Python API Overview

2026-05-05

目录

Python API Overview 10
Import Pattern 10
Coverage Status 10
PDF Export 10
phynexis.utils 11
Import 11
Module Overview 12
C++ Namespace 12
pybind Module 12
See also 12
math 13
Constants 13
Vector Operations 13
Quaternion 14
Example 14
Spherical Harmonics 15
Unexposed C++ API 16
STLModel 16
Constructor 16
Properties 16
Methods 16
Static Methods 18
Example 18
Example: Create from vertices and facets 18
Mesh I/O Functions 19
Unexposed C++ API 20
Visual Preview 20
SurfaceMesh 21
Constructor 21
Properties 21
Example 21
Unexposed C++ API 21
OpenMP 21
Functions 22
Example 22
Note 23
sampling 23
Distribution Sampling 23
Spatial Sampling 24
Orientation Sampling 25
Spherical Sampling 26
Factory 26
Unexposed C++ API 28
DataIO 28
Methods 28
Example 28
Unexposed C++ API 29

SaveOptions / LoadOptions 29
FileFormat 29
SaveOptions 30
SaveVtkOptions 30
LoadOptions 30
Example 31
Format Utility Functions 31
Shape 32
Constructor 32
Properties 33
Methods 33
Example 36
Shape Subclasses 37
ShapeFactory 38
ShapeRegistry 38
Example: Surface nodes and sampling 39
Unexposed C++ API 39
Vectors and Matrices 40
Vector Types 40
Matrix Types 41
Example 41
Known Issues 42
Unexposed C++ API 42
FlatHashMap 43
Available Instantiations 43
Constructor 43
Methods 43
Python Protocols 44
Example 45
Unexposed C++ API 46
Binding Fix Record 46
Voronoi 47
TessellationParams 47
Tessellation 47
Diagram 48
Example 48
Unexposed C++ API 48
vtk 48
Data Classes 48
Read Functions 49
Write Functions 49
Format Detection 49
Example 49
Conversion Functions 49
Known Issues 50
Unexposed C++ API 50
wrappers 50
CorkWrapper 51
EigenWrapper 51
IGLWrapper 51
CGALWrapper 51

mpi 52
TetGenOptions 52
`tetgen_get_tetmesh(vertices, facets, options)` 53
Example 53
Known Issues 53
Unexposed C++ API 54
Cork 54
Description 54
Methods 54
Note 54
Unexposed C++ API 55
Console 55
Level 55
MPIOutputMode 55
Config 55
Logging Functions 57
Control Functions 58
FPE Traps 58
Functions 59
Example 59
Unexposed C++ API 60
LevelSetField 60
Constructor 60
Methods 60
Example 62
Unexposed C++ API 62
MiniMap 62
Constructor 63
Methods 63
Python Protocols 63
Example 63
Unexposed C++ API 64
SDFCalculator 64
Constructor 64
Methods 64
Example 65
Unexposed C++ API 66
StringUtils 66
Functions 66
Unexposed C++ API 67
TimeUtils 67
Functions 67
Unexposed C++ API 68
phynexis.fields 68
Module Status 68
Submodules 68
See also 69
Field / ScalarField 69
Constructors 69
NumPy Interop 70
Element Access 70

Capacity 71
Bulk Operations 72
Arithmetic (Double only) 73
Comparison Semantics 73
Properties 74
Type Aliases and Integer / Bool Fields 74
Known Issues 75
Unexposed C++ API 75
FieldBase 75
Description 75
Constructors 75
Methods 76
Properties 78
Unexposed C++ API 79
Known Issues 79
FieldHolder 79
Constructors 80
Methods 80
Examples 81
Known Issues 82
Unexposed C++ API 82
FieldLayout 83
Description 83
Constructors 83
Methods 84
Known Issues 85
FieldManager 85
Description 85
Constructors 85
Methods 85
Unexposed C++ API 93
Known Issues 93
FieldMeta 93
Enums 93
Constructors 94
Properties 94
Example 94
Unexposed C++ API 95
FieldSchema 95
Description 95
Constructors 95
Methods 96
Unexposed C++ API 98
Known Issues 98
FieldSlot 99
Description 99
Constructors 99
Factory Methods 99
Properties 102
Methods 102
Field Schemas 103

Schema Types 103
Usage 103
CSRMatrix 104
Description 104
Constructors 104
Methods 105
Unexposed C++ API 109
Known Issues 109
Field Views 109
Type Summary 110
Common API (FieldViewBase) 110
ScalarFieldView 111
Vec3FieldView 111
Vec4FieldView 112
Vec6FieldView 113
VecNFieldView 113
VecXFieldView 114
Known Issues 114
LinkedFieldView 115
Related Types 115
LinkedFieldView 116
LinkedFieldRowView 117
Attached Data Adapters 117
Known Issues 118
Operators 118
Math 118
Reduction 120
Prefix Sum 120
Sort 121
Radix Sort 122
Known Issues 122
I/O and Utilities 122
Console Output 122
Field I/O 123
Manager I/O 124
VecXField I/O 125
MPI Utilities 126
Known Issues 127
phynexis.parsim 127
Module Status 127
Submodules 127
Quick Start 127
See also 128
Schemas 128
Schema Types 128
BoundLayout 128
Known Issues 130
Graph 130
ComputationalGraph 130
SetBase 131
NodeSet 131

EdgeSet 132
HyperEdgeSet 133
ShapeStore 133
Views 135
Known Issues 135
Simulator 135
Simulator 136
SimulationConfig 136
Context 136
OperatorSystem 137
RuntimeState 138
DomainSystem 138
SpatialIndex 139
UniformGridIndex 139
ResourceRegistry / ContextResource 139
Known Issues 140
operators 140
Base Classes 140
Phase Identifiers 140
Integration Operators (`operators.integration`) 141
Force Operators (`operators.forces`) 142
Interaction Operators (`operators.interaction`) 143
Output Operators (`operators.output`) 144
Models and Utils 145
RepulsionModel 145
LinearRepulsionModel 146
RepulsionModelFactory 146
NodeGenerator 146
FieldSampler 147
Node Property Utils 148
SpatialIndex / UniformGridIndex 148
Known Issues 148
views 149
Node Views 149
Edge Views 150
HyperEdge Views 150
PropertiesView 151
phynexis.workflow 152
Import 152
Module Overview 152
Free Functions 152
PortInfo / MethodSignature 153
Description 153
PortInfo 153
MethodSignature 154
Deprecated Reference (phynexis v0) 154
ContactModel 155
LinearSpring 155
ParallelBond 155
ContactSolverSettings Class 156
ContactSolverFactory 156

DEMSolver 157
CellManager 157
Domain 158
DomainManager 159
TetMesh 159
Membrane 160
ModifierManager 161
Modifier 162
BreakageAnalysisPD 163
DataDumper 164
MembraneWall 165
ParticleGroup 166
ParticleMotionControl 167
WallGroup 167
WallDispControl 168
WallMotionIntegrator 169
WallServoControl 170
MPIManager 171
PyNetDEM 171
Peridigm 172
DomainSplittor 172
LevelSetSplittor 172
TetMeshSplittor 173
PeriDigmDiscretization 173
PeriDigmDiscretization 174
PeriDigmMaterial 176
PeriDigmDamageModel 176
PeriDigmBlock 177
PeriDigmBoundaryCondition 177
PeriDigmSettings 179
DEMFragment 179
ParticleStrengthParameters 180
PeriDigmSimulator 181
PeriDigmDEMCoupler 182
DEMObjectPool 184
Scene 184
PackGenerator 188
Particle 189
Wall 191
WallBoxPlane 195
WallBoxPlate 195
BondedSpheres 195
BondedVoronois 196
ContactPP 198
ContactPW 199
Shape 200
Sphere 202
PointSphere 202
Plane 202
Triangle 203
Cylinder 204

Ellipsoid 204
SphericalHarmonics 205
PolySuperEllipsoid 205
PolySuperQuadrics 206
LevelSet 207
TriMesh 208
InitPyShapeModule 209
Simulation 209
LevelSetFunction 210
STLReader 211
STLModel 211
WSCVTSampler 213
Voronoi 213
Cork 214
OpenMP 215

Python API Overview

Use the header **Docs** item for the manual / guides. On API pages the left sidebar shows **only** this Python reference tree (no manual outline).

Phynexis exposes its C++ core through pybind11 bindings. The table below lists the **nine lazy-loaded Python submodules** in a typical build. **On this documentation site**, detailed reference pages are maintained primarily under `utils` and `fields` (plus historical notes under `deprecated`). Other rows describe modules that ship with the package; page-level coverage may still be thin—use the **coverage tracker** and the **PDF export** for the broadest automated snapshot.

Module	Python Import	C++ Namespace	Description
utils	`phynexis.utils`	`phynexis::utils`	Utilities: console, SDF, string helpers
fields	`phynexis.fields`	`phynexis::fields`	Field quantities and level sets
parsim	`phynexis.parsim`	`phynexis::parsim`	Parallel simulation framework
netdem	`phynexis.netdem`	`phynexis::netdem`	Discrete Element Method (DEM)
netfem	`phynexis.netfem`	`phynexis::netfem`	Finite Element Method (FEM)
cfddem	`phynexis.cfddem`	`phynexis::cfddem`	CFD-DEM coupling
ml	`phynexis.ml`	`phynexis::ml`	Machine learning interfaces
peridigm	`phynexis.peridigm`	`phynexis::peridigm`	Peridynamics
workflow	`phynexis.workflow`	`phynexis::workflow`	Workflow automation

Import Pattern

```python
import phynexis

# Lazy-loaded submodules
scene = phynexis.netdem.Scene()
mesh = phynexis.netfem.TetMesh()
```

Coverage Status

See progress tracker for the current documentation coverage by module.

PDF Export

A compiled PDF merges the same Markdown tree (including pages with sparse prose). Regenerate locally after large binding updates:

```
python scripts/export-python-api-pdf.py
```

Published build:

Download phynexis-python-api.pdf

phynexis.utils

Utility module providing helper classes and functions for console logging, string manipulation, time measurement, signed distance fields, mesh I/O, sampling, math, and more.

Import

```python
import phynexis

# Access submodule directly
phynexis.utils.info("hello")

# Or bind locally
from phynexis import utils
utils.info("hello")
```

Module Overview

Class / Function	Description
Config	Console output configuration
debug / info / warning / error / fatal / plain	Logging functions
set_level / set_verbose / set_color	Console control functions
to_string	Numeric to string conversion
get_time_micros	High-resolution timestamp
MiniMapIdx64	Lightweight index map
SDFCalculator	Signed distance field calculator
LevelSetField	Level set field representation
STLModel / read / read_off / write_off	3D triangle mesh and mesh I/O
SurfaceMesh	Surface mesh struct
DataIO	Tabular data I/O
SaveOptions / LoadOptions / FileFormat	File save/load options and format resolution
format_from_extension / format_from_path / join_path_file	File format and path utilities
Voronoi	Voronoi tessellation (Tessellation / Diagram)
Cork	Boolean mesh operations (CorkWrapper)
OpenMP / parallel	Thread control bindings
FPE Traps	Floating-point exception trap utilities
Vec2d/Vec2i/Vec3d/Vec3i/Vec4d/Vec4i/Vec6d	Fixed-size vectors
Mat2d/Mat3d	Fixed-size matrices
FlatHashMap	Flat hash map
Shape	Geometric shapes (Sphere, Cuboid, Ellipsoid, Cylinder, ...)
math	Math constants, vector ops, quaternion
sampling	Distribution, spatial, orientation, spherical sampling
vtk	VTK file I/O and conversion
wrappers	Cork, CGAL, Eigen, igl, TetGen wrappers with MPI utilities

C++ Namespace

`phynexis::utils`

pybind Module

`pyutils` (lazy-loaded via `phynexis.utils`)

See also

- Manual home — installation, basic usage, CFD–DEM, visualization
- Python API overview — submodule map and coverage notes
- phynexis.fields — field types and layouts
- Deprecated flat references — legacy v0 pages

math

C++: `phynexis::utils::math` **Python**: `phynexis.utils.math` **Header**: `src/utils/math/math.hpp`

Mathematical constants, vector operations, and quaternion utilities.

Constants

Name	Value	Description
`PI`	3.14159...	Circle constant
`EPSILON`	2.22e-16	Machine epsilon for doubles
`TOL`	1e-6	Default numerical tolerance
`HUGE_VALUE`	1e20	Large sentinel value
`SQRT_2`	1.41421...	Square root of 2
`SQRT_3`	1.73205...	Square root of 3

Vector Operations

`dot(a, b)`

Compute dot product of two vectors.

Parameters:

Parameter	Type	Description
`a`	`Vec3d`	First vector
`b`	`Vec3d`	Second vector

Returns: `float`

`cross(a, b)`

Compute cross product of two vectors.

Returns: `Vec3d`

`norm_l2(v)`

Compute L2 norm (magnitude) of a vector.

Returns: `float`

`normalize(v)`

Normalize a vector in-place.

`to_int64(val)`

Convert a 32-bit integer to a 64-bit integer.

Parameters:

Parameter	Type	Description
`val`	`int`	32-bit integer value

Returns: `int` — 64-bit integer

Quaternion

Module `phynexis.utils.math.quaternion` provides quaternion operations.

`from_rodrigues(angle, axis)`

Create a quaternion from angle-axis (Rodrigues) representation.

Parameters:

Parameter	Type	Description
`angle`	`float`	Rotation angle in radians
`axis`	`Vec3d`	Rotation axis

Returns: `Vec4d`

`to_rodrigues(q)` / `to_matrix(q)` / `from_matrix(m)`

Convert quaternion to/from Rodrigues, rotation matrix.

`multiply(a, b)` / `conjugate(q)` / `normalize(q)` / `add(a, b)`

Quaternion arithmetic.

Example

```python
import phynexis

# Constants
print("PI:", phynexis.utils.math.PI)
print("EPSILON:", phynexis.utils.math.EPSILON)

# Vector ops
a = phynexis.utils.Vec3d(1.0, 0.0, 0.0)
b = phynexis.utils.Vec3d(0.0, 1.0, 0.0)
print("dot:", phynexis.utils.math.dot(a, b))
print("cross:", phynexis.utils.math.cross(a, b))
print("norm:", phynexis.utils.math.norm_l2(a))

# Quaternion
q = phynexis.utils.math.quaternion.from_rodrigues(
    phynexis.utils.math.PI / 2,
    phynexis.utils.Vec3d(0.0, 0.0, 1.0)
)
print("quaternion:", q)
```

Output:

```text
PI: 3.141592653589793
EPSILON: 2.220446049250313e-16
dot: 0.0
cross: Vec3d(0, 0, 1)
norm: 1.0
quaternion: Vec4d(0, 0, 0.707107, 0.707107)
```

Spherical Harmonics

Spherical harmonics evaluation functions used internally by `SphericalHarmonics` shape.

`legendre_p(n, m, x)`

Associated Legendre polynomial P(n, m, x).

Parameters:

Parameter	Type	Description
`n`	`int`	Degree
`m`	`int`	Order
`x`	`float`	Input value

Returns: `float`

`normalization_factor(n, m)`

Compute spherical harmonic normalization factor.

Parameters:

Parameter	Type	Description
`n`	`int`	Degree
`m`	`int`	Order

Returns: `float`

`calculate_ynm_fast(theta, phi, deg)`

Evaluate spherical harmonics at given angles up to specified degree. Accepts scalar or array inputs.

Parameters:

Parameter	Type	Description
`theta`	`float` or `list[float]`	Polar angle(s)
`phi`	`float` or `list[float]`	Azimuthal angle(s)
`deg`	`int`	Maximum degree

Returns: `list[float]` — Ynm coefficients

Example:

```python
import phynexis

# Associated Legendre polynomial
p = phynexis.utils.math.legendre_p(2, 1, 0.5)
print("legendre P(2,1,0.5):", p)

# Normalization factor
nf = phynexis.utils.math.normalization_factor(2, 1)
print("normalization factor:", nf)
```

Output:

```text
legendre P(2,1,0.5): -1.299038105676658
normalization factor: 2.958039891549808
```

Unexposed C++ API

- `flat_array3d` utilities (internal 3D array layout, not exposed via pybind)

STLModel

C++: `phynexis::utils::STLModel` **Python**: `phynexis.utils.STLModel` **Header**: `src/utils/mesh/stl_model.hpp`

3D triangle mesh model supporting STL/OFF file I/O, geometric transformations, decimation, convex hull, and mesh analysis.

Constructor

`STLModel()`

Creates an empty model.

`STLModel(vertices, facets)`

Creates a model from vertex and face arrays.

Parameters:

Parameter	Type	Description
`vertices`	`list[Vec3d]`	Vertex positions
`facets`	`list[Vec3i]`	Triangle face indices

`STLModel(file_path)`

Load from a file path (auto-detects format).

Properties

Property	Type	Description
`vertices`	`list[Vec3d]`	Model vertices (read/write)
`facets`	`list[Vec3i]`	Triangle face indices (read/write)

Methods

`init_from_stl(file_path)` / `init_from_off(file_path)`

Load from STL or OFF file.

`save_to(path, file, options)`

Save model to file with format options.

`load_from(path, file, opt=LoadOptions())`

Load model from file with load options.

Parameters:

Parameter	Type	Description
`path`	`str`	Directory path
`file`	`str`	File name
`opt`	`LoadOptions`	Load options (create_directories, format_hint)

`transpose()` / `rotate(rotation)`

Apply geometric transformations.

`copy_pose()` / `copy_pose_dev()`

Copy the model's current pose.

`remove_unreferenced_vertices()` / `remove_duplicate_vertices()`

Clean up mesh topology.

`reorient_facets()`

Ensure facet normals point outward.

`decimate(target_num_faces)`

Reduce face count via mesh decimation.

`standardize()`

Center at origin and scale to unit size.

`set_size(size)`

Scale to a specific size.

`make_convex()`

Convert to convex hull.

`refine()` / `smooth_mesh(iterations)`

Subdivide or smooth the mesh.

`merge(other_model)`

Merge with another STLModel.

`size()`

Return characteristic size (diagonal of bounding box).

`get_center()` / `get_surface_area()` / `volume()` / `inertia()`

Compute geometric properties.

`is_convex()` / `is_face_outside(facet_index, point)` / `enclose(point)`

Convexity and containment tests.

`bound_aabb()`

Return axis-aligned bounding box.

`print_info()`

Print model summary.

Static Methods

`CenterOf(vertices, facets)` / `SurfaceAreaOf(vertices, facets)` / `VolumeOf(vertices, facets)` / `InertiaOf(vertices, facets)` / `check_convexity_of(vertices, facets)`

Compute properties from raw mesh data without creating an STLModel instance.

