EFDA

EUROPEAN FUSION DEVELOPMENT AGREEMENT

Task Force
INTEGRATED TOKAMAK MODELLING

ITM Training Session, March 2012
IPP Garching

The ITM General Grid Description - Tutorial

H.-J. Klingshirn

TF Leader : G. Falchetto,
Deputies: R. Coelho, D. Coster

EFDA CSU Contact Person: D. Kalupin

CPOs and discretizations

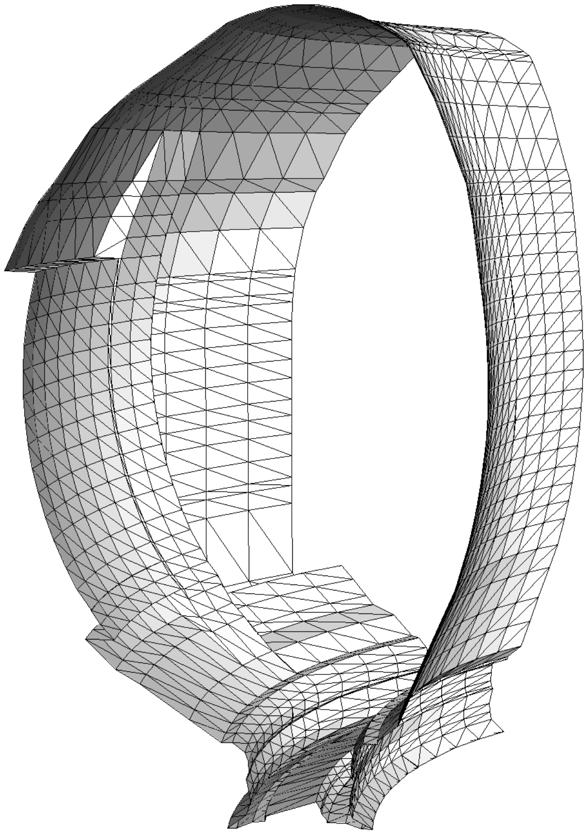
- “Consistent physical object”: one CPO per physics problem
– but many codes with different numerics / discretizations

- In most CPOs: specific discretizations explicitly assumed in the CPO design. Examples:
  - Core CPOs (coreprof/coretransp/coresource): 1d radial grid
  - Equilibrium CPO: choice of rectangular or triangular 2d grids

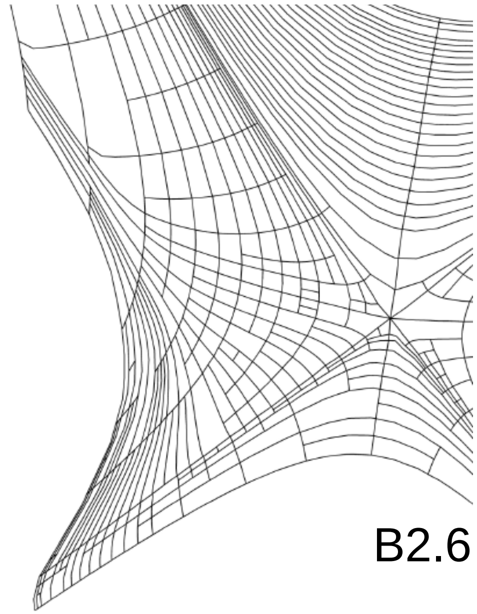
- Can't support every possible discretization explicitly in the CPOs

- Can't (and don't want to!) force use of a given discretization, especially for problems with complex geometry

Complex discretizations

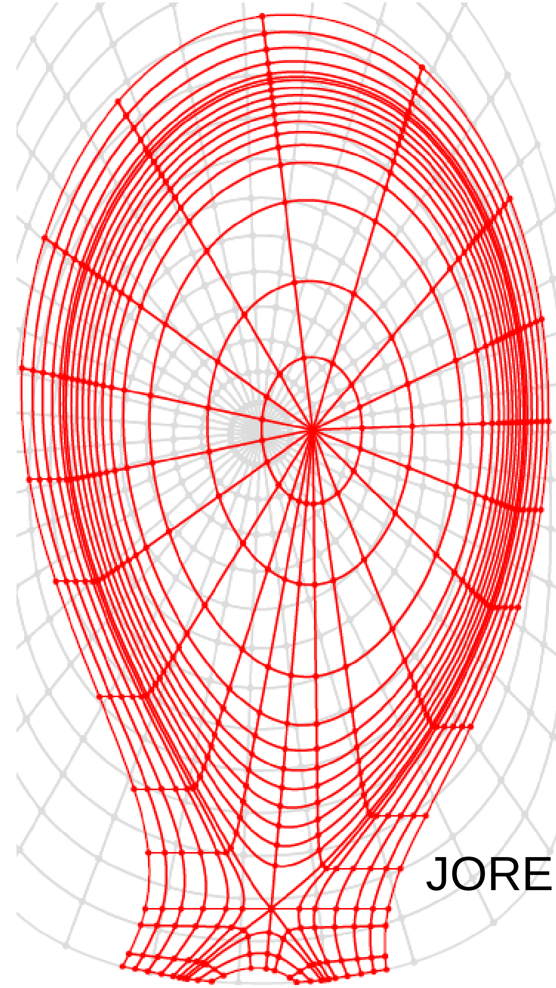


…realistic device geometry…



B2.6

…unstructured grids…



JOREK

…complex geometry representations…

General grid description

Approach: separation of concerns

- The CPOs define **what** quantities are stored (physics)
- **How** they are stored (numerics) is up to the code

The ITM General Grid Description makes this separation of physics and numerics possible:

- It provides a data structure that can efficiently store a wide range of different discretizations with arbitrary dimensions
- By designing CPOs with this data structure, the choice of discretization is deferred to the code

ITM General Grid Description: central components

1. A standardized **method & conventions** how to define and write down the details of a spatial discretization

2. Standardized **data structures**, designed to be part of CPOs, for
   - the grid itself
   - data stored on the grid

1. A dedicated **software library** (the **Grid Service Library**) helping codes to read, write and interpret grids and data

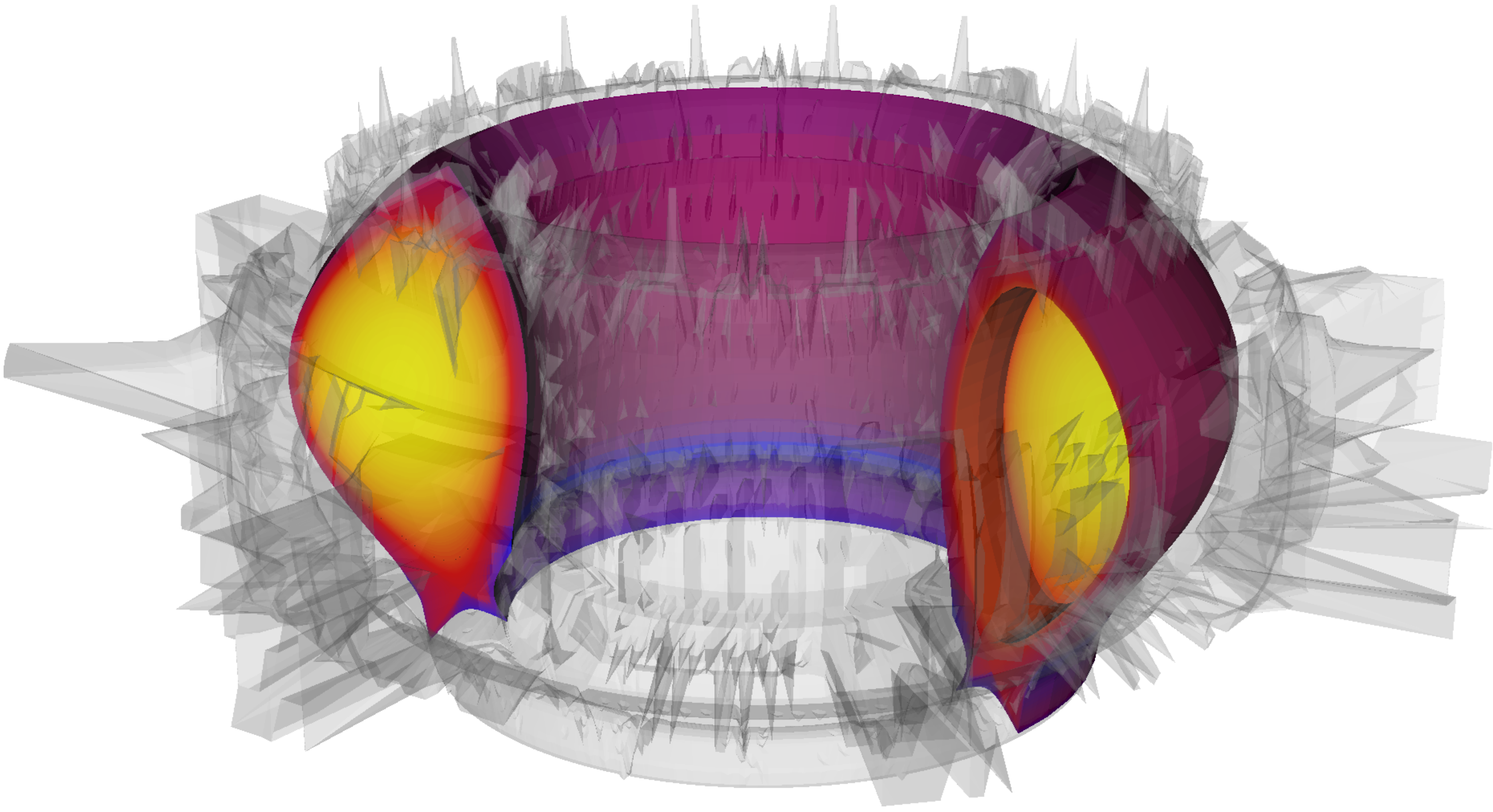
Does this work? Yes :)

- ✓ Supports **wide range of grids** in arbitrary dimensions
- ✓ Large high-dimensional grids & data sets possible with **very little overhead**
- ✓ **Extensible** to support complex geometry and data representations
- ✓ “Simple things are simple, complex things are possible”
- ✓ Allows us to develop general tools for data manipulation and visualization

…but obviously it’s not the silver bullet for everything.

...and there is still the need to defined standard discretizations or conversion procedures to make actors interchangeable

…all possible with general tools.



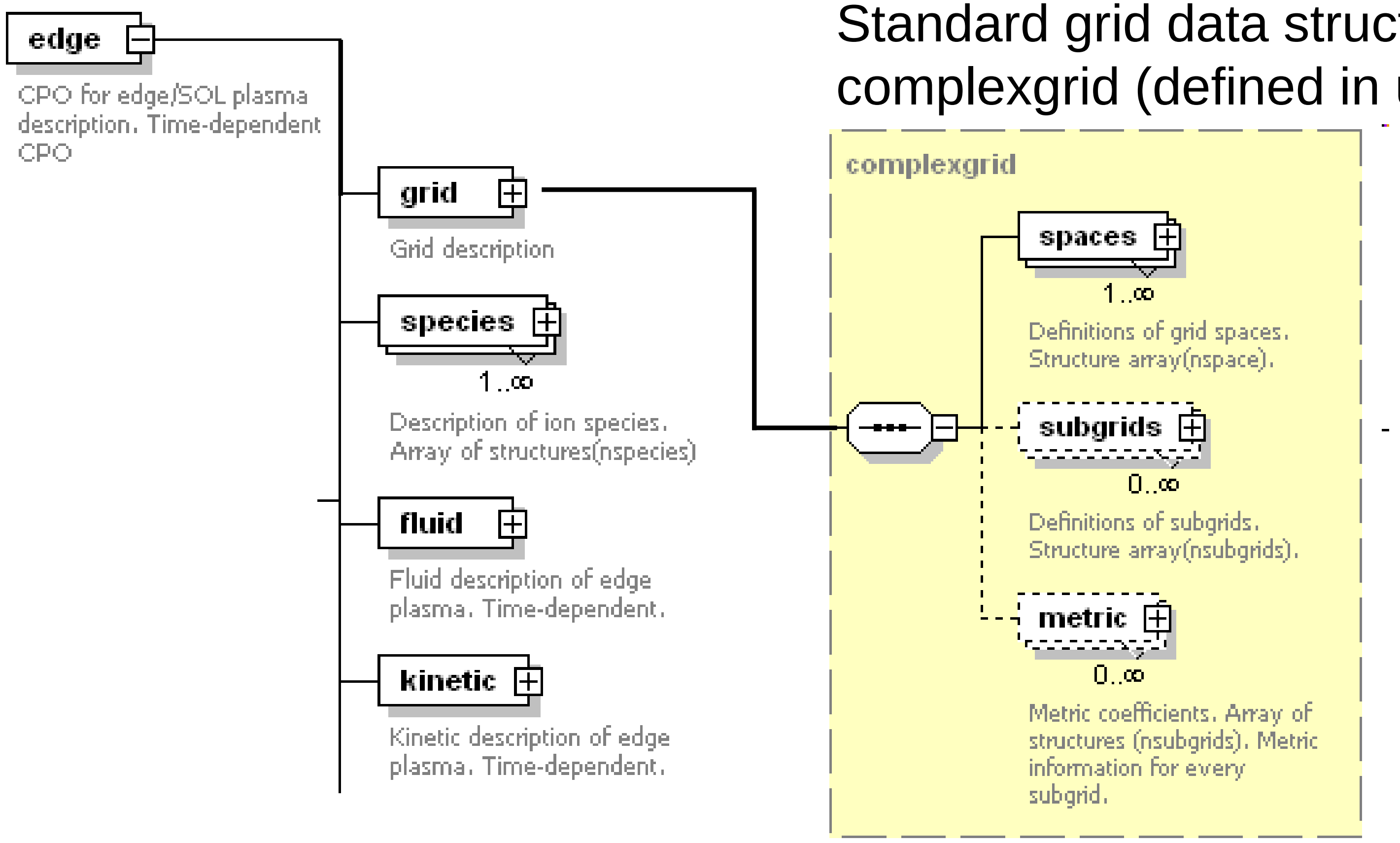
The user perspective

What does it mean for the user?

