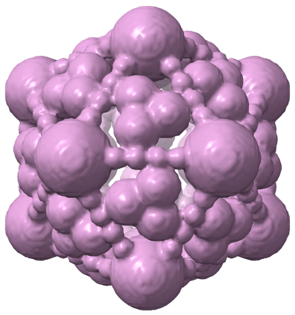
Scipion Developers Course

NATIONAL CENTER FOR BIOTECHNOLOGY

BIOCOMPUTING UNIT

Scipion Developers Course: Sets of Particles

October 29, 2020



3D phantom of an icosahedral virus.

ROBERTO MARABINI

Revision History

Revision	Date	Author(s)	Description
1.0	10.15.2020	RM	created for first developers workshop
1.1	10.29.2020	RM	Fixed a few inconsistencies detected in the workshop

Contents

1 Setup and requeriments **3**

1.1 Intended audience . . . 3

1.2 We'd like to hear from you . . . 3

1.3 Requirements . . . 3

2 Practical Session **3**

2.1 Objectives and Introduction . . . 3

2.2 The data . . . 4

2.3 Playing with the images . . . 5

2.4 Create a protocol that compute the mean and variance of a set of images 6

2.5 Protocols that require image format conversion . . . 9

Appendices **12**

A Computing Sample Mean and Variance in One Pass **12**

1 Setup and requeriments

1.1 Intended audience

This guide has been tailored to developers who wish to program using the *Scipion* framework and have basic knowledge of software architecture and Python programming language.

1.2 We'd like to hear from you

We have tested and verified the different steps described in this guide to the best of our knowledge, but since our programs are in continuous development you may find inaccuracies and errors in this text. Please let us know about any errors, as well as your suggestions for future editions, by writing to scipion@cnb.csic.es.

1.3 Requirements

This tutorial requires *Scipion* with the *xmipp_bundle* installed. Basic knowledge of python is assumed.

2 Practical Session

2.1 Objectives and Introduction

In this practical session you will get familiar with the abstraction layer that provides to *Scipion* developers with an unified interface for accessing and manipulating an image (or set of images). This layer is responsible for translating the commands provided by the programmer into the specific format dependent commands needed by each particular image format.

The session has been divided in two parts. In the first one you will learn how to manipulate images and in the second we will address the problem of file format conversion.

Before going ahead I would like to warn you that, in *Scipion*, the word "image" is used in different contexts with three different meanings:

- Binary file storing the value of a collection of pixels.
- Object of the class `Image` containing metada related to a "binary file".
- Object returned by the class `ImageHandler` that allows *Scipion* developers to access to a "binary file" using an API which does not depend on the file format.

2.2 The data

In this practical session we are going to use a $120 \times 120 \times 120$ phantom (an artificial object) and 10 noisy projections along the same projection direction. Images are in the directory `data` (from the plugin-em-template root).

- **virus.map:** 3D phantom, see surface rendering at figure 1
- **proj.stk:** 10 noisy projections at the same angle figure 2

