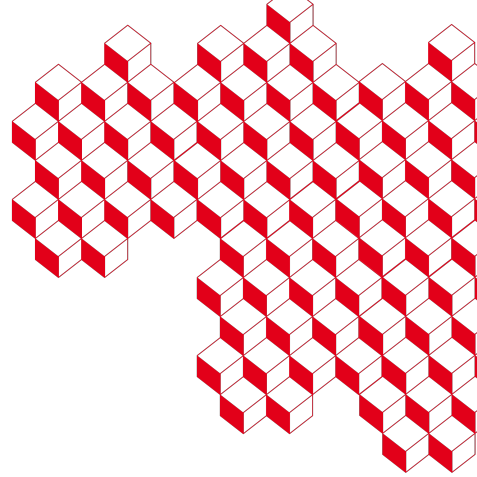
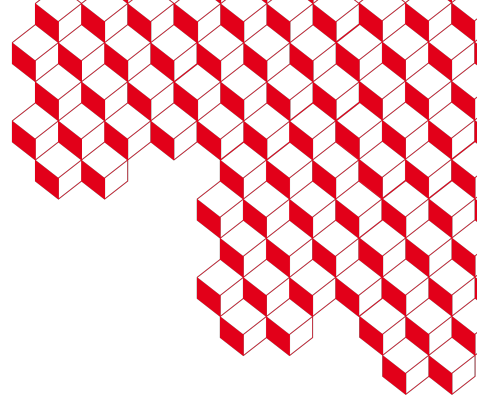


MEDCoupling Introduction and overview

08-03-2024

Aymeric SONOLET, Guillaume BROOKING



MEDCoupling Introduction and overview

08-03-2024

Aymeric SONOLET, Guillaume BROOKING

Outline

Agenda

- Overview of MEDCoupling
- Notebooks `1` and `2` about `DataArray`, `Mesh` and `Field`
- Presentation of `MEDLoader`
- Notebooks `3`, `4`, and `5` about `MEDLoader`
- Presentation of `Remapper`
- Notebook `6` about `Remapper`
- Notebooks of `ExempleComplet/`

Objectives

- Get an overview of the library features
- Understand of main objects: `DataArray`, `Mesh`, `Field`
- Master simple and representative cases

Outline

What is MEDCoupling ?

A C++ library, for manipulating Meshes and Fields

*with almost everything available through a **Python wrapping***

Mesh and Fields

A typical scientific simulation requires:

- **Meshes**, spatial discretization of a geometric domain
- **Fields**, physical data on mesh entities

which are the main input/output of a simulation code *(spatial descr., initial conditions, boundary conditions, etc.)*

Operations on meshes and fields can be complex

Why using MEDCoupling ?

Custom Meshes and Fields for each code

- well suited
- hard to develop
- hard to interact with other codes and tools (e.g. post-processing)

Using MEDCoupling in several codes

- may have limitations
- inherit from all MEDCoupling features
- share development effort
- easy to interact with other codes and tools

Differences with SMESH

- field manipulation
- highly scriptable creation of simple meshes
- an **interface** between simulation codes and tools

MEDCoupling Features

Mesh

- Build from scratch
- Refine and split
- Intersect

Fields

- Projection from one mesh to another
- Projection taking into account physical nature of the field
- Parallel projections

Misc. features

- Convex hull computation
- Duplicate nodes identification
- Degenerated cells reduction, typically a flat triangle
- Point localization
- Algebraic volume, area, length computation
- Mesh concatenation
- Eigen values computation
- Gauss points management
- . . . (many more)

Historical context

1996

ÉdF R&D, data exchange between simulation codes standardization effort

2001

`MED-file` library (ÉdF/CEA), for data exchange, integrated to the SALOME platform

2010

First version of MEDCoupling
Almost 15 years of development

What does `MED` means, in “MEDCoupling” ?

Modèle d’Échange de Données, various projects

Possible confusion with other related (but distinct) projects:

- the `MED` file format (`.med`)
- the `MED-file` library (`med-file/`, `med/`)
- the `MED` GUI module of Salome

More information *here*

Structure of MEDCoupling

Core structures and functions

- `DataArray`, `Mesh`, algorithms, etc.