- **CPO Designer**
  – CPOs have to be defined using the GGD data types
- **Code developer**
  – When writing to the CPO, the grid has to be defined explicitly
  – A code reading the CPO has to interpret the given grid
    - Which can also mean to indicate “can’t handle this”
  – Code coupling still requires careful thought:
    - define standard discretization(s) for specific coupling scenarios, or
    - write general coupling code that can handle different discretizations
  – There is the “Grid Service Library” to help you with all this
- **Code user**
  – General purpose tools can be used to work with CPOs using the General Grid Description

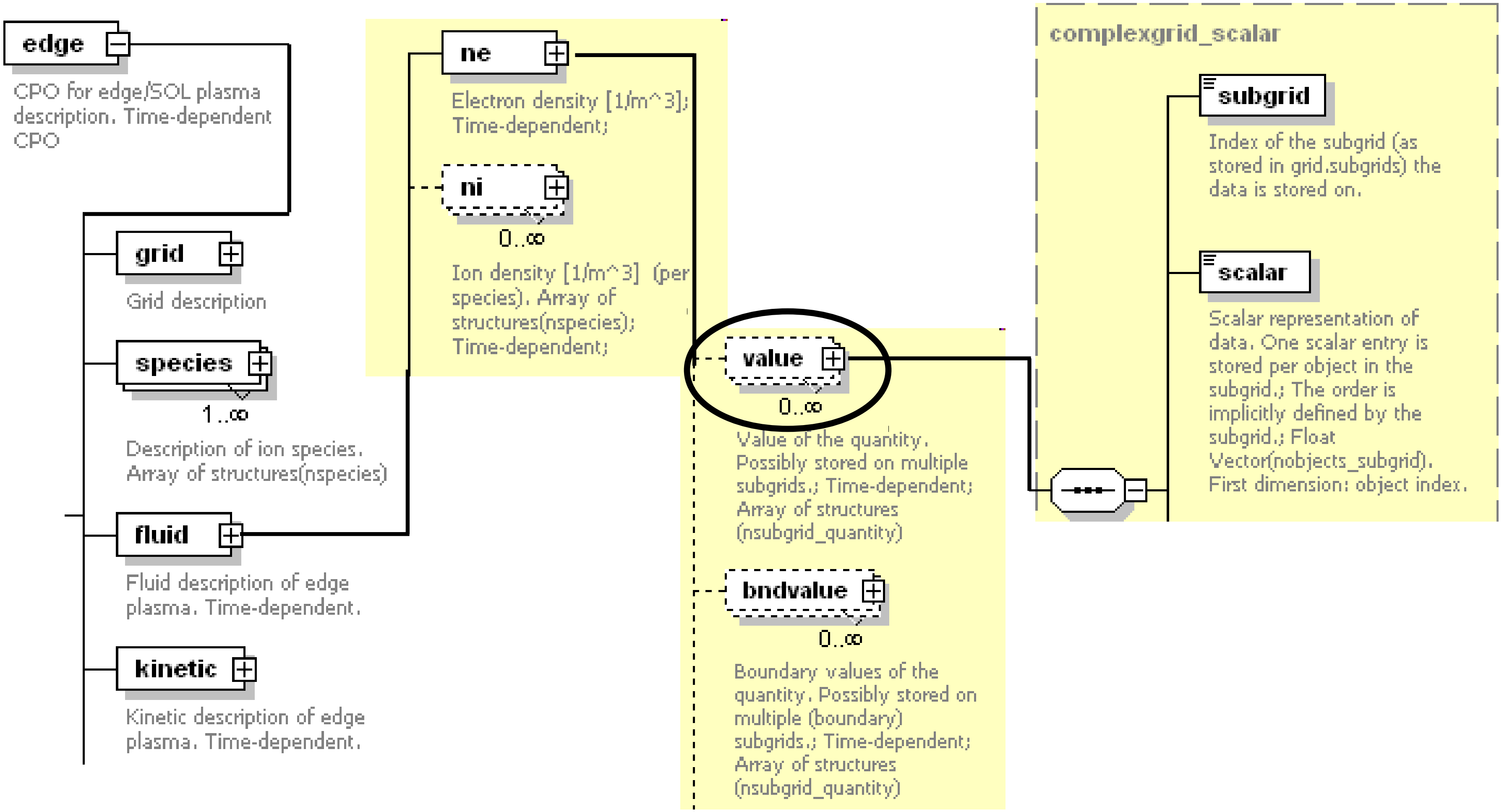
CPO Layout (example: edge CPO) Grid data structure

Standard grid data structure:
complexgrid (defined in utilities.xsd)



- typically placed in same hierarchy level as associated fields
- multiple grid definitions per CPO possible

CPO Layout (example: edge CPO) Data fields



- CPO data fields: standard data type complexgrid_scalar
- use arrays of scalars to allow storage on multiple subgrids

CPO I/O in codes: Writing grids

```
! Use module from grid service library
use itm_grid_structured

integer, parameter :: NPOINTR = 6, NPOINTZ = 5
real(R8) :: rnodes(NPOINTR), znodes(NPOINTZ)


! Write a 2d structured R,Z grid
call gridSetupStructuredSep( &
     & edgecpo%grid, &
     & ndim = 2, &
     & c1 = COORDTYPE_R, x1 = rnodes, &
     & c2 = COORDTYPE_Z, x2 = znodes, &
     & id = '2d structured R,Z grid' )
```

