

ImagesToLARModel, a tool for creation of three-dimensional models from a stack of images

Danilo Salvati

December 14, 2015

Abstract

Here we will present a software for creating a three-dimensional model from a stack of images. This can be useful because of the simplicity of these type of representations. In particular a scope of use can be offered by medicine, where there is an enormous number of images but with very complex two-dimensional representations.

This work will use the LAR representation ([[CL13](#)]) with the Julia language, because of its simplicity, showing how it can be used for quickly process image data.

Contents

1	Introduction	4
1.1	Why Julia	5
2	Software structure	5
2.1	Julia packages	5
2.2	Architecture of ImagesToLARModel	6
3	ImagesToLARModel	7
3.1	Calling modules	7
3.2	Input loading for data preparation	8
3.3	Input loading for images conversion	10
3.4	Data preparation	12
3.5	Starting conversion	13
4	PngStack2Array3dJulia	14
4.1	Module imports	14
4.2	Convert input to png	15
4.2.1	Image resizing	17
4.3	Getting data from a png	20
4.4	Centroids computation	21
4.5	Transform pixels into three-dimensional array	22
5	ImagesConversion	26
5.1	General algorithm	26
5.2	Module imports	28
5.3	Data preparation	28
5.4	Conversion pipeline	29
5.4.1	Images conversion step	33
5.4.2	Boundaries merge step	38
5.4.3	Block merge step	40
5.4.4	Smoothing step	42
5.4.5	Model creation step	46
6	GenerateBorderMatrix	47
6.1	Module imports	47
6.2	Get border matrix from file	47
6.3	Write border matrix on file	48
6.4	Compute border matrix	49
6.5	Transform border matrix	50

7	Lar2Julia	51
7.1	Module imports	51
7.2	Get boundary chain from a model	51
7.3	Get oriented cells from a chain	53
7.4	Transform relationships from arrays of arrays to a sparse matrix	53
8	LARUtils	55
8.1	Module imports	55
8.2	Transformation from matrix to array	55
8.3	Get bases of a LAR model	56
8.4	Double vertices and faces removal	60
8.5	Creation of a LAR model	63
8.6	Removing double faces and vertices from boundaries	70
9	Smoother	72
9.1	Get adjacent vertices	73
9.2	Laplacian smoothing	74
10	Model2Obj	76
10.1	Writing models to file	76
10.2	Merging block models	78
10.2.1	Parallel merging of obj files	80
10.3	Load models from files	86
11	Exporting the library	87
11.1	Installing the library	91
12	Conclusions	91
12.1	Results	91
12.2	Further improvements	91
A	Utility functions	91
B	Tests	91

1 Introduction

This work has the aim to transform a two-dimensional representation of a model (based on a stack of images) into a three-dimensional representation based on the LAR schema. In particular, it will produce a single obj model which can be viewed with standard graphics softwares.

In the past were developed other softwares using same principles (see [PDFJ15]). However, they were optimized for speed and cannot be able to accept huge amounts of data. With the rise of the big data era, we now have more and more data available for research purposes, so softwares must be able to deal with them. A typical hardware environment is based on a cluster of computers where computation can be distributed among a lot of different processes. However, as stated by *Amdahl's law*, the speedup of a program using multiple processors is limited by the time needed for the sequential fraction of the program. So use of parallel techniques for dealing with big data is not important for time performance gain but for memory space gain. In fact, our biggest problem is lack of memory, due to model sizes. As a consequence, every parts of this software is written with the clear objective of minimizing memory usage at the cost of losing something in terms of time performance. So, for example, images will be converted in blocks determined by a grid size (see section 5) among different processes and different machines of the cluster



Figure 1: Amdahl's law

1.1 Why Julia

Ricordare che precedenti versioni erano in python

Semplicità

Efficienza

Capacità di realizzare programmi paralleli con poco sforzo

2 Software structure

2.1 Julia packages

This software will be distributed as a Julia Package. For the actual release (Julia 0.4) a package is a simple git project with the structure showed in figure 2



Figure 2: Julia module structure

Source code must be in folder `src`, while in `test` folder there are module tests with a `runtests.jl` for executing them and with a `REQUIRE` file for specifying tests dependencies. For listing dependencies for the entire project, there is another `REQUIRE` file in main folder. As an example in figure 3 there is the `REQUIRE` file for `ImagesToLARMModel.jl`.

After creating this structure for a project it can be pushed on a git repository and installed on Julia systems. The usual installation procedure use this syntax:

```
Pkg.add("Package-name")
```

This will check for that package in METADATA.jl repository on github where there are all official Julia package. However it is also possible to install an unofficial package (on a public git repository) using this syntax:

```
julia 0.3
JSON
Logging
PyCall
Images
Colors
Clustering
```

Figure 3: REQUIRE contents for `ImagesToLARModel.jl`

```
Pkg.clone("git://repository-address.git")
```

This will install the package on your system with all the dependencies listed in REQUIRE file.

2.2 Architecture of ImagesToLARModel

In previous section we have seen how to create a Julia package for distribute our application. Now we focus on the structure of our application. In `src` folder we can find the following modules:

ImagesToLARModel.jl: main module for the software, it takes input parameters and start images conversion

ImagesConversion.jl: it is called by `ImagesToLARModel.jl` module and controls the entire conversion process calling all other modules

GenerateBorderMatrix.jl: it generates the boundary operator for grid specified in input, saving it in a JSON file

PngStack2Array3dJulia.jl: it is responsible of images loading and conversion into computable data

Lar2Julia.jl: it contains a small subset of LAR functions written in Julia language

LARUtils.jl: it contains utility functions for manipulation of LAR models

Smoother.jl: it contains function for smoothing of LAR models

Model2Obj.jl: it contains function that manipulates obj files

larcc.py: python larcc module for boundary computation. In next releases of the software it will be rewritten in Julia language

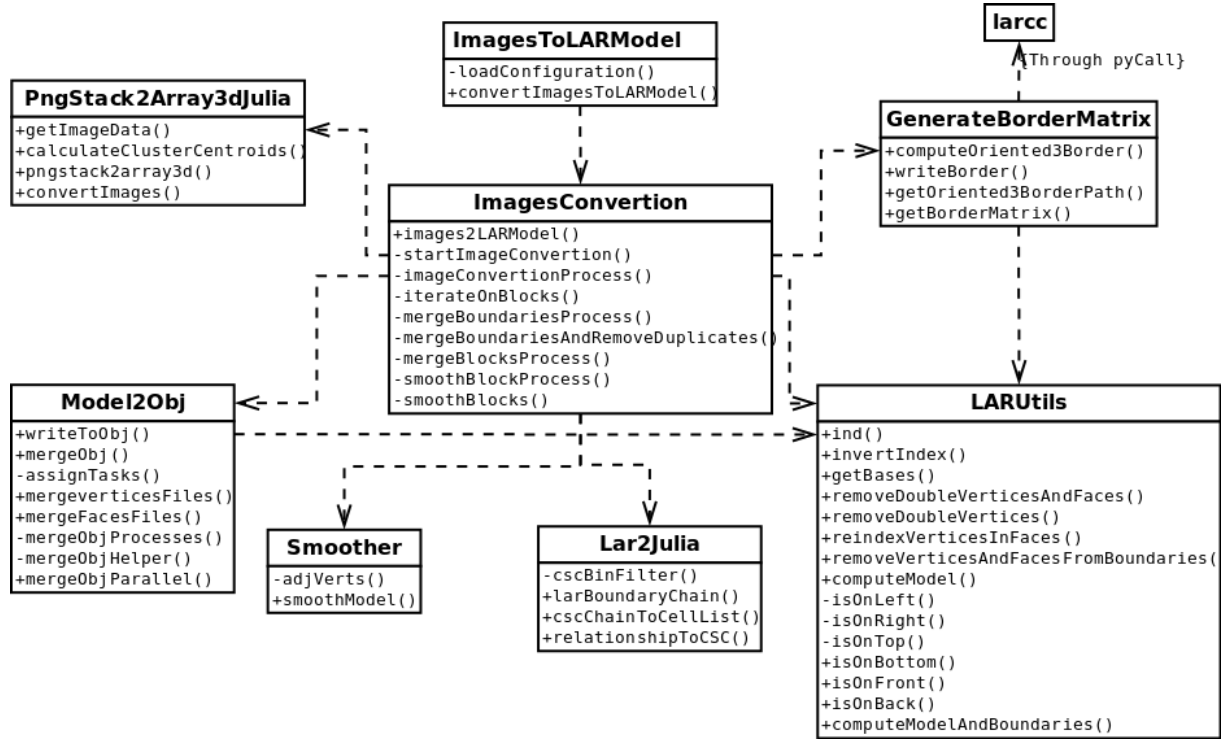


Figure 4: Schema of module dependencies of ImagesToLARModel

In figure 4 there is a simple schema of dependencies between modules.

Next sections of this document will explain in details all these modules showing also the code involved in conversion

3 ImagesToLARModel

This is the main module for the application; it takes the input data and start conversion calling `ImagesConversion.jl`.

3.1 Calling modules

As we have already said, this first module has the responsibility of starting the conversion calling all other modules in the package. In Julia calling modules requires that they are in a path specified by `LOAD_PATH` array. So at the beginning of this module we need to add this line:

`< update load path 6 > ≡`

```
push!(LOAD_PATH, Pkg.dir("ImagesToLARModel/src"))
◇
```

Fragment referenced in 86.

`Pkg.dir()` function gives us the path of the Julia installation, so `Pkg.dir("ImagesToLARModel/src")` returns “*⟨Julia – path⟩/ImagesToLARModel/src*”

After this line we can now import all modules defined here and export public functions:

```
⟨modules import ImagesToLARModel 7a⟩ ≡
    import JSON
    import ImagesConversion
    import PngStack2Array3dJulia

    using Logging

    export convertImagesToLARModel, prepareData

◇
```

Fragment referenced in 86.

3.2 Input loading for data preparation

Data preparation (see Section 3.4 takes several parameters:

- `inputDirectory`: The path of the directory containing the stack of images
- `outputDirectory`: The path of the directory with the output images
- `crop`: Parameter for images resizing (they can be extended or cropped)
- `noise_shape`: Intensity of the denoising filter for images (0 if you want to disable it)

Because of their number it has been realized a function for simply loading them from a JSON configuration file; this is the code:

```
⟨load JSON configuration for data preparation 7b⟩ ≡
```



```

function loadConfigurationPrepareData(configurationFile)
    """
    load parameters from JSON file for data preparation

    configurationFile: Path of the configuration file
    """

    configuration = JSON.parse(configurationFile)

    crop = Void
    try
        crop = configuration["crop"]
    catch
    end

    noise_shape = 0
    try
        noise_shape = configuration["noise_shape"]
    catch
    end

    return configuration["inputDirectory"], configuration["outputDirectory"],
        crop, noise_shape

end
◇

```

Fragment referenced in 86.

A valid JSON file has the following structure:

```

{
    "inputDirectory": "Path of the input directory",
    "outputDirectory": "Path of the output directory",
    "crop": Parameter for images resizing (they can be extended or cropped)
    "noise_shape": A number which indicates the intensity of the denoising
filter (0 if you want to disable denoising)
}

```

For example, we can write:

```
{
  "inputDirectory": "/home/juser/IMAGES/",
  "outputDirectory": "/home/juser/OUTPUT/",
  "crop": [[1,800],[1,600],[1,50]],
  "noise_shape": 0
}
```

crop and *noise_shape* are optional parameters

3.3 Input loading for images conversion

Images conversion takes several parameters:

- `inputDirectory`: The path of the directory containing the stack of images
- `outputDirectory`: The path of the directory containing the output
- `bestImage`: Image chosen for centroid computation (see section 4)
- `nx, ny, nz`: Sizes of the grid chosen for image segmentation (see section 4)
- `DEBUG_LEVEL`: Debug level for Julia logger
- `parallelMerge` (experimental): Choose between sequential or parallel merge of files (see section 10)

Because of their number it has been realized a function for simply loading them from a JSON configuration file; this is the code:

```
<load JSON configuration 9> ≡
function loadConfiguration(configurationFile)
  """
  load parameters from JSON file

  configurationFile: Path of the configuration file
  """

  configuration = JSON.parse(configurationFile)

  DEBUG_LEVELS = [DEBUG, INFO, WARNING, ERROR, CRITICAL]

  parallelMerge = false
  try
    if configuration["parallelMerge"] == "true"
```

```

        parallelMerge = true
    else
        parallelMerge = false
    end
catch
end

return configuration["inputDirectory"], configuration["outputDirectory"],
        configuration["bestImage"],
        configuration["nx"], configuration["ny"], configuration["nz"],
        DEBUG_LEVELS[configuration["DEBUG_LEVEL"]],
        parallelMerge

end
◇

```

Fragment referenced in 86.

A valid JSON file has the following structure:

```

{
  "inputDirectory": "Path of the input directory",
  "outputDirectory": "Path of the output directory",
  "bestImage": "Name of the best image (with extension) ",
  "nx": x grid size,
  "ny": y grid size,
  "nz": border z,
  "DEBUG_LEVEL": julia Logging level (can be a number from 1 to 5)
  "parallelMerge": "true" or "false"
}

```

For example, we can write:

```

{
  "inputDirectory": "/home/juser/IMAGES/",
  "outputDirectory": "/home/juser/OUTPUT/",
  "bestImage": "0009.tiff",
  "nx": 2,
  "ny": 2,
  "nz": 2,
  "DEBUG_LEVEL": 2
}

```

```
}
```

As we can see, in a valid JSON configuration file `DEBUG_LEVEL` can be a number from 1 to 5. Instead, when we explicitly define parameters, `DEBUG_LEVEL` can only be one of the following Julia constants:

- `DEBUG`
- `INFO`
- `WARNING`
- `ERROR`
- `CRITICAL`

parallelMerge is an optional parameter

3.4 Data preparation

⟨ data preparation from JSON file 11a ⟩ \equiv

```
function prepareData(configurationFile)
    """
    Prepare the input data converting all files into png
    format with the desired resizing and denoising

    configurationFile: Path of the configuration file
    """
    inputPath, outputPath, crop,
        noise_shape = loadConfigurationPrepareData(open(configurationFile))

    prepareData(inputPath, outputPath, crop, noise_shape)

end
◇
```

Fragment referenced in 86.

⟨ manual data preparation 11b ⟩ \equiv

```
function prepareData(inputPath, outputPath,
    crop = Void, noise_shape = 0)
    """
    Prepare the input data converting all files into png
```

```

format with the desired resizing and denoising

inputPath: Directory containing the stack of images
outputPath: Directory which will contain the output
crop: Parameter for images resizing (they can be
      extended or cropped)
noise_shape: The shape for image denoising
"""
# Create output directory
try
    mkpath(outputPath)
catch
end

PngStack2Array3dJulia.convertImages(inputPath, outputPath, crop, noise_shape)
end
◇

```

Fragment referenced in 86.

3.5 Starting conversion

As we have already said, this module has the only responsibility to collect data input and starts other modules. These are the functions that start the process and the only exposed to the application users:

```

⟨Start conversion from JSON file 12⟩ ≡
function convertImagesToLARModel(configurationFile)
    """
        Start conversion of a stack of images into a 3D model
        loading parameters from a JSON configuration file

        configurationFile: Path of the configuration file
    """
    inputDirectory, outputDirectory, bestImage, nx, ny, nz,
        DEBUG_LEVEL, parallelMerge = loadConfiguration(open(configurationFile))
    convertImagesToLARModel(inputDirectory, outputDirectory, bestImage,
        nx, ny, nz, DEBUG_LEVEL, parallelMerge)
end
◇

```

Fragment referenced in 86.

\langle *Start manual conversion 13* $\rangle \equiv$

```
function convertImagesToLARModel(inputDirectory, outputDirectory, bestImage,
                                nx, ny, nz, DEBUG_LEVEL = INFO,
                                parallelMerge = false)

    """
    Start conversion of a stack of images into a 3D model

    inputDirectory: Directory containing the stack of images
    outputDirectory: Directory containing the output
    bestImage: Image chosen for centroids computation
    nx, ny, nz: Border dimensions (Possibly the biggest power of two of images dimensions)
    DEBUG_LEVEL: Debug level for Julia logger. It can be one of the following:
        - DEBUG
        - INFO
        - WARNING
        - ERROR
        - CRITICAL
    parallelMerge: Choose if you want to use the algorithm
    for parallel merging (experimental)
    """
    # Create output directory
    try
        mkpath(outputDirectory)
    catch
    end

    Logging.configure(level=DEBUG_LEVEL)
    ImagesConversion.images2LARModel(nx, ny, nz, bestImage,
                                     inputDirectory, outputDirectory, parallelMerge)
end
◇
```

Fragment referenced in 86.