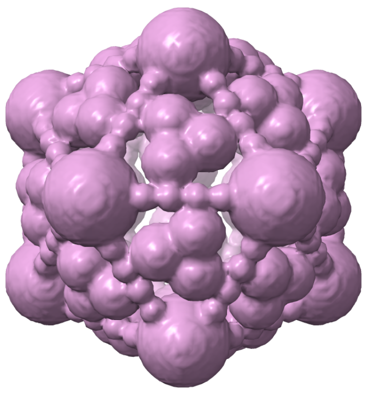


Figure 1: surface rendering of a 3D phantom

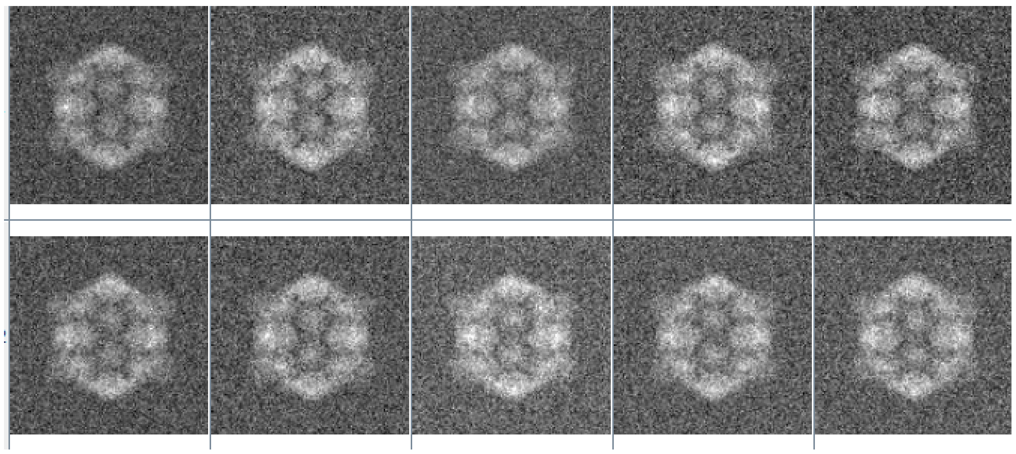


Figure 2: Gallery of 10 noisy projections along the same projection direction.

2.3 Playing with the images

Before creating a real protocol let us play a bit with the image API. In an object of type `Image` you can only store metadata related with the image as can be the name of the binary file in which the pixel values are stored, the sampling rate, information related with alignment, etc. Let us add the sampling rate and the name of the binary file to the `Image` object.

Listing 1: The following script creates an object of the class `Image`.

```
$ scipion3 python # start python inside scipion3
>>> from pwem.objects.data import Image
>>> i = Image()
>>> i.setFileName('myImage.map')
>>> i.setSamplingRate(3.) # sampling rate in A
>>> for a in i.getAttributes(): # print all the pairs (key, value)
                                # stored in the Image i
...     print(a[0], a[1].get())
_index 0
_filename myImage.map
_samplingRate 3.0
```

On the other hand the class `ImageHandler` actually accesses binary files and can modify them. In the following example we are going to access a file and modify its content.

Listing 2: The following script creates a 3×3 binary image using `ImageHandler`

```
>>> import numpy as np
>>> from pwem.emlib.image import ImageHandler # module to handle images
>>> fileName = 'three_by_three.mrc' # an small 3x3 image located
>>> ih = ImageHandler()
>>> image = ih.createImage() #  Uninitialized image of class xmipp.Image
>>> matrix = np.array([[1, 2, 3], [4, 5, 6], [7,8,9]], np.float32)
>>> image.setData(matrix)
>>> image.write("tmp.mrc")
>>> help(image) # list of image methods

# type in the terminal od -f tmp.mrc
# to see the content of file tmp.mrc
# you should see the header followed
# by the matrix
```

2.4 Create a protocol that compute the mean and variance of a set of images

Let us start with the real work. We are going to create a protocol that reads a set of particles (`proj.stk`, figure 2) and computes the average and standard deviation images. The output should look like figure 3a and figure 3b

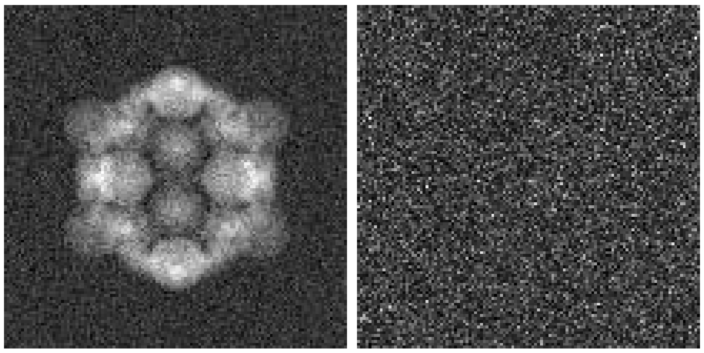

(a) average image (b) variance image

Figure 3: The two images created by the protocol `Statistics`

We suggest you to follow a Test-Driven Development (TDD) approach, which is an approximation to codding where you write a test before you write just enough production code to fulfill that test. Therefore, TDD is one way to think through your requirements or design before your write your functional code.