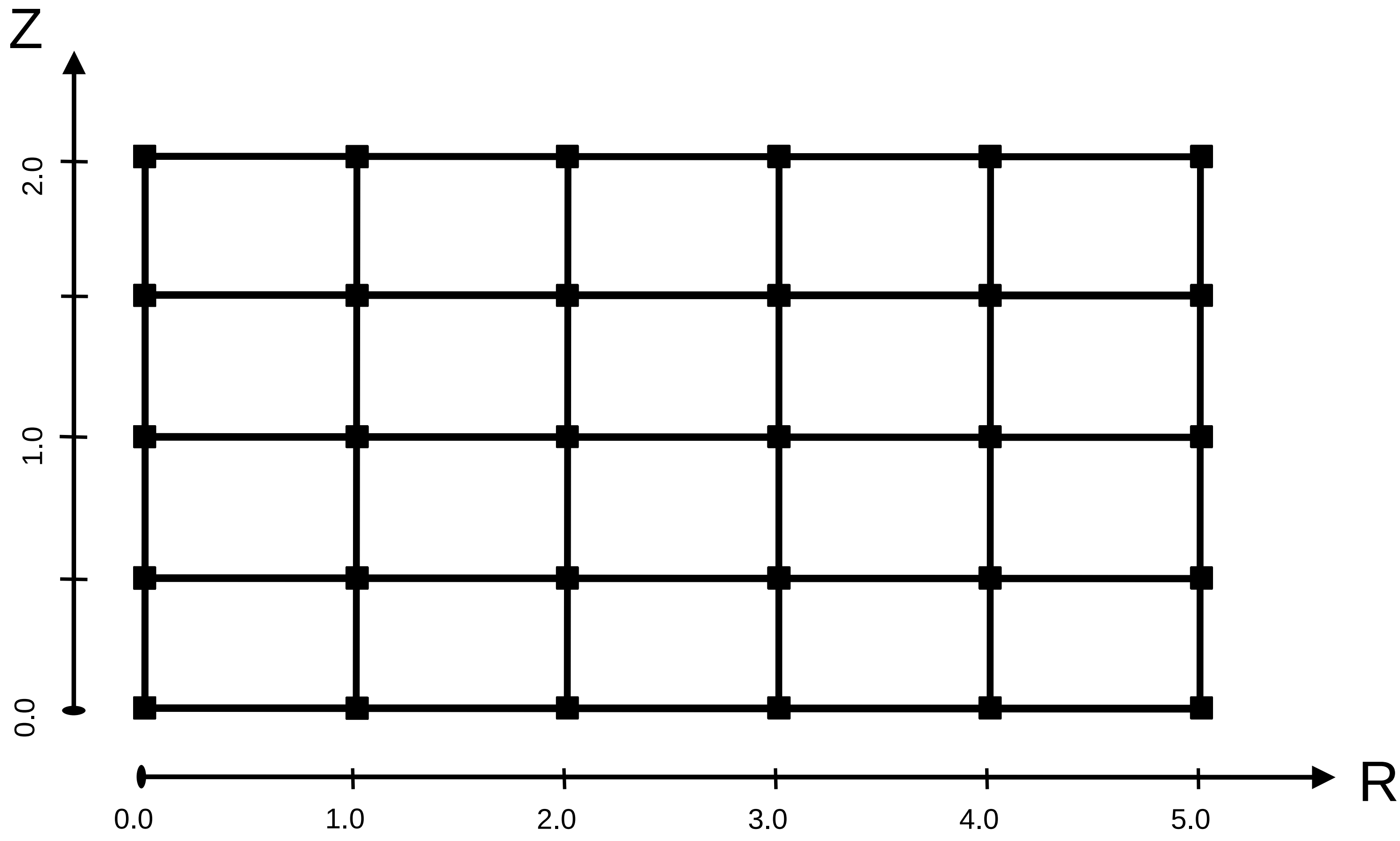
grid data structure

node positions

standardized coordinate types

Grid id / name

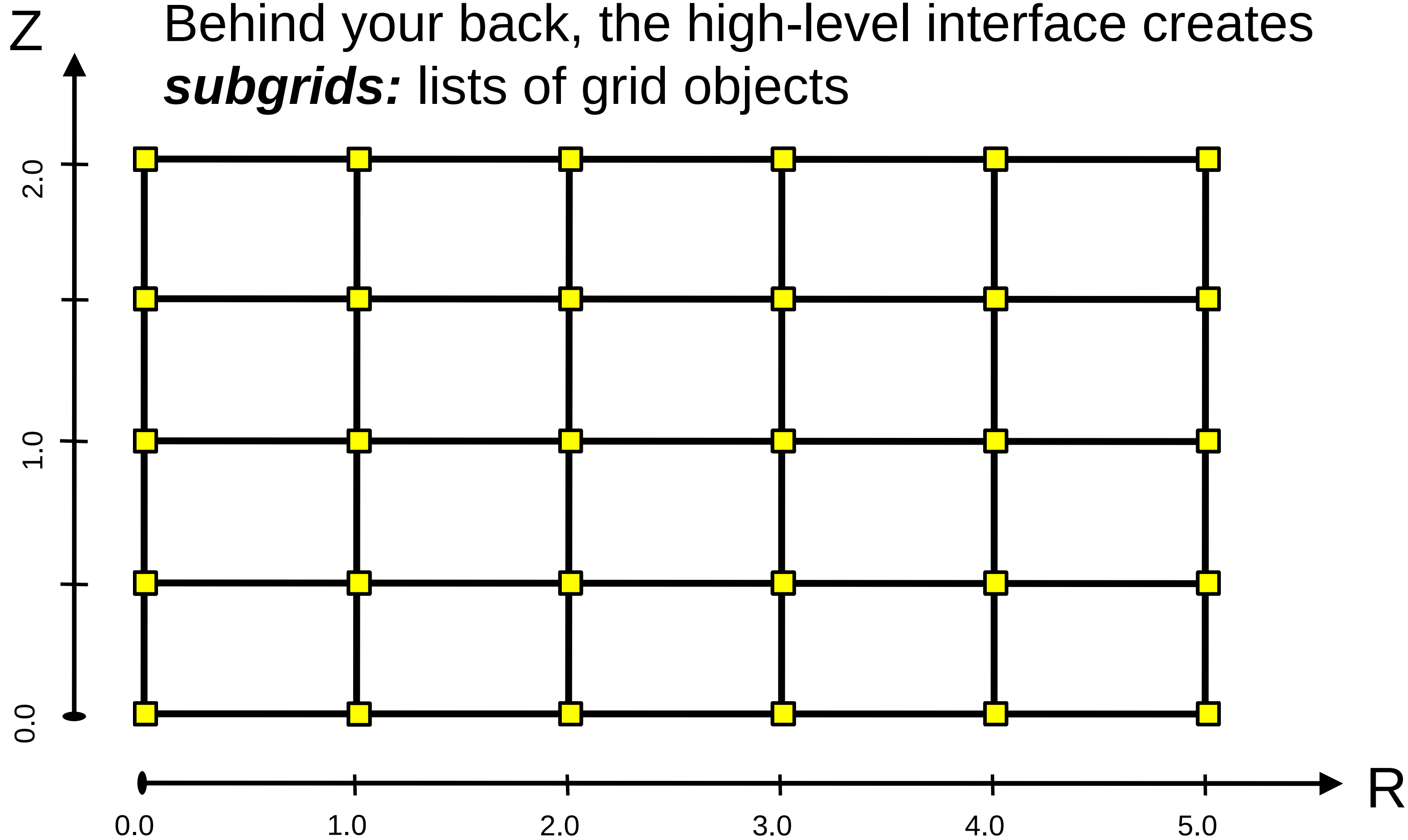
Simple 2d structured grid



2d structured grid in the R,Z plane

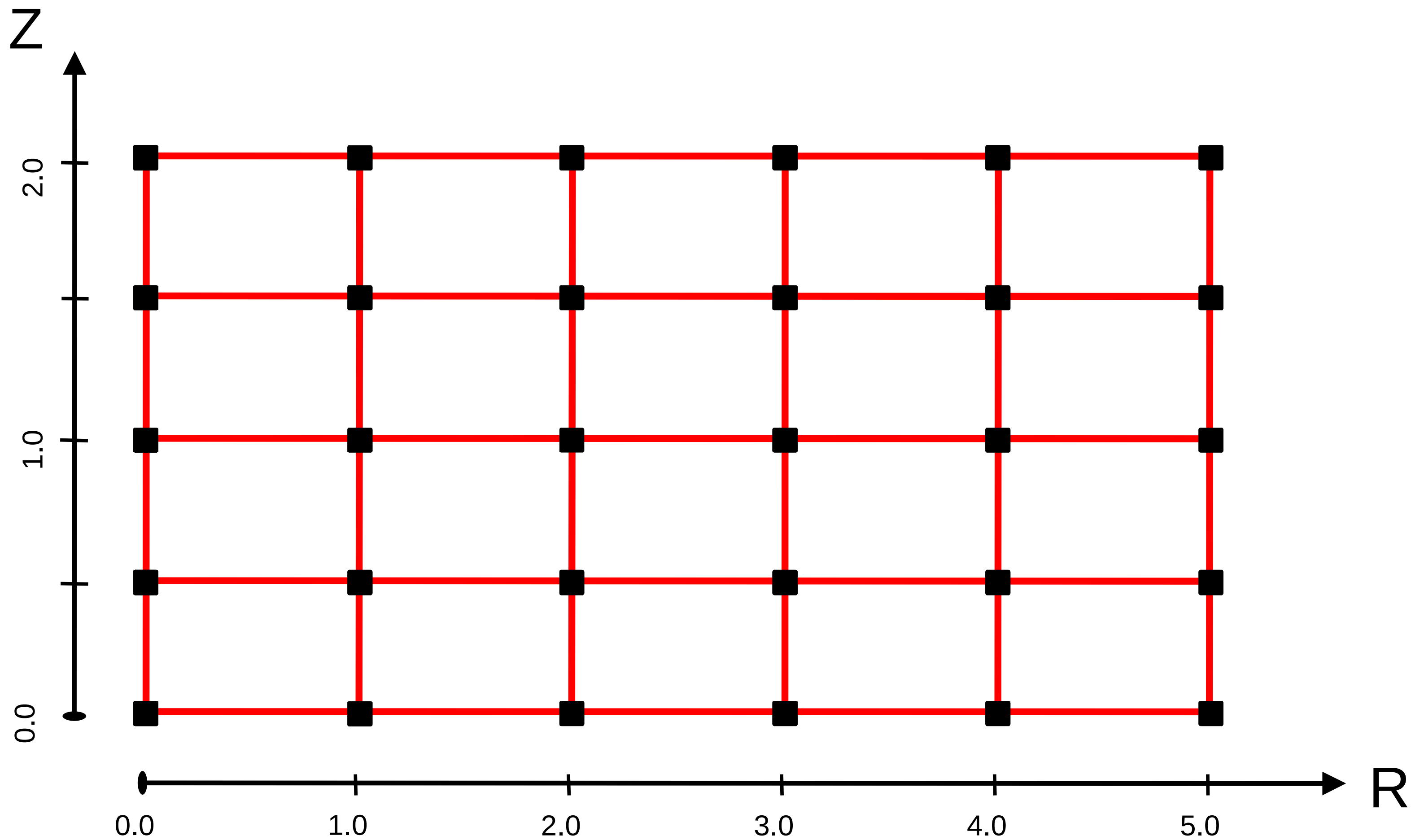
Subgrid: nodes

Behind your back, the high-level interface creates ***subgrids:*** lists of grid objects



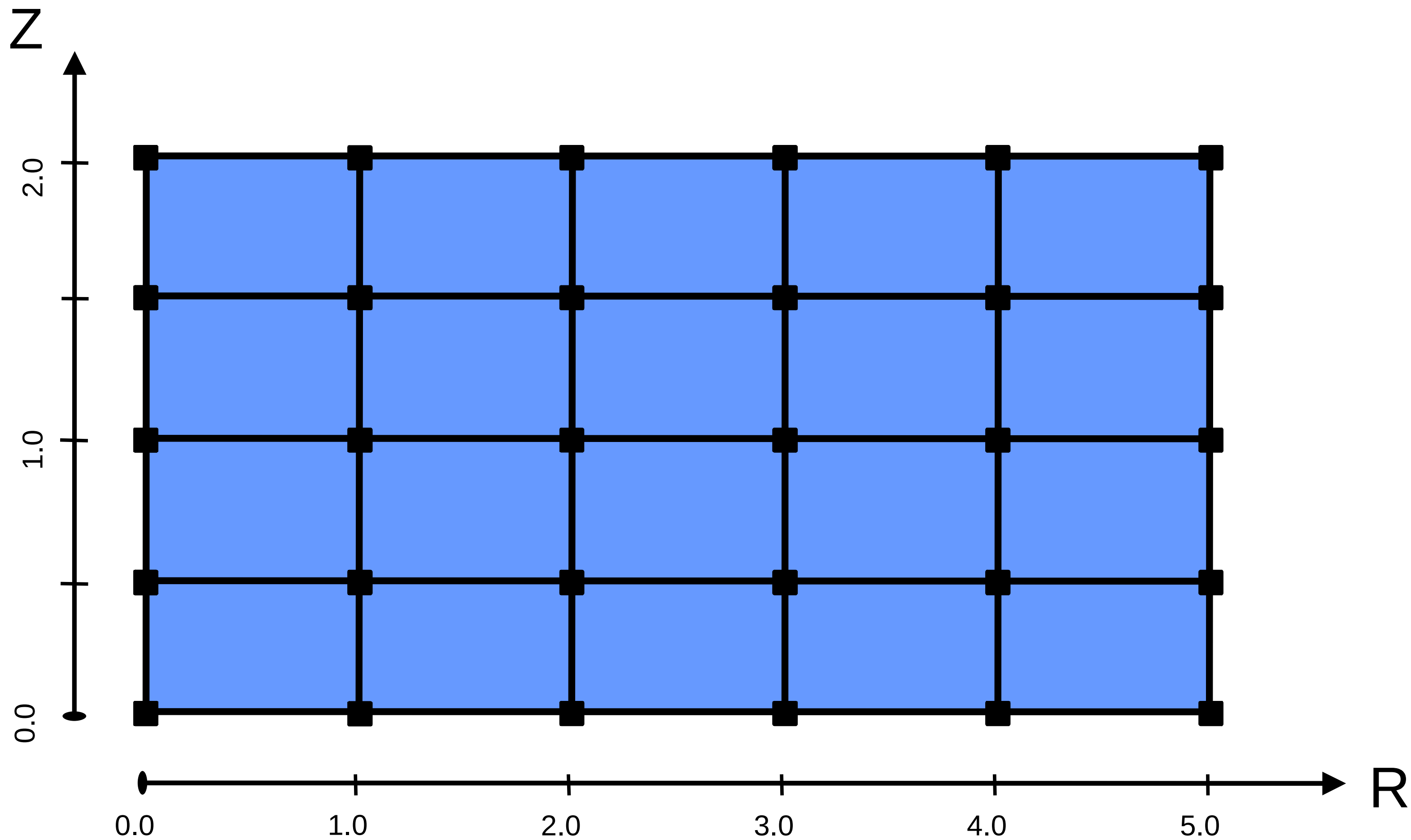
- subgrids are identified by their index
- this example:
  module itm_grid_structured: constant GRID_STRUCT_NODES

Subgrid: edges



Module itm_grid_structured: constant GRID_STRUCT_EDGES

Subgrid: faces (2d cells)



Module itm_grid_structured: constant GRID_STRUCT_FACES

Writing data
Fortran interface

```fortran
real(R8) :: cellData(NPOINTR - 1, NPOINTZ - 1)
real(R8) :: nodeData(NPOINTR, NPOINTZ)


allocate(edgecpo%fluid%ne%value(2))

! Write data on 2d cells ("faces")
call gridStructWriteData( &
     & edgecpo%grid, &
     & edgecpo%fluid%ne%value(1), &
     & GRID_STRUCT_FACES, cellData )

! Write data on nodes
call gridStructWriteData( &
     & edgecpo%grid, &
     & edgecpo%fluid%ne%value(2), &
     & GRID_STRUCT_NODES, nodeData )
```

grid data structure

CPO data field

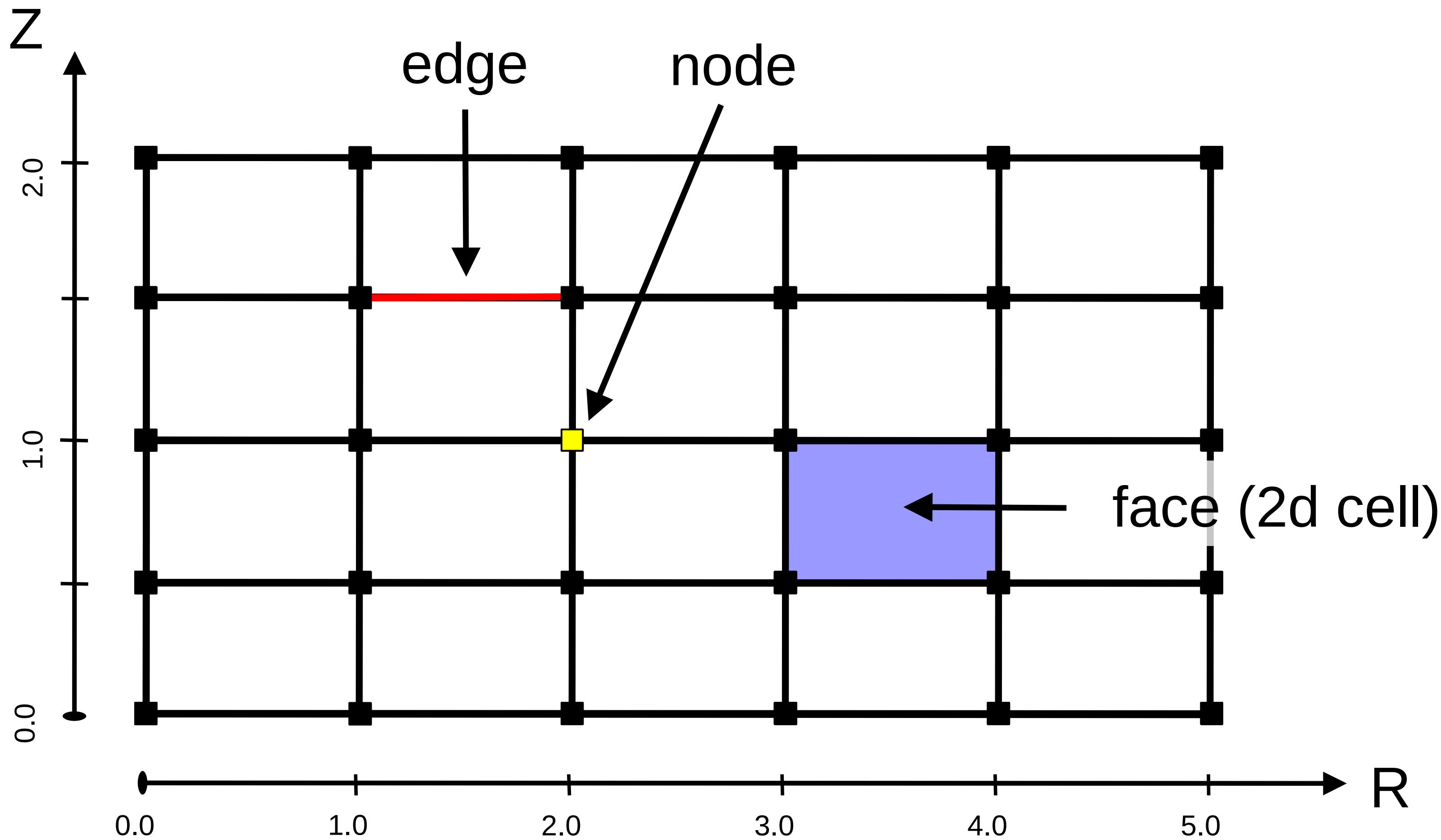
data array

subgrid index

Subgrids are central to reading and writing data

General grid description: some details (optional)

