Generator User Manual

The generator reads a `#@`-directive control file describing a geometric region and produces a point cloud (`nodes.dat`) of quasi-random vertices whose spacing is governed by a target diameter. The output is typically fed into `qvoronoi` to produce the Voronoi + Delaunay files consumed by the converter.

Running the generator tests

Tests are diff-based: the generator produces a deterministic `nodes.dat` which is compared against a committed `reference.nodes`.

Build the generator first (from your build directory):

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

Run all generator tests:

```
ctest -R test_generator
```

Run a single test by name:

```
ctest -R test_generator_3DCylinderQhull -V
```

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

Test layout

Test directories live under `src/generator/tests/<TestName>/` and each contains:

File	Purpose
`mesh.in`	Control file with `#@` directives
`reference.nodes`	Committed golden output to diff against

The `nodes.dat` the test produces is written in-place in the test directory (and is ignored by `.gitignore`).

Determinism

The generator reseeds its RNG from `#@ranint <n>`. Negative values are used verbatim (deterministic output, required for regression tests); non-negative values are replaced by `-time(NULL)` at startup (fresh output every run, usable in production). All committed `reference.nodes` files correspond to `mesh.in` configurations with `#@ranint <negative>`.

Updating a reference file

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

```
cd src/generator/tests/3DCylinderQhull
~/build/oofem/src/generator/generator.exec mesh.in
mv nodes.dat reference.nodes
```

Then re-run the test to confirm it passes.

Running the generator

```
generator.exec <input-file>
```

The input file must be a `#@`-directive control file (see below). The output filename comes from `#@output` in that file.

Control file `#@` directives

The control file is read line by line. Lines that start with `#@` are directives. All other lines — including blank lines, plain `#` comments, and descriptive text — are ignored. Directive order is irrelevant: the parser builds up state as it goes and runs validation at the end (`#@diam` and `#@output` are required; `#@perflag` must have three components if present).

Top-level directives

Directive	Arguments	Purpose
`#@output`	`<filename>`	Output path for the generated point list. Required.
`#@domain`	`<n>`	Dimension of the problem (`2` or `3`). 2D writes `nodes.dat` with a leading `2` and `x y` per line; 3D writes a leading `3` and `x y z`. Defaults to 3 if omitted.
`#@diam`	`<d>`	Target nominal spacing between points. Required. Also sets the geometric tolerance `TOL = 1e-6 * d`.

Directive	Arguments	Purpose
`#@maxiter`	`<n>`	Maximum number of random-placement attempts before aborting.
`#@ranint`	`<n>`	RNG seed. Negative = deterministic; non-negative is replaced by `-time(NULL)` at runtime.
`#@perflag`	`2 <px> <py>` (2D) or `3 <px> <py> <pz>` (3D)	Periodicity flags per axis (0 = non-periodic, 1 = periodic). The 2D form is internally padded with `pz = 0` so all downstream code can address `at(3)` safely.
`#@ranflag`	`<n>`	Random-placement strategy. `0` Bolander (no points on boundary, best for pure SM), `1` Grassl (points on boundary, needed for coupled analyses), `2` Grassl with periodic boundary pairing.
`#@vtk`	*(no arguments)*	Opt in to writing `points.vtk` alongside `nodes.dat` for ParaView inspection. Default off.

Directive	Arguments	Purpose
`#@inclusionfile`	`<path> [itz <t>] [refine <r>]`	Bulk-load inclusions from a packing file produced by `src/aggregate/`. Each `sphere` line becomes an `InterfaceSphere`; each `disk` line (2D) becomes an `InterfaceDisk`. The supplied `itz` and `refine` are applied uniformly. Inclusions are renumbered automatically to avoid collisions with other directives. `ellipsoid` lines trigger a warning (arbitrary-orientation ellipsoid seeding is not implemented); `fibre` lines are silently ignored (handled by the converter, not the generator).

Geometry directives

Geometry directives declare the volumes, surfaces, and lines that points are placed on. Each directive takes a 1-based numeric id; ids within a category (vertex, controlvertex, curve, surface, region, inclusion, refinement) must be unique but don't need to be contiguous.

Directive	Arguments	Purpose
`#@vertex`	`<id> coords {2|3} <x> <y> [<z>] [refine <r>] [radius <r>]`	A single point-source vertex. Typically used to declare bounding-box corners that widen the spatial localiser for periodic partner lookup. The coord arity matches `#@domain`.
`#@controlvertex`	`<id> coords {2|3} <x> <y> [<z>] [refine <r>] [radius <r>]`	A named vertex the converter can reference later (via `#@controlvertex` on its side) for support or load placement. Separate numbering space from `#@vertex`.
`#@curve`	`<id> vertices <n> <v1> ... [refine <r>] [normal 3 <nx> <ny> <nz>]`	Piecewise-linear curve defined by vertex ids (referring to `#@vertex` ids). Points are placed along the curve with spacing `diam * refine`.
`#@surface`	`<id> curves <n> <c1> ... [refine <r>] [normal 3 <nx> <ny> <nz>] [boundaryflag <f>] [boundaryshift 3 <dx> <dy> <dz>]`	Planar (or nearly planar) patch bounded by curves. `boundaryflag 1 boundaryshift ...` makes the surface periodic with a mirror-image partner offset by the shift vector.

