

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

Danilo Salvati

November 3, 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 with the Julia language ([[CL13](#)]), because of its simplicity, showing how it can be used for quickly process image data.

Contents

1	Introduction	3
1.1	Why Julia	3
2	Software structure	3
2.1	Julia packages	3
2.2	Architecture of ImagesToLARModel	4
3	ImagesToLARModel	6
3.1	Calling modules	6
3.2	Input loading	6
3.3	Starting conversion	8
4	PngStack2Array3dJulia	10
5	ImagesConversion	10
6	GenerateBorderMatrix	10
7	Lar2Julia	10
8	LARUtils	10
9	Model2Obj	10
10	Exporting the library	10
10.1	Installing the library	45
11	Conclusions	45
11.1	Results	45
11.2	Further improvements	45
A	Utility functions	45
B	Tests	45

1 Introduction

This work has the objective of transforming 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 software.

The biggest problem we will face is lack of memory. In fact, our aim is to get a very large stack of images (so with the maximum possible resolution) and convert them. However biomedical images used in research are very big so we cannot save them entirely in memory. The idea used here is the same of existing software based on LAR schema ([PDFJ15]) with the introduction of parallelism layer, transforming only little parts of images obtaining models to merge in final result file. A typical hardware environment for this software is a cluster of computers, where processes (which convert images portions) are distributed among different machines.

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 1

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 2 there is the `REQUIRE` file for `ImagesToLARModel.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:

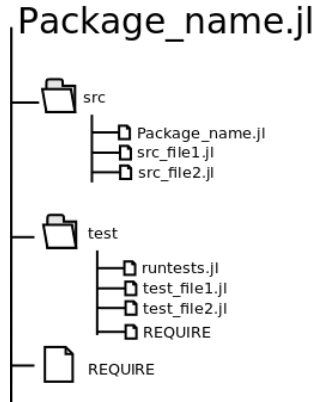


Figure 1: Julia module structure

julia 0.3
 JSON
 Logging
 PyCall
 Images
 Colors

Figure 2: 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

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

In figure 3 there is a simple schema of dependencies between modules.