Projections

- `Interpolation` and field projection

File I/O

- `MEDReader`

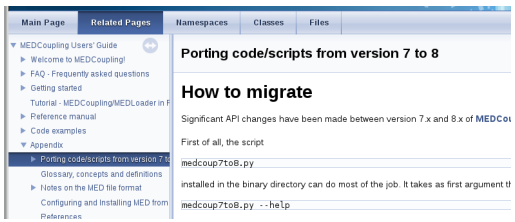
Parallelism

- `DEC`, Data Exchange Channel

Documentation

Online (clickable links)

- User documentation
- FAQ
- Examples
- Reference manual



Directly from within Python

```
help(medcoupling.DataArrayDouble)
```

Outline

Description

Fundamental role of arrays

- nodes coordinates (`floats`)
- nodes indices and cell connectivities (`integers`)
- data of fields (`floats` or `integers`)

DataArray

- Similar to a `NumPy` array, of `integers` or `floats`
- Usually, much more rows than columns
- Indices start at zero
- Important point for I/O: DataArray-s have a name, and so have their components

Example

DataArray of 2D points

- a `DataArrayDouble` (`floats`)

```
mc.DataArrayDouble([(1.0, 2.0),
                    (2.0, 3.5),
                    (1.5, 3.4)])
```

- 3 tuples of 2 components each

DataArray operations

Standard operators

- `da = mc.DataArrayInt([1,2,3])`
- `da *= 2`
- `da = (1-da)`

Similarity with NumPy

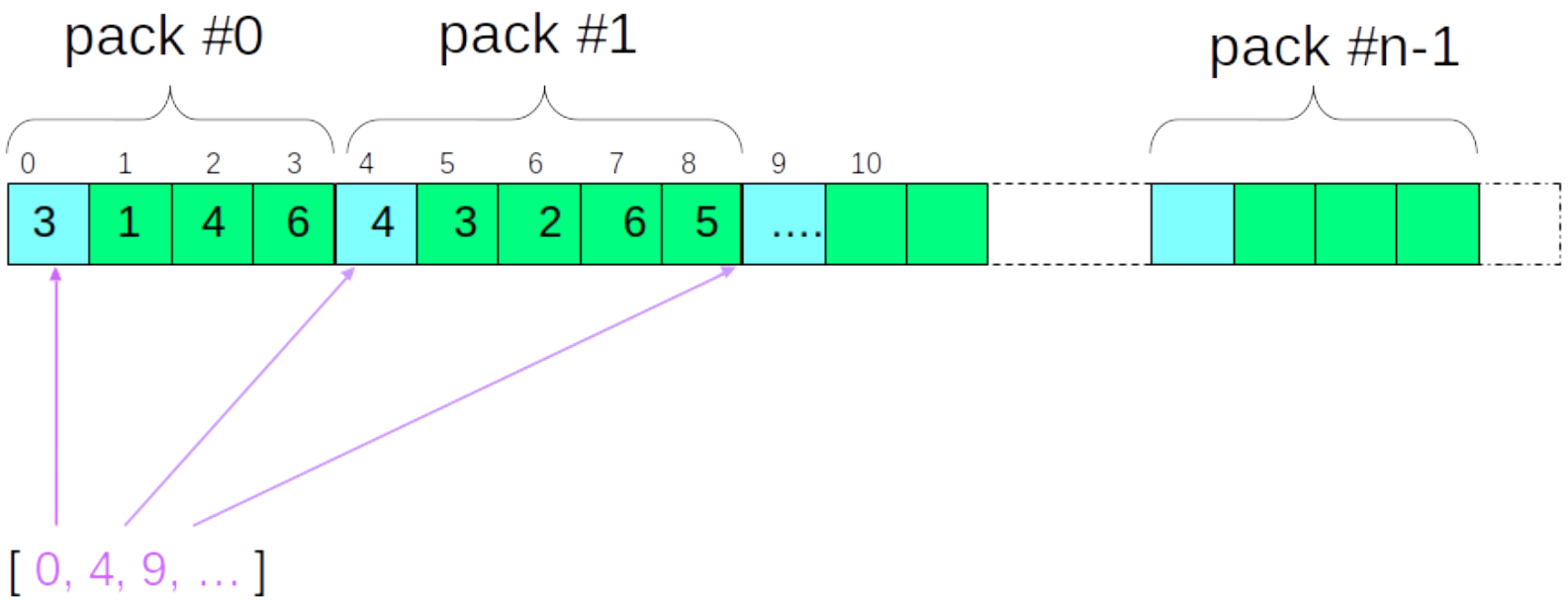
- by default, component-wise operations
- slicing: `da[1:]`
- `DataArrayInt` and `DataArrayDouble` can be converted to `NumPy` arrays
- Link with `SciPy` for linear algebra

