



Command line: classification

Main goals of this tutorial:

- * Classify a data set with PCA.
- * Refine the results of a classification.

What do you need to classify a data set?

Basically you need three input elements:

1) Data

Needs to be formatted as a regular Dynamo data folder.

2) Table

Here you code the whole metadata: alignment parameters, missing wedge descriptors...
Usually, such a table is the result of a previous (or concurrent) alignment procedure, which might have been run in Dynamo or in other software package and then converted into Dynamo table formats.

3) Mask

(optional)

To state which voxels define the area of interest in the classification.

In this tutorial we show how to use command line tools to operate on these elements to create PCA-based classifications

PART I

Creation of synthetic data

(hors d'oeuvre)

In this part we will create a synthetic set of data, table and mask to simulate a real classification procedure.

In our way we will take some time to stop at several commands and procedures that might be useful in other contexts.

Create a new folder (mkdir), go there (cd) and let's get started:

This time, we will use the GUI version of the tutorial generation tool:

```
>> dtutorial;
```

In the GUI that opens (next slides), we can control the generation of data and tables.

We are interested in a table that aligns the generated data, but NOT perfectly, as in real life we will not have “perfect” alignment tables.

This is controlled by the parameters “pshift”, “pnarot”, “paxis” (p stays for “perturbation”) which determines the “approximative” table that we are interested into.

dtutorial will create its own directory and put lots of things there including several tables, a data set and a mask:

8

M: #particles of class 1

0

N: #particles of class 2

em

extension: of particles

Options

0.1

noise:

60

tilt: semiaperture

3

dtilt:

3

dshift:

360

cone:

ftype: single/dual

single

☐ bin

☒ tight

☐ just_shift

☐ just_data

Coarse table

paxis:

5

pnarot:

10

pshift:

3

Create project

virtual project (in Matlab)

vpr

☐ multireference

0

☒ project

project_for_tutcc

destination: computing environment

system_omp

Information

GUI

help

reset

font-

font+

customize fonts

reset message area

Listbox

Folder does not exist

Folder does not exist

Folder does not exist

Folder does not exist

Tutorial

tutorial folder:

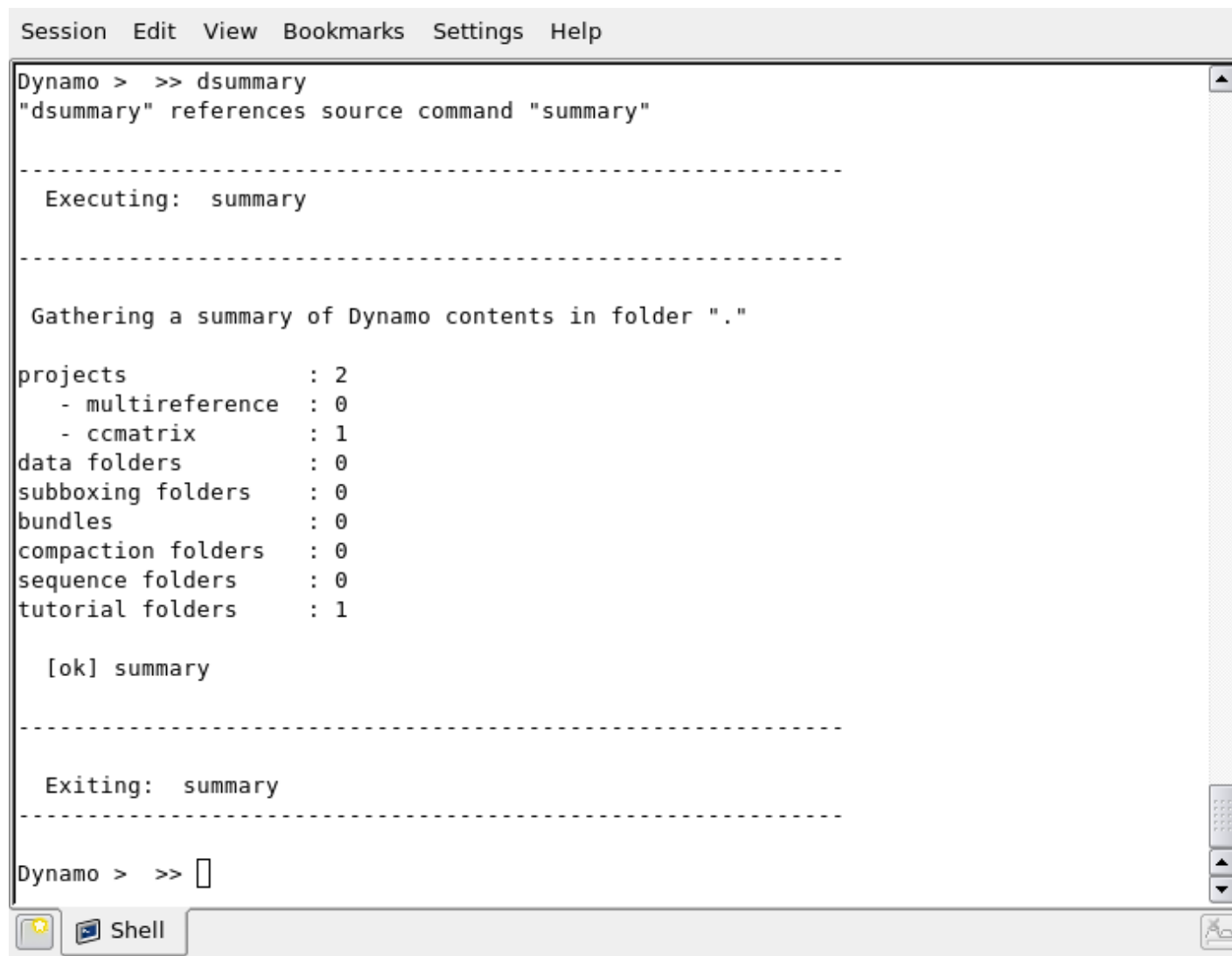
tutcc

Create tutorial

here you control the approximative table

type some name for the tutorial folder

and create it!



```
Session Edit View Bookmarks Settings Help
Dynamo > >> dsummary
"dsummary" references source command "summary"

-----

Executing: summary

-----

Gathering a summary of Dynamo contents in folder "."

projects          : 2
- multireference  : 0
- ccmatrix        : 1
data folders      : 0
subboxing folders : 0
bundles           : 0
compaction folders : 0
sequence folders  : 0
tutorial folders  : 1

[ok] summary

-----

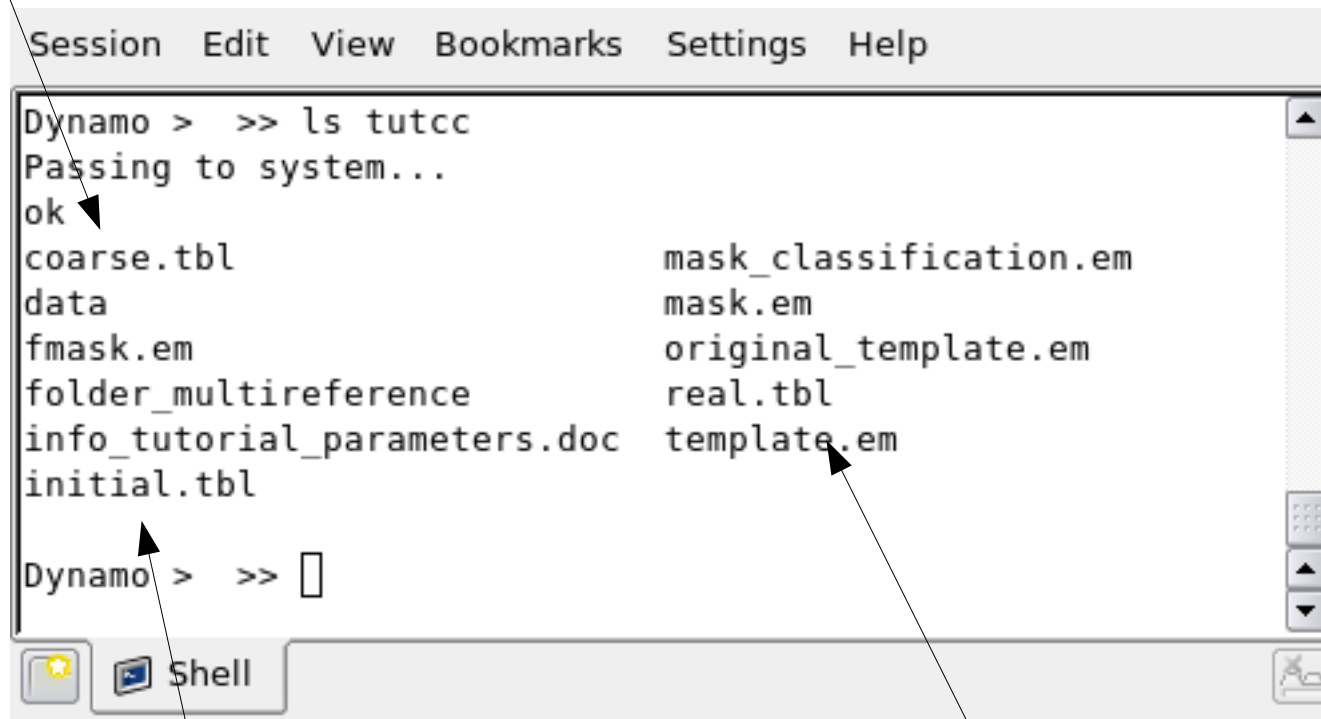
Exiting: summary

-----

Dynamo > >> 
```

run a summary after the tutorial runs just to check that things are ok: the tutorial produced two projects: an alignment project and a ccmatrix project. We could just run this ccmatrix project and it will create a ccmatrix that can be used for classification... but in this tutorial we rather want to focus on how to create such projects in the first place, so we just explore the elements that we have in the tutorial

the approximative table that we can use to create a “realistic” classification:



A screenshot of a terminal window titled "Session Edit View Bookmarks Settings Help". The terminal shows the command "Dynamo > >> ls tutcc" and its output. The output lists files in two columns: coarse.tbl, data, fmask.em, folder_multireference, info_tutorial_parameters.doc, initial.tbl, mask_classification.em, mask.em, original_template.em, real.tbl, and template.em. The prompt "Dynamo > >> " is followed by a cursor. At the bottom, there is a "Shell" button and a status bar with icons.

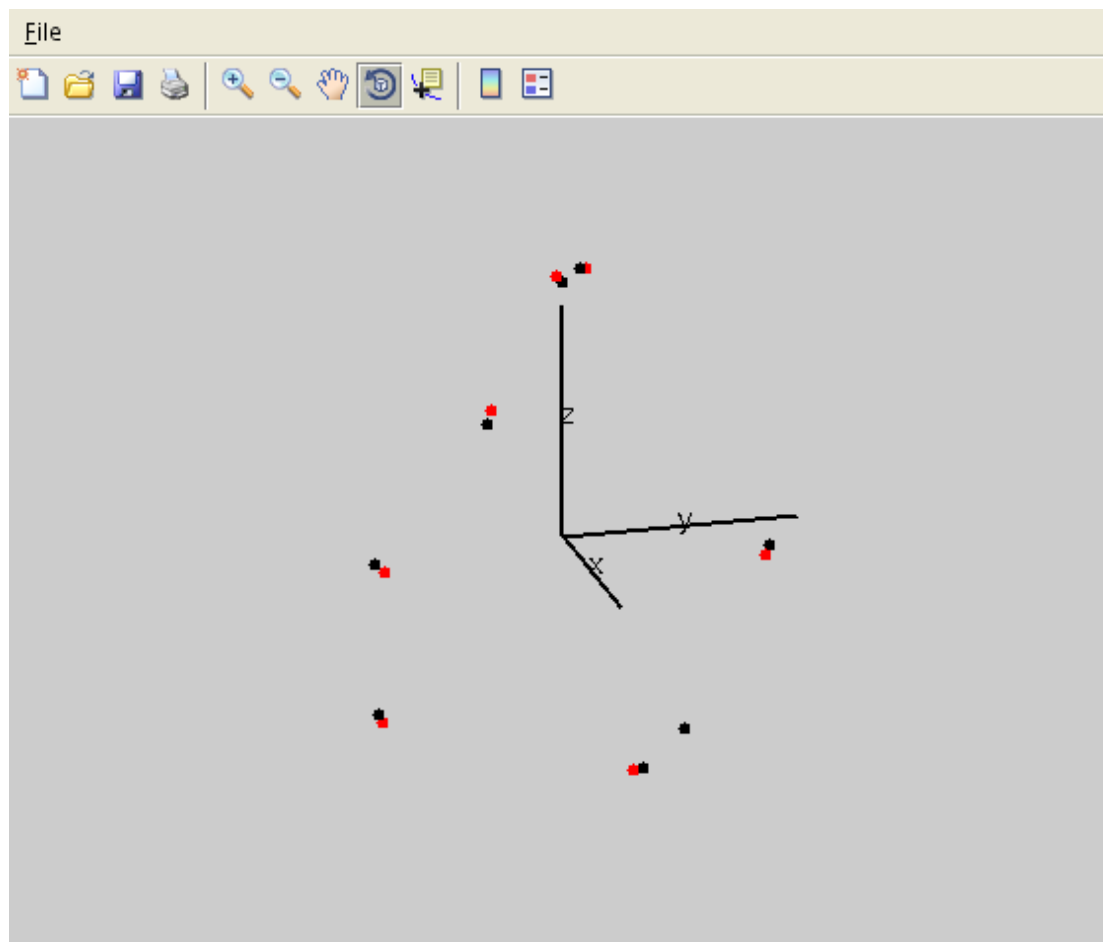
```
Dynamo > >> ls tutcc
Passing to system...
ok
coarse.tbl                mask_classification.em
data                     mask.em
fmask.em                 original_template.em
folder_multireference    real.tbl
info_tutorial_parameters.doc  template.em
initial.tbl

Dynamo > >> 
```

that's a real table with the actual parameters

not used in this tutorial: a “blank” table with alignment parameters set to zero.
typically used to start and alignment project.

```
>> dtplot tutcc/coarse.tbl -cl r  
>> dtplot tutcc/real.tbl -cl b
```

