

Parallel \& Distributed Computing: Lecture 12

Alberto Paoluzzi

November 29, 2018

Test Driven Development (TDD)

- 1 Space Arrangement (Continue from lecture 12)
- 2 Test Driven Development (TDD)
- 3 Example: Topological gift-wrapping
- 4 Julia implementation
- 5 Test-driven development example

Space Arrangement (Continue from lecture 12)

Definitions

- **Arrangement** is the decomposition of the d -dimensional affine or projective space,
- into connected and (relatively open) cells of lower dimensions
- induced by an intersection of a finite collection of geometric objects

Formally:

- arrangement $\mathcal{A}(\mathcal{S}) = X$,
- where $X := \bigcup_{k=0}^d X_k$,
- X_k is called the k -skeleton of the cellular complex X , usually with $d \in \{2, 3\}$,
- providing a cellular decomposition of the space \mathbb{E}^d where the {underlying space} (point-set) of \mathcal{S} is embedded.

Computational goal

Given the **input collection \mathcal{S}** , how to compute and represent the **chain complex**

$$C_{\bullet} := C_3 \xrightleftharpoons[\partial_3]{\delta_2} C_2 \xrightleftharpoons[\partial_2]{\delta_1} C_1 \xrightleftharpoons[\partial_1]{\delta_0} C_0,$$

where C_p ($0 \leq p \leq d$), with $d \in \{2, 3\}$, is a **linear space of p -chains** (sets of p -cells with algebraic structure), and where $\delta_{p-1} = \partial_p^{\top}$.

Arrangement of cellular complexes: a quick tour

Paoluzzi, Shapiro, DiCarlo, 2018 (submitted)

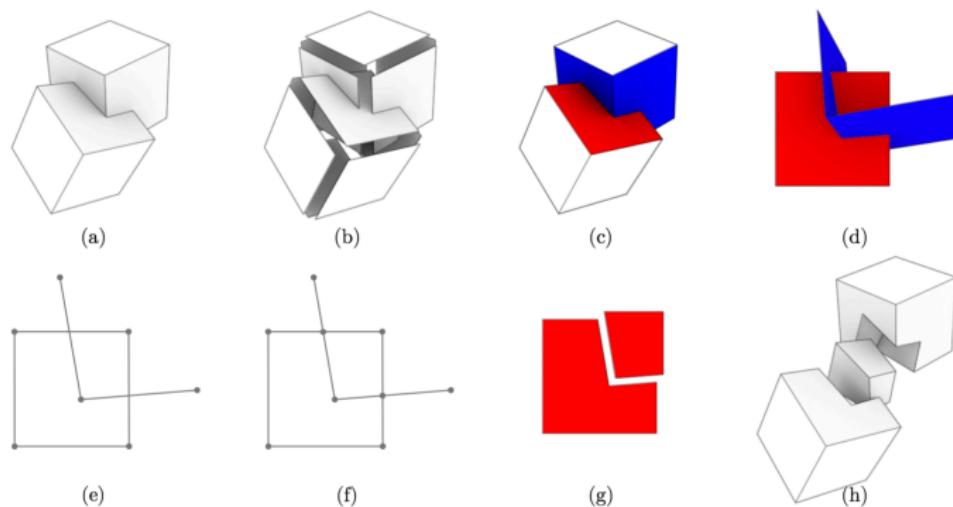


Fig. 1. Cartoon display of the Merge algorithm: (a) the two input solids; (b) the exploded input collection B of 2-cells embedded in \mathbb{E}^3 ; (c) 2-cell σ (red) and the set $\Sigma(\sigma)$ (blue) of possible intersection; (d) affine map of $\sigma \cup \Sigma$ on $z = 0$ plane; (e) reduction to a set of 1D segments in \mathbb{E}^2 ; (f) pairwise intersections; (g) regularized plane arrangement $\mathcal{A}(\sigma \cup \Sigma)$; (h) exploded 3-complex extracted from the 2-complex $X_2(\cup_{\sigma \in B} \mathcal{A}(\sigma \cup \Sigma))$ in \mathbb{E}^3 . The LAR representation of both the input data (not a complex) and the output data (3-complex with three 3-cells) is given in Appendix A

Test Driven Development (TDD)

