# 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

## Mesh and Fields

A typical scientific simulation requires:

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

main input/output of a numerical solver: *(spatial descr., initial conditions, boundary conditions, etc.)*

**Creating or modifying meshes and fields is difficult**

## 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

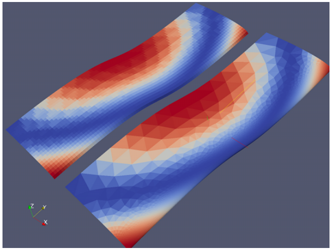
- may have limitations
- inherit from all MEDCoupling features
- share development effort
- easy to interact with other codes and tools

### 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
- Almost everything available in **Python**



## 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
- And many more

## What does MED means ?

### Modèle d'Échange de Données

- Core structures (arrays, meshes and algorithms)
- Projection (interpolation and field projection)
- Parallelism (DEC - Data Exchange Channel)
- File I/O (MEDFiles)
- MEDReader, Paraview plugin to visualize MEDFiles
- MED GUI (aka Fields)

## 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**

## Outline

# DataArrays (1/2)

- To do all this, surely, we need
  - Arrays of points (mesh nodes coordinates, centers of mass, . . . )
  - Arrays of “data” (fields, etc . . . )
- First building block: DataArray
  - “Similar” to a list with several columns
    - Usually much more rows than columns!
  - Example: array of 2D points = DataArrayDouble
    - coo = mc.DataArrayDouble([(1.0,2.0), (2.0,3.5), (1.5,3.4)])
    - Array of 3 tuples and 2 components
- Main sub-types: DataArray’s of int or double
- All indices start at zero
- In Python, DataArray-s are manipulated very much like NumPy arrays
- Important point for I/O: DataArray-s have a name, and so have their components

## DataArrays (2/2)

- All standard operators directly accessible using operators:
  - `da = mc.DataArrayInt([1,2,3])`
  - `da *= 2`
  - `da = (1-da)`
  - Similarity with NumPy: by default, operations are done at the tuple level (no matrix-like operations)
- Advanced operations: intersection (for DAInt), min/max extraction, . . .
- Link with NumPy (need to have a MED compiled with NumPy support!)
  - `DataArrayInt` and `DataArrayDouble` can be converted to NumPy array
    - Efficient in both ways (no copy, just ownership transfer)
  - Advantages
    - mmap, serialize/deserialize with multiprocessing
    - Usage with arrays of bool ( where() clauses )
    - Syntactic sugar for dimension management (translate, newaxis, . . . )
    - Link with SciPy for linear algebra
    - Link with pyCUDA
  - Restrictions
    - Some services missing: `getMinMaxPerComponent()`, `findCommonTuples()`, . . . or to be invoked with another approach
    - No names associated to arrays/components

## Renumbering, Easy index manipulation

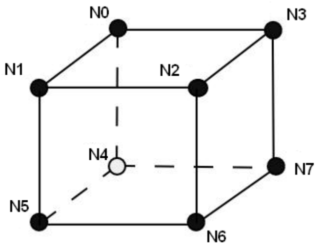
- indexing methods like `numpy`:
  - `da[:, 0]` : extract the first component of all tuples (i.e. first column of the array)
  - `da[[1,0,2]]` : reorganize the array
- Many MEDCoupling functions work with (or return) arrays in the format above.
- See `Array Renumbering` section of the documentation.

