Asphalt Mixture Aging Simulator Documentation

Release 0.1

Jeison Pacateque y Santiago Puerto

Dec 12, 2017

Contents

This project perform a 3D reconstruction of a cylindrical asphalt mixture sample from a set images on a Dicom file format. The components of asphalt mixture will be identified through 3-phase segmentation. Additionally, a 3D model of the asphalt mixture reconstruction was developed. The project was implemented on the Python programming language using the open-source libraries Numpy, Scipy, Pydicom, Scikit-learn, Matplotlib and Mayavi. A simulation of the asphalt mixtures aging process is implemented using numerical methods on the mechanical, themical and chemical fields.

Contents:

CHAPTER 1

Simulation model

class app.simulation.fem_mechanics.**FEMMechanics**(*matrix_materials*)

This class supports the Mechanic FEM Simulation matrix_materials = initial matrix materials (numpy objetc array, array of class Material)

_createStiffnessMatrix()

This function uses the LinearBarElementStiffness to create the stiffness matrix (ki) for each Finite Element to create regarding the matrix materials (MM) it also configures each FE element with their Young's Modulus and their transversal area

_createConectivityMatrix()

This function initialices the conectivity_matrix object using the matrix materials (MM) size for reference, additionally, this method declares the lists where the top and bottom nodes will be

_LinearBarElementForces(*k*, *u*)

This function returns the element nodalforce vector given the element stiffness matrix k and the element nodal displacement vector u.

_LinearBarElementStresses(*k*, *u*, *A*)

This function returns the element nodal stress vector given the element stiffness matrix k, the element nodal displacement vector u, and the cross-sectional area A.

_LinearBarElementStiffness(*E*, *A*, *L*)

This function returns the Finite Element stiffness considering the material Young's modulus (E), the transversal area of the FE (A) and the length of the FE (L)

Definition

$$k = \begin{bmatrix} \frac{EA}{L} & -\frac{EA}{L} \\ -\frac{EA}{L} & \frac{EA}{L} \end{bmatrix}$$

_generalStiffnessMatrixAssemble()

Assembles the General Stiffness Matrix (K) using the size of the matrix materials (MM) as reference. It also uses the conectivity matrix (conectivity_matrix) to asseble the FE for every material on the matrix materials (MM) by using the function _LinearBarAssemble order to locate all the FE in place for simulation

_LinearBarAssemble(*K*, *k*, *i*, *j*)

This function assembles the element stiffness matrix k of the linear bar with nodes i and j into the global stiffness matrix K.This function returns the global stiffness matrix K after the element stiffness matrix k is assembled.

_ElementConectivityMatrix(*width*, *height*)

This function create the nodes and set positions for all elements on a stiffness matrix. It also it also aggregate the top and bottom elements to their own list declared at the function _CreateConectivityMatrix

applySimulationConditions(*force*)

Set the force parameter to apply over the top elements of the FE General Stiffness Matrix (K)

simulate()

Run the simulation with all the configured parameters, the output will be a displacements map handled by the results module. Once the global stiffness matrix K is obtained we have the following structure equation:

Definition

$$[K]\{U\} = \{F\}$$

where U is the global nodal displacement vector and F is the global nodal force vector.At this step the boundary conditions are applied manually to the vectors U and F.

Then the matrix is solved by partitioning and Gaussian elimination. Finally once the unknown displacements and reactions are found, the element forces are obtained for each element as follows:

$$[f] = [k]\{u\}$$

where f is the 2x1 element force vector and u is the 2x1 element displacement vector. The element stresses are obtained by dividing the element forces by the crosssectional area A.

