Converter User Manual

Running the converter tests

Tests are diff-based: the converter produces a deterministic OOFEM `.in` file which is compared against a committed `reference.in`.

Build the converter first (from your build directory):

```
cd ~/build/oofem
make converter.exec
```

Run all converter tests:

```
ctest -R test_converter
```

Run a single test by name:

```
ctest -R test_converter_3DCylinderQhull -V
```

The `-V` flag prints the full output, which is useful for diagnosing failures.

Test layout

Test directories are named `<Case><Mesher>` so the mesher path is obvious from the directory name (e.g. `2DPlateT3D`, `3DFPZQhull`). Each test lives in `src/converter/tests/<TestName>/` and contains:

File	Purpose
`control.in`	Control file template (also the OOFEM input template)
`reference.in`	Committed golden output to diff against
`mesh.nodes` + `mesh.voronoi`	Qhull mesh input (qhull path)
`mesh.t3d`	T3D mesh input (t3d path)
`mesh.sh`	Shell script that runs the full mesh / random-field pipeline (informational, not invoked by ctest)
`random.in` *(optional)*	Genran control file. Present in tests whose `control.in` references a `random.dat` random field via `InterpolatingFunction`. `mesh.sh` invokes `genran.exe random.in` `random.dat` before the converter so the OOFEM-side solve can read the field. `random.dat` and `stat.dat` are gitignored — regenerate locally.

Updating a reference file

After intentionally changing converter output, regenerate the reference from the test directory:

```
cd src/converter/tests/3DCylinderQhull
~/build/oofem/src/converter/converter.exec control.in mesh.nodes mesh.voronoi
cp oofem.in reference.in
```

Then re-run the test to confirm it passes.

Running the converter

The converter needs to know which mesher produced the input. Specify it with either the `--mesher` (or `-m`) CLI flag, or with a `#@mesher t3d|qhull` directive in `control.in`. The CLI flag wins if both are present.

```
# T3D mesh
converter.exec --mesher t3d control.in mesh.t3d

# Qhull mesh (nodes + voronoi)
converter.exec --mesher qhull control.in mesh.nodes mesh.voronoi

# Qhull mesh (nodes + delaunay + voronoi) - legacy 3-file form
converter.exec --mesher qhull control.in mesh.nodes mesh.delaunay mesh.voronoi
```

If `control.in` already declares the mesher (e.g. all the converter tests do), the flag can be omitted:

```
converter.exec control.in mesh.t3d
converter.exec control.in mesh.nodes mesh.voronoi
```

Output is always written to `oofem.in` in the working directory.

Control file `#@` directives

The control file is an OOFEM `.in` template. The converter reads it line by line. Lines that start with `#@` are directives — they either configure the converter or mark an injection point for mesh-derived content, and are **not** written to the output. All other lines (including `#` comments and `#%` regression-check blocks) are passed through verbatim.

Parsing vs consumption

All directives go through a single parser (`Grid::readControlRecords`) regardless of which mesher produced the input. A directive's *applicability* is a property of the consumers — only the qhull writers (`give3DSMOutput`, `give3DTMOutput`) look at `#@notch` / `#@sphereinclusion` / etc., and only the T3D writers look at `#@BC` / `#@LOAD` / etc. A directive from the "wrong" pipeline is silently ignored at write time, not rejected at parse time.

The two tables below split directives by the pipeline that consumes them. Directives in the qhull table have no effect in a T3D run, and vice versa.

Shared directives (both paths)

These are injection markers, replaced in the output by content derived from the mesh.

Directive	Purpose
`#@INSERT_SETS`	Replaced with `set` records for node/element groups referenced by BCs, loads, and export modules
`#@INSERT_LIVELOADS`	Replaced with `NodalLoad`/equivalent records generated from `#@LOAD` requests
`#@INSERT_CROSSSECTION`	Replaced with cross-section records for the active grid type

The counts line (`ndofman ... nelem ... nset ...`) is patched in place once the mesh is processed; it is not an `#@` directive but behaves like one.

Qhull-consumed directives

Consumed by `give3DSMOutput` / `give3DTMOutput` (the qhull writers) and by qhull-pipeline setup. Geometry directives set up the region and localiser; they are consumed, not written out. Parsed but not consumed when the mesher is T3D.

