Geometry and Physics Modeling with Python

A. DiCarlo†  A. Paoluzzi†  G. Scorzelli‡

†University “Roma Tre“, Italy  ‡University of Utah, USA

July 3, 2010
Outline

1. Pyplasm: Plasm \(\rightarrow\) Python
   - Geometric Computing with a functional language
   - Python Embedding
   - Examples

2. Modeling with Chain Complexes
   - Cell complexes vs Chain complexes
   - The Hasse Matrix Representation

3. Chompy: Python \(\rightarrow\) Python \(\cup\) Erlang
   - Dataflow streaming of geometry
   - Distributed Computing via Message Passing

4. Towards Complex Systems Simulations
   - The ProtoPlasm framework
Motivations for a new entry

- Python: multi-paradigm language with efficient built-in data structures and simple/effective approach to OO programming.
Motivations for a new entry

- Python: **multi-paradigm language** with efficient built-in data structures and simple/effective approach to OO programming.

- Python’s elegant syntax and dynamic typing, and its interpreted nature, make it ideal for **scripting** and **RAD**
Motivations for a new entry

- Python: *multi-paradigm language* with efficient built-in data structures and simple/effective approach to OO programming.

- Python’s elegant syntax and dynamic typing, and its interpreted nature, make it ideal for *scripting* and *RAD*.

- We wished for easy access to Biopython, NumPy, SciPy, Femhub, and the geometry libraries already interfaced with Python.
Motivations for a new entry

- Python: **multi-paradigm language** with efficient built-in data structures and simple/effective approach to OO programming.

- Python’s elegant syntax and dynamic typing, and its interpreted nature, make it ideal for **scripting** and **RAD**

- We wished for easy access to **Biopython**, **NumPy**, **SciPy**, **Femhub**, and the geometry libraries already interfaced with Python

The easiest solution?

**Pyplasm: Plasm → Python**
PLaSM (Programming Language for Solid Modeling)
Geometric extension of Backus’ FL (IBM Yorktown)

(Multidimensional) Geometric Programming at Function Level

- Points, curves, surfaces, solids and higher-dim manifolds
PLaSM (Programming Language for Solid Modeling)
Geometric extension of Backus' FL (IBM Yorktown)

(Multidimensional) Geometric Programming at Function Level

- Points, curves, surfaces, solids and higher-dim manifolds
- **Domain**: Hierarchical polyhedral complexes
PLaSM (Programming Language for Solid Modeling)
Geometric extension of Backus’ FL (IBM Yorktown)

(Multidimensional) Geometric Programming at Function Level

- Points, curves, surfaces, solids and higher-dim manifolds
- **Domain**: Hierarchical polyhedral complexes
- **Representations**:
PLaSM (Programming Language for Solid Modeling)
Geometric extension of Backus’ FL (IBM Yorktown)

(Multidimensional) Geometric Programming at Function Level

- Points, curves, surfaces, solids and higher-dim manifolds
- **Domain**: Hierarchical polyhedral complexes
- **Representations**:
  - BSP trees

A. DiCarlo, A. Paoluzzi, G. Scorzelli
Geometry and Physics Modeling with Python
PLaSM (Programming Language for Solid Modeling)
Geometric extension of Backus’ FL (IBM Yorktown)

(Multidimensional) Geometric Programming at Function Level

- Points, curves, surfaces, solids and higher-dim manifolds
- **Domain**: Hierarchical polyhedral complexes
- **Representations**:
  - BSP trees
  - Polytopes
PLaSM (Programming Language for Solid Modeling)
Geometric extension of Backus’ FL (IBM Yorktown)

(Multidimensional) Geometric Programming at Function Level

- Points, curves, surfaces, solids and higher-dim manifolds
- **Domain**: Hierarchical polyhedral complexes
- **Representations**:
  - BSP trees
  - Polytopes
  - Hasse graphs
PLaSM (Programming Language for Solid Modeling)
Geometric extension of Backus’ FL (IBM Yorktown)