4 PngStack2Array3dJulia

This module has the responsibility of convert a png image into an array of values that will be passed to other modules

4.1 Module imports

These are modules needed for this part of the package and the public functions exported

```

⟨ modules import PngStack2Array3dJulia 14a ⟩ ≡
    using Images # For loading png images
    using Colors # For grayscale images
    using PyCall
    using Clustering
    using Logging
    @pyimport scipy.ndimage as ndimage

    export calculateClusterCentroids, pngstack2array3d, getImageData, convertImages
    ◇

```

Fragment referenced in 89b.

We need `Images`, `Clustering` and `Colors` packages for manipulating png images and `PyCall` for using Python functions for noise removal from images. As a consequence, we need a python environment with `scipy` to be able to run the package

4.2 Convert input to png

First thing to do in our program is getting our input folder and convert the stack of images into png format. This process lets us to avoid managing an enormous variety of formats during computation, simplifying code used for transformation.

Conversion needs the following parameters:

- `inputPath`: path of the folder containing the original images
- `outputPath`: path where we will save png images
- `crop`: parameters for images resizing (they can be extended or cropped)

Now we can examine single parts of conversion process. First of all we need to open the single image doing the following operations:

1. Open images using `Images` library (which relies on `ImageMagick`) and save them in greyscale png format
2. resize the images according to the `crop` parameter

```

⟨ Greyscale conversion 14b ⟩ ≡
    rgb_img = convert(Image{ColorTypes.RGB}, img)
    gray_img = convert(Image{ColorTypes.Gray}, rgb_img) ◇

```

Fragment referenced in 15c.

As we can see, we first need to convert image to RGB and then reconverting to greyscale. Without the RGB conversion these rows will return a stackoverflow error due to the presence of alpha channel

```
⟨ Image resizing 15a ⟩ ≡
    if(crop!= Void)
        # Resize images on x-axis and y-axis
        gray_img = resizeImage(gray_img, crop)
    end ◇
```

Fragment referenced in 15c.

The code for image resizing will be better explained in section [4.2.1](#).

Finally we have to reduce noise on the image. The best choice is using a *median filter* from package `scipy.ndimage` because it preserves better the edges of the image:

```
⟨ Reduce noise 15b ⟩ ≡
    # Denoising
    if noise_shape_detect != 0
        imArray = raw(gray_img)
        imArray = ndimage.median_filter(imArray, noise_shape_detect)
        gray_img = grayim(imArray)
    end ◇
```

Fragment referenced in 15c.

Where `imArray` is an array containing all raw data from images

Finally this is the code for the entire function:

```
⟨ Convert to png 15c ⟩ ≡
    function convertImages(inputPath, outputPath,
                           crop = Void, noise_shape_detect = 0)
        """
        Get all images contained in inputPath directory
        saving them in outputPath directory in png format.
        Images will be resized according with the crop parameter
        and if folder contains an odd number of images another one will be
```



```

added

inputPath: Directory containing input images
outputPath: Temporary directory containing png images
crop: Parameter for images resizing (they can be
      extended or cropped)
noise_shape_detect: Shape for the denoising filter
"""

imageFiles = readdir(inputPath)

⟨ Resize images on z-axis 18 ⟩

for imageFile in imageFiles
    img = imread(string(inputPath, imageFile))
    ⟨ Greyscale conversion 14b ⟩
    ⟨ Image resizing 15a ⟩

    ⟨ Reduce noise 15b ⟩

    outputFilename = string(outputPath, imageFile[1:rsearch(imageFile, ".")[1]], "png")
    imwrite(gray_img, outputFilename)

end
end
◇

```

Fragment referenced in 89b.

4.2.1 Image resizing

Now we will see in detail how to resize images on x and y axis. The most important parameter is *crop*, which is a list of numbers that contains the desired dimensions for the stack of images.

Given the list $[[xcropStart, xcropEnd], [ycropStart, ycropEnd], [zcropStart, zcropEnd]]$ we can have different cases based on the list values, as we can see in Figure 5.

So with the same parameter we can both resize or extend images. In particular, when we have to extend them, we have to get the raw data and concatenate a zeroed array to it. On the other hand, for image cropping we can use the `subim` function from the `Images` package.

⟨ *image resizing* 16 ⟩ \equiv

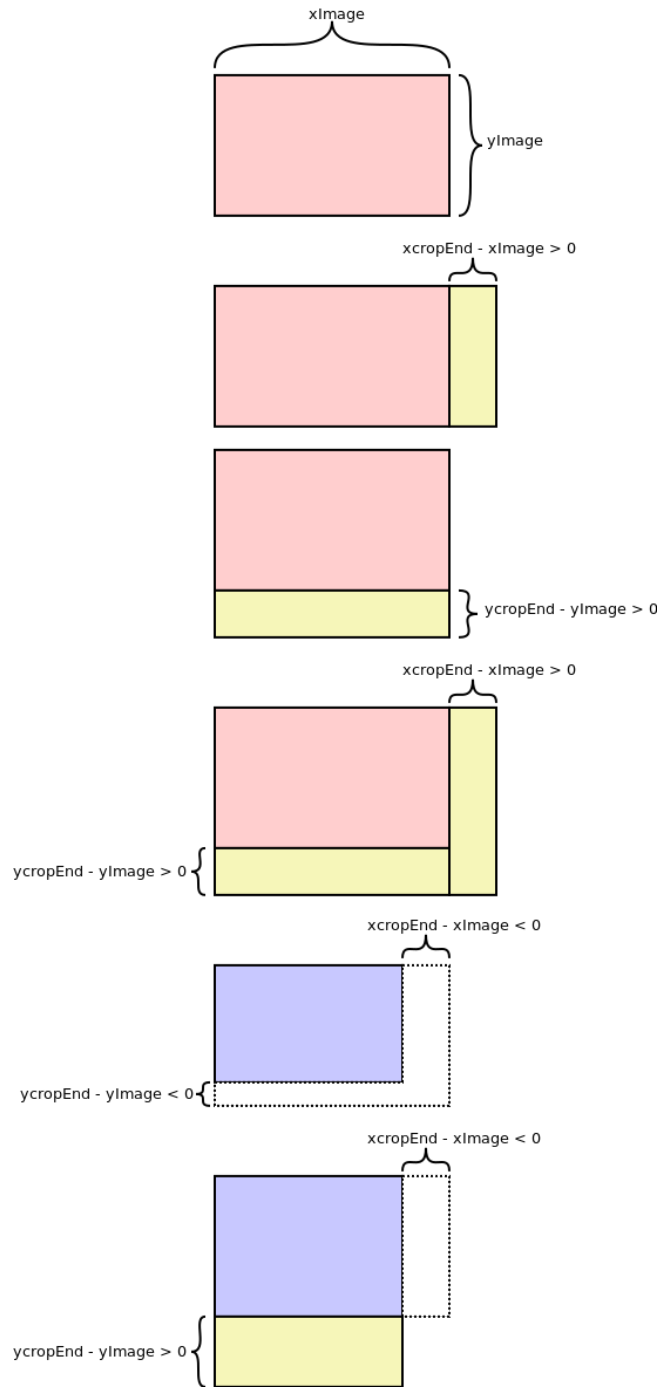


Figure 5: Some interesting resize cases. (a) The original image (b) Extension on the x dimension (c) Extension on the y dimension (d) Extension on both dimensions (e) Crop of both x and y (f) Crop of x and extension of y

```

function resizeImage(image, crop)
    """
    Utility function for images resize

    image: the input image
    crop: a list containing the crop parameters
          for the three dimensions

    returns the resized image
    """
    dim = size(image)
    if(crop[1][2] > dim[1])
        # Extending the images on the x axis
        imArray = raw(image)
        zeroArray = zeros(UInt8, dim[2])
        for i in (1 : (crop[1][2] - dim[1]))
            imArray = vcat(imArray, transpose(zeroArray))
        end
        image = grayim(imArray)
    end

    if(crop[2][2] > dim[2])
        # Extending the images on the y axis
        imArray = raw(image)
        zeroArray = zeros(UInt8, size(image)[1])
        for i in (1: (crop[2][2] - dim[2]))
            imArray = hcat(imArray, zeroArray)
        end
        image = grayim(imArray)
    end
    return subim(image, crop[1][1]:crop[1][2], crop[2][1]:crop[2][2])
end ◊

```

Fragment referenced in 89b.

However our stack of images have three dimensions, where the z-axis is represented by the number of images. So the third list for the crop parameters resize the stack removing or adding other images.

```

⟨ Resize images on z-axis 18 ⟩ ≡
    #Resizing on the z axis
    if(crop!= Void)
        numberOfImages = length(imageFiles)
        if(crop[3][2] > numberOfImages)

```

```

        imageWidth = crop[1][2] - crop[1][1] + 1
        imageHeight = crop[2][2] - crop[2][1] + 1
        for i in 1 : crop[3][2] - numberOfImages
            imArray = zeros(UInt8, imageWidth, imageHeight)
            img = grayim(imArray)
            outputFilename = string(outputPath, "/", imageFiles[end][1:rsearch(imageFiles[end], ".")
                                   "-added-", i, ".png")
            imwrite(img, outputFilename)
        end
    end
    imageFiles = imageFiles[crop[3][1]:min(numberOfImages, crop[3][2])]
end ◊

```

Fragment referenced in 15c.

4.3 Getting data from a png

Now we need to load information data from png images. In particular we are interested in getting width and height of an image. As stated in [W3C] document, a standard PNG file contains a *signature* followed by a sequence of *chunks* (each one with a specific type).

The signature always contain the following values:

137 80 78 71 13 10 26 10

This signature indicates that the remainder of the datastream contains a single PNG image, consisting of a series of chunks beginning with an IHDR chunk and ending with an IEND chunk. Every chunk is preceded by four bytes indicating its length.

As we are interested in width and height we need to parse the IHDR chunk. It is the first chunk in PNG datastream and its type field contains the decimal values:

73 72 68 82

The header also contains:

Width	4 bytes
Height	4 bytes
Bit depth	1 bytes
Color type	1 byte
Compression method	1 byte
Filter method	1 byte
Interlace method	1 byte

So for reading width and height we need first 24 bytes; the first eight contain the signature, then we have four bytes for length, four bytes for the type field and eight bytes for information we are interested in. This is the code:

```

⟨ Get image data 20 ⟩ ≡
function getImageData(imageFile)
    """
    Get width and height from a png image
    """

    input = open(imageFile, "r")
    data = readbytes(input, 24)

    if (convert(Array{Int},data[1:8]) != reshape([137 80 78 71 13 10 26 10],8))
        error("This is not a valid png image")
    end

    w = data[17:20]
    h = data[21:24]

    width = reinterpret{Int32, reverse(w)}[1]
    height = reinterpret{Int32, reverse(h)}[1]

    close(input)

    return width, height
end
◇

```

Fragment referenced in 89b.

4.4 Centroids computation

As we have seen above, this package uses greyscale images for conversion into three-dimensional models and for next steps we need binary images so we can distinguish between the background and the model we want to represent. We can use clustering techniques for obtaining this result. First step is centroids calculation from a chosen image (this choice must be made from the user, because we cannot knowing in advance what is the best image for finding clusters). Moreover we compute these centroids only for an image and then reuse them when we want to cluster all other images, saving processing time. Actually we need only two centroids, because next steps should only recognize between background and foreground pixels. This is the code used for centroid computation:

\langle Centroid computation 21 $\rangle \equiv$

```
function calculateClusterCentroids(path, image, numberOfClusters = 2)
    """
    Loads an image and calculate cluster centroids for segmentation

    path: Path of the image folder
    image: name of the image
    numberOfClusters: number of desired clusters
    """
    imageFilename = string(path, image)

    img = imread(imageFilename) # Open png image with Julia Package

    imArray = raw(img)

    imageWidth = size(imArray)[1]
    imageHeight = size(imArray)[2]

    # Getting pixel values and saving them with another shape
    image3d = Array{Array{UInt8,2}, 0}

    # Inserting page on another list and reshaping
    push!(image3d, imArray)
    pixel = reshape(image3d[1], (imageWidth * imageHeight), 1)

    centroids = kmeans(convert{Array{Float64}}(transpose(pixel)), 2).centers

    return convert{Array{UInt8}}(trunc(centroids))

end
◇
```

Fragment referenced in 89b.

4.5 Transform pixels into three-dimensional array

Now we can study the most important part of this module, where images are converted into data usable by other modules for the creation of the three-dimensional model. The basic concept consists in transforming every single pixel in an integer value representing color, and then clustering them all using centroids computed earlier. So, we can obtain a matrix containing only two values (the two centroids) representing background and foreground of the image.

Now we will follow the code. This function uses four parameters

- path: Path of the images directory
- minSlice: First image to read
- maxSlice: Last image to read
- centroids: Array containing centroids for clustering

For every image we want to transform in the interval $[\text{minSlice}, \text{maxSlice})$ we have to read it from disk and save pixel informations into a multidimensional Array:

```
< Read raw data 22a > ≡
    img = imread(imageFilename) # Open png image with Julia Package
    imArray = raw(img) # Putting pixel values into RAW 3d array ◇
```

Fragment referenced in 24.

The `Images.jl` `raw` function, get all pixel values saving them in an Array. In Figure 6 we can see how the array will be like for a sample greyscale image.

Finally we have to compute clusters obtaining images with only two values:

```
< Clustering images 22b > ≡
    # Image Quantization
    debug("page = ", page)
    debug("image3d[page] dimensions: ", size(image3d[page])[1], "\t", size(image3d[page])[2])
    pixel = reshape(image3d[page], size(image3d[page])[1] * size(image3d[page])[2] , 1)
    kmeansResults = kmeans!(convert(Array{Float64},transpose(pixel)),
                             convert(Array{Float64},centroids))

    qnt = kmeansResults.assignments
    centers = kmeansResults.centers
    if(centers[1] == centers[2])
        # The image has only a value
        index = findmin([abs(centroids[1]-centers[1]),abs(centroids[2]-centers[1])])[2]
        qnt = fill(index, size(qnt))
    end

    # Reshaping quantization result
    centers_idx = reshape(qnt, size(image3d[page],1), size(image3d[page],2))

    # Inserting quantized values into 3d image array
    tmp = Array{UInt8, size(image3d[page],1), size(image3d[page],2)}

    for j in 1:size(image3d[1],2)
```

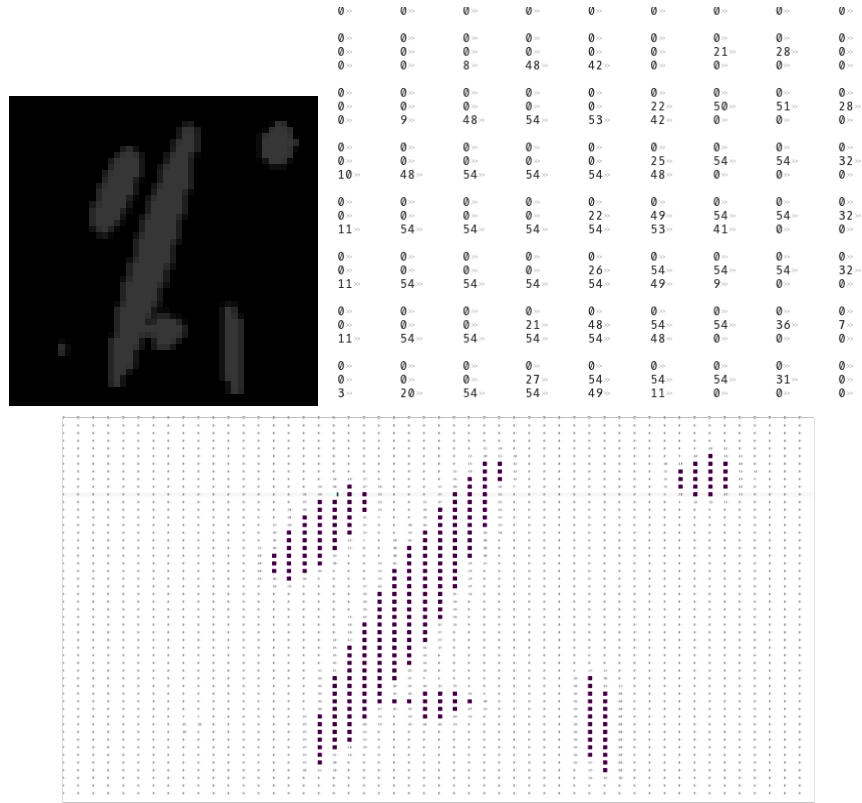


Figure 6: Reading raw data from image. (a) Original greyscale image (b) A view of raw data array (c) The entire raw data array with main color highlighted

```

for i in 1:size(image3d[1],1)
    tmp[i,j] = centroids[centers_idx[i,j]]
end
end

```

```

image3d[page] = tmp ◇

```

Fragment referenced in 24.

