Aggregate User Manual

The aggregate module produces a periodic packing of ellipsoidal aggregates and straight fibres inside a rectangular RVE. It reads a `#@`-directive control file, runs a sequential trial-and-error placer, and writes a `packing.dat` file (and optionally a ParaView-loadable `.vtu`) describing where each inclusion sits and how it is oriented. The packing file is the input to downstream tools — typically the converter, which couples it with a node mesh from the generator to produce a complete OOFEM `.in`.

The module is written from scratch in C++, replacing an earlier Matlab implementation. The numerical parts (Wriggers' interval-volume formula, the Alfano-Greer ellipsoid–ellipsoid test, the trial-and-error placement loop) are faithful ports; the random-orientation sampler was upgraded from the Matlab mixed convention to Shoemake's uniform-on-SO(3) quaternion method.

Running the aggregate tests

Tests are diff-based: `aggregate.exec` produces a deterministic `packing.dat` (and optionally `packing.vtu`) which is compared against committed reference files.

Build the executable first (from your build directory, after `cmake -DUSE_AGGREGATE=ON`):

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

Run all aggregate tests:

```
ctest -R test_aggregate
```

Run a single test by name:

```
ctest -R test_aggregate_SmallRVE -V
```

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

Test layout

There are two kinds of tests, both registered automatically by `src/aggregate/CMakeLists.txt`.

Unit tests are C++ executables that link against `libaggregate` and exercise individual components in isolation:

Test	Source	Covers
`test_aggregate_intersection_unit`	`intersection_test.C`	Alfano-Greer ellipsoid pairs, plane-axis intersection, fibre-ellipsoid intersection
`test_aggregate_grading_unit`	`grading_test.C`	`dlim` edge cases, `GradingCurve` ordering and determinism
`test_aggregate_placer_unit`	`placer_test.C`	ghost-shift counts, sparse placement, deterministic seed, fibre placement

End-to-end tests live under `src/aggregate/tests/<TestName>/` and each contains:

File	Purpose
`control.in`	Control file with `#@` directives
`reference.dat`	Committed golden packing file to diff against
`reference.vtu`	*(optional)* Committed VTU file to diff against if `#@vtu` is set

The driver (`tests/run_test.cmake`) runs `aggregate.exec` on `control.in`, compares `packing.dat` against `reference.dat`, and — only if `reference.vtu` exists in the test directory — also compares `packing.vtu` against it. Any test directory that does not declare a VTU reference simply skips the second check.

Determinism

Reproducibility comes from three places:

- The RNG engine is `std::mt19937` (cross-implementation deterministic).
- Sampling is done via `aggregate::uniform()` in `random.h`, which computes `[0,1)` from the raw `uint32` output rather than going through `std::uniform_real_distribution` (the standard does not pin the latter's output across implementations).
- `Box::writePackingFile` uses `std::scientific` with `max_digits10` precision, so doubles round-trip losslessly to the packing file.

Together these guarantee byte-identical output across builds with the same compiler / libstdc++. Across radically different math libraries the text may differ in the last digit; in that case switch the comparison to a relative tolerance or move to invariant-based testing.

Updating a reference file

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

```
cd src/aggregate/tests/SmallRVE
~/build/oofem/src/aggregate/aggregate.exec control.in
cp packing.dat reference.dat
# only if the test has a #@vtu directive:
cp packing.vtu reference.vtu
```

Then re-run the test to confirm it passes.

Running aggregate

```
aggregate.exec <control-file>
```

The control file's `#@output` directive names the packing file; `#@vtu` optionally adds a VTU file beside it.

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.

Top-level directives

Directive	Arguments	Purpose
`#@output`	`<filename>`	Packing-file output path. Required.
`#@vtu`	`<filename>`	Optional VTU output path. Omit to skip ParaView export.
`#@box`	`2 <lx> <ly>` (2D) or `3 <lx> <ly> <lz>` (3D)	RVE side lengths. The leading count selects the dimension — 2D mode places disks, 3D mode places ellipsoids/spheres and (optionally) fibres. Required.

Directive	Arguments	Purpose
`#@periodicity`	`2 <px> <py>` (2D) or `3 <px> <py> <pz>` (3D)	Per-axis periodicity flag. `1` = periodic (the placer generates ghosts and allows boundary crossings); `0` = real boundary (candidates that touch the boundary are rejected). The arity must match `#@box`.
`#@seed`	`<n>`	Seed passed to `std::mt19937`. The same seed always produces the same packing.
`#@maxiter`	`<n>`	Maximum number of trial placements per inclusion before giving up. Defaults to 10000.
`#@grading`	`dmin <d>  dmax <d>  fraction <f> [shape sphere|ellipsoid]`	Aggregate sieve curve parameters. Optional — omit if no aggregates are wanted. In 3D, `shape ellipsoid` (default) samples random aspect-ratio ellipsoids; `shape sphere` collapses each aggregate to `sx = sy = sz = r`. In 2D the `shape` token is ignored and disks are sampled (single radius per aggregate, written as `disk` lines in `packing.dat`).
`#@fibres`	`fraction <f>  length <l>  diameter <d>`	Fibre placement parameters. Optional — omit if no fibres are wanted. **3D only** — fibres are not supported in 2D (a 2D textile cross-section is a circle, modelled via `#@grading` disks). The number of fibres is computed as `floor(box_volume · fraction / (π · (d/2)² · l))`.