Example

```python
import phynexis

# Create a mesh from a sphere shape
model = phynexis.utils.shape.Sphere(2.0).get_stl_model(200)

print("size:", model.size())
print("center:", model.get_center())
print("volume:", model.volume())
print("surface area:", model.get_surface_area())
print("is convex:", model.is_convex())

# Transformations
model.standardize()
print("after standardize, size:", model.size())
```

Output:

```text
size: 1.9814485316774066
center: Vec3d(-3.05057e-06, -1.61383e-05, -4.46186e-05)
volume: 4.073305749647874
surface area: 12.374617224062705
is convex: True
after standardize, size: 0.9999999999999999
```

Example: Create from vertices and facets

```python
import phynexis

model = phynexis.utils.STLModel()
model.vertices = [
    phynexis.utils.Vec3d(0.0, 0.0, 0.0),
    phynexis.utils.Vec3d(1.0, 0.0, 0.0),
    phynexis.utils.Vec3d(0.0, 1.0, 0.0),
]
model.facets = [phynexis.utils.Vec3i(0, 1, 2)]
```

```python

print("vertices:", len(model.vertices))
print("facets:", len(model.facets))
print("center:", model.get_center())
print("volume:", model.volume())
print("surface area:", model.get_surface_area())
```

Output:

```text
vertices: 3
facets: 1
center: Vec3d(0.333333, 0.333333, 0)
volume: 0.0
surface area: 0.5
```

Mesh I/O Functions

Top-level functions for reading and writing mesh files.

`read(filename)`

Read a mesh from an STL file.

Parameters:

Parameter	Type	Description
`filename`	`str`	Path to STL file

Returns: `STLModel`

`read_off(filename)`

Read a mesh from an OFF file.

Parameters:

Parameter	Type	Description
`filename`	`str`	Path to OFF file

Returns: `STLModel`

`write_off(filename, model)`

Write an `STLModel` to an OFF file.

Parameters:

Parameter	Type	Description
`filename`	`str`	Output OFF file path
`model`	`STLModel`	Model to save

Example:

```python
import phynexis

# Read STL
model = phynexis.utils.read("/Users/lzhshou/Documents/myResearch/myProjects/apaam/repo/phynexis/data/sphere.stl")
print("vertices:", len(model.vertices), "facets:", len(model.facets))

# Write OFF
```

```
phynexis.utils.write_off("/tmp/sphere.off", model)
```

Output:

```text
vertices: 162
facets: 320
```

Unexposed C++ API

- `get_triangle_strips()` — return type not fully exposed

Visual Preview

Export the mesh to VTK and render with `vtk-snap` for visual verification:

```python
import phynexis

model = phynexis.utils.shape.Sphere(2.0).get_stl_model(200)
opts = phynexis.utils.SaveOptions()
opts.overwrite = True
model.save_to('/tmp/', 'stl_preview.vtk', opts)
```

```bash
# requires vtk-snap skill + pyvista
python3 .claude/skills/vtk-snap/vtk-snap.py /tmp/stl_preview.vtk -o /tmp/stl_preview.png
```

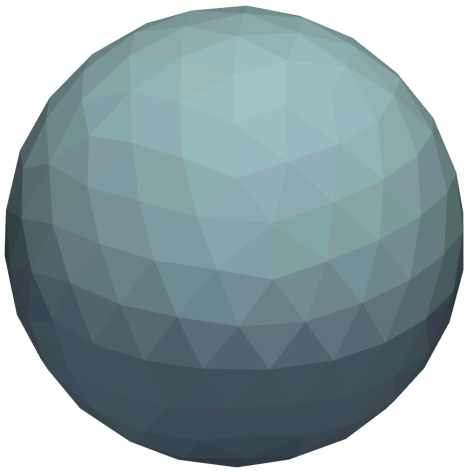

Figure 1: STLModel visual preview

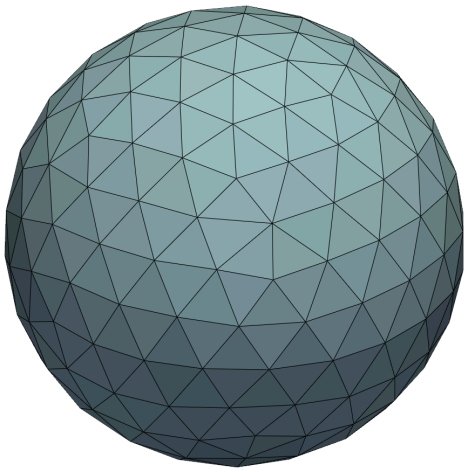

Figure 2: STLModel with edges (parallel projection)

SurfaceMesh

C++: `phynexis::utils::mesh::SurfaceMesh` **Python**: `phynexis.utils.SurfaceMesh` **Header**: `src/utils/mesh/surface_mesh.hpp`

Minimal struct representing a surface mesh as vertices and triangular facets.

Constructor

`SurfaceMesh()`

Creates an empty mesh.

Properties

Property	Type	Description
`vertices`	`list[Vec3d]`	Vertex positions (read/write)
`facets`	`list[Vec3i]`	Triangle face indices (read/write)

Example

```python
import phynexis

sm = phynexis.utils.SurfaceMesh()
sm.vertices = [
    phynexis.utils.Vec3d(0.0, 0.0, 0.0),
    phynexis.utils.Vec3d(1.0, 0.0, 0.0),
    phynexis.utils.Vec3d(0.0, 1.0, 0.0),
]
sm.facets = [phynexis.utils.Vec3i(0, 1, 2)]

print("vertices:", len(sm.vertices))
print("facets:", len(sm.facets))
print("first vertex:", sm.vertices[0])
```

Output:

```text
vertices: 3
facets: 1
first vertex: Vec3d(0, 0, 0)
```

Unexposed C++ API

- `SurfaceMesh` is a simple POD struct. No additional methods are defined.

OpenMP

C++: `phynexis::utils::parallel` / OpenMP runtime **Python**: `phynexis.utils` **Header**: `src/utils/parallel/parallel.hpp`

Thread-level parallelism control. These functions map directly to OpenMP runtime calls.

Functions

`omp_get_max_threads()`

Return the maximum number of threads available.

Returns: `int`

`omp_set_num_threads(num_threads)`

Set the number of threads for subsequent parallel regions.

Parameters:

Parameter	Type	Description
`num_threads`	`int`	Number of threads

`parallel_max_threads()`

Alias for `omp_get_max_threads()`.

`parallel_set_num_threads(num_threads)`

Alias for `omp_set_num_threads()`.

`parallel_in_parallel()`

Return `True` if the current code is executing inside a parallel region.

Returns: `bool`

`parallel_num_procs()`

Return the number of processors available.

Returns: `int`

`parallel_thread_num()`

Return the current thread number within a parallel region (0-based).

Returns: `int`

Example

```python
import phynexis

print("max threads:", phynexis.utils.omp_get_max_threads())
phynexis.utils.omp_set_num_threads(4)
print("set to 4 threads")
print("num procs:", phynexis.utils.parallel_num_procs())
print("in parallel:", phynexis.utils.parallel_in_parallel())
```

Output:

```text
max threads: 10
set to 4 threads
num procs: 10
```

```
4 in parallel: False
```

Note

- These functions affect the **global** OpenMP thread pool.
- Setting threads via `parallel_set_num_threads()` also calls `omp_set_num_threads()` internally.

sampling

C++: `phynexis::utils::sampling` **Python**: `phynexis.utils.sampling` **Header**: `src/utils/sampling/*.hpp`

Sampling utilities for distributions, spatial points, orientations, and spherical arrangements.

Most sampler classes support a common interface. Distribution samplers return a single value; spherical/spatial samplers return a single point or take a `num_samples` argument.

Method	Returns	Description
`sample()`	`float` / `Vec3d`	Draw a random sample (distribution/spatial/orientation)
`sample(N)`	`list[Vec3d]`	Draw N samples at once (spherical samplers)
`reset()`	—	Reset the internal RNG state
`get_seed()`	`int`	Get current random seed
`set_seed(seed)`	—	Set random seed for reproducibility

Distribution Sampling

Generates random scalar values from probability distributions.

`Uniform(min, max)`

Uniform distribution in [min, max]. Alias: `sampling.UniformDistribution`.

Constructor:

Parameter	Type	Description
`min`	`float`	Lower bound
`max`	`float`	Upper bound

`Normal(mean, std)`

Normal (Gaussian) distribution. Alias: `sampling.Normal`.

Constructor:

Parameter	Type	Description
`mean`	`float`	Mean value
`std`	`float`	Standard deviation

`Exponential(lambda)`

Exponential distribution. Alias: `sampling.Exponential`.

Constructor:

Parameter	Type	Description
`lambda`	`float`	Rate parameter

Example:

```python
import phynexis

# Uniform distribution
u = phynexis.utils.sampling.UniformDistribution(0.0, 10.0)
print("uniform sample:", u.sample())
print("uniform sample:", u.sample())
u.reset()
print("after reset:", u.sample())

# Normal distribution
n = phynexis.utils.sampling.Normal(0.0, 1.0)
print("normal sample:", n.sample())

# Exponential distribution
e = phynexis.utils.sampling.Exponential(1.0)
print("exponential sample:", e.sample())
```

Output:

```text
uniform sample: 3.591168210525055
uniform sample: 9.928038367790007
after reset: 3.591168210525055
normal sample: -0.5704137239746693
exponential sample: 1.3265582842979053
```

Spatial Sampling

Generate points within a shape's bounding volume. All take a `Shape` as the first argument and provide `sample()` returning `Vec3d`.

Class	Alias	Extra Constructor Params
`Random(shape)`	`sampling.Random`	—
`Grid(shape, resolution)`	`sampling.Grid`	`resolution: Vec3i` — grid divisions
`CVT(shape)`	`sampling.CVT`	optional `max_iter`, `tol`
`Voronoi(shape, seeds)`	`sampling.Voronoi`	`seeds: list[Vec3d]` — initial seed points
`WCVT(shape, weight_func)`	`sampling.WCVT`	`weight_func` — callable (requires `<functional>` include)

Additional Methods

Spatial samplers provide:

Method	Returns	Description
`get_shape()`	`Shape`	Get bound shape
`set_shape(shape)`	—	Set bound shape
`volume()`	`float`	Sample volume
`is_inside(point)`	`bool`	Test if point is inside
`get_bounding_box()`	`BoundingBox`	Bounding box
`get_center()`	`Vec3d`	Shape center

Type-specific:

Sampler	Method		Description
`Grid`	`get_resolution()` / `set_resolution(v)`		Grid divisions
`CVT`	`set_max_iter(n)` / `set_tol(t)`		Convergence control
`WCVT`	`set_weight_func(f)`	/	Weight function and convergence
	`set_max_iter(n)` / `set_tol(t)`	/	
`Voronoi`	`get_seeds()`	/	Seed management
	`set_seeds(s)`	/	
	`get_voronoi_diagram()`		

Example:

```python
import phynexis

# Create a sphere shape
sphere = phynexis.utils.shape.Sphere(1.0)

# Random points inside sphere
random_sampler = phynexis.utils.sampling.Random(sphere)
print("random point:", random_sampler.sample())
print("volume:", random_sampler.volume())

# Grid sampling
grid_sampler = phynexis.utils.sampling.Grid(sphere, phynexis.utils.Vec3i(3, 3, 3))
print("grid point:", grid_sampler.sample())
```

Output:

```text
random point: Vec3d(0.0801163, 0.315637, 0.00687753)
volume: 4.1887902047863905
grid point: Vec3d(0, 0, 0.471405)
```

Orientation Sampling

Generate random 3D orientations (rotation quaternions as `Vec4d`).

Class	Constructor Parameters	Alias
`Uniform()`	—	`sampling.orientation.Uniform`
`ConeDir(target_dir, body_axis, max_angle, roll_mode, ...)`	`target_dir: Vec3d, body_axis: Vec3d, max_angle: float, roll_mode: RollMode`	—
`ConeRef(reference_quat, max_angle)`	`reference_quat: Vec4d, max_angle: float`	—

Example:

```python
import phynexis

# Uniform orientations
o = phynexis.utils.sampling.orientation.Uniform()
print("orientation quat:", o.sample())
```

Output:

```text
orientation quat: Vec4d(0.556432, -0.514359, -0.475483, 0.449191)
```

Spherical Sampling

Generate points uniformly distributed on a sphere surface.

Class	Alias	Extra Params
`Uniform()`	`sampling.spherical.Uniform`	—
`CVT()`	`sampling.spherical.CVT`	optional `max_iters`, `tol`
`WCVT(weight_func)`	—	`weight_func: callable` (requires `<functional>` include)
`GoldenSpiral()`	`sampling.spherical.GoldenSpiral`	—

Example:

```python
import phynexis

# Uniform spherical sampling (returns list of Vec3d)
su = phynexis.utils.sampling.spherical.Uniform()
pts = su.sample(3)
print("uniform points:", len(pts), "first:", pts[0])

# Golden spiral distribution
gs = phynexis.utils.sampling.spherical.GoldenSpiral()
pts2 = gs.sample(5)
print("golden spiral points:", len(pts2), "first:", pts2[0])
```

Output:

```text
uniform points: 3 first: Vec3d(0.4735, 0.86313, -0.175515)
golden spiral points: 5 first: Vec3d(0.530863, 0, 0.847458)
```

Factory

String-based sampler factory for creating sampler instances by type name.

Method	Returns	Description
`create_uniform_distribution(min, max)`	`UniformDistribution`	Uniform distribution sampler
`create_normal_distribution(mean, std)`	`Normal`	Normal distribution sampler
`create_exponential_distribution(lambda)`	`Exponential`	Exponential distribution sampler
`create_random_spatial_sampler(shape)`	`Random`	Random spatial sampler
`create_grid_spatial_sampler(shape, resolution)`	`Grid`	Grid spatial sampler
`create_spatial_sampler_cvt(shape)`	`CVT`	CVT spatial sampler
`create_spatial_sampler_voronoi(shape, seeds)`	`Voronoi`	Voronoi spatial sampler
`create_spatial_sampler_wcvt(shape, weight_func)`	`WCVT`	WCVT spatial sampler
`create_orientation_sampler_uniform()`	—	Uniform orientation sampler
`create_orientation_sampler_cone_dir(...)`	—	Cone-dir orientation sampler
`create_orientation_sampler_cone_ref(ref_quat, max_angle)`	—	Cone-ref orientation sampler
`create_spherical_sampler_uniform()`	—	Uniform spherical sampler
`create_spherical_sampler_golden_spiral()`	—	Golden spiral spherical sampler
`create_spherical_sampler_cvt()`	—	CVT spherical sampler
`create_spherical_sampler_voronoi()`	—	Voronoi spherical sampler
`create_spherical_sampler_wcvt(weight_func)`	—	WCVT spherical sampler

Strategy Parsers

The Factory also provides string-based strategy parsers:

Method	Returns	Description
`parse_distribution_strategy(str)`	`DistributionStrategy`	Parse strategy name: `"uniform"`, `"normal"`, `"exponential"`
`parse_spatial_strategy(str)`	`SpatialStrategy`	Parse spatial strategy: `"random"`, `"grid"`, `"voronoi"`, `"cvt"`, `"wcvt"`
`parse_spherical_strategy(str)`	`SphericalStrategy`	Parse spherical strategy: `"uniform"`, `"golden_spiral"`, `"voronoi"`, `"cvt"`, `"wcvt"`
`parse_orientation_strategy(str)`	`OrientationStrategy`	Parse orientation strategy: `"uniform"`, `"cone_ref"`, `"cone_dir"`

SamplerType Enum

Value	Description
`DISTRIBUTION`	Probability distribution sampler
`SPATIAL`	Spatial point sampler
`SPHERICAL`	Spherical surface sampler
`ORIENTATION`	Orientation (quaternion) sampler

Unexposed C++ API

- `DistributionStrategy` / `SpatialStrategy` / `OrientationStrategy` / `SphericalStrategy` — internal strategy base classes

DataIO

C++: `phynexis::utils::DataIO` **Python**: `phynexis.utils.DataIO` **Header**: `src/utils/serde/data_io.hpp`

Utility class for importing and exporting tabular data (CSV, TSV, space-separated) with automatic delimiter detection.

Methods

`import_data(filename, lines_to_skip=0)`

Import numeric data from a file. Supports comma, tab, and space delimiters with auto-detection.

Parameters:

Parameter	Type	Default	Description
`filename`	`str`	—	Path to data file
`lines_to_skip`	`int`	0	Number of header lines to skip

Returns: `list[list[float]]` — 2D array of doubles

`save_data(data, head, filename)`

Save tabular data to a file with the given header string.

Parameters:

Parameter	Type	Description
`data`	`list[list[int]]` or `list[list[float]]`	2D data array
`head`	`str`	Header line (e.g. `"x,y,z"`)
`filename`	`str`	Output file path

`file_exist(filename)`

Check if a file exists.

Parameters:

Parameter	Type	Description
`filename`	`str`	File path to check

Returns: `bool`

Example

```python
import phynexis

# Save integer data
```

```python
phynexis.utils.DataIO.save_data(
    [[1, 2, 3], [4, 5, 6]],
    "x,y,z",
    "/tmp/data_int.csv"
)

# Save float data
phynexis.utils.DataIO.save_data(
    [[1.5, 2.5, 3.5], [4.5, 5.5, 6.5]],
    "x,y,z",
    "/tmp/data_float.csv"
)

# Import data (auto-detects delimiter)
data = phynexis.utils.DataIO.import_data("/tmp/data_float.csv")
print("rows:", len(data))
print("first row:", data[0])
print("second row:", data[1])

# File existence check
print("exists:", phynexis.utils.DataIO.file_exist("/tmp/data_float.csv"))
print("missing:", phynexis.utils.DataIO.file_exist("/tmp/nonexistent.csv"))
```

Output:

```text
rows: 2
first row: [1.5, 2.5, 3.5]
second row: [4.5, 5.5, 6.5]
exists: True
missing: False
```

Unexposed C++ API

- `detect_delimiter()` / `parse_line()` — internal helpers

SaveOptions / LoadOptions

C++: `phynexis::utils::SaveOptions`, `LoadOptions`, `SaveVtkOptions`, `FileFormat` **Python**: `phynexis.utils.SaveOptions`, `LoadOptions`, `SaveVtkOptions`, `FileFormat` **Header**: `src/utils/serde/save_options.hpp`

Configuration objects for file save/load operations used by `Shape.save_to()` / `Shape.load_from()` and other serializable types.

FileFormat

Enum defining supported file formats.

Value	Description
`FileFormat.Auto`	Infer from file extension
`FileFormat.Json`	JSON format
`FileFormat.Binary`	Binary format
`FileFormat.VTK`	VTK format
`FileFormat.Stl`	STL mesh format
`FileFormat.Off`	OFF mesh format

SaveOptions

Constructor

`SaveOptions()`
Default: `overwrite=True`, `create_directories=True`, `format_hint=Auto`.