(Multidimensional) Geometric Programming at Function Level

- Points, curves, surfaces, solids and higher-dim manifolds
- **Domain**: Hierarchical polyhedral complexes
- **Representations**:
  - BSP trees
  - Polytopes
  - Hasse graphs

**Geometric operators:**
PLaSM (Programming Language for Solid Modeling)
Geometric extension of Backus’ FL (IBM Yorktown)

(Multidimensional) Geometric Programming at Function Level

- Points, curves, surfaces, solids and higher-dim manifolds
- Domain: Hierarchical polyhedral complexes
- Representations:
  - BSP trees
  - Polytopes
  - Hasse graphs

- Geometric operators:
  - Affine maps
PLaSM (Programming Language for Solid Modeling)
Geometric extension of Backus’ FL (IBM Yorktown)

(Multidimensional) Geometric Programming at Function Level

- Points, curves, surfaces, solids and higher-dim manifolds
- **Domain**: Hierarchical polyhedral complexes
- **Representations**:
  - BSP trees
  - Polytopes
  - Hasse graphs

**Geometric operators**:
- Affine maps
- Hierarchical structures
(Multidimensional) Geometric Programming at Function Level

- Points, curves, surfaces, solids and higher-dim manifolds
- **Domain**: Hierarchical polyhedral complexes
- **Representations**:
  - BSP trees
  - Polytopes
  - Hasse graphs

**Geometric operators**:
- Affine maps
- Hierarchical structures
- Boolean Ops

A. DiCarlo, A. Paoluzzi, G. Scorzelli

Geometry and Physics Modeling with Python
PLaSM (Programming Language for Solid Modeling)
Geometric extension of Backus’ FL (IBM Yorktown)

(Multidimensional) Geometric Programming at Function Level

- Points, curves, surfaces, solids and higher-dim manifolds
- **Domain**: Hierarchical polyhedral complexes
- **Representations**:
  - BSP trees
  - Polytopes
  - Hasse graphs

**Geometric operators**:
- Affine maps
- Hierarchical structures
- Boolean Ops
- Cartesian products
PLaSM (Programming Language for Solid Modeling)
Geometric extension of Backus’ FL (IBM Yorktown)

(Multidimensional) Geometric Programming at Function Level

- Points, curves, surfaces, solids and higher-dim manifolds
- **Domain:** Hierarchical polyhedral complexes
- **Representations:**
  - BSP trees
  - Polytopes
  - Hasse graphs

**Geometric operators:**
- Affine maps
- Hierarchical structures
- Boolean Ops
- Cartesian products
- Minkowski sums
PLaSM (Programming Language for Solid Modeling)
Geometric extension of Backus’ FL (IBM Yorktown)

(Multidimensional) Geometric Programming at Function Level

- Points, curves, surfaces, solids and higher-dim manifolds
- **Domain**: Hierarchical polyhedral complexes
- **Representations**:
  - BSP trees
  - Polytopes
  - Hasse graphs

**Geometric operators**:
- Affine maps
- Hierarchical structures
- Boolean Ops
- Cartesian products
- Minkowski sums
- Charts and atlases
PLaSM (Programming Language for Solid Modeling)
Geometric extension of Backus’ FL (IBM Yorktown)

(Multidimensional) Geometric Programming at Function Level

- Points, curves, surfaces, solids and higher-dim manifolds
- **Domain**: Hierarchical polyhedral complexes
- **Representations**:
  - BSP trees
  - Polytopes
  - Hasse graphs

**Geometric operators**:
- Affine maps
- Hierarchical structures
- Boolean Ops
- Cartesian products
- Minkowski sums
- Charts and atlases
- $d$-Skeletons, $0 \leq d \leq n$
PLaSM (Programming Language for Solid Modeling)
Geometric extension of Backus’ FL (IBM Yorktown)

(Multidimensional) Geometric Programming at Function Level

- Points, curves, surfaces, solids and higher-dim manifolds
- **Domain**: Hierarchical polyhedral complexes
- **Representations**:
  - BSP trees
  - Polytopes
  - Hasse graphs

