Parallel & Distributed Computing: Lecture 12

Alberto Paoluzzi

November 21, 2017

Project: Add-ons modules to LARLIB.jl

Julia unit testing

1. Step three: constuction of a file `larlib.largrid-test.jl`

2. Step Four: integration test

Introduction from largrid.pdf

This report aims to discuss the design and the implementation of the `largrid` module of the LAR-CC library, including also the Cartesian product of general cellular complexes. In particular, we show that both n-dimensional grids of (hyper)-cuboidal cells and their d-dimensional skeletons ($0 \leq d \leq n$), embedded in $\mathbb{E}^n$, may be properly and efficiently generated by assembling the cells produced by a number n of either 0- or 1-dimensional cell complexes, that in such lowest dimensions coincide with simplicial complexes.

In Section 2 we give the simple implementation of generation of lower-dimensional (say, either 0- or 1-dimensional) regular cellular complexes with integer coordinates. In Section 3 a functional decomposition of the generation of either full-dimensional cuboidal complexes in $\mathbb{E}^n$ and of their d-skeletons ($0 \leq d \leq n$) is given, showing in particular that every skeleton can be efficiently generated as a partition in cell subsets produced by the Cartesian product of a proper disposition of 0-1 complexes, according to the binary representation of a subset of the integer interval $[0, 2^n]$. In Section 5 we provide a very simple and general implementation of the topological product of *two* cellular complexes of any topology. When applied to embedded linear cellular complexes (i.e. when the coordinates of 0-cells of arguments are fixed and given) the algorithm produces a Cartesian product of its two arguments. In Section 6 the exporting of the module to different languages is provided. The Section 7 contains the unit tests associated to the various algorithms, that are exported by the used literate environment in the proper test subdirectory—depending on the implementation

Figure 1: Introduction to module

Step three: constuction of a file `larlib.largrid-test.jl`

Unit testing (from Wikipedia)

Unit testing refers to tests that verify the functionality of a specific section of code, usually at function level.

These types of tests are usually written by developers as they work on code (white-box style), to ensure that the specific function is working as expected.

One function might have multiple tests, to catch corner cases or other branches in the code.

Unit testing alone cannot verify the functionality of a piece of software, but rather is used to ensure that the building blocks of the software work independently from each other.

Integration testing (from Wikipedia)

Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design.

Integration testing works to _expose defect_s in the interfaces and interaction between integrated components (modules).

Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.

Julia unit testing

The Julia Language

stable

Search docs

Home

Manual

Introduction

Getting Started

Basic Unit Tests

The `Base.Test` module provides simple *unit testing* functionality. Unit testing is a way to see if your code is correct by checking that the results are what you expect. It can be helpful to ensure your code still works after you make changes, and can be used when developing as a way of specifying the behaviors your code should have when complete.

Simple unit testing can be performed with the `@test()` and `@test_throws()` macros:

`Base.Test.@test` — *Macro.*

```
@test ex
@test f(args...) key=val ...
```

Tests that the expression `ex` evaluates to `true`. Returns a `Pass` `Result` if it does, a `Fail` `Result` if it is `false`, and an `Error` `Result` if it could not be evaluated.

Assignment

Read this manual section . . . (try the examples!)

Exacute the ported (partial) module LarGrid

```
$ run larGrid.jl
```

Assignment

Read the LarGrid.jl file and step-wise copy it to a Julia notebook

Step Four: integration test

Cartesian product of cellular complexes (Math.StackExchange)

Cartesian product of two CW-complexes

8

Let's A and B are CW-complexes. How to construct CW-complex $A \times B$?

Thanks.

(general-topology) (cw-complexes)

5

share cite improve this question

edited Apr 30 '14 at 12:25
Stefan Hamcke
20.2k ● 4 ■ 24 ▲ 73

asked Jan 18 '12 at 16:09
Aspirin
1,976 ● 1 ■ 16 ▲ 30

add a comment

asked 5 years, 10 months ago
viewed 2,829 times
active 3 years, 6 months ago

Linked

- 0 Products of cells in a CW complex
- 0 Attaching maps in a product cell complex
- 2 Cell structure of $S^2 \times S^1$
- 0 Is it possible to view $S^k \times \{0,1\}$ as a CW subcomplex of $S^k \times I$ by defining a proper CW structure on $S^k \times I$?