Specificities

- Advanced operations: intersection (for `DataArrayInt`), `min`/`max` extraction, etc.
- Usage with arrays of booleans (`where()` clauses)
- Syntactic sugar for dimension management (`translate`, `newaxis`, etc.)

Indirect indexing, a typical way to store indices

Indirect indexing format



Notably, used for representation of cells' connectivity

Outline

Description

A Mesh is (in MEDCoupling)

- the spatial discretization of a continuous geometrical domain
- made of **cells**: segments in 1D, surfaces in 2D, and volumes in 3D.
- `structured`, `unstructured`, `extruded`, etc.

Properties of a Mesh

- a **unique array of point coordinates** (`DataArrayDouble` of “nodes”)
- a name (important for I/O), and a time-step identifier
- **a unique mesh dimension**

You can **NOT** mix a mesh representing 3D volumes (e.g. cubes) with 2D areas (e.g. triangles)

Description

Do not confuse

- the mesh dimension: dimension of the cells
- the spatial dimension: number of coordinate of each node

Example

An helix-shaped curve has:

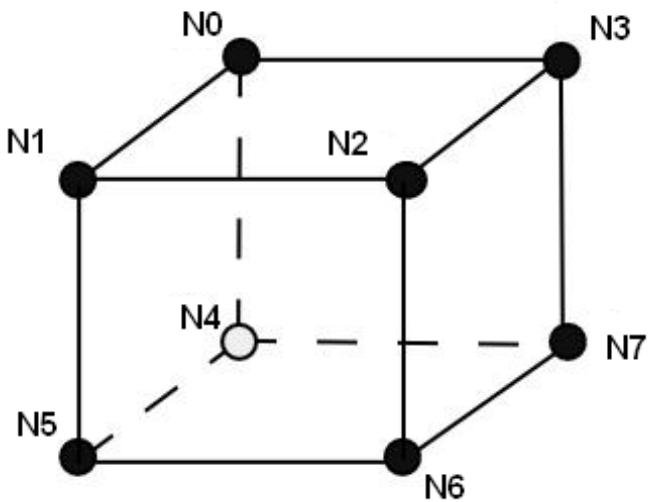
- a mesh dimension of `1` (segments)
- a spatial dimension of `3` (points have 3 coordinates)

A curved surface has:

- a mesh dimension of `2` (e.g. triangles)
- a space dimension of `3`

Cell representation (unstructured meshes)

- A cell is described by the list of its **nodes' indices** (not coordinates)
  - e.g. `[0, 1, 2, 3, 4, 5, 6, 7]`
  - "cell" can be a segment in 1D
- Need for convention: there is more than one way to index a cube's vertices:
- No explicit notion of "edges" (resp. "faces") for a 2D (resp. 3D) cell
- Only points. Other entities can be computed on the spot, with `buildDescendingConnectivity()`