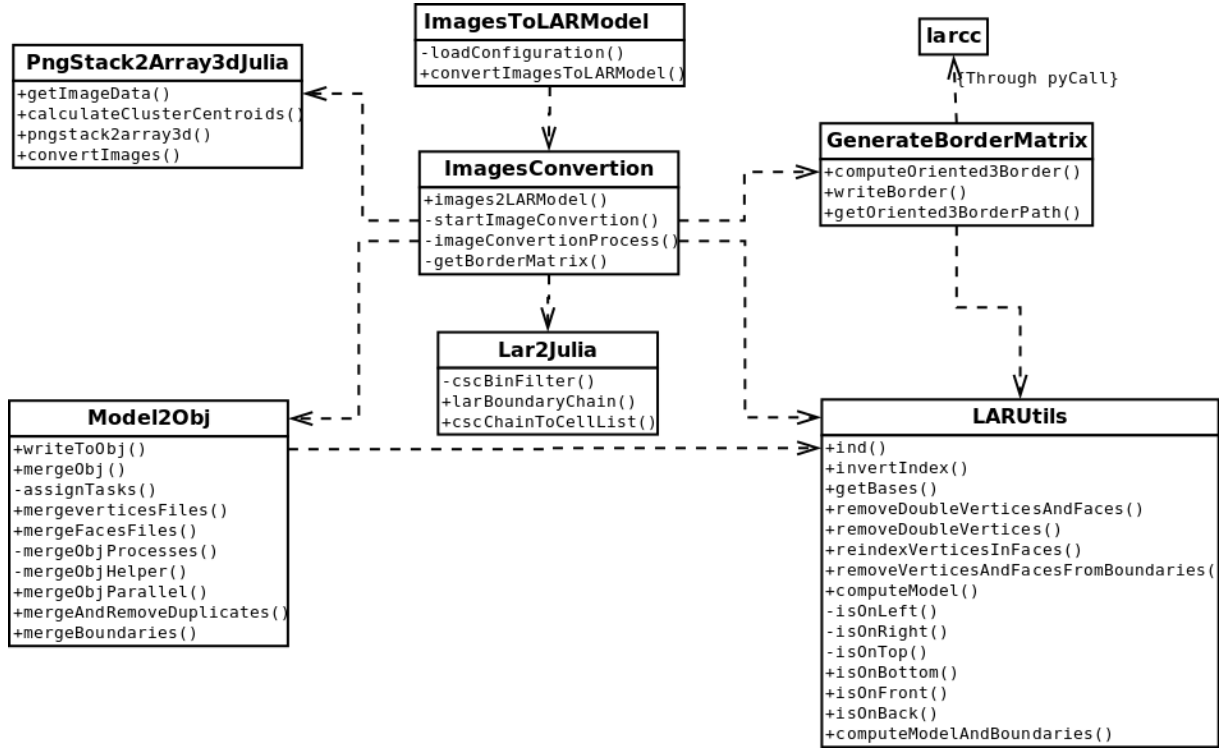


Figure 3: Schema of module dependencies of ImagesToLARModel

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 6a > ≡  
    push!(LOAD_PATH, Pkg.dir("ImagesToLARModel/src"))  
    ◇
```

Fragment referenced in 10a.

`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 6b > ≡  
    import JSON  
    import ImagesConversion  
  
    using Logging  
  
    export convertImagesToLARModel  
  
    ◇
```

Fragment referenced in 10a.

3.2 Input loading

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 9)

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 7⟩ ≡
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 10a.

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"
}
```

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`

3.3 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 8> ≡
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 10a.

```

⟨ Start manual conversion 9 ⟩ ≡
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
    """
    # 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 10a.

- 4 PngStack2Array3dJulia
- 5 ImagesConversion
- 6 GenerateBorderMatrix
- 7 Lar2Julia
- 8 LARUtils
- 9 Model2Obj
- 10 Exporting the library

ImagesToLARModel

```
"src/ImagesToLARModel.jl" 10a≡
    module ImagesToLARModel

        < update load path 6a >

        < modules import ImagesToLARModel 6b >
        < load JSON configuration 7 >
        < Start conversion from JSON file 8 >
        < Start manual conversion 9 >
    end
    ◇
```

ImagesConversion

```
"src/ImagesConversion.jl" 10b≡
    module ImagesConversion

        import GenerateBorderMatrix
        import PngStack2Array3dJulia
        import Lar2Julia
        import Model2Obj
        import LARUtils

        import JSON
```

```

using PyCall
@pyimport scipy.sparse as Pysparse

using Logging

export images2LARModel

"""
This is main module for converting a stack
of images into a 3d model
"""

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

    info("Moving images into temp directory")
    try
        mkdir(string(outputDirectory, "TEMP"))
    catch
    end

    tempDirectory = string(outputDirectory, "TEMP/")

    newBestImage = PngStack2Array3dJulia.convertImages(inputDirectory, tempDirectory, bestImage)

    imageWidth, imageHeight = PngStack2Array3dJulia.getImageData(string(tempDirectory, newBestImage))
    imageDepth = length(readdir(tempDirectory))

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

    # Starting images conversion and border computation
    info("Starting images conversion")

```

```

        startImageConversion(tempDirectory, newBestImage, outputDirectory, borderFilename,
                             imageHeight, imageWidth, imageDepth,
                             nx, ny, nz,
                             numberOfClusters, parallelMerge)

end

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
    """

    # Create clusters for image segmentation
    info("Computing image centroids")
    debug("Best image = ", bestImage)
    centroidsCalc = PngStack2Array3dJulia.calculateClusterCentroids(sliceDirectory, bestImage, n
    debug(string("centroids = ", centroidsCalc))

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

    beginImageStack = 0
    endImage = beginImageStack

    info("Converting images into a 3d model")
    tasks = Array{RemoteRef, 0}
    for zBlock in 0:(imageDepth / imageDz - 1)
        startImage = endImage
        endImage = startImage + imageDz
        info("StartImage = ", startImage)
        info("endImage = ", endImage)

        task = @spawn imageConversionProcess(sliceDirectory, outputDirectory,
                                             beginImageStack, startImage, endImage,
                                             imageDx, imageDy, imageDz,
                                             imageHeight, imageWidth,

