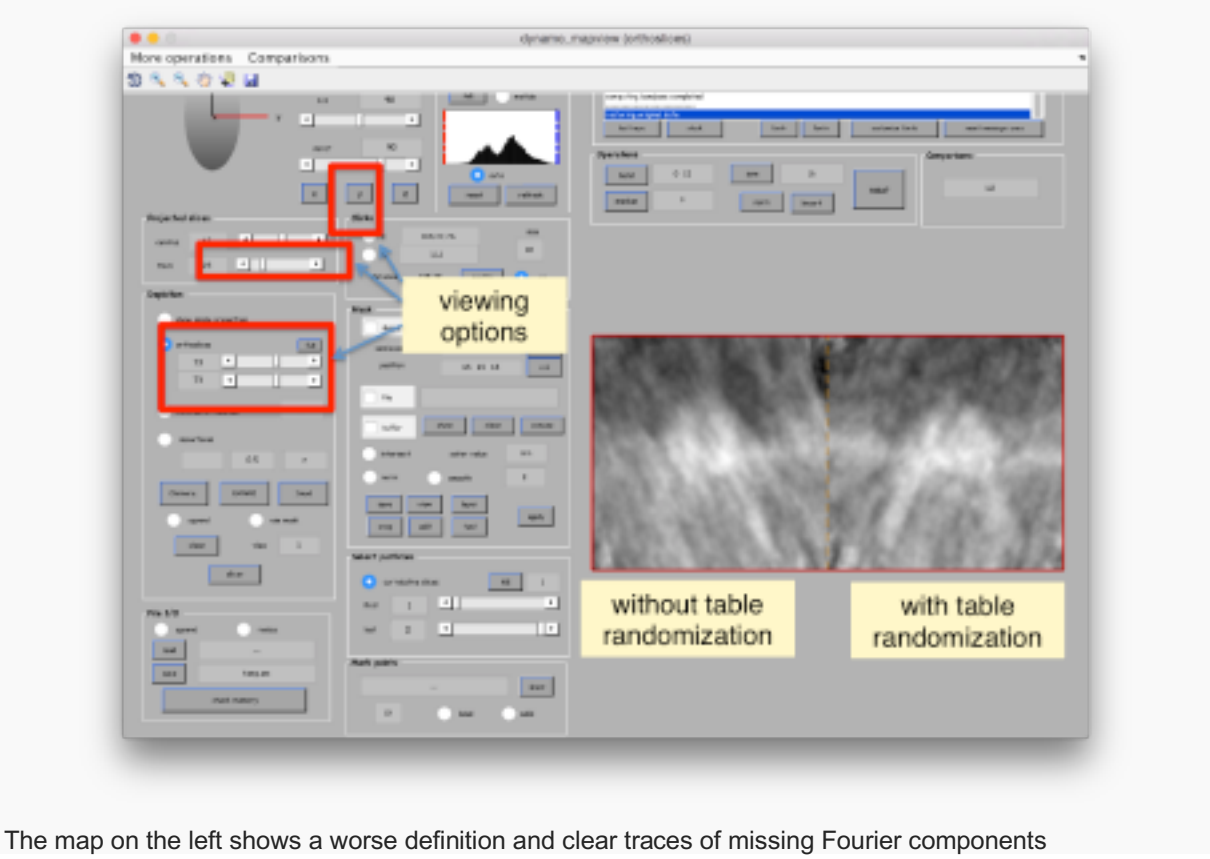
```
dview(oaRandomized.fweight);
```



Fourier component presence with azimuthal randomization

The effect in direct space can be shown by depicting both averages (with and without randomization side to side)

```
dmapview({oa.average,oaRandomized.average});
```



The map on the left shows a worse definition and clear traces of missing Fourier components

Project for rotational alignment

We have now a template and a coherently defined metadata (i.e. table) that align the particles coarsely along the right direction. We write them into disk:

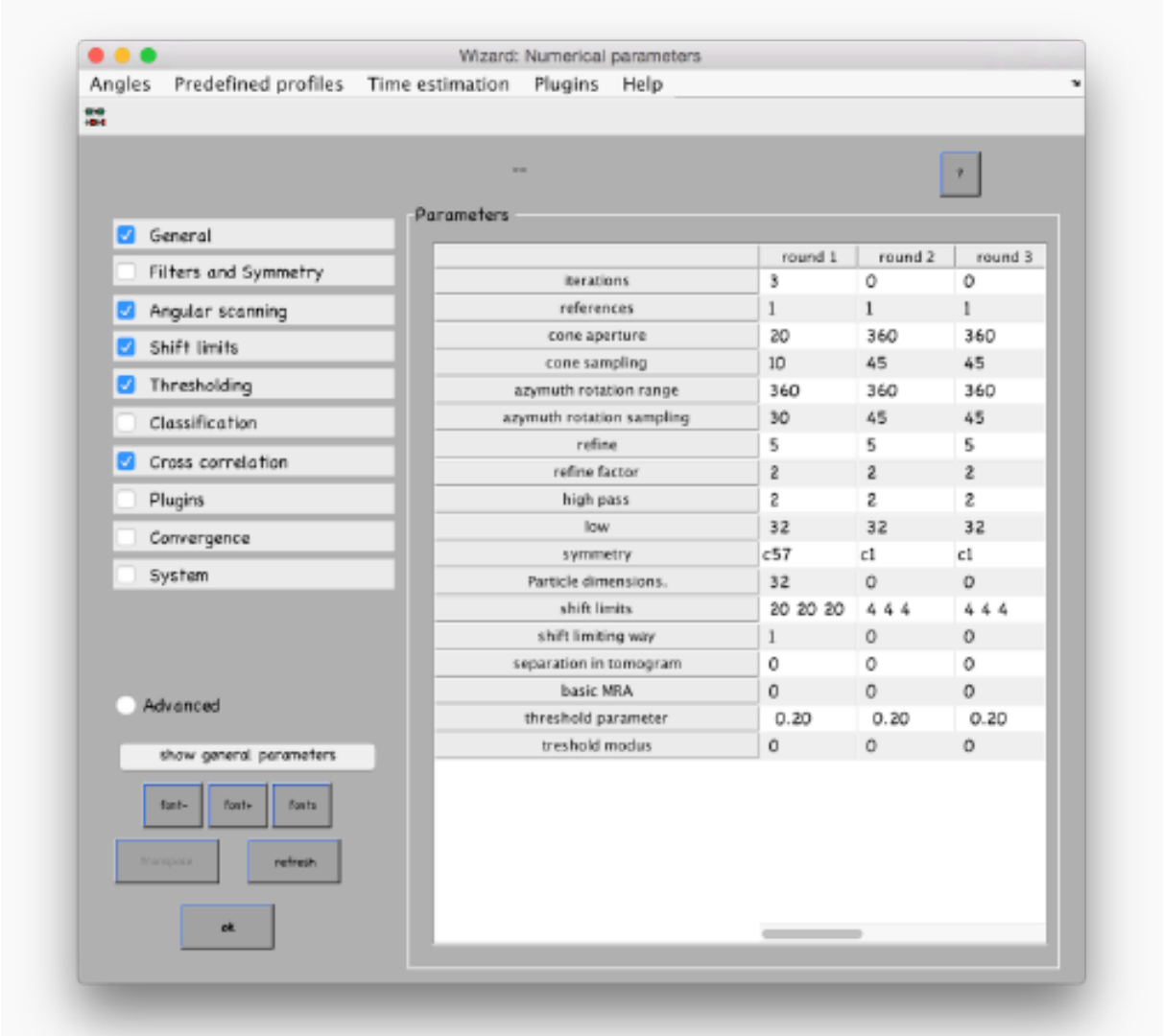
```
dwrite(tConsistentRandomized,'zOriented.tbl');
dwrite(oaRandomized.average,'zOriented.em');
```

and create a project with these files. Note that the data folder remains unchanged.

```
dcp.new('zOriented','d','particlesData','template','zOriented.em','masks','
default','t','zOriented.tbl');
```

In this project, we will ask for a symmetrization 'c57'.. this is just to simulate fully rotational symmetry. We are not assuming that this symmetry is physical: we just want to force any possible symmetry axis in the data along the *z* axis. We also keep binning twice the particles, in order to get the computation times short.

On the numerical parameters tab edit as shown in the picture below:



Alignment parameters in `zOriented`

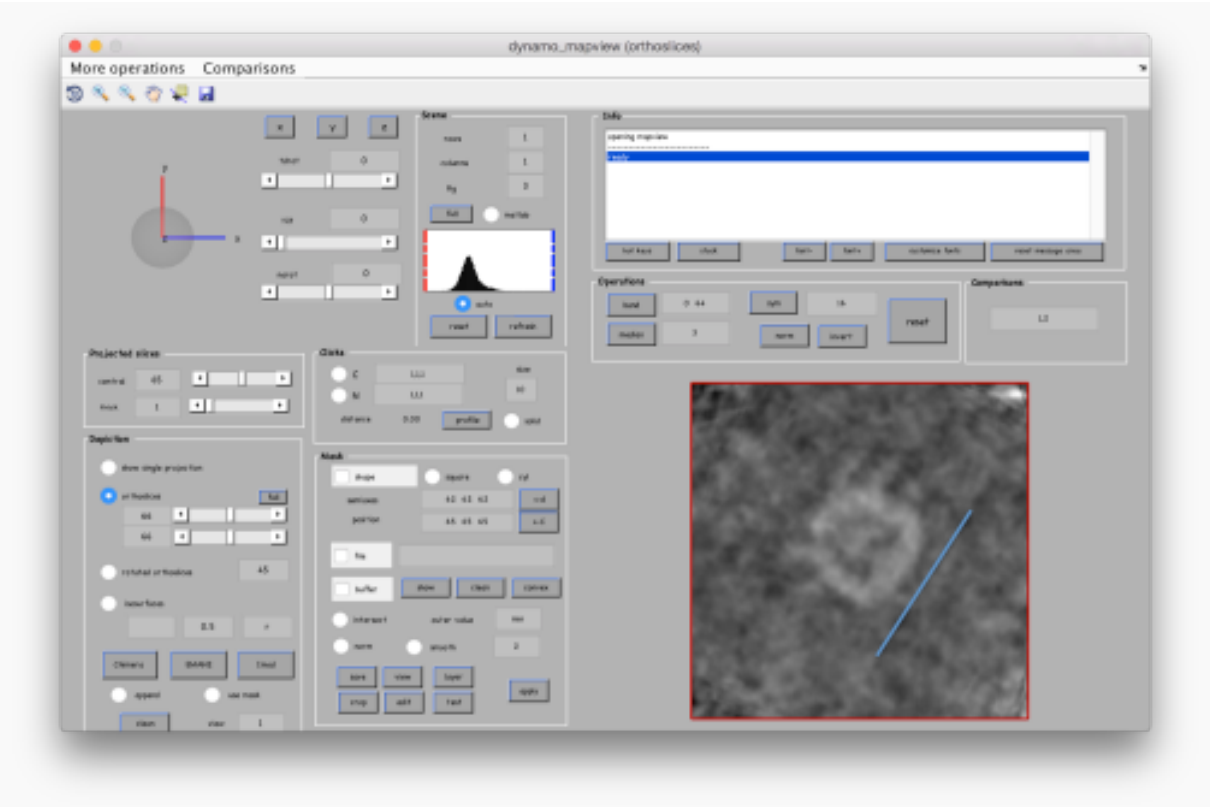
And on the computing environment:

Check, unfold, and run the project:

This will take a few minutes. You can check the progress by typing (on a different terminal window):

tail -f projectname/log.txt

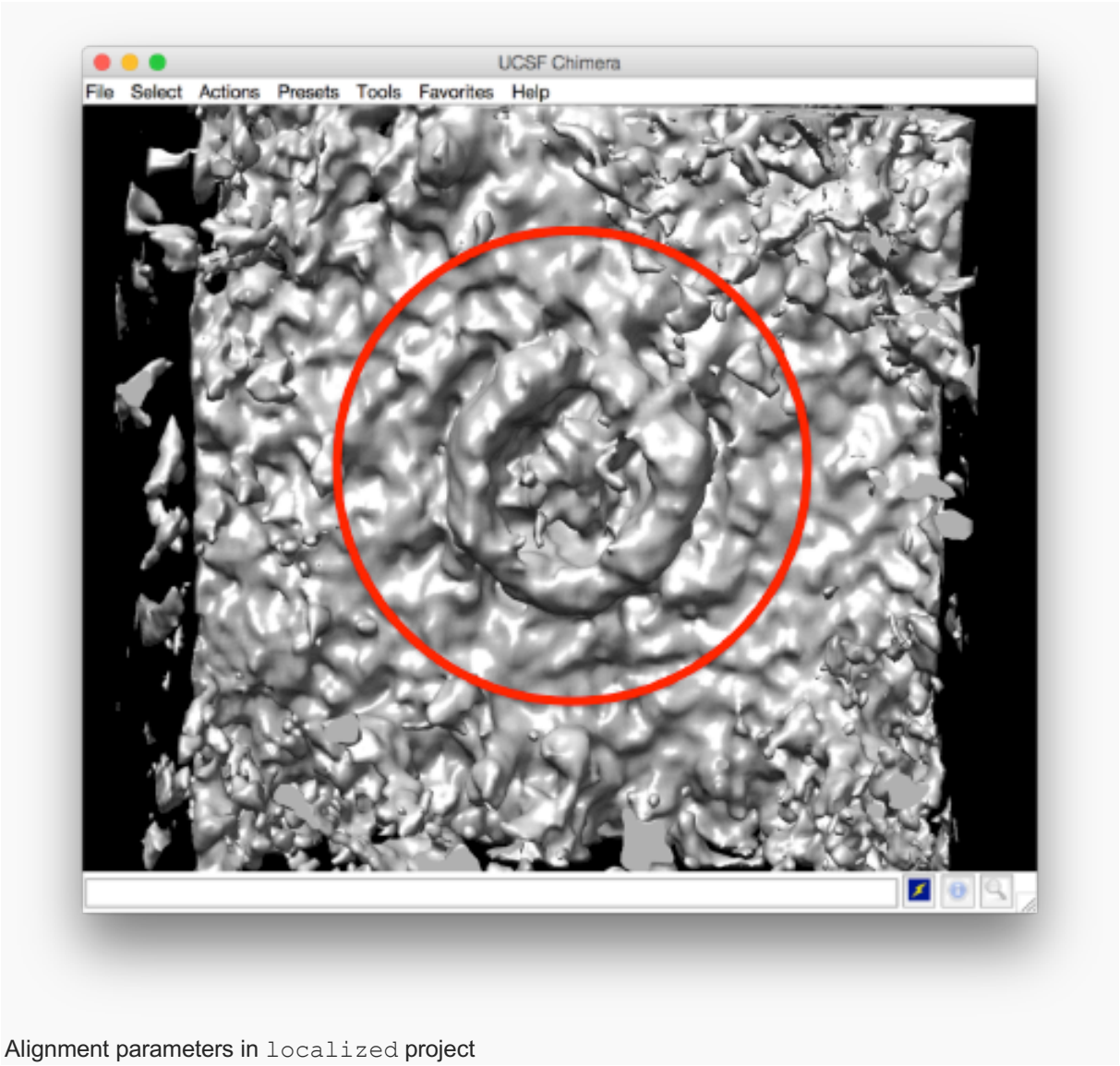
The result shown an incipient formation of the crown...



Result in `zOriented`

and barely identifialble in Chimera: open a new terminal, go to the folder zOriented/results/ite_0003/averages/ and type

chimera average_ref_001_ite_0003.em



Alignment parameters in `localized` project

Project for localized alignment

We don't see clearly what is happening on the area of interest... On one hand we know that we are using a very reduced number of particles (which on top of it are too similarly oriented). But there is still some room to improve in the approach: we can focus the refinement in the area of interest.

Creating a tight mask

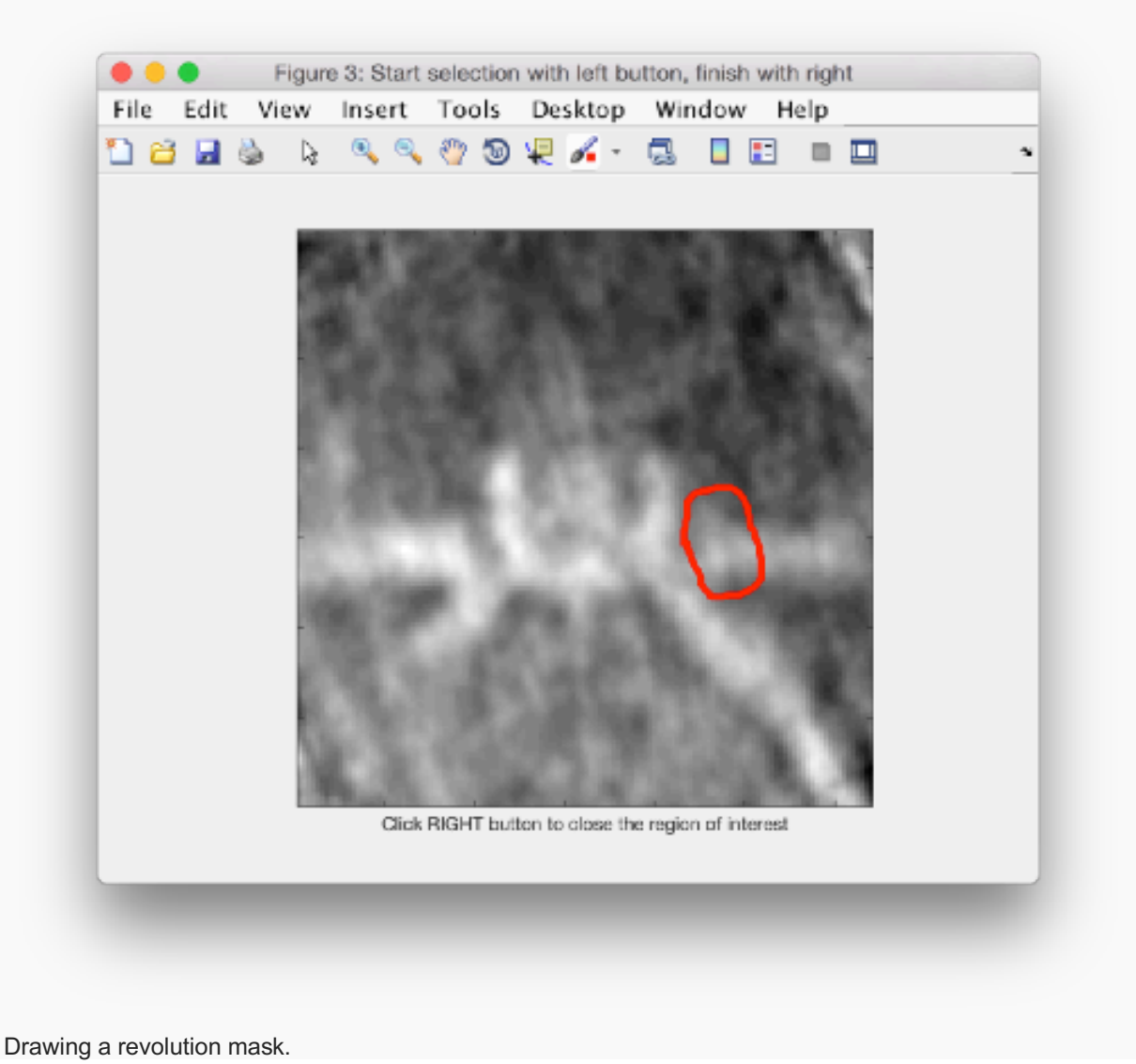
We want to create a mask that encloses the area of interest. To this end, we open the average in `dmapview`, by transferring it from `dview` (*Export*) or just asking `ddb` to pass the average directly to `mapview`

```
ddb zOriented:a:ite=3 -m
```

We tune the browser for viewing along *y*, and we select one of the central slices.