Properties

Property	Type	Access	Description
`overwrite`	`bool`	read/write	Overwrite existing files
`create_directories`	`bool`	read/write	Create parent directories
`format_hint`	`FormatHint`	read/write	Format override hint

SaveVtkOptions

Inherits from `SaveOptions`. Adds VTK-specific settings.

Constructor

`SaveVtkOptions()`
Default: `binary=False`, `use_double=False`.

Properties

Property	Type	Access	Description
`binary`	`bool`	read/write	Write VTK in binary mode
`use_double`	`bool`	read/write	Use double precision

LoadOptions

Constructor

`LoadOptions()`
Default: `create_directories=False`, `format_hint=Auto`.

Properties

Property	Type	Access	Description
`create_directories`	`bool`	read/write	Create parent directories
`format_hint`	`FormatHint`	read/write	Format override hint

Example

```python
import phynexis

# Save options with JSON format
opt = phynexis.utils.SaveOptions()
opt.overwrite = True
opt.create_directories = True
opt.format_hint.value = phynexis.utils.FileFormat.Json
print("format:", opt.format_hint.value)

# VTK options
vtk_opt = phynexis.utils.SaveVtkOptions()
vtk_opt.binary = True
vtk_opt.use_double = True
print("vtk binary:", vtk_opt.binary)

# Load options
load_opt = phynexis.utils.LoadOptions()
load_opt.create_directories = False
print("load create_dirs:", load_opt.create_directories)
```

Output:

```text
format: FileFormat.Json
vtk binary: True
load create_dirs: False
```

Format Utility Functions

Top-level utility functions for resolving file formats.

`format_from_extension(ext)`

Look up the `FileFormat` enum value from a file extension (with dot).

Parameters:

Parameter	Type	Description
`ext`	`str`	File extension including dot, e.g. `".stl"`

Returns: `FileFormat`

`format_from_path(path)`

Look up the `FileFormat` enum value from a file path by extracting its extension.

Parameters:

Parameter	Type	Description
`path`	`str`	File path

Returns: `FileFormat`

`join_path_file(path, file)`

Join a base directory path and a filename.

Parameters:

Parameter	Type	Description
`path`	`str`	Base directory path
`file`	`str`	Filename

Returns: `str`

Example:

```python
import phynexis

# Format resolution
fmt_ext = phynexis.utils.format_from_extension(".stl")
print("from .stl:", fmt_ext)

fmt_path = phynexis.utils.format_from_path("/data/model.stl")
print("from path:", fmt_path)

# Path joining
joined = phynexis.utils.join_path_file("/output", "result.txt")
print("joined:", joined)
```

Output:

```text
from .stl: FileFormat.Stl
from path: FileFormat.Stl
joined: /output/result.txt
```

Shape

C++: `phynexis::utils::shape::Shape` **Python**: `phynexis.utils.Shape` (alias), `phynexis.utils.shape.Shape`
Header: `src/utils/shape/shape.hpp`

Base class for all geometric shapes in phynexis. Provides common interface for volume, inertia, bounding box, signed distance, surface sampling, and serialization. Subclasses implement specific geometries (sphere, cuboid, ellipsoid, etc.).

Constructor

`Shape()`

Creates a default shape (sphere-like defaults: size=1.0, volume=0.5236).

Properties

Property	Type	Access	Description
`tag`	`int`	read-only	Geometry family tag (see Tag enum)
`shape_type`	`int`	read-only	Alias for `tag`
`features`	`int`	read-only	Capability/feature flags
`name`	`str`	read-only	Registered shape name
`render_mesh`	`STLModel`	read-only	Surface mesh for rendering

Methods

`size()`

Return shape size (diameter of equal-volume sphere).

Returns: `float`

`volume()`

Return the volume of the shape.

Returns: `float`

`inertia()`

Return the inertia tensor.

Returns: `Mat3d`

`inertia_principal()`

Return principal moments of inertia as a Vec3d.

Returns: `Vec3d`

`bound_sphere_radius()`

Return bounding sphere radius for broad-phase contact detection.

Returns: `float`

`bound_aabb()` / `bound_aabb(pos, quat)`

Return axis-aligned bounding box as `(min, max)` tuple.

Parameters (overloaded):

Parameter	Type	Description
`pos`	`Vec3d`	Position offset
`quat`	`Vec4d`	Quaternion orientation

Returns: `tuple[Vec3d, Vec3d]`

`is_convex()`

Check if the shape is convex.

Returns: `bool`

`signature()`

Return a stable integer signature for solver lookup and caching.

Returns: `int`

`signed_distance(pos)`

Compute signed distance from a point to the shape surface.

Parameters:

Parameter	Type	Description
`pos`	`Vec3d`	Query point

Returns: `float`

`enclose(pos)`

Check if a point is inside the shape.

Parameters:

Parameter	Type	Description
`pos`	`Vec3d`	Query point

Returns: `bool`

`support_point(dir)`

Get the support point in a given direction.

Parameters:

Parameter	Type	Description
`dir`	`Vec3d`	Direction vector

Returns: `Vec3d`

`surface_projection(pos)`

Find the closest surface point to an intruding node.

Parameters:

Parameter	Type	Description
`pos`	`Vec3d`	Query point

Returns: `Vec3d`

`get_stl_model(num_nodes=200)`

Generate an STL model representation of the shape.

Parameters:

Parameter	Type	Default	Description
`num_nodes`	`int`	200	Number of nodes for mesh generation

Returns: `STLModel`

`update_surface_nodes()` / `update_render_mesh(num_nodes=200)` / `update_shape_properties()`

Refresh internal data (surface nodes, render mesh, cached properties).

`enable_surface_nodes()` / `disable_surface_nodes()` / `surface_nodes_enabled()`

Toggle and query surface node usage for contact solver.

`surface_node_num()` / `surface_node_spacing()`

Return surface node count and spacing.

Returns: `int` / `float`

`surface_nodes()` / `surface_node_areas()`

Access surface node positions and per-node areas.

Returns: `list[Vec3d]` / `list[float]`

`sample_surface_points(n)`

Sample random points on the shape surface.

Parameters:

Parameter	Type	Description
`n`	`int`	Number of points to sample

Returns: `list[Vec3d]`

`support_points(dir)`

Get support points (all contact-relevant points in a direction).

Parameters:

Parameter	Type	Description
`dir`	`Vec3d`	Direction vector

Returns: `list[Vec3d]`

`set_size(d)` / `set_surface_node_num(num)`

Set shape size or surface node count.

`set_surface_nodes(nodes)`

Set surface node positions directly.

Parameters:

Parameter	Type	Description
`nodes`	`list[Vec3d]`	Surface node positions

`transpose(pos)`

Translate the shape to a new position.

Parameters:

Parameter	Type	Description
`pos`	`Vec3d`	New position

`clone()`

Create a copy of the shape.

Returns: `Shape`

`check_principal()`

Check if the shape is in principal orientation.

Returns: `bool`

`print_info()`

Print shape properties to console.

`save_to(path, file, opt=SaveOptions())` / `load_from(path, file, opt=LoadOptions())`

Save/load shape to/from file.

`to_json()` / `from_json(js)`

Serialize/deserialize shape to/from JSON.

Example

```python
import phynexis

# Create a sphere of size 2.0 (radius 1.0)
s = phynexis.utils.shape.Sphere(2.0)
print("tag:", s.tag, "name:", s.name)
print("size:", s.size(), "volume:", s.volume())
print("inertia:", s.inertia())
print("bound_aabb:", s.bound_aabb())
print("is_convex:", s.is_convex())

# Signed distance and enclosure
print("sd at origin:", s.signed_distance(phynexis.utils.Vec3d(0, 0, 0)))
print("enclose(0,0,0):", s.enclose(phynexis.utils.Vec3d(0, 0, 0)))
print("enclose(2,0,0):", s.enclose(phynexis.utils.Vec3d(2, 0, 0)))

# Support point
print("support:", s.support_point(phynexis.utils.Vec3d(1, 0, 0)))

# Generate STL model
stl = s.get_stl_model(50)
print("stl size:", stl.size())

# Transform
s.transpose(phynexis.utils.Vec3d(1.0, 0.0, 0.0))
print("aabb after translate:", s.bound_aabb())
```

Output:

```text
tag: 2 name: sphere
size: 2.0 volume: 4.1887902047863905
inertia: Mat3d([1.67552, 0, 0], [0, 1.67552, 0], [0, 0, 1.67552])
bound_aabb: (Vec3d(-1, -1, -1), Vec3d(1, 1, 1))
is_convex: True
sd at origin: 1.0
enclose(0,0,0): True
```

```
enclose(2,0,0): True
support: Vec3d(1, 0, 0)
stl size: 0.18819206523895264
aabb after translate: (Vec3d(0, -1, -1), Vec3d(2, 1, 1))
```

Shape Subclasses

All subclasses inherit from `Shape` and are available under `phynexis.utils.shape`.

Sphere

```python
s = phynexis.utils.shape.Sphere()      # default size=1.0
s = phynexis.utils.shape.Sphere(2.0)   # size=2.0
```

Cuboid

```python
c = phynexis.utils.shape.Cuboid()  # default center=(0,0,0), length=(1,1,1)
```

Property	Type	Description
`center`	`Vec3d`	Center position (read/write)
`length`	`Vec3d`	Side lengths in x, y, z (read/write)

Ellipsoid

```python
e = phynexis.utils.shape.Ellipsoid()              # default axes (1,1,1)
e = phynexis.utils.shape.Ellipsoid(2.0, 3.0, 4.0) # semi-axes a, b, c
```

Property	Type	Description
`axis_a`	`float`	Semi-axis a (read/write)
`axis_b`	`float`	Semi-axis b (read/write)
`axis_c`	`float`	Semi-axis c (read/write)

Cylinder

```python
cy = phynexis.utils.shape.Cylinder()         # default r=0.5, h=1.0
cy = phynexis.utils.shape.Cylinder(1.0, 2.0) # radius=1.0, height=2.0
```

Property	Type	Description
`radius`	`float`	Cylinder radius (read/write)
`height`	`float`	Cylinder height (read/write)

Plane

```python
p = phynexis.utils.shape.Plane()
p.set_extent(10.0)
p.set_center(0, 0, 0)
p.set_normal(0, 1, 0)
```

Property	Type	Description
`center`	`Vec3d`	Plane center (read/write)
`dir_n`	`Vec3d`	Normal direction (read/write)
`extent`	`float`	Plane extent (read/write)

Triangle

```python
t = phynexis.utils.shape.Triangle()
t = phynexis.utils.shape.Triangle(v0, v1, v2)  # three Vec3d vertices
t.set_vertices(v0, v1, v2)
```

Property	Type	Description
`vertices`	`Mat3d`	3x3 matrix of vertex positions (read/write)
`dir_n`	`Vec3d`	Normal direction (read/write)

PointSphere

Degenerate sphere representing a point mass (size=1.0, volume=0.5236).

```python
ps = phynexis.utils.shape.PointSphere()
```

Other Shapes

The following shapes are also exposed but have more specialized APIs:

- `PolySuperEllipsoid` — Super-ellipsoids with polynomial corrections
- `PolySuperQuadrics` — Super-quadric shapes
- `SphericalHarmonics` — Shapes represented by spherical harmonic coefficients
- `TriMesh` — Triangle mesh shape wrapper
- `LevelSet` — Implicit surface shape (see `LevelSetField` for the underlying grid representation)
- `Polybezier` — Bezier curve-based shape
- `BoundingBox` — Axis-aligned bounding box shape

ShapeFactory

Static factory for creating shapes by name.

Method	Description
`create(name, js)`	Create a shape by registered name with JSON parameters

Note: `ShapeFactory` has no public constructor; use `ShapeFactory.create()` directly.

ShapeRegistry

Singleton registry of available shape types.

Method	Description
`instance()`	Get the singleton registry instance
`is_registered(name)`	Check if a shape type is registered
`print_info()`	Print registered shape types

Note: `get_registered_types()` returns `VecX<String>` (unregistered type); use `print_info()` as a workaround.

Example: Surface nodes and sampling

```python
import phynexis

s = phynexis.utils.shape.Sphere(2.0)
s.enable_surface_nodes()
s.update_surface_nodes()

print("surface_node_num:", s.surface_node_num())
print("surface_nodes_enabled:", s.surface_nodes_enabled())
print("surface_node_spacing:", s.surface_node_spacing())

# Access surface nodes and areas
nodes = s.surface_nodes()
areas = s.surface_node_areas()
print("num nodes:", len(nodes), "num areas:", len(areas))
print("first node:", nodes[0])

# Sample surface points
pts = s.sample_surface_points(5)
print("sampled points:", len(pts))
for p in pts:
    print(" ", p)

# Support points in a direction
supp = s.support_points(phynexis.utils.Vec3d(1.0, 0.0, 0.0))
print("support points:", supp)

# Render mesh
s.update_render_mesh(50)
rm = s.render_mesh
print("render_mesh size:", rm.size())
```

Output:

```text
surface_node_num: 1000
surface_nodes_enabled: True
surface_node_spacing: 0.12629780026048754
num nodes: 1000 num areas: 1000
first node: Vec3d(0.0191123, 0.0177418, 0.999641)
sampled points: 5
  Vec3d(0.510644, -0.334648, 0.790675)
  Vec3d(-0.547768, -0.458434, 0.699488)
  Vec3d(0.170926, 0.70943, 0.683829)
  Vec3d(-0.252001, -0.842175, 0.475395)
  Vec3d(-0.93542, -0.101945, 0.338865)
support points: [Vec3d(1, 0, 0)]
render_mesh size: 0.18819206523895264
```

Unexposed C++ API

- `Feature` enum and `FeatureMask` bitflags
- `pack_common` / `pack_specific` / `unpack_common` / `unpack_specific` (internal serialization hooks)
- `vertices_of_aabb` static method

Vectors and Matrices

C++: `phynexis::utils::Vec2`, `Vec3`, `Vec4`, `Vec6`, `Mat2`, `Mat3` **Python**: `phynexis.utils.Vec2d`, `Vec2i`, `Vec3d`, `Vec3i`, `Vec4d`, `Vec4i`, `Vec6d`, `Mat2d`, `Mat3d` **Header**: `src/utils/types/vec*.hpp`, `mat*.hpp`

Small fixed-size vector and matrix types used throughout phynexis. Exposed with Pythonic protocols (`len()`, indexing, iteration) where supported.

Vector Types

Python Name	C++ Type	Elements	Component Access	Arithmetic
`Vec2d`	`Vec2<double>`	2 floats	`x`, `y`	`+`, `-`, `*`, `/` with scalar and vector
`Vec2i`	`Vec2<int>`	2 ints	`x`, `y`	`+`, `-`
`Vec3d`	`Vec3<double>`	3 floats	`x`, `y`, `z`	`+`, `-`, `*`, `/` with scalar and vector
`Vec3i`	`Vec3<int>`	3 ints	`x`, `y`, `z`	`+`, `-`
`Vec4d`	`Vec4<double>`	4 floats	`x`, `y`, `z`, `w`	`+`, `-`, `*`, `/` with scalar and vector
`Vec4i`	`Vec4<int>`	4 ints	`x`, `y`, `z`, `w`	`+`, `-`
`Vec6d`	`Vec6<double>`	6 floats	`xx`, `yy`, `zz`, `xy`, `xz`, `yz`	—

Constructors

`Vec2d(x, y)` / `Vec2i(x, y)`

`Vec3d(x, y, z)` / `Vec3i(x, y, z)`

`Vec4d(x, y, z, w)` / `Vec4i(x, y, z, w)`

Note: The C++ `Vec4` constructor order is `(w, x, y, z)`, but the Python binding accepts `(x, y, z, w)` in the `repr` output order. After construction, `v.x`, `v.y`, `v.z`, `v.w` correspond to the expected components.

`Vec6d(xx, yy, zz, xy, xz, yz)`
Symmetric tensor storage (e.g. stress/strain components).

Properties / Indexing

Feature	Vec2d/3d/4d/2i/3i/4i/3u	Vec6d
`len(v)`	Yes	No
`v[i]` / `v[i] = val`	Yes	No
Named components	Yes	Yes
`for x in v`	Yes (iteration)	No

Methods

`dot(other)` — Vec3d only
Compute dot product.

Parameters:

Parameter	Type	Description
`other`	`Vec3d`	Other vector

Returns: `float`

`cross(other)` — Vec3d only
Compute cross product.

Parameters:

Parameter	Type	Description
`other`	`Vec3d`	Other vector

Returns: `Vec3d`

Matrix Types

Python Name	C++ Type	Shape	Row Access
`Mat2d`	`Mat2<double>`	2x2	`mat[i]` returns `Vec2d`
`Mat3d`	`Mat3<double>`	3x3	`mat[i]` returns `Vec3d`

Constructors

`Mat2d()` / `Mat3d()`
Creates a zero-initialized matrix.

Indexing

Feature	Mat2d	Mat3d
`mat[i]`	Returns `Vec2d`	Returns `Vec3d`
`mat[i] = row_vec`	Yes	Yes

Example

```python
import phynexis

# Vec3d — most commonly used
v = phynexis.utils.Vec3d(1.0, 2.0, 3.0)
print("v:", v)
print("len:", len(v))
print("x:", v.x, "y:", v.y, "z:", v.z)
print("v[0]:", v[0], "v[1]:", v[1], "v[2]:", v[2])

# Element access and mutation
v.x = 10.0
v[1] = 20.0
print("after mutation:", v)

# Arithmetic
a = phynexis.utils.Vec3d(1.0, 0.0, 0.0)
b = phynexis.utils.Vec3d(0.0, 1.0, 0.0)
print("a + b:", a + b)
print("a * 2.0:", a * 2.0)
print("2.0 * a:", 2.0 * a)
```

```python

# Dot and cross
print("dot:", a.dot(b))
print("cross:", a.cross(b))

# Vec3i
vi = phynexis.utils.Vec3i(1, 2, 3)
print("Vec3i:", vi, "len:", len(vi))

# Vec6d — symmetric tensor
s = phynexis.utils.Vec6d(1.0, 2.0, 3.0, 0.5, 0.6, 0.7)
print("Vec6d xx:", s.xx, "xy:", s.xy)

# Mat3d
m = phynexis.utils.Mat3d()
print("Mat3d:", m)
m[0] = phynexis.utils.Vec3d(1.0, 0.0, 0.0)
m[1] = phynexis.utils.Vec3d(0.0, 1.0, 0.0)
m[2] = phynexis.utils.Vec3d(0.0, 0.0, 1.0)
print("identity:", m)
```

Output:

```text
v: Vec3d(1, 2, 3)
len: 3
x: 1.0 y: 2.0 z: 3.0
v[0]: 1.0 v[1]: 2.0 v[2]: 3.0
after mutation: Vec3d(10, 20, 3)
a + b: Vec3d(1, 1, 0)
a * 2.0: Vec3d(2, 0, 0)
2.0 * a: Vec3d(2, 0, 0)
dot: 0.0
cross: Vec3d(0, 0, 1)
Vec3i: Vec3i(1, 2, 3) len: 3
Vec6d xx: 1.0 xy: 0.5
Mat3d: Mat3d([0, 0, 0], [0, 0, 0], [0, 0, 0])
identity: Mat3d([1, 0, 0], [0, 1, 0], [0, 0, 1])
```

Known Issues

- `Vec3u` (`Vec3<Int64>`) and `Vec4u` (`Vec4<Int64>`) are not exposed in Python. Use `Vec3i` / `Vec4i` instead for integer vector types.
- `Vec6d` does **not** support `len()`, indexing, or iteration. Only named component access (`xx`, `yy`, etc.) is available.
- `Vec4d` constructor: the C++ order is `(w, x, y, z)`, but Python passes `(x, y, z, w)`. The repr output `Vec4d(x, y, z, w)` matches the property names, so `Vec4d(1, 2, 3, 4)` gives `x=2, y=3, z=4, w=1` which is confusing. Prefer constructing with explicit intent or avoid `Vec4d` unless the w-component semantics are well understood.