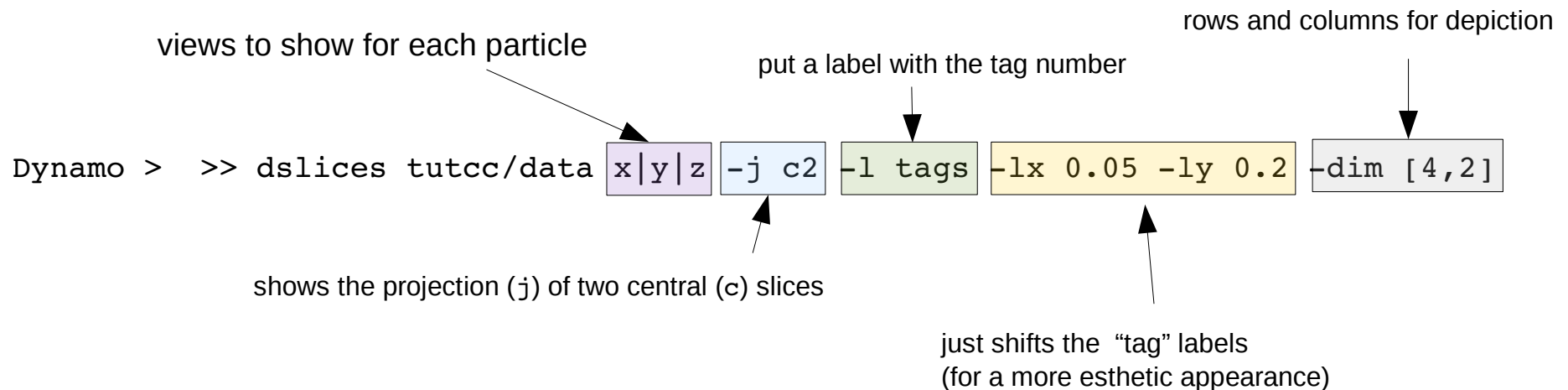


you can check quickly the data set with :

```
Dynamo > >> dslices tutcc/data -j *
```

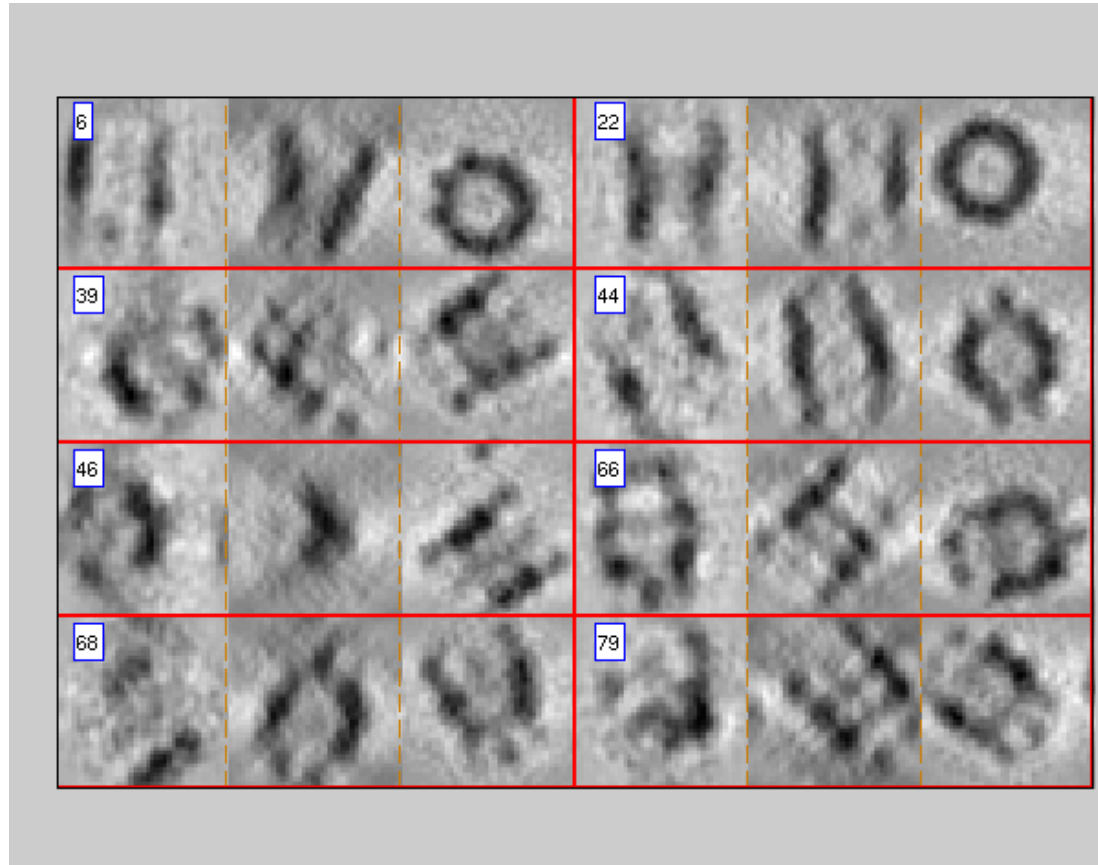
or with a more creative:

```
Dynamo > >> dslices tutcc/data x|y|z -j c2 -l tags -lx 0.05 -ly 0.2 -dim [4,2]
```



Note: if you want to do similar depictions for real data sets (which won't have just eight particles) you might need:

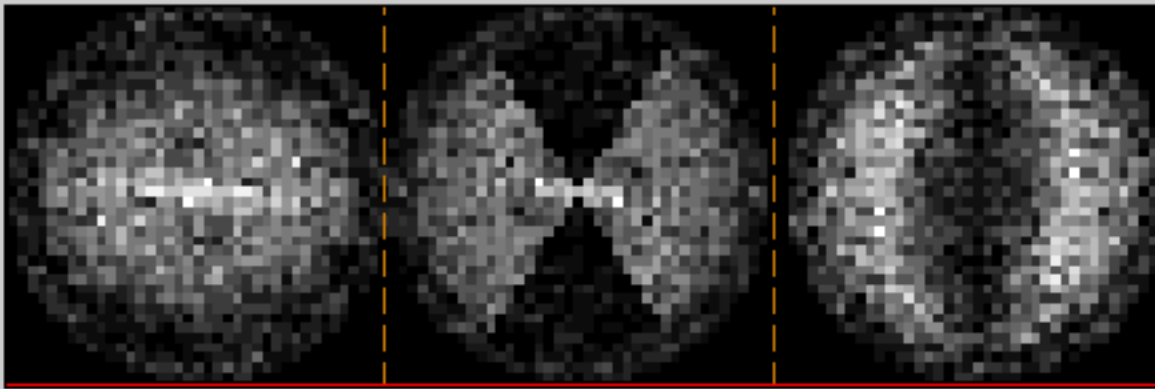
- 1 select subsets of tags
- 2 use the `-otf` flag ("on the fly", to avoid storing all particles simulatenously in memory)
- 3 use `dgallery` instead.



well, those are the eight particles in the data set... as you see, they are missaligned

also, notice that each particle has a missing wedge:

```
Dynamo > >> dwedge_estimate tutcc/data/particle_00006.em -show on
```



So, before entering the classification itself, we continue playing with the created elements:

Now, we want to check how “approximate” is the approximate table coarse.tbl.

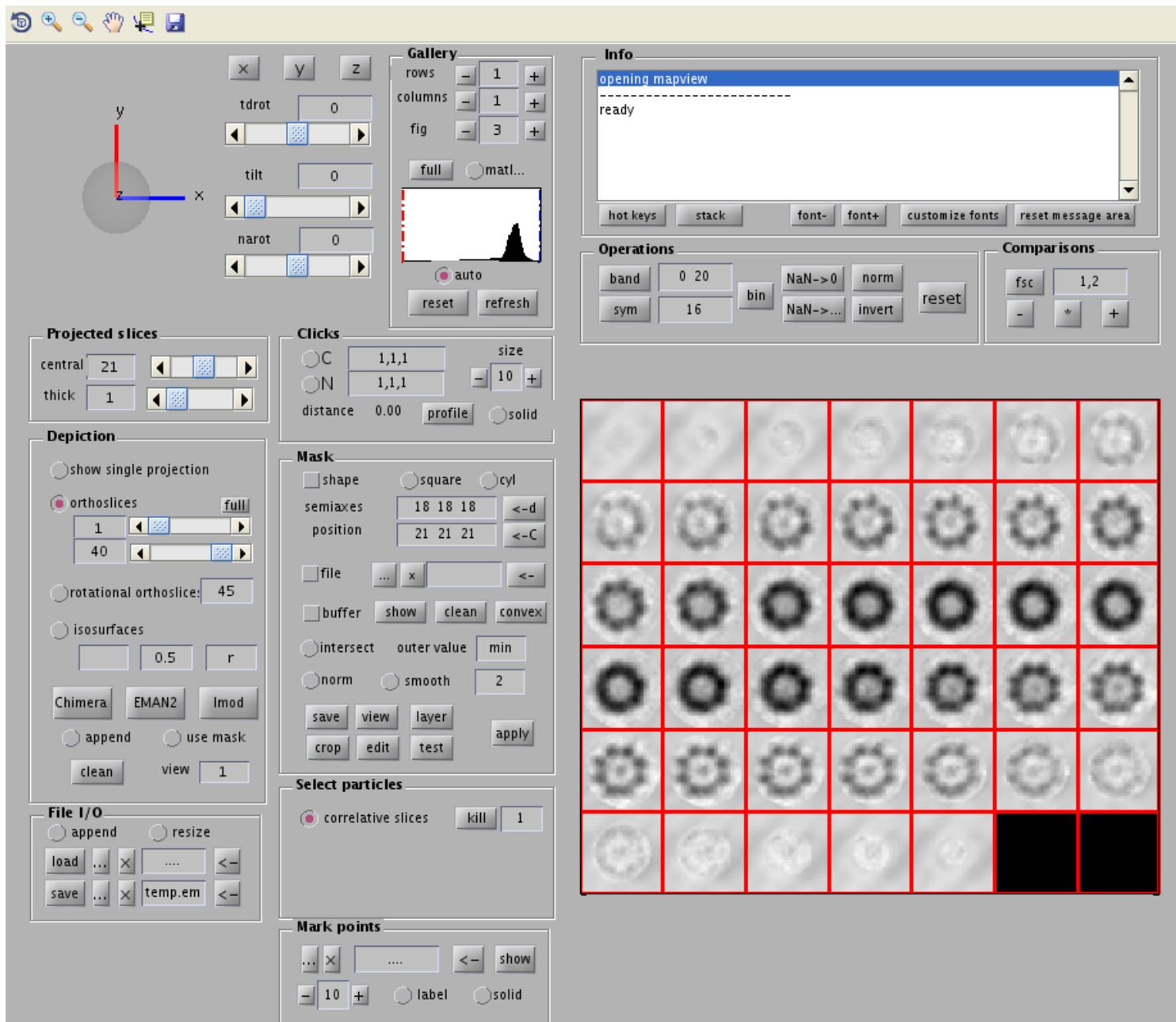
A typical way to explore tables is just by applying them onto the data to produce an average:

```
>> daverage tutcc/data -t tutcc/coarse.tbl -fc on -o acoarse.em
```

