Modeling in digital rock analysis: an image segmentation tutorial

Felipe F. Alves[1], João Leal[1], Ricardo Leiderman[1], Andre Pereira[1]

[1]*Laboratory of Scientific Computing, Dept. of Computer Science, Universidade Federal Fluminense*
Av. Gal. Milton Tavares de Souza, s/n, 24210-310, RJ/Niteroi, Brazil
andre@ic.uff.br

Abstract. Image-based simulations are fundamental to assist in rock characterizations using a fully digital environment. The basic workflow in digital rock physics or petrophysics comprises three general phases: 1) 3-D scan: image acquisition and reconstruction, 2) 3-D modeling: image processing and segmentation, and 3) 3-D analysis: image simulation and characterization. In this work, we focus in the modeling phase, by presenting two educational approaches to perform image segmentation of rock samples. The first approach relies completely in the use of software with user interfaces, whereas the second one is a fully script-based approach relying in open source packages with Python APIs. Although we describe educational approaches, which adopt compact sets of dedicated strategies, they can also be combined to achieve the best performance and learning process. We assume here that the images were acquired by high-resolution X-ray micro-computed tomography (micro-CT), and that they will be employed to perform rock characterizations, such as determination of the effective absolute permeability of porous materials based on the concepts of numerical homogenization. Therefore, step-by-step procedures and implementations to achieve high quality segmented images are described.

Keywords: image segmentation, petrophysics, micro-CT, image-based analysis, modeling

1 Introduction

Computer simulations have been increasingly adopted as an attractive auxiliary tool for laboratory physical tests. However, realistic models are expected in order to obtain accurate responses. The challenge of creating realistic models to perform the simulations can be overcome by means of sophisticated imaging technologies such as high-resolution X-ray micro-computed tomography (micro-CT). This technology allows scanning the internal micro-structure of the material, transforming it into realistic digital models. In this sense, in the context of rock physics and petrophysics, image-based simulations are fundamental for the digital characterization of rocks.

In the basic digital rock characterization workflow, 3-D modeling is the phase that follows image acquisition and precedes image-based analysis, typically consisting of image processing and image segmentation tasks. There is a vast literature on image segmentation. It is possible to find works that present from simple to sophisticated segmentation methods. However, most works focus on describing new developments or applying state-of-the-art methods to solve challenging problems. Although these works have played a key role in advancing the field, we have identified that beginners in the field still have great difficulty in reproducing them or in acquiring the necessary skills to apply those sophisticated method to their problems of interest.

In this work, we present two educational approaches to perform rock image segmentation, following a tutorial-based strategy and describing methods to obtain high-quality segmented images. The procedures to perform image segmentation of porous materials have several peculiarities. In order to segment the rock images addressed in this paper, a sequence of steps are followed. They are meant to take an image from the micro-CT output to a labeled one. The concepts and objectives of each step and the instructions on how to apply those, being it by the use of in-house Python scripts or by use of ImageJ, a publicly available image processing software, will be explained in the next sections.

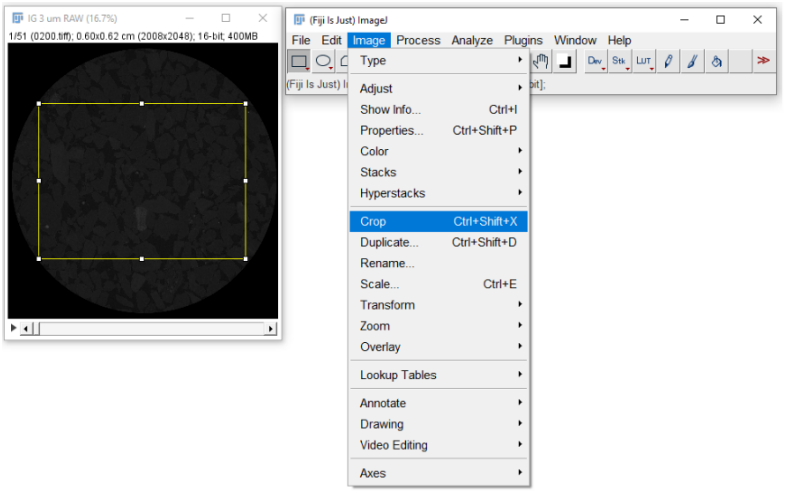
This paper is suggested as a use case in lectures and training for senior undergraduate and graduate students. Since it covers the application of image processing and segmentation in both theoretical and practical perspectives, it can be a powerful tool when teaching subjects such as material modeling, petrophysics, and other classes.