```

```

                                centroidsCalc, boundaryMat)

    push!(tasks, task)
end

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

info("Merging boundaries")
# Merge Boundaries files
Model2Obj.mergeBoundaries(string(outputDirectory, "MODELS"),
                           imageHeight, imageWidth, imageDepth,
                           imageDx, imageDy, imageDz)

info("Merging obj models")
if parallelMerge
    Model2Obj.mergeObjParallel(string(outputDirectory, "MODELS"))
else
    Model2Obj.mergeObj(string(outputDirectory, "MODELS"))
end

end

function imageConversionProcess(sliceDirectory, outputDirectory,
                               beginImageStack, startImage, endImage,
                               imageDx, imageDy, imageDz,
                               imageHeight, imageWidth,
                               centroids, boundaryMat)
    """
    Support function for converting a stack of image on a single
    independent process
    """

    info("Transforming png data into 3d array")
    theImage = PngStack2Array3dJulia.pngstack2array3d(sliceDirectory, startImage, endImage, centroids, boundaryMat)

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

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

```

```

xStart = yBlock * imageDy
#xEnd = xStart + imageDx
#yEnd = yStart + imageDy
xEnd = xStart + imageDy
yEnd = yStart + imageDx
debug("*****")
debug(string("xStart = ", xStart, " xEnd = ", xEnd))
debug(string("yStart = ", yStart, " yEnd = ", yEnd))
debug("theImage dimensions: ", size(theImage)[1], " ", size(theImage[1])[1], " ", size(t

# Getting a slice of theImage array

image = Array{UInt8, (convert{Int, length(theImage)}, convert{Int, xEnd - xStart}, conver
debug("image size: ", size(image))
for z in 1:length(theImage)
    for x in 1 : (xEnd - xStart)
        for y in 1 : (yEnd - yStart)
            image[z, x, y] = theImage[z][x + xStart, y + yStart]
        end
    end
end

nz, nx, ny = size(image)
chains3D = Array{UInt8, 0}
zStart = startImage - beginImageStack
for y in 0:(ny - 1)
    for x in 0:(nx - 1)
        for z in 0:(nz - 1)
            if(image[z + 1, x + 1, y + 1] == foreground)
                push!(chains3D, y + ny * (x + nx * z))
            end
        end
    end
end

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
    # IMPORTANT: inverting xStart and yStart for obtaining correct rotation of the model

```

```

models = LARUtils.computeModelAndBoundaries(imageDx, imageDy, imageDz, yStart, xStart,

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

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

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

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

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

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

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

back_outputFilename = string(outputDirectory, "MODELS/back_output_", xBlock, "-", yBlock)
Model2Obj.writeToObj(V_back, FV_back, back_outputFilename)
else
    debug("Model is empty")
end
end
end
end

function getBorderMatrix(borderFilename)
    """
    TO REMOVE WHEN PORTING OF LARCC IN JULIA IS COMPLETED

    Get the border matrix from json file and convert it in
    CSC format
    """
    # Loading borderMatrix from json file

```

```

borderData = JSON.parsefile(borderFilename)
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

# Converting csr matrix to csc
csrBorderMatrix = Pysparse.csr_matrix((data,col,row), shape=(borderData["ROWCOUNT"],borderData["COLCOUNT"]), dtype=borderData["DATA"])
denseMatrix = pycall(csrBorderMatrix["toarray"],PyAny)

cscBoundaryMat = sparse(denseMatrix)

return cscBoundaryMat

end
end
◇