Very likely you have downloaded the `scipion-em-template` repository (otherwise do it using the command `git clone https://github.com/scipion-em/scipion-em-template.git; cd scipion-em-template`). Update it (`git pull`) and switch to the branch called `day3rmarabini` (`git checkout tags/3_1.statistics_protocol_template`, you may need to fetch it first with `git fetch --all --tags`). In this branch you will find a new protocol called `protocol_statistics` with the corresponding test. This protocol reads a `setOfParticles`, does nothing with them and creates as output two `Images`. These `Images` are empty in the sense that they do not point to any binary file. Execute the test `scipion3 tests myplugin.tests.test_protocol_statistics`, open *Scipion* (`scipion3 last`) and double check that a new project called `TestStatistics` has been created and that it contains two protocols called `import particles` and `compute statistics` respectively.

Now your work starts. You need to compute the average and variance image. Average images are created by adding together many pictures. If the images being averaged contain some commonalities, a pattern emerges revealing the regularities existing in the intensity patterns across all the images. On the other hand the variance images highlights those areas in which the differences are greater.

The steps you will need to fulfill are:

- execute the test, it will complain that the images `average.mrc` and `variance.mrc` do not exists
- modify the code as described in the following steps so that this two files are created
- get the `setOfImages` from the input parameters (self.inputParticles.get())
- get image dimensions and number of images (`self.inputParticles.get().getDim()` and `self.inputParticles.get().getSize()` are your friend here)
- create two numpy arrays for the average and variance images. (`numpy.zeros([xdim, ydim]), numpy.float32`)
- iterate through all particles (`for particle in self.inputParticles.get():`)
- using particle `ImageHandler().read(particle.getLocation())` read image and extract matrix (`getData`).
- the value of the average image at pixel (i, j) is $\mu(i, j) = \frac{1}{N}\sum_{n=1}^{N} x_n(i, j)$. Where N is the number of images and $x_n(i, j)$ the value of the pixel (i, j) for the image x_n.
- the value of the variance image at pixel (i, j) is $\sigma_n^2(i, j) = \frac{1}{N}\sum_{n=1}^{N} (x_n(i, j) - \mu(i, j))^2$. (see appendix A for an algorithm to compute the variance in a computationally efficient way).
- assign the np.arrays to two `xmipp.Image` (`setData`)
- and save them in the directory protocol extra directory (`self._getExtraPath(filename)`) with file names `average.mrc` and `variance.mrc` respectively.
- finally, assign the created filenames to the Two `Images` in the `createOutputStep` function

- Note that the default "Analyze" which shows all images produced by the protocols is just what we need in this case.

Remember: (1) you may execute the protocol at any time running the test and in case you introduce any print statement to debug your code the output will be redirected to the *Scipion* output log, (2) The solution to this exercise can be browsed in github.com project scipion-em:scipion-em-template selecting the tag `3_2.statistics_protocol_solution`,

2.5 Protocols that require image format conversion

Some image processing software can only read a limited set of file formats. Since we are using `ImageHandler` for reading, our protocol is quite tolerant from the image format point of view but this is not always true and, when using third party software, many times we need to convert the data before it can be used as input. Let's imagine that we want to create a thumbnail of our image.

A possible solution is to use the python PIL module:

Listing 3: The following script creates 64x64 thumbnails using PIL.

```
>>> from PIL import Image # this is PIL Image class
                          # not scipion Image class
>>> im = Image.open("tmp.jpg")
>>> size = 64, 64
>>> im.thumbnail(size)
>>> im.save("out.jpg")
```

but unfortunately `PIL` will not read electron microscopy images. Therefore, we need to convert the image before calling `PIL`.