Definitions

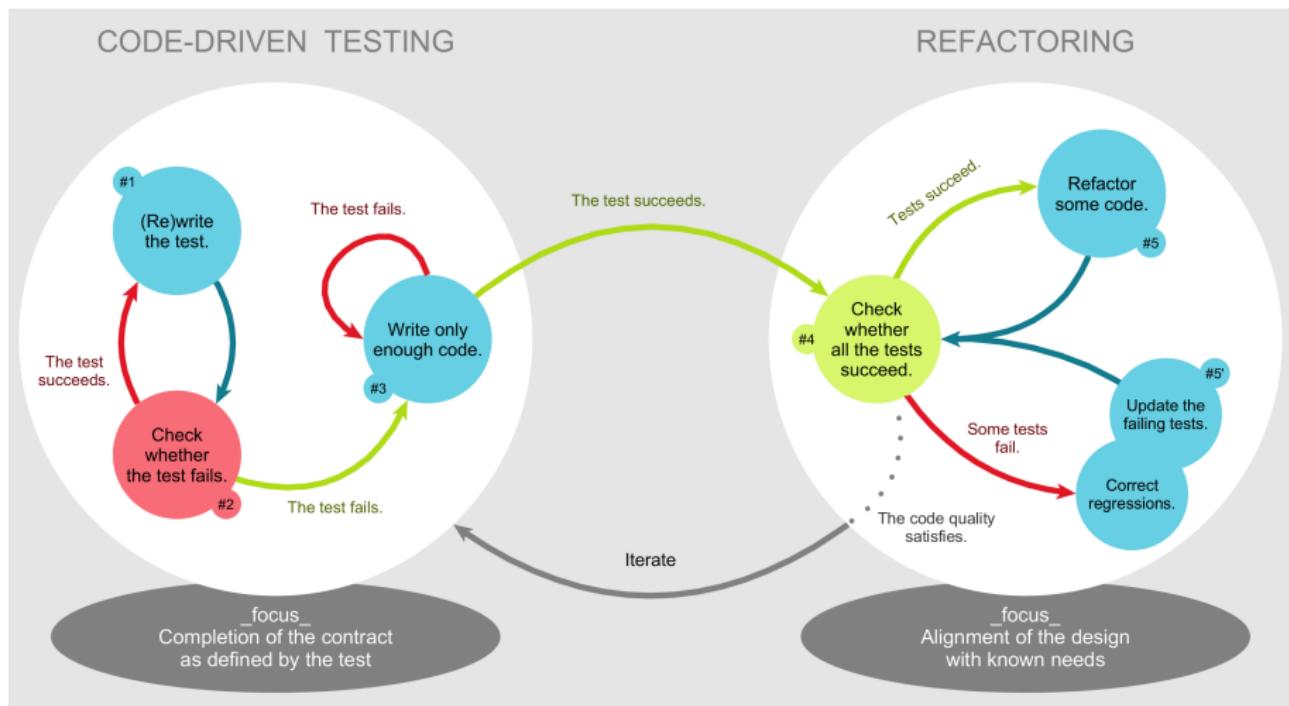
(*Wikipedia**)

Test-driven development (TDD) is a **software development process** that relies on the repetition of a **very short development cycle**

- requirements are turned into very specific test cases, then the software is improved to pass the new tests, only
- opposed to software development that allows software to be added that is not proven to meet requirements
- also apply the concept to improving and debugging legacy code developed with older techniques.

State diagram of DDT

(*Wikipedia**)

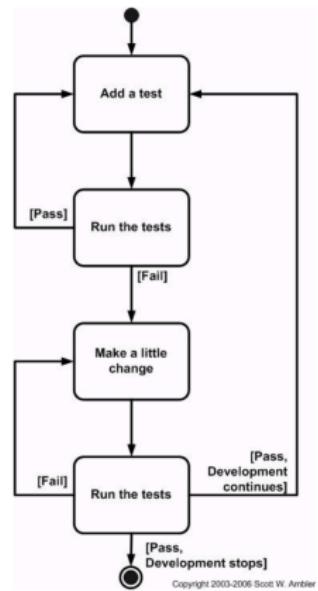


Test-Driven Development workflow

The following sequence is based on the book Beck, K. [Test-Driven Development by Example](#), Addison Wesley - Vaseem, 2003.

See also [Test-Driven Development by Example](#)

- ① Add a test
- ② Run all tests and see if the new test fails
- ③ Write the code
- ④ Run tests
- ⑤ Refactor code
- ⑥ Repeat



Copyright 2003-2006 Scott W. Ambler

Example: Topological gift-wrapping

Arrangement of cellular complexes: a quick tour

Paoluzzi, Shapiro, DiCarlo, 2018 (submitted)

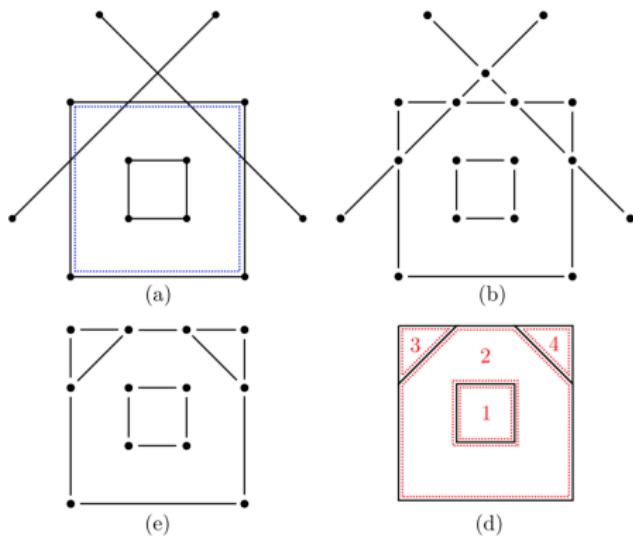


Fig. 2. Basic case: computation of the regularized arrangement of a set of lines in \mathbb{E}^2 : (a) the input, i.e. the 2-cell σ (blue) and the line segment intersections of $\Sigma(\sigma)$ with $z = 0$; (b) all pairwise intersections; (c) removal of the 1-subcomplex external to σ . To this purpose an efficient and robust point classification algorithm [?] is used, ; (d) interior cells of the regularized 2-complex $X_2 = \mathcal{A}(\sigma \cup \Sigma)$ generated as arrangement of \mathbb{E}^2 produced by $\sigma \cup \Sigma$.