Let us depict it with mapview (next slide):

```
>> dmapview acoarse.em
```

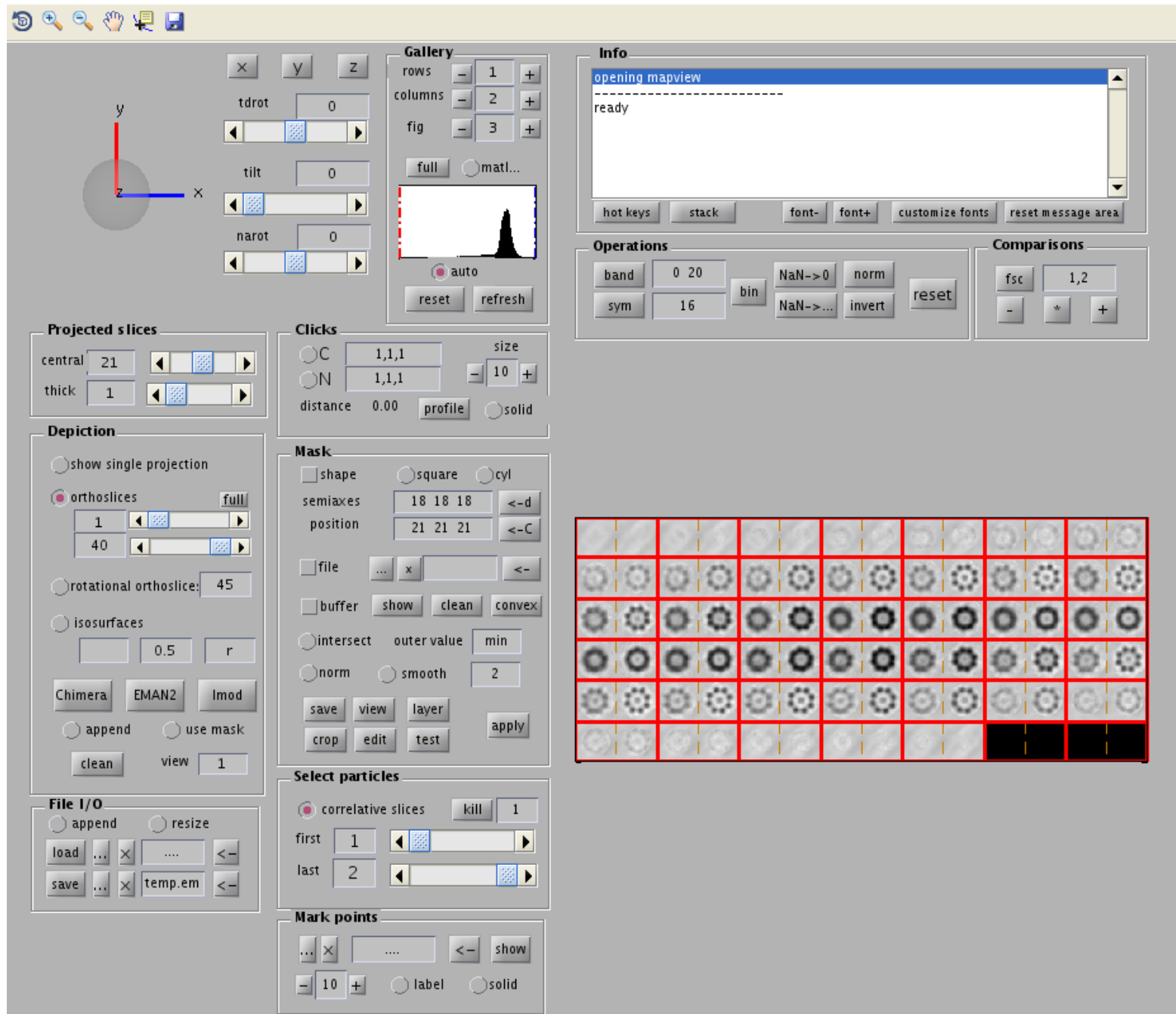
and check that it is a reasonable approximation to the alignment parameters



just for test let us compare this with the average that the real table would produce:

```
>> daverage tutcc/data -t tutcc/real.tbl -fc on -o areal.em
```

```
>> dmapview areal.em -append on;
```



you can see how the coarse table produced a blurrier version of what we would get with the real parameters

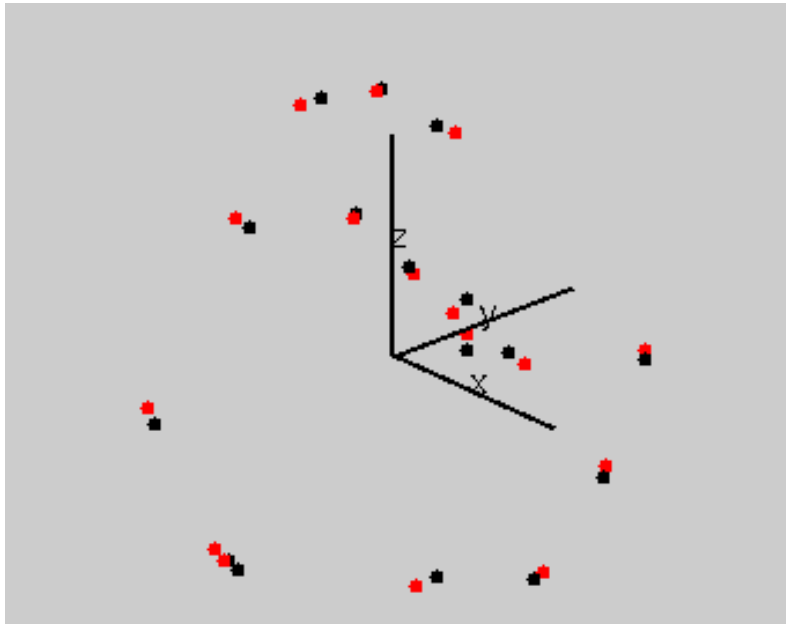
ok, now we know how a coarse table looks like and how it approximates the real metadata.

But now, in our tutorial we used only particles on one class... let us create a second tutorial set with two types of particles, so that we can test later our classification procedures.

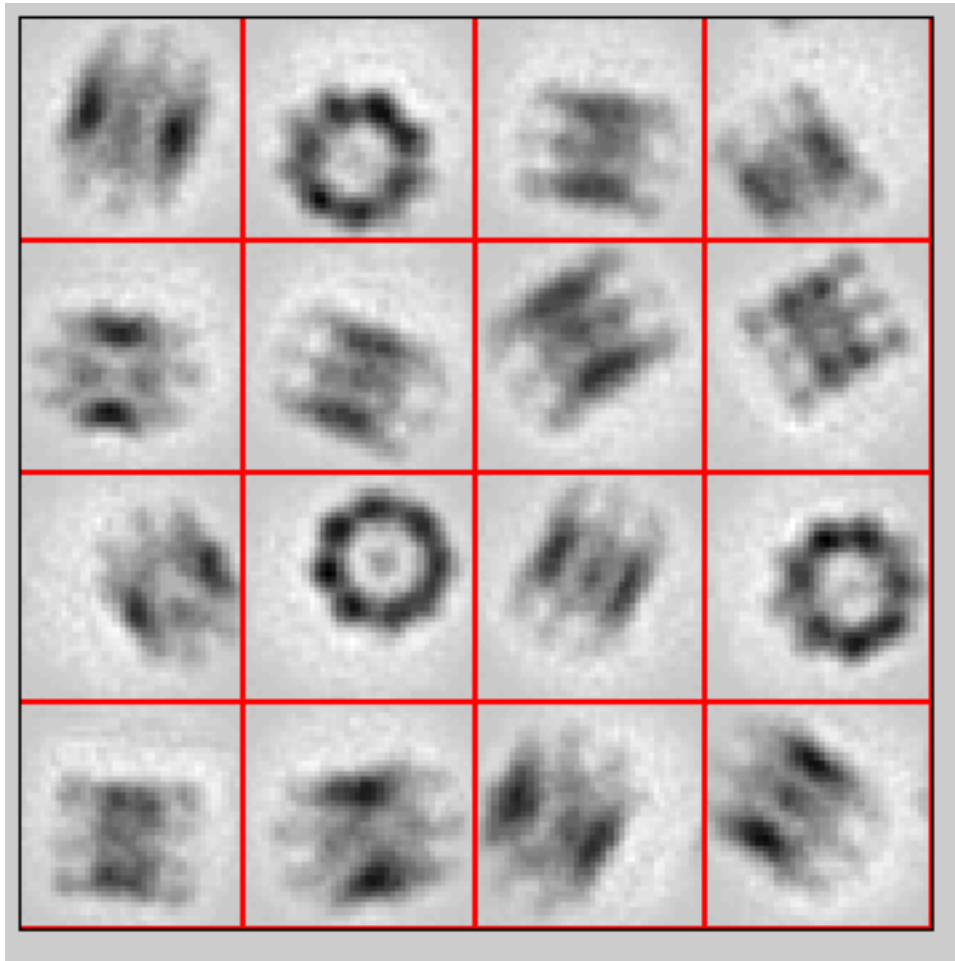
From the command line we can run the order:

```
>> dtutorial tut -M 8 -N 8 -tight on;
```

which will create the folder `tut` with 16 particles, 8 in each class.



```
>> dslices tut/data -j *
```



ok, the raw, unaligned particles from both classes look pretty much the same.
In the next part, we will disentangle them with a PCA approach.

PART II

Classifying the data

In this part we will operate a command line classification on the elements (data and table) that we created in part I

We thus start by creating a ccmatrix for the table, data and mask combination we produced.

In the simplest syntax of the command dvccmatrix we can just create a project (lets call it pcc) with those elements:

```
>> dvccmatrix pcc -t tut/coarse.tbl -d tut/data -m tut/mask.em
```

pcc will just compute the ccmatrix of the data set aligned by the table and restricted to the mask.

Two important notices are:


- 1 The command dvccmatrix admits many modifications to change computation and execution settings, that have been set to zero with this syntax.
- 2 For illustrative purposes we are going to operate the classification step by step: we compute by now only the ccmatrix, later we will compute the PCA analysis (divided into Xmatrix computation and spectral computation: eigenvectors and eigencoordinates) and then the classification itself (by kmeans).

Note however that dvccmatrix can produce projects that tackle all or several of these steps separately with the flags "steps". For instance we could have typed:

```
>> dvccmatrix pccall -t tut/coarse.tbl -d tut/data -m tut/mask.em -steps all
```

what you just generated is a project of type "ccmatrix". You can get operative information on it with the general `dynamo_vpr_info` (or `dvinfo`)

```
-----  
: (size: 40 x 40 x 40)  
-----  
valid unfolded project      : YES  
"destination"               : matlab  
"how_many_processors"       : 1  
"cluster header"            : cluster_header.sh(NOT FOUND)
```

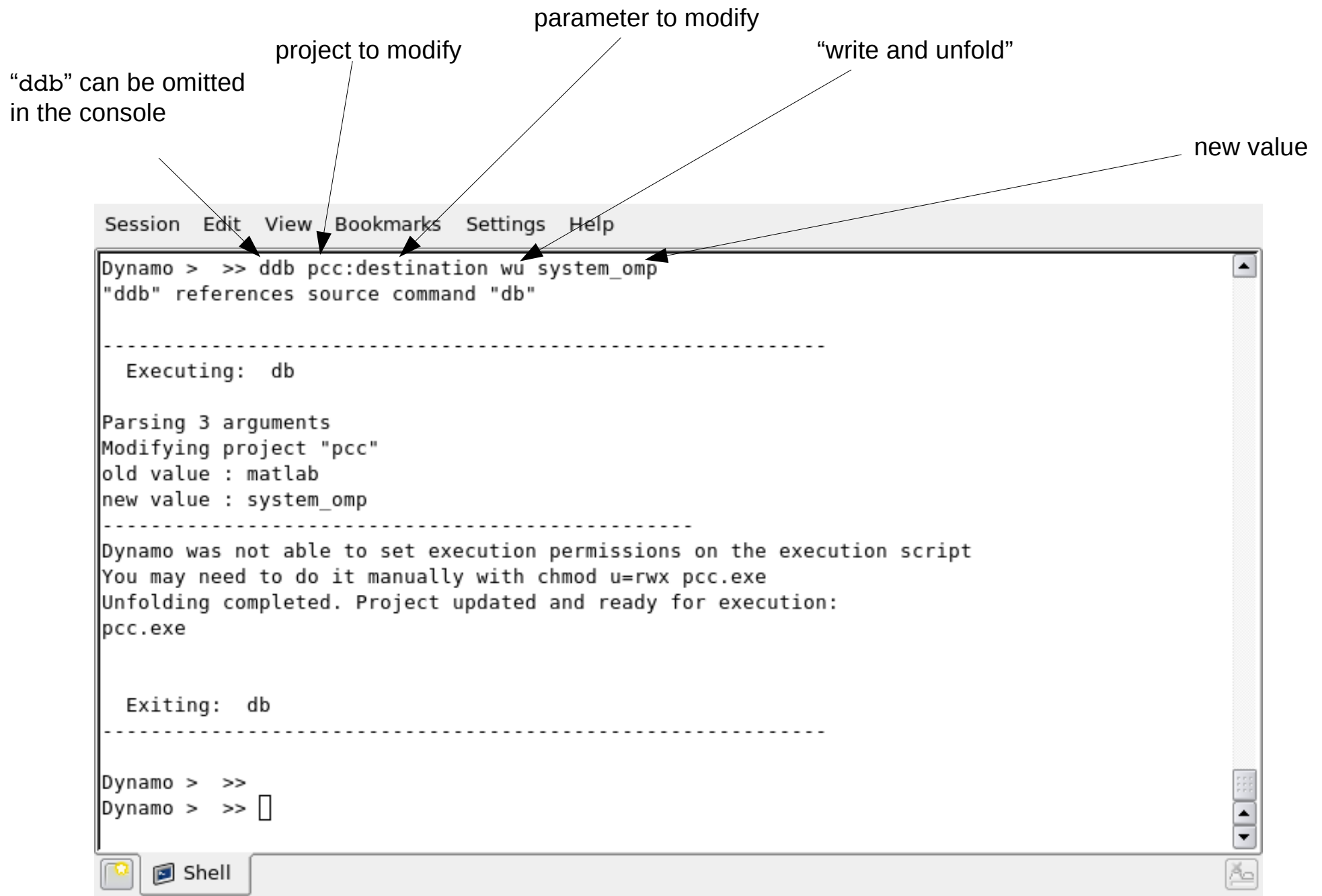


by default the created project will run in Matlab.

You might prefer to execute it in a system shell for several reasons:

- 1) you want to keep the current working shell free for interactive work
- 2) You want to use the multicore capacities of your machines (not accessible through the Matlab version)
- 3) You are not using the Dynamo -Matlab licence

So, we just change the “destination” parameter in our project. we use the ddb command for that:



You can also modify other parameters. If you have a multicore machine you can use the parameter “how_many_processors” (or in shift form “cores”) to divide the task among several processors.

Well, we have a ccmatrix project.