`#@grading` semantics

The grading curve uses Wriggers' interval-volume formula on a Fuller-style distribution truncated below `dmin` and above `dmax`. Because the distribution is truncated, the *modelled* volume is less than the nominal `fraction · box_volume` — the missing piece (everything below `dmin`) is implicitly attributed to the matrix. Don't be surprised if the placed-aggregate volume is, e.g., 50–70% of `fraction · box_volume`.

`dmax` must be at least `2·dmin` (one full sieve octave). Each ellipsoid sampled inside an interval respects `sx ≤ sy ≤ sz` and an aspect ratio `sx/sz ∈ [0.5, 1]`. With `shape sphere` the constraints collapse to a single radius

`r ∈ [n/2, m/2]` per sieve interval `[n, m]`, and the output line becomes `sphere <id> centre 3 ... radius <r>` instead of the six-parameter ellipsoid form. Spherical aggregates are particularly useful when the downstream consumer (typically the generator) only implements surface seeding for spheres — see `tests/SmallRVEsphere/` for an example.

2D mode

`#@box 2 lx ly` switches the module into 2D-disk mode. The grading curve samples a single radius per inclusion (uniform in `[lower/2, upper/2]` per sieve interval), the placer uses an analytic distance-based overlap test instead of the 3×3 Alfano-Greer eigenvalue path, and the packing file writes `disk` lines:

```
# packing v1
# box 0.05 0.05
# periodicity 1 1
disk 1 centre 2 0.0298 0.0078 radius 0.00687
disk 2 centre 2 0.0050 0.0029 radius 0.00488
...
```

The 2D path is the typical entry point for textile-reinforced concrete modelling (where the textile cross-sections are circular). See `tests/2DRVE/` for a 50×50 mm doubly-periodic example and the matching generator-side `#@inclusionfile` consumer.

`#@fibres` semantics

The placer determines an integer fibre count from the requested volume fraction, then attempts to place each one against existing aggregates. **Fibre-fibre overlaps are not checked** (matching the Matlab convention) — only fibre-vs-ellipsoid intersections cause rejection.

When `#@grading` and `#@fibres` are both present, the placer follows the Matlab three-stage order:

1. Place "group 1" aggregates (the large end, above the `dlim`-derived volume threshold) — these go down before fibres so the big inclusions have first pick of the volume.
2. Place fibres against the now-occupied aggregates.
3. Place "group 2" aggregates (the small end) into the remaining space, avoiding both ellipsoids and fibres.

The split point is `dlim · α` where `α = 1.5` and `dlim` is computed analytically from `aggregateFraction`, `fibreFraction`, `fibreDiameter`, `fibreLength`, and `dmax` via `aggregate::dlim()` in `dlim.h`.

Output format

`packing.dat`

A line-oriented text file. The first three lines are metadata comments; the remaining lines are inclusion records.

3D form:

```
# packing v1
# box <lx> <ly> <lz>
# periodicity <px> <py> <pz>
ellipsoid <id> centre 3 <cx> <cy> <cz> angles 3 <phix> <phiy> <phiz> radii 3 <sx> <sy> <sz>
ellipsoid <id> ...
sphere <id> centre 3 <cx> <cy> <cz> radius <r>
sphere <id> ...
fibre <id> endpointcoords 6 <ax> <ay> <az> <bx> <by> <bz> diameter <d>
...
```

2D form (no `z` column in the header, no `fibre` lines):

```
# packing v1
# box <lx> <ly>
# periodicity <px> <py>
disk <id> centre 2 <cx> <cy> radius <r>
...
```

Each inclusion line starts with its type keyword (`disk`, `ellipsoid`, `sphere`, `fibre`) and a per-type sequential id. The remaining tokens follow OOFEM's `keyword count v1 v2 ...` convention, parseable by the same machinery the generator and converter already use. `sphere` lines are written automatically when `Ellipsoid::writeTo` detects three equal semi-axes — the typical case under `#@grading shape sphere`. `disk` lines are the 2D analog.