**Geometric operators**:
- Affine maps
- Hierarchical structures
- Boolean Ops
- Cartesian products
- Minkowski sums
- Charts and atlases
- $d$-Skeletons, $0 \leq d \leq n$
- Convex hulls
PLaSM (Programming Language for Solid Modeling)
Geometric extension of Backus’ FL (IBM Yorktown)

(Multidimensional) Geometric Programming at Function Level

- Points, curves, surfaces, solids and higher-dim manifolds
- **Domain**: Hierarchical polyhedral complexes
- **Representations**:
  - BSP trees
  - Polytopes
  - Hasse graphs

**Geometric operators**:
- Affine maps
- Hierarchical structures
- Boolean Ops
- Cartesian products
- Minkowski sums
- Charts and atlases
- $d$-Skeletons, $0 \leq d \leq n$
- Convex hulls
- Domain integrals of polynomials
### Three general rules to write pyplasm code

#### Plasm primitives

**ALL CAPS**

- All capital letters
Three general rules to write pyplasm code

Plasm primitives

ALL CAPS all capital letters

Application

is always unary VIEW(CUBOID([1,4,9]))
Three general rules to write **pyplasm** code

<table>
<thead>
<tr>
<th><strong>Plasm primitives</strong></th>
<th><strong>Application</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALL CAPS</strong></td>
<td><strong>is always unary</strong></td>
</tr>
<tr>
<td>all capital letters</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Higher-level functions</strong></th>
<th><strong>Unusual but legal in Python</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>COLOR(RED)(CUBE(1))</strong></td>
<td></td>
</tr>
</tbody>
</table>
Boolean ops example: polygon filling

```
List = [
[[0,0],[4,2],[2.5,3],
 [4,5],[2,5],[0,3],
 [-3,3],[0,0]],

[[0,3],[0,1],[2,2],
 [2,4],[0,3]],

[[2,2],[1,3],[1,2],
 [2,2]]
]
```

polylines = STRUCT(ARRAY(POLYLINE)(List))
Boolean ops example: polygon filling

```python
List = [
    [[0,0],[4,2],[2.5,3],[4,5],[2,5],[0,3],[-3,3],[0,0]],
    [[0,3],[0,1],[2,2],[2,4],[0,3]],
    [[2,2],[1,3],[1,2],[2,2]]
]

polygon = SOLIDIFY(polylines)
```