Then, we select the area that we want to use as seed in this plane to create a revolution solid. Change orthoslices to 64/64. We press on `[shift] + [s]` on the selected slide, letting a new window pop up. In this window, you handdraw an area on the XZ plane (left click to start drawing,

right click to stop). *Dynamo* will rotate the region about the *z* axis, creating a revolution solid into the `file temp_drawn_revolution_mask.em`



Drawing a revolution mask.

We have to change the default name given by *Dynamo* to a relevant one:

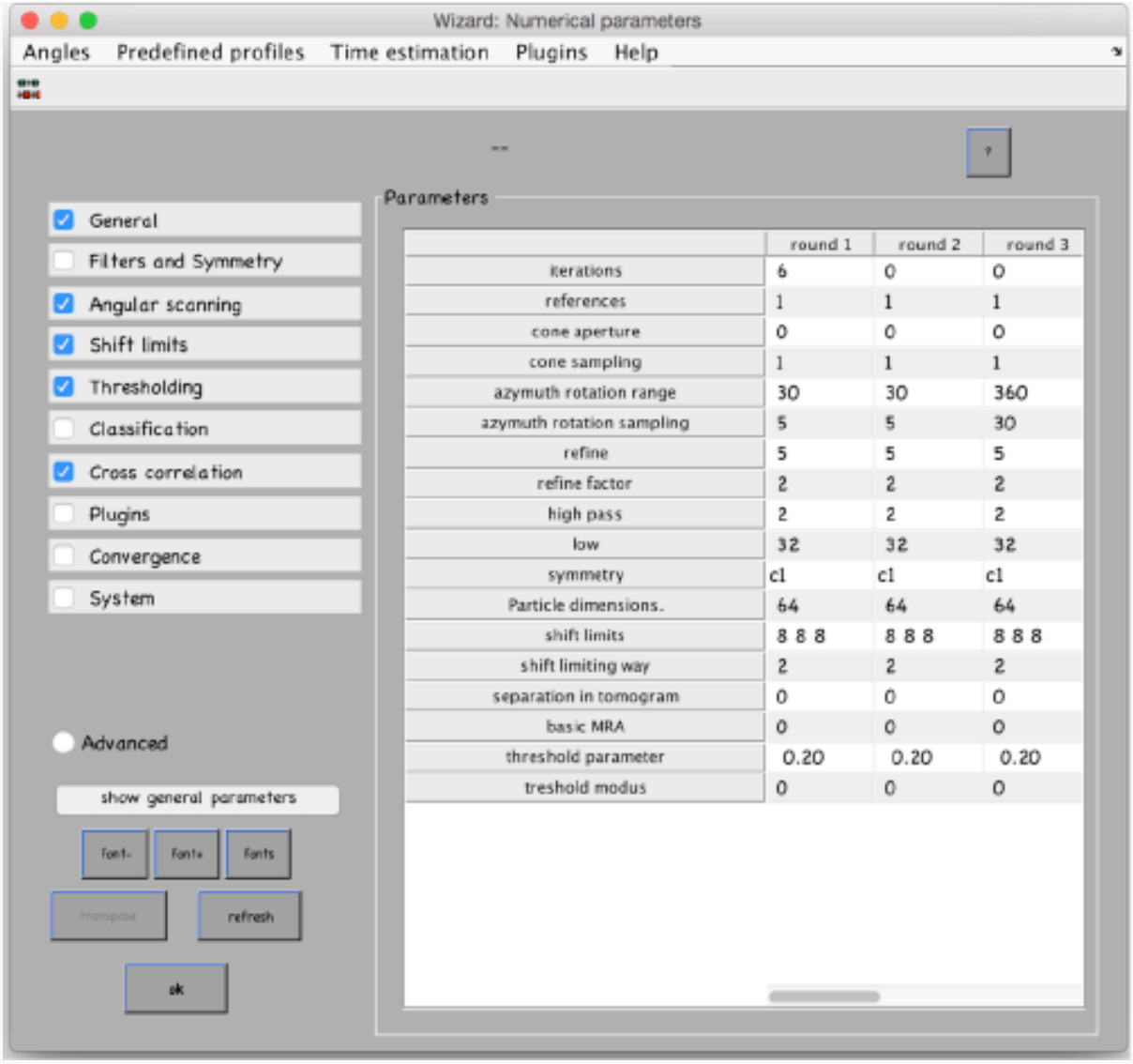
```
!mv temp_drawn_revolution_mask.em teethMask.em
```

and now continue the previous project by transferring the results of `zOriented` to a fresh project (lets call it `localized`)

```
dynamo_vpr_branch zOriented localized -b 1 -noise 0 -ite 3

dcp localized
```

and insert the tight mask `teethMask.em` that we just created as alignment mask of the project. (click on masks and browse the alignment mask). Note that we do **not** want impose any symmetry in our parameters, and we want to reduce the binning (particle size = 64):



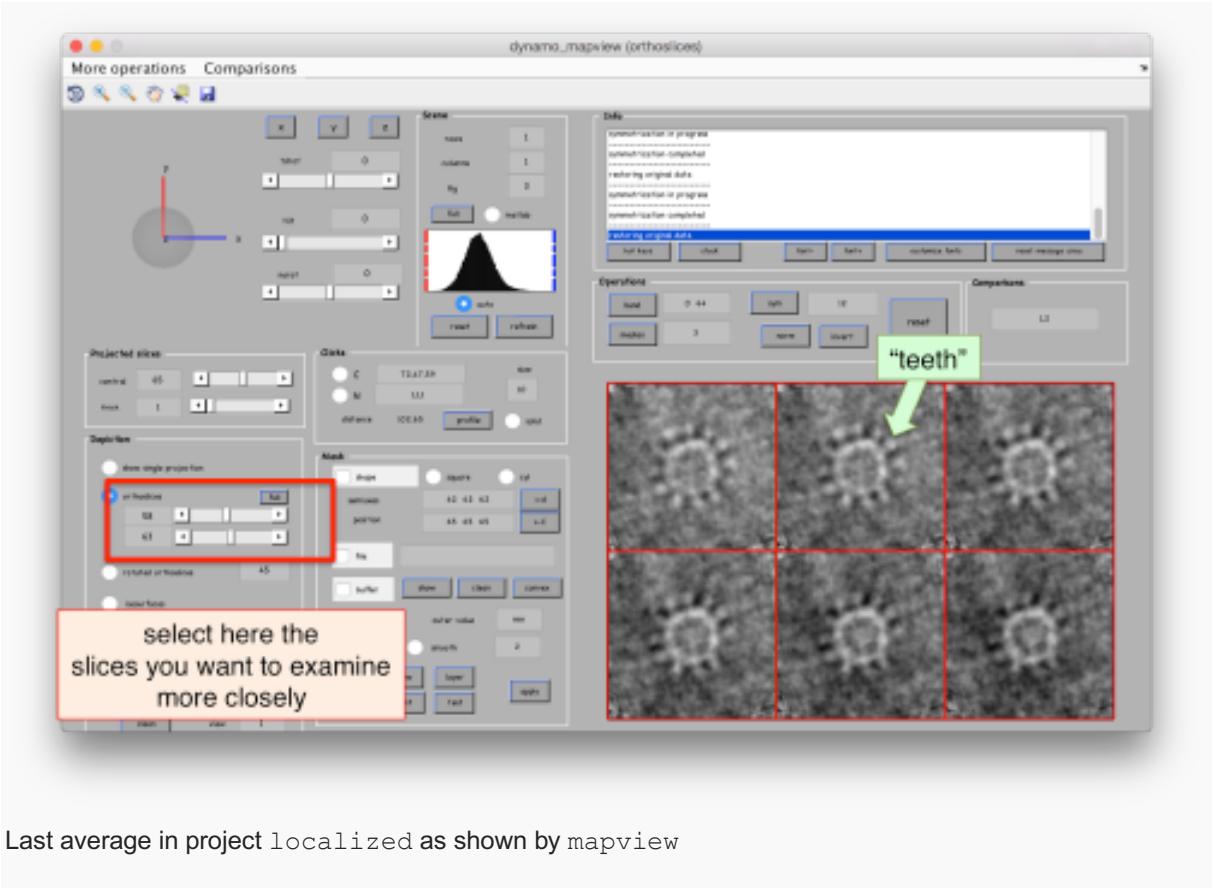
Alignment parameters in `localized` project