Describing discretizations: General approach



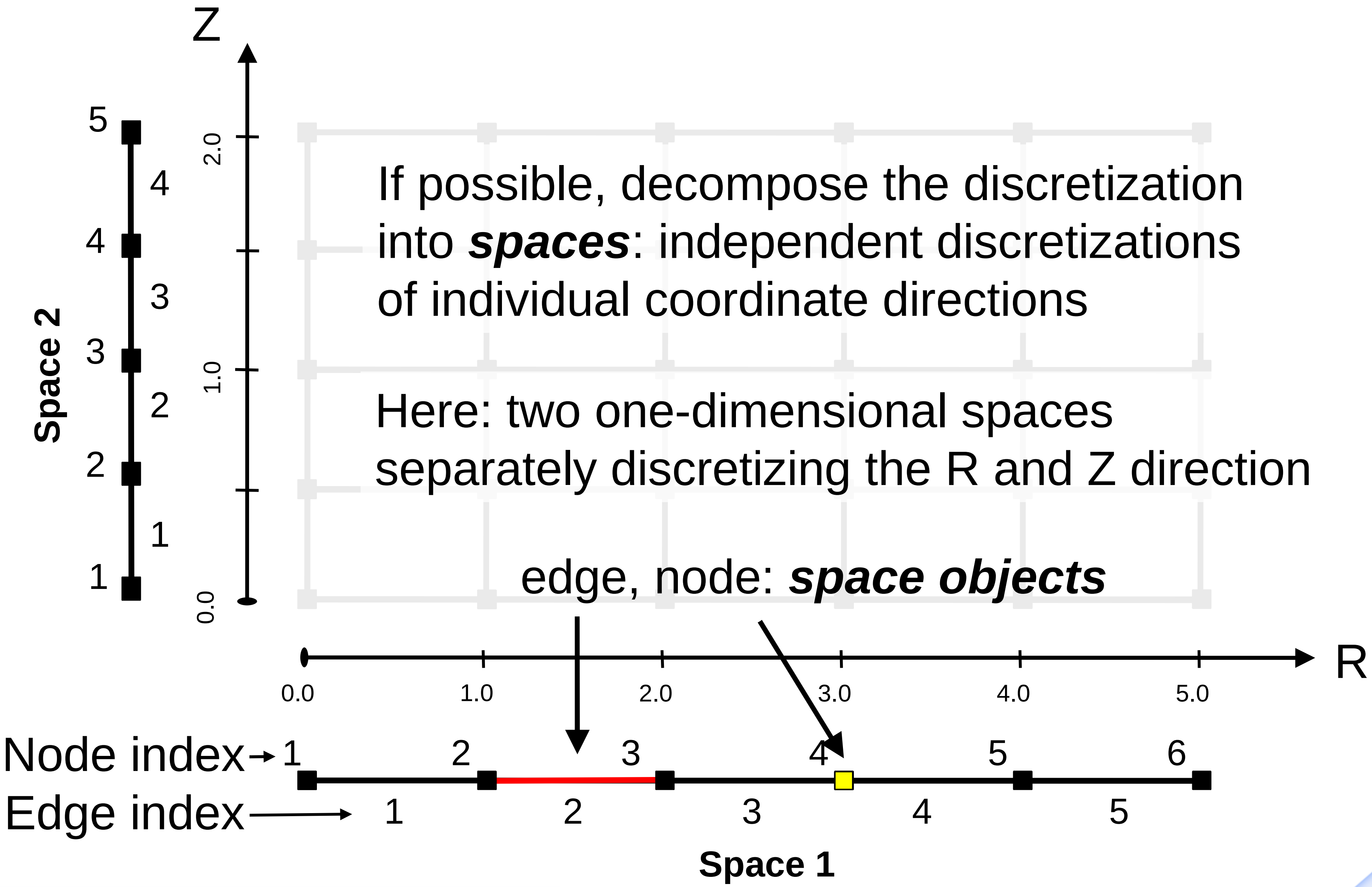
The general grid description identifies and describes ***grid objects***: nodes, edges, …

Describing discretizations: Space decomposition

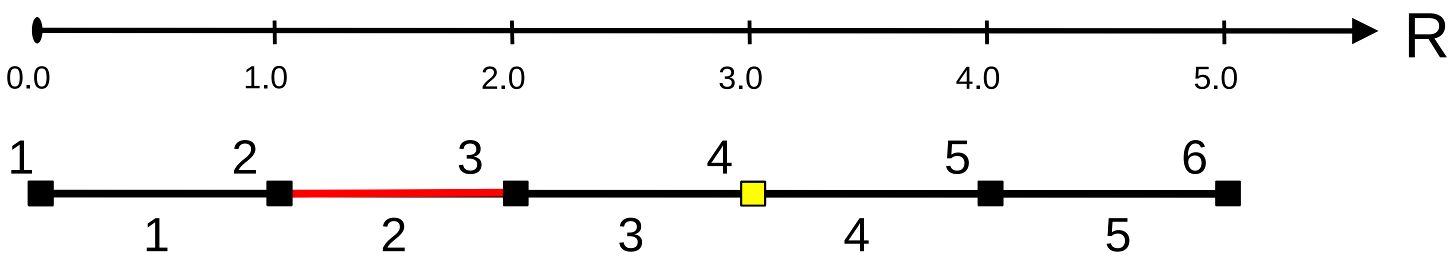
If possible, decompose the discretization into ***spaces***: independent discretizations of individual coordinate directions

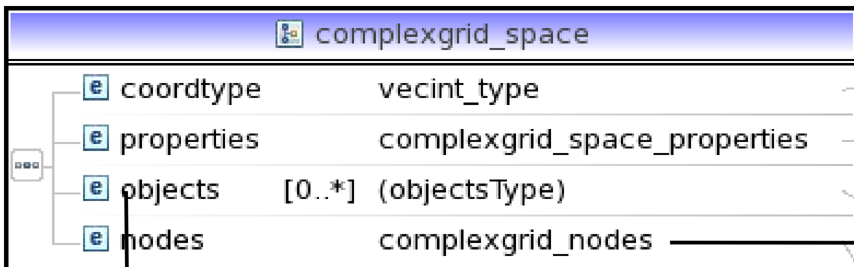
Here: two one-dimensional spaces separately discretizing the R and Z direction

edge, node: ***space objects***

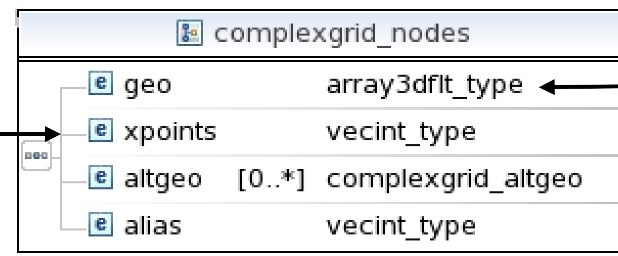


Storing subobject information



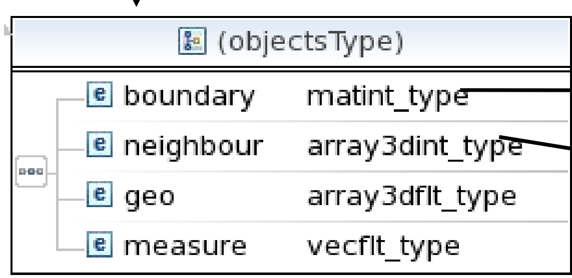


0d space objects: nodes



Node positions

n-dimensional space objects (n>0): assembled from (n-1)d space objects

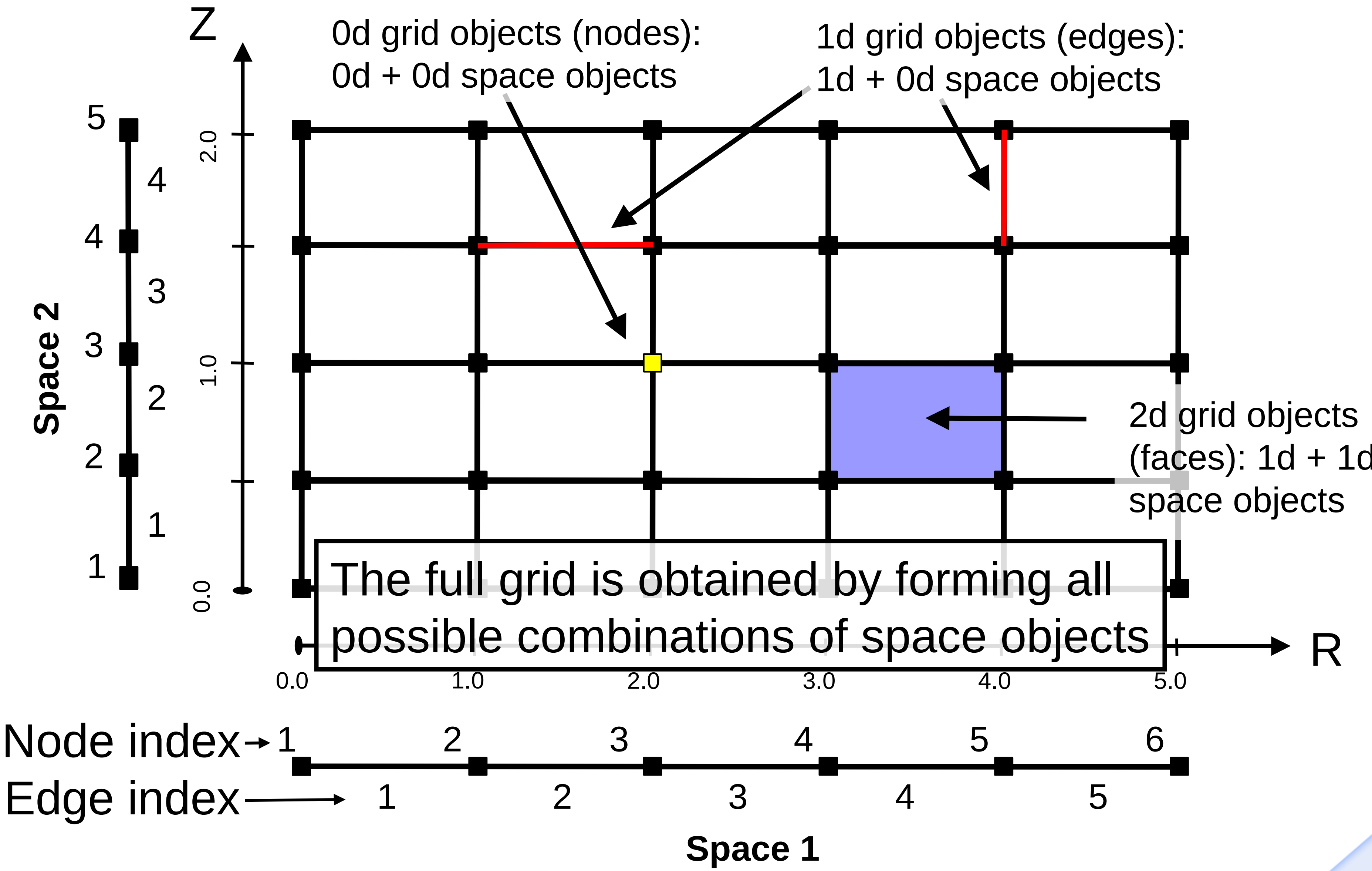


Boundaries of objects

Connectivity

These fields store space object indices

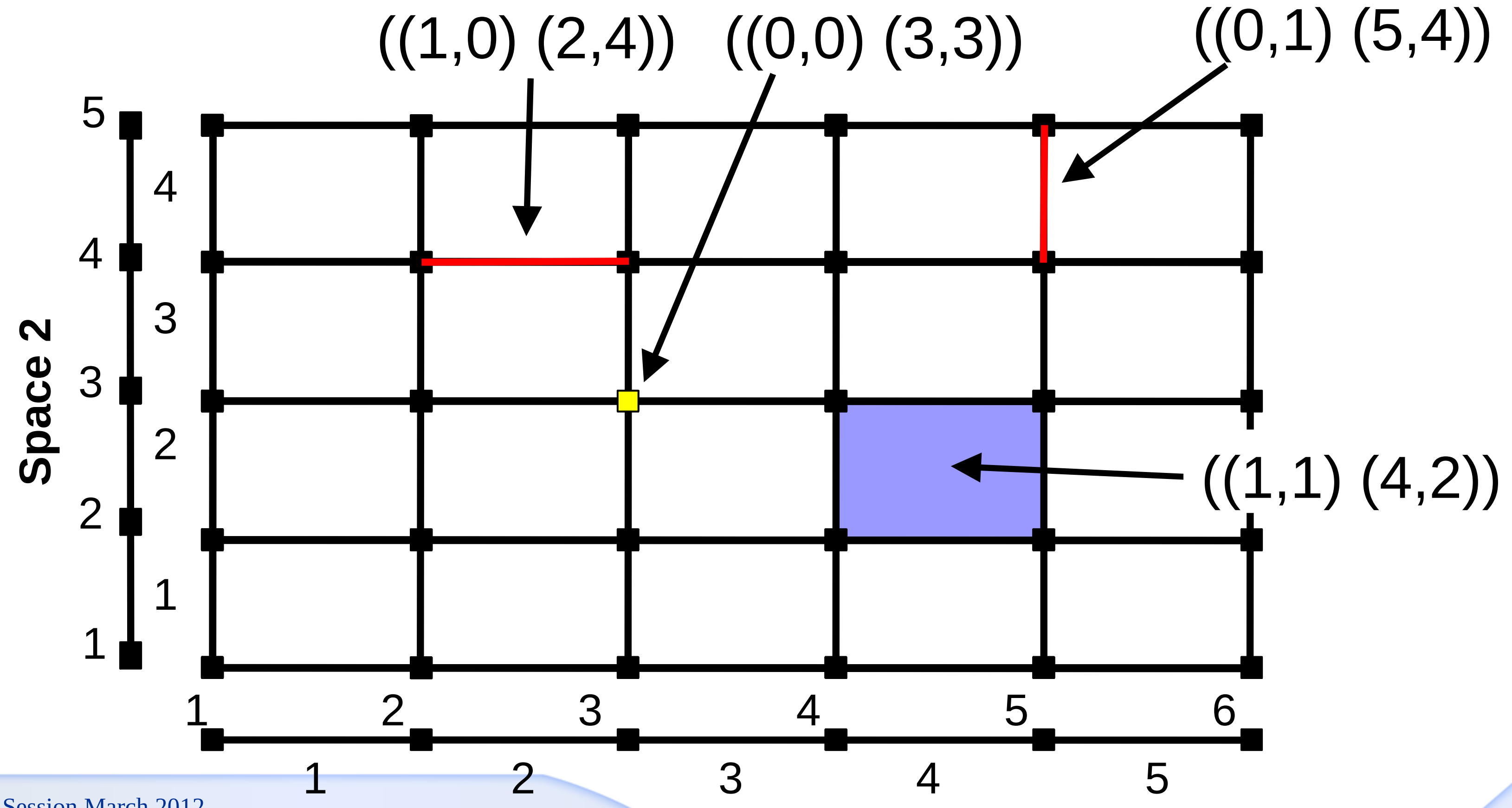
Describing discretizations: Space combination/multiplication



Object Descriptor: $((c_1, c_2,...,c_n)\ (i1,i2,...,in))$

Object class:
c_j = subobject dim. in space j

Object index:
i_j = subobject index in space j



Global object order

- The space objects have an explicit (*local*) order in their respective space (simply the order in which they are defined)
- For the implicitly defined grid objects, a *global* order is imposed by adopting a simple counting convention (think linear address computation for multidimensional Fortran arrays):

Grid objects of a common object class are counted by varying the leftmost index of the index tuple first.