2 Methodology Using Software with Apps

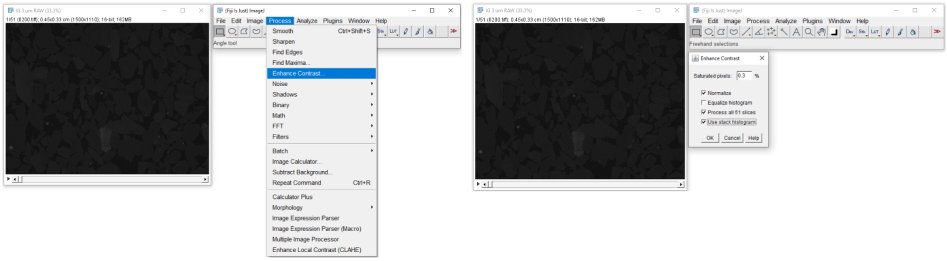
As a first approach, the entire workflow to segment the sample images used as the example in this paper will be completed using exclusively the publicly available application FIJI/ImageJ, which is a very user friendly tool that requires no coding knowledge at all. Notice that, in this application, it's possible to load a single image at a time, or to load an image stack, which is the ideal choice for segmenting several micro-CT slices at a time.

2.1 Image Pre-processing

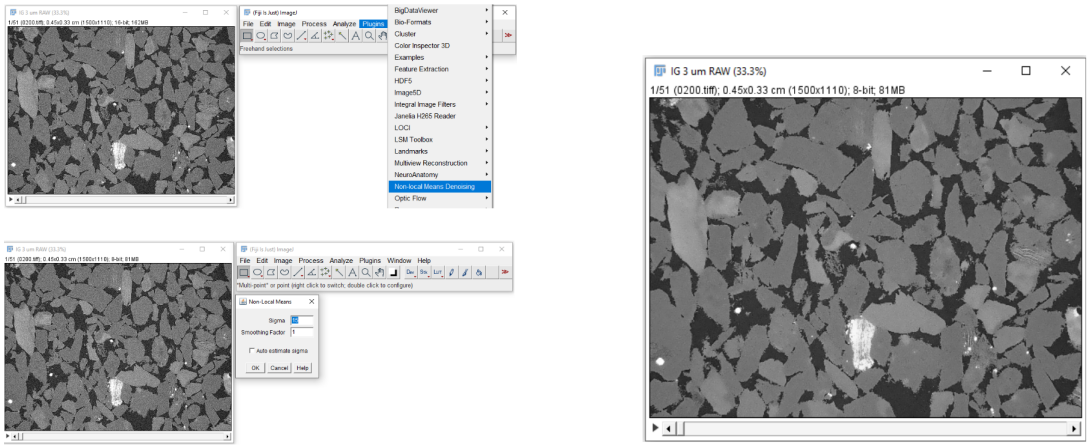
Regardless of the chosen approach to deal with the stack of images created by micro-CT scanning of the rock samples, many common steps must be taken. The first step to process the micro-CT images is to crop the region that is relevant to the analysis. This occurs because a large part of a micro-CT image can be the residual background of the scanning, as can be seen in Fig. 1a. Therefore, a cropping operation is required to remove this background.



(a) Cropping sequence



(b) Normalizing sequence



(c) Denoising sequence

(d) Fully pre-processed image

Figure 1. Pre-processing steps using the ImageJ

Using image-processing techniques, it is possible to recover valuable information from the images. After the cropping, the images must undergo another step called normalization. This operation maps the range of pixel values in the original image to a new range, usually, from 0 to 1. In the case of 16-bit images, the intensities range between 0 and 65535, which represent the minimum and the maximum value of this format. The main reason for performing a normalization operation is to achieve a more balanced distribution of pixel intensity values. This significantly enhances image contrast, which, in turn aids in the next steps.