```

GenerateBorderMatrix

```

"src/GenerateBorderMatrix.jl" 16≡
module GenerateBorderMatrix

type MatrixObject
    ROWCOUNT
    COLCOUNT
    ROW
    COL
    DATA
end

```



```

export computeOriented3Border, writeBorder, getOriented3BorderPath

import LARUtils
using PyCall

import JSON

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

# 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

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

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

    rowcount = boundaryMatrix[:shape][1]
    colcount = boundaryMatrix[:shape][2]

    row = boundaryMatrix[:indptr]
    col = boundaryMatrix[:indices]
    data = boundaryMatrix[:data]

    # Writing informations on file
    outfile = open(outputFile, "w")

    matrixObj = MatrixObject(rowcount, colcount, row, col, data)
    JSON.print(outfile, matrixObj)
    close(outfile)
end

```

```

end

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
end
◇

```

Lar2Julia

```

"src/Lar2Julia.jl" 18≡
module Lar2Julia

export larBoundaryChain, cscChainToCellList

import JSON

using Logging

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

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
end
◇

```

LARUtils

```

"src/LARUtils.jl" 20≡
module LARUtils

using Logging

export ind, invertIndex, getBases, removeDoubleVerticesAndFaces, computeModel, computeModelAnd

function ind(x, y, z, nx, ny)
    """
        Transform coordinates into linearized matrix indexes
    """
    return x + (nx + 1) * (y + (ny + 1) * (z))
end

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

function getBases(nx, ny, nz)
    ""
    Compute all LAR relations
    ""

    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,z+1,nx,ny),ind(x,y+1,z+1,nx,ny),ind(x+1,y+1,z+1,nx,ny)]
    end

    # 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

    # 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

    # 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

# Building VV relationship
VV = map((x)->[x], 0:length(V)-1)

# 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

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

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
  end
end

```

```

        end
        return v1[2] < v2[2]
    end
    return v1[1] < v2[1]
end

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

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{Int}, 0}()
    indices = zeros{Int, length(V)}
    prevv = Nothing
    i = 1
    for (v, ind) in orderedVerticesAndIndices
        if v == prevv
            indices[ind] = i - 1
        else
            push!(newVertices, v)
            indices[ind] = i
            i += 1
        end
    end
end

```

```

        prevv = v
    end
end
return newVertices, indices
end

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

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

    # Creating an array of faces with explicit vertices

```



```

FV_vertices = Array(Array{Array{Int}}, 0)

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

V_final = Array(Array{Int}, 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

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

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

```

```

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

V_model = Array{Array{Int}, 0}
FV_model = Array{Array{Int}, 0}

vertex_count = 1

#b2cells = Lar2Julia.cscChainToCellList(objectBoundaryChain)
# Get all cells (independently from orientation)
b2cells = findn(objectBoundaryChain)[1]

debug("b2cells = ", b2cells)

for f in b2cells
    old_vertex_count = vertex_count
    for vtx in FV[f]
        push!(V_model, [convert{Int}(V[vtx + 1][1] + xStart),
                        convert{Int}(V[vtx + 1][2] + yStart),
                        convert{Int}(V[vtx + 1][3] + zStart)])
        vertex_count += 1
    end

    push!(FV_model, [old_vertex_count + facesOffset, old_vertex_count + 1 + facesOffset, old_v
    push!(FV_model, [old_vertex_count + facesOffset, old_vertex_count + 3 + facesOffset, old_v
end

# Removing double vertices
return removeDoubleVerticesAndFaces(V_model, FV_model, facesOffset)

end

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
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

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

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
    """

    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, T}(V_base[vtx + 1][1] + xStart),
                     convert{Int, T}(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

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

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}

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

#b2cells = Lar2Julia.cscChainToCellList(objectBoundaryChain)
# 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_bot)
    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
end
◇

```

Model2Obj

```

"src/Model2Obj.jl" 31≡
    module Model2Obj

        import LARUtils

        using Logging

        export writeToObj, mergeObj, mergeObjParallel

        function writeToObj(V, FV, outputFilename)
            """
            Take a LAR model and write it on obj 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
    end
end