Arrangement of cellular complexes: a quick tour

Paoluzzi, Shapiro, DiCarlo, 2018 (submitted)

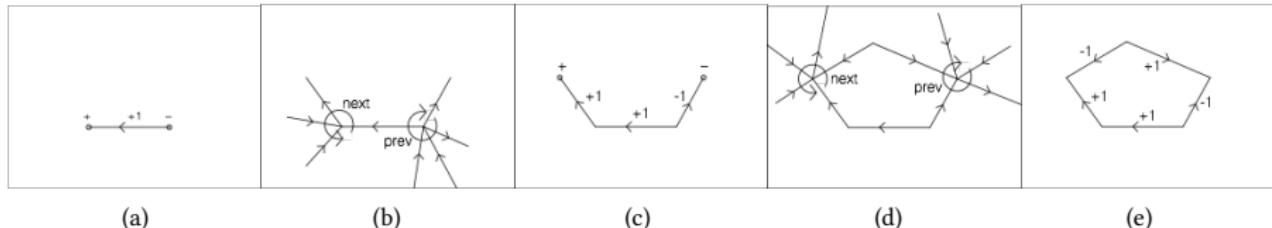


Fig. 11. Extraction of a minimal 1-cycle from $\mathcal{A}(X_1)$: (a) the initial value for $c \in C_1$ and the signs of its oriented boundary; (b) cyclic subgroups on $\delta\partial c$; (c) new (coherently oriented) value of c and ∂c ; (d) cyclic subgroups on $\delta\partial c$; (e) final value of c , with $\partial c = \emptyset$.

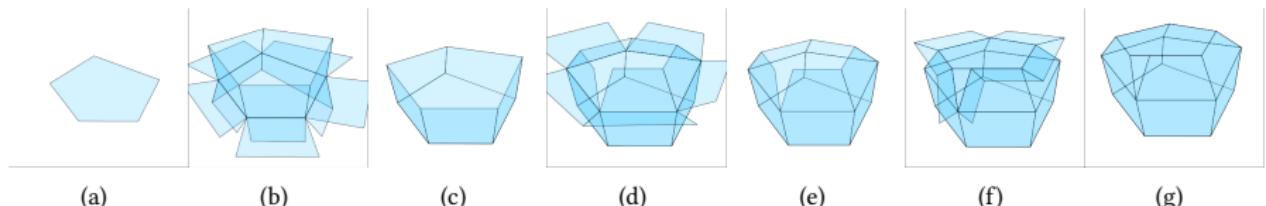


Fig. 9. Extraction of a minimal 2-cycle from $\mathcal{A}(X_2)$: (a) 0-th value for $c \in C_2$; (b) cyclic subgroups on $\delta\partial c$; (c) 1-st value of c ; (d) cyclic subgroups on $\delta\partial c$; (e) 2-nd value of c ; (f) cyclic subgroups on $\delta\partial c$; (g) 3-rd value of c , such that $\partial c = 0$, hence stop.

Arrangement of cellular complexes: a quick tour

Paoluzzi, Shapiro, DiCarlo, 2018 (submitted)

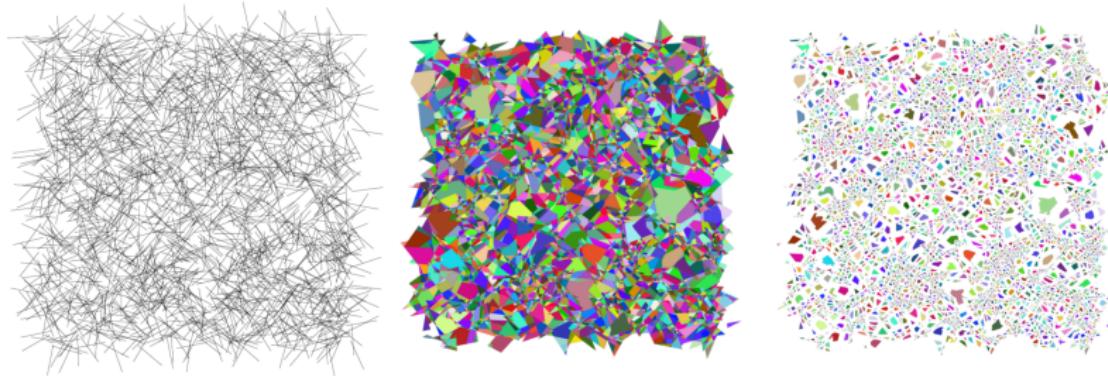


Fig. 5. The regularized 2D arrangement X_2 of the plane generated by a set of random line segments. Note that cells $\sigma \in X_2$ are not necessarily convex. The Euler characteristic is $\chi = \chi_0 - \chi_1 + \chi_2 = 11361 - 20813 + 9454 = 2$.