We can see that sometimes the `Clustering.jl` library returns the same values for both centroid centers. This could happen when the images is completely empty or it has only colored pixels. So, we need to check this cases and fill the assignments array `qnt` with the right values based on the `centroids` parameter.

This is the complete code:

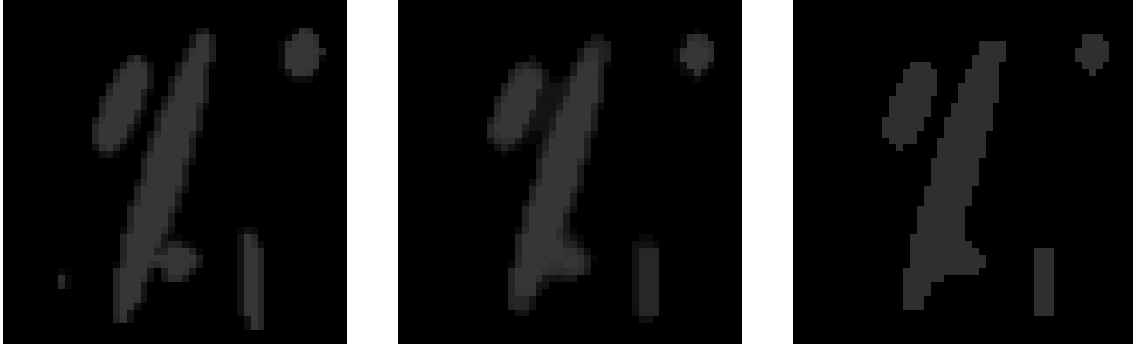


Figure 7: Image transformation. (a) Original greyscale image (b) Denoised image (c) Two-colors image

```

< Pixel transformation 24 > ≡
function pngstack2array3d(path, minSlice, maxSlice, centroids)
    """
    Import a stack of PNG images into a 3d array

    path: path of images directory
    minSlice and maxSlice: number of first and last slice
    centroids: centroids for image segmentation
    """

    # image3d contains all images values
    image3d = Array(Array{UInt8,2}, 0)

    debug("maxSlice = ", maxSlice, " minSlice = ", minSlice)
    files = readdir(path)

    for slice in minSlice : (maxSlice - 1)
        debug("slice = ", slice)
        imageFilename = string(path, files[slice + 1])
        debug("image name: ", imageFilename)
        < Read raw data 22a >
        debug("imArray size: ", size(imArray))

        # Inserting page on another list and reshaping
        push!(image3d, imArray)
    end

    # Quantization
    for page in 1:length(image3d)

```

```

    < Clustering images 22b >

    end

    return image3d
end
◇

```

Fragment referenced in 89b.

5 ImagesConversion

Now we will study the most important module for this package: `ImagesConversion`. It has the responsibility of doing the entire conversion process delegating tasks to the other modules.

5.1 General algorithm

Now we will examine, in a general way, the algorithm used for conversion from a two-dimensional to a three-dimensional representation of our biomedical models.

We have already seen in section 4 how to get information from a png image, obtaining arrays with only two values; one for the **background** color and one for **foreground** color. This is only the first step of the complete conversion process.

Now we focus only on a single image of the stack. Our two-dimensional representation, consists of pixels of two different colors (where only the one associated with foreground is significant); so we can obtain a three-dimensional representation simply replacing every foreground pixel with a small cube. Focusing on the entire stack of images, the full three-dimensional representation can be obtained simply overlapping all the image representations

This algorithm is very simple, however we does not considered problems concerning lack of memory. In fact, we could have images so big that we cannot build these models entirely in memory; moreover they would require a lot of CPU time for computation. A good solution to these problems consists in taking our representation based on images and divide according to a **grid**.

So, instead of converting the entire model with a unique process, we can subdivide the input among a lot of processes, where every process will execute the conversion process on

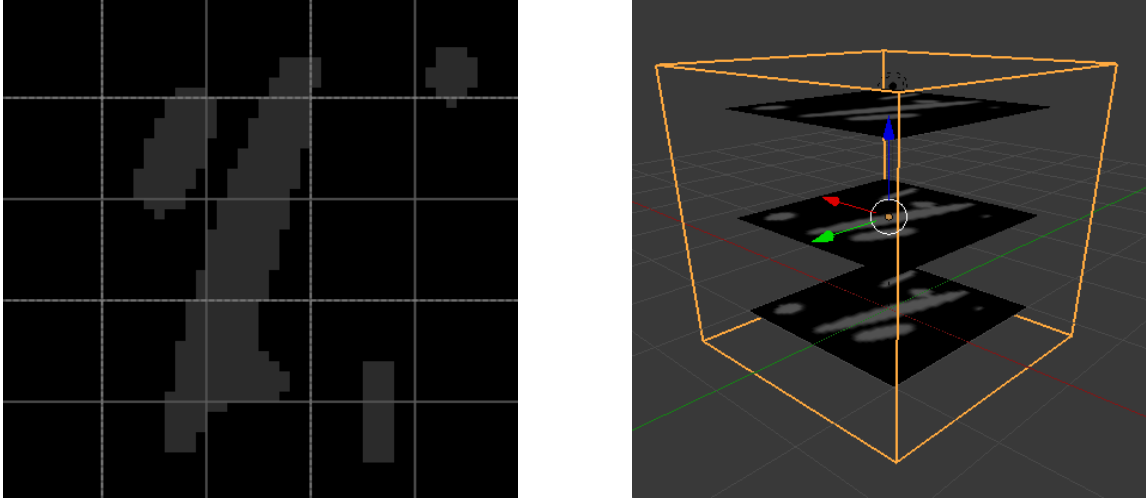


Figure 8: The grid used for parallel computation (a) 2D grid on a single image (b) 3D grid for the stack of images

a small number of **blocks** according to the grid subdivision.

Summing up we can define the following terms, which will be used in next parts of this documentation:

- **Grid:** It is the subdivision of the entire stack of images, with sizes defined by the user. They should be powers of two (for increasing performance during border matrix computation which we will see in section 6)
- **Block:** It is a single cell of the grid
- **xBlock:** It is the x-coordinate of a block
- **yBlock:** It is the y-coordinate of a block
- **zBlock:** It is the z-coordinate of a block

$xBlock$ and $yBlock$ are defined on a single image, while $zBlock$ is defined on different images; in the code it will often be replaced by terms **StartImage** and **EndImage**, which indicate the first image and the last image of that block respectively.

In next subsections we will examine the conversion algorithm in detail, showing what happens for every block of the grid.

5.2 Module imports

These are modules needed for this part of the package and the public functions exported.

```
< modules import ImagesConversion 27a > ≡
import GenerateBorderMatrix
import PngStack2Array3dJulia
import Lar2Julia
import Model20bj
import LARUtils
import Smoother

using Logging

export images2LARModel
◇
```

Fragment referenced in 87a.

5.3 Data preparation

As a first thing, we will see how to prepare our data for conversion process. Firstly we need to compute the LAR boundary operator for finding boundaries of our cells (for the generation see section 6) getting width and height from our images.

Later we can start conversion with all these parameters calling `startImageConversion` function, which will be explained in next subsection.

```
< main function for ImagesConversion 27b > ≡
function images2LARModel(nx, ny, nz, bestImage,
                        inputDirectory, outputDirectory,
                        parallelMerge)
    """
    Convert a stack of images into a 3d model
    """

    info("Starting model creation")

    numberOfClusters = 2 # Number of clusters for
                        # images segmentation

    imageWidth, imageHeight = PngStack2Array3dJulia.getImageData(
                                string(inputDirectory, bestImage))
    imageDepth = length(readdir(inputDirectory))
```

```

# Computing border matrix
info("Computing border matrix")
try
  mkdir(string(outputDirectory, "BORDERS"))
catch
end
borderFilename = GenerateBorderMatrix.getOriented3BorderPath(
  string(outputDirectory, "BORDERS"), nx, ny, nz)

# Starting images conversion and border computation
info("Starting images conversion")
startImageConversion(inputDirectory, bestImage, outputDirectory, borderFilename,
  imageHeight, imageWidth, imageDepth,
  nx, ny, nz,
  numberOfClusters, parallelMerge)

end
◇

```

Fragment referenced in 87a.

5.4 Conversion pipeline

Now we can see how conversion of images works. In section 5.1 we have seen how to execute the single conversion of a pixel into a voxel using our grid for parallel computation. However, with that algorithm, we obtain models with internal boundaries between blocks and with squared edge. So we need to create a **conversion pipeline** which will progressively refine our models. In Figure 9 there are the steps used for our conversion

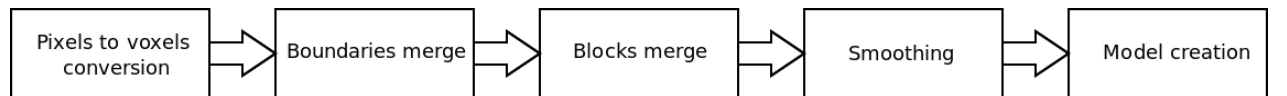


Figure 9: Images conversion pipeline

Every single step of the pipeline, is executed in parallel for every block of the grid; so we need a general purpose function for blocks iteration which will take as a parameter a function that will execute it. So we can define the `iterateOnBlocks` function which takes the following parameters:

- **inputDirectory**: Directory which contains input files for the process function
- **imageHeight, imageWidth, imageDepth**: Sizes of the stack of images

- **imageDx, imageDy, imageDz:** Sizes of the grid
- **processFunction:** Function that contains instructions for execution of a single step of the pipeline for a single block
- **outputDirectory:** Directory which will contains the output
- **centroids:** Centroids from the best image
- **boundaryMat:** Boundary operator for the chosen grid

This function will iterate on all blocks of the image grid executing the process function, which will be different for every pipeline step. This is the code used:

(parallel block iteration 29) ≡

```
function iterateOnBlocks(inputDirectory,
                        imageHeight, imageWidth, imageDepth,
                        imageDx, imageDy, imageDz,
                        processFunction, outputDirectory,
                        centroidsCalc, boundaryMat)

    """
    Simple function that iterates on blocks for executing
    a task described by a processFunction

    inputDirectory: Directory which contains input files for the process function
    imageHeight, imageWidth, imageDepth: Images sizes
    imageDx, imageDy, imageDz: Sizes of cells grid
    processFunction: Function that will be executed on a separate task on
    the entire z-Block
    outputDirectory: Directory which will contains the output
    centroidsCalc: Centroids from the best image
    boundaryMat: Boundary operator for the chosen grid
    """

    beginImageStack = 0
    endImage = beginImageStack

    tasks = Array(RemoteRef, 0)
    for zBlock in 0:(imageDepth / imageDz - 1)
        startImage = endImage
        endImage = startImage + imageDz
        task = @spawn processFunction(inputDirectory,
                                    startImage, endImage,
                                    imageDx, imageDy,
                                    imageWidth, imageHeight,
                                    outputDirectory,
                                    centroidsCalc, boundaryMat)
```

```

        push!(tasks, task)
    end

    # Waiting for tasks
    for task in tasks
        wait(task)
    end
end ◇

```

Fragment referenced in 87a.

First of all we need to iterate on the grid finding the `zBlock` coordinate; we saw earlier that the `imageDz` parameter must be a divisor of the image depth, so we will have exactly $imageDepth/imageDz$ blocks on the `z` coordinate. Moreover, at every `zBlock` correspond a `startImage` and an `endImage` where $endImage - startImage = imageDz$.

Now we can simply parallelize the conversion process spawning a new process for every `zBlock`, so we open at most `imageDz` images for process. Finally, we have to wait for tasks completion.

Now we can see the entire pipeline for images conversion.

First of all we need to compute the centroids from the best image using module `PngStack2Array3dJulia` and get the previously computed border matrix in csc sparse array format

```

⟨ compute centroids and get border matrix 30 ⟩ ≡
    # Create clusters for image segmentation
    info("Computing image centroids")
    debug("Best image = ", bestImage)
    centroidsCalc = PngStack2Array3dJulia.calculateClusterCentroids(sliceDirectory,
                                                                    bestImage, numberOfClusters)
    debug(string("centroids = ", centroidsCalc))

    try
        mkdir(string(outputDirectory, "BORDERS"))
    catch
    end
    debug("Opening border file: border_", imageDx, "-", imageDy, "-", imageDz, ".json")
    boundaryMat = GenerateBorderMatrix.getBorderMatrix(
                                                                    string(outputDirectory, "BORDERS/", "border_",
                                                                    imageDx, "-", imageDy, "-", imageDz, ".json"))
◇

```

Fragment referenced in 31b.

Now we can start the pipeline:

```
< pipeline conversion 31a > ≡
    # Starting pipeline conversion
    info("Starting images conversion")
    < pixels to voxels conversion step 36 >

    info("Merging boundaries")
    < boundaries merge step 39a >

    info("Merging blocks")
    < block merge step 41a >

    info("Smoothing models")
    < smoothing step 45a >

    info("Merging obj models")
    < final file merge 45b >
    end ◇
```

Fragment referenced in 31b.

As we can see, last pipeline step does not require iteration on all grid blocks. This is the code for the function that starts the pipeline, with the parts explained earlier:

```
< start conversion of images 31b > ≡

function startImageConversion(sliceDirectory, bestImage, outputDirectory, borderFilename,
                             imageHeight, imageWidth, imageDepth,
                             imageDx, imageDy, imageDz,
                             numberOfClusters, parallelMerge)
    """
    Support function for converting a stack of images into a model

    sliceDirectory: directory containing the image stack
    imageForCentroids: image chosen for centroid computation
    """

    < compute centroids and get border matrix 30 >
```



```

    < pipeline conversion 31a >
end
◇

```

Fragment referenced in 87a.

5.4.1 Images conversion step

Now we will focus on the first step of our pipeline conversion: *images conversion*. First thing to do is read an image calling the `PngStack2Array3dJulia`, after that is necessary to sort the centroid array for choosing correct background and foreground pixels.

```

< image read and centroids sort 32a > ≡
    info("Transforming png data into 3d array")
    theImage = PngStack2Array3dJulia.pngstack2array3d(sliceDirectory,
                                                    startImage, endImage, centroids)

    centroidsSorted = sort(vec(reshape(centroids, 1, 2)))
    background = centroidsSorted[1]
    foreground = centroidsSorted[2]
    debug(string("background = ", background, " foreground = ", foreground))
◇

```

Fragment referenced in 35b.

Now we can start iterating on other blocks of the grid:

```

< block iteration 32b > ≡
    for xBlock in 0:(imageWidth / imageDx - 1)
        for yBlock in 0:(imageHeight / imageDy - 1)

            xStart = xBlock * imageDx
            yStart = yBlock * imageDy
            zStart = startImage

            xEnd = xStart + imageDx
            yEnd = yStart + imageDy

            imageDz = length(theImage)

            debug("*****")

```

```

debug(string("xStart = ", xStart, " xEnd = ", xEnd))
debug(string("yStart = ", yStart, " yEnd = ", yEnd))
debug("theImage dimensions: ", size(theImage)[1], " ",
      size(theImage[1])[1], " ", size(theImage[1])[2]) ◇

```

Fragment referenced in 35b.

Here $xStart$ and $yStart$ are the absolute coordinates of the model and are calculated from the block coordinates. Now we can get the current slice for the entire image (with size $(imageDx, imageDy, imageDz)$), check values for every single pixel into it and, if it represents a foreground point, put it into an array called `chain3D`. This structure contains indexes of the linearized array created from the matrix. In Figure 10 there is a sample conversion from the matrix to the array

$$\begin{pmatrix} 0^0 & 0^2 \\ 0^1 & 0^3 \end{pmatrix} \begin{pmatrix} 46^4 & 0^6 \\ 46^5 & 46^7 \end{pmatrix} \rightarrow 0^0 \ 0^1 \ 0^2 \ 0^3 \ 46^4 \ 46^5 \ 0^6 \ 46^7 \\
 \begin{pmatrix} 0^0 & 0^2 \\ 0^1 & 0^3 \end{pmatrix} \begin{pmatrix} 0^4 & 46^6 \\ 46^5 & 46^7 \end{pmatrix} \rightarrow 0^0 \ 0^1 \ 0^2 \ 0^3 \ 0^4 \ 46^5 \ 46^6 \ 46^7$$

Figure 10: Transformation of a matrix resulting from a 2x2x2 grid into a linearized array (with cells indexes) (a) First example (b) Second example

As we can see from that figure, from a 2x2x2 grid we can obtain eight values for the single block (or **cell**), where the indexes for the foreground pixels represent indexes of non-empty cells in a 2x2x2 cuboidal geometry. This is the code for getting foreground pixels:

```

⟨ get image slice 33 ⟩ ≡
chains3D = Array(Int, 0)
for z in 1 : imageDz
  for y in 1 : imageDy
    for x in 1 : imageDx
      if(theImage[z][x + xStart, y + yStart] == foreground)
        index = x - 1 + (y - 1) * imageDx + (z - 1) * (imageDx * imageDy)
        push!(chains3D, index)
      end
    end
  end
end
end ◇

```

Fragment referenced in 35b.