About cells

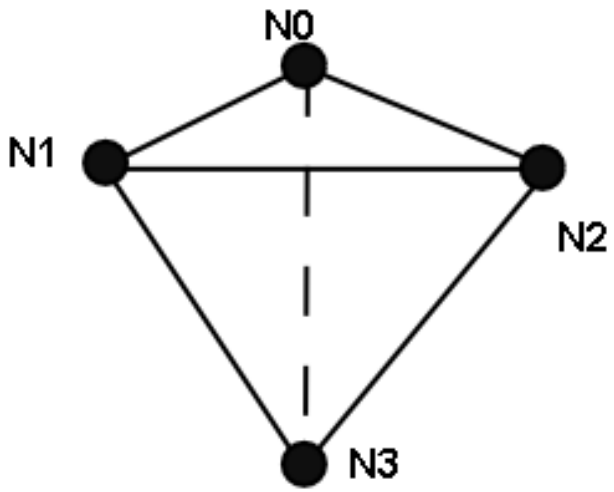
Label of cell types

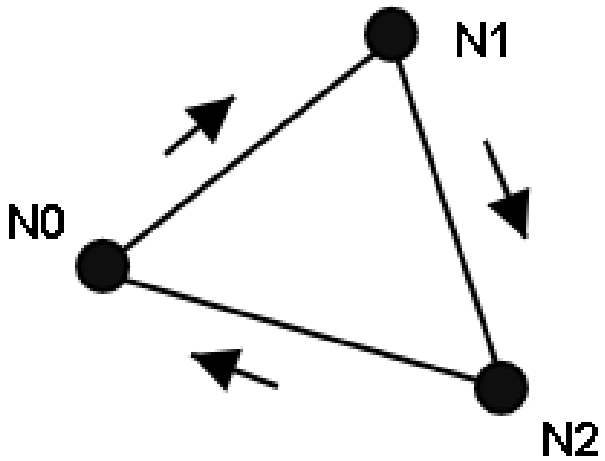
- `HEXA8`: a hexahedron with 8 nodes (i.e. a linear cube)
- `HEXA20`: g. hexahedron with 20 nodes (can represent a “quadratic” element, a “cube with curved faces”).
- `TETRA4`, `TETRA10`, etc.

See the `MED-file` documentation, for more information.

Numbering order

- In 2D, reverse trigonometric convention is used

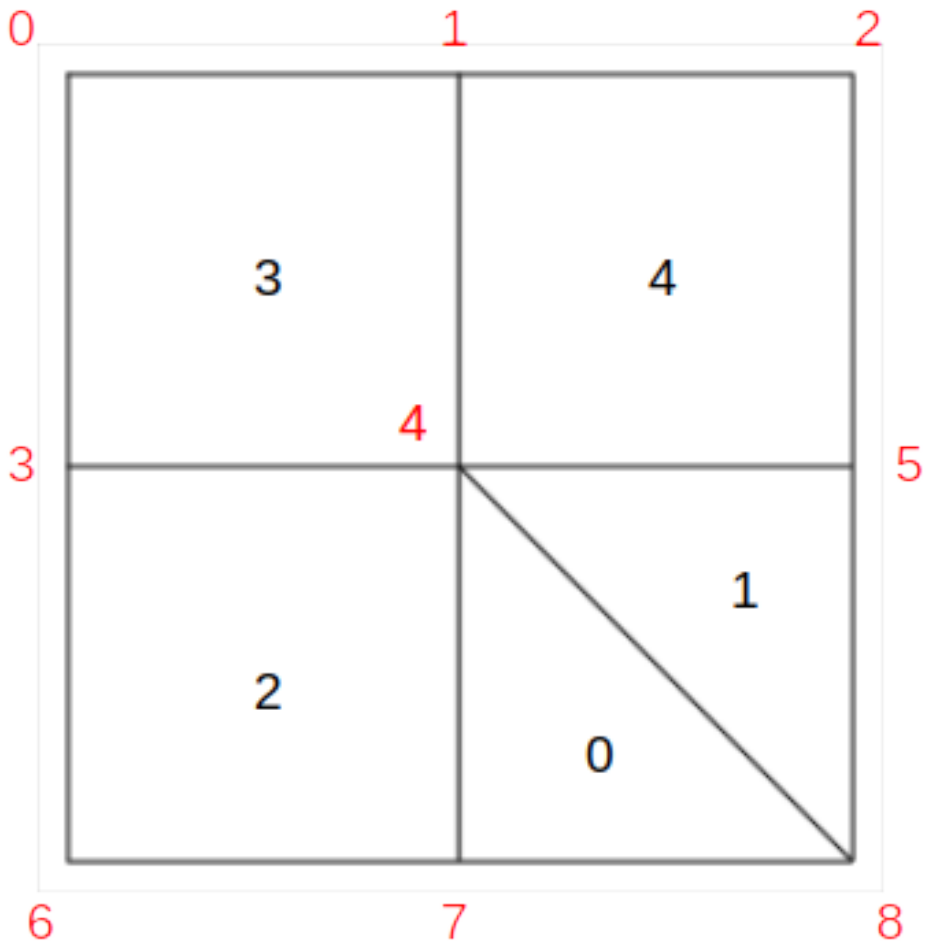




Example

Unstructured 2D mesh

- composed of `QUAD4` and `TRI3`
- The cell `2` has the connectivity `[3, 4, 7, 6]`