Let's inspect the output:

ddb localized:1 -m

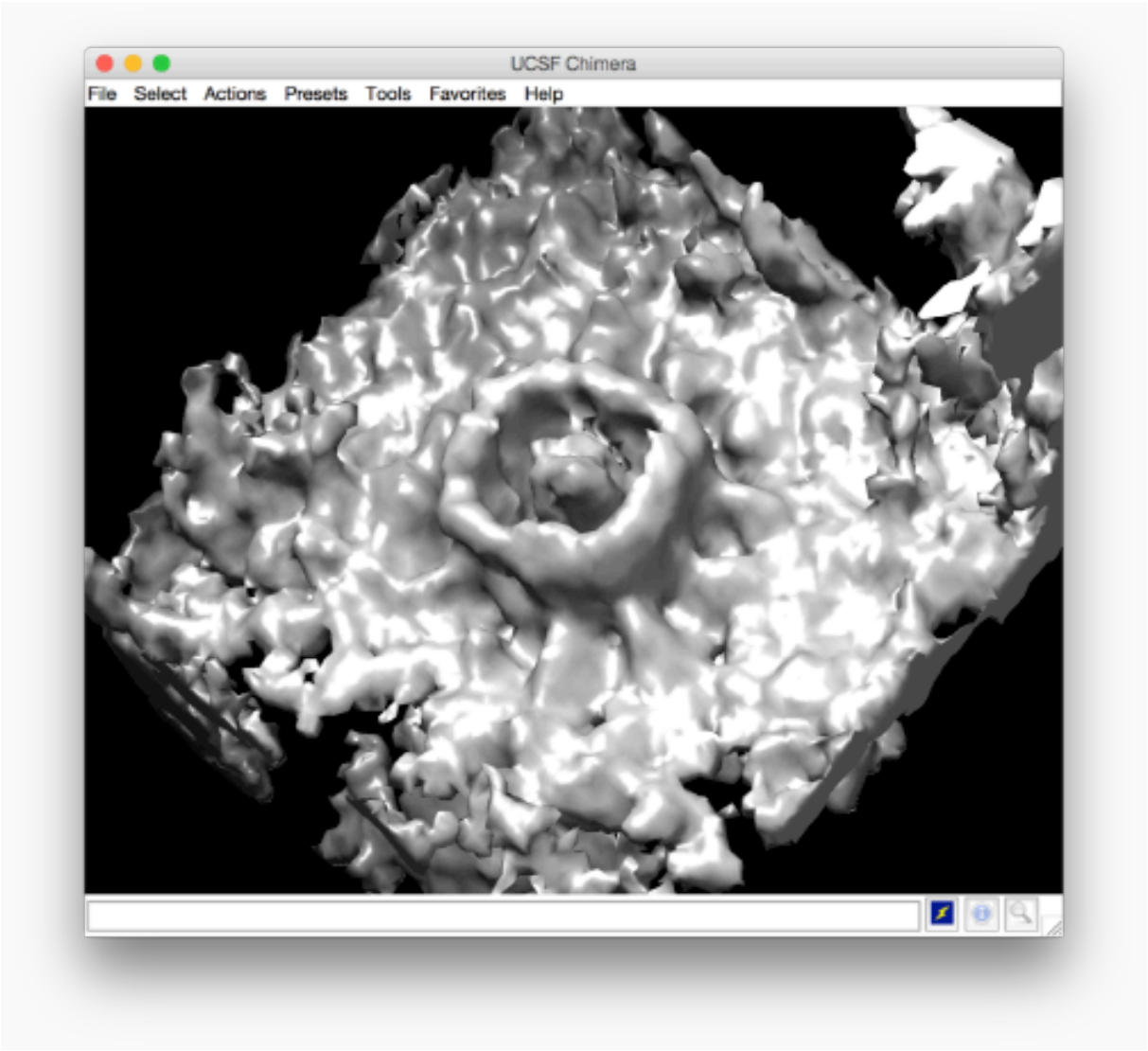
Last average in project `localized` as shown by `dview`

There are different things that we like in this average



Last average in project `localized` as shown by `mapview`

1. Last average in project `localized` as shown by `mapview` The localized masking worked well, producing a clearer insight into the region of relevance
2. Although we are using signal only inside the mask, the material outside of the mask does '*not* become smeared. This is clear mark that the alignment is real and non artifactual. That's an important point to check when we use small masks.
3. The symmetry in the area becomes apparent on the bare eye. They can be distinguished clearly with mapview and even in Chimera.



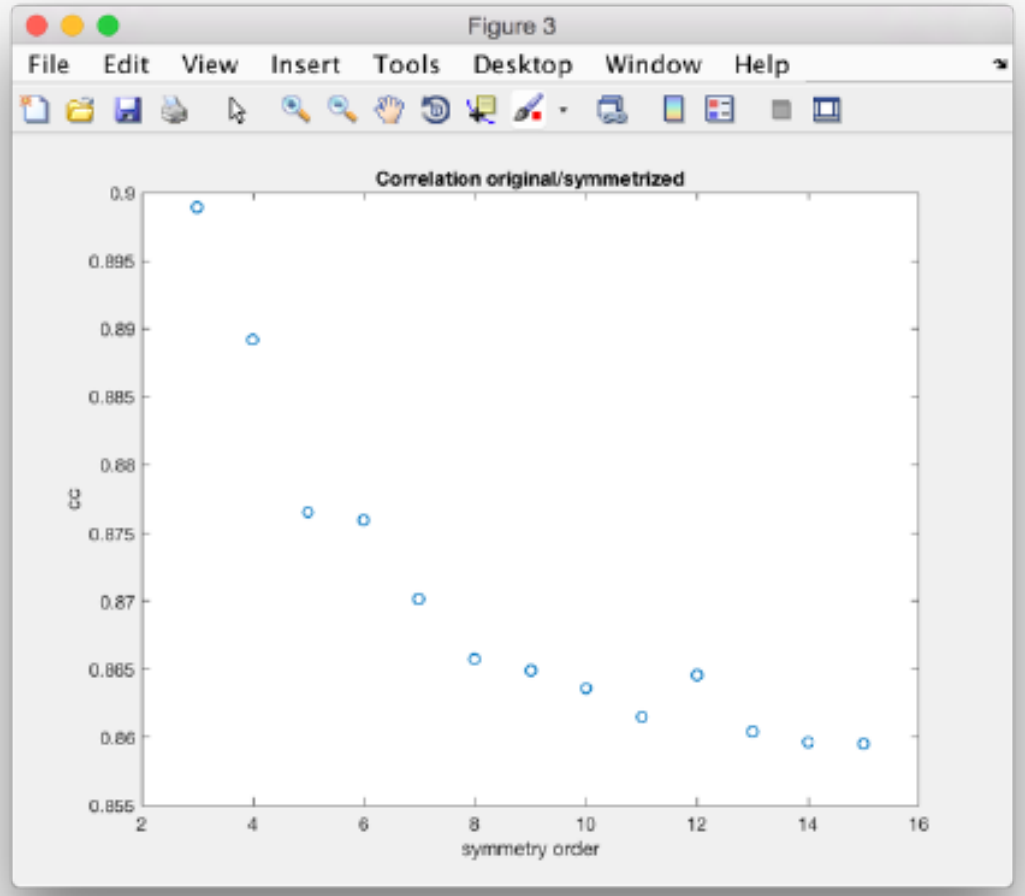
Testing the symmetry

We can check for different symmetry operators to confirm the presence of symmetry.

Basic command

The basic command for this task is `dynamo_symmetry_scan`. In its fundamental syntax, you need to state the type of symmetry operator to be tested, and a set of operator.

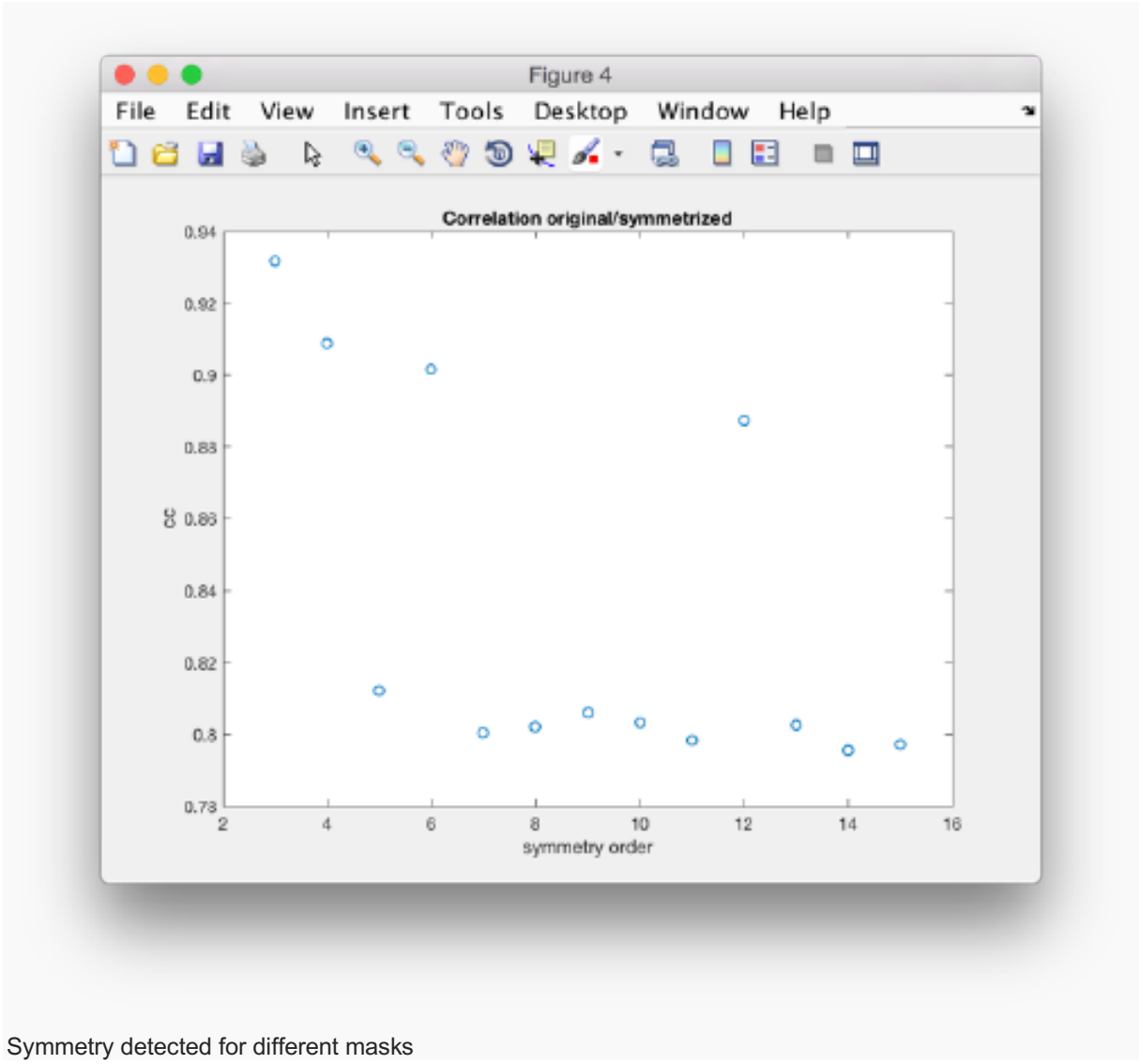
```
 stm =
dynamo_symmetry_scan('localized:a','c','order',3:15,'type','pearson','nfig'
,3);
```