Directive	Arguments	Purpose
`#@prism`	`<id> box 6 <xmin> <ymin> <zmin> <xmax> <ymax> <zmax> [refine <r>] [edgerefine <re>] [surfacerefine <rs>] [regionrefine <rr>]`	3D axis-aligned box region. The three per-stage refine factors (`edgerefine`, `surfacerefine`, `regionrefine`) apply to the spacing used when seeding points on region edges, on region faces, and in the region interior, respectively.
`#@rect`	`<id> box 4 <xmin> <ymin> <xmax> <ymax> [refine <r>] [edgerefine <re>] [regionrefine <rr>]`	2D analog of `#@prism` — axis-aligned rectangle region. Edge points are seeded at `edgerefine·diam` spacing along all four edges; the interior is then filled by random placement at `regionrefine·diam` spacing. (No surface refinement parameter — 2D regions have no faces.)
`#@cylinder`	`<id> line 6 <x1> <y1> <z1> <x2> <y2> <z2> radius <r> [refine <r>]`	Solid cylinder region whose axis runs between two points.
`#@sphere`	`<id> centre 3 <x> <y> <z> radius <r> [refine <r>]`	Solid sphere region.

Directive	Arguments	Purpose
`#@disk`	`<id> centre 2 <cx> <cy> radius <r> [refine <r>]`	2D analog of `#@sphere` — solid disk region. Seeds the centre, a random ring of points on the circumference, and a random interior fill (with mirror across the boundary).
`#@intersphere`	`<id> centre 3 <x> <y> <z> radius <r> [refine <r>] [itz <t>]`	Spherical inclusion that places points on its surface plus an ITZ halo of thickness `itz`. Used to form rebar / aggregate / ITZ triples for the converter.
`#@interfacedisk`	`<id> centre 2 <cx> <cy> radius <r> [refine <r>] [itz <t>]`	2D analog of `#@intersphere` — circular inclusion with ITZ halo. Places a ring of points on the circle plus a sister ring at `radius + itz`.
`#@interfacecylinder`	`<id> line 6 <x1> <y1> <z1> <x2> <y2> <z2> radius <r> [refine <r>] [itz <t>]`	Cylindrical inclusion with ITZ halo (straight axis through the two line points).
`#@interfaceplane`	`<id> line 6 <x1> <y1> <z1> <x2> <y2> <z2> diameter <d> [refine <r>] [itz <t>]`	Planar interface with ITZ halo.
`#@interfacesurface`	`<id> curves <n> <c1> ... [refine <r>]`	Curve-bounded interface surface.

Directive	Arguments	Purpose
`#@refineprism`	`<id> box 6 <xmin> <ymin> <zmin> <xmax> <ymax> <zmax> [refine <r>]`	Local refinement box — inside this box the target spacing becomes `diam * refine`, overriding the enclosing region's refinement factor.

Directive	Arguments	Purpose
`#@notch`	`<id> box {4|6} <coords> [edgerefine <re>] [surfacerefine <rs>]`	Axis-aligned notch box (2D form `box 4 xmin ymin xmax ymax`, 3D form `box 6 xmin ymin zmin xmax ymax zmax`). The generator suppresses point placement strictly inside the box (boundary points are kept) and seeds vertices on the box surface — corners + edges (at `edgerefine·diam` spacing) + faces (at `surfacerefine·diam` spacing in 3D) — so the dual mesh partitions cleanly along the notch boundary. The companion converter-side `#@notch ... delete` directive then removes elements with midpoint inside the same box. Use this when the notch represents a physical void; for pure material reassignment (no real boundary) only the converter-side `#@notch ... material <m>` directive is needed and no generator-side declaration is required.

Directive	Arguments	Purpose

Refinement factors

All `refine`-like fields scale the baseline target spacing `diam`. A `refine 0.5` on a region makes points there roughly half as far apart as the baseline `diam`. `#@prism`'s per-stage `edgerefine` / `surfacerefine` / `regionrefine` allow finer control: typical values are something like `edgerefine 0.5 surfacerefine 0.7 regionrefine 1.` to over-sample region edges and surfaces relative to the interior.

Periodicity

`#@perflag 3 px py pz` selects periodic wrap per axis. When any axis is periodic, the generator emits mirror-image partners along that axis. Typical use with regions:

- Non-periodic run: `#@perflag 3 0 0 0` (`ranflag 1`)
- Fully periodic run: `#@perflag 3 1 1 1` (`ranflag 2` or `0`)
- Mixed-periodicity specimens: set the non-periodic axes to 0.

A periodic region normally also needs `#@vertex` entries at the widened bounding-box corners so the octree localiser can find cross-boundary neighbours.