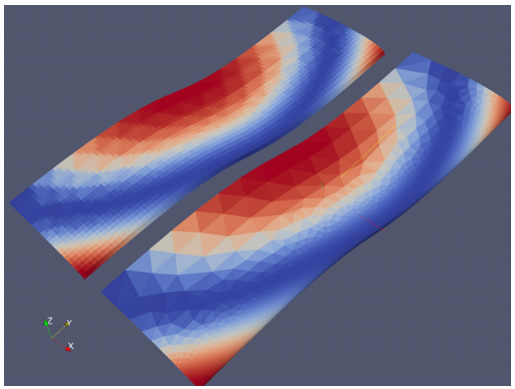


Outline

Description

A Field is

- the representation of a multi-component quantity, associated with entities of a spatial domain
- at a low level, the association of a `DataArray` and a `Mesh`
- defined `ON_NODES`, `ON_CELLS`, or on more complex items
- assigned a temporal discretization (`NO_TIME`, `ONE_TIME`, etc.)
- assigned an `extensive` or `intensive` physical nature (optional, for interpolation)

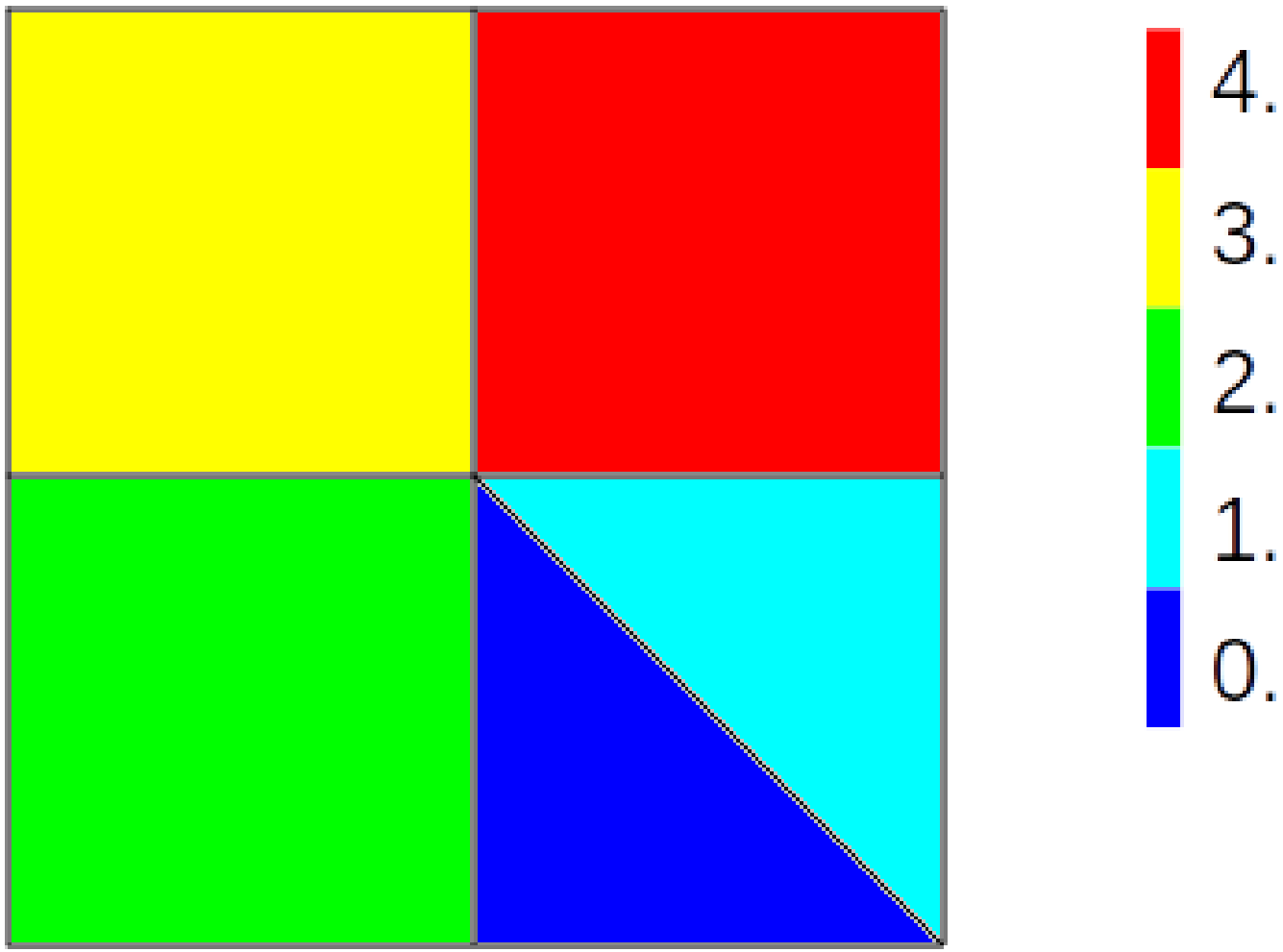


For instance

- Magnetic field (Bx, By, Bz)
- Temperature field (T)

Example

- An `ON_CELLS` Field



Outline

Note for C++ developers

source code

- 150k lines of C++
- 40k lines of Python

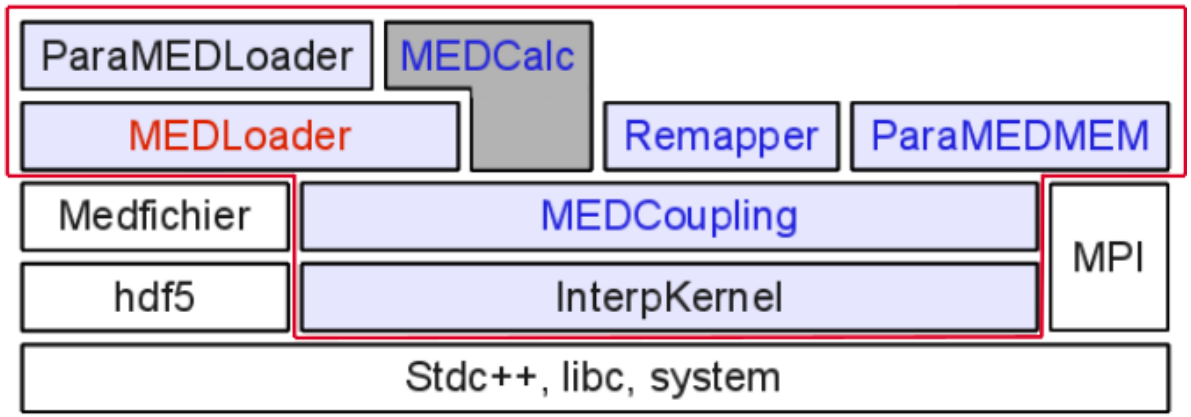
build

- `cmake`

tests

- ~1600 unit tests
- memory leak check for each unit tests (`valgrind`)

Dependency Structure



- Several sub-parts each dedicated to a specific task
  - `MEDCoupling`: memory model and general processing
  - `MEDLoader`: persistence
  - `ParaMEDMEM`: parallelism
- A big effort to have little dependencies
- Parts are:
  - Swigged
  - Swigged and wrapped with `CORBA`
  - System dependencies

Base classes

`RefCountObject` abstract class

- Similarities with VTK code structure
  - Ease the interaction with VTK (ParaView plugins)
  - Historically, there's been a reflection about using VTK directly
- All (significant) MEDCoupling classes inherit from
  - RefCountObject
    - A pointer and a counter
    - Memory management philosophy: someone owns the object after its creation
    - incrRef() to take ownership of the object / decrRef() to release it
    - Template class MCAuto is here to help
    - Smart pointer behavior (no Boost dependency)
    - No need to decrRef() if used
    - valgrind is your friend
  - Careful, some MEDCoupling functions
    - give you pointer ownership: e.g. mergeNodes()
    - Some don't: getCoords()