It's important to notice that there are two possible ways to perform this operation: it could be done using the histogram of each stack slice, or by using the combined histogram of all stack slices. The second approach was chosen, as some images may lack high or low-intensity pixels, depending on the position of rock components. This can lead to relative differences in the histogram's essential features. Histograms, in general, are frequency distributions, and in the case of images they describe the frequency of the intensity values that occur in an image. As histograms depict image statistics in an easily interpreted visual format, they can be used to improve the visual appearance of the images [1].

The last pre-processing step consists in applying a denoising filter. For the images chosen for demonstration in this paper, as well as for a large number of micro-CT images, a non-local means denoising filter is a great option [1]. This filter's purpose is to reduce image noise, while preserving the textures and borders from the scanned materials. This way, the resulting image retains most of the important features of the original images.

After importing the image stack, the steps shown in Figs. 1a, 1b and 1c should be carried out to perform the cropping, normalization and denoising operations, respectively. Figure 1d shows the result of the entire workflow. Notice that, as mentioned on the normalization step, it's very important to check "Use stack histogram", as this improves the results during the segmentation process. All of the above operations were performed in the entire image sequence at once.

2.2 Image Segmentation

Once the images are pre-processed, it is possible to move to the segmentation process. Three different methods are introduced here: **image thresholding**, which was done by using ImageJ software and also by using the developed Python scripts; a **machine learning** algorithm, implemented in ImageJ's Weka Trainanable Segmentation plug-in; and a **deep learning** algorithm, implemented in Python scripts. These operations are described in the following subsections.

Thresholding. For the first segmentation method, image thresholding was used. This process consists in evaluating each pixel individually, and if its intensity is greater than a certain threshold value, it gets labeled a certain class. However, if the pixel intensity value is less than the threshold, it is assigned another class. In the end, all of the image's pixels are labeled with a number representing their respective class instead of their intensities. Figure 2d is an example of the segmented images obtained by the described method.

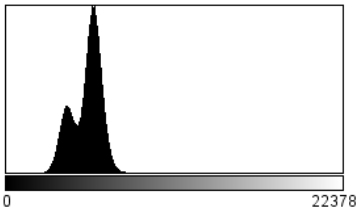
The pre-processing steps described above are specially important when segmenting by thresholding, since the higher contrast and the decreased noise increases the differences between material phases. This can be visually perceived, but in a more technical way, the use of image histograms is desired. In Figs. 2a and 2b, it can be noticed that after the pre-processing operations two peaks can be more clearly seen, with each one representing pixels in each of the material's phases (pore and solid phases).

Another relevant factor is the type of threshold used. In this case, the best results were obtained when Otsu threshold [2] was chosen. It can be explained by the fact that this thresholding technique minimizes the variance in each class, which is excellent for the type of histogram presented by those images. So, in order to perform thresholding segmentation using ImageJ software, the steps shown in Fig. 2c should be followed. In this specific example, the primary goal is to distinguish pores from all solid phases. Therefore, a single thresholding operation is sufficient. If it becomes necessary to identify all the existing phases, multiple thresholding operations will need to be performed.

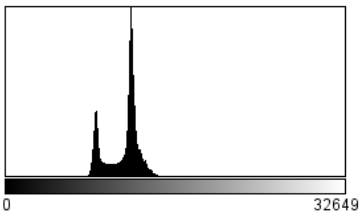
Trainable Weka Segmentation. Artificial Intelligence plays a crucial role in image segmentation, particularly in highly challenging scenarios where various textures render conventional techniques ineffective. However, utilizing AI poses specific challenges, the most prominent being the necessity to provide a machine learning model with training data. In this context, classifying pixels in a set of images requires some pixels from the entire dataset to be previously labeled and employed as training data.