Directive	Arguments	Purpose
`#@grid`	`<type>`	Selects the output generator. Available types: `3dSM`, `3dTM`, `2dSM`, `2dTM`. The 3D writers cover plain, periodic, and (3D-SM only) fibre-bearing analyses; the 2D writers cover plain SM (with optional periodicity via `latticeboundary2d`) and plain TM. A staggered SMTM coupled analysis is expressed as separate templates — one per subproblem — each with its own `#@grid` directive. See `Grid::resolveGridType`.
`#@mesher`	`t3d` | `qhull`	Declares which mesher produced the input. Lets `main.C` dispatch without relying on the CLI `--mesher` flag.
`#@diam`	`<d>`	Nominal grain diameter — must match the `diam` used when the mesh was generated.

Directive	Arguments	Purpose
`#@perflag`	`2 <px> <py>` (2D) or `3 <px> <py> <pz>` (3D)	Periodicity flags per axis (0 = non-periodic, 1 = periodic). For `3dSM` / `2dSM`, any axis = 1 switches the writer into periodic mode (CTLNODE, `lattice3Dboundary` / `latticeboundary2d` for boundary-crossing elements). For `3dTM`, any axis = 1 emits `latticemt3dboundary` for boundary-crossing Voronoi edges (requires the OOFEM-side `Lattice3dboundary_mt` element to be registered). 2D TM has no periodic OOFEM element and so cannot be made periodic.
`#@thickness`	`<t>`	Out-of-plane thickness for 2D `lattice2D` / `latticemt2D` / `latticeboundary2d` elements (`thick` field of the OOFEM input). Defaults to 1.0. Ignored for 3D writers.
`#@ranint`	`<seed>`	Random integer seed. Non-negative values are replaced with `-time(NULL)`.

Directive	Arguments	Purpose
`#@prism`	`<id> box 6 <xmin> <ymin> <zmin> <xmax> <ymax> <zmax> [refine <r>] [edgerefine <re>] [surfacerefine <rs>] [regionrefine <rr>]`	Defines a 3D box-shaped region. The `*refine` tokens exist for mirroring the generator's `mesh.in` — the converter ignores them, but keeping them in sync keeps the two files readable as a pair.
`#@rect`	`<id> box 4 <xmin> <ymin> <xmax> <ymax>`	2D analog of `#@prism` — axis-aligned rectangle. Required for `#@grid 2dSM` / `#@grid 2dTM`; the converter uses it for boundary tests, Voronoi-vertex projection, and the inside-rect element filter. The generator's `#@disk` (solid 2D disk region) has no converter analog — for circular inclusions on the converter side use `#@diskinclusion`.
`#@cylinder`	`<id> line 6 <x1> <y1> <z1> <x2> <y2> <z2> radius <r>`	Defines a cylindrical region with axis from `(x1,y1,z1)` to `(x2,y2,z2)` and the given `<r>`. Used alongside (or instead of) `#@prism` to scope the domain for the Voronoi dual.

Directive	Arguments	Purpose
`#@interfacecylinder`	`<id> line 6 <x1> <y1> <z1> <x2> <y2> <z2> radius <r> [itz <t>]`	Cylindrical inclusion with ITZ halo (axis as for `#@cylinder`). Material classification works as for `#@sphereinclusion`: endpoints both inside -> `inside` material; endpoints straddle the `radius + t/2` boundary -> `interface` material. The `itz` token is optional; if omitted, ITZ defaults to `#@diam`. Use `#@cylinderinclusion` instead when you need to specify `inside`/`interface` material ids directly.
`#@fibre`	`<id> endpoints 6 <x1> <y1> <z1> <x2> <y2> <z2> diameter <d>`	Declares a straight fibre. The converter discretises it into reinforcement nodes at intersections with matrix Voronoi cells, builds `lattice3D` segments along the fibre, and adds `latticelink3D` couplings to the surrounding matrix vertices.

Directive	Arguments	Purpose
`#@notch`	`<id> box {4|6} <coords> (material <m> | delete)`	Axis-aligned notch box. 2D form `box 4 xmin ymin xmax ymax`; 3D form `box 6 xmin ymin zmin xmax ymax zmax`. Two modes: • `material <m>` — material-reassignment mode. Matrix elements whose midpoint falls inside get `crossSect <m> mat <m>` instead of the default. The notch is just a region of softened/alternative material; no boundary discretisation needed on the generator side. • `delete` — element-deletion mode. Matrix elements with midpoint inside the box are dropped entirely (the notch is a physical void). Pair with the generator-side `#@notch` directive so the dual mesh has a clean cell partition along the notch surface. Delete-mode notches also trigger Voronoi-vertex projection: vertices that bound a Voronoi edge crossing the notch surface get snapped onto the nearest notch face, so cross-section polygons (3D SM) and TM nodes hug the boundary. Multiple `#@notch` directives may be given; the first