Now that we have full cells for the geometry, we can convert them into a *LAR model*. In particular, we are interested in cell boundaries for every block (as we want to obtain only the boundaries for the final model) so we can call function `larBoundaryChain` from `Lar2Julia` module (which will be explained in section 7). In Figure 11 there are some examples of models extracted from a single $2 \times 2 \times 2$ block.

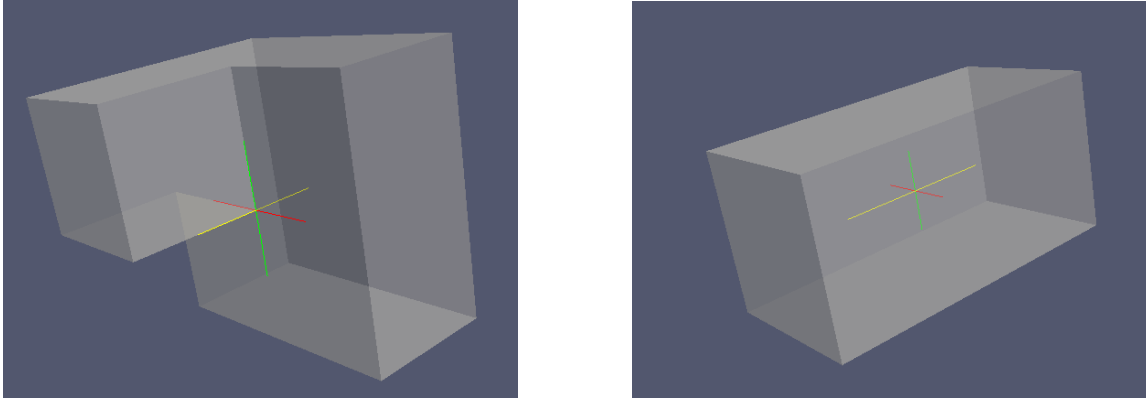


Figure 11: Sample models of $2 \times 2 \times 2$ blocks

After model computation, next step is getting vertices and faces from model cells writing results to file. However, as we have already said, we are only interested in boundaries of the final model while now we have only boundaries of a single block. Consequently, we have to separate boundaries from the inner faces of the block on different files (boundaries separation will be explained in section 8). As we can see later, we will merge boundaries together deleting common faces on both block borders, obtaining a model without internal faces. These are pieces of code for getting the inner block model with the boundaries and for file writing:

```

⟨get inner model and boundaries 34⟩ ≡
    models = LARUtils.computeModelAndBoundaries(imageDx, imageDy, imageDz,
                                                xStart, yStart, zStart, objectBoundaryChain)

    V, FV = models[1][1] # inside model
    V_left, FV_left = models[2][1]
    V_right, FV_right = models[3][1] # right boundary
    V_top, FV_top = models[4][1] # top boundary
    V_bottom, FV_bottom = models[5][1] # bottom boundary
    V_front, FV_front = models[6][1] # front boundary
    V_back, FV_back = models[7][1] # back boundary
    ◇

```

Fragment referenced in 35b.

< write block models to file 35a > ≡

```
# Writing all models on disk
model_outputFilename = string(outputDirectory, "MODELS/model_output_", xBlock,
                                "-", yBlock, "_", startImage, "_", endImage)
Model2Obj.writeToObj(V, FV, model_outputFilename)

left_outputFilename = string(outputDirectory, "MODELS/left_output_", xBlock,
                              "-", yBlock, "_", startImage, "_", endImage)
Model2Obj.writeToObj(V_left, FV_left, left_outputFilename)

right_outputFilename = string(outputDirectory, "MODELS/right_output_", xBlock,
                               "-", yBlock, "_", startImage, "_", endImage)
Model2Obj.writeToObj(V_right, FV_right, right_outputFilename)

top_outputFilename = string(outputDirectory, "MODELS/top_output_", xBlock,
                             "-", yBlock, "_", startImage, "_", endImage)
Model2Obj.writeToObj(V_top, FV_top, top_outputFilename)

bottom_outputFilename = string(outputDirectory, "MODELS/bottom_output_", xBlock,
                                "-", yBlock, "_", startImage, "_", endImage)
Model2Obj.writeToObj(V_bottom, FV_bottom, bottom_outputFilename)

front_outputFilename = string(outputDirectory, "MODELS/front_output_", xBlock,
                               "-", yBlock, "_", startImage, "_", endImage)
Model2Obj.writeToObj(V_front, FV_front, front_outputFilename)

back_outputFilename = string(outputDirectory, "MODELS/back_output_", xBlock,
                              "-", yBlock, "_", startImage, "_", endImage)
Model2Obj.writeToObj(V_back, FV_back, back_outputFilename) ◇
```

Fragment referenced in 35b.

This is the `processFunction` for this pipeline step

< image conversion process 35b > ≡

```
function imageConversionProcess(sliceDirectory,
                                startImage, endImage,
                                imageDx, imageDy,
                                imageWidth, imageHeight,
                                outputDirectory,
                                centroids, boundaryMat)

    """
    Support function for converting a stack of image on a single
    independent process
```

```

"""

< image read and centroids sort 32a >

< block iteration 32b >

    < get image slice 33 >

    if(length(chains3D) != 0)
        # Computing boundary chain
        debug("chains3d = ", chains3D)
        debug("Computing boundary chain")
        objectBoundaryChain = Lar2Julia.larBoundaryChain(boundaryMat, chains3D)
        debug("Converting models into obj")
        try
            mkdir(string(outputDirectory, "MODELS"))
        catch
        end
        < get inner model and boundaries 34 >

        < write block models to file 35a >
    else
        debug("Model is empty")
    end
end
end
end
end ◇

```

Fragment referenced in 87a.

This is the code for starting this pipeline step:

```

< pixels to voxels conversion step 36 > ≡
    @time iterateOnBlocks(sliceDirectory,
        imageHeight, imageWidth, imageDepth,
        imageDx, imageDy, imageDz,
        imageConversionProcess, outputDirectory,
        centroidsCalc, boundaryMat) ◇

```

Fragment referenced in 31a.

5.4.2 Boundaries merge step

Next step of our pipeline consists in *boundaries merge*. In fact, we have already seen that for every non-empty cell we create files for the inner parts and for the boundaries of the block. So if we want a final model without boundaries between internal blocks, we need to merge them removing duplicated faces on both sides (see Section 8.6 for a better explanation of this step). The following is the `processFunction`:

\langle *boundary merge process function 37* $\rangle \equiv$

```
function mergeBoundariesProcess(modelDirectory,
                                startImage, endImage,
                                imageDx, imageDy,
                                imageWidth, imageHeight,
                                outputDirectory = None,
                                centroidsCalc = None, boundaryMat = None)

"""
Helper function for mergeBoundaries.
It is executed on different processes

modelDirectory: Directory containing model files
startImage: Block start image
endImage: Block end image
imageDx, imageDy: x and y sizes of the grid
imageWidth, imageHeight: Width and Height of the image
"""
for xBlock in 0:(imageWidth / imageDx - 1)
    for yBlock in 0:(imageHeight / imageDy - 1)

        # Merging right Boundary
        firstPath = string(modelDirectory, "/right_output_", xBlock, "-", yBlock,
                           "_", startImage, "_", endImage)
        secondPath = string(modelDirectory, "/left_output_", xBlock, "-", yBlock + 1,
                             "_", startImage, "_", endImage)
        mergeBoundariesAndRemoveDuplicates(firstPath, secondPath)

        # Merging top boundary
        firstPath = string(modelDirectory, "/top_output_", xBlock, "-", yBlock,
                           "_", startImage, "_", endImage)
        secondPath = string(modelDirectory, "/bottom_output_", xBlock, "-", yBlock,
                             "_", endImage, "_", endImage + (endImage - startImage))
        mergeBoundariesAndRemoveDuplicates(firstPath, secondPath)

        # Merging front boundary
        firstPath = string(modelDirectory, "/front_output_", xBlock, "-", yBlock,
                           "_", startImage, "_", endImage)
        secondPath = string(modelDirectory, "/back_output_", xBlock + 1, "-", yBlock,
```

```

        "_", startImage, "_", endImage)
    mergeBoundariesAndRemoveDuplicates(firstPath, secondPath)
end
end
end ◇

```

Fragment referenced in 87a.

For every block we do the following merges:

- right boundary with the left boundary of the next block on the right
- top boundary with the bottom boundary of the next block on the top
- front boundary with the back boundary of the next block on the front

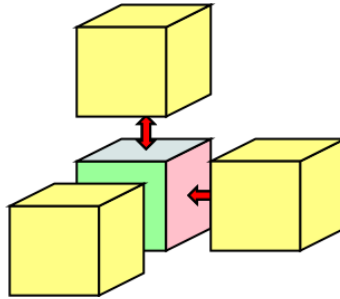


Figure 12: Merging of boundary faces. For a single block we need adjacent blocks on the right, top and front

all merges are executed by the function `mergeBoundariesAndRemoveDuplicates` which does the work calling the `Model2Obj` and `LARUtils` libraries for loading and cleaning of the boundaries models.

```

⟨merge boundaries utility function 38⟩ ≡
function mergeBoundariesAndRemoveDuplicates(firstPath, secondPath)
    """
    Merge two boundary files removing common faces between
    them

    firstPath, secondPath: Prefix of paths to merge
    """

    firstPathV = string(firstPath, "_vtx.stl")

```

```

firstPathFV = string(firstPath, "_faces.stl")

secondPathV = string(secondPath, "_vtx.stl")
secondPathFV = string(secondPath, "_faces.stl")

if(isfile(firstPathV) && isfile(secondPathV))

    V, FV = Model2Obj.getModelsFromFiles([firstPathV, secondPathV],
                                         [firstPathFV, secondPathFV])
    V, FV = LARUtils.removeVerticesAndFacesFromBoundaries(V, FV)

    # Writing model to file
    rm(firstPathV)
    rm(firstPathFV)
    rm(secondPathV)
    rm(secondPathFV)
    Model2Obj.writeToObj(V, FV, firstPath)
end
end ◇

```

Fragment referenced in 87a.

This is the code used to start this pipeline step:

```

⟨ boundaries merge step 39a ⟩ ≡
    @time iterateOnBlocks(string(outputDirectory, "MODELS"),
                          imageHeight, imageWidth, imageDepth,
                          imageDx, imageDy, imageDz,
                          mergeBoundariesProcess, None,
                          None, None) ◇

```

Fragment referenced in 31a.

5.4.3 Block merge step

At this step of the computation, we have files with the inner parts of a single block model and the remaining boundaries. Now we need to merge the blocks removing double vertices and faces, so we can save space and prepare our model to the *smoothing step*. This is the code of the `processFunction`:

```

⟨ Block merge process function 39b ⟩ ≡

```



```

function mergeBlocksProcess(modelDirectory,
                           startImage, endImage,
                           imageDx, imageDy,
                           imageWidth, imageHeight,
                           outputDirectory = None,
                           centroidsCalc = None, boundaryMat = None)

"""
Helper function for mergeBlocks.
It is executed on different processes

modelDirectory: Directory containing model files
startImage: Block start image
endImage: Block end image
imageDx, imageDy: x and y sizes of the grid
imageWidth, imageHeight: Width and Height of the image
"""
for xBlock in 0:(imageWidth / imageDx - 1)
  for yBlock in 0:(imageHeight / imageDy - 1)

    blockCoordsV = string(xBlock, "-", yBlock, "_", startImage,
                          "_", endImage, "_vtx.stl")
    blockCoordsFV = string(xBlock, "-", yBlock, "_", startImage,
                          "_", endImage, "_faces.stl")

    arrayV = [string(modelDirectory, "/left_output_", blockCoordsV),
              string(modelDirectory, "/right_output_", blockCoordsV),
              string(modelDirectory, "/top_output_", blockCoordsV),
              string(modelDirectory, "/bottom_output_", blockCoordsV),
              string(modelDirectory, "/front_output_", blockCoordsV),
              string(modelDirectory, "/back_output_", blockCoordsV),
              string(modelDirectory, "/model_output_", blockCoordsV)]

    arrayFV = [string(modelDirectory, "/left_output_", blockCoordsFV),
              string(modelDirectory, "/right_output_", blockCoordsFV),
              string(modelDirectory, "/top_output_", blockCoordsFV),
              string(modelDirectory, "/bottom_output_", blockCoordsFV),
              string(modelDirectory, "/front_output_", blockCoordsFV),
              string(modelDirectory, "/back_output_", blockCoordsFV),
              string(modelDirectory, "/model_output_", blockCoordsFV)]

    V, FV = Model2Obj.getModelsFromFiles(arrayV, arrayFV)
    V, FV = LARUtils.removeDoubleVerticesAndFaces(V, FV, 0)
    for i in 1:length(arrayV)
      if(isfile(arrayV[i]))
        rm(arrayV[i])
        rm(arrayFV[i])

```

```

        end
    end

    Model2Obj.writeToObj(V, FV, string(modelDirectory, "/model_output_",
        xBlock, "-", yBlock, "_", startImage, "_", endImage))

    end
end
end ◇

```

Fragment referenced in 87a.

For a better explanation of the `LARUtils` function that remove duplicated vertices, you can see Section [8.4](#)

This is the code for block merge starting

```

⟨ block merge step 41a ⟩ ≡
    @time iterateOnBlocks(string(outputDirectory, "MODELS"),
        imageHeight, imageWidth, imageDepth,
        imageDx, imageDy, imageDz,
        mergeBlocksProcess, None,
        None, None) ◇

```

Fragment referenced in 31a.

5.4.4 Smoothing step

Now we have obtained models without internal boundaries between blocks and without double vertices and faces in a single block. However this partial model has squared edges, so we need to smooth them. The `processFunction` for this step, is the following:

```

⟨ Smooth block process function 41b ⟩ ≡
    function smoothBlocksProcess(modelDirectory,
        startImage, endImage,
        imageDx, imageDy,
        imageWidth, imageHeight,
        outputDirectory = None,
        centroidsCalc = None, boundaryMat = None)

    """"
    Smoothes a block in a single process

    modelDirectory: Path of the directory containing all blocks

```

```

        that will be smoothed
startImage, endImage: start and end image for this block
imageDx, imageDy: sizes of the grid
imageWidth, imageHeight: sizes of the images
"""

for xBlock in 0:(imageWidth / imageDx - 1)
  for yBlock in 0:(imageHeight / imageDy - 1)

    # Loading the current block model
    blockFileV = string(modelDirectory, "/model_output_", xBlock, "-", yBlock,
                        "_", startImage, "_", endImage, "_vtx.stl")
    blockFileFV = string(modelDirectory, "/model_output_", xBlock, "-", yBlock,
                        "_", startImage, "_", endImage, "_faces.stl")

    if isfile(blockFileV)
      # Loading only model of the current block
      blockModelV, blockModelFV = Model2Obj.getModelsFromFiles([blockFileV], [blockFileFV])
      blockModelV, blockModelFV = LARUtils.removeDoubleVerticesAndFaces(blockModelV,
                                                                    blockModelFV, 0)

      # Loading a unique model from this block and its adjacents
      modelsFiles = Array(String, 0)
      for x in xBlock - 1:xBlock + 1
        for y in yBlock - 1:yBlock + 1
          for z in range(startImage - (endImage - startImage), (endImage - startImage), 3)
            push!(modelsFiles, string(modelDirectory, "/model_output_",
                                      x, "-", y, "-", z, "-", z + (endImage - startImage)))
          end
        end
      end

      modelsFilesV = map((s) -> string(s, "_vtx.stl"), modelsFiles)
      modelsFilesFV = map((s) -> string(s, "_faces.stl"), modelsFiles)

      modelV, modelFV = Model2Obj.getModelsFromFiles(modelsFilesV, modelsFilesFV)
      modelV, modelFV = LARUtils.removeDoubleVerticesAndFaces(modelV, modelFV, 0)

      # Now I have to save indices of vertices of the current block model
      blockVerticesIndices = Array{Int, 1}(0)
      for i in 1:length(blockModelV)
        for j in 1:length(modelV)
          if blockModelV[i] == modelV[j]
            push!(blockVerticesIndices, j)
          end
        end
      end
    end
  end
end

```

```

# Now I can apply smoothing on this model
V_sm, FV_sm = Smoother.smoothModel(modelV, modelFV)

# Now I have to get only block vertices and save them on the new model
V_final = Array(Array{Float64}, 0)
for i in blockVerticesIndices
    push!(V_final, V_sm[i])
end
outputFilename = string(modelDirectory, "/smoothed_output_", xBlock, "-",
                        yBlock, "_", startImage, "_", endImage)
Model2Obj.writeToObj(V_final, blockModelFV, outputFilename)
end
end
end
end
end ◇

```

Fragment referenced in 87a.