A. DiCarlo, A. Paoluzzi, G. Scorzelli
Geometry and Physics Modeling with Python
Boolean ops example: polygon filling

List = [
  [[0,0],[4,2],[2.5,3],
   [4,5],[2,5],[0,3],
   [-3,3],[0,0]],

  [[0,3],[0,1],[2,2],
   [2,4],[0,3]],

  [[2,2],[1,3],[1,2],
   [2,2]]
]
Boolean ops example: polygon filling

List = 
[[0,0],[4,2],[2.5,3],
 [4,5],[2,5],[0,3],
 [-3,3],[0,0]],

[[0,3],[0,1],[2,2],
 [2,4],[0,3]],

[[2,2],[1,3],[1,2],
 [2,2]]
Boolean ops example: polygon filling

List = [
  [[0,0],[4,2],[2.5,3],
   [4,5],[2,5],[0,3],
   [-3,3],[0,0]],

  [[0,3],[0,1],[2,2],
   [2,4],[0,3]],

  [[2,2],[1,3],[1,2],
   [2,2]]
]
Coding a new pyplasm primitive

```python
from pyplasm import *

def EXPLODE (params):
    sx, sy, sz = params
    def explode0 (scene):
        centers = AA(MED([1,2,3]))(scene)
        scalings = N(len(centers))(S([1,2,3])([sx,sy,sz]))
        scaledCenters = AA(UK)(AA(APPLY)(TRANS([scalings,AA(MK)(centers)])))
        translVectors = AA(VECTDIFF)(TRANS([scaledCenters, centers]))
        translations = AA(T([1,2,3]))(translVectors)
        return STRUCT(AA(APPLY)(TRANS([translations, scene])))
    return explode0
```
The pyplasm EXPL in plain python

```python
from pyplasm import *

def EXPLODE (dims):
    dims = [dims] if ISNUM(dims) else dims

def EXPLODE0 (params):
    params = [params] if ISNUM(params) else params

def EXPLODE1 (scene):
    centers = [MED(INTSTO(RN(obj)))(obj) for obj in scene]
    scalings = len(centers) * [S(dims)(params)]
    scaledCenters = [UK(APPLY(pair)) for pair in
        zip(scalings, [MK(p) for p in centers])]
    translVectors = [ VECTDIFF((p,q)) for (p,q) in zip(scaledCenters, centers) ]
    translations = [ T(dims)(v) for v in translVectors ]
    return STRUCT([ APPLY([t,obj]) for t, obj in zip(translations,scene) ])

EXPL = EXPLODE0
```

EXPL = EXPLODE
EXPL examples

List = SPLITCELLS(SPHERE(1)([8,12]))
EXPL examples

\[ \text{List} = \text{SPLITCELLS}(\text{SPHERE}(1)([8,12])) \]

\[ \text{pol1} = \text{EXPL}([1,2,3])([1.5,1.5,1.5])(\text{List}) \]

\[ \text{pol2} = \text{EXPL}(3)(1.5)(\text{List}) \]
Minkowsky sum of cell complexes with a convex cell

\[
polList1 = \text{SPLITCELLS}(\text{SK}(1)(pol1)) \quad \text{polList2} = \text{SPLITCELLS}(\text{pol1})
\]
Minkowsky sum of cell complexes with a convex cell

\[
polList1 = SPLITCELLS(SK(1)(pol1))
\]

\[
polList2 = SPLITCELLS(pol1)
\]
def fun (poly): return COMP([ EXPL([1.5,1.5,1.5]), SPLITCELLS, 
(OFFSET([.1,.1,.1]) ])(poly)
Minkowsky sum of cell complexes with a convex cell

\[ \text{def fun (poly): return COMP([ EXPL([1.5,1.5,1.5]), SPLITCELLS, (OFFSET([.1,.1,.1]) ])(poly)} \]

fun(SPHERE(1)([8,12]))

fun(SKELETON(1)(SPHERE(1)([8,12])))
Cartesian product on cell complexes


\[ a = 10^{*}[1,-5]+[1]; \quad b = \text{MINUS}(a); \quad P = \text{PROD} \]
Cartesian product on cell complexes


\[ a = 10*[1,-5]+[1]; \quad b = \text{MINUS}(a); \quad P = \text{PROD} \]

\[
P([\text{Q}(b),\text{Q}(b)])
\]
Cartesian product on cell complexes

\[ a = 10 \cdot [1, -5] + [1]; \quad b = \text{MINUS}(a); \quad P = \text{PROD} \]

\[ P(B) \]

\[ \text{STRUCT}([ P([ Q(a), Q(a) ]), P([ Q(a), Q(b) ]), P([ Q(b), Q(a) ])) ] \]

Cartesian product on cell complexes

```python
plan = PROD([Q(10), Q([5,5])])
section = MKPOL([[0,4],[10,4],[10,7],[5,10],[0,7],[0,0],[10,0]], [[1,2,3,4,5],[1,2,6,7]], [[1],[2]])
```
Cartesian product on cell complexes

\[
\text{plan} = \text{PROD}([ \text{Q}(10), \text{Q}([5,5]) ]) \\
\text{section} = \text{MKPOL}([ \text{[[0,4],[10,4],[10,7],[5,10],[0,7],[0,0],[10,0]], [[1,2,3,4,5],[1,2,6,7]], [[1],[2]] ])
\]

\[
\text{SK}(1)(\text{plan}) \\
\text{SK}(1)(\text{section})
\]
Cartesian product on cell complexes

\[
\text{plan} = \text{PROD}([\ Q(10),\ Q([5,5])\ ])
\]
\[
[0,4],[10,4],[10,7],[5,10],[0,7],[0,0],[10,0],\ [1,2,3,4,5],[1,2,6,7],\ [1],[2]\ ]
\]

\[
\text{section} = \text{MKPOL}([\ ])
\]