2D meshes

For a 2D run set `#@domain 2`, `#@perflag 2 px py`, declare the specimen with `#@rect`, and use `coords 2 x y` for `#@vertex` / `#@controlvertex`. Internally the generator stores every vertex as 3D with `z = 0` so the spatial localiser, vertex list, and downstream plumbing are unchanged; only `nodes.dat` strips the `z` column. The 2D output looks like

```
2
432
-1.0000000000000000e-02 -1.0000000000000000e-02
...
```

— qvoronoi consumes this directly and produces a 2D Voronoi tessellation.

2D directive analogues:

3D	2D
`#@prism`	`#@rect`
`#@sphere`	`#@disk`
`#@intersphere`	`#@interfacedisk`

(`#@interfacecylinder`, `#@interfaceplane`, `#@interfacesurface`, `#@cylinder` have no 2D analogues yet — they are 3D-only.)

A minimal 2D specimen:

```
#@output nodes.dat
#@domain 2
#@diam 0.008
#@maxiter 100000
#@perflag 2 0 0
#@ranint -1
#@ranflag 1
#@vertex 1 coords 2 -0.01 -0.01 refine 1. radius 0.
#@vertex 2 coords 2  0.11  0.06 refine 1. radius 0.
#@controlvertex 1 coords 2 0.0  0.025
#@controlvertex 2 coords 2 0.10 0.025
#@rect 1 box 4 0. 0. 0.10 0.05 refine 1.
```

This is `tests/2DRectQhull/mesh.in`. Add `#@interfacedisk` lines for aggregate inclusions, or a `#@notch box 4 ...` for a void.

Notches

The `#@notch` directive declares a region the generator should treat as a physical void:

```
#@notch 1 box 4 0.046 0.0 0.054 0.025 surfacerefine 0.5 edgerefine 0.5
```

Effect at generation time:

1. Vertices that would be placed strictly inside the notch box are suppressed (`addVertex` returns false for them).
2. Vertices are explicitly seeded on the notch surface — corners + edges at `edgerefine·diam` spacing (and, in 3D, faces at `surfacerefine·diam` spacing). This ensures the dual mesh has a clean cell partition along the notch surface, which the converter-side `#@notch ... delete` then deletes elements crossing.

Pair with the converter-side `#@notch <id> box {4|6} ... delete` to get a full-pipeline void. For pure material reassignment (notch behaves as a soft material region rather than a void) the generator side is not needed — only the converter declares `#@notch ... material <m>`.

Loading inclusions from a packing file

For mesoscale specimens with many inclusions it is often impractical to declare each one with an inline `#@intersphere`. Instead, run the aggregate placer (see `src/aggregate/`) to produce a `packing.dat` and reference it from the generator's control file:

```
#@inclusionfile packing.dat itz 0.5e-3 refine 0.5
```

Per-line semantics:

Packing-file keyword	Generator action
`sphere`	Instantiate an `InterfaceSphere` with the line's centre and radius and the directive's `itz` and `refine`.
`disk` (2D)	Instantiate an `InterfaceDisk` with the line's 2D centre and radius and the directive's `itz` and `refine`.
`ellipsoid`	Print a warning and skip — the generator does not yet implement arbitrary-orientation ellipsoid surface seeding. Generate spheres in aggregate via `#@grading shape sphere` to avoid this.
`fibre`	Silently ignored; fibres are read by the converter (for beam/link element generation), not by the generator.

A self-contained example lives in `src/generator/tests/3DInclusionFile/`, which uses a small hand-written `packing.dat` so the test does not depend on the aggregate binary.

Example

```
# 3D sphere with elastic inclusion for hydraulic-fracture analysis
#@output nodes.dat
#@domain 3
#@diam 12.e-3
#@maxiter 10000
#@ranint -1
#@perflag 3 0 0 0
#@ranflag 1
# bounding-box corners widen the localiser
#@vertex 1 coords 3 -116.e-3 -116.e-3 -116.e-3 refine 1. radius 0.
#@vertex 2 coords 3  116.e-3  116.e-3  116.e-3 refine 1. radius 0.
#@controlvertex 1 coords 3 0. 0. 0.
#@sphere 1 centre 3 0. 0. 0. radius 58.e-3 refine 0.7
#@intersphere 1 centre 3 0. 0. 0. radius 8.e-3 refine 0.5 itz 0.5e-3
```

This is `src/generator/tests/3DSphereQhull/mesh.in`. A matching `src/converter/tests/3DSphereQhull/control.in` template consumes the Voronoi tessellation built from the resulting `nodes.dat` to produce the final OOFEM `.in`.

Adding a new directive

1. Add a `#@<tag>` branch to `Grid::readControlRecords` in `grid.C`.
2. If the directive targets a new entity type, add a component class with an `initializeFromTokens(std::istringstream &)` method (match the pattern used by `Vertex` / `Prism` / `Sphere`).
3. Add an entry to the table above.
4. If it affects point generation, extend the appropriate generator routine (e.g. `Prism::generatePoints`) — the `#@` layer only reads directives into Grid state; actual point placement logic stays where it already is.

Do not reintroduce the old `initializeFrom(GeneratorInputRecord &)` path — the OOFEM-style reader layer was removed in favour of the unified `#@` parser.