Unexposed C++ API

- `VecX<T>` (dynamic-size vector) — used internally, some instantiations not registered for Python
- Matrix arithmetic operators (`+`, `-`, `*`) on `Mat2d`/`Mat3d`
- `data()` pointer access
- `operator+=` / `-=` / `*=` / `/=` compound assignments

FlatHashMap

C++: `phynexis::utils::FlatHashMap<Key, Value>` **Python**: `phynexis.utils.FlatHashMap` (alias for `FlatHashMap64_32`) **Header**: `src/utils/types/flat_hash_map.hpp`

Flat hash map with open addressing. Optimized for cache-friendly iteration and small data performance. Exposed as template instantiations with Pythonic dict-like interface.

Available Instantiations

Python Name	Key Type	Value Type	Description
`FlatHashMap`	`int64`	`int32`	Default alias (64→32)
`FlatHashMap64_32`	`int64`	`int32`	64-bit key, 32-bit value
`FlatHashMap64_64`	`int64`	`int64`	64-bit key, 64-bit value
`FlatHashMap32_32`	`int32`	`int32`	32-bit key, 32-bit value
`FlatHashMap64_64i`	`int64`	`int64`	Alias for `FlatHashMap64_64`

Constructor

`FlatHashMap()`

Creates an empty hash map with default capacity.

`FlatHashMap(bucket_count)`

Creates a hash map with pre-allocated bucket count.

Parameters:

Parameter	Type	Description
`bucket_count`	`int`	Initial number of buckets

Methods

`empty()`

Return True if the map contains no elements.

`size()` / `__len__()`

Return the number of key-value pairs.

`max_size()`

Return the maximum possible size.

`reserve(count)`

Pre-allocate space for at least `count` elements.

`clear()`

Remove all elements.

`insert(key, value)`

Insert a key-value pair. Returns `(inserted, value)` tuple.

`erase(key)`

Remove the element with the given key. Returns number of elements removed (0 or 1).

`find(key)`

Lookup a key. Returns `(found, value)` tuple.

`contains(key)` / `key in map`

Check if a key exists.

`count(key)`

Return 1 if key exists, 0 otherwise.

`at(key)`

Access value by key with bounds checking.

`load_factor()`

Return current load factor (size / capacity).

`max_load_factor()` / `max_load_factor(ml)`

Get or set the maximum load factor threshold for rehashing.

`rehash(count)`

Force rehash to at least `count` buckets.

Python Protocols

`map[key]` (getitem)

Access or default-insert a value.

`map[key] = value` (setitem)

Insert or update a key-value pair.

`key in map` (contains)

Check membership.

`for k, v in map` (iteration)

Iterate over key-value pairs.

`len(map)`

Return element count.

Example

```python
import phynexis

# Create default map (int64 -> int32)
m = phynexis.utils.FlatHashMap()
print("empty:", m.empty())

# Insert via setitem
m[10] = 100
m[20] = 200
m[30] = 300

print("size:", m.size())
print("m[10]:", m[10])
print("m.at(20):", m.at(20))
print("contains 10:", m.contains(10))
print("contains 99:", 99 in m)
print("count 10:", m.count(10))
print("len:", len(m))

# find
found, val = m.find(10)
print("find(10):", found, val)
found2, val2 = m.find(99)
print("find(99):", found2, val2)

# insert method
result = m.insert(40, 400)
print("insert(40, 400):", result)

# items / keys / values
print("items:", m.items())
print("keys:", m.keys())
print("values:", m.values())

# iteration
for k, v in m:
    print(f"  {k} -> {v}")

# erase
m.erase(10)
print("after erase 10, size:", m.size())

# load factor
print("load_factor:", m.load_factor())

# reserve and rehash
m.reserve(100)
print("after reserve, size:", m.size())
m.rehash(256)
print("after rehash, load_factor:", m.load_factor())

# clear
m.clear()
```

```python
print("after clear, size:", m.size())

# Bucket count constructor
m2 = phynexis.utils.FlatHashMap(64)
print("m2 size:", m2.size())

# 64->64 variant
m64 = phynexis.utils.FlatHashMap64_64()
m64[1] = 100
m64[2] = 200
print("64_64 size:", m64.size())

# 32->32 variant
m32 = phynexis.utils.FlatHashMap32_32()
m32[1] = 100
print("32_32 size:", m32.size())
```

Output:

```text
empty: True
size: 3
m[10]: 100
m.at(20): 200
contains 10: True
contains 99: False
count 10: 1
len: 3
find(10): True 100
find(99): False 0
insert(40, 400): (True, 400)
items: [(20, 200), (10, 100), (30, 300), (40, 400)]
keys: [20, 10, 30, 40]
values: [200, 100, 300, 400]
  20 -> 200
  10 -> 100
  30 -> 300
  40 -> 400
after erase 10, size: 3
load_factor: 0.046875
after reserve, size: 3
after rehash, load_factor: 0.01171875
after clear, size: 0
m2 size: 0
64_64 size: 2
32_32 size: 1
```

Unexposed C++ API

The following methods are not yet exposed to Python:

- Copy/move constructors and assignment operators
- `emplace()` / `try_emplace()` — template methods difficult to bind
- Iterator-based `erase(pos)` — iterator not fully exposed
- `cbegin()` / `cend()` — const iterators

Binding Fix Record

2026-05-03: `items()` / `keys()` / `values()` originally returned `VecX<T>` which threw `TypeError` because the type was not registered. Changed to return `py::list`. Issue fixed. See `discovery-log.md` for details.

Voronoi

C++: `phynexis::utils::voronoi` **Python**: `phynexis.utils.voronoi` **Header**: `src/utils/voronoi/*.hpp`

Voronoi tessellation on surfaces and in volumes. The Python API exposes `Tessellation` (static computation methods) and `Diagram` (result container).

TessellationParams

Configuration for Voronoi computation.

Constructor

`TessellationParams()`
Defaults: `max_iter=1000`, `tol=0.001`, `use_cork=True`.

Properties

Property	Type	Description
`max_iter`	`int`	Maximum iterations
`tol`	`float`	Convergence tolerance
`use_cork`	`bool`	Use Cork for boolean clipping

Tessellation

Static methods for computing Voronoi diagrams.

`compute(vt_seeds, stl_model, params)`

Compute Voronoi tessellation from seed points on a surface mesh.

Parameters:

Parameter	Type	Description
`vt_seeds`	`list[Vec3d]`	Seed points
`stl_model`	`STLModel`	Surface mesh
`params`	`TessellationParams`	Optional parameters

Returns: `Diagram`

`compute_centroidal(num_seeds, stl_model, params)`

Compute centroidal Voronoi tessellation (CVT) with automatic seed placement.

Parameters:

Parameter	Type	Description
`num_seeds`	`int`	Number of seeds
`stl_model`	`STLModel`	Surface mesh
`params`	`TessellationParams`	Optional parameters

Returns: `Diagram`

Diagram

Result container for tessellation (no public constructor).

Example

```python
import phynexis

# Create a mesh from a sphere shape
model = phynexis.utils.shape.Sphere(2.0).get_stl_model(200)

# Compute CVT
params = phynexis.utils.voronoi.TessellationParams()
params.max_iter = 100
params.tol = 0.01

diag = phynexis.utils.voronoi.Tessellation.compute_centroidal(5, model, params)
print("diagram:", diag)
```

Output:

```text
diagram: <phynexis.lib.pyutils.voronoi.Diagram object at 0x...>
```

Unexposed C++ API

- `SphericalDiagram`, `SphericalTessellation` — Partially bound
- `Tessellation::compute` overload with implicit surfaces

vtk

C++: `phynexis::utils::vtk` **Python**: `phynexis.utils.vtk` **Header**: `src/utils/vtk/*.hpp`

VTK file I/O utilities for reading and writing point data, surface meshes, tetrahedral meshes, and line sets. Supports legacy VTK and VTK-HDF formats.

Data Classes

Class	Description
`VtkData`	Container for VTK dataset with points, cells, point data, cell data
`VtkTopology`	Cell connectivity and type information
`PointData`	Scalar/vector data attached to mesh points
`CellData`	Scalar/vector data attached to mesh cells

Read Functions

Function	Description
`read_points(filename)`	Read point cloud from VTK
`read_surface(filename)`	Read surface mesh from VTK
`read_tetrahedra(filename)`	Read tetrahedral mesh from VTK
`read_lines(filename)`	Read line/polygon data from VTK

Write Functions

Function	Description
`write_points(filename, points, point_data=None)`	Write point cloud to VTK
`write_surface(filename, vertices, facets, point_data=None, cell_data=None)`	Write surface mesh to VTK
`write_tetrahedra(filename, vertices, tets, point_data=None, cell_data=None)`	Write tetrahedral mesh to VTK
`write_with_lines(filename, vertices, lines, point_data=None)`	Write line set to VTK

Format Detection

Function	Description
`is_vtkhdf_format(filename)`	Check if file is VTK-HDF format
`detect_vtkhdf_format(filename)`	Detect VTK-HDF format variant

Example

```python
import phynexis

# Format detection
print("is_vtkhdf:", phynexis.utils.vtk.is_vtkhdf_format("/path/to/data.vtkhdf"))

# Data classes exist but their constructors and methods may require
# additional VecX type registrations for full functionality.
```

Conversion Functions

Top-level functions for converting between VTK data and phynexis types.

`to_vtk(vtk_data, model)`

Populate a `VtkData` object from an `STLModel` or `Shape`.

Parameters:

Parameter	Type	Description
`vtk_data`	`VtkData`	Target VTK data container (modified in-place)
`model`	`STLModel` or `Shape`	Source model

`from_vtk(vtk_data, model)`

Populate an `STLModel` from a `VtkData` object.

Parameters:

Parameter	Type	Description
`vtk_data`	`VtkData`	Source VTK data
`model`	`STLModel`	Target model (modified in-place)

Example:

```python
import phynexis

# Create a model from scratch
model = phynexis.utils.STLModel()
model.vertices = [
    phynexis.utils.Vec3d(0.0, 0.0, 0.0),
    phynexis.utils.Vec3d(1.0, 0.0, 0.0),
    phynexis.utils.Vec3d(0.0, 1.0, 0.0),
]
model.facets = [phynexis.utils.Vec3i(0, 1, 2)]
print("original vertices:", len(model.vertices))

# Convert to VtkData
out_data = phynexis.utils.vtk.VtkData()
phynexis.utils.to_vtk(out_data, model)
print("vtk points:", len(out_data.points()))

# Convert back from VtkData to STLModel
model2 = phynexis.utils.STLModel()
phynexis.utils.from_vtk(out_data, model2)
print("restored vertices:", len(model2.vertices))
print("restored facets:", len(model2.facets))
```

Output:

```text
original vertices: 3
vtk points: 3
restored vertices: 3
restored facets: 1
```

Known Issues

- `VtkData` constructor and property access (`points()`, `facets()`, etc.) may fail if called on an uninitialized object. Always populate via `to_vtk()` or `read_surface()` before accessing data.

Unexposed C++ API

- `VtkReader` / `VtkWriter` internal implementations
- `VtkHDF` reader/writer specialized for HDF5 backend
- `format_detection` internals

wrappers

C++: `phynexis::utils::wrappers` **Python**: `phynexis.utils.wrappers` **Header**: `src/utils/wrappers/*.hpp`

Thin wrappers around external libraries (Cork, CGAL, Eigen, igl, TetGen) providing mesh boolean operations, linear algebra, mesh processing, and tetrahedralization.

CorkWrapper

Boolean mesh operations via the Cork library.

Method	Description
`mesh_union(m1, m2)`	Union of two meshes
`mesh_intersect(m1, m2)`	Intersection of two meshes
`mesh_difference(m1, m2)`	Difference (m1 − m2)
`mesh_xor(m1, m2)`	Symmetric difference

Note: `CorkWrapper` has no default constructor exposed. Access methods via the class itself (static/class methods).

EigenWrapper

Linear algebra utilities via Eigen.

Method	Description
`solve_linear_eqn(A, b)`	Solve Ax = b
`get_eigen_vector(A, index)`	Get eigenvector by index

IGLWrapper

Mesh processing utilities via libigl.

Method	Description
`convex_hull(vertices, facets)`	Compute convex hull
`mesh_decimate(vertices, facets, target)`	Mesh simplification
`mesh_refine(vertices, facets)`	Mesh subdivision
`reorient_facets(vertices, facets)`	Ensure consistent facet orientation
`remove_duplicate_vertices(vertices, facets)`	Remove duplicate vertices
`remove_unreferenced_vertices(vertices, facets)`	Clean unreferenced vertices
`marching_cubes(field, corner, spacing, isovalue)`	Extract isosurface
`mesh_intersect(ray_origin, ray_dir, vertices, facets)`	Ray-mesh intersection
`facet_components(vertices, facets)`	Connected component labeling
`check_winding(vertices, facets)`	Check winding order
`get_points_inside_mesh(points, vertices, facets)`	Point-in-mesh test
`tetmesh_boundary(vertices, tets)`	Extract boundary surface from tetrahedral mesh

CGALWrapper

Computational geometry via CGAL.

Method	Description
`get_alpha_shape(points, alpha)`	Compute alpha shape
`get_tetmesh(vertices, facets)`	Tetrahedral mesh generation
`smooth_mesh(vertices, facets, iterations)`	Mesh smoothing

mpi

MPI wrapper utilities (if MPI is available).

Function	Description
`comm_rank()`	Get the rank of the current process
`comm_size()`	Get total number of MPI processes
`is_initialized()`	Check if MPI has been initialized

Returns: `int` / `bool`

TetGenOptions

Configuration object for tetrahedral mesh generation.

Constructor

`TetGenOptions()`
Default: `max_volume=0.0`, `min_dihedral=0.0`, `verbose=False`, `quiet=True`, `no_boundary_split=False`, `refine=False`.

Properties

Property	Type	Access	Description
`max_volume`	`float`	read/write	Maximum tetrahedron volume (0 = no limit)
`min_dihedral`	`float`	read/write	Minimum dihedral angle constraint
`verbose`	`bool`	read/write	Enable verbose output
`quiet`	`bool`	read/write	Suppress console output
`no_boundary_split`	`bool`	read/write	Preserve boundary facets
`refine`	`bool`	read/write	Refine existing mesh

Static Methods

`default_options()`
Return a default options instance.

`quality()`
Return options preset for quality mesh generation.

`volume_constrained()`
Return options preset for volume-constrained meshing.

`from_mesh_size()`
Return options preset derived from mesh size.

`tetgen_get_tetmesh(vertices, facets, options)`

Generate a tetrahedral mesh from a surface mesh using TetGen.

Parameters:

Parameter	Type	Description
`vertices`	`list[Vec3d]`	Surface mesh vertices
`facets`	`list[Vec3i]`	Surface triangle indices
`options`	`TetGenOptions`	Optional (default: `default_options()`)

Returns: `tuple(list[Vec3d], list[Vec4i])` — Tetrahedral mesh (vertices, tets)

Example

```python
import phynexis

# List available methods
print("CorkWrapper:", [x for x in dir(phynexis.utils.wrappers.CorkWrapper) if not x.startswith("_")])
print("IGLWrapper:", [x for x in dir(phynexis.utils.wrappers.IGLWrapper) if not x.startswith("_")])
print("CGALWrapper:", [x for x in dir(phynexis.utils.wrappers.CGALWrapper) if not x.startswith("_")])

# TetGenOptions usage
opts = phynexis.utils.wrappers.TetGenOptions()
print("TetGenOptions defaults:", opts.max_volume, opts.verbose, opts.quiet)

# Presets
quality_opts = phynexis.utils.wrappers.TetGenOptions.quality()
print("quality preset max_volume:", quality_opts.max_volume)

# mpi info
print("mpi initialized:", phynexis.utils.wrappers.mpi.is_initialized())
```

Output:

```text
CorkWrapper: ['mesh_difference', 'mesh_intersect', 'mesh_union', 'mesh_xor']
IGLWrapper: ['check_winding', 'convex_hull', 'facet_components', 'get_points_inside_mesh', 'marching_cubes', 'mesh_decimate', 'mesh_intersect', 'mesh_refine', 'remove_duplicate_vertices', 'remove_unreferenced_vertices', 'reorient_facets', 'tetmesh_boundary']
CGALWrapper: ['get_alpha_shape', 'get_tetmesh', 'smooth_mesh']
TetGenOptions defaults: 0.0 False True
quality preset max_volume: 0.0
mpi initialized: False
```

Known Issues

- `CorkWrapper` has no default constructor. Methods must be called as `CorkWrapper.mesh_union(...)` rather than creating an instance.
- Most wrapper methods use `VecX<Vec3d>&` (mutable reference) arguments in their lambda bindings. Even though `VecX<T>` type_caster is now registered, pybind11 cannot bind a temporary converted object to a non-const reference. Methods like `IGLWrapper.convex_hull`, `CorkWrapper.mesh_union`, etc. require binding fixes to accept values or use output tuples instead of output reference parameters.
- `TetGenOptions` is exposed but its constructor and properties may have limited binding coverage.

Unexposed C++ API

- `EigenAdapter` / `EigenWrapper` internal matrix conversion helpers
- `mpi` wrapper full MPI collective operations

Cork

C++: `phynexis::utils::wrappers::CorkWrapper` **Python**: `phynexis.utils.wrappers.CorkWrapper` **Header**: `src/utils/wrappers/cork_wrapper.hpp`

Boolean operations on triangle meshes (intersection, union, difference, XOR) via the Cork library.