Before launching it, we consider the command `dynamo_vpr_ccinfo` (`dvccinfo`), which is very useful to control in which state is a classification task: what has been done, which elements are available, how they were computed. Just type the command on the project we created:

```
>> dvccinfo pcc;
```

Session Edit View Bookmarks Settings Help

```
Dynamo > >> dvccinfo pcc
"dvccinfo" references source command "vpr_ccinfo"
```

```
-----
Executing: vpr_ccinfo
```

```
Parsing 1 arguments
-----
```

```
Project: "pcc" (iteration: 1)
project type      : ccmatrix
-----
```

```
Parameters in round 1
```

```
ccmatrix          : 1
ccmatrix_type     : align ;
ccmatrix_batch    : 128
Xmatrix           : 0
Xmatrix_maxMb     : 100
PCA               : 0
PCA_neigs         : 4
kmeans            : 0
kmeans_ncluster   : 2
kmeans_ncoefficients : 3
-----
```

```
Computed files
```

```
tags file         : not available
ccmatrix file     : not available
ccmatrix actions  : undefined
Xmatrix (blocks)  : no blocks of Xmatrix available
Eigenvolumes      : not available
eigentable        : not available
Subaverages       : not available
-----
```

```
Exiting: vpr_ccinfo
-----
```

```
Dynamo > >> 
```

← settings as found in project

← what has been computed
(nothing in our case)

Shell

We can start the computations: we just launch the project in a system shell, different to the one we are using for our interactive procedure. Of course, Dynamo needs to be activated on that shell/

first you get information about what the data flow in the project...

```
[casdanie@cina-hpws01 testcc]$ ./pcc.exe
Reading the card ./pcc/cards/ite_0001/card_iterref_ref_001_ite_0001.card
sending to system order dynamo_ccmatrix_compute ./pcc/cards/ite_0001/card_iterref_ref_001_ite_0001.card 0
Initializing MATLAB Compiler Runtime version 7.14
Starting Up: "MCR libraries starting dynamo_ccmatrix_compute. [kernel function]"
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    CC matrix computations for ite 1, ref 1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
action chain read from virtual project: (round 1)
align ;
MASK (classification)
    Sidelength 40, 28671 active voxels (45%)
    stored in the database as "mask_ccmatrix"
TAGS
    Identity and ordering of particles appearing in the ccmatrix
    total in table :16
    in disk       :16
    actually used  :16
    Stored as database item "tags_ccmatrix"
ACTIONS ON PARTICLES
    align ;
    Stored as database item "actions_ccmatrix"

Starting ccmatrix computations:
```

By default, particles are just aligned before comparing them in pairs.
In real life we might need to use the parameter "ccmatrix_type" to force dynamo to bin them (for speed)
or symmetrize them (for a better SNR)

...and then some execution information...

the “blocks” referred to here are submatrices of the ccmatrix that are computed separately for two reasons:

- 1 parallelism and
- 2 memory optimization

In this tutorial both are rather irrelevant, as we have just 16 particles with sidelength of 40. In real life, with certainly more and possibly bigger particles you might want to use the parameter “ccmatrix_batch” to force the creation of more blocks each one tackling less particles at a time.

```
Session Edit View Bookmarks Settings Help

Starting ccmatrix computations:

A total of 1 blocks will be distributed among 1 processors.
Each block will occupy up to 375.00 Mb in memory
Computing block (1,1) with block number 0 assigned to processor 0... took 5.13 seconds (46.88 Mb)
Rough estimation for total computing time expected in processor 0: 5s
matrix contribution from this processor (0) written in file ./pcc/temp/ite_0001/intermediate_results/proc_c
ontribution_ccmatrix_proc_00000_ref_001_ite_0001.em
NO DISPLAY
result: 0 (hopefully 0) from submitting dynamo_ccmatrix_compute to system from OpenMP wrapper program dy
namo_ccmatrix_compute_omp.cc
OMP finished. All computations done.
Initializing MATLAB Compiler Runtime version 7.14
Starting Up:"MCR libraries starting dynamo_ccmatrix_assemble. [kernel function]"
reading contribution ./pcc/temp/ite_0001/intermediate_results/proc_contribution_ccmatrix_proc_00000_ref_001
_ite_0001.em to the cross correlation matrix

CCMATRIX:
stored as "ccmatrix" in the database, file:
./pcc/results/ite_0001/ccmatrix/ccmatrix_ref_001_ite_0001.em

Assembling of cross correlation matrix finished at 15-Aug-2012 12:33:08
Actions related to ccmatrix computation and analysis in this round:
align ;

Done with ccmatrix-related computations of project pcc, ref 1 ite 1,

NO DISPLAY
[casdanie@cina-hpws01 testcc]$
```

```
Session Edit View Bookmarks Settings Help

-----
Executing: vpr_ccinfo

Parsing 1 arguments
-----

Project: "pcc" (iteration: 1)
project type : ccmatrix
-----

Parameters in round 1
ccmatrix      : 1
ccmatrix_type : align ;
ccmatrix_batch : 128
Xmatrix       : 0
Xmatrix_maxMb : 100
PCA           : 0
PCA_neigs     : 4
kmeans        : 0
kmeans_ncluster : 2
kmeans_ncoefficients : 3
-----

Computed files
tags file      : 16 tags
ccmatrix file  : 16 X 16
ccmatrix actions : align ;
Xmatrix (blocks) : no blocks of Xmatrix available
[Attention]: 16 tags registered in file, but the Xmatrix blocks covers only 0
             You will not be able to run a PCA until you create a suitable Xmatrix.
             As a ccmatrix already exists, you can just switch on the parameter "reuse_ccm"
             and run dynamo_vpr_ccmatrix on this project.
Eigenvolumes   : not available
eigentable     : not available
Subaverages    : not available
-----

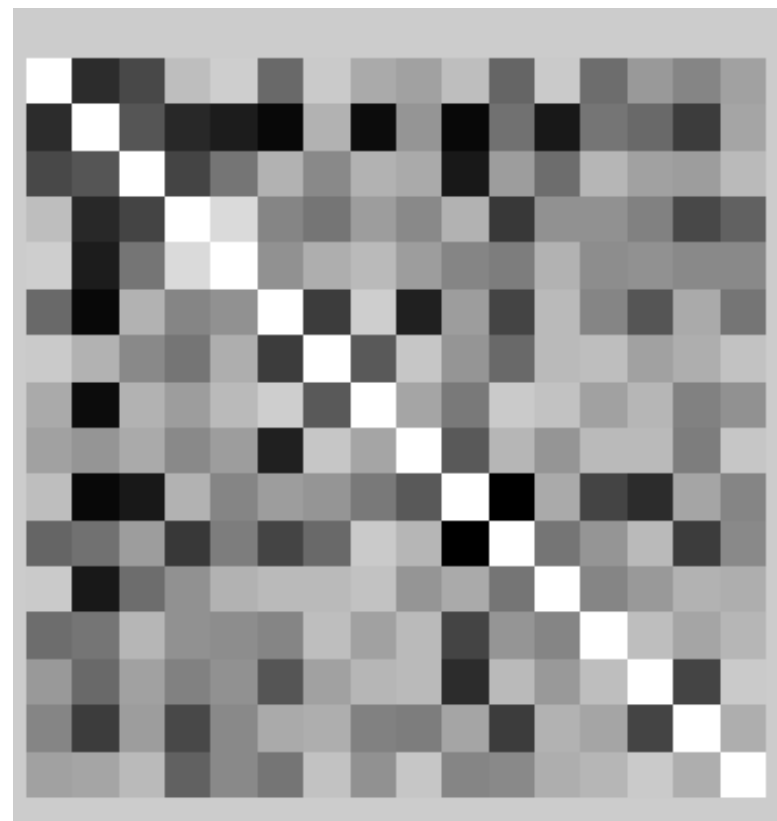
Exiting: vpr_ccinfo
-----

Dynamo > >> 
```

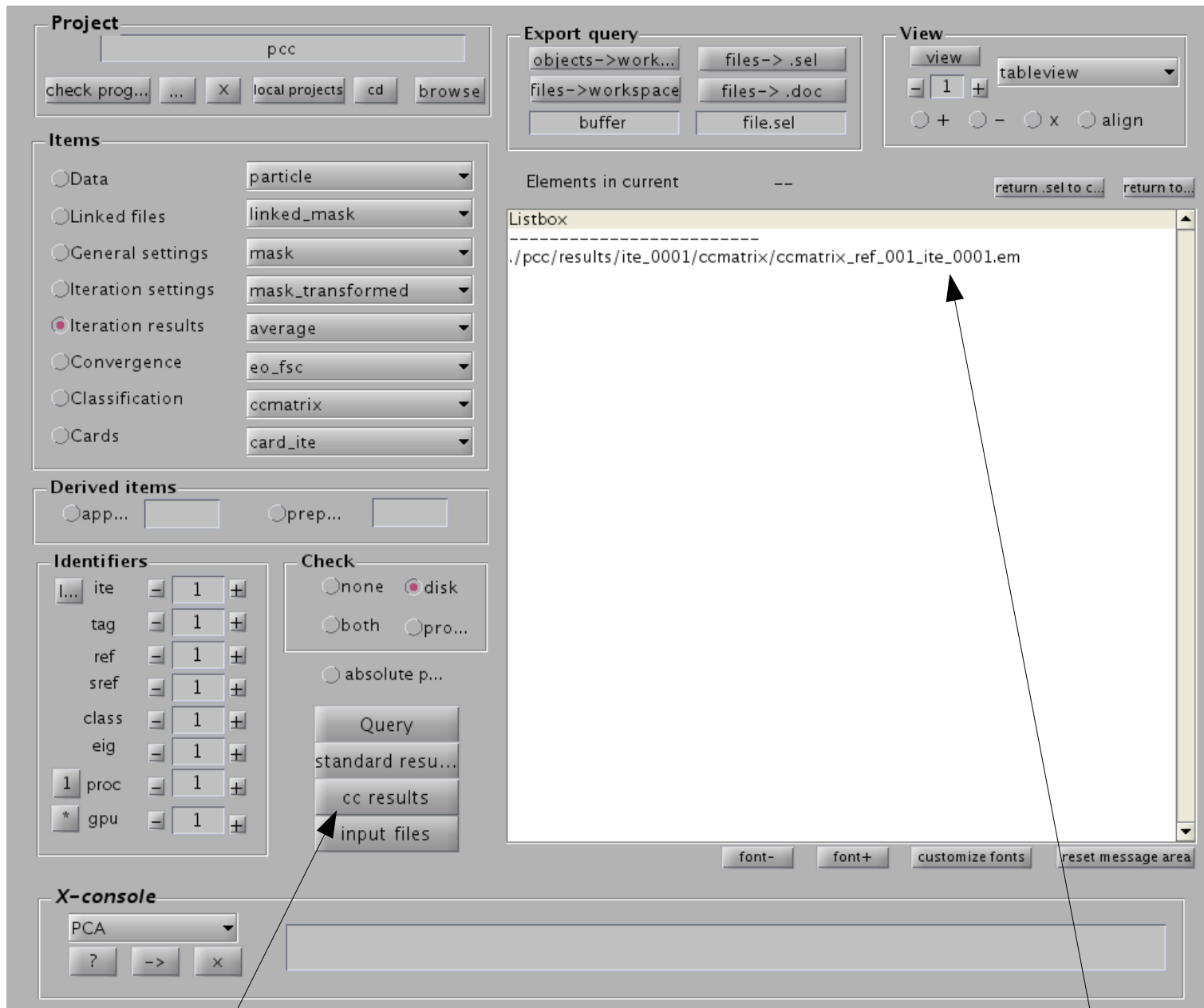
this are does not change:
they are just settings

but we start to have results stored
in the database of the project

—



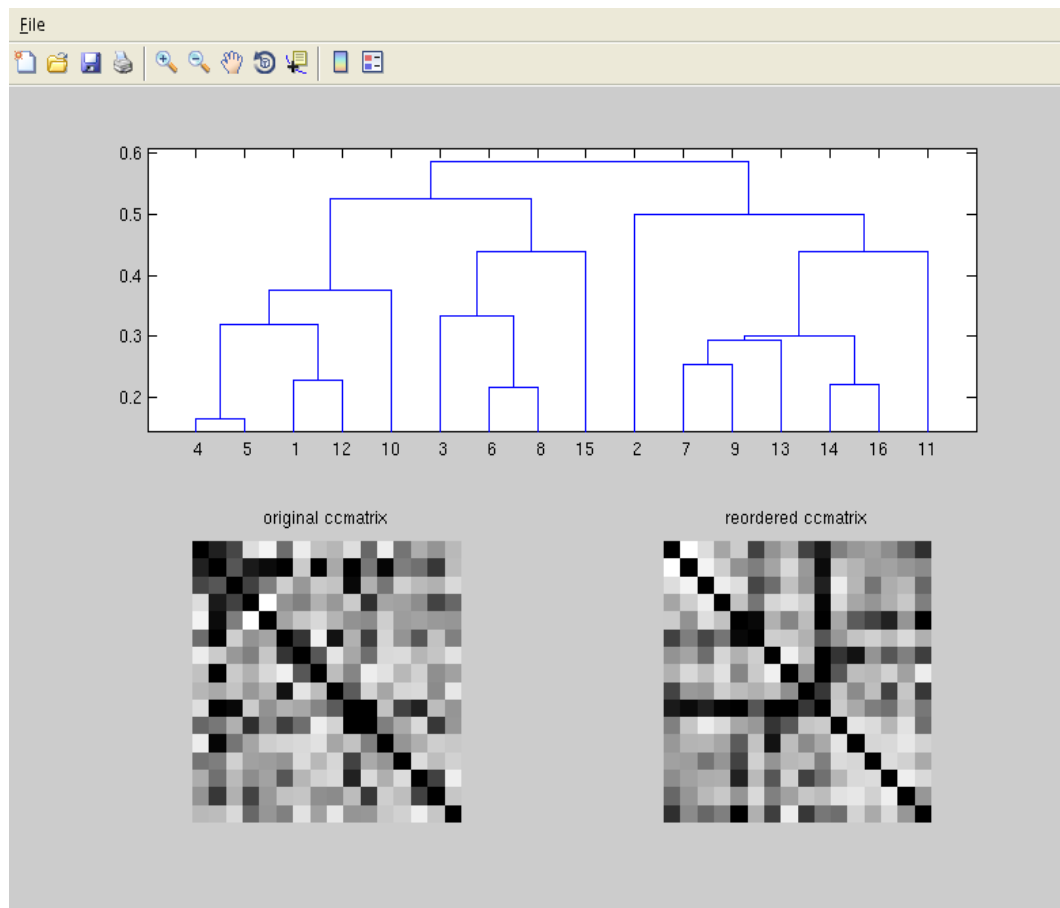
... where



click here to get the results related to classification tasks currently in the project. In this case, just the ccmatrix is retrieved

A first attempt with an easy clustering procedure (just Hierarchical Ascending, no PCA) does not show a very clear separation:

```
>> ddendrogram -ccm pcc:ccm
```



... so we continue with a full PCA analysis

Having computed a ccmatrix we can move to the next step towards PCA classification.