**Examples**

- Cartesian product on cell complexes
- Geometric Computing with a functional language
- Python Embedding
- Towards Complex Systems Simulations
- Python → Python
- Chompy: Python → Python ∪ Erlang
- Pyplasm: Plasm → Python

A. DiCarlo, A. Paoluzzi, G. Scorzelli

Geometry and Physics Modeling with Python
Cartesian product on cell complexes
Intersection of extrusions. Special case of Cartesian product
Cartesian product on cell complexes

Intersection of extrusions. Special case of Cartesian product

\[ \text{house} = \text{GPROD}([[1,2,3],[1,3,2]])([[\text{Obj1},\text{Obj2}]]) \]

\[ \text{EXPL}([1.1,1.1,1.1])(\text{SPLITCELLS}(\text{house})) \]

A. DiCarlo, A. Paoluzzi, G. Scorzelli

Geometry and Physics Modeling with Python
Cartesian product on cell complexes

Intersection of extrusions. Special case of Cartesian product

\[
\text{house} = \text{GPROD}([[1,2,3],[1,3,2]])([[\text{Obj1}, \text{Obj2}]])
\]

\[
\text{EXPL}([1.1,1.1,1.1,1.1])(\text{SPLITCELLS(house)})
\]

\[
\text{sk1} = \text{OFFSET}([.2,.2,.4])(\text{SK(1)(house)})
\]
Scene graph of assemblies
Scene graph of assemblies

\[
\text{str1} = \text{STRUCT}(10*\text{[sk1,T(1)(11)]})
\]
Scene graph of assemblies

\[
\text{str1} = \text{STRUCT}(10*[\text{sk1,T(1)(11)])}
\]

\[
\text{str2} = \text{STRUCT}(10*[\text{str1,T(2)(31)])}
\]
A chain complex is a sequence of linear spaces of $d$-chains, $0 \leq d \leq n$, with a sequence of boundary operators, each mapping the space of $d$-chains to the space of $(d - 1)$-chains.

The dual of the chain complex is the cochain complex.

The duals of the boundary operators $\partial$ are the coboundary operator $\delta$, that map the spaces of $d$-cochains to the spaces of $(d + 1)$-cochains.
Chain complexes and Cochain complexes

- **All meshes**, say partitioning either the boundary or the interior of a model, and their associated physical fields, are properly represented by a (co)chain complex.

- Such a complex therefore gives a complete discrete representation of any type of field over any type of geometric model.

- Huge geometric structures may be properly and efficiently represented by **sparse matrices**, and therefore efficiently manipulated through linear computational algebra, in particular by using the last-generation of highly parallel vector GPUs.
Chain complexes and Cochain complexes
This representation apply to all cell complexes

- The **(co)chain representation** captures formally and unambiguously all the combinatorial relationships of abstract, geometric, and physical modelling, via the standard topological operators of boundary and coboundary.

- This representation apply to **all cell complexes**, without restriction of type, dimension, codimension, orientability, manifoldness, etc.

- Furthermore, this approach unifies the geometric and physical computation in a **common** formal computational structure.
Chains/cochains over a cell complex

A small 2-complex

Real-valued chains attach a signed $d$-measure to $d$-cells such as length to 1-cells, area to 2-cells, volume to 3-cells. They restore part of the geometrical information left out by the purely topological construction of a cell complex.
Chains/cochains over a cell complex

- Real-valued chains attach a signed $d$-measure to $d$-cells

A small 2-complex
Chains/cochains over a cell complex

- Real-valued chains attach a signed $d$-measure to $d$-cells
- such as length to 1-cells, area to 2-cells, volume to 3-cells
Chains/cochains over a cell complex