Description

`CorkWrapper` has no default constructor in Python. It is typically accessed through the `wrappers` submodule or used via the `Tessellation` class when `use_cork=True`.

Methods

`mesh_intersect(v1, f1, v2, f2, out_v, out_f, out_info)`

Compute intersection of two meshes.

`mesh_union(v1, f1, v2, f2, out_v, out_f, out_info)`

Compute union of two meshes.

`mesh_difference(v1, f1, v2, f2, out_v, out_f, out_info)`

Compute difference (mesh1 − mesh2).

`mesh_xor(v1, f1, v2, f2, out_v, out_f, out_info)`

Compute symmetric difference.

Parameters:

Parameter	Type	Description
`v1, v2`	`list[Vec3d]`	Input mesh vertices
`f1, f2`	`list[Vec3i]`	Input mesh faces
`out_v`	`list[Vec3d]`	Output vertices (inout)
`out_f`	`list[Vec3i]`	Output faces (inout)
`out_info`	`list[int]`	Output labels (inout, optional)

Note

- The overload without `out_info` is also available.
- A specialized `mesh_intersect` with tolerance and direction exists for ray-like intersection tests.
- `CorkWrapper` instance access in Python is currently limited. Consider using STLModel-based boolean operations if available.

Unexposed C++ API

- Default constructor for direct instantiation

Console

C++: `phynexis::utils::console` **Python**: `phynexis.utils` **Header**: `src/utils/console.hpp`

Console logging and output control for phynexis. Supports leveled output (debug/info/warning/error/fatal), colorized terminal output, timestamps, and MPI rank display.

Level

C++: `phynexis::utils::console::Level` **Python**: `phynexis.utils.Level`

Enum controlling the minimum log level for output filtering.

Value	Numeric	Description
`Level.DEBUG`	0	Debug messages
`Level.INFO`	1	Informational messages (default minimum)
`Level.WARNING`	2	Warning messages
`Level.ERROR`	3	Error messages
`Level.FATAL`	4	Fatal error messages

MPIOutputMode

C++: `phynexis::utils::console::MPIOutputMode` **Python**: `phynexis.utils.MPIOutputMode`

Enum controlling MPI output behavior.

Value	Description
`MPIOutputMode.ONLY_MASTER`	Only rank 0 outputs (default)
`MPIOutputMode.ALL_WITH_RANK`	All ranks output with rank prefix

Config

C++: `phynexis::utils::console::Config` **Python**: `phynexis.utils.Config`

Configuration object controlling the appearance and filtering of console output.

Constructor

`Config()`
Creates a config with default settings (min_level=info, timestamps on, colors on).

Static Methods

`Config.debug()`
Create a Config with `min_level=DEBUG` for verbose debugging output.

`Config.release()`
Create a Config with `min_level=WARNING` for production/release output (only warnings and above).

`Config.verbose()`
Create a Config with `min_level=DEBUG` and `show_location=True` for detailed debug output.

Properties

Property	Type	Access	Description
`min_level`	`Level`	read/write	Minimum log level to display
`show_timestamp`	`bool`	read/write	Show timestamps in output
`show_level`	`bool`	read/write	Show level tag (INFO, WARN, etc.)
`show_location`	`bool`	read/write	Show file/line location
`color_output`	`bool`	read/write	Enable ANSI color codes
`show_rank`	`bool`	read/write	Show MPI rank prefix

Methods

`set_min_level(level)`
Set minimum display level. Returns the Config object for chaining.

Parameters:

Parameter	Type	Description
`level`	`Level`	Minimum level to display

`set_show_timestamp(enabled)`
Toggle timestamp display. Returns the Config object for chaining.

`set_show_level(enabled)`
Toggle level tag display. Returns the Config object for chaining.

`set_show_location(enabled)`
Toggle source location display. Returns the Config object for chaining.

`set_color_output(enabled)`
Toggle colorized output. Returns the Config object for chaining.

`set_show_rank(enabled)`
Toggle MPI rank prefix display. Returns the Config object for chaining.

Example:

```python
import phynexis

cfg = phynexis.utils.custom()
cfg.show_timestamp = False
cfg.color_output = False

phynexis.utils.info("no timestamp, no color")

# Config factory methods
debug_cfg = phynexis.utils.Config.debug()
```

```
release_cfg = phynexis.utils.Config.release()
print("debug min_level:", debug_cfg.min_level)
print("release min_level:", release_cfg.min_level)
```

Output:

```text
[INFO] no timestamp, no color
debug min_level: Level.DEBUG
release min_level: Level.WARNING
```

Logging Functions

Module-level functions for leveled console output.

`debug(msg)`

Print a debug-level message. Only visible when min_level ≤ 0.

`info(msg)`

Print an info-level message.

`plain(msg)`

Print a plain message without any level prefix or formatting.

`warning(msg)`

Print a warning-level message.

`error(msg)`

Print an error-level message.

`fatal(msg)`

Print a fatal-level message and terminate.

Parameters:

Parameter	Type	Description
`msg`	`str`	Message string to print

Broadcast Variants

Each level has an `_all` variant that broadcasts to all MPI ranks:

- `debug_all(msg)`
- `info_all(msg)`
- `warning_all(msg)`
- `error_all(msg)`
- `fatal_all(msg)`

Example:

```python
import phynexis

phynexis.utils.info("simulation started")
phynexis.utils.debug("debug info: dt=1e-5")
phynexis.utils.warning("mesh quality low")
phynexis.utils.plain("raw output without prefix")
```

Output:

```text
[INFO] [12:01:05.232] simulation started
[DEBUG] [12:01:05.233] debug info: dt=1e-5
[WARNING] [12:01:05.233] mesh quality low
raw output without prefix
```

Control Functions

`set_level(level)`

Set the global minimum log level.

Parameters:

Parameter	Type	Description
`level`	`Level`	Minimum level, e.g. `Level.WARNING`

`set_verbose(enabled)`

Enable or disable verbose (debug) output. Equivalent to `set_level(0)`.

`set_color(enabled)`

Enable or disable colorized output globally.

`custom()`

Return a reference to the global `Config` object. Modifications to the returned object take effect immediately.

Returns: `Config` — global configuration (mutable reference)

Example:

```python
import phynexis

phynexis.utils.set_level(phynexis.utils.Level.WARNING)  # Only warning and above
phynexis.utils.info("hidden")
phynexis.utils.warning("visible")
```

Output:

```text
[WARNING] visible
```

FPE Traps

C++: `phynexis::utils::fpe` **Python**: `phynexis.utils` **Header**: `src/utils/fpe_trap.hpp`

Floating-point exception (FPE) trap utilities for debugging numerical issues. When enabled, the program catches floating-point exceptions (division by zero, invalid operations, overflow) and prints a backtrace with source locations.

All functions are prefixed with `fpe_` to avoid name collisions.

Functions

`fpe_enable_traps()`

Enable hardware floating-point exception traps for: - Division by zero - Invalid operations (e.g. `sqrt(-1)`) - Overflow

Note: On macOS, hardware traps may not be available; signal handlers still work.

`fpe_enable_traps_for_thread()`

Enable FPE traps for the current thread. In multi-threaded programs (e.g. OpenMP), each worker thread should call this after creation.

`fpe_hardware_traps_available()`

Check if hardware FPE traps are available on this platform.

Returns: `bool`

`fpe_init(enable)`

Initialize the FPE trap mechanism: install the SIGFPE handler and optionally enable hardware traps.

Parameters:

Parameter	Type	Description
`enable`	`bool`	If True, install handler and enable traps

`fpe_init_from_env()`

Initialize FPE traps from the `PHYNEXIS_SIGFPE` environment variable. FPE traps are enabled by default; set `PHYNEXIS_SIGFPE=0` to disable. Similar to OpenFOAM's `FOAM_SIGFPE` but with opposite default.

Returns: `bool` — True if traps were enabled

`fpe_install_sigfpe_handler()`

Install the SIGFPE signal handler that prints a backtrace on floating-point exceptions.

`fpe_print_backtrace()`

Print a backtrace to stderr for debugging purposes.

Example

```python
import phynexis

# Initialize FPE traps from environment
enabled = phynexis.utils.fpe_init_from_env()
print("FPE traps enabled:", enabled)
print("Hardware traps available:", phynexis.utils.fpe_hardware_traps_available())

# Install handler and enable traps
phynexis.utils.fpe_install_sigfpe_handler()
phynexis.utils.fpe_enable_traps()
print("FPE protection active")
```

Output:

```text
FPE traps enabled: True
```

```
2 Hardware traps available: True
3 FPE protection active
```

Unexposed C++ API

- `sigfpe_handler(sig, info, ucontext)` — Raw signal handler (takes C signal types, not Python-bindable)

LevelSetField

C++: `phynexis::utils::LevelSetField` **Python**: `phynexis.utils.LevelSetField` **Header**: `src/utils/level_set_field.hpp`

Represents a level set function ϕ on a regular 3D grid. Used for implicit surface representation, interface tracking, and shape operations.

Constructor

`LevelSetField()`

Creates a level set field with default grid parameters: - Corner: `(-0.5, -0.5, -0.5)` - Spacing: `0.05` - Dimensions: `(21, 21, 21)`

Methods

`get_corner()`

Return the grid corner position.

Returns: `Vec3d`

`get_spacing()`

Return the grid spacing.

Returns: `float`

`get_dimensions()`

Return the grid dimensions.

Returns: `Vec3i`

`set_corner(corner)`

Set the grid corner.

Parameters:

Parameter	Type	Description
`corner`	`Vec3d`	Lower corner of the grid

`set_spacing(sp)`

Set the grid spacing.

Parameters:

Parameter	Type	Description
`sp`	`float`	Cell spacing

`set_dimension(dim)`

Set the grid dimensions.

Parameters:

Parameter	Type	Description
`dim`	`Vec3i`	Number of cells in each dimension

`get_phi(id)` / `set_phi(id, value)` / `phi(i, j, k)`

Access level set values at grid cells.

Parameters:

Parameter	Type	Description
`id`	`Vec3i`	Grid cell index
`i, j, k`	`int`	Individual indices
`value`	`float`	New phi value

`init_from_sdf_calculator(sdf)`

Initialize phi values by sampling an `SDFCalculator`.

`signed_distance(pos)`

Interpolate signed distance at an arbitrary point in space.

`gradient_interpolate(pos)`

Interpolate gradient at a point.

`gradient_minus(id)` / `gradient_plus(id)`

Compute gradient at a grid point using backward/forward differences.

`reinitialization()` / `reinitialization(iterations, dt)`

Reinitialize the level set function to maintain signed distance property.

`is_initialized()`

Check if phi_table has been allocated.

Returns:`bool`

`index_3d(i, j, k)`

Compute the flat array index for a 3D grid coordinate.

Parameters:

Parameter	Type	Description
i	int	First dimension index
j	int	Second dimension index
k	int	Third dimension index

Returns: `int` — Flat array index

`get_phi_table()`

Return a reference to the internal phi table (flat `VecX<double>` array).

Returns: `list[float]` — Internal phi values

Example

```python
import phynexis

# Create and configure grid
lsf = phynexis.utils.LevelSetField()
lsf.set_corner(phynexis.utils.Vec3d(0.0, 0.0, 0.0))
lsf.set_spacing(0.1)
lsf.set_dimension(phynexis.utils.Vec3i(11, 11, 11))

print("corner:", lsf.get_corner())
print("spacing:", lsf.get_spacing())
print("dimensions:", lsf.get_dimensions())

# Set phi values
lsf.set_phi(phynexis.utils.Vec3i(5, 5, 5), 1.0)
print("phi at center:", lsf.phi(5, 5, 5))

# Query signed distance
pos = phynexis.utils.Vec3d(0.5, 0.5, 0.5)
print("sd at", pos, ":", lsf.signed_distance(pos))
```

Output:

```text
corner: Vec3d(0, 0, 0)
spacing: 0.1
dimensions: Vec3i(11, 11, 11)
phi at center: 1.0
sd at Vec3d(0.5, 0.5, 0.5) : -0.5
```

Unexposed C++ API

- `get_phi_array()` — Returns `FlatArray3D<double>` (template type, not directly exposable)

MiniMap

C++: `phynexis::utils::MiniMap<Int64, Int64>` **Python**: `phynexis.utils.MiniMapIdx64` **Header**: `src/utils/mini_map.hpp`

Lightweight hash map optimized for index-to-index lookups. Uses flat hash map internally for cache-friendly access.

Constructor

`MiniMapIdx64()`

Creates an empty map.

Methods

`exist(key)`

Check if a key exists in the map.

Parameters:

Parameter	Type	Description
`key`	`int`	Key to look up

Returns: `bool` — True if key exists

`size()`

Return the number of entries.

Returns: `int` — Entry count

`clear()`

Remove all entries.

Python Protocols

`map[key]` (getitem)

Retrieve value by key. Raises `KeyError` if key not found.

`map[key] = value` (setitem)

Insert or update a key-value pair.

`key in map` (contains)

Check membership. Equivalent to `exist(key)`.

Example

```python
import phynexis

m = phynexis.utils.MiniMapIdx64()

# Insert entries
m[10] = 100
m[20] = 200
m[30] = 300

print(f"size: {m.size()}")
print(f"m[10] = {m[10]}")
print(f"20 in map: {20 in m}")
```

```python
print(f"99 in map: {99 in m}")
print(f"exist(10): {m.exist(10)}")

# Update
m[10] = 999
print(f"updated m[10] = {m[10]}")

# Clear
m.clear()
print(f"after clear: {m.size()}")
```

Output:

```text
size: 3
m[10] = 100
20 in map: True
99 in map: False
exist(10): True
updated m[10] = 999
after clear: 0
```

Unexposed C++ API

`MiniMap` is a template class. Only the `Int64→Int64` instantiation (`MiniMapIdx64`) is exposed. Other instantiations (e.g., `int→double`) are not available in Python.

SDFCalculator

C++: `phynexis::utils::SDFCalculator` **Python**: `phynexis.utils.SDFCalculator` **Header**: `src/utils/sdf_calculator.hpp`

Computes signed distance fields (SDFs) from triangle meshes using an AABB tree accelerated by libigl. Supports point queries, surface projections, closest facet lookups, and batch evaluation.

Constructor

`SDFCalculator()`

Creates an empty calculator. Must be initialized via `init_from_stl()` before use.

Methods

`init_from_stl(vertices, facets)`

Initialize from raw vertex and face arrays.

Parameters:

Parameter	Type	Description
`vertices`	`list[Vec3d]`	Mesh vertex positions
`facets`	`list[Vec3i]`	Triangle face indices

`init_from_stl(stl_model)`

Initialize from an `STLModel` object.

Parameters:

Parameter	Type	Description
`stl_model`	`STLModel`	Loaded STL model

`initialize()`

Reset to empty state.

`signed_distance(pos)`

Compute signed distance at a point. Negative = inside, positive = outside (depending on mesh orientation).

Parameters:

Parameter	Type	Description
`pos`	`Vec3d`	Query point

Returns: `float` — Signed distance

`surface_projection(pos)`

Find the closest surface point on the mesh.

Parameters:

Parameter	Type	Description
`pos`	`Vec3d`	Query point

Returns: `Vec3d` — Closest surface point

`closest_facet(pos)`

Find the index of the closest triangle facet.

Parameters:

Parameter	Type	Description
`pos`	`Vec3d`	Query point

Returns: `int` — Facet index

`signed_distances(points)`

Batch compute signed distances for multiple points.

Parameters:

Parameter	Type	Description
`points`	`list[Vec3d]`	Query points

Returns: `list[float]` — Signed distances

Example

```python
import phynexis

```

```python
# Create a mesh from a sphere shape
model = phynexis.utils.shape.Sphere(2.0).get_stl_model(200)

# Build SDF
sdf = phynexis.utils.SDFCalculator()
sdf.init_from_stl(model)

# Query points
center = phynexis.utils.Vec3d(0.0, 0.0, 0.0)
outside = phynexis.utils.Vec3d(2.0, 0.0, 0.0)

print("sd at center:", sdf.signed_distance(center))
print("sd at (2,0,0):", sdf.signed_distance(outside))
```

Output:

```text
sd at center: 0.4967424702987953
sd at (2,0,0): -1.5008614999652305
```

Unexposed C++ API

The following constructors and methods are not yet exposed:

- `SDFCalculator(const SDFCalculator&)` — Copy constructor
- `SDFCalculator(SDFCalculator&&)` — Move constructor
- Assignment operators

StringUtils

C++: `phynexis::utils::to_string` **Python**: `phynexis.utils.to_string` **Header**: `src/utils/string_utils.hpp`

Numeric-to-string conversion with consistent formatting.

Functions

`to_string(value)`

Convert a numeric value to a formatted string.

Overloads:

Signature	Description
`to_string(value: int) -> str`	Integer formatting
`to_string(value: float) -> str`	Scientific notation for doubles

Parameters:

Parameter	Type	Description
`value`	`int` | `float`	Numeric value to convert

Returns: `str` — Formatted string

Example:

```python
import phynexis

print(phynexis.utils.to_string(42))
print(phynexis.utils.to_string(3.14))
print(phynexis.utils.to_string(1e-6))
print(phynexis.utils.to_string(-2.5e3))
```

Output:

```text
42
3.140000e+00
1.000000e-06
-2.500000e+03
```

Unexposed C++ API

The following C++ string utilities are not yet exposed to Python:

- `SplitString` / `JoinString` — string split/join
- `Trim` / `ToLower` / `ToUpper` — string transforms
- `FormatString` — formatted string (C++ fmt style)

TimeUtils

C++: `phynexis::utils::get_time_micros` **Python:** `phynexis.utils.get_time_micros` **Header:** `src/utils/time_utils.hpp`

High-resolution time measurement utilities.

Functions

`get_time_micros()`

Returns the current time in microseconds since an arbitrary epoch. Useful for measuring elapsed time between two calls.

Returns: `int` — Time in microseconds

Example:

```python
import phynexis
import time

t0 = phynexis.utils.get_time_micros()
time.sleep(0.1)  # 100 ms
t1 = phynexis.utils.get_time_micros()

elapsed_ms = (t1 - t0) / 1000.0
print(f"elapsed: {elapsed_ms:.2f} ms")
```

Output:

```text
elapsed: 100.12 ms
```

Unexposed C++ API

- `Timer` class — Scoped RAII timer with automatic logging
- `SleepMicros` / `SleepMillis` — Sleep functions

phynexis.fields

Python: `phynexis.fields` **pybind module**: `pyfields`

Field data structures for structured and unstructured grids. Provides typed scalar/vector fields, field layouts, views, schemas, and mathematical operators.