Symmetry detected for different masks

With a more localized mask:

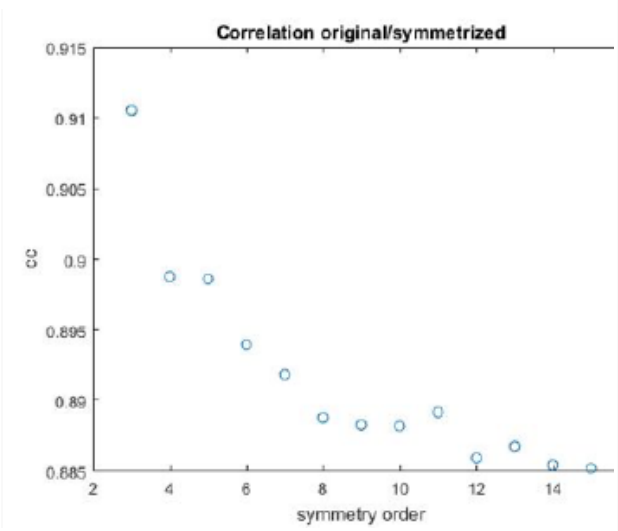
```
 slm =
dynamo_symmetry_scan('localized:a','c','order',3:15,'type','pearson','mask'
,'teethMask.em','nfig',4);
```

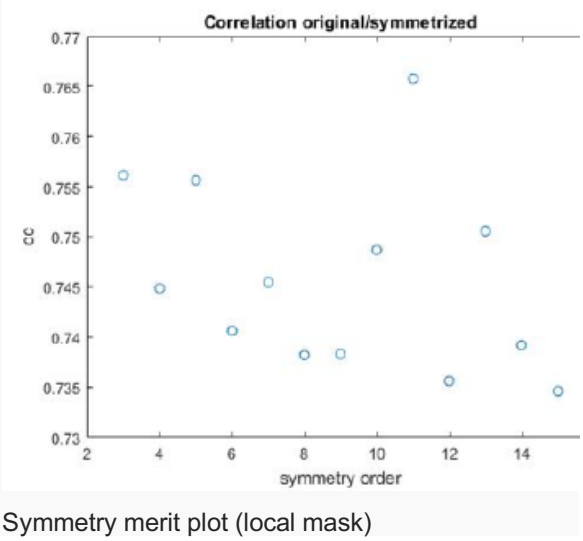


Symmetry detected for different masks

Interpretation of results

If you got the results above, you might have jumped to the conclusion that they already constitute an objective hint at a real C12 symmetry. This is, however, not true. The fact is that, with a slightly different picking of particles, you could have obtained the profiles in these pictures:

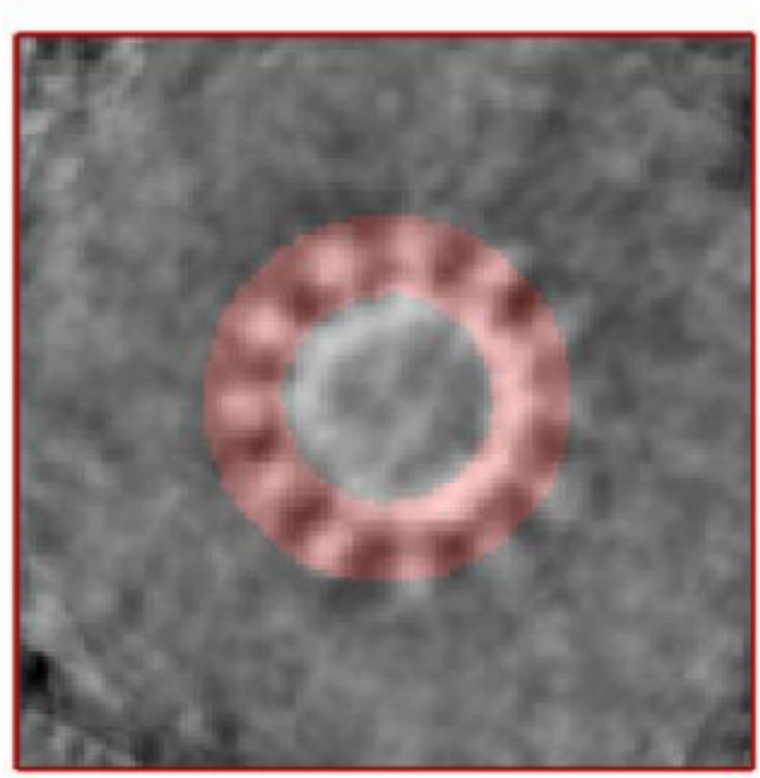




Symmetry merit plot (local mask)

Symmetry merit plot (global mask)

These images represent the result of the two symmetry-exploring commands described above obtained by a different user. . Surprisingly, they point at possible symmetry orders of 11 and 13. So, what is going on here? How is it possible that two different users that operate exactly the same walkthrough get contradictory symmetry estimations? The reason is that different users will pick manually slightly different particle centers, and this have a direct impact on the symmetry estimation. In short, the problem is that the density map `localized:a` is not garanteed to have its possible symmetry axis located along the center of the box, and this misleads `dynamo_symmetry_scan`. This can bechecked by opening the average in `mapview` and overlaying the file `'teethMask.em'` on it.

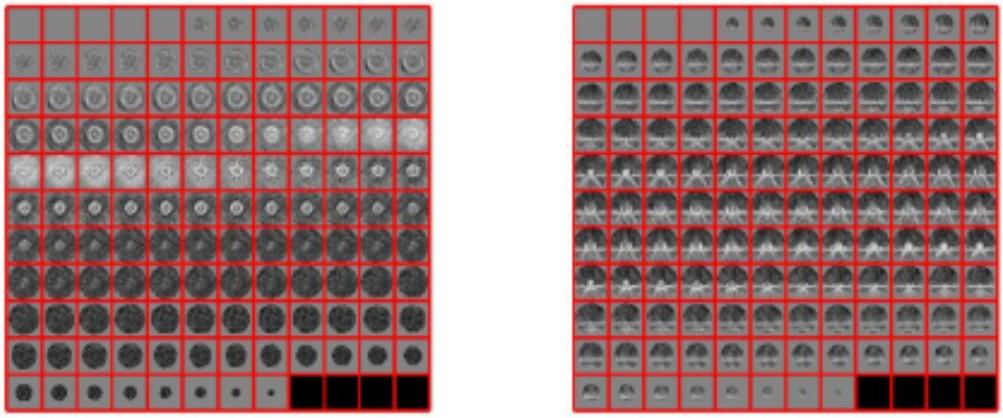


Mapview map n file `localized:a` overlayed with file `'teethMask.em'`

In this non-centered density map, any symmetry determination can only be an artifact. To solve this, we create a centered version:

```
ddb zOriented:a -r zo % extract average of zOriented as the ws variable
zo
ddb localized:a -r localizedAverage % extract average of localized as
the ws variable localized Average
sal =
dalign(localizedAverage,zo,'cr',0,'cs',30,'ir',0,'dim',64,'limm',1,'lim
',[20,20,20]); % aligns localizedAverage against zo (like that
localizedAverage is centered)
centerLocalized = sal.aligned_particle;
```

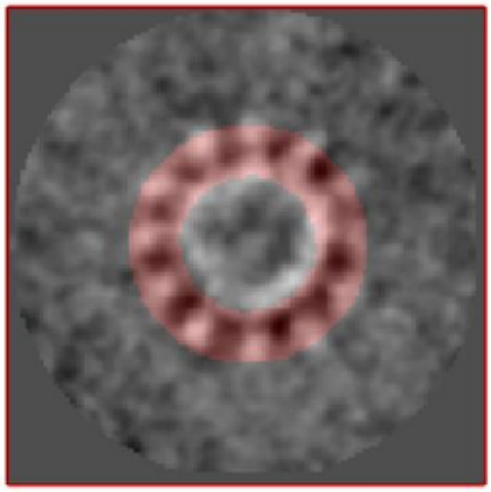
```
dview(centerLocalized);
```



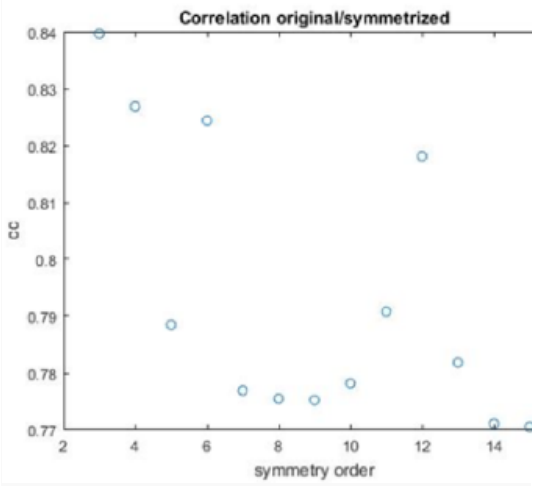
Recentered average (x and y views)