Therefore, the second segmentation method to be described in this paper, is the use of a machine learning algorithm, implemented in the open-source software Fiji/ImageJ [3], in a plug-in called Trainable Weka Segmentation [4]. This plug-in allows the user to manually select pixels and assign them different classes. After this, a classifier model is created, trained, using user input data as training labels, and saved. It can later be loaded and

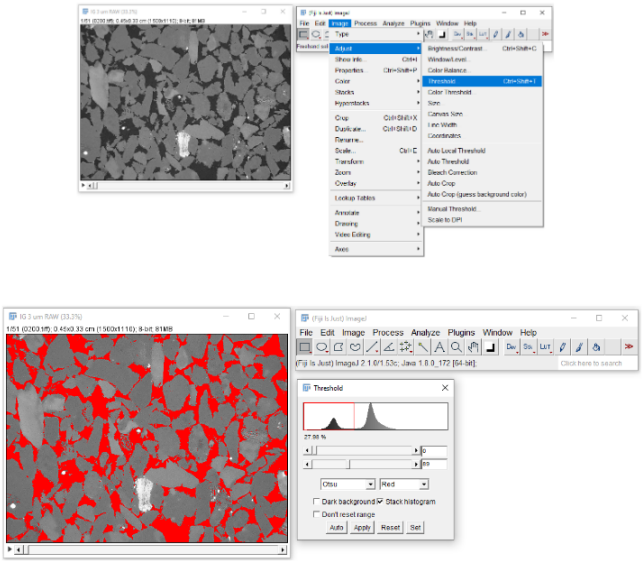
run to segment a complete image data set. In order to use trainable Weka segmentation, the steps shown in Fig. 3 should be followed. This plug-in allows user to create more than two classes, if recognition of multiple phases is desired. In the above example, pores were classified as class 2, while all other phases were classified as class 1. Usually, manually selecting around 10 regions for each class is enough to achieve very good segmentation results.



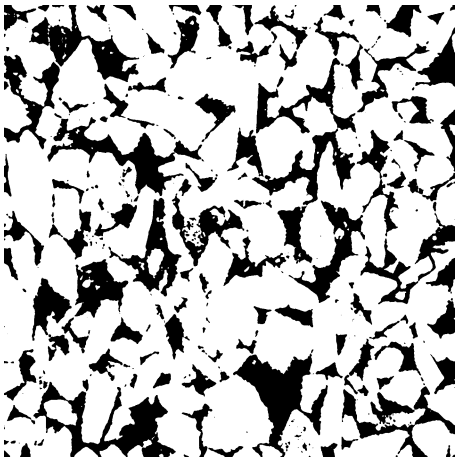
(a) Image histogram before pre-processing