- Real-valued chains attach a signed $d$-measure to $d$-cells
- such as length to 1-cells, area to 2-cells, volume to 3-cells
- they restore part of the geometrical information left out by the purely topological construction of a cell complex
Hasse graph
Chains/cochains over a cell complex

already used in PyPlasm as the basic data structure
The Hasse matrix

A complete representation of the measured incidence between all cells of all dimensions

Discrete Physics using Metrized Chains. 2009. SIAM/ACM Conf. on Geometric and Physical Modeling

\[ \mathbf{H}(\mathbf{K}) = \begin{bmatrix} \delta^0 & \delta^1 t & \vdots & \vdots & \vdots \\ \vdots & \delta^2 & \delta^3 t & \vdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ k_0 & \vdots & \vdots & \ddots & \vdots \\ k_1 & \vdots & \vdots & \vdots & \ddots \\ \vdots & \vdots & \vdots & \vdots & \ddots \\
\end{bmatrix} \]

Block structure of the Hasse matrix

A. DiCarlo, A. Paoluzzi, G. Scorzelli

Geometry and Physics Modeling with Python
The Hasse matrix
A complete representation of the measured incidence between all cells of all dimensions

Discrete Physics using Metrized Chains. 2009. SIAM/ACM Conf. on Geometric and Physical Modeling

Block structure of the Hasse matrix

\[
H(K) = \begin{bmatrix}
[\delta^0] & [\delta^1]^t \\
[\delta^2] & [\delta^3]^t \\
\vdots & \vdots \\
k_0 & k_2 & \cdots & \cdots & \cdots & k_1 & k_3 \\
\end{bmatrix}
\]

- \( H(K) \) is the sparse bidiagonal matrix of coboundary operators
The Hasse matrix
A complete representation of the measured incidence between all cells of all dimensions

Discrete Physics using Metrized Chains. 2009. SIAM/ACM Conf. on Geometric and Physical Modeling

Block structure of the Hasse matrix

\[ H(K) = \begin{bmatrix}
[\delta^0] & [\delta^1]^t \\
[\delta^2] & [\delta^3]^t \\
\vdots & \vdots \\
\vdots & \vdots \\
\end{bmatrix} \]

- \( H(K) \) is the sparse bidiagonal matrix of coboundary operators
- \( H(K)^\top \) is the sparse bidiagonal matrix of boundary operators
The Hasse matrix
A complete representation of the measured incidence between all cells of all dimensions

Discrete Physics using Metrized Chains. 2009. *SIAM/ACM Conf. on Geometric and Physical Modeling*

Block structure of the Hasse matrix

\[
H(K) = \begin{bmatrix}
\delta^0 & \delta^1 & \cdots & \delta^d \\
\delta^1 & \delta^2 & \cdots & \delta^{d+1} \\
\vdots & \vdots & \ddots & \vdots \\
\delta^d & \delta^{d+1} & \cdots & \delta^{2d}
\end{bmatrix}
\]

- $H(K)$ is the sparse bidiagonal matrix of coboundary operators
- $H(K)^T$ is the sparse bidiagonal matrix of boundary operators
- $k_0, k_2, \ldots$ are the sizes of $d$-skeletons of even dimensions
The Hasse matrix

A complete representation of the measured incidence between all cells of all dimensions

Discrete Physics using Metrized Chains. 2009. SIAM/ACM Conf. on Geometric and Physical Modeling

Block structure of the Hasse matrix

\[ H(K) = \begin{pmatrix}
  [\delta^0] & [\delta^1]^t & \cdots \\
  [\delta^2] & [\delta^3]^t & \cdot & \cdot & \cdot \\
  \cdot & \cdot & \cdot & \cdot & \cdot \\
  \cdots & \cdots & \cdots & \cdots & \cdots \\
  k_0 & k_2 & \cdots & \cdots & k_1 \\
\end{pmatrix} \]

- \( H(K) \) is the sparse bidiagonal matrix of coboundary operators
- \( H(K)^\top \) is the sparse bidiagonal matrix of boundary operators
- \( k_0, k_2, \ldots \) are the sizes of \( d \)-skeletons of even dimensions
- \( k_1, k_3, \ldots \) are the sizes of \( d \)-skeletons of odd dimensions
Chompy: to compute with (co)chain complexes using multi-paradigm and concurrent computer languages (Python and Erlang, respectively).