Directive	Arguments	Purpose
`#@sphereinclusion`	`<id> centre 3 <x> <y> <z> radius <r> itz <t> inside <mi> interface <mif>`	Spherical inclusion with ITZ halo. Effective radius is `r + t/2`. An element whose endpoints both fall inside gets `crossSect mi mat mi`; an element whose endpoints straddle the sphere boundary (one inside, one outside — the ITZ zone) gets `crossSect mif mat mif`. Applied after `#@notch` on the inherited default material. The control file is expected to declare matching `latticecs` / material pairs for both `mi` and `mif`. Use `#@bodyload` to attach a body load to any of the resulting materials.
`#@diskinclusion`	`<id> centre 2 <cx> <cy> radius <r> itz <t> inside <mi> interface <mif>`	2D analog of `#@sphereinclusion` — circular inclusion with ITZ halo. Internally stored as `SphereInclusionSpec` with `cz = 0`; the existing 3D midpoint / straddle classification works for 2D points unchanged because every 2D vertex has `z = 0`.

Directive	Arguments	Purpose
`#@cylinderinclusion`	`<id> line 6 <x1> <y1> <z1> <x2> <y2> <z2> radius <r> itz <t> inside <mi> interface <mif>`	Straight-axis cylindrical inclusion with ITZ halo. The classification uses perpendicular distance from each endpoint to the infinite axis through the two line points; inclu- sion semantics match `#@sphereinclusion`. Use-case: rebar + ITZ + matrix partitioning for corrosion-cracking studies.

Directive	Arguments	Purpose
`#@inclusionfile`	`<path> itz <t> inside <mi> interface <mif>`	Bulk-load inclusions from a packing file produced by `src/aggregate/`. Each `sphere` line in the file becomes a `SphereInclusionSpec` (semantics identical to `#@sphereinclusion`, sharing the directive's `itz` / `inside` / `interface`); each `fibre` line becomes a `Fibre` (semantics identical to `#@fibre`, renumbered to avoid collisions). `ellipsoid` lines trigger a warning — the converter's material-classification logic only handles spheres. Use this directive instead of dozens of inline `#@sphereinclusion` declarations when the packing comes from aggregate.

Directive	Arguments	Purpose
`#@bodyload`	`<mat> <bc_id>`	Per-material body load. Any element whose `crossSect` / `mat` resolves to `<mat>` gets `bodyloads 1 <bc_id>` appended. Decoupled from inclusion directives so each test can place its `StructTemperatureLoad` (or similar BC) on the material it needs — e.g. on the matrix for an eigenstrain load (Wong), or on the interface for an eigendisplacement load (corrosion cylinder). Multiple entries allowed (one per material).

Directive	Arguments	Purpose
`#@couplingflag`	*(no arguments)*	Toggle. When present, each emitted element gets `couplingflag 1 couplingnumber <N> <ids...>` appended after the polycoords block. For SM lattice3D / lattice3Dboundary (at Delaunay lines), the ids are the Voronoi-line cross-section elements; for TM latticemt3D (at Voronoi lines), they are the Delaunay-line cross-section elements. Image-side entries (`outsideFlag == 1`) are swapped for their periodic partner via `givePeriodicElement()`. Used by staggered SMTM analyses (e.g. Wong percolation) where SM and TM subproblems exchange per-element data.

Directive	Arguments	Purpose
`#@rigidarm`	`<master_ctl_id> face <axis> <side> mastermask 6 <m1>..<m6> doftype 6 <d1>..<d6>`	SM-only. Any inside Delaunay vertex on the specified face of the region's bounding box (`axis` in {1,2,3} for x/y/z; `side` in {`min`,`max`}) is emitted as `rigidarmnode ... master <N> mastermask ... doftype ...` instead of a regular `node`, slaved to the controlvertex referenced by `<master_ctl_id>`. The master controlvertex itself is kept as a regular node. Multiple `#@rigidarm` directives may be given (one per face); the first matching one wins. Use-case: cantilever ends clamped as rigid bodies attached to support/load control points.