Related

- 3 Why is a finite CW complex compact?
- 5 Products of CW-complexes
- 4 Product of CW complexes question
- 0 Suspension of a CW complex
- 0 Attaching maps in a product cell complex

1 Answer

active oldest **votes**

6 ✓

Choose a CW-complex structure for A using cells e_α with attaching maps φ_α. Do the same for B using cells e_β and attaching maps φ_β. Then the products $e_\alpha \times e_\beta$ are cells and the maps $\varphi_\alpha \times \varphi_\beta$ are attaching maps for a CW-complex structure on $A \times B$.

If you are looking for details of a proof, one can be found in Hatcher's book where the statement appears as Theorem A.6.

share cite improve this answer

answered Jan 18 '12 at 17:23
wckronholm
2,992 ● 1 ■ 11 ▲ 27

Cartesian product of cellular complexes (LAR + Julia)

```
function larModelProduct(modelOne, modelTwo)
    (V, cells1) = modelOne
    (W, cells2) = modelTwo
    k = 1
    vertices = OrderedDict()
    for v in V
        for w in W
            id = [v;w]
            if haskey(vertices, id) == false
                vertices[id] = k
                k = k + 1
            end
        end
    end
    cells = []
    for c1 in cells1
        for c2 in cells2
            cell = []
            for vc in c1
                for wc in c2
                    push!(cell, vertices[[V[vc];W[wc]]] )
                end
            end
            push!(cells, cell)
        end
    end
    vertexmodel = []
    for v in keys(vertices)
        push!(vertexmodel, v)
    end
    return (vertexmodel, cells)
end
```

Cartesian product of cellular complexes (example)

```
geom_0 = [[x] for x=0.:10]
topol_0 = [[i,i+1] for i=1:9]
geom_1 = [[0.],[1.],[2.]]
topol_1 = [[1,2],[2,3]]
model_0 = (geom_0,topol_0)
model_1 = (geom_1,topol_1)

model_2 = larModelProduct(model_0, model_1)
model_3 = larModelProduct(model_2, model_1)
```

PyPlasm rendering

```
using PyCall
@pyimport larlib as p

function larView(V::Array{Any,1},CV::Array{Any,1})
    V = hcat(V[1],V...)
    W = [Any[V[h,k] for h=1:size(V,1)] for k=1:size(V,2)]
    p.VIEW(p.STRUCT(p.MKPOLS(PyObject([W,CV,[]]))))
end

function larExplodedView(V::Array{Any,1},CV::Array{Any,1})
    V = hcat(V[1],V...)
    W = [Any[V[h,k] for h=1:size(V,1)] for k=1:size(V,2)]
    p.VIEW(p.EXPLODE(1.2,1.2,1.2)(p.MKPOLS(PyObject([W,CV,[]]))))
end

larView(model_3...)
larExplodedView(model_3...)
larExplodedView(model_3[1],model_3[2])
```

Other rendering methods

```
function larExplodedView(V::Array{Any,2},CV::Array{Any,2})
    Z = hcat(V[:,1],V)
    W = [Any[Z[h,k] for h=1:size(Z,1)] for k=1:size(Z,2)]
    CV = hcat(CV'...)
    CW = [Any[CV[h,k] for h=1:size(CV,1)] for k=1:size(CV,2)]
    p.VIEW(p.EXPLODE(1.2,1.2,1.2)(p.MKPOLS(PyObject([W,CW,[]]))))
end

function larExplodedView(V::Array{Int64,2},CV::Array{Array{Int64,1},1})
    Z = hcat(V[:,1],V)
    W = [Any[Z[h,k] for h=1:size(Z,1)] for k=1:size(Z,2)]
    CW = [Any[cell[h] for h=1:length(cell)] for cell in CV]
    p.VIEW(p.EXPLODE(1.2,1.2,1.2)(p.MKPOLS(PyObject([W,CW,[]]))))
end

shape = (40,20,10)
cubes = larCuboids(shape,true)
V,FV = cubes[1],cubes[2][3]
larExplodedView(V,FV)
```

Conclusion

No good system integration of `larModelProduct` with `larCuboids`