The first step is the computation of an Xmatrix object. This is actually just an operational non interesting step. The reason to give the user the possibility of computing it explicitly is merely technical: An Xmatrix can be really huge (as it contains all available voxels in the data set), and operating with it might require to chunk it in subblocks that are tractable in the memory.

This is steered with parameter 'MaxMb'. But in our case we can just use the default parameters.

Additionally, we can tell Dynamo to just continue with the computations inside the project, by:

```
>> dPCA_Xmatrix -p pcc
```

```
PCA_Xmatrix
```

```
Memory to allocate for Xmatrix object: 3.50 Mb
```

```
  Xmatrix block number #1. 1 particles read and processed in 0.83 seconds.
```

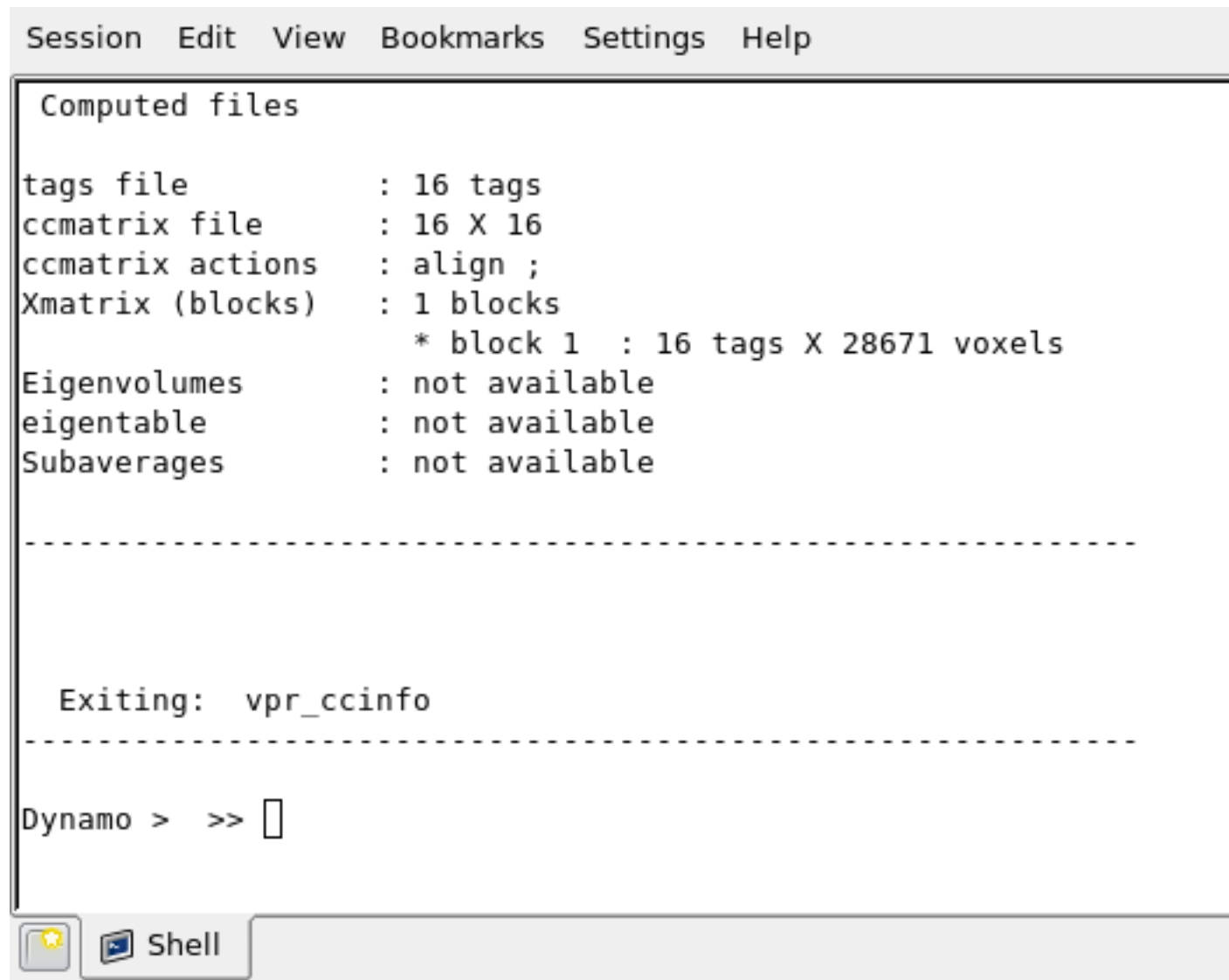
```
  Still processing 15, should take 12.49 seconds
```

```
Warning writing .em file ./pcc/temp/ite_0001/Xmatrix_ref_001_ite_0001.em, empty obje
```

```
Xmatrix stored in 1 blocks as ./pcc/temp/ite_0001/Xmatrix_ref_001_ite_0001_block_*
```

```
[ok]    PCA_Xmatrix completed
```

again, we can run a `dvccinfo` on project `pcc` afterwards to check how things are going:



The screenshot shows a software window with a menu bar (Session, Edit, View, Bookmarks, Settings, Help) and a main text area. The text area displays the output of a `dvccinfo` command. The output lists various computed files and their properties, followed by a dashed line and the text "Exiting: vpr_ccinfo". Below this, the prompt "Dynamo > >> " is shown. At the bottom, there is a tab labeled "Shell" with a small icon.

```
Session Edit View Bookmarks Settings Help

Computed files

tags file          : 16 tags
ccmatrix file      : 16 X 16
ccmatrix actions   : align ;
Xmatrix (blocks)   : 1 blocks
                   * block 1 : 16 tags X 28671 voxels
Eigenvolumes       : not available
eigentable         : not available
Subaverages        : not available

-----

Exiting: vpr_ccinfo

-----

Dynamo > >> 
```

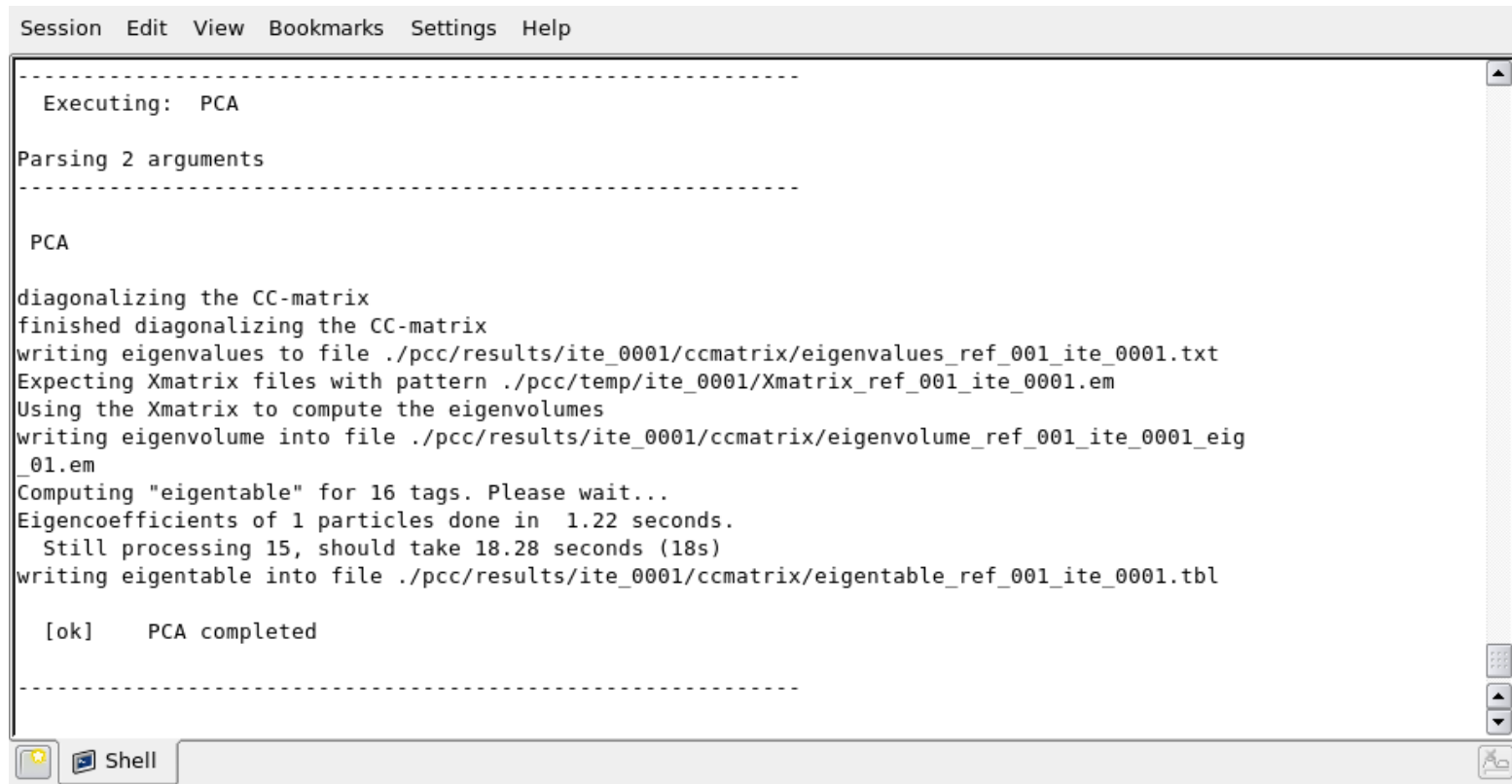
so, we can now create the PCA (eigenvolumes/eigentable)
and then the class averages (called “subaverages”)

Again, PCA computations might accept different modifications:

- actions on the particles (bin, sym...)
- number of eigenvalues

But we will proceed with the default settings, and proceed inside the project, so we can simply write:

```
>> dPCA -p pcc;
```



```
Session Edit View Bookmarks Settings Help

-----
Executing: PCA
Parsing 2 arguments
-----

PCA

diagonalizing the CC-matrix
finished diagonalizing the CC-matrix
writing eigenvalues to file ./pcc/results/ite_0001/ccmatrix/eigenvalues_ref_001_ite_0001.txt
Expecting Xmatrix files with pattern ./pcc/temp/ite_0001/Xmatrix_ref_001_ite_0001.em
Using the Xmatrix to compute the eigenvolumes
writing eigenvolume into file ./pcc/results/ite_0001/ccmatrix/eigenvolume_ref_001_ite_0001_eig_01.em
Computing "eigtable" for 16 tags. Please wait...
Eigencoefficients of 1 particles done in 1.22 seconds.
  Still processing 15, should take 18.28 seconds (18s)
writing eigtable into file ./pcc/results/ite_0001/ccmatrix/eigtable_ref_001_ite_0001.tbl

[ok]   PCA completed
-----

Shell
```

Here it will take some seconds... in real life this can be quite computing intensive, but it will rarely grow to a bottleneck.

```

kmeans_ncoefficients : 3
-----

Computed files

tags file           : 16 tags
ccmatrix file       : 16 X 16
ccmatrix actions    : align ;
Xmatrix (blocks)    : 1 blocks
                    * block 1 : 16 tags X 28671 voxels
Eigenvolumes        : 10 eigenvolumes
                    * eig #1  : 40 X 40 X 40
eigentable          : 16 particles X 50 columns (10 eigencomponents)
Subaverages         : not available
-----

```

dvccinfo informs us that the PCA elements are indeed in place.

Before proceeding with the classification itself we analyze the obtained elements

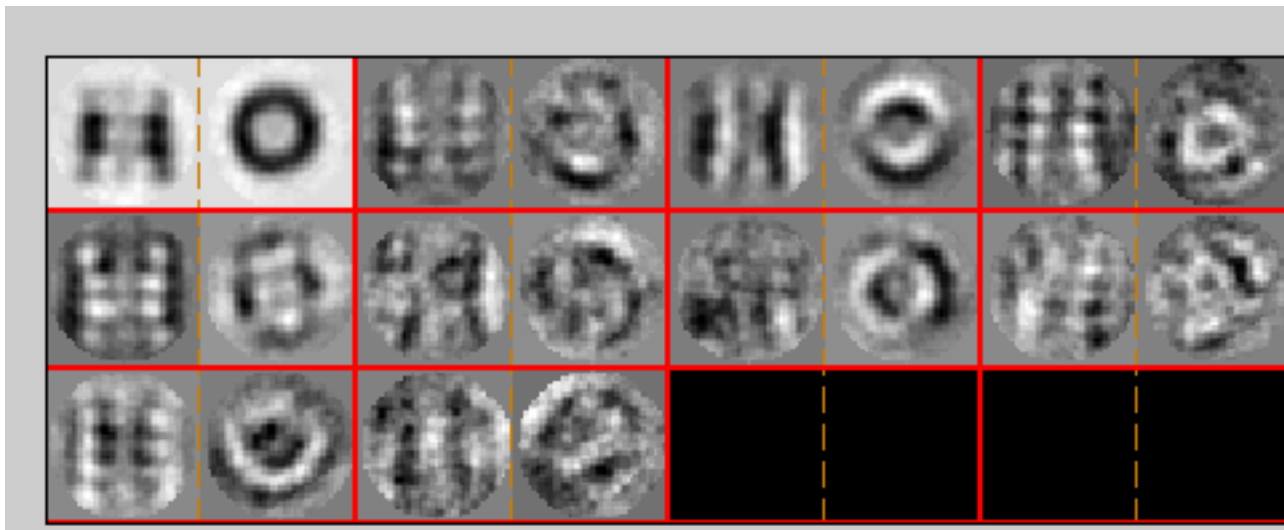
Let us take a look on the eigenvolumes.