Directive	Arguments	Purpose
`#@slaveside`	`<master_ctl_id> face <axis> <min|max> dofs <list>`	SM-only. Like `#@rigidarm` but uses `DT_simpleSlave` (identical-value slaving) instead of rigid-arm kinematics — the slave DOF takes exactly the master's value, with no rotation transfer. Each inside Delaunay vertex on the specified face is emitted as `node ... dofidmask ... doftype <flags> mastermask <ids>`, slaved to the controlvertex `<master_ctl_id>`. Only the DOFs listed after `dofs` are slaved (entries: `1`=D_u, `2`=D_v, `3`=D_w, `4`=R_u, `5`=R_v, `6`=R_w; for 2D lattice nodes the writer emits `dofidmask 3 1 2 6` and slaves the listed subset). The master controlvertex itself is kept as a regular node. Works for both 2D (`#@rect`) and 3D (`#@prism`) regions; multiple `#@slaveside` directives may be combined (one per face/master). Use-case: pulling one face of a non-periodic specimen uniformly in a single direction (e.g. direct-tension test where the right face follows a load-controlled master node).

Directive	Arguments	Purpose
`#@material_around`	`<ctl_id> material <m>`	SM-only. Any Delaunay line with either endpoint equal to the named controlvertex gets material `<m>`. Precedence: `#@material_around` < `#@notch` < `#@sphereinclusion` / `#@cylinderinclusion` — later rules override earlier ones. Use-case: cantilever lattice lines attached to the support/load points pick up the elastic material rather than the default matrix material.

Directive	Arguments	Purpose
`#@controlvertex`	`<id> coords {2|3} <x> <y> [<z>]`	Declares a specific mesh-node location whose nearest Delaunay-vertex id is exposed at write time via the inline placeholder `#@CTL<id>`. Must also appear as a `controlvertex` entry in the generator's `mesh.in` so the mesher seeds a node at that coordinate. The coord arity matches the grid's dimension. Typical use: naming support/load/monitor nodes whose ids feed into `hpc`, `OutputManager`, BC sets, and check rules.
`#@CTL<id>`	*(inline placeholder)*	Substituted on each non-directive line with the mapped OOFEM node id (compact in non-periodic mode, raw in periodic mode). Longer ids substitute first so that e.g. `#@CTL12` isn't shadowed by `#@CTL1`.

Directive	Arguments	Purpose
`#@CTLNODE`	*(inline placeholder, not a line directive)*	Substituted with the numeric id of the periodic control node before each non-directive line is written. Use anywhere the OOFEM template needs that id (e.g. `hpc 2 #@CTLNODE 2 hpcw 1 1.`, `OutputManager tstep_all dofman_output {#@CTLNODE}`, `#NODE number #@CTLNODE dof 2 unknown d`). Inert when `perflag` is fully non-periodic.
`#@pov`	*(no arguments)*	Opt in to writing the auxiliary POV-Ray rendering files (`*.vor.line.pov`, `*.vor.cross.pov`, `*.del.line.pov`, `*.del.cross.pov`). Default off — POV files are not generated unless this directive is present.
`#@vtk`	*(no arguments)*	Opt in to writing the ParaView `.vtu` files (`*.voronoielement.vtu`, `*.delaunayelement.*.vtu`, `*.fibre.beamElement.vtu`, etc.). Default off — VTK files are not generated unless this directive is present.

T3D-consumed directives

Consumed by the T3D writers (`giveOutputT3d` / `writeT3dNodesOofem` / `writeT3dElemsOofem`). These directives describe BCs, loads, and section data attached to named T3D entities (vertices, curves, surfaces). Parsed but not consumed when the mesher is qhull.

Directive	Arguments	Purpose
`#@BC`	`<entType> <entID> <dofMask> <values...>`	Dirichlet BC on a T3D entity. Generates `BoundaryCondition` records and a matching set.
`#@LOAD`	`<entType> <entID> <q> [ltf <ltfID>]`	Distributed load on a T3D entity. Emitted via `#@INSERT_LIVELOADS`.
`#@DIR`	`<entID> <dx> <dy> <dz>`	Direction vector associated with an entity (used for shell/beam orientation).
`#@THICKNESS`	`<entID> <t>`	Shell/plate thickness for an entity.
`#@3DSECTION`	`<args>`	3D section data for shell/beam elements.
`#@SHELLWIDTHSCALE`	`<args>`	Per-edge width scaling for shell elements.

Directive	Arguments	Purpose
`#@element`	`<entityKind>` `<entityID>` `<elementName>` `<crossSect> <mat>`	Per-region element override for the T3D writer. `<entityKind>` is one of `vertex`, `curve`, `surface`, `patch`, `shell`. Edges classified to the given entity emit `<elementName>` instead of the writer's hardcoded default (`lattice3D`, `lattice3d`, or `lattice3Dnl` depending on geometry). Edges that don't match any directive use the default. Resolution priority for an edge that touches multiple entities: curve > surface > region.