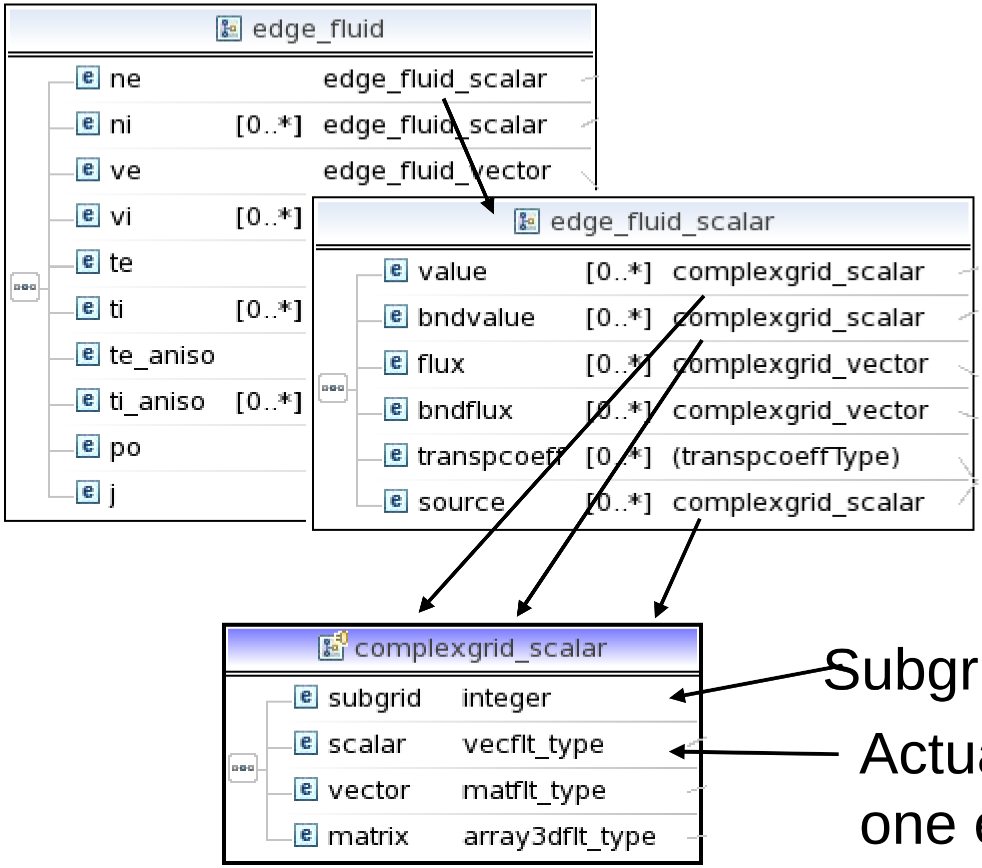
This imposes an ordering of grid objects of a common class. Every grid object can therefore be uniquely identified by:

Its object descriptor:
((c1, c2,...,cn) (i1,i2,...,in))

or

Its object class and
global index ig:
((c1, c2,...,cn) ig)

Storing data on grids: subgrids



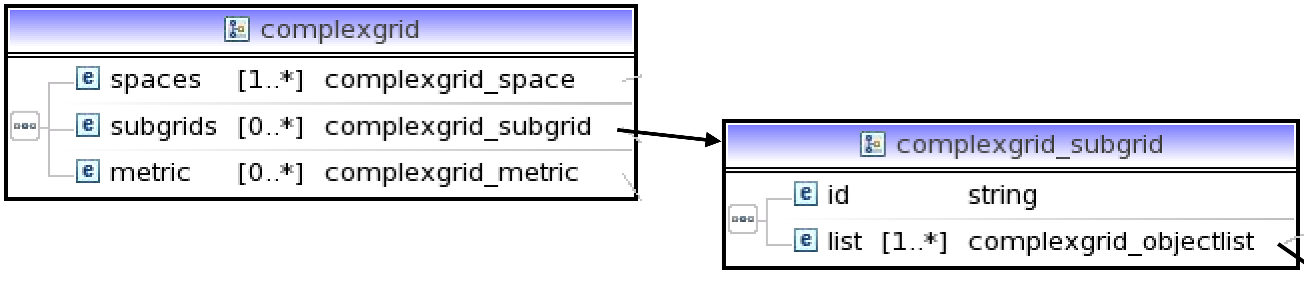
- data is stored on a ***subgrid***
- *subgrid* = list of grid objects
- the data is then stored simply as a vector, one entry per object in the subgrid

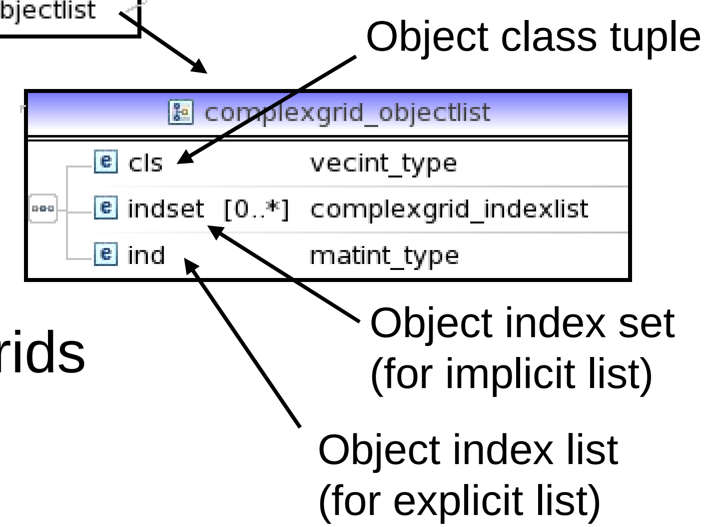
Subgrid index

Actual data:
one entry per subgrid object,
in the order defined in the subgrid

(vector, matrix: for complex
data representations)

Subgrid definition





- A subgrid is a selection of a subset of grid objects (of common dimension)

- There can be an arbitrary number of subgrids (which are part of the grid description).
A specific subgrid is identified by its index.

- A subgrid is a list of *object lists*. Each object list can be either
  - ***explicit****:* an explicit list of object descriptors
  - ***implicit****:* an implicit list of object descriptors, selecting a range or an entire class of objects

Subgrids: Implicit object list notation

Object List Descriptor: ((c1, c2,...,cn) (s1,s2,...,sn))

Object class: cj = subobject dim. in space j (same as in object descriptor)

Object index set: specifies multiple indices

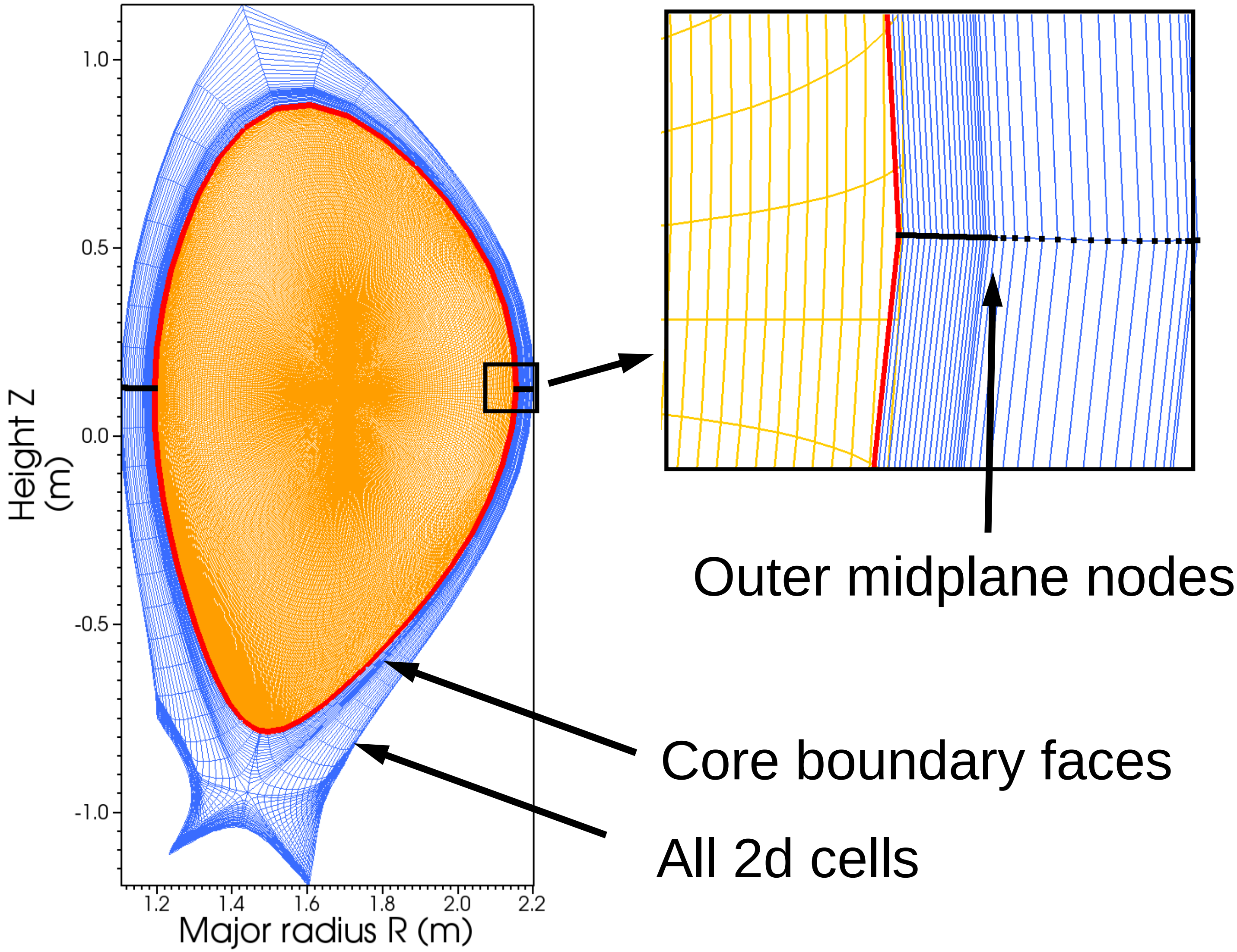
sj = [i1, i2, …, iN] → explicit list of indices
sj = (i1,i2) → range of indices from i1 to i2
sj = UNDEFINED → all possible indices

Object order: implicit object lists inherit the object order intrinsic to the underlying grid definition

Why is this important? Space splitting and implicitly defined object order allow efficient handling of datasets on very large (5d, 6d,...) structured grids.