We can scan again the symmetry operators on the recentered volume.

```
 slmCentered =
dynamo_symmetry_scan(centerLocalized,'c','order',3:15,'type','pearson','mas
k','teethMask.em','nfig',4);
```

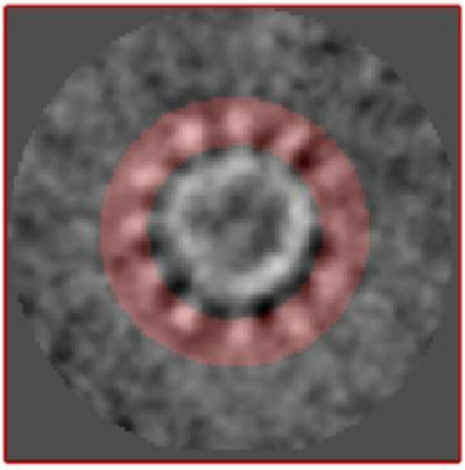


`'teethMask.em'` overlayed on the recentered average

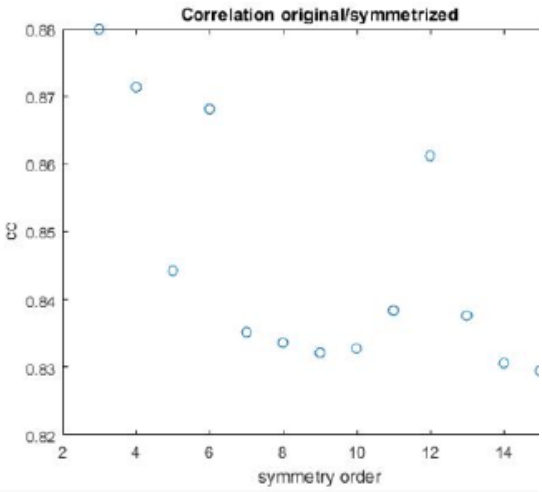


Symmetry merit plot after recentering using `'teethMask.em'`

Now the peaks located that at 6 and 12 are trustworthy. You can even create a new mask on the recentered average, carefully excluding intensities from the central ring structure to completely rule out any artefact.



Mask constructed *ad hoc* on the recentered average



Symmetry merit plot using *ad hoc* constructed mask

Subboxing

The *subboxing* technique consists in redefining the area of interest inside a previously defined average. In this example, we have use the full crowns as the subject of alignment and averaging. This has allowed us to use the whole signal carried by the full crown on each of the particles to drive the alignment robustly. This approach, however, will not allow us to identify heterogeneity and flexibility inside the individual crowns. Additionally, the possible hetereogeneity inside the crown will decrease the quality of the alignment when the crown is treated as a whole.

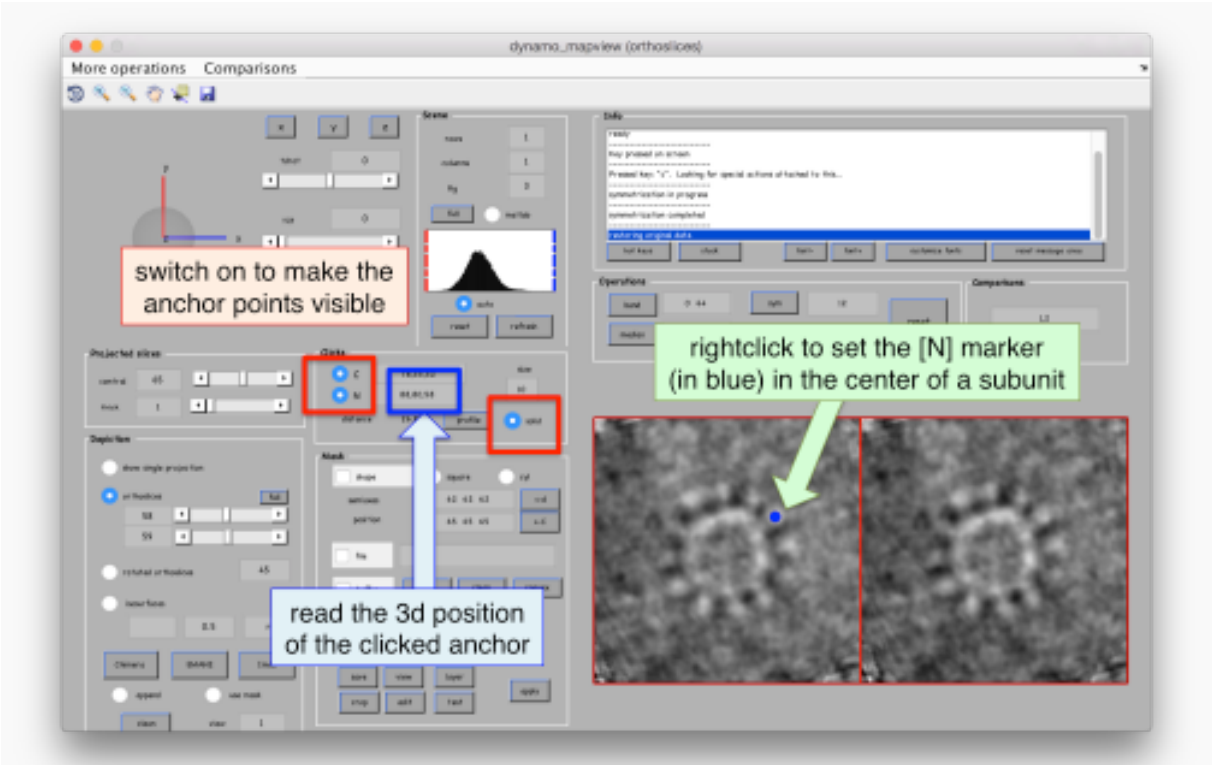
With the *subboxing* technique, we can use our previous results to setup an approach that centers the alignment on the individual teeth. Each one the particles in our new data set will be a *subbox* (a tooth) extracted from the previos *box* (the crown) using the alignment parametes for the full boxes gained by the projects that we have ran so far.