An useful procedure is using a database query to dump a set of files into a .sel file (the .sel file is just a text file that lists other files)

```
>> ddb pcc:eigenvolume:eig=* -sel my_eigvs
```

So that now we can take a look on the created eigenvolumes:

```
>> dslices my_eigvs.sel x|z -j c2 -ns true
```



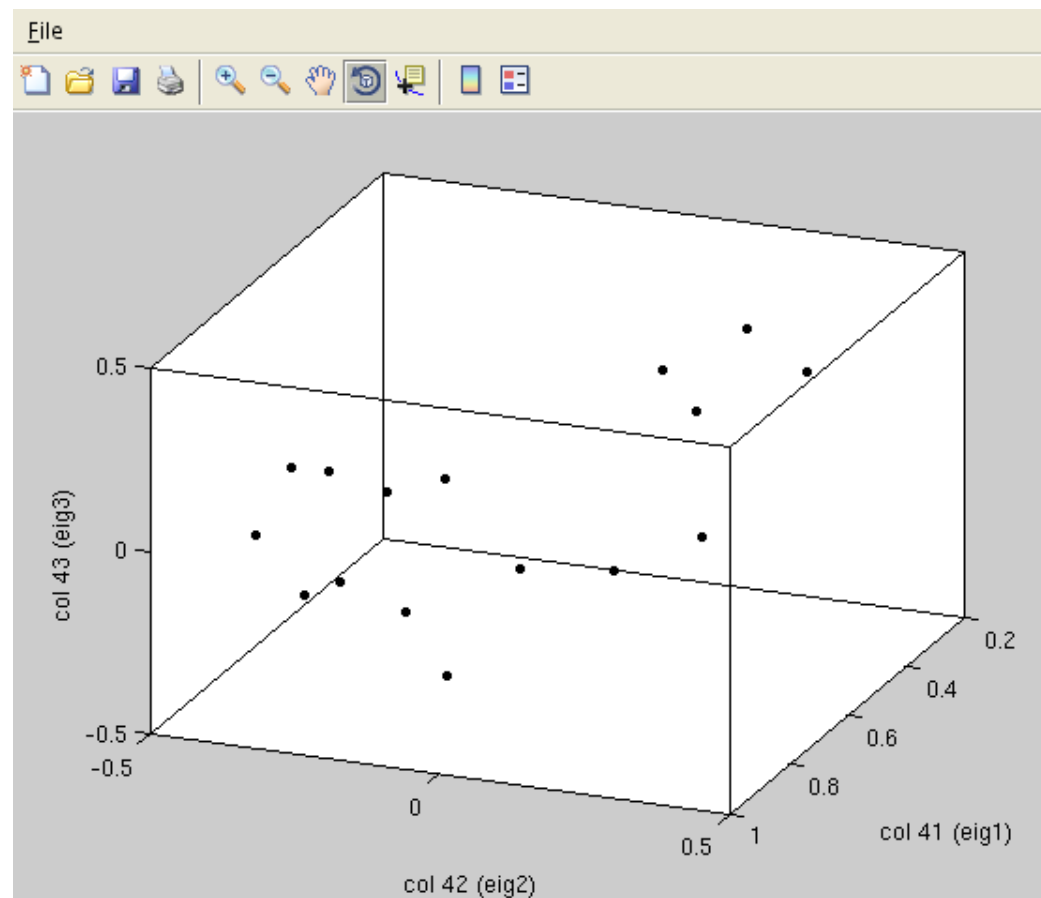
If you really hate the ddb-style command, remember that you can use other less obscure options (desktop, vpr_results, ccmatrix_analysis) to access the results of a computation stored in the database of a project using more intuitive syntax (or graphic interfaces)

The “eigentable” records the components of each particle along each vector.

Scatterplots are normally useful for depictions.

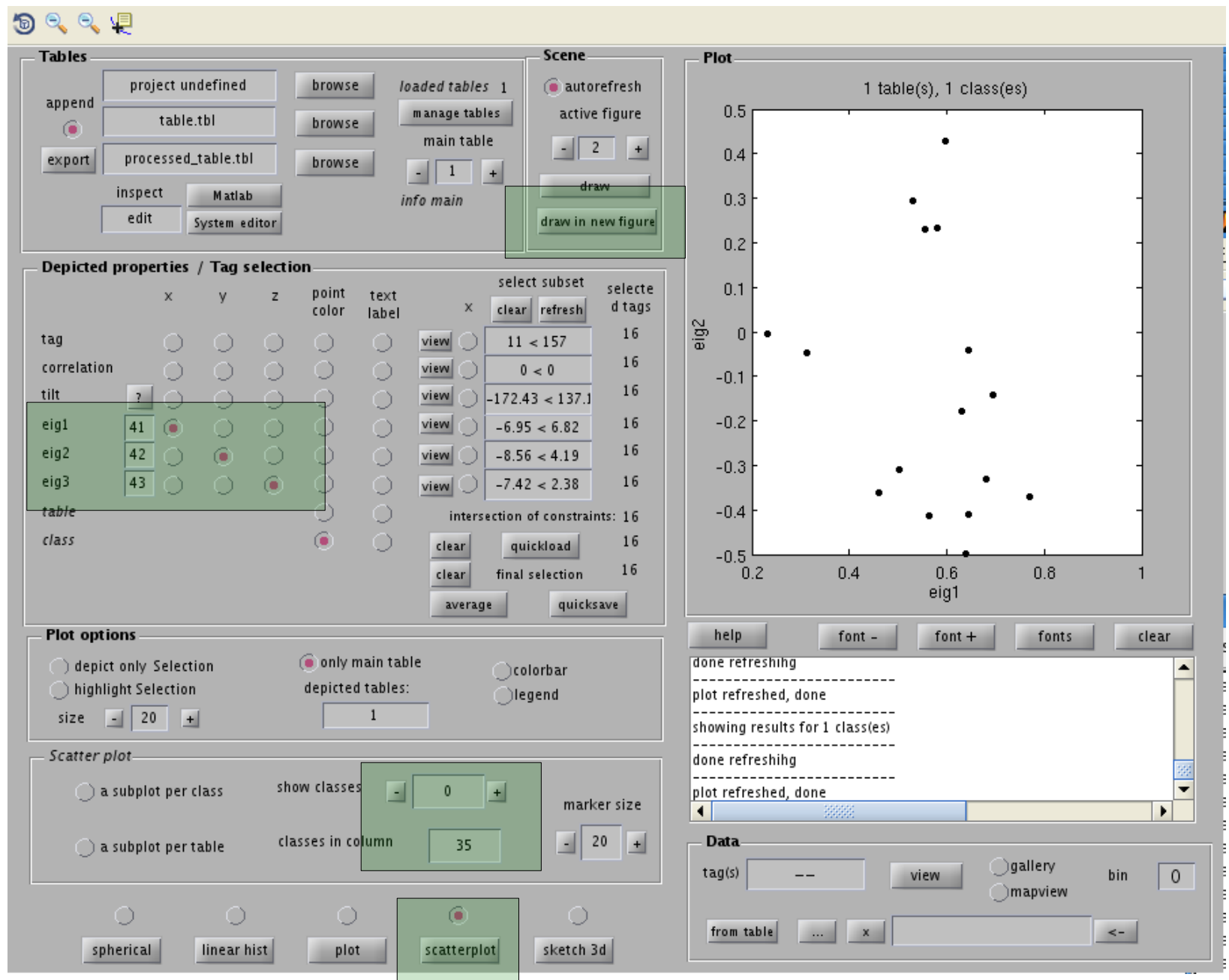
The command line tool for table depiction includes a “profile” called “eigenvalues” for this task:

```
>> dtplot pcc:eigentable -pf eigenvalues
```



... although it fighting a little bit with the interactive tool `tableview` to get this representation might also prove useful

```
>> dtview -t pcc:eigentable;
```



In any case, the visualization of both eigenvectors and eigenvalues does not look specially promising (they rather suggest misalignment issues).

We still proceed with our classification performing a kmeans classification on the PCA coordinates of the particles.

As in the case of dXmatrix and dPCA, dkmeans can read settings from and write results into a project, or also accept manual input for specific modifications.

We will let dkmeans use the project to automatize input and output, but specifically request that we want two clusters.

```
>> dkmeans -p pcc -n 2 -reps 100;
```

The parameter “reps” tells kmeans to repeat the classification 100 times, as the seeds for the classification are randomly generated. We want to get a representative classification.

```

Deleting previous subaverages:
./pcc/results/ite_0001/ccmatrix/subaverage_i
./pcc/results/ite_0001/ccmatrix/subaverage_i
Going for subaverage associated to (new) mer
Output will be written in ./pcc/results/ite_
fmask compensation method: "table"
  Table contained 11 particles; averaging 11

11,12,60,77,88,95,97,104,131,154,157,

Computing relative weights of fourier compon
applying compensation of missing wedge/pyram
Going for subaverage associated to (new) mer
Output will be written in ./pcc/results/ite_
fmask compensation method: "table"
  Table contained 5 particles; averaging 5 ta

42,49,61,87,118,

Computing relative weights of fourier compon
applying compensation of missing wedge/pyram
done with subaverages

[ok]    kmeans completed

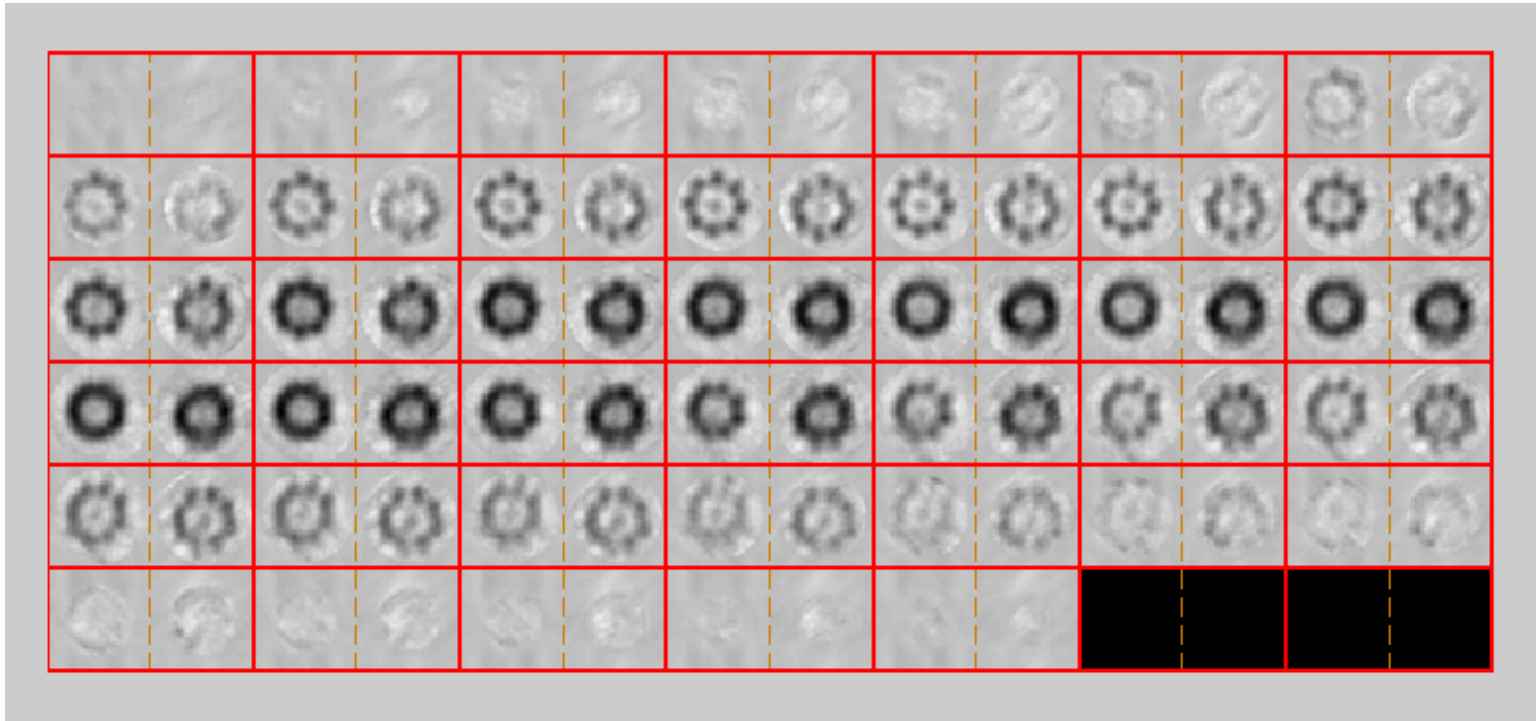
```

actually, you see that the classification is not so good: in the two classes that we construct, one has 11 particles and the other one 5.

But we know we should have 8 and 8 by construction.

Let us first take a look onto the results

```
>> ddb pcc:subaverage:sref=* -m
```



By construction in the tutorial, we know that one class gathers “small” particles and the other “big” particles.