Module Status

Class	Status	Doc
`FieldMeta`	done	field-meta
`FieldType`	done	field-meta
`ValueType`	done	field-meta
`Field` / `ScalarField` / `Int32Field` / `Int64Field` / `BoolField`	done	field
`FieldHolder`	done	field-holder
`FieldBase`	done	field-base
`FieldLayout`	done	field-layout
`FieldManager`	done	field-manager
`FieldSlot`	done	field-slot
`FieldSchema`	done	field-schema
`CSRMatrix`	done	csr-matrix
`FieldViewBase` / `ScalarFieldView` / `Vec3FieldView` / `Vec4FieldView` / `Vec6FieldView` / `VecNFieldView` / `VecXFieldView`	done	field-views
`LinkedFieldView` / `LinkedFieldRowView`	done	linked-field-view
`Operators` (math, reduction, prefix_sum, sort, radix_sort)	done	operators
`I/O and Utilities` (VTK, binary, JSON, MPI, console)	done	io-utils
Others	pending	—

Submodules

- **core** — `Field`, `FieldMeta`, `FieldLayout`, `FieldManager`, `FieldSchema`, `FieldSlot`, `FieldBase`, `FieldHolder`
- **containers** — `VecXField`, `CSRMatrix`
- **views** — `ScalarFieldView`, `Vec3FieldView`, `VecXFieldView`, `LinkedFieldView`, `FieldViewBase`
- **schema** — `ScalarFieldSchema`, `Vec3FieldSchema`, `Vec4FieldSchema`, `Vec6FieldSchema`, `VecNFieldSchema`, `VecXFieldSchema`
- **operators** — Math, reduction, prefix sum, sort, radix sort

- **utils** — I/O, MPI, console output

See also

- Manual home — installation, simulations, visualization
- Python API overview — all submodules at a glance
- Deprecated flat references — legacy v0 pages

Field / ScalarField

C++: `phynexis::fields::Field<T>` **Python**: `phynexis.fields.ScalarField` / `Int32Field` / `Int64Field` / `BoolField` **Header**: `src/fields/core/field.hpp`

Concrete one-dimensional homogeneous field container, the workhorse type of the `fields` module. The C++ template `Field<T>` is exposed in Python as one non-templated class per primitive value type. `Field` is an alias for `ScalarField` (i.e. `Field<double>`), and `Idx32Field` / `Idx64Field` are aliases for `Int32Field` / `Int64Field`.

Python class	C++ instantiation	Element type
`ScalarField` (alias `Field`)	`Field<double>`	`float` (64-bit)
`Int32Field` (alias `Idx32Field`)	`Field<Int32>`	`int` (32-bit)
`Int64Field` (alias `Idx64Field`)	`Field<Int64>`	`int` (64-bit)
`BoolField`	`Field<bool>`	`bool`

Constructors

Signature	Description
`ScalarField()`	Empty field, no name.
`ScalarField(size)`	Default-initialised field of length `size`.
`ScalarField(size, value)`	Field of length `size` filled with `value`.
`ScalarField(name, size=0)`	Named field, optionally pre-sized.
`ScalarField(name, size, value)`	Named field, pre-sized and filled.
`ScalarField(data, name)`	Build from a `VecX<T>` buffer with a name.
`ScalarField(other)`	Copy constructor.

The `ScalarField(data: VecX<T>)` and `ScalarField(name, data: VecX<T>)` overloads exist in the binding but require a `VecX<T>` instance, which is not exposed in Python. Use `from_numpy` (see below) for the numpy round-trip instead.

Example:

```python
import phynexis

F = phynexis.fields

print(repr(F.ScalarField()))
print(repr(F.ScalarField(5)))
print(repr(F.ScalarField(5, 3.14)))
print(repr(F.ScalarField("rho", 5)))
print(repr(F.ScalarField("rho", 5, 1000.0)))
```

Output:

```text
ScalarField(size=0, name='')
ScalarField(size=5, name='')
ScalarField(size=5, name='')
ScalarField(size=5, name='rho')
ScalarField(size=5, name='rho')
```

NumPy Interop

`to_numpy()`

Returns a 1-D `numpy.ndarray` with a **copy** of the field data.

Returns: `numpy.ndarray` — same dtype as the field element type.

class method `from_numpy(array)`

Creates a new field whose contents are copied from `array`. The element type is fixed by the binding type, so the dtype must match (`float64` for `ScalarField`, `int32` for `Int32Field`, etc.).

Parameters:

Parameter	Type	Description
`array`	`numpy.ndarray`	1-D array; dtype must match the field type

Example:

```python
import numpy as np
import phynexis

F = phynexis.fields

sf = F.ScalarField.from_numpy(np.linspace(0.0, 1.0, 5))
print(repr(sf))
print(sf.to_numpy())
```

Output:

```text
ScalarField(size=5, name='')
[0.   0.25 0.5  0.75 1.  ]
```

Element Access

Method / operator	Description
`field[idx]`	Read/write element. **No bounds check** — out-of-range indices read uninitialised memory.
`len(field)`	Equivalent to `field.size()`.
`at(index)`	Bounds-checked read/write. Raises `IndexError` if out of range.
`at_const(index)`	Bounds-checked read-only access.
`data()`	Returns the value at index 0 (not a buffer). Mostly useful as a presence check.

Example:

```python
import phynexis

F = phynexis.fields

sf = F.ScalarField("v", 4, 0.0)
sf[0], sf[1], sf[2], sf[3] = 1.0, 2.0, 3.0, 4.0
print("len(sf):", len(sf))
print("sf[2]:", sf[2])
print("sf.at(2):", sf.at(2))

try:
    sf.at(99)
except IndexError as e:
    print("sf.at(99):", repr(e))
```

Output:

```text
len(sf): 4
sf[2]: 3.0
sf.at(2): IndexError('Field index out of range')
```

`field[idx]` does not validate the index. Negative indices or indices beyond `size()` will return arbitrary memory contents instead of raising. Use `field.at(idx)` whenever the index is not statically known.

Capacity

Method	Description
`size()`	Number of elements currently stored.
`empty()`	`True` if `size() == 0`.
`capacity()`	Allocated buffer size; always `>= size()`.
`resize(new_size)`	Grow / shrink to `new_size`; new elements are default-initialised.
`resize(new_size, value)`	Grow to `new_size`, padding with `value`.
`reserve(capacity)`	Pre-allocate to avoid reallocation. Does not change `size()`.
`clear()`	Sets `size()` to 0. Capacity is retained.
`push_back(value)`	Append a single element.
`pop_back()`	Remove the last element.

Example:

```python
import phynexis

F = phynexis.fields

sf = F.ScalarField()
sf.reserve(100)
print("after reserve(100): size=", sf.size(), "capacity=", sf.capacity())

sf.push_back(1.0)
sf.push_back(2.0)
print("after 2 push_back:", sf.to_numpy(),
      "size=", sf.size(), "capacity=", sf.capacity())

sf.pop_back()
print("after pop_back:", sf.to_numpy())

sf.clear()
```

```
print("after clear: size=", sf.size(), "capacity=", sf.capacity())

sf.resize(3)
print("resize(3):", sf.to_numpy())

sf.resize(5, 9.0)
print("resize(5, 9.0):", sf.to_numpy())
```

Output:

```text
after reserve(100): size= 0 capacity= 100
after 2 push_back: [1. 2.] size= 2 capacity= 100
after pop_back: [1.]
after clear: size= 0 capacity= 100
resize(3): [0. 0. 0.]
resize(5, 9.0): [0. 0. 0. 9. 9.]
```

Bulk Operations

Method	Description
`fill(value)`	Set all elements to `value`.
`append(other)`	Concatenate `other` (same field type) onto the end of this field.
`append_span(buffer)`	Append from a `VecX<T>` buffer (not exposed in Python — see notes below).
`swap(other)`	Swap contents with another field of the same type, including names.
`subset(mask)`	Build a `FieldHolder` containing only elements where `mask[i] == True`. The mask must be a `BoolField` of the same length.

Example:

```python
import numpy as np
import phynexis

F = phynexis.fields

# fill
sf = F.ScalarField("a", 4, 0.0)
sf.fill(7.0)
print("after fill(7):", sf.to_numpy())

# append
a = F.ScalarField("a", 3, 1.0)
b = F.ScalarField("b", 2, 9.0)
a.append(b)
print("a.append(b):", a.to_numpy())

# swap (note: name is swapped too)
a = F.ScalarField("a", 3, 1.0)
b = F.ScalarField("b", 3, 2.0)
a.swap(b)
print(f"after swap: a name={a.name} data={a.to_numpy()}")
print(f"            b name={b.name} data={b.to_numpy()}")

# subset
sf = F.ScalarField.from_numpy(np.array([10.0, 20.0, 30.0, 40.0, 50.0]))
mask = F.BoolField.from_numpy(np.array([True, False, True, False, True]))
holder = sf.subset(mask)
```

```
print("subset returns:", type(holder).__name__)
```

Output:

```text
after fill(7): [7. 7. 7. 7.]
a.append(b): [1. 1. 1. 9. 9.]
after swap: a name=b data=[2. 2. 2.]
            b name=a data=[1. 1. 1.]
subset returns: FieldHolder
```

Arithmetic (Double only)

`ScalarField` (i.e. `Field<double>`) supports element-wise arithmetic with both scalars and other `ScalarField`s of the same length. **Integer and bool fields do not implement these operators** — only `Field<double>`.

Operator	Field, Field	Field, scalar	scalar, Field
+	element-wise	broadcast	broadcast
-	element-wise	broadcast	broadcast
*	element-wise	broadcast	broadcast
/	element-wise	broadcast	—

Example:

```python
import phynexis

F = phynexis.fields

a = F.ScalarField("a", 4, 1.0)
b = F.ScalarField("b", 4, 2.0)

print("a + b:", (a + b).to_numpy())
print("a - b:", (a - b).to_numpy())
print("a * b:", (a * b).to_numpy())
print("b / a:", (b / a).to_numpy())
print("a + 5:", (a + 5).to_numpy())
print("5 + a:", (5 + a).to_numpy())
print("5 - a:", (5 - a).to_numpy())
print("3 * a:", (3 * a).to_numpy())
```

Output:

```text
a + b: [3. 3. 3. 3.]
a - b: [-1. -1. -1. -1.]
a * b: [2. 2. 2. 2.]
b / a: [2. 2. 2. 2.]
a + 5: [6. 6. 6. 6.]
5 + a: [6. 6. 6. 6.]
5 - a: [4. 4. 4. 4.]
3 * a: [3. 3. 3. 3.]
```

The mismatched-size case raises `RuntimeError("Field sizes must match")`.

Comparison Semantics

`==` and `!=` test **identity**, not value equality:

```cpp
return data_ == other.data_ && name_ == other.name_;
```

That is, two fields compare equal only if they share the same internal pointer *and* have the same name — effectively only when comparing a field with itself or an alias.

```python
import phynexis

F = phynexis.fields

a = F.ScalarField("x", 3, 1.0)
b = F.ScalarField("x", 3, 1.0)  # same name and contents, different storage
print("identical contents, separate fields a == b:", a == b)
print("self-comparison           a == a:", a == a)
```

Output:

```text
identical contents, separate fields a == b: False
self-comparison           a == a: True
```

For **value-equality**, compare `to_numpy()` arrays.

Properties

Property	Type	Access	Description
`name`	`str`	read/write	Field name (used by `FieldManager`, I/O, etc.).

Type Aliases and Integer / Bool Fields

```python
import phynexis

F = phynexis.fields

print("Field is ScalarField:    ", F.Field is F.ScalarField)
print("Idx32Field is Int32Field:", F.Idx32Field is F.Int32Field)
print("Idx64Field is Int64Field:", F.Idx64Field is F.Int64Field)

i32 = F.Int32Field("idx", 5, 7)
i32[2] = 999
print("Int32Field:", i32.to_numpy())

i64 = F.Int64Field("idx", 3, 10**12)
print("Int64Field:", i64.to_numpy())

bf = F.BoolField("active", 4, True)
bf[1] = False
print("BoolField :", bf.to_numpy())
```

Output:

```text
Field is ScalarField:     True
Idx32Field is Int32Field: True
Idx64Field is Int64Field: True
Int32Field: [  7   7 999   7   7]
Int64Field: [1000000000000 1000000000000 1000000000000]
BoolField : [ True False  True  True]
```

Known Issues

- **`field[idx]` is unchecked.** Use `field.at(idx)` whenever the index might be out of range or negative — `field[-1]` returns garbage instead of the last element.
- **`data()` returns the first element**, not a buffer pointer. Use `to_numpy()` to obtain the data as an array.
- **`np.array(field, copy=False)` segfaults.** The class declares `pybind11::buffer_protocol()` but no `def_buffer` callback. Use `to_numpy()`.
- **`==` is identity equality**, not value equality (see above).
- **`Field<T>` is not registered as a subclass of `FieldBase`** in pybind, so `isinstance(field, FieldBase)` is `False` and the `FieldBase`-only methods (`ensure_capacity`, `is_same_type`, `erase_indices`, `reorder`, `reset_value`, `swap_elements`, `copy_element`) are unreachable from Python field instances. Tracked in `binding-repair-backlog.md` (2026-05-03 entry).

Unexposed C++ API

The following C++ methods exist on `Field<T>` but are not bound:

- `value_type()` / `field_type()` — type tag queries (`ValueType`, `FieldType`).
- `is_same_type(other)` — type-tag equivalent of `isinstance` for fields.
- `erase_indices(indices)`, `reorder(new_to_old)` — bulk index operations.
- `reset_value(i)` / `reset_values()` / `set_value(i, void*)` / `get_value_ptr(i)` — generic value pokes used by `FieldHolder` internals.
- `subset_inplace(mask)` — in-place variant of `subset`.
- `save_to(path, file, opt)` / `load_from(path, file, opt)` — file persistence; the alternative free-function entry points are `write_binary` / `read_binary` (documented separately under field I/O).
- `to_binary(out)` / `from_binary(in)` / `pack(out, idx)` / `unpack(in, idx)` — binary serialisation hooks consumed by the manager.
- `Field<T>(name, comp_id, size, value)` — component-field constructor used by schema-driven layouts.
- Iteration via `begin()` / `end()` — Python-side iteration is available through `to_numpy()` or `[i]` access.

FieldBase

C++: `phynexis::fields::FieldBase` **Python**: `phynexis.fields.FieldBase` **Header**: `src/fields/core/field_base.hpp`

Description

`FieldBase` is the abstract base class for all field containers in phynexis. It defines the common interface shared by `ScalarField`, `Int32Field`, `Int64Field`, `BoolField`, and all other typed field specializations.

Important: `FieldBase` cannot be instantiated directly in Python (or C++). Use concrete subclasses such as `ScalarField` instead.

Known Issue: Due to a pybind11 binding defect, `Field<T>` is not currently registered as a subclass of `FieldBase`. As a result, `FieldBase`-only methods (`ensure_capacity`, `erase_indices`, `reorder`, `swap_elements`, `copy_element`, `reset_value`, `is_same_type`) are **unreachable from any concrete field instance**. The methods that do work (`size`, `empty`, `resize`, `reserve`, `clear`, `name`) are available because each subclass re-binds them independently.

Constructors

`FieldBase` has no exposed constructor. It is an abstract class.

Methods

`size()`

Returns the number of elements in the field.

Returns: `int` — element count

Example:

```python
import phynexis
sf = phynexis.fields.ScalarField("pressure", 100, 0.0)
print(sf.size())
```

Output:

```text
100
```

`empty()`

Returns `True` if the field contains no elements.

Returns: `bool`

Example:

```python
import phynexis
sf = phynexis.fields.ScalarField("empty_field", 0, 0.0)
print(sf.empty())
```

Output:

```text
True
```

`resize(new_size, value=None)`

Changes the number of elements. If `value` is provided, new elements are initialized to it; otherwise default construction is used.

Parameters:

Parameter	Type	Default	Description
`new_size`	`int`	—	New element count
`value`	`float` / `int` / `bool`	`None`	Value for new elements

Example:

```python
import phynexis
sf = phynexis.fields.ScalarField("x", 3, 1.0)
sf.resize(5)
print("size after resize:", sf.size())
sf.resize(5, 7.0)
print("values:", [sf[i] for i in range(5)])
```

Output:

```text
size after resize: 5
values: [1.0, 1.0, 1.0, 7.0, 7.0]
```

`reserve(capacity)`

Pre-allocates internal storage for at least `capacity` elements without changing the logical size.

Parameters:

Parameter	Type	Description
`capacity`	`int`	Minimum storage capacity

`clear()`

Removes all elements, setting size to 0.

Example:

```python
import phynexis
sf = phynexis.fields.ScalarField("x", 5, 1.0)
sf.clear()
print(sf.size(), sf.empty())
```

Output:

```text
0 True
```

`ensure_capacity(total_needed)`

Ensures the field can hold at least `total_needed` elements. Equivalent to `reserve(total_needed)` if capacity is insufficient.

Parameters:

Parameter	Type	Description
`total_needed`	`int`	Required minimum capacity

Status: ⚠️ Unreachable — see Known Issues below.

`erase_indices(idx_sorted_unique)`

Removes elements at the given indices. Indices must be sorted in ascending order and unique.

Parameters:

Parameter	Type	Description
`idx_sorted_unique`	`list[int]`	Sorted unique indices to remove

Status: ⚠️ Unreachable — see Known Issues below.

`reorder(new_to_old)`

Reorders elements in-place according to the permutation map.

Parameters:

Parameter	Type	Description
`new_to_old`	`list[int]`	For each new position `i`, `new_to_old[i]` gives the old index

Status: ⚠️ Unreachable — see Known Issues below.

`swap_elements(i, j)`

Swaps the elements at indices `i` and `j`.

Parameters:

Parameter	Type	Description
`i`	`int`	First index
`j`	`int`	Second index

Status: ⚠️ Unreachable — see Known Issues below.

`copy_element(src_index, dst_index)`

Copies the value at `src_index` to `dst_index`.

Parameters:

Parameter	Type	Description
`src_index`	`int`	Source index
`dst_index`	`int`	Destination index

Status: ⚠️ Unreachable — see Known Issues below.

`reset_value(index)`

Resets the element at `index` to its type's default value (`0.0`, `0`, or `False`).

Parameters:

Parameter	Type	Description
`index`	`int`	Index to reset

Status: ⚠️ Unreachable — see Known Issues below.

`is_same_type(other)`

Returns `True` if `other` has the same `ValueType` and `FieldType`.

Parameters:

Parameter	Type	Description
`other`	`FieldBase`	Another field to compare with

Returns: `bool`

Status: ⚠️ Unreachable — see Known Issues below.