```

```

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))
        debug("Opening ", vtx_file)

        # Writing vertices on the obj file
        for ln in eachline(f)
            write(obj_file, ln)
            number_of_vertices += 1
        end
        # Saving number of vertices
        vertices_counts[i] = number_of_vertices
        close(f)
    end

    for i in 1 : length(faces_files)
        faces_file = faces_files[i]
        f = open(string(modelDirectory, "/", faces_file))
        debug("Opening ", 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
        end
    end
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

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
    """
    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

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

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
    close(f2)

    close(f1)
end

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(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

function mergeObjHelper(vertices_files, faces_files)
    """
    Support function for mergeObj. It takes vertices and faces files
    and execute 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)
    #append!(numberOfVertices, mergeObjProcesses(vertices_files[taskArray[i] : (taskArray[i + 1] - 1)])
end

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

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)

    #mergeObjProcesses(faces_files[taskArray[i] : (taskArray[i + 1] - 1)],
    #                  numberOfVertices[taskArray[i] : (taskArray[i + 1] - 1)])
end

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

push!(tasks, task)
#mergeObjProcesses(faces_files[taskArray[length(taskArray)] : end],
#                  numberOfVertices[taskArray[length(taskArray)] : end])

for task in tasks
    wait(task)
end

end

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

function mergeAndRemoveDuplicates(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 = Array(Array{Int}, 0)
    FV = Array(Array{Int}, 0)

    offset = 0

    # First of all open files and retrieve LAR models

    f1_V = open(firstPathV)
    f1_FV = open(firstPathFV)

    for ln in eachline(f1_V)
        splitted = split(ln)
        push!(V, [parse(splitted[2]), parse(splitted[3]), parse(splitted[4])])
        offset += 1
    end

    for ln in eachline(f1_FV)
        splitted = split(ln)
        push!(FV, [parse(splitted[2]), parse(splitted[3]), parse(splitted[4])])
    end

    close(f1_V)
    close(f1_FV)

    f2_V = open(secondPathV)
    f2_FV = open(secondPathFV)

    for ln in eachline(f2_V)
        splitted = split(ln)
        push!(V, [parse(splitted[2]), parse(splitted[3]), parse(splitted[4])])
    end

    for ln in eachline(f2_FV)
        splitted = split(ln)
        push!(FV, [parse(splitted[2]) + offset, parse(splitted[3]) + offset, parse(splitted[4])])
    end

    close(f2_V)
    close(f2_FV)

    V_final, FV_final = LARUtils.removeVerticesAndFacesFromBoundaries(V, FV)

```

```

    # Writing model to file
    rm(firstPathV)
    rm(firstPathFV)
    rm(secondPathV)
    rm(secondPathFV)
    writeToObj(V_final, FV_final, firstPath)
end
end

function mergeBoundariesProcess(modelDirectory, startImage, endImage,
                                imageDx, imageDy,
                                imageWidth, imageHeight)

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

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

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

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

function mergeBoundaries(modelDirectory,
                          imageHeight, imageWidth, imageDepth,
```

```

                                imageDx, imageDy, imageDz)
"""
Merge boundaries files. For every cell of size
(imageDx, imageDy, imageDz) in the model grid,
it merges right faces with next left faces, top faces
with the next cell bottom faces, and front faces
with the next cell back faces

modelDirectory: directory containing models
imageHeight, imageWidth, imageDepth: images sizes
imageDx, imageDy, imageDz: sizes of cells grid
"""

beginImageStack = 0
endImage = beginImageStack

tasks = Array{RemoteRef, 0}
for zBlock in 0:(imageDepth / imageDz - 1)
    startImage = endImage
    endImage = startImage + imageDz
    task = @spawn mergeBoundariesProcess(modelDirectory, startImage, endImage,
                                         imageDx, imageDy,
                                         imageWidth, imageHeight)

    push!(tasks, task)
end

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