- linear and higher order— dimension-independent simplicial complexes
Chompy: to compute with (co)chain complexes using multi-paradigm and concurrent computer languages (Python and Erlang, respectively).

- linear and higher order—dimension-independent simplicial complexes
- $d$-complexes of convex cells
Chompy: to compute with (co)chain complexes using multi-paradigm and concurrent computer languages (Python and Erlang, respectively).

- linear and higher order—dimension-independent simplicial complexes
- $d$-complexes of convex cells
- Cartesian product of cell complexes
Chompy: to compute with (co)chain complexes using multi-paradigm and concurrent computer languages (Python and Erlang, respectively).

- linear and higher order—dimension-independent simplicial complexes
- $d$-complexes of convex cells
- Cartesian product of cell complexes
- skeleton and boundary extraction
Chompy: to compute with (co)chain complexes using multi-paradigm and concurrent computer languages (Python and Erlang, respectively).

- linear and higher order—dimension-independent simplicial complexes
- $d$-complexes of convex cells
- Cartesian product of cell complexes
- skeleton and boundary extraction
- various types of local and global cell and (co)chain refinement
**Chompy**: to compute with (co)chain complexes using multi-paradigm and concurrent computer languages (Python and Erlang, respectively).

- linear and higher order— dimension-independent simplicial complexes
- \(d\)-complexes of convex cells
- Cartesian product of cell complexes
- skeleton and boundary extraction
- various types of local and global cell and (co)chain refinement
- finite integration of polynomials over subcomplexes
Chompy: to compute with (co)chain complexes using multi-paradigm and concurrent computer languages (Python and Erlang, respectively).

- linear and higher order— dimension-independent simplicial complexes
- $d$-complexes of convex cells
- Cartesian product of cell complexes
- skeleton and boundary extraction
- various types of local and global cell and (co)chain refinement
- finite integration of polynomials over subcomplexes
- and so on ...
Erlang language
Concurrent processing done right

- Powerful (open source and multi-platform) concurrent functional programming language and runtime system
Erlang language
Concurrent processing done right

- Powerful (open source and multi-platform) concurrent functional programming language and runtime system
- Purely functional (single assignment, dynamic typing), easy to understand and to debug
Erlang language
Concurrent processing done right

- Powerful (open source and multi-platform) concurrent functional programming language and runtime system
- Purely functional (single assignment, dynamic typing), easy to understand and to debug
- Fits well with multicores CPUs, clusters and SMP architectures
Erlang language
Concurrent processing done right

- Powerful (open source and multi-platform) concurrent functional programming language and runtime system
- Purely functional (single assignment, dynamic typing), easy to understand and to debug
- Fits well with multicore CPUs, clusters and SMP architectures
- Even hot swapping of programs is supported — code can be changed without stopping a system
Erlang language
Concurrent processing done right

- Powerful (open source and multi-platform) concurrent functional programming language and runtime system
- Purely functional (single assignment, dynamic typing), easy to understand and to debug
- Fits well with multicore CPUs, clusters and SMP architectures
- Even hot swapping of programs is supported — code can be changed without stopping a system
- Developed by Ericsson to support distributed, fault-tolerant, soft real-time, non-stop applications
Disco: Erlang \cup Python

Distributed computing framework developed by Nokia Research Center to solve real problems in handling massive amounts of data.