Properties

Property	Type	Access	Description
`name`	`str`	read/write	Field name identifier

Example:

```python
import phynexis
sf = phynexis.fields.ScalarField("old_name", 10, 0.0)
print(sf.name)
sf.name = "new_name"
print(sf.name)
```

Output:

```text
old_name
new_name
```

Unexposed C++ API

The following C++ APIs are not exposed via pybind:

- `append(other)` — append another field of the same type
- `reset_values()` — reset all elements to default
- `set_value(index, void* value)` — untyped value setter (not Python-friendly)
- `get_value_ptr(index)` — returns untyped pointer (not Python-friendly)
- `value_type()` — returns `ValueType` enum
- `field_type()` — returns `FieldType` enum
- `subset(mask)` — extract elements where `mask[i]` is `True`; returns `FieldHolder`
- `subset_inplace(mask)` — in-place subset
- `to_binary(out)` / `from_binary(in)` — binary serialization
- `pack(out, indices)` / `unpack(in, target_indices)` — selective binary I/O

Known Issues

`Field<T>` not registered as subclass of `FieldBase`

`pyfield.cpp` uses `pybind11::class_<Field<T>>(m, name)` without declaring `FieldBase` as the base class. As a result:

- `isinstance(scalar_field, FieldBase)` returns `False`
- `FieldBase`-only methods bound in `pyfield_base.cpp` are invisible on concrete field instances
- The methods that **do** work on subclasses (`size`, `empty`, `resize`, `reserve`, `clear`, `name`) are available because `pyfield.cpp` re-binds them independently

Workaround: None. Requires binding fix in `phynexis/bindings/fields/core/pyfield.cpp`:

```cpp
// Current (broken)
pybind11::class_<Field<T>>(m, name.c_str(), pybind11::buffer_protocol())

// Fixed
pybind11::class_<Field<T>, FieldBase>(m, name.c_str(), pybind11::buffer_protocol())
```

Once fixed, the duplicated bindings of `size`, `empty`, `resize`, `reserve`, `clear`, `name` on `Field<T>` can be removed (inherited from `FieldBase`).

File: `phynexis/bindings/fields/core/pyfield.cpp` **Status**: open — tracked in `binding-repair-backlog.md`

FieldHolder

C++: `phynexis::fields::FieldHolder` **Python**: `phynexis.fields.FieldHolder` **Header**: `src/fields/core/field_holder.hpp`

Owning move-only smart pointer around a heap-allocated `FieldBase`. C++ code uses `FieldHolder` to return polymorphic field instances by value while keeping ownership clear (it deletes the inner pointer on destruction). The most common way to obtain one from Python is via `Field<T>.subset(mask)`.

`FieldHolder` is non-copyable; the move constructor is not exposed to Python, so treat each holder as single-use.

Constructors

Signature	Description
`FieldHolder()`	Empty holder; `get()` is `None` and `bool(h)` is `False`.
`FieldHolder(ptr)`	Adopts an existing `FieldBase*`. Pass `None` to construct an empty holder.

Practical sources of holders:

- `field.subset(mask)` — returns a new holder containing the masked subset.

Example:

```python
import numpy as np
import phynexis

F = phynexis.fields

empty = F.FieldHolder()
print("empty bool:", bool(empty), "get():", empty.get())

sf = F.ScalarField.from_numpy(np.array([10.0, 20.0, 30.0, 40.0, 50.0]))
mask = F.BoolField.from_numpy(np.array([True, False, True, False, True]))
holder = sf.subset(mask)
print("holder bool:", bool(holder))
print("inner field:", holder.get().to_numpy())
```

Output:

```text
empty bool: False get(): None
holder bool: True
inner field: [10. 30. 50.]
```

Methods

`get()`

Returns the contained field without transferring ownership. The Python object returned reflects the **runtime** type of the field (e.g. `ScalarField`), so you can call its full interface directly. Returns `None` if the holder is empty.

Returns: `ScalarField` / `Int32Field` / `Int64Field` / `BoolField` / `None`.

`release()`

Transfers ownership of the inner field out of the holder. After release the holder is empty (`bool(holder)` is `False`) and the returned object is owned by the Python interpreter — it remains alive even if the holder is garbage collected.

Returns: the inner field (typed concretely, e.g. `ScalarField`) or `None` if the holder was already empty.

`reset(ptr=None)`

Replaces the inner pointer. The previous content (if any) is deleted. Calling `reset()` with no argument or `reset(None)` empties the holder.

Parameters:

Parameter	Type	Default	Description
`ptr`	`FieldBase` or `None`	`None`	New pointer to adopt.

`as_field_double()`

Down-cast the inner field to `Field<double>` (alias `ScalarField`). Raises `RuntimeError("std::bad_cast")` if the inner field is not a double field, or if the holder is empty.

Returns: `ScalarField`.

`as_field_int()`

Down-cast the inner field to `Field<Int32>` (alias `Int32Field`). Raises `RuntimeError("std::bad_cast")` on type mismatch or empty holder.

Returns: `Int32Field`.

`__bool__()`

`True` when the holder owns a non-null pointer.

Examples

Lifetime semantics

`get()` returns a borrowed reference — the holder still owns the data, so the returned field becomes invalid after the holder is collected. `release()` transfers ownership to the returned object and is safe to keep beyond the holder's lifetime.

```python
import numpy as np
import phynexis

F = phynexis.fields

sf = F.ScalarField.from_numpy(np.array([10.0, 20.0, 30.0]))
mask = F.BoolField.from_numpy(np.array([True, False, True]))
h = sf.subset(mask)

inner = h.release()
print("released:", inner.to_numpy())

del h  # safe — ownership already transferred
print("still alive after del h:", inner.to_numpy())

# Re-releasing an empty holder yields None
print("second release:", h.release())  # noqa: F821
```

(Re-using `h` here would actually fail because `del h` removes the binding — the comment is illustrative.)

Down-cast and type mismatch

```python
import numpy as np
import phynexis

F = phynexis.fields

double_holder = F.ScalarField.from_numpy(
    np.array([1.0, 2.0, 3.0])
).subset(F.BoolField.from_numpy(np.array([True, True, True])))

int_holder = F.Int32Field.from_numpy(
    np.array([10, 20, 30], dtype=np.int32)
```

```
).subset(F.BoolField.from_numpy(np.array([True, False, True])))

print("as double:", double_holder.as_field_double().to_numpy())
print("as int:", int_holder.as_field_int().to_numpy())

# Wrong cast raises bad_cast
try:
    double_holder.as_field_int()
except RuntimeError as e:
    print("wrong cast:", repr(e))
```

Output:

```
as double: [1. 2. 3.]
as int: [10 30]
wrong cast: RuntimeError('std::bad_cast')
```

Reset

```python
import numpy as np
import phynexis

F = phynexis.fields

sf = F.ScalarField.from_numpy(np.array([1.0, 2.0]))
mask = F.BoolField.from_numpy(np.array([True, True]))
h = sf.subset(mask)
print("before reset:", bool(h))

h.reset()       # equivalent to reset(None)
print("after reset():", bool(h))
```

Output:

```
before reset: True
after reset(): False
```

Known Issues

- The casts only cover `double` and `Int32`. There is no `as_field_int64()` or `as_field_bool()`, so `Int64Field` / `BoolField` subsets are reachable only through `get()`.
- `subset()` does not propagate the source field's `name`; the resulting field has `name == ""`.
- Because `Field<T>` is not registered as a subclass of `FieldBase` (see `binding-repair-backlog.md`), passing a generic `FieldBase`-typed reference back to Python loses no information in practice — pybind already resolves to the concrete subclass via runtime type info — but `isinstance(holder.get(), FieldBase)` is still `False`.

Unexposed C++ API

- `template <typename T> T &as()` — generic templated cast; only the `Double` and `Int32` instantiations are exposed via the named `as_field_double` / `as_field_int` lambdas.
- Move constructor / move assignment — not bound (pybind does not need them for ownership transfer; `release()` is the Python-visible equivalent).

FieldLayout

C++: `phynexis::fields::FieldLayout` (alias for `VecX<FieldMeta>`) **Python**: `phynexis.fields.FieldLayout`
Header: `src/fields/core/field_layout.hpp`

Description

`FieldLayout` is a sequential container of `FieldMeta` objects. It describes the structure of a group of fields — for example, which fields exist in a simulation domain, their types, and their value types. Each element is a `FieldMeta` instance.

In C++, `FieldLayout` is a type alias for `VecX<FieldMeta>`. In Python it is exposed as a standalone class with list-like semantics.

Constructors

`FieldLayout()`

Creates an empty layout.

Example:

```python
import phynexis
fl = phynexis.fields.FieldLayout()
print(len(fl), fl.empty())
```

Output:

```text
0 True
```

`FieldLayout(metas)`

Creates a layout from a list of `FieldMeta` objects.

Parameters:

Parameter	Type	Description
`metas`	`list[FieldMeta]`	Initial field metadata entries

Example:

```python
import phynexis
F = phynexis.fields

meta1 = F.FieldMeta("pressure", F.FieldType.Scalar, F.ValueType.Double)
meta2 = F.FieldMeta("velocity", F.FieldType.VecX, F.ValueType.Double)
fl = F.FieldLayout([meta1, meta2])
print(len(fl))
```

Output:

```text
2
```

Methods

`__len__()`

Returns the number of `FieldMeta` entries.

Returns: `int`

`__getitem__(index)`

Returns the `FieldMeta` at `index`.

Parameters:

Parameter	Type	Description
`index`	`int`	Zero-based index

Returns: `FieldMeta`

Note: Negative indices are **not supported** and raise `IndexError`.

Example:

```python
import phynexis
F = phynexis.fields

meta = F.FieldMeta("temperature", F.FieldType.Scalar, F.ValueType.Double)
fl = F.FieldLayout([meta])
print(fl[0])
```

Output:

```text
<FieldMeta object at 0x...>
```

`size()`

Same as `len(layout)`. Returns the number of entries.

Returns: `int`

`empty()`

Returns `True` if the layout contains no entries.

Returns: `bool`

`append(meta)`

Appends a `FieldMeta` to the end of the layout.

Parameters:

Parameter	Type	Description
`meta`	`FieldMeta`	Metadata to append

Example:

```python
import phynexis
F = phynexis.fields

fl = F.FieldLayout()
fl.append(F.FieldMeta("x", F.FieldType.Scalar, F.ValueType.Double))
fl.append(F.FieldMeta("y", F.FieldType.Scalar, F.ValueType.Double))
print(fl.size())
```

Output:

```text
2
```

Known Issues

Negative indexing not supported

`layout[-1]` raises `IndexError` instead of returning the last element. The binding performs a direct bounds check without normalising negative indices.

Workaround: Use `layout[len(layout) - 1]` or iterate over the layout.

File: `phynexis/bindings/fields/core/pyfield_layout.cpp` **Status**: noted

FieldManager

C++: `phynexis::fields::FieldManager` **Python**: `phynexis.fields.FieldManager` **Header**: `src/fields/core/manager.hpp`

Description

`FieldManager` provides centralized lifecycle management for a collection of fields. It maintains a `KeyedObjectPool` of `FieldBase*` pointers, tracks a `FieldLayout`, and ensures all managed fields share the same size. Use it to create, resize, subset, and persist groups of fields together.

Constructors

`FieldManager()`

Creates an empty manager with no fields and size 0.

Example:

```python
import phynexis
fm = phynexis.fields.FieldManager()
print(fm.field_size(), fm.get_field_count())
```

Output:

```text
0 0
```

Methods

`create_field(meta)`

Creates a single field from metadata and registers it.

Parameters:

Parameter	Type	Description
`meta`	`FieldMeta`	Metadata for the new field

Returns: `FieldHandle` — handle to the created field

Example:

```python
import phynexis
F = phynexis.fields

fm = F.FieldManager()
meta = F.FieldMeta("pressure", F.FieldType.Scalar, F.ValueType.Double)
h = fm.create_field(meta)
print(h)
fm.resize_fields(10)
print(fm.field_size())
```

Output:

```text
FieldHandle(index=0, epoch=0)
10
```

`create_fields(metas)`

Creates multiple fields from a `FieldLayout` (or list of `FieldMeta`).

Parameters:

Parameter	Type	Description
`metas`	`FieldLayout` or `list[FieldMeta]`	Layout describing the fields to create

Returns: `list[FieldHandle]`

Example:

```python
import phynexis
F = phynexis.fields

fm = F.FieldManager()
layout = F.FieldLayout([
    F.FieldMeta("x", F.FieldType.Scalar, F.ValueType.Double),
    F.FieldMeta("v", F.FieldType.VecX, F.ValueType.Double),
])
handles = fm.create_fields(layout)
print([str(h) for h in handles])
```

Output:

```text
['FieldHandle(index=0, epoch=0)', 'FieldHandle(index=1, epoch=0)']
```

`ensure_fields(layout)` / `ensure_fields(metas)` / `ensure_fields(schema)`

Ensures all fields described by the layout/schema exist; creates any that are missing. Existing fields are left untouched.

Parameters:

Parameter	Type	Description
`layout`	`FieldLayout`	Layout to ensure
`metas`	`list[FieldMeta]`	List of metadata (auto-converted to layout)
`schema`	`FieldSchema`	Schema to ensure

Returns: `bool` — `True` if any new field was created

Example:

```python
import phynexis
F = phynexis.fields

fm = F.FieldManager()
fm.create_field(F.FieldMeta("x", F.FieldType.Scalar, F.ValueType.Double))
fm.resize_fields(5)

# "x" already exists, "y" will be created
fm.ensure_fields([
    F.FieldMeta("x", F.FieldType.Scalar, F.ValueType.Double),
    F.FieldMeta("y", F.FieldType.Scalar, F.ValueType.Double),
])
print(fm.get_field_count())
```

Output:

```text
2
```

`remove_field(name)` / `remove_field(handle)`

Removes a field by name or by handle.

Parameters:

Parameter	Type	Description
`name`	`str`	Field name
`handle`	`FieldHandle`	Field handle

Example:

```python
import phynexis
F = phynexis.fields

fm = F.FieldManager()
fm.create_field(F.FieldMeta("x", F.FieldType.Scalar, F.ValueType.Double))
fm.remove_field("x")
print(fm.has_field("x"))
```

Output:

```text
False
```

`has_field(name)`

Returns `True` if a field with the given name exists.

Parameters:

Parameter	Type	Description
`name`	`str`	Field name to check

Returns: `bool`

`valid_field_handle(handle)`

Checks if a handle is still valid (index in range and epoch matches).

Parameters:

Parameter	Type	Description
`handle`	`FieldHandle`	Handle to validate

Returns: `bool`

`get_field(handle)`

Retrieves the concrete field instance for a handle. The runtime type is resolved via RTTI (e.g. `ScalarField`, `VecXdField`).

Parameters:

Parameter	Type	Description
`handle`	`FieldHandle`	Field handle

Returns: concrete `Field` subclass (e.g. `ScalarField`)

Example:

```python
import phynexis
F = phynexis.fields

fm = F.FieldManager()
h = fm.create_field(F.FieldMeta("p", F.FieldType.Scalar, F.ValueType.Double))
fm.resize_fields(5)
f = fm.get_field(h)
print(type(f).__name__)
```

Output:

```text
ScalarField
```

`find_field(name)`

Finds a field by name and returns its handle.

Parameters:

Parameter	Type	Description
`name`	`str`	Field name

Returns: `FieldHandle`

`field_size()`

Returns the shared element count of all managed fields.

Returns: `int`

`resize_fields(new_size)`

Resizes **all** managed fields to `new_size`.

Parameters:

Parameter	Type	Description
`new_size`	`int`	New element count for all fields

`reserve_fields(capacity)`

Reserves capacity for all managed fields without changing their sizes.

Parameters:

Parameter	Type	Description
`capacity`	`int`	Capacity to reserve

`clear_fields()`

Clears all elements from all fields (size becomes 0). Fields themselves remain registered.

`clear()`

Removes all fields and resets the manager to its initial empty state.

`subset_fields(mask)`

Extracts a subset of elements where `mask[i]` is `True`, returning a new `FieldManager`.

Parameters:

Parameter	Type	Description
`mask`	`BoolField`	Boolean mask field

Returns: `FieldManager` — new manager containing the subset

Known Issue: The returned manager may have `field_size() == 0` even when the subset is non-empty. This appears to be a C++ source behavior; the field count and names are correct.

Example:

```python
import phynexis
F = phynexis.fields

fm = F.FieldManager()
fm.create_field(F.FieldMeta("x", F.FieldType.Scalar, F.ValueType.Double))
fm.resize_fields(5)

mask = F.BoolField("mask", 5, True)
mask[0] = False
mask[2] = False

fm2 = fm.subset_fields(mask)
print(fm2.get_field_count(), fm2.get_field_names())
```

Output:

```text
1 ['x']
```

`subset_fields_inplace(mask)`

Same as `subset_fields` but modifies the current manager in-place.

Parameters:

Parameter	Type	Description
`mask`	`BoolField`	Boolean mask field

`check_size_consistency()`

Returns `True` if all fields have the same size.

Returns: `bool`

`get_inconsistent_fields()`

Returns the names of fields whose size differs from the majority.

Returns: `list[str]`

`reset_field_values(index)`

Resets all field values at `index` to their type defaults.

Parameters:

Parameter	Type	Description
`index`	`int`	Element index to reset

`copy_field_values(src_index, dst_index)`

Copies all field values from `src_index` to `dst_index`.

Parameters:

Parameter	Type	Description
`src_index`	`int`	Source index
`dst_index`	`int`	Destination index

`set_field_value(index, name, value)`

Sets a single field value by name at the given index.

Parameters:

Parameter	Type	Description
`index`	`int`	Element index
`name`	`str`	Field name
`value`	`float` / `int` / `bool`	Value to set

Example:

```python
import phynexis
F = phynexis.fields

fm = F.FieldManager()
fm.create_field(F.FieldMeta("p", F.FieldType.Scalar, F.ValueType.Double))
fm.resize_fields(3)
fm.set_field_value(1, "p", 99.5)
ok, val = fm.get_field_value(1, "p")
print(val)
```

Output:

```text
{'p': 99.5}
```

`get_field_value(index, name)`

Reads a single field value by name at the given index.

Parameters:

Parameter	Type	Description
`index`	`int`	Element index
`name`	`str`	Field name

Returns: `tuple[bool, dict]` — `(success, {name: value})`

`set_field_values(index, values)`

Sets multiple field values at an index from a JSON-like dict.

Parameters:

Parameter	Type	Description
`index`	`int`	Element index
`values`	`dict`	`{field_name: value, ...}`

`get_field_values(index)`

Reads all field values at an index as a dict.

Parameters:

Parameter	Type	Description
`index`	`int`	Element index

Returns: `dict` — `{field_name: value, ...}`

Example:

```python
import phynexis
F = phynexis.fields

fm = F.FieldManager()
fm.create_field(F.FieldMeta("p", F.FieldType.Scalar, F.ValueType.Double))
fm.create_field(F.FieldMeta("t", F.FieldType.Scalar, F.ValueType.Double))
fm.resize_fields(2)
fm.set_field_value(0, "p", 1.0)
fm.set_field_value(0, "t", 300.0)
print(fm.get_field_values(0))
```

Output:

```text
{'p': 1.0, 't': 300.0}
```

`save_to(path, file, opt=None)`

Saves all fields to a binary file.