Subgrids
Example: SOLPS-B2, single null

- Cells
- Core
- Core_boundary
- Core_cut
- Faces
- Inner_PFR_wall
- Inner_baffle
- Inner_divertor
- Inner_midplane
- Inner_target
- Inner_throat
- Main_chamber_wall
- Outer_PFR_wall
- Outer_baffle
- Outer_divertor
- Outer_midplane
- Outer_target
- Outer_throat
- PFR_cut
- SOL
- Separatrix
- x_aligned_faces
- x_points
- y_aligned_faces



Grid Service Library

- The Grid Service Library (GSL) provides functions to simplify working with the General Grid Description data structures (read/write grids and data)

- Current approach:
  - Provide dedicated implementations for separate languages, exploiting their strengths (like UAL)
  - Some basic functionality present in all implementations, consistency ensured by unit test framework
  - Advanced functionality will diverge depending on typical use cases for the different languages

- Current languages:
  - **Fortran 90**: procedural, typical for codes
  - **Python**: object oriented, typical for post-processing tools/glue scripts

} Can serve as starting point for other implementations

Getting the service library

- GForge: maintained in itmggd project
- Getting a local copy:

svn co http://gforge.efda-itm.eu/svn/itmggd	
cd grid	
source setup.csh	Sets up F90&Python environment
testgrid setup	Writes some example grids to database
testgrid all	Runs integration unit tests for all languages
…Test all implementations: OK	

- Public copy currently provided at ~**klingshi/bin/itm-grid** (will move to a better place in the future)
- Your environment has to be set up for 4.09a
- Instructions at https://www.efda-itm.eu/ITM/html/imp3_grid.html

Grid Service Library: Directory layout

```
•grid/
  • f90/
     • src/
        • service/        Fortran service library modules
        • examples/       Example programs
        • test/           Unit tests
  • python/
     • itm/
        • grid/           service library classes
        • visit/          ualconnector/Visit integration
        • test/           Unit tests
```

Documentation:

- General: IMP3 section of documentation website
- Documentation partially generated from source (Doxygen/Sphinx)
  `make doc; make doc_release`
  → https://www.efda-itm.eu/ITM/doxygen/imp3/grid_service_library/

Grid Service Library: Fortran 90

Structured in modules. Some more interesting ones:

- **`itm_grid_access`**: accessing basic grid properties
- **`itm_grid_object`**: handling grid objects
- **`itm_grid_subgrid`**: handling subgrids
- **`itm_grid_structured`**: high-level interface for structured grids
- **`itm_grid_simplex`**: high-level interface for simplex grids (triangles…)

Subroutines & functions acting on on standard data types:

```
!> Get the total number of objects
!> of the given dimension in the given space
integer function gridSpaceNObject( space, dim ) result( objcount )
  type(type_complexgrid_space), intent(in) :: space
  integer, intent(in) :: dim
```

Hands-On:
Fortran Grid Service Library

Please go to the documentation website → IMP3 →

IMP3 General Grid Description and
Grid Service Library - Tutorial

VisIt integration

- The general grid description enables general operations with grids and data
- Example: general visualization tools for complex discretizations and data sets
- LLNL VisIt (https://wci.llnl.gov/codes/visit/)
  - Coupling to VisIt at the moment done with helper program "ualconnector"
  - In the future this will be simpler: you will be able to access the plots through the normal methods (itmvisit, VisIt kepler actor)

ualconnector: Basic usage

```
~klingshi/bin/itm-grid/ualconnector
    -s 17151,898,100.0 -c edge
    -s 17151,899,100.0 -c edge
    -u coster -t aug -v 4.09a
```

← can specify multiple CPOs

Options:

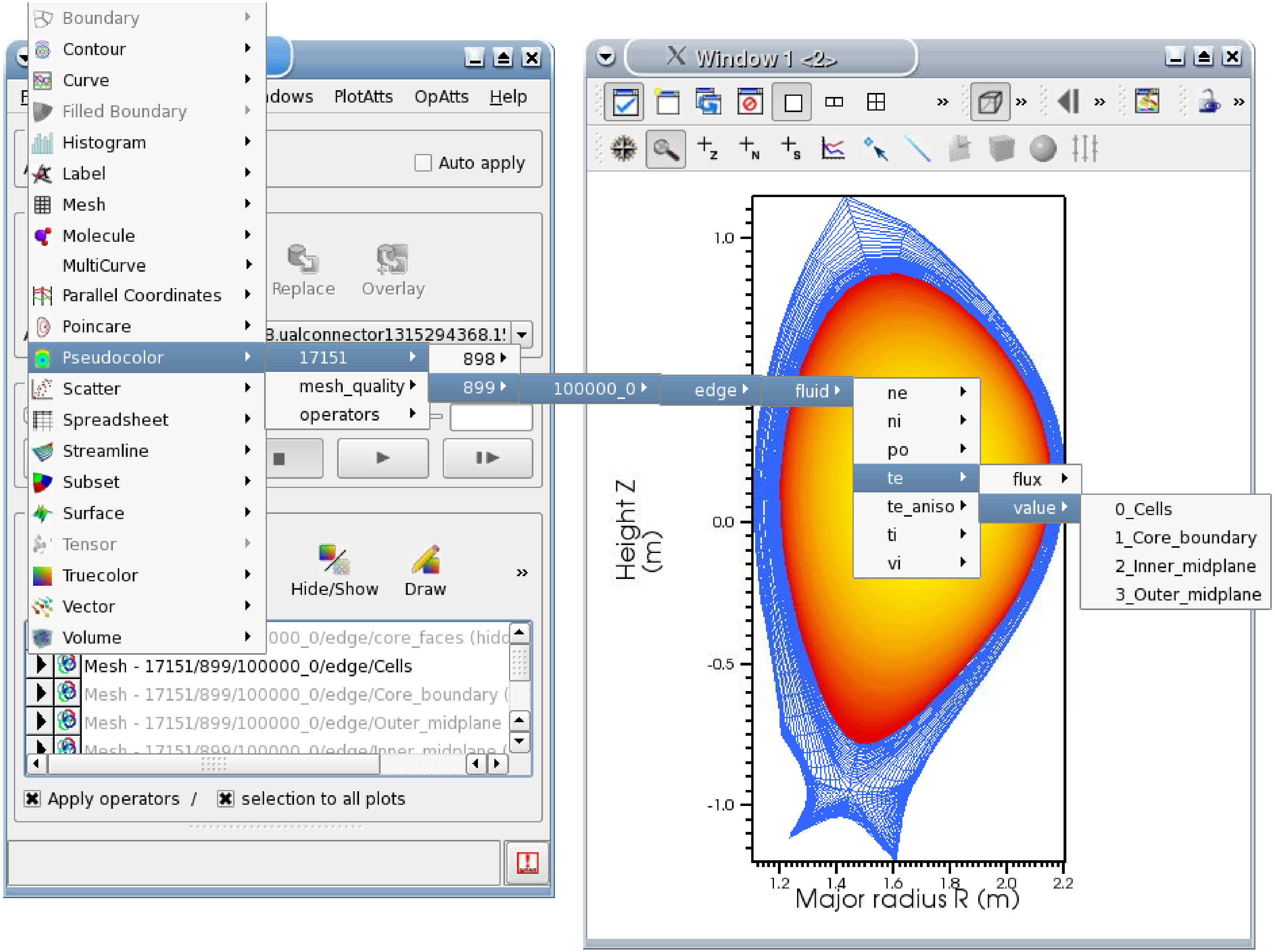
-s run,shot,time
-c cpo-name

} have to be specified in pairs

-u username
-t Tokamak name
-v data version

- this will automatically launch & connect a VisIt 2.3 instance
- your environment has to be set up for data version 4.09a
- currently only makes sense for the edge CPO

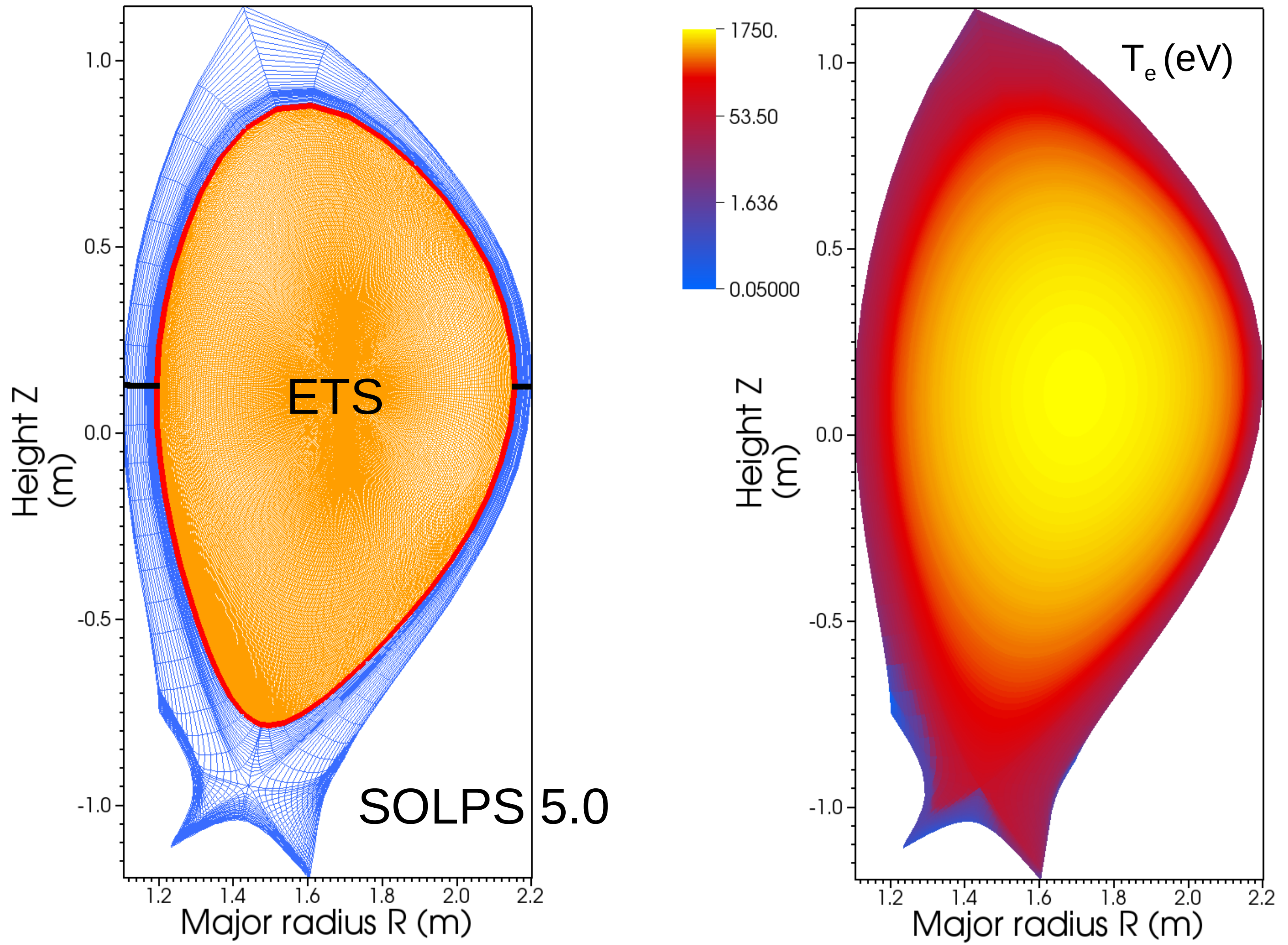
VisIt GUI


Boundary
Contour
Curve
Filled Boundary
Histogram
Label
Mesh
Molecule
MultiCurve
Parallel Coordinates
Poincare
Pseudocolor
Scatter
Spreadsheet
Streamline
Subset
Surface
Tensor
Truecolor
Vector
Volume
PlotAtts
OpAtts
Help
Auto apply
Replace
Overlay
17151
mesh_quality
operators
898
899
100000_0
edge
fluid
ne
ni
po
te
te_aniso
ti
vi
flux
value
0_Cells
1_Core_boundary
2_Inner_midplane
3_Outer_midplane
Hide/Show
Draw
Mesh - 17151/899/100000_0/edge/Cells
Mesh - 17151/899/100000_0/edge/Core_boundary
Mesh - 17151/899/100000_0/edge/Outer_midplane
Apply operators / selection to all plots
Window 1 <2>
Height Z (m)
Major radius R (m)
1.0
0.0
-0.5
-1.0
1.2
1.4
1.6
1.8
2.0
2.2

Hands-On:

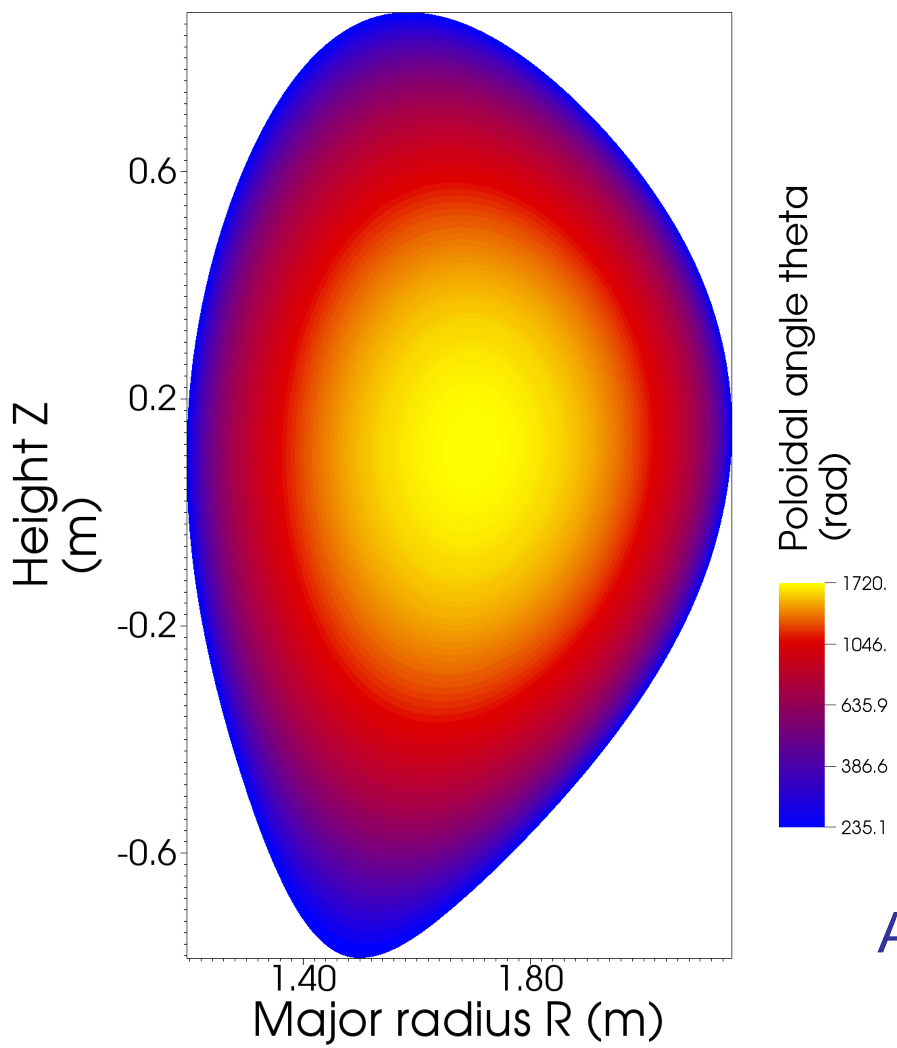
General Grid Description – Visualization with VisIt