## Meshes

- In the MED world, a mesh is
  - The spatial discretization of a continuous geometrical domain
  - Associated to only one array of underlying point coordinates (a DataArrayDouble, called the nodes of the mesh)
  - With only one single mesh dimension
    - You can NOT mix a mesh representing 3D volumes (say cubes) with 2D areas (triangles)
  - A mesh also has a name (important for I/O) and a time-step ID
- Do not confuse
  - Mesh dimension: dimension of the cells
  - Spatial dimension: number of components in the array of coordinates
  - Example: an helix-shaped curve (= a wire shaped like a corkscrew) has
    - Mesh dim = 1 (this is a simple wire made of segment cells)
    - Spatial dimension = 3
  - A curved surface has mesh dimension = 2, space dimension = 3
- A mesh is made of cells: segments in 1D, surfaces in 2D, volumes in 3D.
- Main types of mesh in MEDCoupling: structured, unstructured and extruded

# Cell representation (unsctructured 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

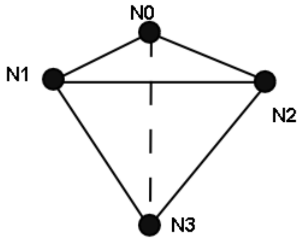
### Label of cell types

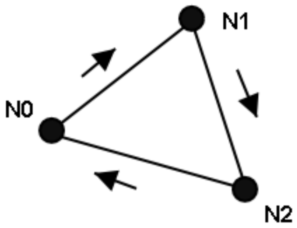
- `HEXA8`: a hexahedron with 8 ponts (i.e. a linear cube)
- `HEXA20`: g. hexahedron with 20 points points (can represent a “quadratic” element, a “cube with curved faces”).
- `TETRA4`, `TETRA10`, etc.

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

### Numbering order

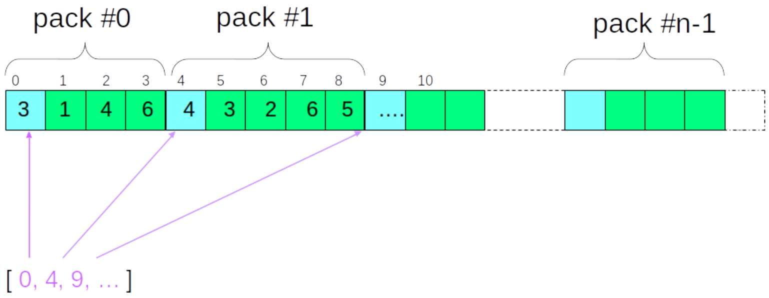
- In 2D, reverse trigonometric convention is used





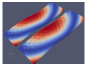
## Indirect indexing, a typical way to store indices

### Indirect indexing format



Notably, used for representation of cells' connectivity

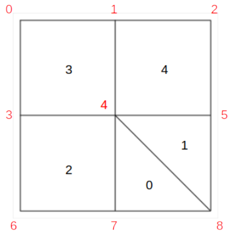
# Fields, more than an array of values



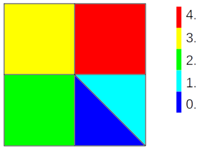
- represents some physical quantity associated with a spatial domain
  - No continuous description of the physical quantity over the domain
  - A finite set of values, associated to some constituents of the mesh
  - At a low level, a DataArray associated to a given mesh
    - Can have more than one component!
- A field can be supported by nodes (vertices) ON_NODES, cells (elements) of the mesh ON_CELLS, or more complex items
- A field has a temporal discretization
  - NO_TIME
  - ONE_TIME
  - CONST_ON_TIME_INTERVAL
- For some operations (interpolation), one need to define the physical nature of the field (extensive field or intensive field)
- More on all this with interpolation . . .
- Examples
  - Magnetic field: tensor field of double values: 3 components (Bx, By, Bz)
  - Temperature field: scalar field of double values: 1 component

# Illustration

- Unstructured 2D mesh, of `QUAD4` and `TRI3`
- The cell `2` has the connectivity `[3, 4, 7, 6]` (clockwise)

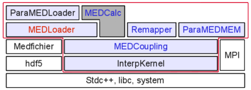


- A P0 (i.e. `ON_CELLS` field on this mesh), with 5 values



## Outline

## 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

## Outline

## Note for C++ developers

### A few words about the code

- source code
  - 150k lines of C++
  - 40k lines of Python
- tests
  - ~1600 unit tests
  - memory leak check for each unit tests (`valgrind`)
- build
  - `cmake`

## 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()