Arrangement of cellular complexes: a quick tour

Paoluzzi, Shapiro, DiCarlo, 2018 (submitted)

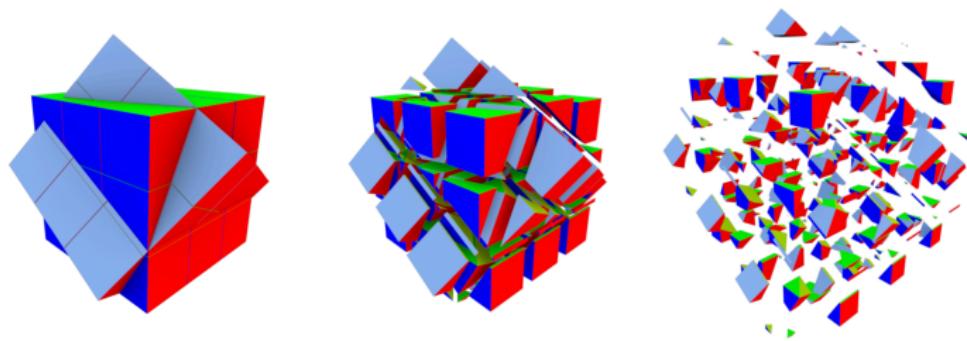


Fig. 6. Merge of two rotated $3 \times 3 \times 3$ cuboidal 3-complexes: (a) the initial positions of the two input 3-complexes, each with 3^3 unit cubic cells; (b) 3-cells of merged complex, with space explosion of small scaling parameter; (c) larger space explosion. Each exploded cell is translated by a scaling-induced movement of its centroid. Output cells are not necessarily convex. In the merged complex we get 236 three-cells and 816 two-cells, both possibly non-convex. Ratio new/old 3-cells is 4.3704.

Arrangement of cellular complexes: a quick tour

Paoluzzi, Shapiro, DiCarlo, 2018 (submitted)

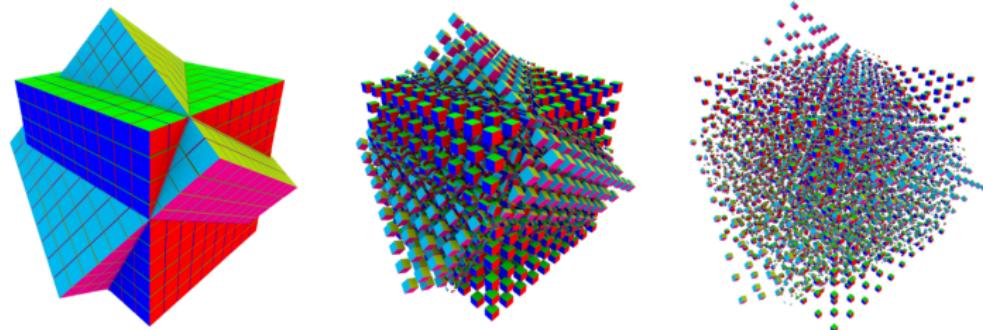


Fig. 7. Merge of two 10^3 3-complexes. In the merged complex we get 8655 3-cells, 26600 2-cells (faces), 26732 1-cells (edges), and 8787 0-cells (vertices). Please note that the Euler characteristic is $\chi = \chi_0 - \chi_1 + \chi_2 - \chi_3 = 8787 - 26732 + 26600 - 8655 = 0$, since the decomposed 3-complex (no its exploded image!) is homeomorphic to the n -sphere, where $\chi = 1 + (-1)^n$, with $n = 3$. Ratio new/old 3-cells is 4.3275.

Arrangement of cellular complexes: a quick tour

Paoluzzi, Shapiro, DiCarlo, 2018 (submitted)