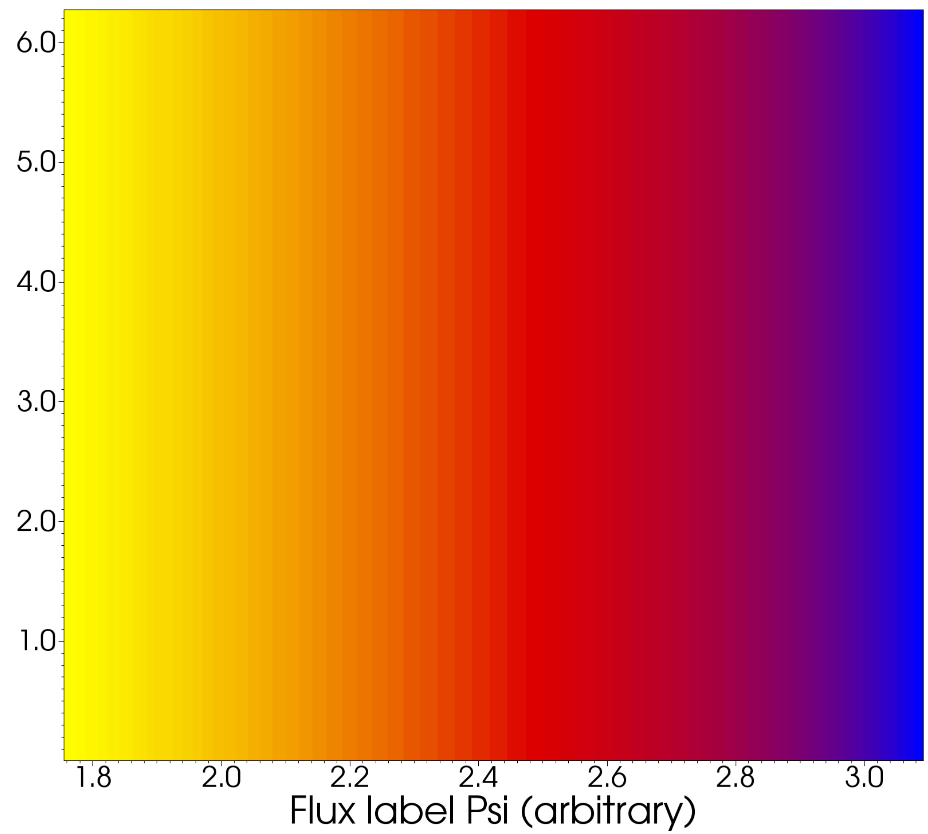
Application: core-edge coupling



Some advanced features

Advanced features: alternate geometries





Alternate geometries:

Explicitly include node positions in alternate coordinate systems

More features

- Additional object properties
  - measures: length, area, volume …
  - metric information: jacobian, metric tensor
  - Identification of x-point nodes
- Periodicity
  - either directly through object connectivity
  - or indirectly through node aliasing

…we are quickly moving into area of experimental features – much of this still
has to be tested in real applications ….

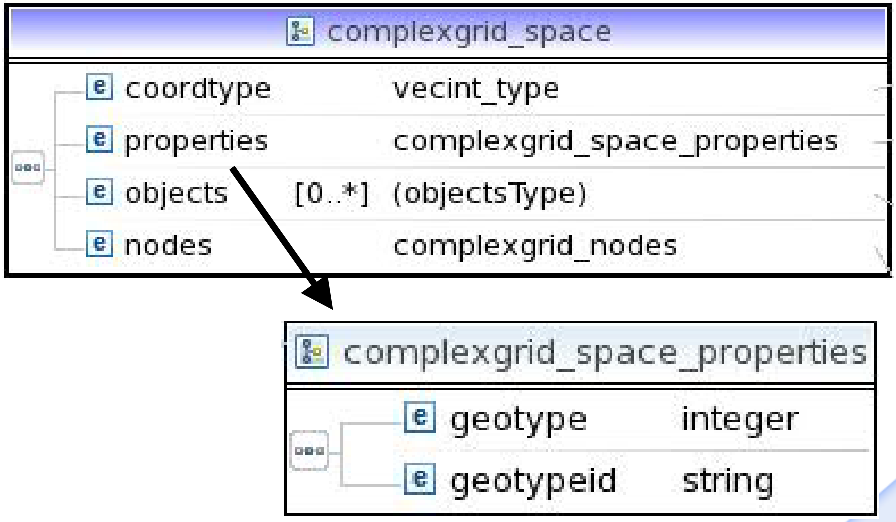
Experimental features: non-standard representations

“Standard” geometry & data representation:

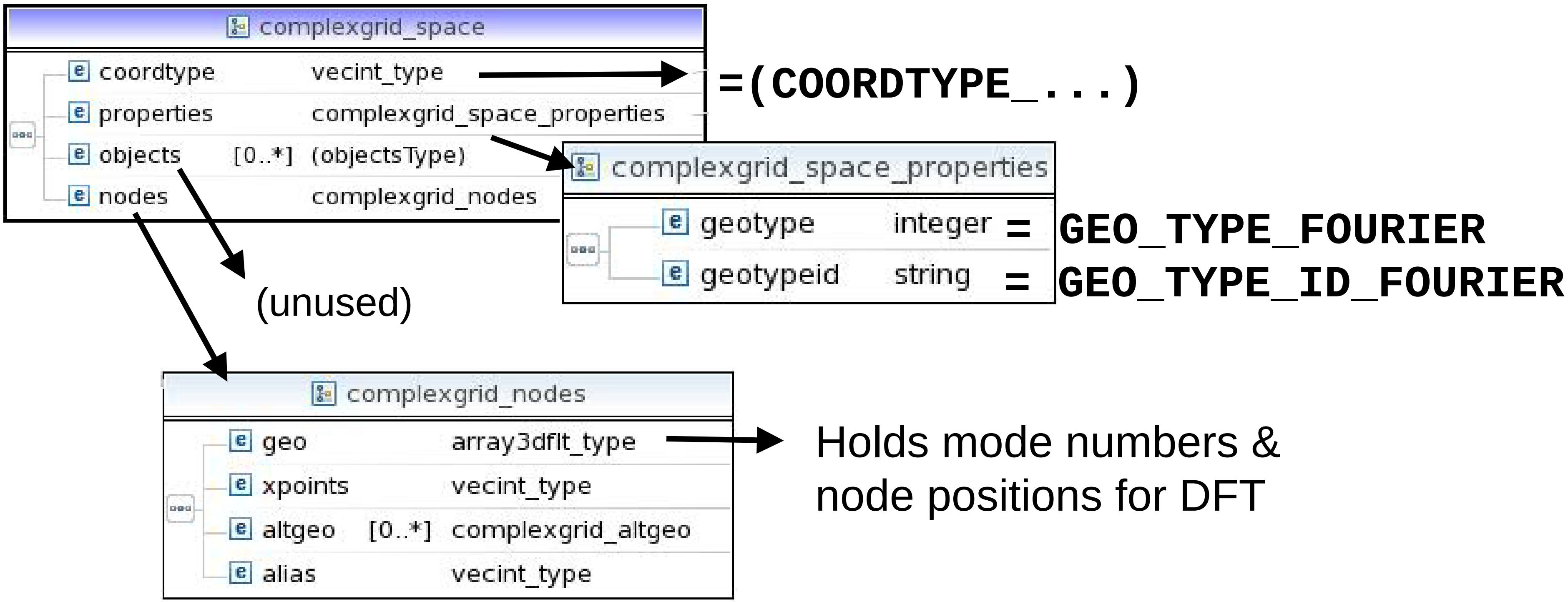
- Grid geometry directly given by node positions and object boundaries
- Data fields: one constant value per grid object (node value, area/volume average)

Non-standard representations:

- Flag in space data structure indicates alternate interpretation of space definition
- The rest is up to you

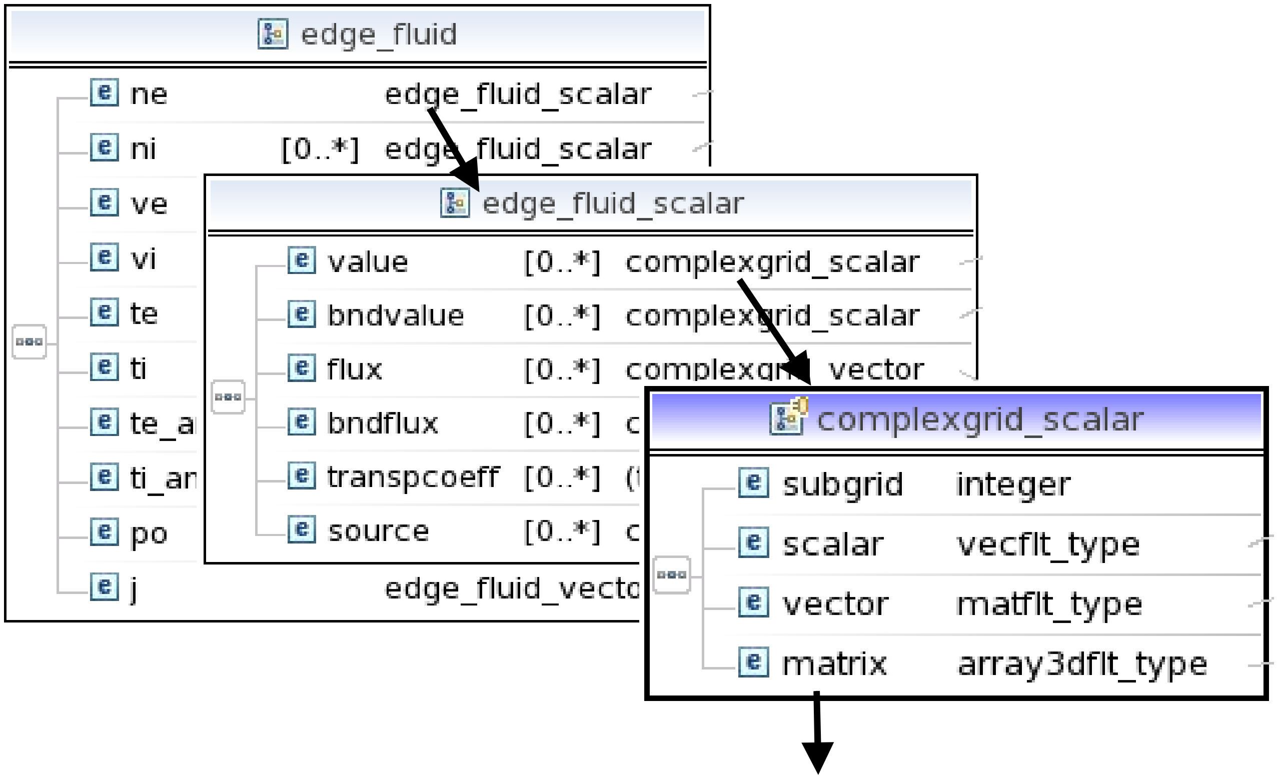


Example: Fourier representation, space data structure



- The space data structure holds information required for an unambiguous inversion of the DFT
- Geometry data with additional degrees of freedom also possible for implicitly defined grid objects

Example: Fourier representation, data field structure

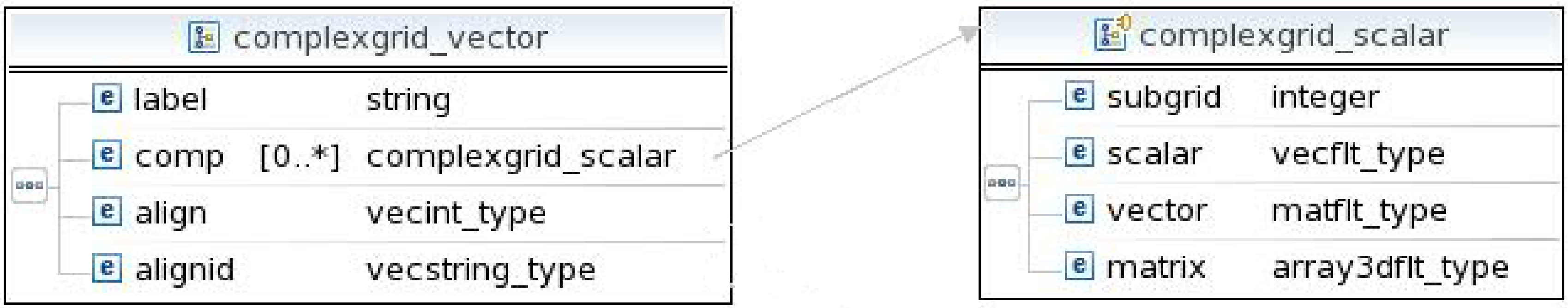


`matrix(i,:,1)` = real
`matrix(i,:,2)` = imaginary

} components of DFT for object with index i in subgrid

To spare us pain, the DFT definition of FFTW is used.
…but of course you can just as well propose your own!