An explanation of the smoothing algorithm used there, can be found in Section 9.2. What we need to remember here, is the importance of having the adjacent vertices for every vertex of our block. In fact, according to the chosen smoothing algorithm, every vertex is replaced with a new one with coordinates computed from the mean positions of its adjacent. However, loading of the entire model into memory cannot be done because of its sizes; so we created a simple algorithm which loads only near blocks to the current one. In fact, for every block we want to smooth, we load the twenty six adjacent blocks on all directions and the chosen one. We create a unique model with it (removing double vertices and faces) and then smoothing it with the algorithm in **Smoother** module. Finally we save only smoothed vertices for the chosen block and continue with the other blocks. In Figure 13 there is a graphical explanation for the algorithm.

Moreover, this **processFunction** can only execute a single iteration of the smoothing algorithm, so we need a function that can be able to execute more times the algorithm:

```

⟨ execute smoothing function 43 ⟩ ≡
function smoothBlocks(modelDirectory,
                      imageHeight, imageWidth, imageDepth,
                      imageDx, imageDy, imageDz)
    """
    Smoothes all blocks of the
    model
    """

```

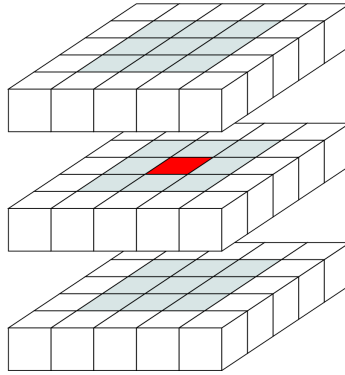


Figure 13: Smoothing of a single block. The red block at the center of the figure is the current one, while the other twenty six colored ones are the blocks that will be part of the model which will be smoothed for this iteration

```

iterations = 1
for i in 1:iterations
    info("Iteration ", i)

    iterateOnBlocks(modelDirectory,
                    imageHeight, imageWidth, imageDepth,
                    imageDx, imageDy, imageDz,
                    smoothBlocksProcess,
                    None, None, None)

    # Moving smoothed file for next iterations

    beginImageStack = 0
    endImage = beginImageStack
    for zBlock in 0:(imageDepth / imageDz - 1)
        startImage = endImage
        endImage = startImage + imageDz
        for xBlock in 0:(imageWidth / imageDx - 1)
            for yBlock in 0:(imageHeight / imageDy - 1)

                f_V = string(modelDirectory, "/smoothed_output_", xBlock, "-", yBlock, "_",
                             startImage, "_", endImage, "_vtx.stl")
                f_FV = string(modelDirectory, "/smoothed_output_", xBlock, "-", yBlock, "_",
                              startImage, "_", endImage, "_faces.stl")

                if(isfile(f_V))
                    if VERSION >= v"0.4"
                        mv(f_V, replace(f_V, "smoothed", "model"), remove_destination = true)

```


Fragment referenced in 31a.

6 GenerateBorderMatrix

This module has the responsibility for the generation of the border matrix operator for models boundary computation.

6.1 Module imports

These are modules needed for this part of the package and the public functions exported

```
< modules import GenerateBorderMatrix 46 > ≡
import LARUtils
using PyCall

import JSON

export computeOriented3Border, writeBorder, getOriented3BorderPath, getBorderMatrix

@pyimport sys
# Search for python modules in package folder
unshift!(PyVector(pyimport("sys")["path"]), Pkg.dir("ImagesToLARModel/src"))
@pyimport larcc # Importing larcc from local folder
◇
```

Fragment referenced in 87b.

We can notice some lines for importing `larcc` python library, which will be used in subsection [6.5](#)

6.2 Get border matrix from file

As we have already seen in previous sections, we need to compute boundaries for every block of the model grid. This can be done using the topological boundary operator from LAR package. However, the resulting matrix depends only on grid sizes; so it could be reused for other models. Consequently first time we need a border operator we compute it and then save it on disk for next conversions. This function does that work searching for a file containing the border and, if it does not exist, calculate and save it:

$\langle \text{get Border matrix 47a} \rangle \equiv$

```
function getOriented3BorderPath(borderPath, nx, ny, nz)
    """
    Try reading 3-border matrix from file. If it fails matrix
    is computed and saved on disk in JSON format

    borderPath: path of border directory
    nx, ny, nz: image dimensions
    """

    filename = string(borderPath, "/border_", nx, "-", ny, "-", nz, ".json")
    if !isfile(filename)
        border = computeOriented3Border(nx, ny, nz)
        writeBorder(border, filename)
    end
    return filename
end ◇
```

Fragment referenced in 87b.

6.3 Write border matrix on file

We have already seen that for performance reasons border operator matrix is saved on file; here we will see code used for this scope. Firstly, we have defined a function `writeBorder`, which takes as parameters a `PyObject` containing a matrix (computed in subsection 6.4) and the output file path. When porting of `larcc` library will be completed, code for conversion of python csr matrix into csc julia matrix will not be necessary.

$\langle \text{write Border matrix 47b} \rangle \equiv$

```
function writeBorder(boundaryMatrix, outputFile)
    """
    Write 3-border matrix on json file

    boundaryMatrix: matrix to write on file
    outputFile: path of the outputFile
    """

    fullBorder = pycall(boundaryMatrix["toarray"], PyAny)
    cscBorder = sparse(fullBorder)
    row = findn(cscBorder)[1]
    col = findn(cscBorder)[2]
    data = nonzeros(cscBorder)
```



```

matrixObj = MatrixObject(0, 0, row, col, data)

outfile = open(string(outputFile), "w")
JSON.print(outfile, matrixObj)
close(outfile)
end ◇

```

Fragment referenced in 87b.

We can see that, in final JSON file, we write an object called `MatrixObject` which has the following definition:

\langle *Matrix object for JSON file 48a* $\rangle \equiv$

```

type MatrixObject
  ROWCOUNT
  COLCOUNT
  ROW
  COL
  DATA
end ◇

```

Fragment referenced in 87b.

The most important fields of this object are the last three ones; the first two contain all coordinates of the non-zero elements, the last contains all non-zero elements of the sparse matrix. So considering the full matrix V we will have that $S[ROW[k], COL[k]] = V[k]$.

6.4 Compute border matrix

Here we can see code used for computation of the border operator. As we can see, we call the python `larcc` module, from the `LAR` module, which returns a `PyObject` containing a *sparse csr matrix*. In next versions this function will be probably changed and the code for boundary computation will be moved in `LAR2Julia` module (also transforming all csr matrix in csc matrix) avoiding python calls.

\langle *compute border matrix 48b* $\rangle \equiv$

```

# Compute the 3-border operator
function computeOriented3Border(nx, ny, nz)
  """

```

```

    Compute the 3-border matrix using a modified
    version of larcc
    """
    V, bases = LARUtils.getBases(nx, ny, nz)
    boundaryMat = larcc.signedCellularBoundary(V, bases)
    return boundaryMat

end ◇

```

Fragment referenced in 87b.

6.5 Transform border matrix

Last function we will see, extracts the `MatrixObject` in Section 6.3 converting it into a common Julia csc sparse matrix

```

⟨ transform border matrix in csc format 49 ⟩ ≡
function getBorderMatrix(borderFilename)
    """
    Get the border matrix from json file and convert it in
    CSC format
    """
    # Loading borderMatrix from json file
    borderData = JSON.parsefile(borderFilename)

    # Converting Any arrays into Int arrays
    row = Array{Int64, length(borderData["ROW"])}
    col = Array{Int64, length(borderData["COL"])}
    data = Array{Int64, length(borderData["DATA"])}

    for i in 1: length(borderData["ROW"])
        row[i] = borderData["ROW"][i]
    end

    for i in 1: length(borderData["COL"])
        col[i] = borderData["COL"][i]
    end

    for i in 1: length(borderData["DATA"])
        data[i] = borderData["DATA"][i]
    end
    return sparse(row, col, data)
end ◇

```

Fragment referenced in 87b.

7 Lar2Julia

This module contains functions used in LAR library which are converted using Julia syntax. Next versions of the software will contain more and more functions from the original LAR library (which is written in python)

7.1 Module imports

These are modules used for Lar2Julia and the public functions

```
< modules import Lar2Julia 50a >  $\equiv$   
import JSON  
  
using Logging  
  
export larBoundaryChain, cscChainToCellList, relationshipListToCSC  $\diamond$ 
```

Fragment referenced in 88a.

7.2 Get boundary chain from a model

Now we will observe how to compute the boundary chain of a LAR model given the list of non-empty cells and the boundary operator stored as a csc sparse matrix. This algorithm is very simply: firstly we need to convert the list of cells into a sparse array containing the LAR model. So, the resulting array (which will be called `cscChain`) will contain a one for every `cscChain[i][1] $\forall i \in$ brcCellList`. Next, we just have to compute the product between the two sparse matrices and convert all values of the result into one of these: $\{-1; +1; 0\}$ using function `cscBinFilter`.

```
< get boundary chain 50b >  $\equiv$   
function larBoundaryChain(cscBoundaryMat, brcCellList)  
    """  
    Compute boundary chains  
    """  
  
    # Computing boundary chains  
    n = size(cscBoundaryMat)[1]
```

```

m = size(cscBoundaryMat)[2]

debug("Boundary matrix size: ", n, "\t", m)

data = ones(Int64, length(brcCellList))

i = Array{Int64, length(brcCellList)}
for k in 1:length(brcCellList)
    i[k] = brcCellList[k] + 1
end

j = ones{Int64, length(brcCellList)}

debug("cscChain rows length: ", length(i))
debug("cscChain columns length: ", length(j))
debug("cscChain data length: ", length(brcCellList))

debug("rows ", i)
debug("columns ", j)
debug("data ", data)

cscChain = sparse(i, j, data, m, 1)
cscmat = cscBoundaryMat * cscChain
out = cscBinFilter(cscmat)
return out
end

function cscBinFilter(CSCm)
    k = 1
    data = nonzeros(CSCm)
    sgArray = copysign(1, data)

    while k <= nnz(CSCm)
        if data[k] % 2 == 1 || data[k] % 2 == -1
            data[k] = 1 * sgArray[k]
        else
            data[k] = 0
        end
        k += 1
    end

    return CSCm
end
◇

```

Fragment referenced in 88a.

7.3 Get oriented cells from a chain

Another operation that could be useful (even if it is not actually used in the package) consists in getting of “+1” oriented cells from a chain. For obtaining this result, it is necessary to get all non-zeros element from the sparse Julia array (remembering that if the user manually write a zero into the array it will be returned from `nonzeros` function anyway) and then returning only indices of cells that have a “+1” in nonzero element array.

```
< get oriented cells from a chain 52a > ≡
function cscChainToCellList(CSCm)
    """
    Get a csc containing a chain and returns
    the cell list of the "+1" oriented faces
    """
    data = nonzeros(CSCm)
    # Now I need to remove zero element (problem with Julia nonzeros)
    nonzeroData = Array{Int64, 0}
    for n in data
        if n != 0
            push!(nonzeroData, n)
        end
    end

    cellList = Array{Int64,0}
    for (k, theRow) in enumerate(findn(CSCm)[1])
        if nonzeroData[k] == 1
            push!(cellList, theRow)
        end
    end
    return cellList
end ◊
```

Fragment referenced in 88a.

7.4 Transform relationships from arrays of arrays to a sparse matrix

Another function which can be useful for our purposes is conversion between different representations of the LAR relationships. For example we often use a representation based on list of list of int but if we want to apply topological operators (such as the incident operators) we need to convert it into a matrix of values. In Figure 14 we can see an example of a LAR relationship with different representations.

```
< transform relationships to csc 52b > ≡
```

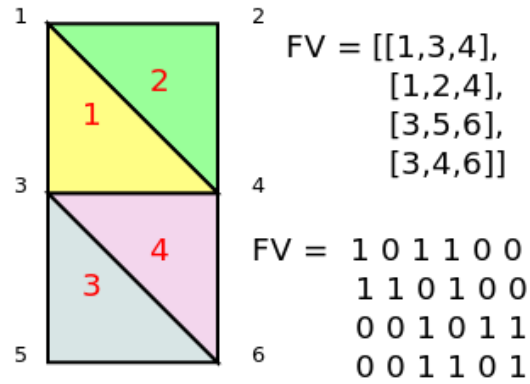


Figure 14: Different representations for faces of a simple LAR model. the first one is based on a list of list of int, while the other is a simple matrix where for every value we have $FV[i][j] = 1 \iff \text{face } i \text{ contains the vertex } j$

```
function relationshipListToCSC(larRelation)
  """
  Get a LAR relationship
  and convert it into a CSC matrix
  """

  # Build I and J arrays for creation of
  # sparse matrix
  data = Array{Int, 0}
  I = Array{Int, 0}
  J = Array{Int, 0}
  for (k,row) in enumerate(larRelation)
    for col in row
      push!(I, k)
      push!(J, col)
      push!(data, 1)
    end
  end

  return sparse(I, J, data)
end ◇
```

Fragment referenced in 88a.

8 LARUtils

This module contains functions used for manipulation of LAR models

8.1 Module imports

These are modules used in LARUtils and the functions exported

```
< modules import LARUtils 54a > ≡  
    using Logging  
  
    export ind, invertIndex, getBases, removeDoubleVerticesAndFaces,  
           computeModelAndBoundaries  
    ◇
```

Fragment referenced in 88b.

8.2 Transformation from matrix to array

First utility functions we will see, transform a matrix into an array and vice versa. We have already seen in section 5.4.1 uses of this linearized matrices; now we can focus on code for transformation.

```
< conversion from matrix to array 54b > ≡  
    function ind(x, y, z, nx, ny)  
        """  
        Transform coordinates into linearized matrix indexes  
        """  
        return x + (nx + 1) * (y + (ny + 1) * (z))  
    end ◇
```

Fragment referenced in 88b.

Here we have defined also the inverse transformation from the array to the matrix, which is useful for obtaining vertices coordinates from a cell

```
< conversion from array to matrix 54c > ≡
```

```

function invertIndex(nx,ny,nz)
    """
    Invert indexes
    """
    nx, ny, nz = nx + 1, ny + 1, nz + 1
    function invertIndex0(offset)
        a0, b0 = trunc(offset / nx), offset % nx
        a1, b1 = trunc(a0 / ny), a0 % ny
        a2, b2 = trunc(a1 / nz), a1 % nz
        return b0, b1, b2
    end
    return invertIndex0
end ◇

```

Fragment referenced in 88b.

8.3 Get bases of a LAR model

For generation of LAR models from an array of non-empty cells, we need to define a function for obtaining a base for every model, which will contain all LAR relationships:

- **V**: the array of vertices of a LAR model
- **VV**: the relationship between a vertex and itself
- **EV**: the relationship between an edge and its vertices
- **FV**: the relationship between a face and its vertices
- **CV**: the relationship between a cell and its vertices

From a geometrical point of view these bases create a chain composed from $nx \times ny \times nz$ square cells (where nx ny and nz are the grid size).

Now we will see in details how to obtain all LAR relationships.

First of all we need to compute vertices for the geometry:

```

⟨ compute vertices 55 ⟩ ≡
    # Calculating vertex coordinates (nx * ny * nz)
    V = Array{Int64}[]
    for z in 0:nz
        for y in 0:ny
            for x in 0:nx
                push!(V, [x,y,z])
            end
        end
    end
end ◇

```

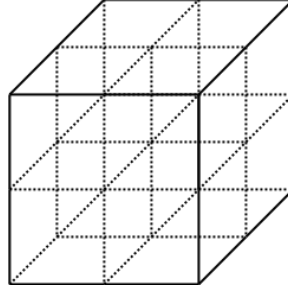



Figure 15: LAR bases geometry for a $2 \times 2 \times 2$ grid

Fragment referenced in 58.

So we assume that our cube geometry has only integers coordinates that can vary from $(0,0,0)$ to (nx,ny,nz)

Next we have to compute the CV relationship:

```

< compute CV 56a > ≡
  # Building CV relationship
  CV = Array{Int64}[]
  for z in 0:nz-1
    for y in 0:ny-1
      for x in 0:nx-1
        push!(CV,the3Dcell([x,y,z]))
      end
    end
  end
end ◇

```

Fragment referenced in 58.