ALGORITHM 3: Fragmentation of 2-cells

```

Input:  $\mathcal{S}_2 \subset \mathcal{S}_{d-1}$       # collection of all 2-cells from  $\mathcal{S}_{d-1}$  input in  $\mathbb{B}^d$ 
Output:  $[\partial_2]$       # CSC signed matrix
 $\widetilde{\mathcal{S}}_2 = \emptyset$       # initialisation of collection of local fragments
for  $\sigma \in \mathcal{S}_2$  do      # for each 2-cell  $\sigma$  in the input set
     $M = SubManifoldMap(\sigma)$       # affine transform s.t.  $\sigma \mapsto x_3 = 0$  subspace
     $\Sigma = M \mathcal{I}(\sigma)$       # apply the transformation to (possible) incidencies to  $\sigma$ 
     $\mathcal{S}_1(\sigma) = \emptyset$       # collection of line segments in  $x_3 = 0$ 
    for  $\tau \in \Sigma$  do      # for each 2-cell  $\tau$  in  $\Sigma$ 
         $\mathcal{P}(\tau), \mathcal{L}(\tau) = \emptyset, \emptyset$       # intersection points and int. segment(s) with  $x_3 = 0$ 
        for  $\lambda \in X_1(\tau)$  do      # for each 1-cell  $\lambda$  in  $X_1(\tau)$ 
            if  $\lambda \notin \{q \mid x_3(q) = 0\}$  then  $\mathcal{P}(\tau) += \{p\}$       # append the intersection point of  $\lambda$  with  $x_3 = 0$ 
        end
         $\mathcal{L}(\tau) = Points2Segments(\mathcal{P}(\tau))$       # Compute a set of collinear intersection segments
         $\mathcal{S}_1(\sigma) += \mathcal{L}(\tau)$       # accumulate intersection segments with  $\sigma$  generated by  $\tau$ 
    end
     $X_2(\sigma) = \mathcal{A}(\mathcal{S}_1(\sigma))$       # arrangement of  $\sigma$  space induced by a soup of 1-complexes
     $\widetilde{\mathcal{S}}_2 += M^{-1} X_2$       # accumulate local fragments, back transformed in  $\mathbb{B}^d$ 
end
 $[\partial_1] = QuotientBases(\widetilde{\mathcal{S}}_2)$       # identification of 0- and 1-cells using kd-trees and canonical LAR
 $[\partial_2] = Algorithm\_1([\partial_1])$       # output computation GRANDE STRONZATA
return  $[\partial_2]$ 

```

Arrangement of cellular complexes: a quick tour

Paoluzzi, Shapiro, DiCarlo, 2018 (submitted)

ALGORITHM 1: Computation of signed $[\partial_d^+]$ matrix

```

/* Pre-condition: d equal to space dimension, s.t.  $(d - 1)$ -cells are shared by two  $d$ -cells */  

/* */  

Input:  $[\partial_{d-1}]$  # CSC signed matrix  

Output:  $[\partial_d^+]$  # CSC signed matrix  

 $m, n = [\partial_{d-1}].size; marks = Zeros(n)$  # initializations  

while Sum(marks) <  $2n$  do  

     $\sigma = Choose(marks)$  # select the  $(d - 1)$ -cell seed of the column extraction  

    if marks[ $\sigma$ ] == 0 then  $[c_{d-1}] = [\sigma]$   

    else if marks[ $\sigma$ ] == 1 then  $[c_{d-1}] = [-\sigma]$   

     $[c_{d-2}] = [\partial_{d-1}][c_{d-1}]$  # compute boundary  $c_{d-2}$  of seed cell  

    while  $[c_{d-2}] \neq []$  do # loop until boundary becomes empty  

        corolla = []  

        for  $\tau \in c_{d-2}$  do # for each hinge  $\tau$  cell  

             $[b_{d-1}] = [\tau]^t[\partial_{d-1}]$  # compute the  $\tau$  coboundary  

            pivot =  $\{|b_{d-1}|\} \cap \{|c_{d-1}|\}$  # compute the  $\tau$  support  

            if  $\tau > 0$  then adj = Next(pivot, Ord( $b_{d-1}$ )) # compute the new adj cell  

            else if  $\tau < 0$  then adj = Prev(pivot, Ord( $b_{d-1}$ ))  

            if  $\partial_{d-1}[\tau, adj] \neq \partial_{d-1}[\tau, pivot]$  then corolla[adj] =  $c_{d-1}[pivot]$  # orient adj  

            else corolla[adj] =  $-(c_{d-1}[pivot])$   

        end  

         $[c_{d-1}] += corolla$  # insert corolla cells in current  $c_{d-1}$   

         $[c_{d-2}] = [\partial_{d-1}][c_{d-1}]$  # compute again the boundary of  $c_{d-1}$   

    end  

    for  $\sigma \in c_{d-1}$  do marks[ $\sigma$ ] += 1 # update the counters of used cells  

     $[\partial_d^+] += [c_{d-1}]$  # append a new column to  $[\partial_d^+]$   

end  

return  $[\partial_d^+]$ 
```

Julia implementation

Julia implementation (in the works)

<https://github.com/cvdlab/LinearAlgebraicRepresentation.jl>

Test-driven development example

Start with a flexible test suite

Create a branch in the repo ...

Use the bash-git-prompt ...

```
GNU nano 2.0.6          File: .bash_profile

# Git Bash Completion
# if [ -f $(brew --prefix)/etc/bash_completion ]; then
#   . $(brew --prefix)/etc/bash_completion
# fi

# Git Prompt
# GIT_PS1_SHOWDIRTYSTATE=true
# export PS1='[\u@mbp \w$(_git_ps1)]\$ '

# bash-git-prompt
if [ -f "$(brew --prefix)/opt/bash-git-prompt/share/gitprompt.sh" ]; then
  __GIT_PROMPT_DIR=$(brew --prefix)/opt/bash-git-prompt/share
  source "$(brew --prefix)/opt/bash-git-prompt/share/gitprompt.sh"
fi
```