- Disco users start Disco jobs in **Python scripts**.
Disco: Erlang ∪ Python

Distributed computing framework developed by Nokia Research Center to solve real problems in handling massive amounts of data

- Disco users start Disco jobs in Python scripts.
- Jobs requests are sent over HTTP to the master.
Disco: Erlang ∪ Python

Distributed computing framework developed by Nokia Research Center to solve real problems in handling massive amounts of data

- Disco users start Disco jobs in Python scripts.
- Jobs requests are sent over HTTP to the master.
- Master is an Erlang process that receives requests over HTTP.
Disco: Erlang ∪ Python

Distributed computing framework developed by Nokia Research Center to solve real problems in handling massive amounts of data

- Disco users start Disco jobs in Python scripts.
- Jobs requests are sent over HTTP to the master.
- Master is an Erlang process that receives requests over HTTP.
- Master launches another Erlang process, worker supervisor, on each node over SSH.
**Disco: Erlang ∪ Python**

Distributed computing framework developed by Nokia Research Center to solve real problems in handling massive amounts of data.

- Disco users start Disco jobs in **Python scripts**.
- Jobs requests are sent **over HTTP** to the master.
- Master is an **Erlang process** that receives requests over HTTP.
- Master launches another Erlang process, worker supervisor, **on each node** over SSH.
- Worker supervisors **run Disco jobs** as Python processes.
Progressive BSP (Binary Space Partition) tree
Back to Chompy dataflow streaming

3 types of geometry nodes: **FULL**, **EMPTY** and **FUZZY** cells

![Diagram of three types of geometry nodes: FULL, EMPTY, and FUZZY cells.](image)
Progressive BSP (Binary Space Partition) tree
Back to Chompy dataflow streaming

3 types of geometry nodes: **FULL**, **EMPTY** and **FUZZY** cells

- The **FUZZY** cells, to be split at next step, are in light gray
Progressive BSP (Binary Space Partition) tree
Back to Chompy dataflow streaming

3 types of geometry nodes: FULL, EMPTY and FUZZY cells

- The FUZZY cells, to be split at next step, are in light gray
- The FULL cells are in dark gray
Progressive BSP (Binary Space Partition) tree
Back to Chompy dataflow streaming

3 types of geometry nodes: FULL, EMPTY and FUZZY cells

- The FUZZY cells, to be split at next step, are in light gray
- The FULL cells are in dark gray
- The EMPTY cells are not shown (of course :o)
Dataflow graph of the generating expression
Dataflow graph of the generating expression

- Dataflow graph of the **pyplasm expression** that produces the mechanical piece
Dataflow graph of the generating expression

- Dataflow graph of the **pyplasm expression** that produces the mechanical piece
- The dataflow generation is from **source preprocessing**
Dataflow graph of the generating expression

- Dataflow graph of the *pyplasm expression* that produces the mechanical piece
- The dataflow generation is from *source preprocessing*
- The various processes will run *concurrently* in an Erlang environment
Progressive BSP: generation of the 2-circle
Splitting of both (a) the model and (b) the computation
Progressive BSP: generation of the 2-circle
Splitting of both (a) the model and (b) the computation

- Dataflow refinement based on progressive splits of convex cells with **BSP tree nodes** (hyperplanes)
Progressive BSP: generation of the 2-circle

Splitting of both (a) the model and (b) the computation

- Dataflow refinement based on progressive splits of convex cells with **BSP tree nodes** (hyperplanes)
- Model partition induced by the BSP subtree closest to the root, to be detailed independently on different **computational nodes**
Progressive BSP: biquadratic rational B-spline
Progressive BSP: biquadratic rational B-spline

- each refinement is generated by splitting and is contained within the previous cell
Progressive BSP: biquadratic rational B-spline

- each refinement is generated by splitting and is contained within the previous cell
- In this case (approximation of a surface with a solid mesh) all of the cells are either EMPTY or FUZZY, i.e. there are no solid cells
Progressive BSP: the Leaning tower of Pisa

A. DiCarlo, A. Paoluzzi, G. Scorzelli
Progressive BSP: the Leaning tower of Pisa

- Dataflow refinement of convex cells with **BSP tree nodes** (hyperplanes)
Progressive BSP: the Leaning tower of Pisa

- Dataflow refinement of convex cells with **BSP tree nodes** (hyperplanes)
- Model partition induced by the BSP subtree closest to the root, detailed on different **computational nodes**
Thanks for your attention !!