Defining a subboxed subunit

First, we need to define the location that we want to subbox. We use `mapview`

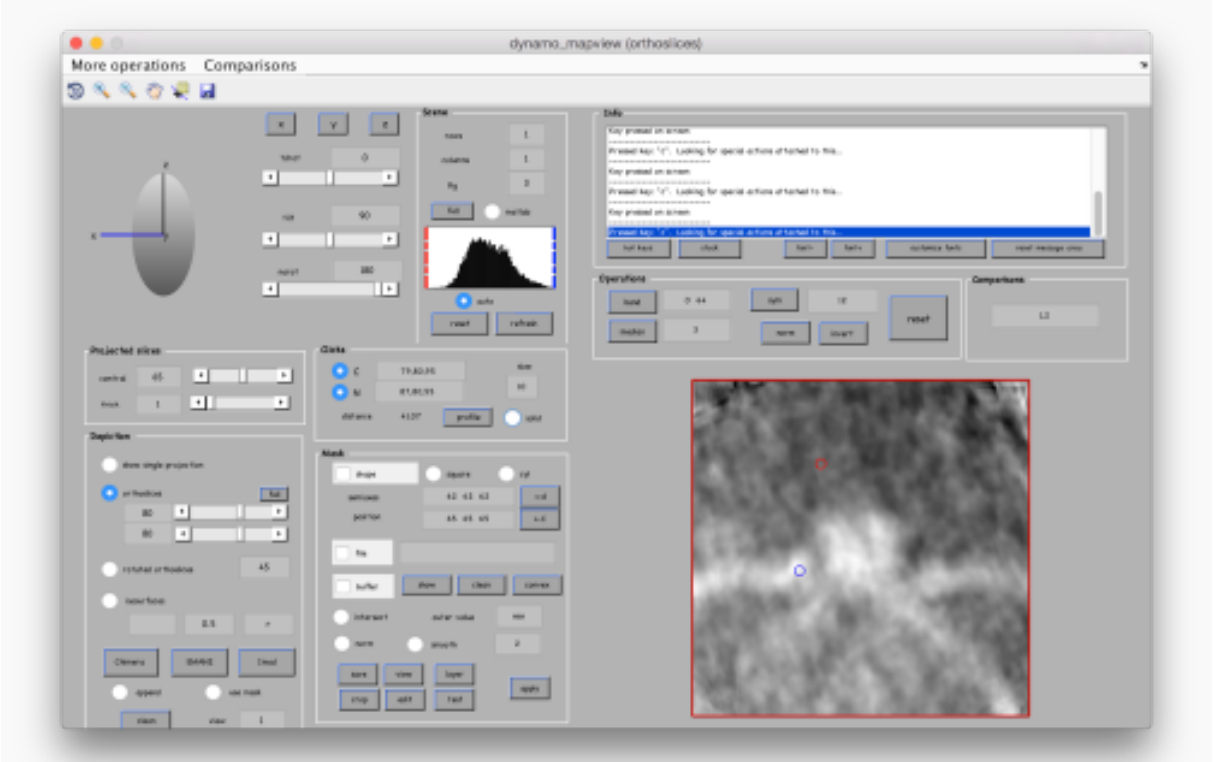
```
dmapview localized:a
```

to show the last average in the project `localized`



Selecting the center of the tooth subunit with `mapview`

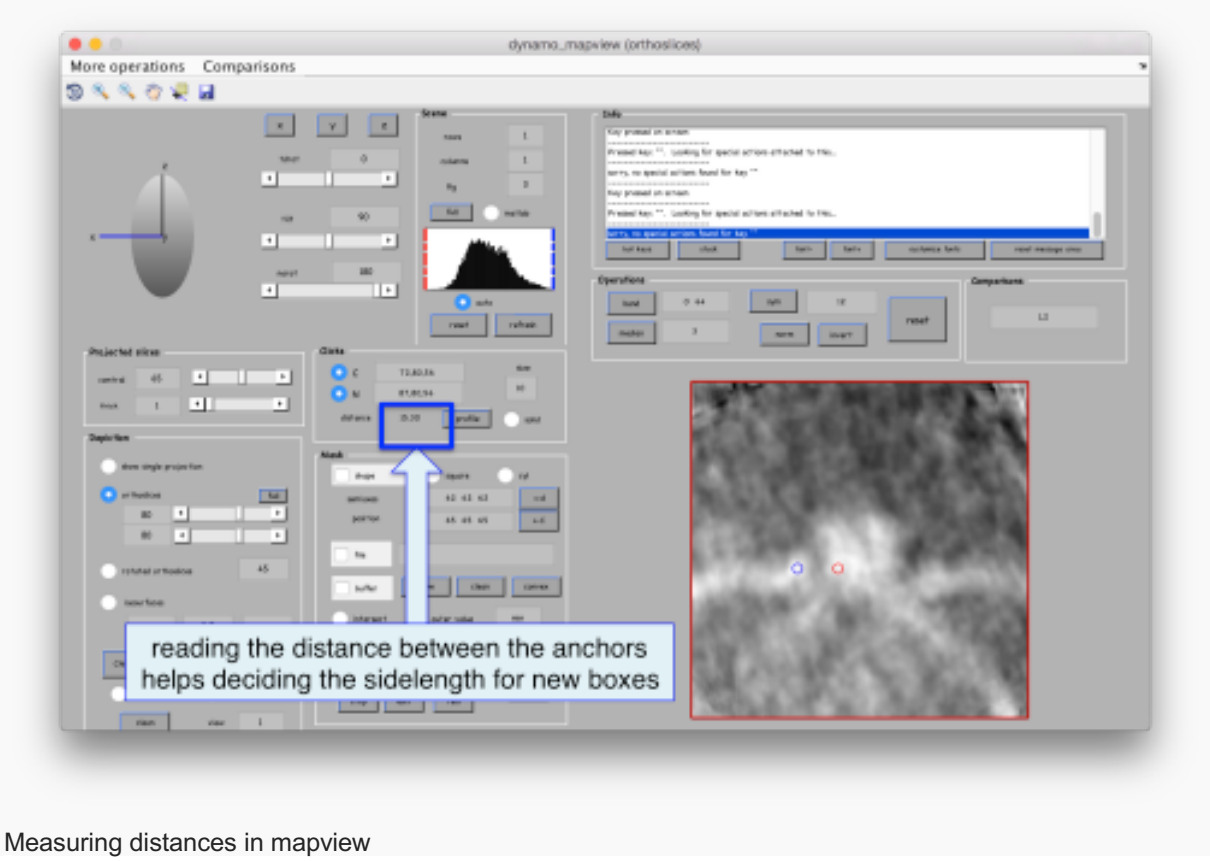
You probably want to explore several views in mapviews (*x'/'y' and 'z') to make sure you (roughtly) hit the center of the subunit of interest. Guiding yourself only for a view on* z *can be misleading.*



Selecting the center of the tooth subunit with `mapview` on the *y* view

.

As we are in this browser, we can check the size of the subunit that we want to *subbox*, using the two markers C and N. This will be useful later when we want to actually crop physical particles using `dtcrop`, when we will be required to input a sidelength for the subboxed data folder.



Measuring distances in mapview

.

We define a variable to contain the position that we read in mapview for the blue (North) anchor.

```
rSubunitFromCenter = [88,80,53 ] - [64,64,64];
```

We subtract the half sidelength of the full box, because the next command will need the position of the asymmetrical unit expressed in relation to center of the box.

Creating a subboxing table

We extract the last refined table:

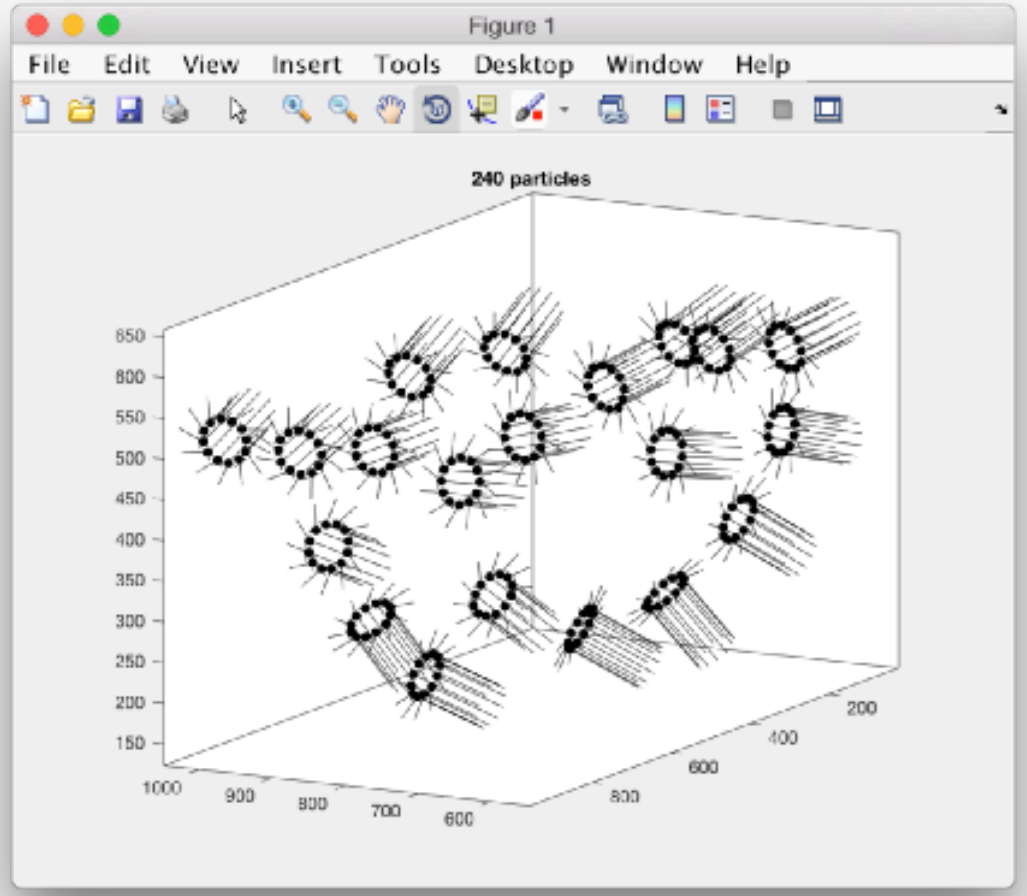
```
ddb localized:rt -r t
```

and define positions related by C12 symmetry along the axis

```
ts = dynamo_subboxing_table(t,rSubunitFromCenter,'sym','c12');
```

The new subboxed table `ts` will have 12 times as many rows as the original one. The system of reference on each particle will point *z* in the direction of its original box, but the *x* and *y* orientations of each tooth will be symmetrically related in the same crown. We can depict this geometrical relationship by plotting a sketch of all the particles in the table:

```
figure;dtplot(ts,'m','sketch','sketch_length',100,'sm',30);view(-
151,12);axis equal;
```



Sketch of orientations of the new subboxed table `ts`

.

Creating a subboxed data folder

Now we have all we need to go back to the original tomogram and crop the subboxed particles into a new data folder using the sidelength that we checked with `mapview`:

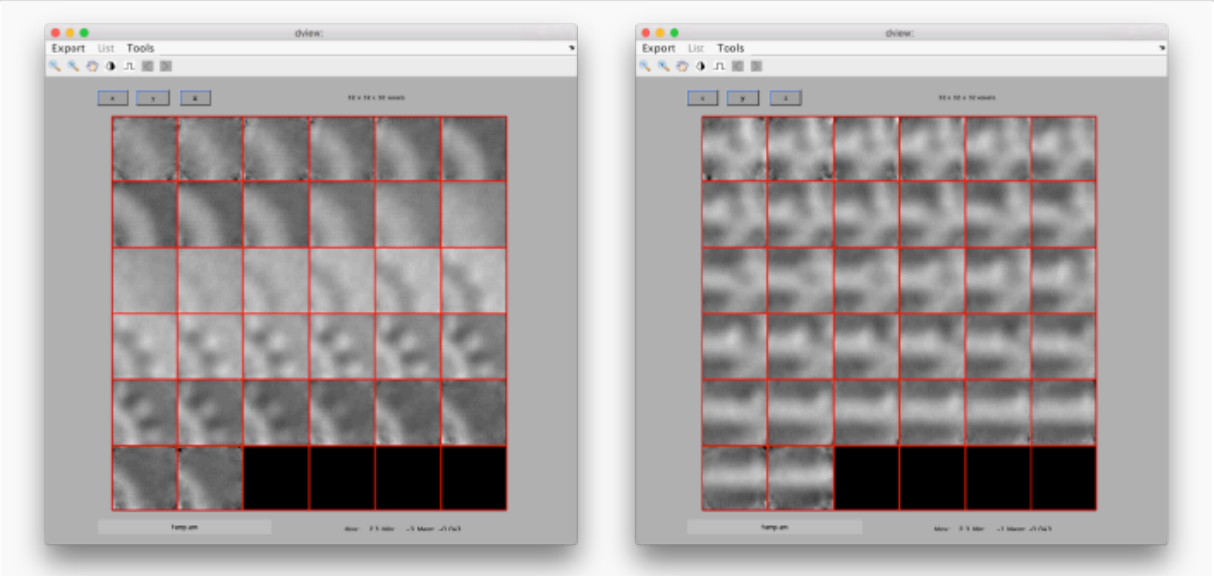
```
dtcrop('crop.rec',ts,'subboxData',32);
```

We can run our typical sanity check to ensure that everything ran correctly

```
osb = daverage('subboxData','t',ts,'fc',1);
```

To visualize it you can write:

```
dview(osb.average);
```



Density map of the average of subboxed teeth

.