For every coordinate in the space delimited by the grid size, it is called function `the3Dcell`, which get the coordinate values returning a cell in the three-dimensional space:

```

< compute three-dimensional cells 56b > ≡
  function the3Dcell(coords)
    x,y,z = coords
    return [ind(x,y,z,nx,ny), ind(x+1,y,z,nx,ny), ind(x,y+1,z,nx,ny),
            ind(x,y,z+1,nx,ny), ind(x+1,y+1,z,nx,ny), ind(x+1,y,z+1,nx,ny),
            ind(x,y+1,z+1,nx,ny), ind(x+1,y+1,z+1,nx,ny)]
  end
  ◇

```

Fragment referenced in 58.

Now we have to compute the FV relationship, which will be widely used in this package:

```
 $\langle \text{compute FV 57a} \rangle \equiv$   
  # Building FV relationship  
  FV = Array{Int64}[]  
  v2coords = invertIndex(nx,ny,nz)  
  
  for h in 0:(length(V)-1)  
    x,y,z = v2coords(h)  
  
    if (x < nx) && (y < ny)  
      push!(FV, [h,ind(x+1,y,z,nx,ny),ind(x,y+1,z,nx,ny),ind(x+1,y+1,z,nx,ny)])  
    end  
  
    if (x < nx) && (z < nz)  
      push!(FV, [h,ind(x+1,y,z,nx,ny),ind(x,y,z+1,nx,ny),ind(x+1,y,z+1,nx,ny)])  
    end  
  
    if (y < ny) && (z < nz)  
      push!(FV, [h,ind(x,y+1,z,nx,ny),ind(x,y,z+1,nx,ny),ind(x,y+1,z+1,nx,ny)])  
    end  
  
  end  $\diamond$ 
```

Fragment referenced in 58.

Finally we have the VV relationship (which is trivial)

```
 $\langle \text{compute VV 57b} \rangle \equiv$   
  # Building VV relationship  
  VV = map((x)->[x], 0:length(V)-1)  $\diamond$ 
```

Fragment referenced in 58.

and the EV relationship

```
 $\langle \text{compute EV 57c} \rangle \equiv$ 
```

```

# Building EV relationship
EV = Array{Int64}[]
for h in 0:length(V)-1
    x,y,z = v2coords(h)
    if (x < nx)
        push!(EV, [h,ind(x+1,y,z,nx,ny)])
    end
    if (y < ny)
        push!(EV, [h,ind(x,y+1,z,nx,ny)])
    end
    if (z < nz)
        push!(EV, [h,ind(x,y,z+1,nx,ny)])
    end
end ◇

```

Fragment referenced in 58.

This is the complete code for the function `getBases`

```

⟨ get LAR bases 58 ⟩ ≡
function getBases(nx, ny, nz)
    """
    Compute all LAR relations
    """

    ⟨ compute three-dimensional cells 56b ⟩

    ⟨ compute vertices 55 ⟩

    ⟨ compute CV 56a ⟩

    ⟨ compute FV 57a ⟩

    ⟨ compute VV 57b ⟩

    ⟨ compute EV 57c ⟩

    # return all basis
    return V, (VV, EV, FV, CV)
end ◇

```

Fragment referenced in 88b.

8.4 Double vertices and faces removal

Another useful function for our models is *removal of double vertices and faces*. In fact, when we produce a LAR model getting only full cell from the geometry in Figure 15 we could obtain double vertices (and consequently double faces). Figure 16 shows an example of a model with these vertices:

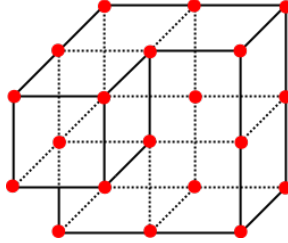


Figure 16: A sample model taken from a $2 \times 2 \times 2$ grid with double vertices between faces in red (remember that we have only the boundaries faces for the model as we have seen in section 5.4.1)

As we can see, for every model there are a lot of double vertices, so we need to remove them for obtaining a compact representation and for next smoothing of the objects. First of all we have to identify double vertices, so it can be useful to define an order between them. Unfortunately Julia does not define a function for order array containing coordinates (which is format used in V array); so we have to define first a custom ordering function:

```

⟨ vertices comparator function 59 ⟩ ≡
    function lessThanVertices(v1, v2)
        """
        Utility function for comparing vertices coordinates
        """

        if v1[1] == v2[1]
            if v1[2] == v2[2]
                return v1[3] < v2[3]
            end
            return v1[2] < v2[2]
        end
        return v1[1] < v2[1]
    end ◇

```

Fragment referenced in 61b.

Now we can remove double vertices from the V array simply ordering them and removing all consecutive equal vertices. This procedure is more complex than a simple call to Julia `unique` function for removal of double elements because we need the new vertices indices for renaming faces (as we can see later)

```

⟨removal of double vertices 60⟩ ≡
function removeDoubleVertices(V)
    """
    Remove double vertices from a LAR model

    V: Array containing all vertices of the model
    """

    # Sort the vertices list and returns the ordered indices
    orderedIndices = sortperm(V, lt = lessThanVertices, alg=MergeSort)

    orderedVerticesAndIndices = collect(zip(sort(V, lt = lessThanVertices),
                                             orderedIndices))

    newVertices = Array{Array{Float64}, 0}()
    indices = zeros{Int, length(V)}
    prevv = Void
    i = 1
    for (v, ind) in orderedVerticesAndIndices
        if v == prevv
            indices[ind] = i - 1
        else
            push!(newVertices, v)
            indices[ind] = i
            i += 1
            prevv = v
        end
    end
    return newVertices, indices
end ◇

```

Fragment referenced in 61b.

As we can see the algorithm does the following steps:

1. Sort of vertices list
2. Set the current vertex index counter to 1
3. For every couple (*vertex*, *index* into V array) do:

- (a) If the current *vertex* is equal to the previous one put into the indices array at position *index* the value for the current vertex index count
- (b) If the current *vertex* is not equal to the previous one save it into a new V array, insert the indices array at position *index* the current index count and increment it by one

So at the end of this function the array *newVertices* will contain all unique vertices, while the *indices* array will contain the correct index for every vertex into *newVertices* and the index corresponding to the saved vertex for every deleted vertex.

Now we can use these informations for renaming all faces.

$\langle \text{renaming of faces 61a} \rangle \equiv$

```
function reindexVerticesInFaces(FV, indices, offset)
    """
    Reindex vertices indices in faces array

    FV: Faces array of the LAR model
    indices: new Indices for faces
    offset: offset for faces indices
    """

    for f in FV
        for i in 1: length(f)
            f[i] = indices[f[i] - offset] + offset
        end
    end
    return FV
end ◇
```

Fragment referenced in 61b.

Here we can observe a *offset* parameter, which is necessary only if we are renaming faces whose indices doesn't start from zero; actually in `ImagesToLARModel` it is always equal to zero.

Finally for removing double faces, we only have to call `unique` function on renamed faces. This is the final code

$\langle \text{removal of double vertices and faces 61b} \rangle \equiv$

⟨ vertices comparator function 59 ⟩

```
function removeDoubleVerticesAndFaces(V, FV, facesOffset)
    """
    Removes double vertices and faces from a LAR model

    V: Array containing all vertices
    FV: Array containing all faces
    facesOffset: offset for faces indices
    """

    newV, indices = removeDoubleVertices(V)
    reindexedFaces = reindexVerticesInFaces(FV, indices, facesOffset)
    newFV = unique(FV)

    return newV, newFV

end
```

⟨ removal of double vertices 60 ⟩

⟨ renaming of faces 61a ⟩ ◇

Fragment referenced in 88b.

8.5 Creation of a LAR model

Now we can see code used for creation of a LAR model given the sparse array containing full cells of our block (**objectBoundaryChain** as we had seen in Section 7.2). We also need the following parameters:

- **imageDx, imageDy, imageDz**: The grid size
- **xStart, yStart, zStart**: The coordinate offsets for the current block vertices
- **facesOffset**: The offset for faces of this block

First thing to do is define models that will be returned from the function:

```
⟨ models definition 62 ⟩ ≡
    V_model = Array(Array{Int}, 0)
    FV_model = Array(Array{Int}, 0)

    V_left = Array(Array{Int}, 0)
```

```

FV_left = Array(Array{Int},0)

V_right = Array(Array{Int},0)
FV_right = Array(Array{Int},0)

V_top = Array(Array{Int},0)
FV_top = Array(Array{Int},0)

V_bottom = Array(Array{Int},0)
FV_bottom = Array(Array{Int},0)

V_front = Array(Array{Int},0)
FV_front = Array(Array{Int},0)

V_back = Array(Array{Int},0)
FV_back = Array(Array{Int},0) ◇

```

Fragment referenced in 67.

We can see from Figure 17 that our grid is divided into seven parts.

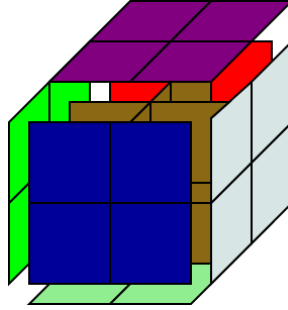


Figure 17: Decomposition of a LAR model into seven parts: the inside model (brown), the left boundary (green), the right boundary (light blue), the top boundary (purple), the bottom boundary (light green), the front boundary (blue), the back boundary (red)

We need this decomposition because we are interested in boundaries of the entire model, while we currently have boundaries only for blocks. So we need to split the inner parts of a single block model, as we need to freely merge boundaries between adjacent blocks removing the common faces. Function for boundaries merging are shown in subsection 8.6.

After model definition we have to get the cells indices from the block boundary chain and for every non-empty cell we have found, choose the correct model for it. We can

observe that every boundary face has a fixed coordinate; for example all faces on the top boundary have the maximum z-coordinate, or faces on right boundary have the maximum y-coordinate (as shown in Figure 18)

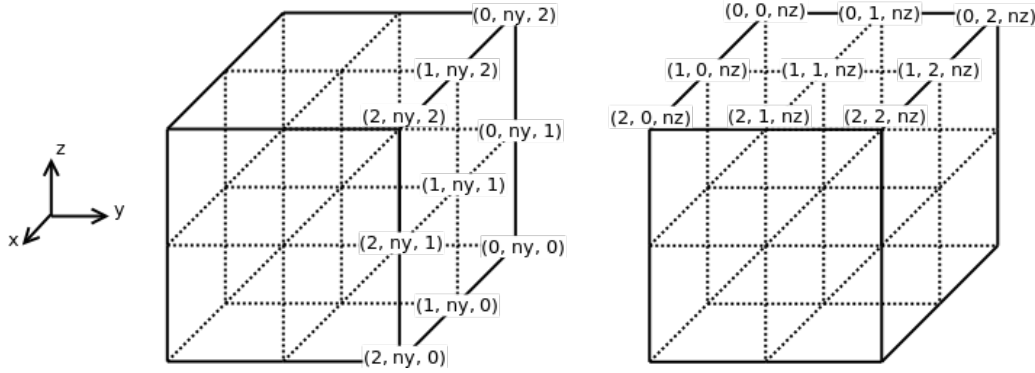


Figure 18: Boundaries coordinates for top and right boundaries of a $2 \times 2 \times 2$ grid. We can observe that every boundary has a fixed coordinate

So we can define a series of functions for checking the membership of a given face to a boundary exploiting these fixed coordinates:

```

⟨ check membership of a face to a boundary 64 ⟩ ≡
function isOnLeft(face, V, nx, ny, nz)
    """
    Check if face is on left boundary
    """

    for(vtx in face)
        if(V[vtx + 1][2] != 0)
            return false
        end
    end
    return true

end

function isOnRight(face, V, nx, ny, nz)
    """
    Check if face is on right boundary
    """

    for(vtx in face)
        if(V[vtx + 1][2] != ny)
            return false
        end
    end
    return true
end

```

```

        end
    end
    return true

end

function isOnTop(face, V, nx, ny, nz)
    """
    Check if face is on top boundary
    """

    for(vtx in face)
        if(V[vtx + 1][3] != nz)
            return false
        end
    end
    return true
end

function isOnBottom(face, V, nx, ny, nz)
    """
    Check if face is on bottom boundary
    """

    for(vtx in face)
        if(V[vtx + 1][3] != 0)
            return false
        end
    end
    return true
end

function isOnFront(face, V, nx, ny, nz)
    """
    Check if face is on front boundary
    """

    for(vtx in face)
        if(V[vtx + 1][1] != nx)
            return false
        end
    end
    return true
end

function isOnBack(face, V, nx, ny, nz)

```

```

"""
Check if face is on back boundary
"""

for(vtx in face)
  if(V[vtx + 1][1] != 0)
    return false
  end
end
return true
end ◇

```

Fragment referenced in 67.

After choosing of the right model, we have to insert our face into it. We can do it with the following function, which takes vertices and faces of the base and the model, the face, and the offset of the current face for the model chosen:

```

⟨ add a face to a model 66 ⟩ ≡
function addFaceToModel(V_base, FV_base, V, FV, face, vertex_count)
"""
  Insert a face into a LAR model

  V_base, FV_base: LAR model of the base
  V, FV: LAR model
  face: Face that will be added to the model
  vertex_count: Indices for faces vertices
"""
  new_vertex_count = vertex_count
  for vtx in FV_base[face]
    push!(V, [convert(Int, V_base[vtx + 1][1] + xStart),
              convert(Int, V_base[vtx + 1][2] + yStart),
              convert(Int, V_base[vtx + 1][3] + zStart)])
    new_vertex_count += 1
  end
  push!(FV, [vertex_count, vertex_count + 1, vertex_count + 3])
  push!(FV, [vertex_count, vertex_count + 3, vertex_count + 2])

  return new_vertex_count
end ◇

```

Fragment referenced in 67.

As we can see, for every face we put into the model FV array two faces, in fact our final representation is not based on square faces but on triangular faces.

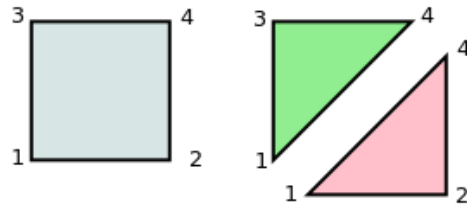


Figure 19: Triangulation of a single face

This is the complete code for creation of a model

```

< LAR model creation 67 > ≡
  < check membership of a face to a boundary 64 >

function computeModelAndBoundaries(imageDx, imageDy, imageDz,
                                   xStart, yStart, zStart,
                                   objectBoundaryChain)
    """
    Takes the boundary chain of a part of the entire model
    and returns a LAR model splitting the boundaries

    imageDx, imageDy, imageDz: Boundary dimensions
    xStart, yStart, zStart: Offset of this part of the model
    objectBoundaryChain: Sparse csc matrix containing the cells
    """

    < add a face to a model 66 >

    < models definition 62 >

    V, bases = getBases(imageDx, imageDy, imageDz)
    FV = bases[3]

    vertex_count_model = 1
    vertex_count_left = 1
    vertex_count_right = 1
    vertex_count_top = 1
    vertex_count_bottom = 1
    vertex_count_front = 1
    vertex_count_back = 1

    # Get all cells (independently from orientation)

```

```

b2cells = findn(objectBoundaryChain)[1]

debug("b2cells = ", b2cells)

for f in b2cells
    old_vertex_count_model = vertex_count_model
    old_vertex_count_left = vertex_count_left
    old_vertex_count_right = vertex_count_right
    old_vertex_count_top = vertex_count_top
    old_vertex_count_bottom = vertex_count_bottom
    old_vertex_count_front = vertex_count_front
    old_vertex_count_back = vertex_count_back

    # Choosing the right model for vertex
    if(isOnLeft(FV[f], V, imageDx, imageDy, imageDz))
        vertex_count_left = addFaceToModel(V, FV, V_left, FV_left,
                                            f, old_vertex_count_left)
    elseif(isOnRight(FV[f], V, imageDx, imageDy, imageDz))
        vertex_count_right = addFaceToModel(V, FV, V_right, FV_right,
                                            f, old_vertex_count_right)
    elseif(isOnTop(FV[f], V, imageDx, imageDy, imageDz))
        vertex_count_top = addFaceToModel(V, FV, V_top, FV_top,
                                            f, old_vertex_count_top)
    elseif(isOnBottom(FV[f], V, imageDx, imageDy, imageDz))
        vertex_count_bottom = addFaceToModel(V, FV, V_bottom, FV_bottom,
                                            f, old_vertex_count_bottom)
    elseif(isOnFront(FV[f], V, imageDx, imageDy, imageDz))
        vertex_count_front = addFaceToModel(V, FV, V_front, FV_front,
                                            f, old_vertex_count_front)
    elseif(isOnBack(FV[f], V, imageDx, imageDy, imageDz))
        vertex_count_back = addFaceToModel(V, FV, V_back, FV_back,
                                            f, old_vertex_count_back)
    else
        vertex_count_model = addFaceToModel(V, FV, V_model, FV_model,
                                            f, old_vertex_count_model)
    end
end

# Removing double vertices
return [removeDoubleVerticesAndFaces(V_model, FV_model, 0)],
[removeDoubleVerticesAndFaces(V_left, FV_left, 0)],
[removeDoubleVerticesAndFaces(V_right, FV_right, 0)],
[removeDoubleVerticesAndFaces(V_top, FV_top, 0)],
[removeDoubleVerticesAndFaces(V_bottom, FV_bottom, 0)],
[removeDoubleVerticesAndFaces(V_front, FV_front, 0)],

```

```

    [removeDoubleVerticesAndFaces(V_back, FV_back, 0)]
end ◇

```

Fragment referenced in 88b.