This trend is however not clear by inspection of the two subaverages, which rather appears to have classified particles according to slight orientational changes: as we suspected, we get misalignment issues.

.....so ... what can we do?

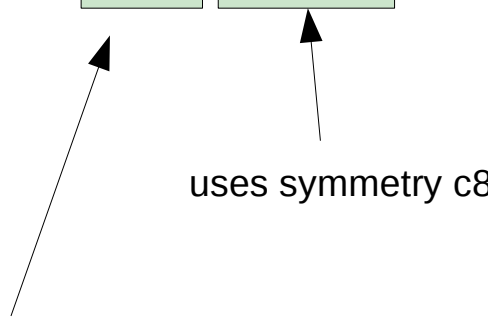
Changing the classification parameters

We can make our classification more robust by using more information.

If we know (or want to use the hypothesis) that particles have a C8 symmetry, we can use that data all over the classification.

This time we will produce the classification in one step, as we are already familiar with the pipeline:

```
dvccmatrix pccs -t tut/coarse.tbl -d tut/data -m tut/mask.em  
-s cu -sym c8 -destination system_omp -steps all
```



just to “save” the project in modus “cu”, meaning:

- “check” (stops if errors are found)
- and “unfold” (creates an project ready for execution)

we let the project compute at once:

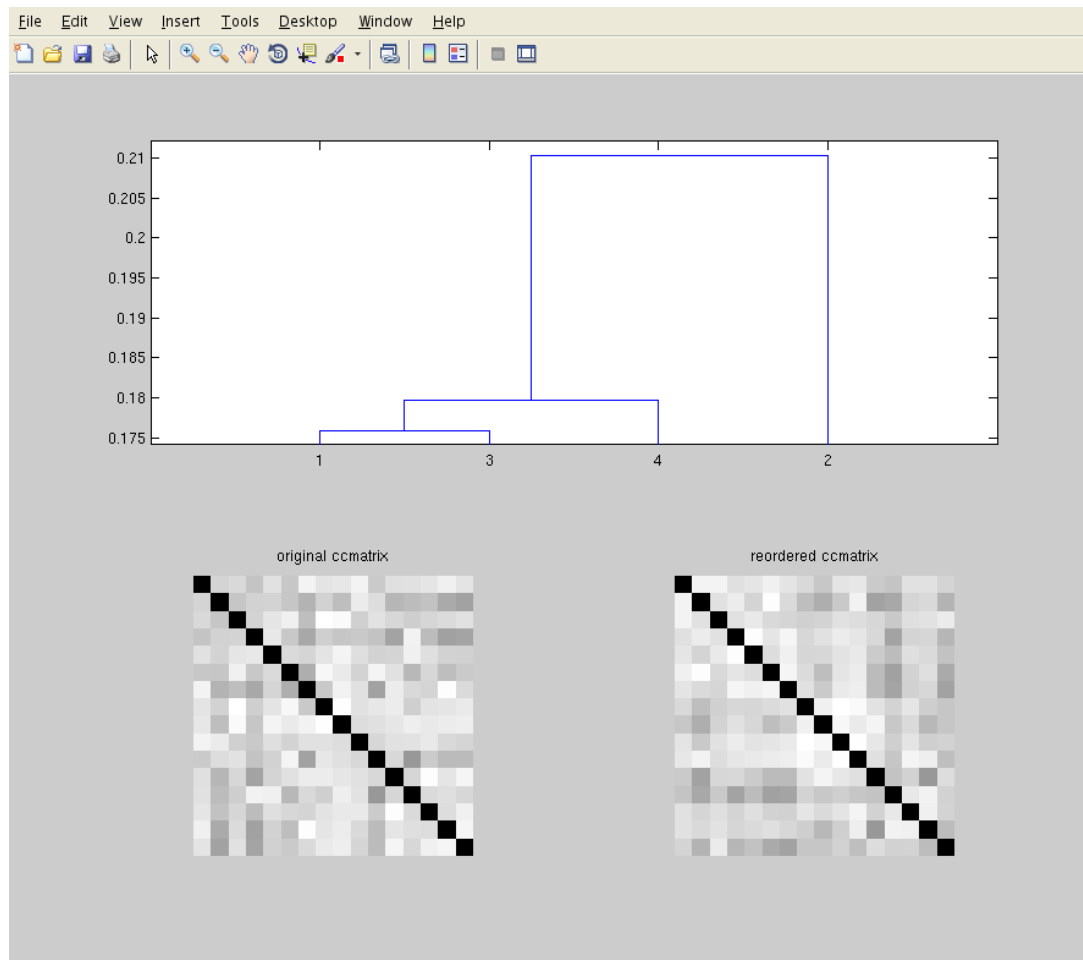
- ccmatrix
- PCA_Xmatrix
- PCA (eigenvalues, eigenvectors and eigentable)
- kmeans classification

If the previous order worked, you can now execute the produced execution script in a system shell

Did the use of symmetry give a better classification?

A first hint is that the matrix-based clustering can now give a better (although not really definitory) result:

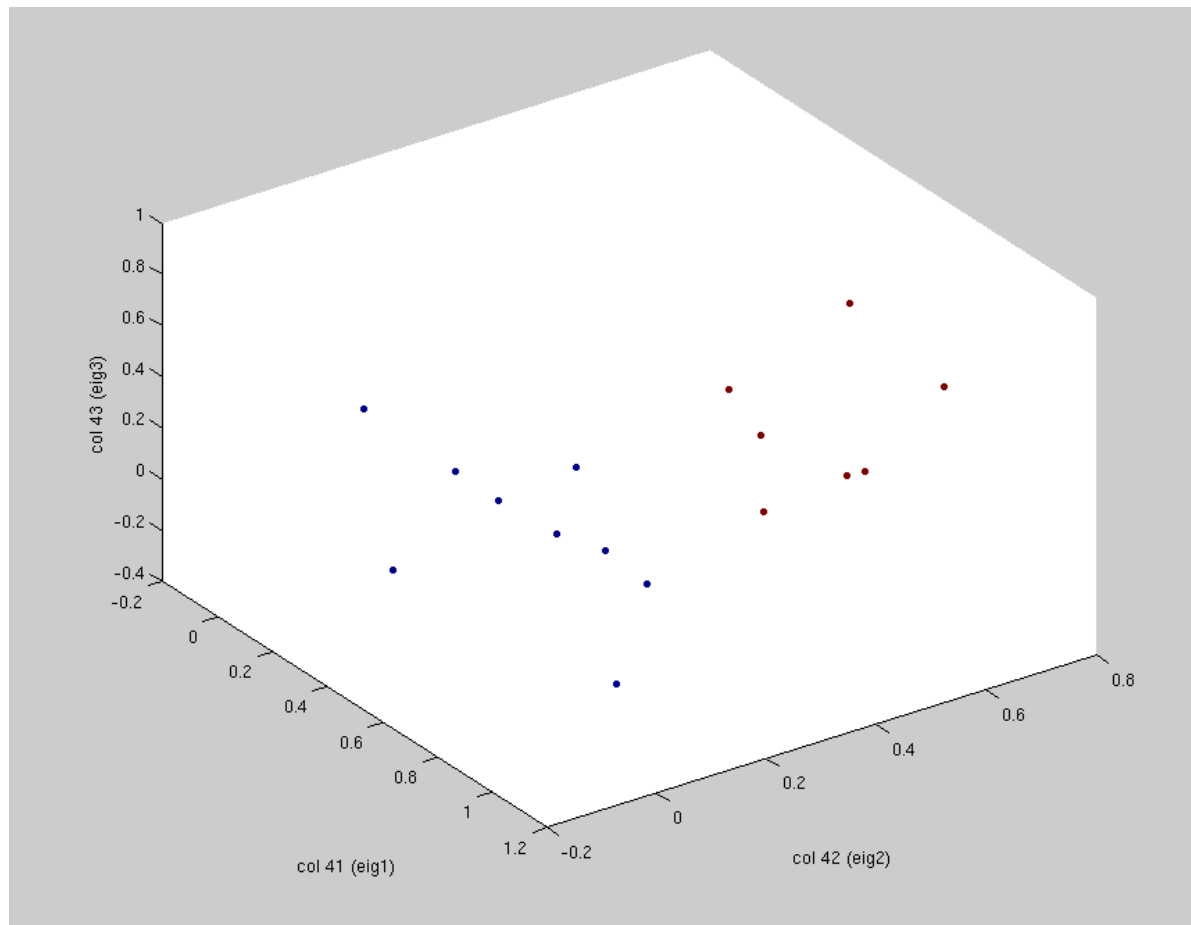
```
>> ddendrogram -ccm pccs:ccm -p 4
```



... but we would expect that PCA should give better results than simple clustering so that let us go for it

The resulting clustering looks better:

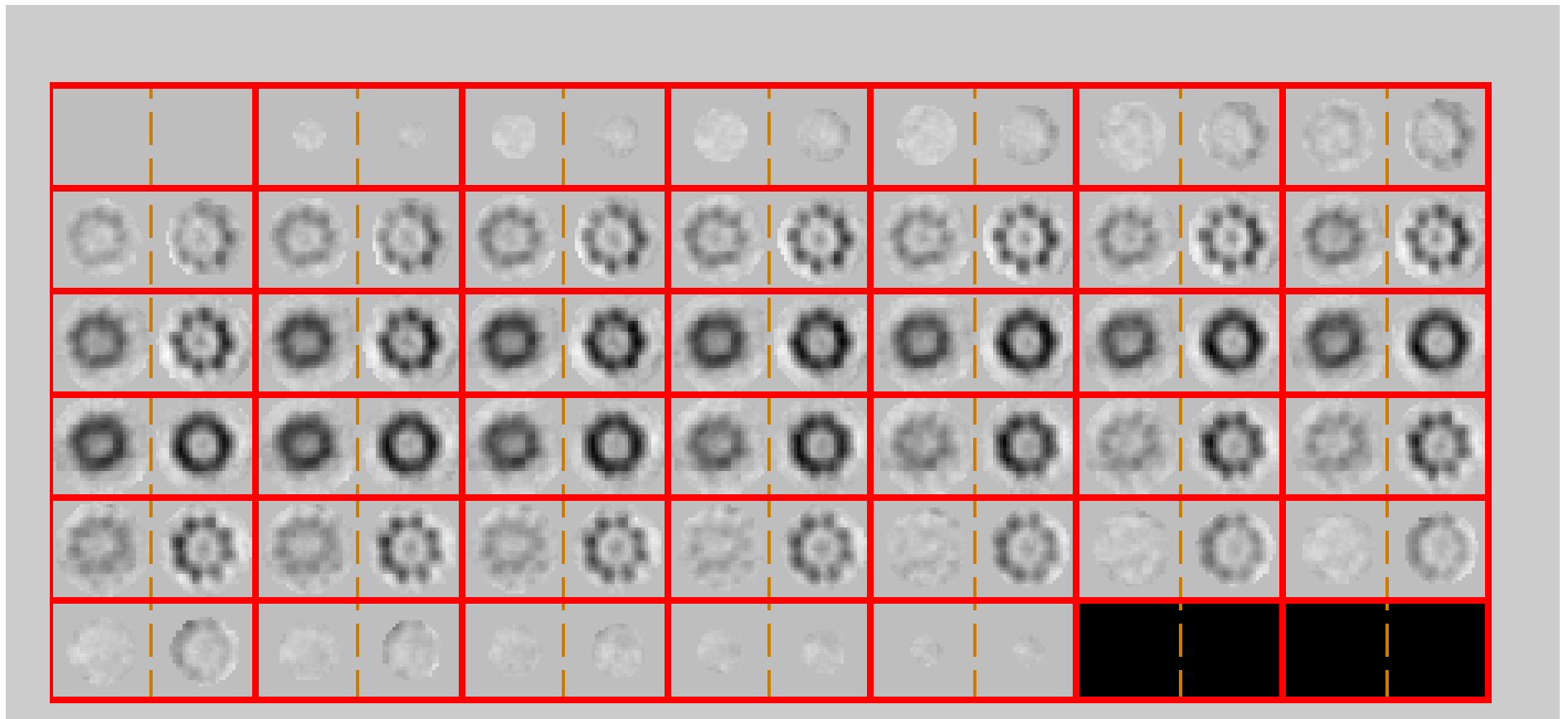
```
>> dtplot pccs:eigtable -pf eigenvalues
```