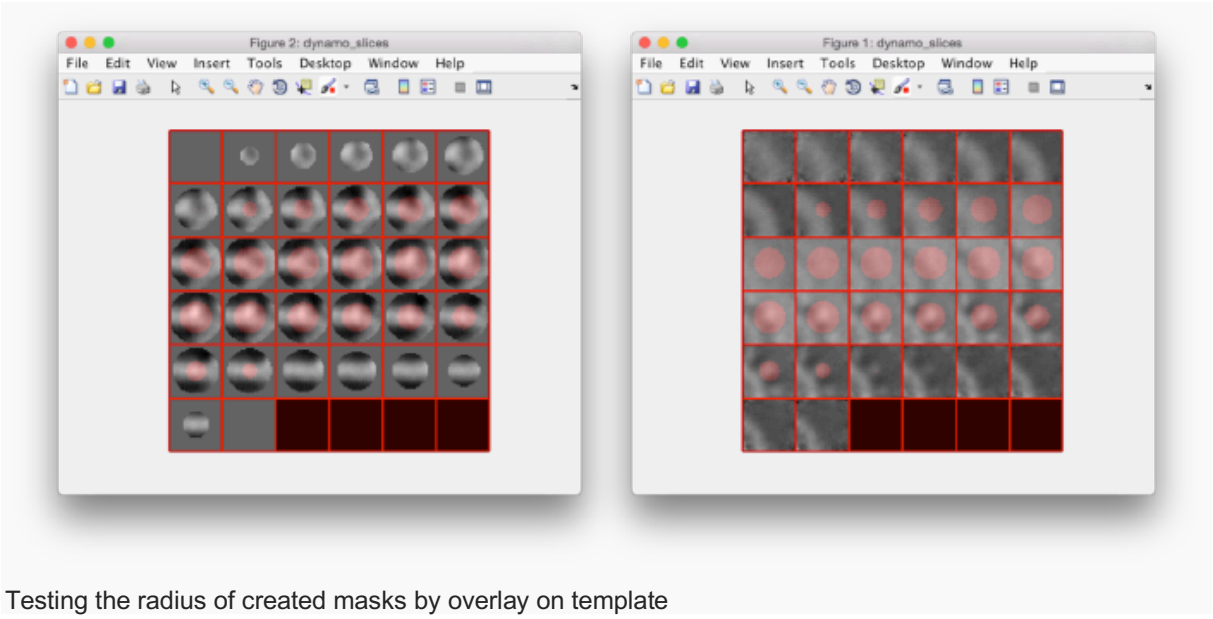
As it looks like we expected, we just write it into a file, which we will use to define a new project.

```
dwrite(osb.average,'subboxRaw.em');
```

Defining masks

There are some tools to quickly format masks for project. If we want to check a reasonable radius for our mask, we can use the `dsphere` command and overlay the created mask on the template

```
cs = dynamo_sphere(10,32);
figure;dslices(osb.average,'y','ov',cs,'ovas','mask');
figure;dslices(osb.average,'ov',cs,'ovas','mask');
```



Testing the radius of created masks by overlay on template

.

In fact, this mask is probably too risky: we need a minimum of signal to drive the alignment

```
dwrite(cs,'maskTooth32.em');
```

Subboxing project

We create the project through:

```
dcp.new('subboxBig','d','subboxData','template','subboxRaw.em','masks','def
ault','t','subboxData/crop.tbl','show',0);
```

In this case, we use `'show' 0` would suppress the dcm GUI. We use typically this option when we want to enter project parameters through the command line. This is exactly equivalent to entering parameters through the GUI.

```
dvput subboxBig mask maskTooth32.em
dvput subboxBig ite_r1 3
dvput subboxBig cr_r1 4
dvput subboxBig cs_r1 2
dvput subboxBig ir_r1 4
dvput subboxBig is_r1 2
dvput subboxBig rf_r1 2
dvput subboxBig rff_r1 2
dvput subboxBig dim_r1 32
dvput subboxBig lim_r1 [4,4,4]
dvput subboxBig limm_r1 1
```

A list of the names of the parameters can be displayed through the command `dvhelp`. The computing environment is set using the `destination` parameter, shortnamed `dst`. To run it in matlab use:

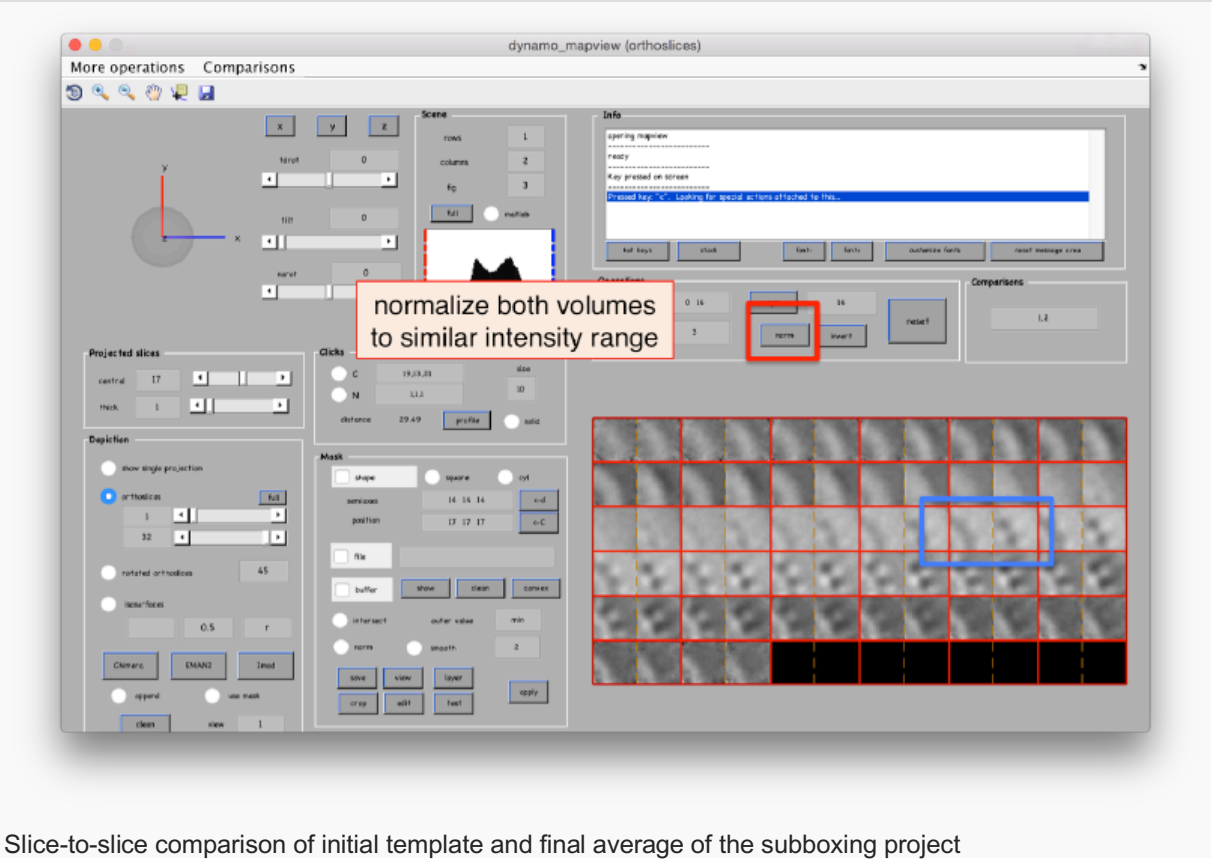
```
dvput -dst matlab_parfor
```

For standalone running, use:

```
dvput -dst standalone
```

After running the project, we can check the effect of the independent refinement of the "tooth" units:

```
ddb subboxBig:a:ite=[0,3] -m
```

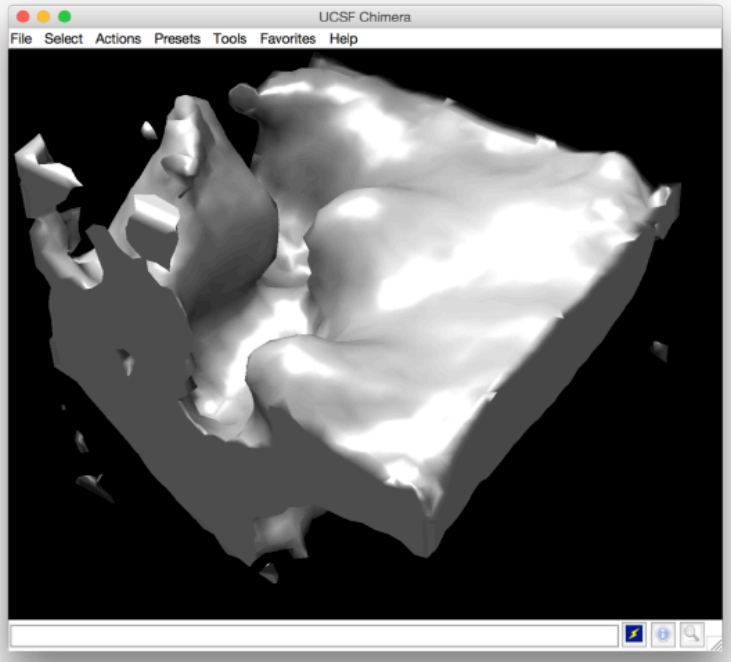


Slice-to-slice comparison of initial template and final average of the subboxing project

.

or send the average to Chimera through:

```
ddb subboxBig:a -c
```



Density map of the average of subboxed teeth after refinement

.

Visualization

You can continue working on this data in the Walkthrough on creation of 3d scenes, where you will learn to depict tomgraphic slices, place templates in table positions, and depict graphical elements representing model geometries.