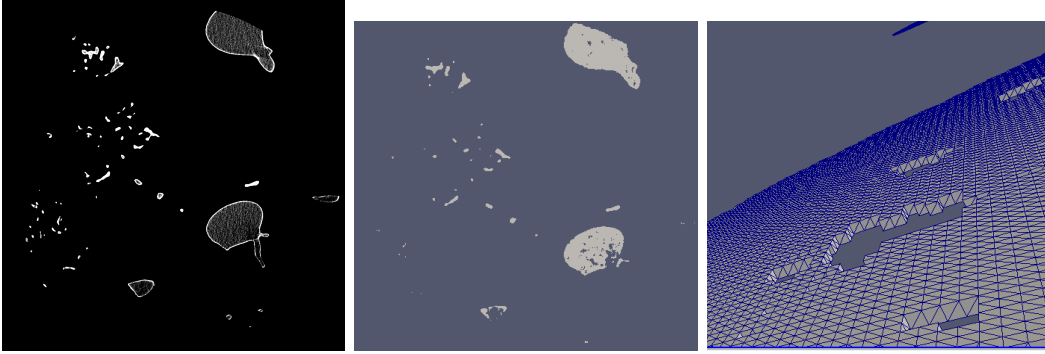


Figure 20: Creation of a sample model. (a) The original image (b) The three-dimensional model (c) The three-dimensional model (detail with triangular faces)

8.6 Removing double faces and vertices from boundaries

In previous section, we have seen how to create a LAR model from the chain list. However this model contains all borders between blocks, while we are only interested in borders for the entire image. So, we will see functions for boundaries merging with removal of double faces and vertices from both sides.

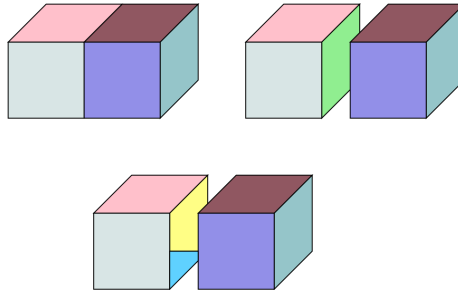


Figure 21: Removal of double faces from boundaries. (a) Two adjacent blocks (b) The same blocks exploded on x axis (c) Result of the removal on the exploded blocks

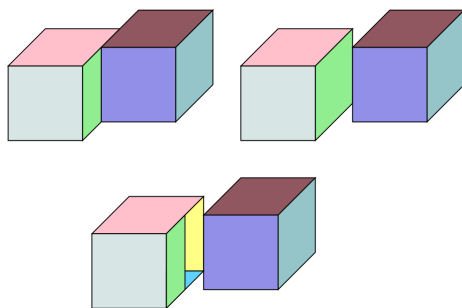


Figure 22: Same as Figure 21 with another model

The algorithm for the removal is very simply. First of all we need to remove double vertices from models in the usual way using `removeDoubleVertices` function and re-indexing all faces. Next we count all elements in re-indexed faces array removing elements with more than one occurrence and create an array of faces with an explicit representation of vertices (*FV_vertices*). Now we can safely remove double vertices on the other side of the boundary without losing the correct indexing in the faces. Finally we can create the final faces array with only remaining vertices comparing coordinates in *FV_vertices* with the ones in the last vertices array.

Code for this function is the following:

```

< Removal of double vertices and faces from boundaries 70 > ≡
function removeVerticesAndFacesFromBoundaries(V, FV)
    """
    Remove vertices and faces duplicates on
    boundaries models

    V,FV: lar model of two merged boundaries
    """

    newV, indices = removeDoubleVertices(V)
    uniqueIndices = unique(indices)

    # Removing double faces on both boundaries
    FV_reindexed = reindexVerticesInFaces(FV, indices, 0)
    FV_unique = unique(FV_reindexed)

    FV_cleaned = Array{Array{Int}, 0}
    for f in FV_unique
        if(count((x) -> x == f, FV_reindexed) == 1)
            push!(FV_cleaned, f)
        end
    end
end

```

```

# Creating an array of faces with explicit vertices
FV_vertices = Array(Array{Array{Float64}}, 0)

for i in 1 : length(FV_cleaned)
    push!(FV_vertices, Array(Array{Float64}, 0))
    for vtx in FV_cleaned[i]
        push!(FV_vertices[i], newV[vtx])
    end
end

V_final = Array(Array{Float64}, 0)
FV_final = Array(Array{Int}, 0)

# Saving only used vertices
for face in FV_vertices
    for vtx in face
        push!(V_final, vtx)
    end
end

V_final = unique(V_final)

# Renumbering FV
for face in FV_vertices
    tmp = Array{Int, 0}
    for vtx in face
        ind = findfirst(V_final, vtx)
        push!(tmp, ind)
    end
    push!(FV_final, tmp)
end

return V_final, FV_final
end ◇

```

Fragment referenced in 88b.

9 Smoother

This module contains functions used for smoothing LAR models

9.1 Get adjacent vertices

As we will see in next subsection, for executing a smoothing algorithm we need to know adjacent vertices to a given one. So we need a *VV relationship*, where for every vertex index i , we have a list of adjacent vertices.

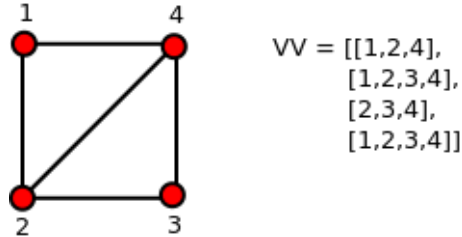


Figure 23: *VV relationship* for a simple model

Algorithm is very simple and exploit the following property: *for triangular faces all vertices are linked together*. So we just need to search for every vertex i all faces that contain it and add all their vertices to a list. *VV* will contain a concatenation of all these lists

```

⟨get adjacent vertices 72⟩ ≡
function adjVerts(V, FV)
    """
    Compute the adjacency graph of vertices
    of a LAR model

    V, FV: LAR model

    Returns the list of indices of vertices adjacent
    to a vertex
    """
    VV = Array{Int}[]
    for i in 1:length(V)
        row = Array{Int, 0}
        for face in FV
            if i in face
                for v in face
                    push!(row, v)
                end
            end
        end
        if length(row) == 0
            push!(row, i)
        end
    end
end

```

```

    push!(VV, collect(unique(row)))
end
return VV
end ◇

```

Fragment referenced in 90a.

9.2 Laplacian smoothing

There are many different algorithms for mesh smoothing. The simpler and the one we used in this library is **laplacian smoothing**. For each vertex in a mesh, a new position is chosen according to local information (such as the coordinates of neighbors) and the vertex is moved there. If that mesh is topologically a rectangular grid (so each internal vertex is connected to four neighbors) then this operation produces the *Laplacian* of the mesh.

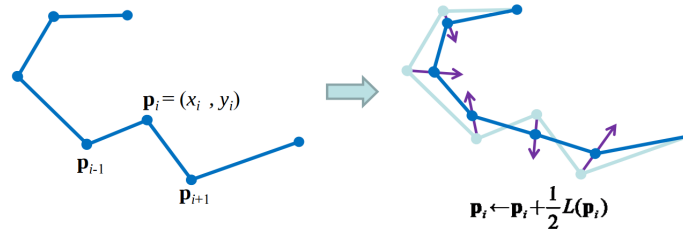


Figure 24: Laplacian smoothing (picture taken from the *Geometry Processing Algorithms* course at Stanford University)

As we can see from Figure 24, with substitution of every vertex position with the mean of the neighbors positions, we can obtain a curve with smoothed edges. This procedure can be repeated many times, so we can obtain a smoother model. For example, in Figure 26, we can see this algorithm applied on a sample mesh with three iterations.

This is the code for the smoothing function; it takes a single LAR model and returns the smoothed model.

```

⟨laplacian smoothing 73⟩ ≡
function smoothModel(V, FV)
    """
    Execute a Laplacian smoothing on a LAR model returning
    the new smoothed model

    V, FV: LAR model
    """

```

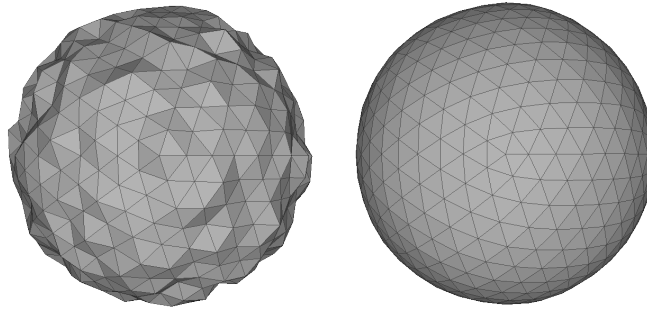


Figure 25: Laplacian smoothing for a sample mesh. (a) Original mesh (b) Mesh after three iterations of the smoothing algorithm (picture taken from a *Digital Geometry Processing* course at IMPA)

```

VV = adjVerts(V, FV)
newV = Array(Array{Float64},0)
V_temp = Array(Array{Float64},0)

for i in 1:length(VV)
    ads = VV[i]
    # Get all coordinates for adjacent vertices
    coords = Array(Array{Float64}, 0)
    for v in ads
        push!(coords, V[v])
    end

    # Computing sum of all vectors
    sum = [0.0, 0.0, 0.0]
    for v in coords
        sum += v
    end

    # Computing convex combination of vertices
    push!(newV, sum/length(ads))

end

return newV, FV
end ◇

```

Fragment referenced in 90a.

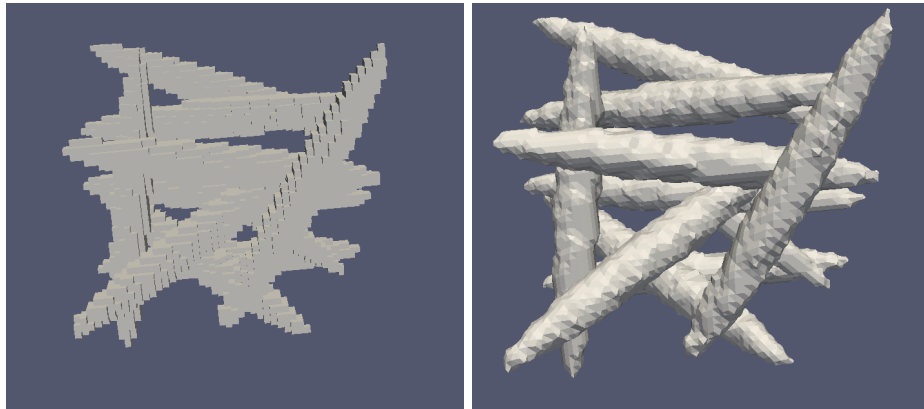


Figure 26: Smoothing of a sample model made with ImagesToLARModel

10 Model2Obj

This module contains functions used for reading/writing LAR models into obj files on disk

10.1 Writing models to file

Up to now we have seen how to manipulate LAR models obtaining a three-dimensional representation for our stack of images. However we have not seen how to visualize them using external software yet. We have chosen the *wavefront obj file format* for external visualization, which is very simple and common. The syntax used is the following:

- All vertices are described with their coordinates and written on a single row according to the following syntax: `v xCoord yCoord zCoord`
- All faces are described with their vertex index (calculated from their row) according to the following syntax: `f vertex1 vertex2 ...vertexn`

In Figure 27 there is an example of an obj file

We can see that this kind of representation is very similar to the LAR representation schema, so we just have to read every element of V and FV arrays and write them on disk. Because of we will use these informations a lot, it is more convenient to save the vertices and the faces on different files, so they can be loaded with less efforts. This is code used by the library:

```
<write obj files 75> ≡
function writeToObj(V, FV, outputFilename)
    """
    Take a LAR model and write it on obj file
```

```

v 0 0 0
v 1 0 0
v 0 1 0
v 1 1 0
v 0 0 1
v 1 0 1
v 1 1 1
f 1 2 4 3
f 5 6 8 7
f 1 2 6 5

```

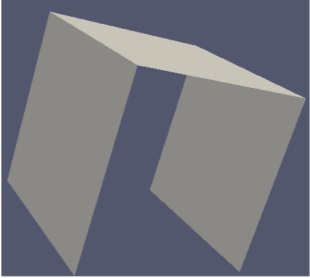


Figure 27: Obj sample file

```

V: array containing vertices coordinates
FV: array containing faces
outputFilename: prefix for the output files
"""

if (length(V) != 0)
    outputVtx = string(outputFilename, "_vtx.stl")
    outputFaces = string(outputFilename, "_faces.stl")

    fileVertex = open(outputVtx, "w")
    fileFaces = open(outputFaces, "w")

    for v in V
        write(fileVertex, "v ")
        write(fileVertex, string(v[1], " "))
        write(fileVertex, string(v[2], " "))
        write(fileVertex, string(v[3], "\n"))
    end

    for f in FV

        write(fileFaces, "f ")
        write(fileFaces, string(f[1], " "))
        write(fileFaces, string(f[2], " "))
        write(fileFaces, string(f[3], "\n"))
    end

    close(fileVertex)
    close(fileFaces)

end

```

end ◇

Fragment referenced in 89a.

10.2 Merging block models

Now we have seen how to write on disk a LAR model using the *wavefront obj file format*, however, as we have already seen, we have a lot of models from every block; so we need a function for the creation of the final merged model. The code which executes this task is very simple; this is the function:

⟨ *serial file merge 77* ⟩ ≡

```
function mergeObj(modelDirectory)
    """
    Merge stl files in a single obj file

    modelDirectory: directory containing models
    """

    files = readdir(modelDirectory)
    vertices_files = files[find(s -> contains(s, string("_vtx.stl")), files)]
    faces_files = files[find(s -> contains(s, string("_faces.stl")), files)]
    obj_file = open(string(modelDirectory, "/", "model.obj"), "w") # Output file

    vertices_counts = Array{Int64, length(vertices_files)}
    number_of_vertices = 0
    for i in 1:length(vertices_files)
        vtx_file = vertices_files[i]
        f = open(string(modelDirectory, "/", vtx_file))

        # Writing vertices on the obj file
        for ln in eachline(f)
            splitted = split(ln)
            write(obj_file, "v ")
            write(obj_file, string(convert{Int, round(parse(splitted[2]) * 10)), " "))
            write(obj_file, string(convert{Int, round(parse(splitted[3]) * 10)), " "))
            write(obj_file, string(convert{Int, round(parse(splitted[4]) * 10)), "\n"))
            number_of_vertices += 1
        end
        # Saving number of vertices
        vertices_counts[i] = number_of_vertices
        close(f)
    end
end
```

```

for i in 1 : length(faces_files)
  faces_file = faces_files[i]
  f = open(string(modelDirectory, "/", faces_file))
  for ln in eachline(f)
    splitted = split(ln)
    write(obj_file, "f ")
    if i > 1
      write(obj_file, string(parse(splitted[2]) + vertices_counts[i - 1], " "))
      write(obj_file, string(parse(splitted[3]) + vertices_counts[i - 1], " "))
      write(obj_file, string(parse(splitted[4]) + vertices_counts[i - 1]))
    else
      write(obj_file, string(splitted[2], " "))
      write(obj_file, string(splitted[3], " "))
      write(obj_file, splitted[4])
    end
    write(obj_file, "\n")
  end
  close(f)
end
close(obj_file)

# Removing all tmp files
for vtx_file in vertices_files
  rm(string(modelDirectory, "/", vtx_file))
end

for fcs_file in faces_files
  rm(string(modelDirectory, "/", fcs_file))
end

end ◇

```

Fragment referenced in 89a.

As we can see, we take all files contained into the model folder (distinguishing between files containing vertices from those containing faces) and write all their lines into the final model file. However this is simply for vertices files, while it is a bit complicated for faces, because it is necessary to change their indexes according to the current vertices positions into the file. So we need to memorize the offset for every file counting the number of vertices added at every time we open a new file containing vertices. Moreover we can see that we convert vertices coordinates into integer values; this is useful because some softwares do not read float vertices coordinates, so we first make the model ten times bigger (so we still have the first decimal number) and then round it.

The creation of this final model is quite slow, so we can try to speedup the entire software parallelizing it. In the next part we will see how this can be done.