(b) Image histogram after pre-processing



(c) Thresholding sequence



(d) Image segmented by thresholding

Figure 2. Thresholding steps using the ImageJ

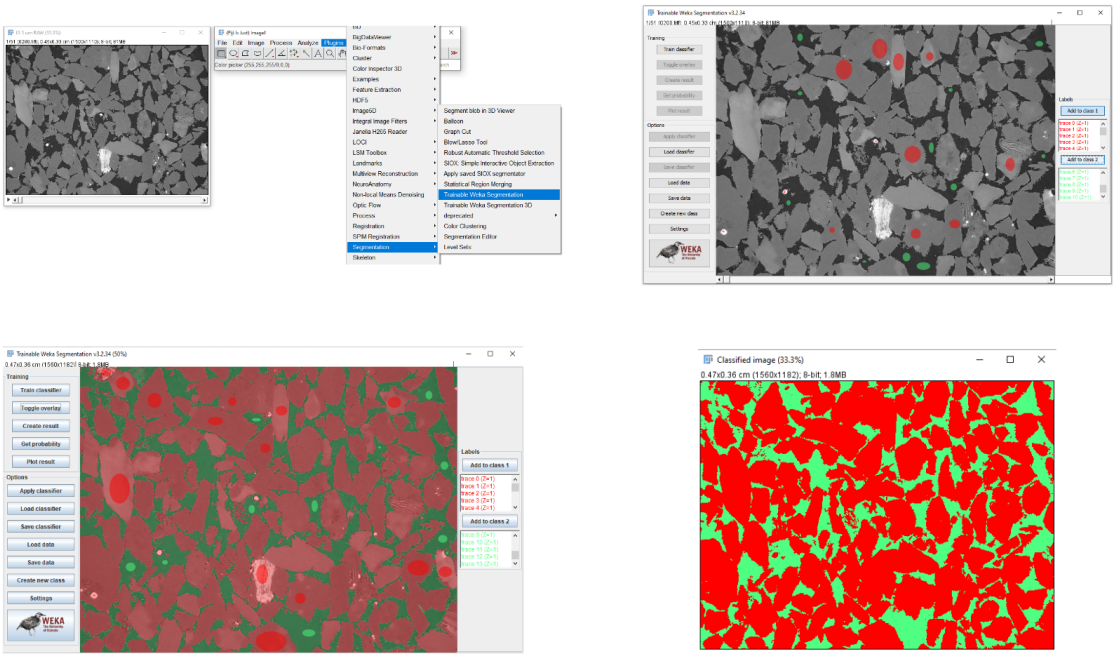


Figure 3. Trainable Weka Segmentation sequence

3 Methodology Using Packages with Scripts

Another alternative to segment images is to use scripts. In this study, Python was chosen in conjunction with some packages and libraries, such as openCV, numpy, scikit-learn, skimage, tensorflow, Keras, glob and os. All of them can be found online for free and are open-source. In the next subsections, the pre-processing and segmentation methodologies will be detailed, along with the scripts used in each step.

3.1 Image Pre-processing

As mentioned before, in the developed Python script, the pre-processing sequence consists of cropping the original micro-CT stack, normalizing the slices of the image stack and application of non-local means denoising filter. With the goal of making the code as educational as possible, each one of these steps are executed inside a single Python function, which will be presented and explained in the following sub-sections (Fig. 4).

Image Cropping. To remove the borders of the micro-CT images and reduce memory usage, a cropping algorithm was implemented. It can be seen in Listing 1, where a function is defined. It takes an image stack and the dimensions for the cropped image, and returns a stack already cropped. Also, the cropped image is taken from the original image's center.

Listing 1. Image Cropping

```
def crop_image(stack,new_dim):
   n_images, H, L = stack.shape
   return stack[:,(H-new_dim[0])//2:(H+new_dim[0])//2,(L-new_dim[1])//2:(L+new_dim[1])//2]
```

Image Normalization. As previously discussed, a normalization operation is required to increase image contrast, which will help on the following steps of the image segmentation. For this purpose, the function shown in Listing 2 was used. It receives an image stack and reads the minimum and maximum value of pixels' intensity. After this, it maps the pixel range of each image to a range from 0 to 1 and then back to the stack's original range.

Listing 2. Image Normalization

```
def normalize(stack):
   max = np.max(stack)
   min = np.min(stack)
   max_poss = np.iinfo(stack.dtype).max
   for i in range(stack.shape[0]):
      aux_img = (stack[i]-min)/(max-min) # between 0 and 1
      stack[i] = aux_img * max_poss # to complete uint16
   return stack
```