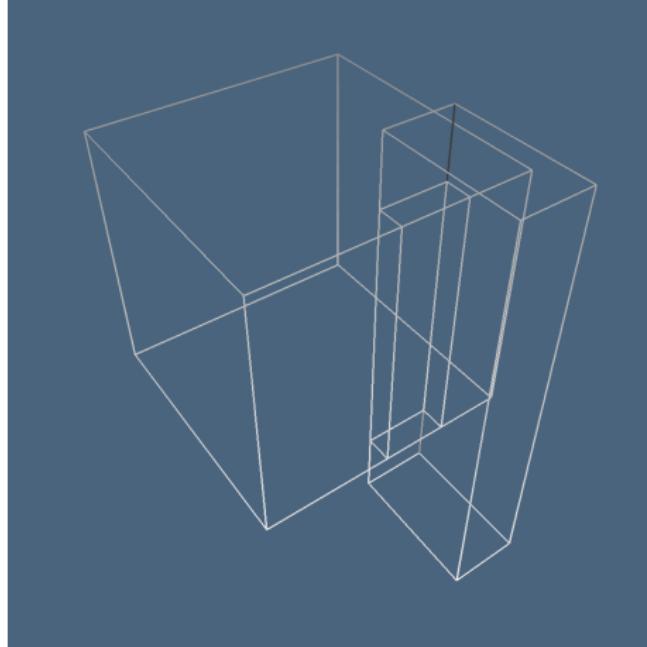
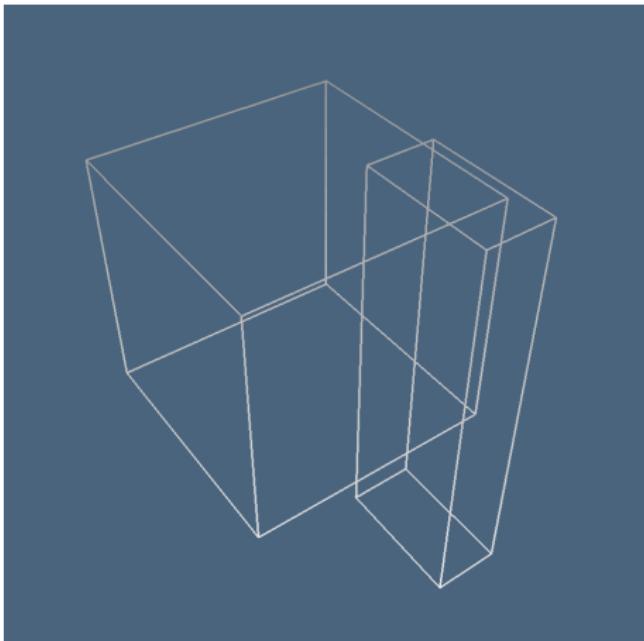
Test suite of data configurations, and get errors

```
using LinearAlgebraicRepresentation
Lar = LinearAlgebraicRepresentation
using Plasm

#translate = Lar.t(0,0,0) #OK
#translate = Lar.t(0.25,0,0) #OK
#translate = Lar.t(0.25,0,0.25) #OK
#translate = Lar.t(0.25,0,-0.25) #OK
#translate = Lar.t(.25,.25,1.0) #OK
#translate = Lar.t(.25,1,-.25) #OK
translate = Lar.t(.25,0.9,-.25) #OK
#translate = Lar.t(.25,.25,-.25) #KO

V1,(_,EV1,FV1,CV1) = Lar.cuboid([1,1,1],true)
V2,(_,EV2,FV2,CV2) = Lar.cuboid([.25,.5,1.5],true)
assembly = Lar.Struct([(V1,FV1,EV1),translate,(V2,FV2,EV2)])
W,FW,EW = Lar.struct2lar(assembly)
Plasm.view(W,EW)
V,(EV,FV,CV),(copEV,copFE,copCF) = Lar.chaincomplex( W,FW,EW )
Plasm.view(V,EV)
```

Test suite of data configurations, and get errors



This example works ...

Test suite of data configurations, and get errors

This one enters an infinite loop ... WHY ?

ANSWER: because loops in faces were not considered (error due to the naif implementation)

```
#translate = Lar.t(0,0,0) #OK
#translate = Lar.t(0.25,0,0) #OK
#translate = Lar.t(0.25,0,0.25) #OK
#translate = Lar.t(0.25,0,-0.25) #OK
#translate = Lar.t(.25,.25,1.0) #OK
#translate = Lar.t(.25,1,-.25) #OK
#translate = Lar.t(.25,0.9,-.25) #OK
translate = Lar.t(.25,.25,-.25) #KO

V1,(_,EV1,FV1,CV1) = Lar.cuboid([1,1,1],true)
V2,(_,EV2,FV2,CV2) = Lar.cuboid([.25,.5,1.5],true)
assembly = Lar.Struct([(V1,FV1,EV1),translate,(V2,FV2,EV2)])
W,FW,EW = Lar.struct2lar(assembly)
Plasm.view(W,EW)
V,(EV,FV,CV),(copEV,copFE,copCF) = Lar.chaincomplex( W,FW,EW )
```