```

PngStack2Array3dJulia

```

"src/PngStack2Array3dJulia.jl" 40≡
module PngStack2Array3dJulia

export calculateClusterCentroids, pngstack2array3d, getImageData, convertImages

using Images # For loading png images
using Colors # For grayscale images

```



```

using PyCall # For including python clustering
using Logging
@pyimport scipy.ndimage as ndimage
@pyimport scipy.cluster.vq as cluster

NOISE_SHAPE_DETECT=10

function getImageData(imageFile)
    """
    Get width and height from a png image
    """

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

    if (data[2:4] != [80, 78, 71] && data[13:16] != [73, 72, 68, 82])
        error("This is not a 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

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

    rgb_img = convert{Image{ColorTypes.RGB}}, img)
    gray_img = convert{Image{ColorTypes.Gray}}, rgb_img)
    imArray = raw(gray_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)

# Segmenting image using kmeans
# https://en.wikipedia.org/wiki/Image\_segmentation#Clustering\_methods

centroids,_ = cluster.kmeans(pixel, numberOfClusters)

return centroids
end

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

        # Converting image in grayscale
        rgb_img = convert{Image{ColorTypes.RGB}}(img)
        gray_img = convert{Image{ColorTypes.Gray}}(rgb_img)
        imArray = raw(gray_img) # Putting pixel values into RAW 3d array
        debug("imArray size: ", size(imArray))
    end
end

```

```

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

end

# Removing noise using a median filter and quantization
for page in 1:length(image3d)

    # Denoising
    image3d[page] = ndimage.median_filter(image3d[page], NOISE_SHAPE_DETECT)

    # 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)
    qnt,_ = cluster.vq(pixel,centroids)

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

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

    image3d[page] = tmp

end

return image3d
end

function convertImages(inputPath, outputPath, bestImage)
    """
    Get all images contained in inputPath directory
    saving them in outputPath directory in png format.
    If images have one of two odd dimensions, they will be resized
    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
bestImage: Image chosen for centroids computation

Returns the new name for the best image
"""

```
imageFiles = readdir(inputPath)
numberOfImages = length(imageFiles)
outputPrefix = ""
for i in 1: length(string(numberOfImages)) - 1
    outputPrefix = string(outputPrefix,"0")
end

newBestImage = ""
imageNumber = 0
for imageFile in imageFiles
    img = imread(string(inputPath, imageFile))

    # resizing images if they do not have even dimensions
    dim = size(img)
    if(dim[1] % 2 != 0)
        debug("Image has odd x; resizing")
        xrange = 1: dim[1] - 1
    else
        xrange = 1: dim[1]
    end

    if(dim[2] % 2 != 0)
        debug("Image has odd y; resizing")
        yrange = 1: dim[2] - 1
    else
        yrange = 1: dim[2]
    end

    img = subim(img, xrange, yrange)

    outputFilename = string(outputPath, outputPrefix[length(string(imageNumber)):end], imageNumber, ".png")
    imwrite(img, outputFilename)

    # Searching the best image
    if(imageFile == bestImage)
        newBestImage = string(outputPrefix[length(string(imageNumber)):end], imageNumber, ".png")
    end

    imageNumber += 1
end
```

```

end

# Adding another image if they are odd
if(numberOfImages % 2 != 0)
    debug("Odd images, adding one")
    bestImage = imread(string(outputPath, "/", newBestImage))
    imArray = zeros(Uint8, size(bestImage))
    img = grayim(imArray)
    outputFilename = string(outputPath, "/", outputPrefix[length(string(imageNumber)):end], im
    imwrite(img, outputFilename)
end

return newBestImage
end
end
◇

```

10.1 Installing the library

11 Conclusions

11.1 Results

11.2 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.

A Utility functions

B Tests

Generation of the border matrix

"test/generateBorderMatrix.jl" 45≡

[illegible]

```

end

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

executeAllTests()

◇

```

Conversion of a png stack to a 3D array

```

"test/pngStack2Array3dJulia.jl" 47≡
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" 48≡
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()

◇

```