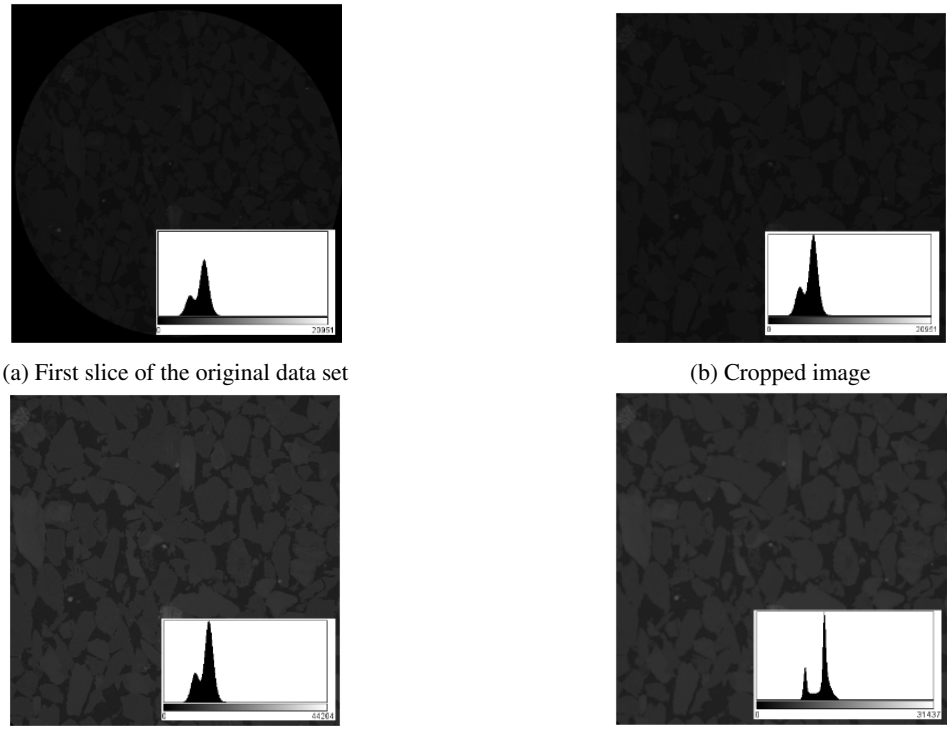
Image Denoising. The application of non-local means denoising filter was done by the use of two functions already implemented in the *Scikit Image* package, "estimate_sigma" and "denoise_nl_means". These functions allow the application of the filter to the entire image stack, instead of doing it slice by slice, as seen in Listing 3.

Listing 3. Image Denoised

```
def non_local_means(stack):
   max_poss = np.iinfo(stack.dtype).max
   stack_type = stack.dtype
   sigma_est = np.mean(estimate_sigma(stack))
   stack=denoise_nl_means(stack,h=1.15*sigma_est,fast_mode=True,patch_size=5,patch_distance=3)
   return (stack*max_poss).astype(stack_type)
```

3.2 Image Segmentation

As previously mentioned, after the pre-processing step, the images undergo segmentation to generate their labels. In Python, both the threshold and U-net methods were employed. The implementation of each method will be explained in the subsequent subsections.



(a) First slice of the original data set

(b) Cropped image

(c) Normalized image

(d) Filtered image

Figure 4. Pre-processing steps using Python scripts

Threshold. The function presented in Listing 4 is using the openCV packages to threshold each image using the Otsu method. It also return the result stack as a 8-bit image, since the pixels' values is now only its label.

Listing 4. Image Threshold

```
def threshold(stack):
   for i in range(stack.shape[0]):
      _,aux=cv2.threshold(stack[i],0,np.iinfo(stack.dtype).max,cv2.THRESH_BINARY+cv2.THRESH_OTSU)
      stack[i] = aux
   return stack.astype(np.uint8)
```

U-net. Even though conventional machine learning algorithms are very capable of achieving great results in image segmentation tasks, they can be out-performed by more modern Deep Learning Algorithms. In this context, there is a huge amount of possibilities when it comes to choosing the best network architecture, as different problems will require different approaches. For this paper, the U-net architecture will be used, mostly for its ability to segment the images with a fairly small training data set [5].

Since the U-net code is extensive, it will be made available in a git repository, while only its working process will be discussed here, with a few examples. Furthermore, as a neural network, U-net will need training data, which are images from our interest data set already segmented. This can be obtained by any other method explained before in this study. For this training, 50 images were used as training data set, from a total of 1401. After this data set has been generated, data augmentation can be done, so the U-net can have more inputs and develop a better segmentation capability. This was obtained by rotating the images and implemented in the provided code. It's shown in Listing 5. After this, each image can generate up to 7 new images. So, the training data set went from 50 to 400 images. Also in the data augmentation, the labels are encoded. Their values, that after threshold become discrete, are now mapped as integer from 0 to number of classes minus 1. This is shown in Listing 6. Further, they are changed into categorical, which is necessary to generate the U-net model. This code is shown in Listing 7.