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

CHAPTER 2

Image processing module

class app.imgprocessing.segmentation.**Segmentation**

This class provides the methods to reduce and segment the images of the toy model. It assumes that the set of Dicom files that represents the toy model have been tranformed in an array of numpy.

reduction(*img*, *factor=0.2222222222222222*)

This method reduces the set of images of the toymodel by a given scale factor or zoom factor, using the C-Spline algorithm which is provided by the ndimage module of scipy

C-Spline algorithm consists in funding a function which is the linear combination of piecewise definened functions known as Basis Splines (B-Splines), which are smooth functions whose first, second and third derivative pass through one point of the given discrete set:

$$\beta^3(x) = \begin{cases} \frac{2}{3} - |x|^2 + \frac{|x|^3}{2} & 0 \leqslant |x| < 1 \\ \frac{(2-|x|)^3}{6} & 1 \leqslant |x| < 2 \\ 0 & 2 \leqslant |x| \end{cases}$$

The linear combination of the B-Spline functions can be expressed by the next equation:

$$\zeta(x) = \sum_{k \in Z} c(K)\beta^n(x - K)$$

Parameters

- **img** (*3d numpy array*) – representation of the toy model
- **factor** (*float*) – zoom factor in which the image is reduced

Returns the image rescaled

Return type 3d numpy array

view(*original*, *segmented*, *reduced*)

This method is implemented for test purposes, it takes as arguments an untreated slice, a segmented slice and a reduced and segmented slice showing its differences on screen using a matplotlib figure

view(original, segmented, reduced)

histogram(*img_red*)

Plots an histogram of the materials distibution over the toy model using matplotlib. It is neccesary to reduce the toy model before plotting the histogram. For test porpuses

histogram(reduced_toymodel)

clasify(*img*, *normalize=True*, *n_clusters=3*)

This method segments or clasify the values of the given image in three different groups of values, thus the different Hounsfield unit values found in the image are replaced by only three different values. If the parameter normalize is true, these values are:

- **0** for the air-void
- **1** for the mastic
- **2** for the aggregates

For this porpuse, a implementation of the K-means algorithm, provided by the cluster module of the Scikit-learn library, is used. K-means algorithm takes a dataset X of N values, and a parameter K specifies how many cluster to create. K-means finds evenly-spaced sets of points in subsets of Euclidean spaces called Voronoi diagrams. Each found partitions will be a uniformly shaped region called Voronoi cell, one for each material. This process is executed in two steps:

- The assign step consists in calculating a Voronoi diagram having a set of centroids μ_n. The clusters are updated to contain the closest points in distance to each centroid as it is described by the equation:

$$c_k = \{X_n : \|X_n - \mu_k\| \leqslant \|X_n - \mu_l\|\}$$

- The upadate step, given a set of clusters, recalculates the centroids as the means of all points belonging to a cluster:

$$\mu_k = \frac{1}{C_k} \sum_{X_n \in C_k} Xn$$

The k-means algorithm loops through the two previous steps until the assignments of clusters and centroids no longer change. The convergence is guaranteed but the solution might be a local minimum as shown in the next equation:

$$\sum_{k=1}^{K} \sum_{X_n \in C_k} \|X_n - \mu_k\|^2 \text{, with respect to } C_k, \mu_k$$

Parameters

- **img** (*2d numpy array*) – a slice of the toy model
- **= 3 # number of clusters** (*n_clusters*) – void, aggregate and mastic

:type int :param boolean normalize: If it is true, the segmentation mark the three

group of values as 0, 1 and 2. If it is false, the marks are the default values generated by k-means.

Returns the image segmented, with only three different possible values

Return type 3d numpy array

segment_all_samples(*samples*)

Take the given samples, uses K-Means algorithm with each sample slice and returns all the segmented samples. it also cuts irrelevant data corresponding to voids outside of the lenth of the radius of the toymodel

Parameters **samples** (*list of 2d numpy arrays*) – the set of slices of the toy model

Returns the toy model with its materials classified in airvoids, mastic and aggregates.

Return type list of 2d numpy arrays

app.imgprocessing.slice_mask.**sector_mask**(*shape*, *centre=(50, 50)*, *radius=50*, *angle_range=(0, 360)*)

This method provides a circular mask over a numpy array (image), its purpose is to differentiate the air void pixels within the cylindrical toymodel from the air void space outward.

Parameters

- **shape** (*two-dimensional tuple of integers*) – shape of the image
- **centre** (*two-dimensional tuple of integers*) – point from where the circular mask is applied
- **radius** (*float*) – length of the radius for the circular mask
- **angle_range** (*two-dimensional tuple of integers*) – circular sector where the mask is applied, the whole circle by default

Returns mask for a circular sector

Return type 2d boolean numpy array

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

C

F

H

R

S

V