This format is shared with the generator and the converter:

- The **generator** consumes it via `#@inclusionfile <path> [itz <t>] [refine <r>]` — each `sphere` line becomes an `InterfaceSphere`, each `disk` line becomes an `InterfaceDisk` (both seed surface points plus an ITZ halo). `ellipsoid` lines trigger a warning (the generator does not yet implement arbitrary-orientation ellipsoid surface seeding); `fibre` lines are silently ignored (handled by the converter, not the generator).
- The **converter** consumes it via `#@inclusionfile <path> itz <t> inside <m> interface <m>` — each `sphere` line becomes a `SphereInclusionSpec` driving material classification of every Voronoi/Delaunay edge whose endpoints fall inside or straddle the sphere+ITZ boundary; each `fibre` line becomes a `Fibre` for beam-and-link generation. (`disk` lines for the 2D path are handled via `#@diskinclusion` rather than `#@inclusionfile` — `#@inclusionfile` consumption on the converter side is currently 3D-only.)

`packing.vtu`

ParaView-compatible XML UnstructuredGrid. Each ellipsoid is tessellated into an `(N+1)²` grid of points and `N²` quad cells (default `N = 10`, matching the Matlab visualisation). Each fibre becomes 2 points and a single line cell. Two cell-data arrays let you filter and colour:

Array	Type	Values
`kind`	Int32	`0` for ellipsoid surface, `1` for fibre
`id`	Int32	the inclusion's sequential id

To visualise:

```
paraview packing.vtu
```

In ParaView, apply a `Threshold` on `kind` to show ellipsoids and fibres on separate layers, or colour by `id` to follow individual inclusions through filters.

Example

```
#@output packing.dat
#@vtu packing.vtu
#@box 3 0.05 0.05 0.05
#@periodicity 3 1 1 1
#@seed 42
#@maxiter 1000
#@grading dmin 0.005 dmax 0.020 fraction 0.3
```

This is `src/aggregate/tests/SmallRVEvtu/control.in`. With this seed the placer produces 29 aggregates in a 50 mm cubic periodic RVE — the committed `reference.dat` and `reference.vtu` capture exactly that configuration.

Architecture overview

File	Role
`aggregate.C`	`main()` — orchestrates the read → grading → group1 → fibres → group2 → write pipeline
`box.h/.C`	`Box` — owns dimensions, periodicity, RNG seed, parsed parameters, and the `unique_ptr<Inclusion>` lists for both real and ghost inclusions
`inclusion.h`	Abstract base for any geometric inclusion (`number`, `typeName()`, `writeTo()`)
`ellipsoid.h/.C`	`Ellipsoid` — centre, Euler angles (Rz · Ry · Rx), semi-axes, with cached 4×4 Alfano-Greer quadric form
`fibre.h/.C`	`Fibre` — two endpoints + diameter
`intersection.h/.C`	Free-function pairwise overlap predicates (Alfano-Greer ellipsoid–ellipsoid, ellipsoid–axis-plane, line-segment–ellipsoid)
`gradingcurve.h/.C`	`GradingCurve` — Wriggers interval-volume + sieve halving
`dlim.h/.C`	Analytic group-split threshold (companion-matrix root finder via Eigen)
`placer.h/.C`	`Placer` — trial-and-error placement loop with periodic ghost generation
`random.h`	Portable `uniform()` / `normal()` / `uniformOnSphere()` over `std::mt19937`
`vtu.h/.C`	Standalone XML UnstructuredGrid writer
`aggregateerror.h`	`[[noreturn]] error()` and `errorf()` fatal-error helpers

The whole module depends on Eigen (header-only) and on a small slice of oofemlib (no FE concepts; just convenience). It does **not** depend on the generator or the converter, and nothing in oofemlib depends on it.

Adding a new directive

1. Add a `#@<tag>` branch to `Box::applyDirective` in `box.C`.
2. If the directive carries new parameters, add a struct (like `GradingParameters` / `FibreParameters`) and a getter in `box.h`.
3. Add an entry to the directive table above.
4. If the directive triggers a new pipeline stage, add the orchestration in `aggregate.C` (`main` is intentionally the only place that knows about the high-level sequence).

Adding a new inclusion type

1. Create `<name>.h/.C` with a class derived from `Inclusion`. Implement `typeName()` (the keyword written to the packing file) and `writeTo(ostream&)`.
2. If the new type can be parsed from a packing file directly, add an `initializeFromTokens(std::istringstream&)` method matching the pattern used by `Ellipsoid` and `Fibre`.
3. Add free intersection predicates in `intersection.h/.C` for any new type pair the placer needs to test.
4. Extend `Placer` with a `place<Name>` method and route `Box`'s inclusion lists through `dynamic_cast` checks where needed.
5. Extend `vtu.C` if the new type should appear in ParaView output — choose a tessellation strategy and assign a fresh `kind` value.