Another important issue to discuss is the model's architecture. Different U-net architectures were tested and the one that most fit for both efficiency and results was also provided. With all this defined, the U-net model can now be trained. The training was done with the code available in the git repository. It took a total of 4 hours and resulted in a 96% of accuracy when compared to the threshold method. After this, the trained model can be used to segment all data set without having to normalize or denoise, reducing the time spent.

Listing 5. Data augmentation

```
def save_stack(stack,path,inicial_value=0):
   for i in range(stack.shape[0]):
      cv2.imwrite(path + "\%04i.tiff" %(i+inicial_value), stack[i])
   return
save_stack(stack,final_path,0)
save_stack(np.flip(stack,1),final_path,stack.shape[0]*1)
save_stack(np.flip(stack,2),final_path,stack.shape[0]*2)
save_stack(np.flip(np.flip(stack,1),2),final_path,stack.shape[0]*3)
stack = np.rot90(stack,1,(1,2))
save_stack(stack,final_path,stack.shape[0]*4)
save_stack(np.flip(stack,1),final_path,stack.shape[0]*5)
save_stack(np.flip(stack,2),final_path,stack.shape[0]*6)
save_stack(np.flip(np.flip(stack,1),2),final_path,stack.shape[0]*7)
```

Listing 6. Label encoder

```
def encode_masks(stack):
   from sklearn.preprocessing import LabelEncoder
   labelencoder = LabelEncoder()
   n_images, height, width = stack.shape
   stack_reshaped = stack.reshape(-1,1)
   stack_reshaped_encoded = labelencoder.fit_transform(np.ravel(stack_reshaped))
   stack = stack_reshaped_encoded.reshape(n_images, height, width).astype(np.uint8)
   return stack
```

Listing 7. Image to categorical

```
imasks_cat=to_categorical(y_train, num_classes=n_classes)
y_train_cat=imasks_cat.reshape((y_train.shape[0],y_train.shape[1],y_train.shape[2],n_classes))
```

4 Conclusions

In this paper, we described image processing and segmentation as modeling phase in digital rock characterizations, by presenting two educational approaches. The first approach relied completely in the use of software with user interfaces, whereas the second one was a fully script-based approach relying in open source packages with Python APIs. These approaches can be combined to achieve the best performance and learning process. Papers on image segmentation often describe new developments or applications of state-of-the-art methods to solve challenging segmentation problems. This leads to the a lack of important steps for the understanding of the reader, who possibly does not have extensive experience with all the subjects involved in the process of image segmentation.

Acknowledgements. This research was carried out in association with the ongoing R&D project registered as ANP n o 21289–4, “Desenvolvimento de modelos matemáticos, estatísticos e computacionais para o aperfeiçoamento da caracterização petrofísica de reservatórios por Ressonância Magnética Nuclear (RMN)” (UFF/Shell Brasil/ANP), sponsored by Shell Brasil under the ANP R&D levy as “Compromisso de Investimentos com Pesquisa e Desenvolvimento”. The authors also recognize the support from CAPES, CNPq and FAPERJ.

Authorship statement. The authors hereby confirm that they are the sole liable persons responsible for the authorship of this work, and that all material that has been herein included as part of the present paper is either the property (and authorship) of the authors, or has the permission of the owners to be included here.

References

[1] W. Burger and M. Burge. *Principles of digital image processing: fundamental techniques*. Springer, 2009.
[2] M. Sezgin and B. l. Sankur. Survey over image thresholding techniques and quantitative performance evaluation. *Journal of Electronic imaging*, vol. 13, n. 1, pp. 146–168, 2004.
[3] J. Schindelin, I. Arganda-Carreras, E. Frise, V. Kaynig, M. Longair, T. Pietzsch, S. Preibisch, and others. Fiji: an open-source platform for biological-image analysis. *Nature methods*, vol. 9, n. 7, pp. 676–682, 2012.
[4] D. Legland, I. Arganda-Carreras, and P. Andrey. Morpholibj: integrated library and plugins for mathematical morphology with imagej. *Bioinformatics*, vol. 32, n. 22, pp. 3532–3534, 2016.
[5] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. *Medical Image Computing and Computer-Assisted Intervention. MICCAI 2015*, vol. 1, n. 1, pp. 1–8, 2015.