Experimental features: vector data type



- A complexgrid_vector is a vector of complexgrid_scalars
- The components can possibly be aligned to something:
    - another vector quantity
    - a set of base vectors, defined as part of the grid
- This is work in progress, application driven

Python

Grid Service Library: Python

Basic design:

- classes wrapping data structures, which implement methods acting on them (could include functionality in the UAL objects, but this would lead to lots of complications)
- Python *Inspection* capabilities allow dynamic analysis of CPO structure and contents (without prior knowledge)

Main classes (class name:wrapped data structure)

- **`itm.grid.cpo_tools.Cpo: general CPO wrapper`**
- **`itm.grid.base.Grid: complexgrid`**
- **`itm.grid.base.SubGrid: complexgrid_subgrid`**
- **`Itm.grid.data.ScalarData: complexgrid_scalar`**

…objects typically created through the **`itm.grid.cpo_tools.Cpo`** CPO wrapper object

Reading grids: Python interface

- High-level handling of CPOs

Create wrapper object for CPO

```
In [1]: import itm.grid.cpo_tools as ct
In [2]: cpo = ct.Cpo(17151, 599, 'edge', 2.5, \
                     'coster', 'aug', '4.09a')
In [3]: cpo.grid.ndim
Out[3]: 2
In [4]: cpo.grid.coord_ids
Out[4]: ['Major radius R', 'Height Z']
```

Get grid object

- Subgrids

```
In [5]: cpo.grid.subgrid(1).id
Out[5]: 'Cells'
In [6]: len(cpo.grid.subgrid(1))
Out[6]: 3456
```

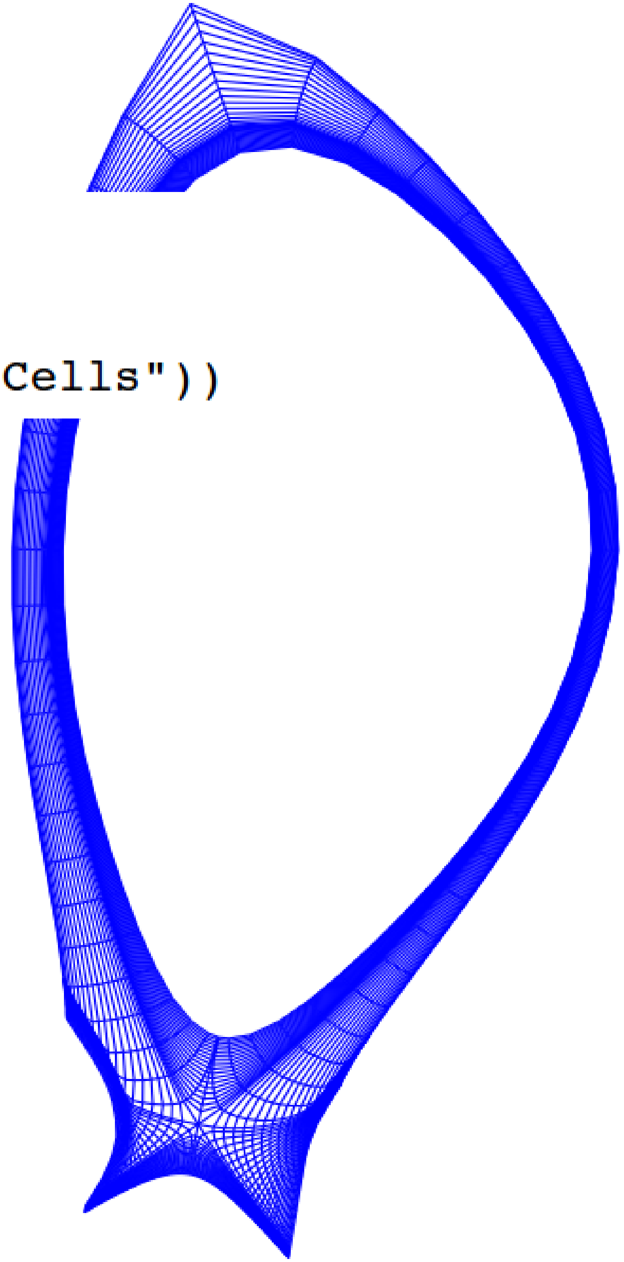
Subgrid objects act as sequences of `itm.grid.base.Object`

Reading grids: Python interface (ctd.)

- Plots via matplotlib

```
In [7]: import itm.grid.plot as gp
In [8]: plot = gp.Plot2d(cpo.grid)
In [9]: plot.plot_subgrid( \
        cpo.grid.subgrid_index_for_id("Cells"))
```

Subgrid lookup



Reading data: Python interface

```
In [11]: len(cpo.list_data())
Out[11]: 143
In [12]: cpo.list_data()[108].path
Out[12]: '/fluid/ne/value/array/0'
In [13]: cpo.list_data()[108].values
Out[13]:
array([  1.52043143e+19,   1.55988565e+19,   1.62138386e+19, ...,
         3.20049654e+19,   2.35636379e+19,   2.09316196e+19])
```

← Find used data fields in CPOs & return them as **`itm.grid.data.ScalarData`**

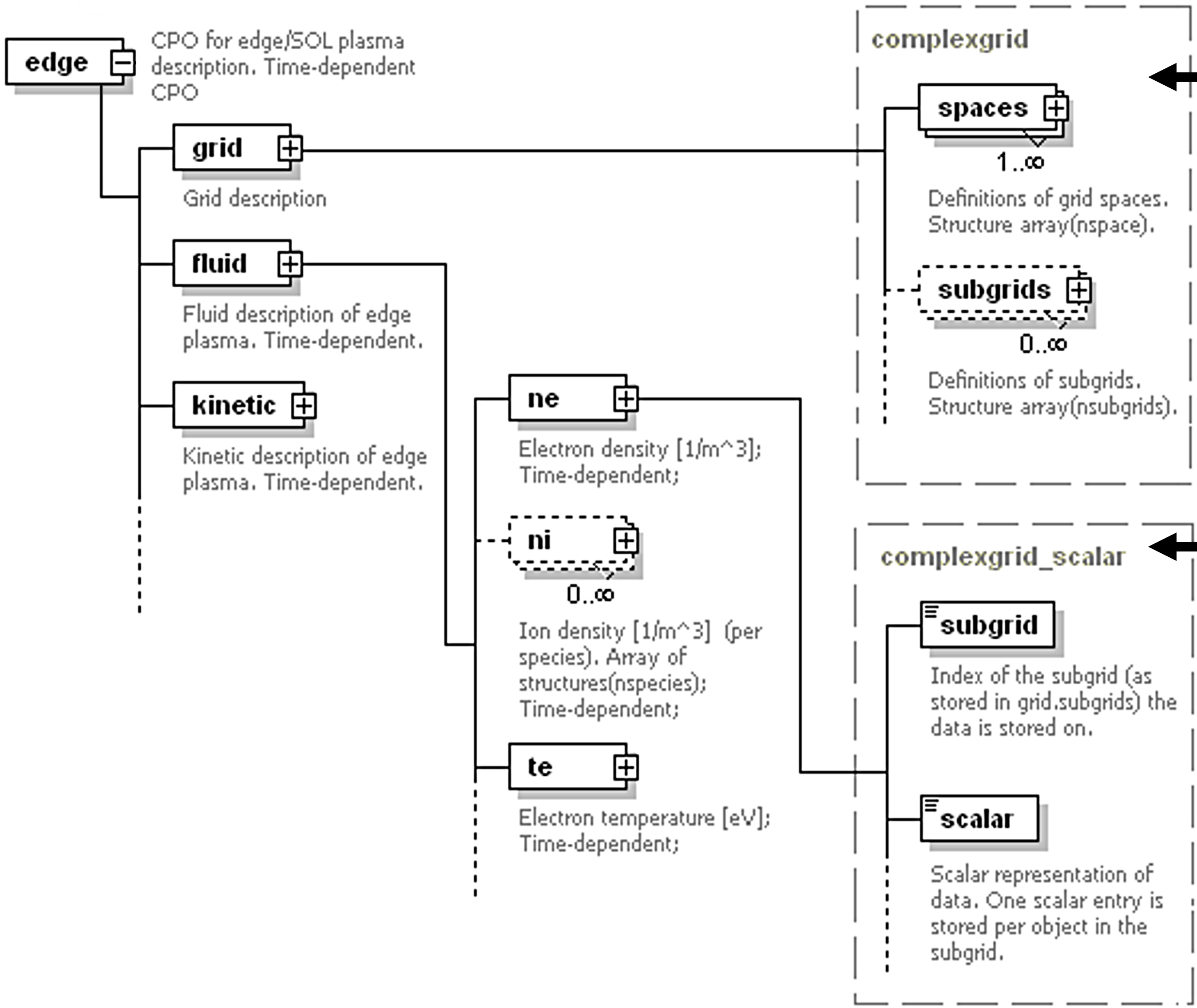
← Retrieve data

```
In [14]: cpo.list_data_lists()[24].path
Out[14]: '/fluid/ne/value/array'
In [15]: cpo.list_data_lists()[24].\
          values_for_objects(\
            cpo.grid.subgrid_for_id("Core boundary"))
Out[15]:
array([  4.30457187e+19,   4.30546443e+19,   4.30226645e+19,
         4.29851739e+19,   4.29729903e+19,   4.29782881e+19,
         ...
```

← Find used data arrays

← Automatic lookup of data for specific objects

How to use it?
1. CPO design



Add standard grid data structure in an appropriate place

Use standard data structures for CPO fields

… actually makes CPO design simpler

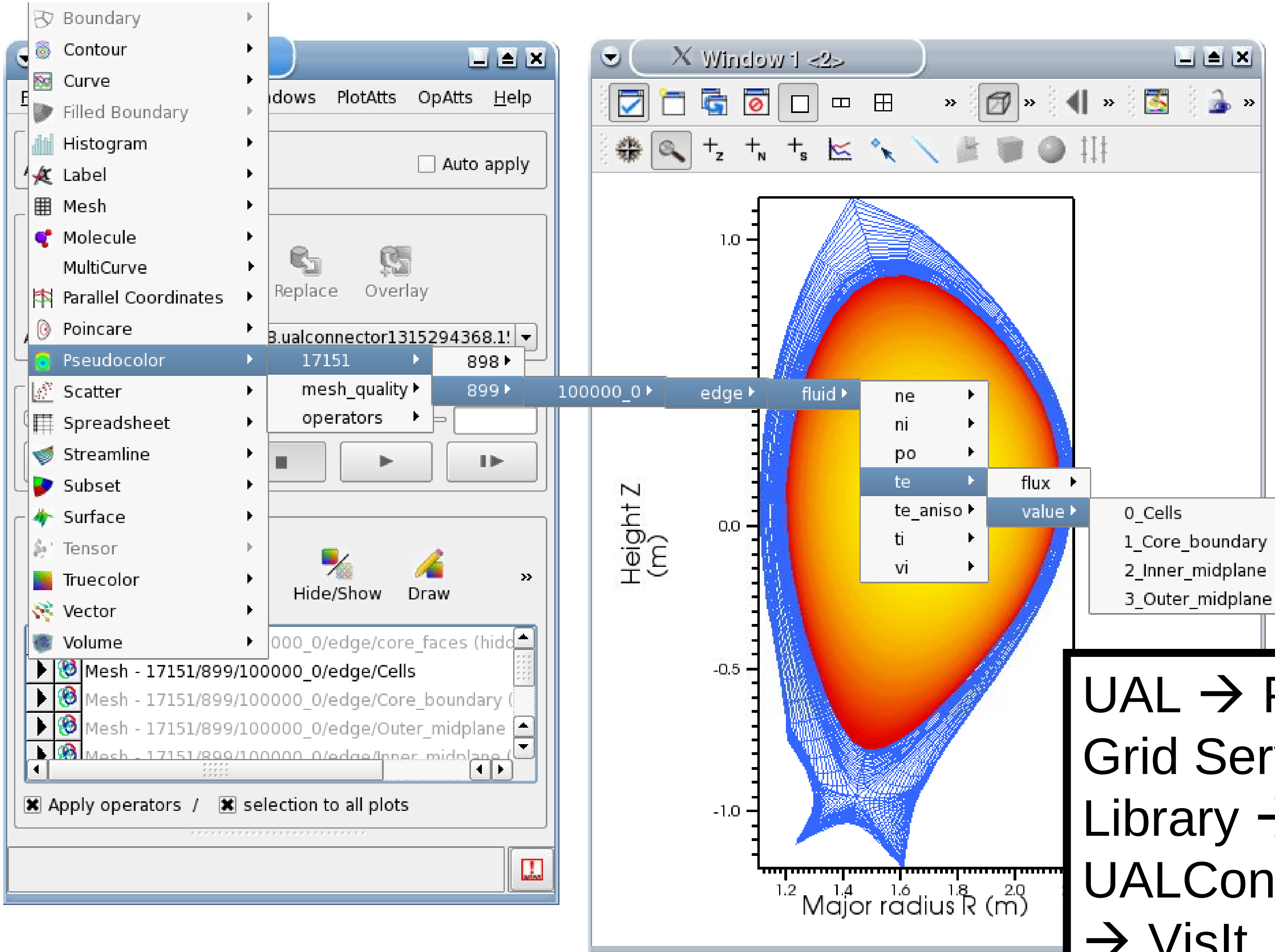
How to use it?
2. Grid & data I/O

```
! Use module from grid service library
use itm_grid_structured

! Write a 2d structured R,Z grid
call gridSetupStructuredSep( &
     & edgecpo%grid, &
     & ndim = 2, &
     & c1 = COORDTYPE_R, x1 = rnodes, &
     & c2 = COORDTYPE_Z, x2 = znodes, &
     & id = '2d structured R,Z grid' )

! Write data on the 2d cells ("faces")
call gridStructWriteData( &
     & edgecpo%grid, &
     & edgecpo%fluid%ne%value(1), &
     & GRID_STRUCT_FACES, celldata )
```

Grid Service Library:

- high-level interface for standard discretizations
- low-level interface helps with assembling and reading complex discretizations

How to use it?
3. General tools



UAL → Python Grid Service Library → UALConnector → VisIt