`<entType>` uses the numeric codes from `Grid::entityTypeFromString`: 1 vertex, 2 curve, 3 surface, 5 patch, 6 shell.

`2dSM` writer — 2D structural mechanics

`#@grid 2dSM` dispatches to `Grid::give2DSMOutput`, which emits one `lattice2D` element per Delaunay edge inside the rect (or `latticeboundary2d` for periodic-crossing edges). The OOFEM line format is

```
lattice2D <id> nodes 2 <n1> <n2> crossSect <m> mat <m>
          gpCoords 2 <gx> <gy> width <w> thick <t>
```

with `gpCoords` the midpoint of the lattice element, `width` the length of the dual Voronoi cross-section edge, and `thick` the out-of-plane thickness from `#@thickness`. The companion test domain type is `domain 2dlattice` (or `2dLattice`).

Periodic mode (any axis of `#@perflag` set) appends a CTLNODE whose coordinates are the specimen dimensions and whose DOFs hold the macro strains:

```
node <ctlNode> coords 2 <lx> <ly> dofidmask 3 31 32 42
```

Boundary-crossing Delaunay edges are emitted as

```
latticeboundary2d <id> nodes 3 <inside> <partner> <ctlNode>
                  crossSect <m> mat <m> gpCoords 2 <gx> <gy>
                  width <w> thick <t> location <code>
```

where `<partner>` is the periodic partner of the outside endpoint and `<code>` is the 1..8 compass-direction shift code consumed by `Lattice2dBoundary::giveSwitches`.

`2dTM` writer — 2D mass transport

`#@grid 2dTM` dispatches to `Grid::give2DTMOutput`. Mirrors `2dSM` with Voronoi/Delaunay roles swapped: nodes are Voronoi vertices, each `latticemt2D` element is a Voronoi edge, the cross-section width is the length of the dual Delaunay edge:

```
latticemt2D <id> nodes 2 <n1> <n2> mat <m> dim 1
            thick <t> width <w> gpCoords 2 <gx> <gy>
```

The companion domain type is `domain 2dMassLatticeTransport`. 2D TM periodicity is **not implemented** — OOFEM has no `Lattice2dBoundary_mt` element. Setting `#@perflag` periodic with `#@grid 2dTM` still emits `latticemt2D` for inside edges; boundary-crossing edges are dropped.

Voronoi-vertex projection at boundaries

For 2D and for delete-mode 3D notches, the converter projects Voronoi vertices that bound a *crossing* Voronoi edge onto the nearest face of the rect / notch (`Grid::project2DVoronoiVerticesToBoundaries` and `project3DVoronoiVerticesToNotches`). This serves two purposes:

- **Transport nodes land on the surface.** TM elements that cross a boundary become elements with one inside endpoint and one on-the-boundary endpoint, matching the 3D outer-boundary behaviour of `Prism::modifyVoronoiCrossSection`.
- **SM cross-sections are clipped at the surface.** The dual Voronoi edge length used as `lattice2D width` reflects the part inside the specimen rather than extending past it.

Selectivity matters: only Voronoi vertices that bound a crossing edge are projected. Vertices "deep outside" (only connected to other outside vertices) keep their qhull positions and are filtered out at emission time. A blanket "clamp every outside vertex" pass would collapse far-out vertices onto the same rect corner and create coincident TM nodes.

For notches, projection only fires for `#@notch ... delete` boxes — material-mode notches are pure material overrides and don't represent a physical surface,

so projecting Voronoi vertices for them would distort polygons without a coherent boundary to snap to.

3D periodic mass transport — `latticemt3dboundary`

`#@grid 3dTM` with `#@perflag` periodic emits `latticemt3dboundary` elements for boundary-crossing Voronoi edges. This requires the OOFEM-side `Lattice3dboundary_mt` element registered under the `latticemt3dboundary` keyword (in `src/tm/Elements/LatticeElements/`). The converter test `tests/3DPerQhullTM/` exercises the full pipeline.