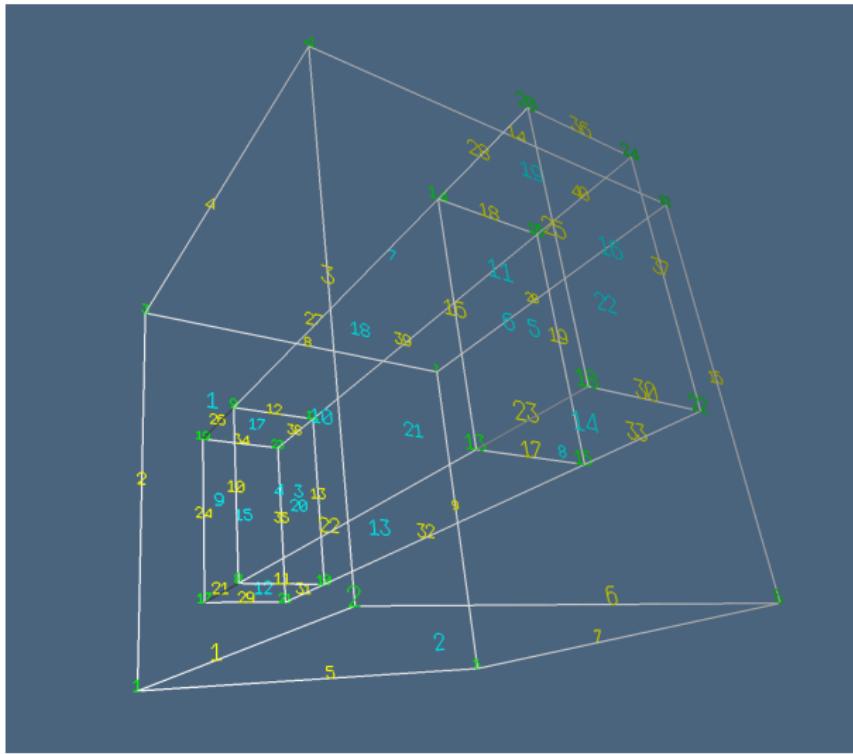
Create more direct test data (w.r.t. error)

```
(V, EV, FV) = ([0.0 0.0 0.0 0.0 0.0 1.0 1.0 1.0 0.25 0.25 0.5
0.5 1.0 0.25 0.25 0.5 0.5 0.25 0.25 0.25 0.25 0.5 0.5 0.5
0.5; 0.0 0.0 1.0 1.0 0.0 0.0 1.0 0.25 0.75 0.25 0.75 1.0
0.25 0.75 0.25 0.75 0.25 0.75 0.75 0.25 0.25 0.75 0.75;
0.0 1.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0 1.0 1.0
1.0 -0.25 1.25 -0.25 1.25 -0.25 1.25 -0.25 1.25],
Array[Int64,1][[1,2],[1,3],[2,4],[3,4],[1,5],[2,6],[5,6],
[3,7],[5,7],[8,9],[8,10],[9,11],[10,11],[4,12],[6,12],[13,
14],[13,15],[14,16],[15,16],[7,12],[8,17],[8,13],[13,18],
[17,19],[18,20],[9,19],[9,14],[14,20],[17,21],[18,22],[10,
21],[10,15],[15,22],[19,23],[21,23],[20,24],[22,24],[11,23],
[11,16],[16,24]],#, [4,14], [3,9]],
Array[Int64,1][[4,2,3,1],[2,5,6,1],[7,9,10,3,11,5,8,1],[9,
10,11,8],[4,13,14,2,16,15,6,12],[13,14,16,15],[7,4,3,12],[7,
5,6,12],[9,19,17,8],[9,13,14,8],[20,13,14,18],[10,17,8,21],
[13,10,8,15],[13,15,22,18],[23,19,17,21],[20,24,22,18],[23,
9,19,11],[9,14,16,11],[24,14,16,20],[23,10,11,21],[10,16,11,
15],[16,15,22,24]])
```

```
Plasm.view(Plasm.numbering(.25)((V, [[k] for k=1:size(V,2)],EV,FV))))
```

```
#V, (EV,FV,CV), (copEV, copFE, copCF) = Lar.chaincomplex(V,FV,EV)
```

Use the simplest data to test step-by-step the TDD



Find the first coded algorithm in error, and rewrite it
if extensive tests do not find what is wrong

Actually, in the [previous implementation](#) one did not take care of [loops in the faces](#)

Restart from pseudocode, if any !!