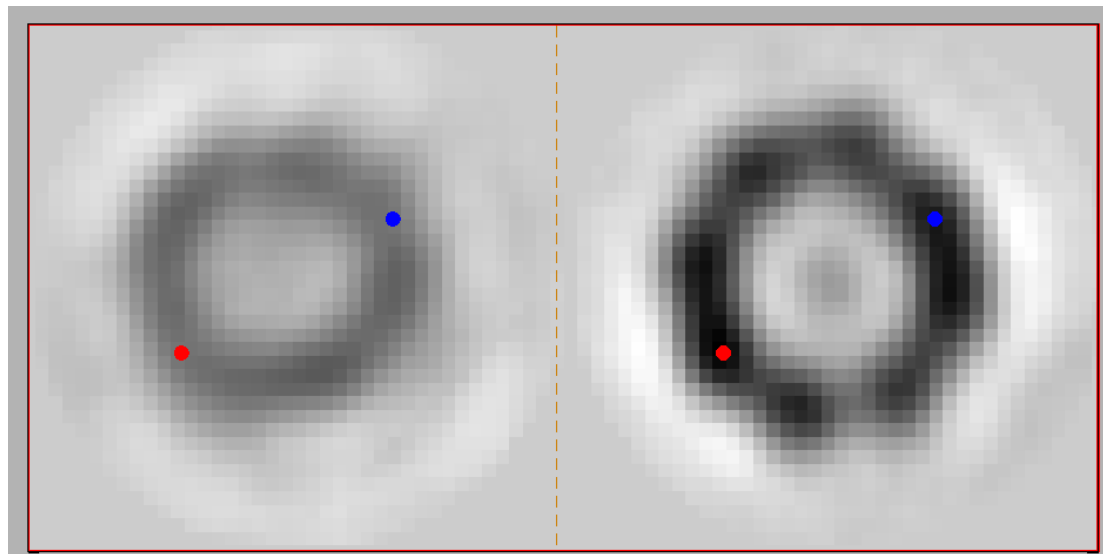
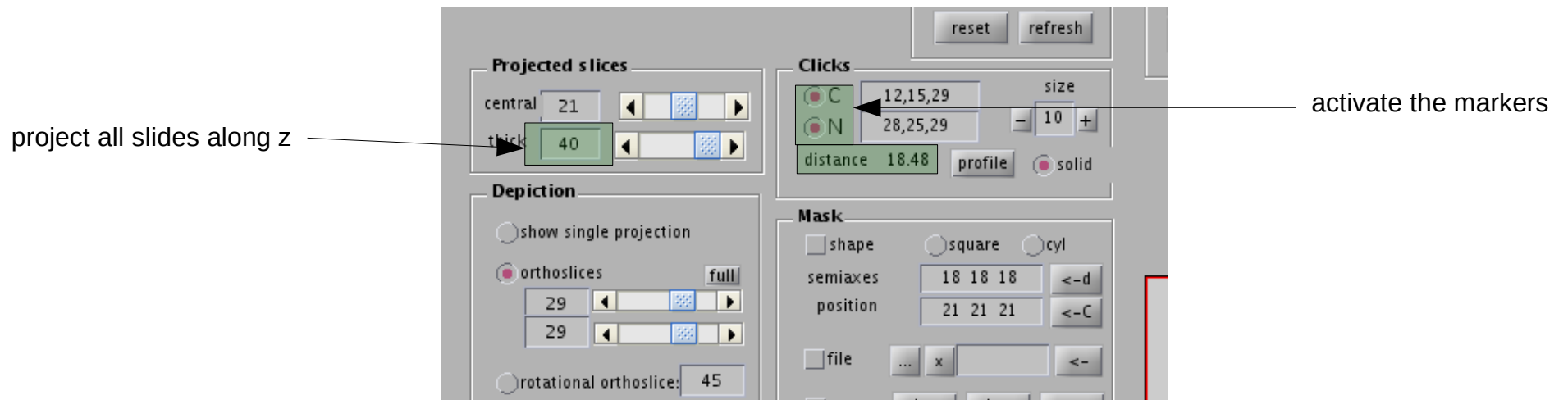
... well, the automated profile “eigenvalues” in `dtplot` does not give a very visual depiction...
you might want to explore the other parameters of `dtplot` or use `dtview` for a more aesthetic depiction

Now it looks like the classification made a better job:
A class appear blurrier than the other , but the main features can be distinguished

```
>> dmapview pccs:subaverage:sref=*
```



you can check this more accurately with the C and N markers in mapview by clicking on screen:



Part III

“project sourcing”: Refining a classification

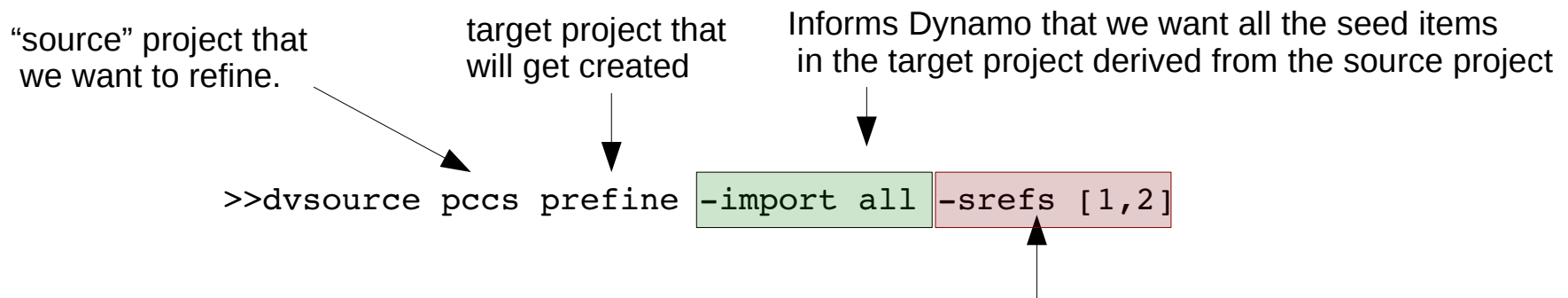
An obvious thing to do would be to align the classes separately.

In this part we see how to create alignment projects that fulfill this task in an automated way.

Creating alignment projects ab initio can be tedious.

“Sourcing” is one of the techniques to construct new projects from results or settings other “source” projects.

In our case, we can just write:



“srefs” is the parameter telling Dynamo that we want to refine a classification.

In our original project “pccs” we were analysing a single reference channel, and we produced there two “subreferences” (which we can assimilate to the concept of “class averages”).

With this syntax, Dynamo will know that we want to

- 1) Use the subaverages in the original source project as initial templates in target project (also, “subtables” will be used as initial tables, etc)
- 2) We asked to continue the refinement of two subreferences in a single target alignment project
Thus, this target alignment project will be of multireference type.

```

-----
Importing "subtable" from source  to be used as "table_initial" in target

    source: "subtable"  ite:1  ref:1 sref:1 --> target: "table_initial"  ref:1
... copied
    source: "subtable"  ite:1  ref:1 sref:2 --> target: "table_initial"  ref:2
... copied

-----
Importing "subaverage" from source  to be used as "template_initial" in target

    source: "subaverage"  ite:1  ref:1 sref:1 --> target: "template_initial"  ref:1
... copied
    source: "subaverage"  ite:1  ref:1 sref:2 --> target: "template_initial"  ref:2
... copied

-----
Importing "subfmask" from source  to be used as "fmask_initial" in target

    source: "subfmask"  ite:1  ref:1 sref:1 --> target: "fmask_initial"  ref:1
... copied
    source: "subfmask"  ite:1  ref:1 sref:2 --> target: "fmask_initial"  ref:2
... copied

```

Note this screen capture from the command `dvsources`, informing which items from the database of the “source” project have landed as which items in the database of the “target” alignment project.

Note that a “source” project can be any kind of project: alignment, classification, single or multireference....

Now `prefine` has the correct “seed” files (data, templates, etc, etc) but we still need to input the correct numeric settings (angles, symmetries, etc) because the project has been created with default settings.

In this case, we now that we just want to slightly refine the alignment parameters in the table: for this task the provided parameters are an overkill: the project will be way too time consuming.

You can check the parameters with `dvinfo`

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
SETTINGS: summary
DYNAMO PARAMETER      :   round 1  round 2  round 3  round 4  round 5
-----
"ite"                  :         4      4      4      4      4
-----
"cone_range"           :       60.0    20.0    10.0     5.0     1.0
-----
"cone_sampling"         :       15.0    10.0     5.0     2.0     0.5
-----
"inplane_range"         :       60.0    20.0    10.0     5.0     1.0
-----
"inplane_sampling"      :       15.0    10.0     5.0     2.0     0.5
-----
"high"                 :         1      1      1      1      1
```

You probably know the GUI `dynamo_project_manager` already, where you can easily modify an existing project. But in this tutorial, we use the command line tools for this task

```
>> dvput prefine cd -ite [2,0,0,0,0,0,0];
```

```
Dynamo > >> dvput prefine cd -ite [2,0,0,0,0,0,0];  
"dvput" references source command "vpr_put"
```

```
-----  
Executing: vpr_put
```

```
Expanding:
```

```
ite
```

```
[modify] flag #1: "ite_r1" (valid round project parameter)
```

```
[modify] value #1: 2 (numeric)
```

```
[modify] flag #2: "ite_r2" (valid round project parameter)
```

```
[modify] value #2: 0 (numeric)
```

```
[modify] flag #3: "ite_r3" (valid round project parameter)
```

```
[modify] value #3: 0 (numeric)
```

```
[modify] flag #4: "ite_r4" (valid round project parameter)
```

```
[modify] value #4: 0 (numeric)
```

```
[modify] flag #5: "ite_r5" (valid round project parameter)
```

```
[modify] value #5: 0 (numeric)
```

```
[modify] flag #6: "ite_r6" (valid round project parameter)
```

```
[modify] value #6: 0 (numeric)
```


```
[modify] flag #7: "ite_r7" (valid round project parameter)
```

```
[modify] value #7: 0 (numeric)
```

```
ok, project "prefine" seems safe enough.
```

Now we modify the angular settings, the symmetrization, a multigrid parameter and the running environment:

>> dvput predefine **cu** **-cr 20 -cs 5 -ir 20 -is 5** **-sym c8 -rff 2** **-destination system_omp;**



...and check and unfold
project predefine after modification

note that dvput:

- 1) understands also the shortforms of the parameters
i.e. "cr" gets translated into "cone_range"
- 2) assumes iteration 1 as default

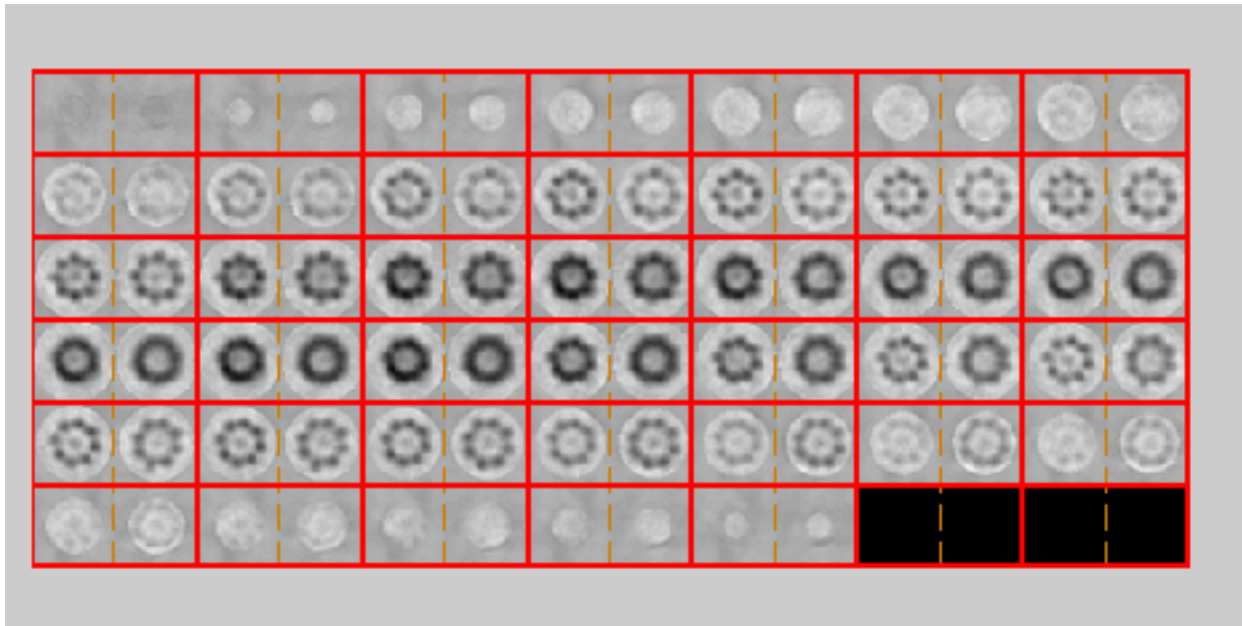
```
Session Edit View Bookmarks Settings Help
Dynamo >
>> dvput predefine cu -cr 20 -cs 5 -ir 20 -is 5 -sym c8 -rff 2 -destination system_omp;
"dvput" references source command "vpr_put"

-----
Executing: vpr_put

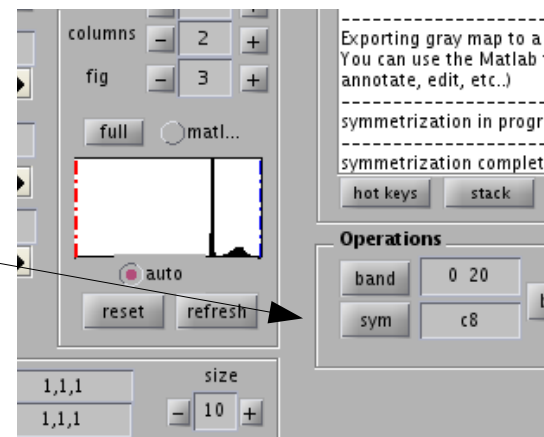
[modify] flag #1: "cone_range_r1" (valid round project parameter)
[modify] value #1: 20 (numeric)
[modify] flag #2: "cone_sampling_r1" (valid round project parameter)
[modify] value #2: 5 (numeric)
[modify] flag #3: "inplane_range_r1" (valid round project parameter)
[modify] value #3: 20 (numeric)
[modify] flag #4: "inplane_sampling_r1" (valid round project parameter)
[modify] value #4: 5 (numeric)
[modify] flag #5: "sym_r1" (valid round project parameter)
[modify] value #5: c8 (char)
[modify] flag #6: "refine_factor_r1" (valid round project parameter)
[modify] value #6: 2 (numeric)
[modify] flag #7: "destination" (valid project parameter)
[modify] value #7: system_omp (char)
```

The two averages appear now much better defined

```
>> ddb pccs_refine:a:ref=* -m
```



If we symmetrize the representation:





... the size effects are easier to recognize...

you can for instance check
the intensity profiles along
homologous regions

Note the settings for viewing:
slice 16, viewing direction y, etc