Random-field pipeline (`mesh.sh` + genran)

OOFEM `InterpolatingFunction` records can read a regular-grid random field from `random.dat`. To keep the full pipeline reproducible from the test directory, tests that consume `random.dat` ship a `random.in` genran control file, and `mesh.sh` runs genran between the qhull and converter steps:

```
~/build/oofem/src/generator/generator.exec mesh.in
mv nodes.dat mesh.nodes
qvoronoi p Fv < mesh.nodes > mesh.voronoi
~/Software/genran.git/genran.exe random.in random.dat
~/build/oofem/src/converter/converter.exec control.in mesh.nodes mesh.voronoi
```

Both `random.dat` and the `stat.dat` summary genran writes are gitignored — they're build artefacts, regenerated from `random.in` each run. See `tests/3DCylinderQhull/` for the canonical example.

Unified `3dSM` writer — conventions for matrix, fibres, and links

`#@grid 3dSM` dispatches to a single writer (`Grid::give3DSMOutput`) that handles plain SM, periodic SM, and periodic SM with fibres, driven entirely by `#@perflag` and the presence of `#@fibre` records. The previously separate `3dFPZ`, `3dFPZFibre`, `3dFibreBenchmark`, `3dGopSha`, `3dTension`, `3dSphere`, and `3dCylinder` grid types have been retired; their analyses are expressed entirely as `control.in` templates using `#@notch` / `#@sphereinclusion` / `#@cylinderinclusion` for per-region material overrides and `#@controlvertex` / `#@CTL<id>` for support/load points.

For the fibre case the writer emits three classes of element, all of them `lattice3D` family, distinguished by their cross-section reference. The control file is expected to declare three `latticecs` records and three materials in this exact slot order:

Slot	Cross-section / material	Element kind	Geometry source
1	`latticecs 1 material 1` `shape 2latticedamage 1` `...` (or any matrix material)	matrix `lattice3D` / `lattice3Dboundary`	polygonal `polycoords` from qhull Voronoi facet

Slot	Cross-section / material	Element kind	Geometry source
2	`latticecs 2 material 2 shape 1 radius <r>latticelinearelastic 2 ...` (or any fibre material)	fibre segment `lattice3D` / `lattice3Dboundary` (Timoshenko frame)	circular cross-section, `radius` taken from `latticecs`
3	`latticecs 3 material 3latticeslip 3 ...` (or any bond material)	`latticelink3D` / `latticelink3Dboundary` (fibre<->matrix coupling)	`length`, `diameter`, `dirvector`, `L_end` written by the converter

The counts header in `control.in` must read `ncrosssect 3 nmat 3 ...` whenever fibres are present (drop to 1 for plain SM). The writer prepends `ndofman` and `nelem` automatically.

Reinforcement-node ids are offset by the raw Delaunay-vertex count `N_DelV`: the i-th reinforcement node is written as `node (N_DelV + i)`. The periodic control node id is `N_DelV + N_reinf + 1`, available throughout the template via the `#@CTLNODE` placeholder.

Unified `3dTM` writer — plain and periodic

`#@grid 3dTM` dispatches to `Grid::give3DTMOutput`, which handles plain and periodic mass transport on the Voronoi dual mesh. When `#@perflag` has any periodic axis, the writer pins the first emitted Voronoi vertex (`bc 1 1`), emits `latticemt3Dboundary` for lines whose endpoints straddle a periodic face (with image-side endpoints swapped for their mirror partners and a `location 2 ...` suffix), and appends a control node with `ndofs 3 dofIDmask 3 1 2 3 bc 3 1 1 2` — available in the template via `#@CTLNODE`. Non-periodic behaviour is unchanged (compact 1..N vertex ids, no control node, no boundary elements). Per-element material is resolved through `#@notch` / `#@sphereinclusion` / `#@cylinderinclusion` just like the SM writer, and `#@bodyload` / `#@couplingflag` apply symmetrically.

Adding a new analysis type

The long-term goal is that supporting a new analysis type means writing a new `control.in`, not touching C++. When adding a directive:

1. Register it in the unified `Grid::readControlRecords` parser. The parser is shared; there is no per-pipeline parser to edit.
2. Wire it into the writer(s) that should consume it — one of the qhull writers (`give3DSMOutput` / `give3DTMOutput`), the T3D writers, or both.

3. Add it to the qhull-consumed or T3D-consumed table above, depending on where you wired it up. A directive consumed by both pipelines goes in a shared row — none exist today, but nothing in the parser prevents that.

Keep `mesh.in` and the `#@prism`/`#@cylinder`/`#@fibre` etc. lines in `control.in` mirrored so that readers can see the mesh parameters in both files. Prefer extending `give3DSMOutput` / `give3DTMOutput` over adding a new grid type.