ALGORITHM 1: Computation of signed $[\partial_d^+]$ matrix

```

/* Pre-condition: d equal to space dimension, s.t.  $(d - 1)$ -cells are shared by two  $d$ -cells */  

/*  

Input:  $[\partial_{d-1}]$  # CSC signed matrix  

Output:  $[\partial_d^+]$  # CSC signed matrix  

 $m, n = [\partial_{d-1}].size; marks = Zeros(n)$  # initializations  

while Sum(marks) <  $2n$  do  

     $\sigma = Choose(marks)$  # select the  $(d - 1)$ -cell seed of the column extraction  

    if marks[ $\sigma$ ] == 0 then  $[c_{d-1}] = [\sigma]$   

    else if marks[ $\sigma$ ] == 1 then  $[c_{d-1}] = [-\sigma]$   

     $[c_{d-2}] = [\partial_{d-1}][c_{d-1}]$  # compute boundary  $c_{d-2}$  of seed cell  

    while  $[c_{d-2}] \neq []$  do # loop until boundary becomes empty  

        corolla = []  

        for  $\tau \in c_{d-2}$  do # for each hinge  $\tau$  cell  

             $[b_{d-1}] = [\tau]^t[\partial_{d-1}]$  # compute the  $\tau$  coboundary  

            pivot =  $\{|b_{d-1}|\} \cap \{|c_{d-1}|\}$  # compute the  $\tau$  support  

            if  $\tau > 0$  then adj = Next(pivot, Ord( $b_{d-1}$ )) # compute the new adj cell  

            else if  $\tau < 0$  then adj = Prev(pivot, Ord( $b_{d-1}$ ))  

            if  $\partial_{d-1}[\tau, adj] \neq \partial_{d-1}[\tau, pivot]$  then corolla[adj] =  $c_{d-1}[pivot]$  # orient adj  

            else corolla[adj] =  $-(c_{d-1}[pivot])$   

        end  

         $[c_{d-1}] += corolla$  # insert corolla cells in current  $c_{d-1}$   

         $[c_{d-2}] = [\partial_{d-1}][c_{d-1}]$  # compute again the boundary of  $c_{d-1}$   

    end  

    for  $\sigma \in c_{d-1}$  do marks[ $\sigma$ ] += 1 # update the counters of used cells  

     $[\partial_d^+] += [c_{d-1}]$  # append a new column to  $[\partial_d^+]$   

end  

return  $[\partial_d^+]$ 
```

Start from comments, and expand them

```
#####
# Input:[∂d-1] # Compressed Sparse Column (CSC) signed matrix (aij), where aij ∈ {-1,0,1}
# Output: [∂d+] # CSC signed matrix
#
# [∂d+] = [] ; m,n = [∂d-1].shape ; marks = Zeros(n) # initializations
# while Sum(marks) < 2n do
#     σ = Choose(marks) # select the (d - 1)-cell seed of the column extraction
#     if marks[σ] == 0 then [cd-1] = [σ]
#     else if marks[σ] == 1 then [cd-1] = [-σ]
#     [cd-2] = [∂d-1] [cd-1] # compute boundary cd-2 of seed cell
#     while [cd-2] ≠ [] do # loop until boundary becomes empty
#         corolla = []
#         for τ ∈ cd-2 do # for each "hinge" τ cell
#             [bd-1] = [τ]^t [∂d-1] # compute the τ coboundary
#             pivot = {bd-1} n {cd-1} # compute the τ support
#             if τ > 0 then adj = Next(pivot,Ord(bd-1)) # compute the new adj cell
#             else if τ < 0 then adj = Prev(pivot,Ord(bd-1))
#             if ∂d-1[τ,adj] ≠ ∂d-1[τ,pivot] then corolla[adj] = cd-1[pivot] # orient adj
#             else corolla[adj] = -(cd-1[pivot])
#         end
#         [cd-1] += corolla # insert corolla cells in current cd-1
#         [cd-2] = [∂d-1][cd-1] # compute again the boundary of cd-1
#     end
#     for σ ∈ cd-1 do marks[σ] += 1 # update the counters of used cells
#     [∂d+] += [cd-1] # append a new column to [∂d+]
# end
# return [∂d+]
#####

```

Look at the debugged code

`./algorithm-one.jl`

Look at the output: $[\partial_3^+]$, here called copFC

```
copFC = algorithm_one(V,FV,EV,copFE)
Matrix(copFC)
```

```
julia> Matrix(copFC)
22×5 Array{Int8,2}:
 1  -1   0   0   0
 1  -1   0   0   0
 -1   1   0   0   0
 0   0   1  -1   0
 1  -1   0   0   0
 0   0   1   0  -1
 -1   1   0   0   0
 -1   1   0   0   0
 1   0   0  -1   0
 0   1  -1   0   0
 -1   0   0   0   1
 -1   0   0   1   0
 0  -1   1   0   0
 1   0   0   0  -1
 -1   0   0   1   0
 -1   0   0   0   1
 -1   0   0   1   0
 0   1  -1   0   0
 -1   0   0   0   1
 -1   0   0   1   0
 0  -1   1   0   0
 -1   0   0   0   1
```

Let interpret the matrix of the boundary transformation [∂_3^+]

- The transformation $[\partial_3]$ goes from C_3 (3-chains) to C_2 (2-chains)
- incidence from **cells to faces** is **by row**
- incidence from **faces to cells** is **by columns**
- there are **five 3-cells** (**including the exterior space**) in the **arrangement** (3-space partition) induced by the **input 2-complex X** , whose 2-skeleton X_2 is given by the **set of 2-faces**.

Topological gift-wrapping algorithm

Notice that 3-cells were unknown before the algorithm execution