10.2.1 Parallel merging of obj files

Now we will see how to parallelize the final file merging. However this is useful only for certain conditions; in fact in traditional filesystems the disk can be accessed from only one process at the same time, so parallelizing this task is totally useless. However for parallel filesystems it is a different matter because we can have parallel writes on storage.

The first simple algorithm we can think for parallel file merging, takes two files for every process and merge them creating a unique file. Then, this process can be repeated until we have only one final file. In Figure 28 there is a simple schema for this algorithm.

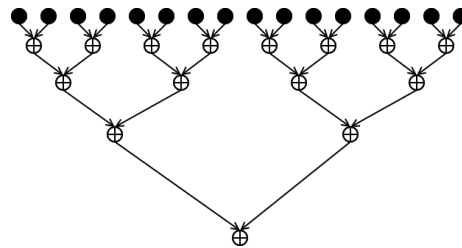


Figure 28: The first parallel algorithm for file merge. Black circles represent the original files, while circles with the cross, represent merged files

How we can see in that figure, if we have a process for every merge operation and sixteen files, we will have eight processes for the first merge, four processes for the second merge steps, two processes for the third step and one process for the final merge. Speaking in a general way, told n the number of files we want to merge, we will use $\lfloor n/2 \rfloor$ processes for every step. Probably if we would not have a balanced tree we could use the number of processes in a more efficient way for all our steps. In Figure 29 there is a non-balanced tree where the number of processes for every merge step is maximized.

Now we can examine the code used for obtaining this result. First of all we need a function for assigning files to the right process creating the tree in Figure 29:

```
< assign tasks 79 > ≡
function assignTasks(startInd, endInd, taskArray)
    """
    This function choose the first files to merge
    creating a tree where number of processes is maximized

    startInd: starting index for array subdivision
    endInd: end index for array subdivision
    taskArray: array containing indices of files to merge for first
```

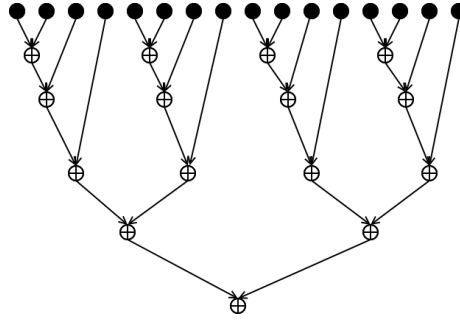



Figure 29: A better algorithm for file merge

```

"""
if (endInd - startInd == 2)
  push!(taskArray, startInd)
elseif (endInd - startInd < 2)
  if (endInd % 4 != 0 && startInd != endInd)
    # Stop recursion on this branch
    push!(taskArray, startInd)
  end
  # Stop recursion doing nothing
else
  assignTasks(startInd, startInd + trunc((endInd - startInd) / 2), taskArray)
  assignTasks(startInd + trunc((endInd - startInd) / 2) + 1, endInd, taskArray)
end
end ◇

```

Fragment referenced in 89a.

Now we need some functions for merging files; in particular we have to merge vertices files (contemporary counting the offsets for the faces) and faces files. This is the code for the first function:

```

⟨merge vertices file 80⟩ ≡
function mergeVerticesFiles(file1, file2, startOffset)
  """
  Support function for merging two vertices files.
  Returns the number of vertices of the merged file

  file1: path of the first file
  file2: path of the second file
  startOffset: starting face offset for second file

```

```

"""

f1 = open(file1, "a")

f2 = open(file2)
debug("Merging ", file2)
number_of_vertices = startOffset
for ln in eachline(f2)
    write(f1, ln)
    number_of_vertices += 1
end
close(f2)

close(f1)

return number_of_vertices
end ◇

```

Fragment referenced in 89a.

As we can see the algorithm is very simple; what we do is appending the content of the second file into the first one and returns the number of appended vertices, which will be used as offsets for faces. This is the code for concatenation of faces files:

```

⟨ merge faces file 81 ⟩ ≡
function mergeFacesFiles(file1, file2, facesOffset)
    """
    Support function for merging two faces files

    file1: path of the first file
    file2: path of the second file
    facesOffset: offset for faces
    """

    f1 = open(file1, "a")

    f2 = open(file2)
    for ln in eachline(f2)
        splitted = split(ln)
        write(f1, "f ")
        write(f1, string(parse(splitted[2]) + facesOffset, " "))
        write(f1, string(parse(splitted[3]) + facesOffset, " "))
        write(f1, string(parse(splitted[4]) + facesOffset, "\n"))
    end
end

```

```

close(f2)

close(f1)
end ◇

```

Fragment referenced in 89a.

These concatenation functions, are called by the `mergeObjProcesses`, which is executed by a single Julia process:

```

⟨parallel merge obj process function 82⟩ ≡
function mergeObjProcesses(fileArray, facesOffset = Nothing)
    """
    Merge files on a single process

    fileArray: Array containing files that will be merged
    facesOffset (optional): if merging faces files, this array contains
        offsets for every file

    if it is merging vertices files it returns the offset
    for the corresponding faces
    """

    if(contains(fileArray[1], string("_vtx.stl")))
        # Merging vertices files
        offsets = Array{Int, 0}
        push!(offsets, countlines(fileArray[1]))
        vertices_count = mergeVerticesFiles(fileArray[1], fileArray[2], countlines(fileArray[1]))
        rm(fileArray[2]) # Removing merged file
        push!(offsets, vertices_count)
        for i in 3: length(fileArray)
            vertices_count = mergeVerticesFiles(fileArray[1], fileArray[i], vertices_count)
            rm(fileArray[i]) # Removing merged file
            push!(offsets, vertices_count)
        end
        return offsets
    else
        # Merging faces files
        mergeFacesFiles(fileArray[1], fileArray[2], facesOffset[1])
        rm(fileArray[2]) # Removing merged file
        for i in 3 : length(fileArray)
            mergeFacesFiles(fileArray[1], fileArray[i], facesOffset[i - 1])
            rm(fileArray[i]) # Removing merged file
        end
    end
end

```

```

    end
end ◇

```

Fragment referenced in 89a.

The function can be called for both faces and vertices files; for the last case, however, we need to specify the *facesOffset* parameter.

Now we can put together the above functions with the following code:

```

⟨ merge obj helper function 83 ⟩ ≡
function mergeObjHelper(vertices_files, faces_files)
    """
    Support function for mergeObj. It takes vertices and faces files
    and executes a single merging step

    vertices_files: Array containing vertices files
    faces_files: Array containing faces files
    """
    numberOfImages = length(vertices_files)
    taskArray = Array{Int, 0}()
    assignTasks(1, numberOfImages, taskArray)

    # Now taskArray contains first files to merge
    numberOfVertices = Array{Int, 0}()
    tasks = Array{RemoteRef, 0}()
    for i in 1 : length(taskArray) - 1
        task = @spawn mergeObjProcesses(vertices_files[taskArray[i] : (taskArray[i + 1] - 1)])
        push!(tasks, task)
    end

    # Merging last vertices files
    task = @spawn mergeObjProcesses(vertices_files[taskArray[length(taskArray)] : end])
    push!(tasks, task)

    for task in tasks
        append!(numberOfVertices, fetch(task))
    end

    debug("NumberOfVertices = ", numberOfVertices)

    # Merging faces files
    tasks = Array{RemoteRef, 0}()
    for i in 1 : length(taskArray) - 1

```

```

        task = @spawn mergeObjProcesses(faces_files[taskArray[i] : (taskArray[i + 1] - 1)],
                                         numberOfVertices[taskArray[i] : (taskArray[i + 1] - 1)])
        push!(tasks, task)
    end

    #Merging last faces files
    task = @spawn mergeObjProcesses(faces_files[taskArray[length(taskArray)] : end],
                                     numberOfVertices[taskArray[length(taskArray)] : end])

    push!(tasks, task)

    for task in tasks
        wait(task)
    end

    end ◇

```

Fragment referenced in 89a.

As we can see, this is the code for distribution of our work among all processes. We have chosen to spawn a new process following the tree in Figure 29, passing to the `mergeObjProcesses` function the files given to the task with the function `assignTasks`.

Finally we just have to define the main function for parallel merging

```

⟨ merge obj parallel 84 ⟩ ≡
function mergeObjParallel(modelDirectory)
    """
    Merge stl files in a single obj file using a parallel
    approach. Files will be recursively merged two by two
    generating a tree where number of processes for every
    step is maximized
    Actually use of this function is discouraged. In fact
    speedup is influenced by disk speed. It could work on
    particular systems with parallel accesses on disks

    modelDirectory: directory containing models
    """

    files = readdir(modelDirectory)

    # Appending directory path to every file
    files = map((s) -> string(modelDirectory, "/", s), files)

    # While we have more than one vtx file and one faces file

```

```

while(length(files) != 2)
  vertices_files = files[find(s -> contains(s,string("_vtx.stl")), files)]
  faces_files = files[find(s -> contains(s,string("_faces.stl")), files)]

  # Merging files
  mergeObjHelper(vertices_files, faces_files)

  files = readdir(modelDirectory)
  files = map((s) -> string(modelDirectory, "/", s), files)
end

mergeVerticesFiles(files[2], files[1], 0)
mv(files[2], string(modelDirectory, "/model.obj"))
rm(files[1])

end ◇

```

Fragment referenced in 89a.

10.3 Load models from files

Another important functionality for our library, consists in model loading from our faces and vertices files. This is useful when passing from a pipeline step to another one. For simplicity, we offer a unique function that loads an array of vertices files (with the corresponding array of faces files), merge the values into a unique model and returns it.

⟨ *Load models from file 85* ⟩ ≡

```

function getModelsFromFiles(arrayV, arrayFV)
  """
  Get a LAR models for two arrays of vertices
  and faces files

  arrayV: Array containing all vertices files
  arrayFV: Array containing all faces files
  """

  V = Array(Array{Float64}, 0)
  FV = Array(Array{Int}, 0)
  offset = 0

  for i in 1:length(arrayV)
    if isfile(arrayFV[i])
      f_FV = open(arrayFV[i])

```

```

    for ln in eachline(f_FV)
        splitted = split(ln)
        push!(FV, [parse(splitted[2]) + offset, parse(splitted[3]) + offset,
            parse(splitted[4]) + offset])
    end
    close(f_FV)

    f_V = open(arrayV[i])
    for ln in eachline(f_V)
        splitted = split(ln)
        push!(V, [parse(splitted[2]), parse(splitted[3]),
            parse(splitted[4])])
        offset += 1
    end
    close(f_V)
end
end
return V, FV
end ◇

```

Fragment referenced in 89a.

11 Exporting the library

ImagesToLARModel

```

"src/ImagesToLARModel.jl" 86≡
    module ImagesToLARModel

        < update load path 6 >

        < modules import ImagesToLARModel 7a >
        < load JSON configuration 9 >
        < load JSON configuration for data preparation 7b >
        < data preparation from JSON file 11a >
        < manual data preparation 11b >
        < Start conversion from JSON file 12 >
        < Start manual conversion 13 >
    end
    ◇

```

ImagesConversion

```
"src/ImagesConversion.jl" 87a≡  
    module ImagesConversion  
  
        < modules import ImagesConversion 27a >  
  
        < main function for ImagesConversion 27b >  
  
        < parallel block iteration 29 >  
  
        < start conversion of images 31b >  
  
        < image conversion process 35b >  
  
        < boundary merge process function 37 >  
  
        < merge boundaries utility function 38 >  
  
        < Block merge process function 39b >  
  
        < Smooth block process function 41b >  
  
        < execute smoothing function 43 >  
  
    end  
    ◇
```

GenerateBorderMatrix

```
"src/GenerateBorderMatrix.jl" 87b≡  
    module GenerateBorderMatrix  
  
        < Matrix object for JSON file 48a >  
  
        < modules import GenerateBorderMatrix 46 >  
  
        < compute border matrix 48b >  
  
        < write Border matrix 47b >  
  
        < get Border matrix 47a >
```



```

    < transform border matrix in csc format 49 >
end
◇

```

Lar2Julia

```

"src/Lar2Julia.jl" 88a≡
    module Lar2Julia

        < modules import Lar2Julia 50a >

        < get boundary chain 50b >

        < get oriented cells from a chain 52a >

        < transform relationships to csc 52b >
    end
◇

```

LARUtils

```

"src/LARUtils.jl" 88b≡
    module LARUtils

        < modules import LARUtils 54a >

        < conversion from matrix to array 54b >

        < conversion from array to matrix 54c >

        < get LAR bases 58 >

        < removal of double vertices and faces 61b >

        < Removal of double vertices and faces from boundaries 70 >

        < LAR model creation 67 >
    end
◇

```

Model2Obj

```
"src/Model2Obj.jl" 89a≡
module Model2Obj

    using Logging

    export writeToObj, mergeObj, mergeObjParallel

    ⟨ write obj files 75 ⟩

    ⟨ serial file merge 77 ⟩

    ⟨ assign tasks 79 ⟩

    ⟨ merge vertices file 80 ⟩

    ⟨ merge faces file 81 ⟩

    ⟨ parallel merge obj process function 82 ⟩

    ⟨ merge obj helper function 83 ⟩

    ⟨ merge obj parallel 84 ⟩

    ⟨ Load models from file 85 ⟩
end
◇
```

PngStack2Array3dJulia

```
"src/PngStack2Array3dJulia.jl" 89b≡
module PngStack2Array3dJulia

    ⟨ modules import PngStack2Array3dJulia 14a ⟩
    ⟨ image resizing 16 ⟩

    ⟨ Convert to png 15c ⟩
    ⟨ Get image data 20 ⟩
    ⟨ Centroid computation 21 ⟩
    ⟨ Pixel transformation 24 ⟩
end
◇
```

Smother

```
"src/Smother.jl" 90a≡  
    module Smother  
    export smoothModel  
  
    ⟨ get adjacent vertices 72 ⟩  
  
    ⟨ laplacian smoothing 73 ⟩  
    end  
    ◇
```

11.1 Installing the library

12 Conclusions

12.1 Results

12.2 Further improvements

References

- [CL13] CVD-Lab, *Linear Algebraic Representation*, Tech. Report 13-00, Roma Tre University, October 2013.
- [PDFJ15] Alberto Paoluzzi, Antonio DiCarlo, Francesco Furiani, and Miroslav Jirik, *CAD models from medical images using LAR*, Computer-Aided Design and Applications **13** (2015), To appear.
- [W3C] W3C, *Portable Network Graphics (PNG) Specification (Second Edition)*, Tech. report.

A Utility functions

B Tests

Generation of the border matrix


```

end

function executeAllTests()
    @time testComputeOriented3Border()
    @time testWriteBorder()
    println("Tests completed.")
end

executeAllTests()

◇

```

Conversion of a png stack to a 3D array

```

"test/pngStack2Array3dJulia.jl" 92≡
push!(LOAD_PATH, "../..")
import PngStack2Array3dJulia
using Base.Test

function testGetImageData()
    """
    Test function for getImageData
    """

    width, height = PngStack2Array3dJulia.getImageData("images/0.png")

    @test width == 50
    @test height == 50

end

function testCalculateClusterCentroids()
    """
    Test function for calculateClusterCentroids
    """

    path = "images/"
    image = 0
    centroids = PngStack2Array3dJulia.calculateClusterCentroids(path, image, 2)

    expected = [0, 253]
    centroids = vec(reshape(centroids, 1, 2))

```

```

    @test sort(centroids) == expected
end

function testPngstack2array3d()
    """
    Test function for pngstack2array3d
    """
    path = "images/"
    minSlice = 0
    maxSlice = 4
    centroids = PngStack2Array3dJulia.calculateClusterCentroids(path, 0, 2)
    image3d = PngStack2Array3dJulia.pngstack2array3d(path, minSlice, maxSlice, centroids)

    @test size(image3d)[1] == 5
    @test size(image3d[1])[1] == 50
    @test size(image3d[1])[2] == 200

end

function executeAllTests()
    @time testCalculateClusterCentroids()
    @time testPngstack2array3d()
    @time testGetImageData()
    println("Tests completed.")
end

executeAllTests()

◇

```

Test for LAR utilities

```

"test/LARUtils.jl" 93≡
push!(LOAD_PATH, "../..")
import LARUtils
using Base.Test

function testInd()
    """
    Test function for ind
    """

    nx = 2

```

```

ny = 2

@test LARUtils.ind(0, 0, 0, nx, ny) == 0
@test LARUtils.ind(1, 1, 1, nx, ny) == 13
@test LARUtils.ind(2, 5, 4, nx, ny) == 53
@test LARUtils.ind(1, 1, 1, nx, ny) == 13
@test LARUtils.ind(2, 7, 1, nx, ny) == 32
@test LARUtils.ind(1, 0, 3, nx, ny) == 28
end

function executeAllTests()
    @time testInd()
    println("Tests completed.")
end

executeAllTests()

◇

```