Reusing the protocol we just created (`statistics`) modify it so a thumbnail is created for each input particle and saved in an output stack. I have created a test file for you. You may execute it by typing: `scipion3 tests myplugin.tests.test_protocol_thumbnail`. Since the protocol has not been implemented, the test will fail. In order to pass the test you will need to:

- make a copy of the file `protocol_statistics.py` and call it `protocol_thumbnail.py`
- change the `_label` from `Statistics` to `Thumbnail`
- change the name of the class from `ProtStatistics` to `ProtThumbnail`
- import the new class in the `__init__.py`
- add an extra parameter to the protocol: the size of the output thumbnail (parameter type = `IntParam`, parameter name = `size`)
- rename the function `computeStatistics` to `computeThumbnail`. Clean it, you only need to keep the lines

```
setOfParticles = self.inputParticles.get()
for particle in setOfParticles:
```

- in the loop though all particles convert the image from `Spider` to `jpg` with the statement:

```
imageScipion = ih.read(particle.getLocation());
imageScipion.write(outTmpFileName)}
```

- thumbnail each image using the above code (listing 3)
- Do not forget to assign to `size` the value supplied in the form.

```
size = self.size.get()
...
imPIL.thumbnail((size, size))
```

- save the results in a image stack. In order to create the stack use `ImageHandler` and provide as output names (1, out.stk), (2, out.stk), etc. The extension `stk` informs `ImageHandler` that a `Spider` stack file is needed and the numerals 1, 2, etc place the image as the first, second, etc image in the stack.

```
                    imageScipion = ih.read("path to PIL output")
                    imageScipion.write((index +1, fileName))
```

- Now we have the output binary file but we need to create the corresponding *Scipion* object in `createOutputStep`. Create an object of the class `setOfParticles`. Check listing 4 for an example.
- Again, the default viewer should just want we need.

Listing 4: "Creating an output set of particles"

```
partSet = self.protocol._createSetOfParticles()
outFileName = self._getExtraPath('out.stk')

for index, p1 in enumerate(self.inputParticles.get())
    # some code
    ...
    ...
    # create output setOfParticles
    p2 = Particle()
    p2.setLocation(index + 1, outFileName))
    partSet.append(particle)
```

You may have noticed that even if the thumbnails are being produced and you can visualize then the test fails. This is because it is testing if the attribute `setPart.thumbnail_size` exists. *Scipion* objects may incorporate any arbitrary attribute, as `size`, providing is an object of a class defined by *Scipion*. `size` type is `Float`, a class that can store a float number and knows how to persist it in *Scipion* database. Just add the line `setPartOut.thumbnail_size = self.size` in `createOutputStep` and the object `setPartOut` will be persisted with the attribute `thumbnail_size`.

That will be all for today ;-). Remember that the solution to this exercise can be browsed in github.com project scipion-em:scipion-em-template selecting the tag `3_3.All_solved`.

Appendices

A Computing Sample Mean and Variance in One Pass

An easy computation gives us the following identity that suggests a method for computing the variance in a single pass, by simply accumulating the sums of x_n and x_n^2:

$$\sigma^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n - \mu)^2 = \frac{\sum_{n=1}^{N}(x_n^2 - 2\mu x_n + \mu^2)}{N} = \frac{(\sum x_n^2) - 2N\mu^2 + N\mu^2}{N} = \frac{\sum x_n^2 - N\mu^2}{N}$$

Pseudocode for a one-pass variance computation could then look like:

```
variance(samples):
  sum := 0
  sumsq := 0
  for x in samples:
    sum := sum + x
    sumsq := sumsq + x**2
  mean := sum/N
  variance := sumsq - N*mean**2)/N
```

If you are thinking in using this implementation of variance in your software be careful since if the variance is small compared to the square of the mean, computing the difference may lead to catastrophic cancellation where significant leading digits are eliminated and the result has a large relative error. In fact, you may even compute a negative variance, which is mathematically impossible.