Parameters:

Parameter	Type	Default	Description
`path`	`str`	—	Parent directory
`file`	`str`	—	Filename
`opt`	`SaveOptions`	`SaveOptions()`	Save options

`load_from(path, file, opt=None)`

Loads fields from a binary file.

Parameters:

Parameter	Type	Default	Description
`path`	`str`	—	Parent directory
`file`	`str`	—	Filename
`opt`	`LoadOptions`	`LoadOptions()`	Load options

Example:

```python
import phynexis, tempfile, os
F = phynexis.fields
```

```
3
4  fm = F.FieldManager()
5  fm.create_field(F.FieldMeta("p", F.FieldType.Scalar, F.ValueType.Double))
6  fm.resize_fields(5)
7
8  with tempfile.TemporaryDirectory() as td:
9      fm.save_to(td, "snapshot.dat")
10     fm2 = F.FieldManager()
11     fm2.load_from(td, "snapshot.dat")
12     print(fm2.get_field_count(), fm2.field_size())
```

Output:

```
1 1 5
```
text

`inspect(filepath)`

Reads metadata from a snapshot file without loading data.

Parameters:

Parameter	Type	Description
`filepath`	`str`	Full path to the snapshot file

`get_field_layout()`

Returns the current `FieldLayout` describing all registered fields.

Returns: `FieldLayout`

`get_field_count()`

Returns the number of registered fields.

Returns: `int`

`get_field_names()`

Returns the names of all registered fields.

Returns: `list[str]`

`get_field_ptrs()`

Returns a list of concrete field instances. Each element is the runtime-resolved type (e.g. `ScalarField`, `VecXdField`).

Returns: `list[Field]`

`print_info()`

Prints a summary of the manager to stdout.

Example:

```python
import phynexis
F = phynexis.fields

fm = F.FieldManager()
fm.create_field(F.FieldMeta("p", F.FieldType.Scalar, F.ValueType.Double))
fm.resize_fields(5)
fm.print_info()
```

Output:

```text
=== FieldManager ===
field count: 1
field size: 5
 - p (double)
```

Unexposed C++ API

The following C++ APIs are not exposed via pybind:

- `remove_fields(handles)` / `remove_fields(names)` — batch remove by `VecX<FieldHandle>` or `VecX<String>`

Known Issues

`subset_fields` returns manager with `field_size() == 0`

When calling `subset_fields(mask)`, the returned `FieldManager` correctly contains the subset fields (names and count are accurate), but `field_size()` reports 0. This appears to be a C++ source-side behavior rather than a binding issue.

Workaround: Use `subset_fields_inplace(mask)` if in-place mutation is acceptable, or verify field sizes with `get_field_count()` and per-field `size()`.

Status: noted — source issue, not binding bug

FieldMeta

C++: `phynexis::fields::FieldMeta` **Python**: `phynexis.fields.FieldMeta` **Header**: `src/fields/core/field_meta.hpp`

Schema-level metadata describing a field's name, value type, and field type (scalar vs variable-length vector). Used by `FieldSchema` and `FieldManager` for runtime field registration and I/O.

Enums

`FieldType`

Discriminates scalar fields from variable-length vector fields.

Value	Description
`Scalar`	One value per grid index
`VecX`	Variable-length vector per grid index

`ValueType`

Type tag for scalar values. Maps to phynexis primitive types.

Value	C++ Type	Size
`Bool`	`bool`	1 byte
`Char`	`char`	1 byte
`Float`	`float`	4 bytes
`Double`	`double`	8 bytes
`Int8`	`int8_t`	1 byte
`Int16`	`int16_t`	2 bytes
`Int32`	`int32_t`	4 bytes
`Int64`	`int64_t`	8 bytes

Constructors

`FieldMeta()`

Default constructor. Creates a scalar double field named `"default"`.

`FieldMeta(name, field_type, value_type)`

Create metadata with explicit parameters.

Parameters:

Parameter	Type	Description
`name`	`str`	Field name
`field_type`	`FieldType`	`FieldType.Scalar` or `FieldType.VecX`
`value_type`	`ValueType`	One of the `ValueType` enum values

Properties

Property	Type	Access	Description
`name`	`str`	read/write	Field name
`field_type`	`FieldType`	read/write	Scalar or VecX
`value_type`	`ValueType`	read/write	Primitive value type

Example

```python
import phynexis

# Default construction
meta = phynexis.fields.FieldMeta()
print("default:", meta.name, meta.field_type, meta.value_type)

# Explicit construction
meta2 = phynexis.fields.FieldMeta(
    "velocity",
    phynexis.fields.FieldType.VecX,
    phynexis.fields.ValueType.Double,
)
print("explicit:", meta2.name, meta2.field_type, meta2.value_type)

```

```python
# Modify properties
meta2.name = "pressure"
meta2.field_type = phynexis.fields.FieldType.Scalar
meta2.value_type = phynexis.fields.ValueType.Float
print("modified:", meta2.name, meta2.field_type, meta2.value_type)

# Enum values
print("FieldType:", list(phynexis.fields.FieldType.__members__.keys()))
print("ValueType:", list(phynexis.fields.ValueType.__members__.keys()))
```

Output:

```text
default: default FieldType.Scalar ValueType.Double
explicit: velocity FieldType.VecX ValueType.Double
modified: pressure FieldType.Scalar ValueType.Float
FieldType: ['Scalar', 'VecX']
ValueType: ['Bool', 'Char', 'Float', 'Double', 'Int8', 'Int16', 'Int32', 'Int64']
```

Unexposed C++ API

- `to_string(FieldType)` / `from_string(String)` — string conversion helpers
- `to_json(json&, FieldType)` / `from_json(json&, FieldType&)` — JSON serialization
- `to_json(json&, const FieldMeta&)` / `from_json(json&, FieldMeta&)` — JSON serialization

FieldSchema

C++: `phynexis::fields::FieldSchema` **Python**: `phynexis.fields.FieldSchema` **Header**: `src/fields/core/field_schema.hpp`

Description

`FieldSchema` defines a structured layout of fields. It is a collection of `FieldSlot` objects that describe the names, types, and dimensions of all fields in a data model. A schema can be converted to a `FieldLayout` (used by `FieldManager` to create fields) or used to resolve/normalize JSON dictionaries.

Constructors

`FieldSchema()`

Creates an empty schema with default name `"default"`.

Example:

```python
import phynexis
schema = phynexis.fields.FieldSchema()
print(schema.name(), schema.num_slots())
```

Output:

```text
default 0
```

Methods

`append(slot)` / `append(schema)`

Appends a `FieldSlot` or merges another `FieldSchema`. Duplicate slot names are silently skipped.

Parameters:

Parameter	Type	Description
`slot`	`FieldSlot`	Slot to append
`schema`	`FieldSchema`	Schema whose slots are appended

Example:

```python
import phynexis
F = phynexis.fields

schema = F.FieldSchema()
schema.append(F.FieldSlot.make_scalar("pressure"))
schema.append(F.FieldSlot.make_vec3("velocity"))
print(schema.num_slots())

# Merge another schema
schema2 = F.FieldSchema()
schema2.append(F.FieldSlot.make_scalar("temperature"))
schema.append(schema2)
print(schema.num_slots())
```

Output:

```text
2
3
```

`name()`

Returns the schema name.

Returns: `str`

`num_slots()`

Returns the number of slots.

Returns: `int`

`slots()`

Returns all slots as a list.

Returns: `list[FieldSlot]`

Example:

```python
import phynexis
F = phynexis.fields

schema = F.FieldSchema()
schema.append(F.FieldSlot.make_scalar("p"))
schema.append(F.FieldSlot.make_vec3("v"))
for s in schema.slots():
    print(s.name, s.dimension)
```

Output:

```text
p 1
v 3
```

`slot(index)`

Returns the slot at the given index.

Parameters:

Parameter	Type	Description
`index`	`int`	Zero-based slot index

Returns: `FieldSlot`

`make_layout()`

Converts the schema to a `FieldLayout`. Vector slots (e.g. `vec3`) are expanded into individual scalar component entries.

Returns: `FieldLayout`

Example:

```python
import phynexis
F = phynexis.fields

schema = F.FieldSchema()
schema.append(F.FieldSlot.make_scalar("pressure"))
schema.append(F.FieldSlot.make_vec3("velocity"))
schema.append(F.FieldSlot.make_vecx("features"))

layout = schema.make_layout()
print(len(layout))
for meta in layout:
    print(meta.name, meta.field_type, meta.value_type)
```

Output:

```text
5
pressure FieldType.Scalar ValueType.Double
velocity.x FieldType.Scalar ValueType.Double
velocity.y FieldType.Scalar ValueType.Double
velocity.z FieldType.Scalar ValueType.Double
features FieldType.VecX ValueType.Double
```

`resolve_json(j)`

Normalizes a JSON dictionary according to the schema.

- Keys that match schema slots are kept.
- Vector values (lists) are expanded into component keys (`velocity` → `velocity.x`, `velocity.y`, `velocity.z`).
- Extra keys not in the schema are dropped.
- `null` values are skipped.

Parameters:

Parameter	Type	Description
`j`	`dict`	Input dictionary

Returns: `dict` — normalized dictionary

Example:

```python
import phynexis
F = phynexis.fields

schema = F.FieldSchema()
schema.append(F.FieldSlot.make_scalar("pressure"))
schema.append(F.FieldSlot.make_vec3("velocity"))

j = {
    "pressure": 1.0,
    "velocity": [2.0, 3.0, 4.0],
    "extra": 999,  # will be dropped
}
result = schema.resolve_json(j)
print(result)
```

Output:

```text
{'pressure': 1.0, 'velocity.x': 2.0, 'velocity.y': 3.0, 'velocity.z': 4.0}
```

`print_info()`

Prints a human-readable summary of the schema to stdout.

Example:

```python
import phynexis
F = phynexis.fields

schema = F.FieldSchema()
schema.append(F.FieldSlot.make_scalar("pressure"))
schema.append(F.FieldSlot.make_vec3("velocity"))
schema.print_info()
```

Output:

```text
FieldSchema: default
  Slots: 2
    pressure (scalar, double, 1)
    velocity (scalar, double, 3)
```

Unexposed C++ API

The following C++ APIs are not exposed via pybind:

- Schema construction helpers (`add_scalar`, `add_vec3`, `add_vec4`, `add_vec6`, `add_vecn`, `add_vecx`) — equivalent to `append(FieldSlot.make_*(...))` in Python.
- `to_json` / `from_json` free functions — JSON serialization of the schema itself.

Known Issues

`slots()` returns a snapshot

`schema.slots()` returns a Python `list` of `FieldSlot` objects. If you call `append()` after `slots()`, the returned list is not automatically updated. Call `slots()` again to get the latest state.

FieldSlot

C++: `phynexis::fields::FieldSlot` **Python**: `phynexis.fields.FieldSlot` **Header**: `src/fields/core/field_slot.hpp`

Description

`FieldSlot` describes a single field entry within a `FieldSchema`. It stores the field's name, type, dimension, and component naming rules. Think of it as a blueprint for one column in a structured data layout.

`FieldSlot` is a plain data struct exposed to Python with read/write attributes.

Constructors

`FieldSlot()`

Creates a default slot with name `"default"`, `FieldType.Scalar`, `ValueType.Double`, and dimension 1.

Example:

```python
import phynexis
slot = phynexis.fields.FieldSlot()
print(slot.name, slot.field_type, slot.value_type, slot.dimension)
```

Output:

```text
default FieldType.Scalar ValueType.Double 1
```

Factory Methods

Factory methods are the recommended way to create `FieldSlot` instances. They set sensible defaults and generate appropriate component names.

`FieldSlot.make_scalar(name, type=ValueType.Double)`

Creates a scalar slot.

Parameters:

Parameter	Type	Default	Description
`name`	`str`	—	Field name
`type`	`ValueType`	`ValueType.Double`	Value type

Returns: `FieldSlot`

Example:

```python
import phynexis
F = phynexis.fields

s = F.FieldSlot.make_scalar("pressure", F.ValueType.Double)
print(s.name, s.dimension, s.get_field_names())
```

Output:

```text
pressure 1 ['pressure']
```

`FieldSlot.make_vec3(name, type=ValueType.Double)`

Creates a 3-component vector slot. Component names default to `['x', 'y', 'z']`.

Parameters:

Parameter	Type	Default	Description
`name`	`str`	—	Base name (prefix)
`type`	`ValueType`	`ValueType.Double`	Value type

Returns: `FieldSlot`

Example:

```python
import phynexis
F = phynexis.fields

s = F.FieldSlot.make_vec3("velocity")
print(s.get_field_names())
print(s.get_component_names())
```

Output:

```text
['velocity.x', 'velocity.y', 'velocity.z']
['x', 'y', 'z']
```

`FieldSlot.make_vec4(name, type=ValueType.Double)`

Creates a 4-component vector slot. Component names default to `['x', 'y', 'z', 'w']`.

Parameters:

Parameter	Type	Default	Description
`name`	`str`	—	Base name
`type`	`ValueType`	`ValueType.Double`	Value type

Returns: `FieldSlot`

`FieldSlot.make_vec6(name, type=ValueType.Double)`

Creates a 6-component vector slot. Component names default to `['xx', 'yy', 'zz', 'xy', 'xz', 'yz']`.

Parameters:

Parameter	Type	Default	Description
`name`	`str`	—	Base name
`type`	`ValueType`	`ValueType.Double`	Value type

Returns: `FieldSlot`

`FieldSlot.make_vecn(name, dimension, type=ValueType.Double, delimiter=".", component_names=[])`

Creates an N-component vector slot with arbitrary dimension.

Parameters:

Parameter	Type	Default	Description
`name`	`str`	—	Base name
`dimension`	`int`	—	Number of components
`type`	`ValueType`	`ValueType.Double`	Value type
`delimiter`	`str`	`"."`	Separator between base name and component
`component_names`	`list[str]`	`[]`	Custom component names (auto-generated if empty)

Returns: `FieldSlot`

Example:

```python
import phynexis
F = phynexis.fields

s = F.FieldSlot.make_vecn("pos", 5, delimiter="_")
print(s.get_field_names())
```

Output:

```text
['pos_c0', 'pos_c1', 'pos_c2', 'pos_c3', 'pos_c4']
```

`FieldSlot.make_vecx(name, type=ValueType.Double)`

Creates a variable-length vector (`VecX`) slot.

Parameters:

Parameter	Type	Default	Description
`name`	`str`	—	Field name
`type`	`ValueType`	`ValueType.Double`	Value type

Returns: `FieldSlot`

Example:

```python
import phynexis
F = phynexis.fields

s = F.FieldSlot.make_vecx("features")
print(s.name, s.field_type, s.dimension)
```

Output:

```text
features FieldType.VecX 1
```

Properties

Property	Type	Access	Description
`name`	`str`	read/write	Field name or base name
`field_type`	`FieldType`	read-only	`Scalar` or `VecX`
`value_type`	`ValueType`	read-only	`Double`, `Int32`, `Bool`, etc.
`dimension`	`int`	read-only	Number of components (1 for scalar, 3 for vec3, etc.)
`delimiter`	`str`	read/write	Separator used in expanded field names
`component_names`	`list[str]`	read/write	Component names (e.g. `['x', 'y', 'z']`)

Methods

`get_field_names()`

Returns the fully expanded field names for this slot.

- **Scalar**: returns `[name]`
- **Vec3**: returns `[name.x, name.y, name.z]`
- **VecN**: returns `[name.c0, name.c1, ...]` or custom component names

Returns: `list[str]`

Example:

```python
import phynexis
F = phynexis.fields

s1 = F.FieldSlot.make_scalar("p")
s2 = F.FieldSlot.make_vec3("v")
s3 = F.FieldSlot.make_vecn("c", 2, component_names=["r", "g"])

print(s1.get_field_names())
print(s2.get_field_names())
print(s3.get_field_names())
```

Output:

```text
['p']
['v.x', 'v.y', 'v.z']
['c.r', 'c.g']
```

`get_component_names()`

Returns the component names for this slot.

Returns: `list[str]`

Example:

```python
import phynexis
F = phynexis.fields

s = F.FieldSlot.make_vec6("stress")
print(s.get_component_names())
```

Output:

```text
['xx', 'yy', 'zz', 'xy', 'xz', 'yz']
```

Field Schemas

C++: `phynexis::fields::{Scalar,Vec3,Vec4,Vec6,VecN,VecX}FieldSchema` **Python**: `phynexis.fields.{ScalarFieldSchema,V`
Header: `src/fields/schema/*.hpp`

Typed field schema classes that extend FieldSchema. Each defines a single-slot field with a specific component layout (scalar, fixed-vector, variable-length, or runtime-sized).

All constructors accept a `base_name` (the slot name prefix) and an optional `ValueType` (default `Double`).

Schema Types

Class	Components	Slot Name Pattern	Extra Parameters
`ScalarFieldSchema`	1	`{base_name}`	—
`Vec3FieldSchema`	3	`{base_name}_x`, `{base_name}_y`, `{base_name}_z`	—
`Vec4FieldSchema`	4	`{base_name}_w`, `{base_name}_x`, …	—
`Vec6FieldSchema`	6	`{base_name}_xx`, `{base_name}_yy`, …	—
`VecNFieldSchema`	N (fixed)	`{base_name}_0` … `{base_name}_{N-1}`	`dimension: int` (default 1)
`VecXFieldSchema`	runtime	`{base_name}` (single slot)	—

Usage

Create individual schemas and compose them into a multi-slot `FieldSchema`:

Example:

```python
import phynexis
F = phynexis.fields

# Create typed field schemas
density_schema = F.ScalarFieldSchema("density")
velocity_schema = F.Vec3FieldSchema("velocity")
stress_schema = F.Vec6FieldSchema("stress")
data_schema = F.VecNFieldSchema("data", F.ValueType.Double, 7)
misc_schema = F.VecXFieldSchema("misc")

# Compose into a multi-slot schema
model = F.FieldSchema()
model.append(density_schema)
model.append(velocity_schema)
model.append(stress_schema)
model.append(data_schema)
model.append(misc_schema)

print("Model name:", model.name())
print("Slots:", model.num_slots())
```

Output:

```text
Model name: default
Slots: 5
```

The composed schema can be used to create `FieldMeta` entries (via `make_layout()`) for building fields with `FieldManager`.

CSRMatrix

C++: `phynexis::fields::CSRMatrix<ValueType, IndexType>` **Python**: `phynexis.fields.CSRMatrix` (alias `CSRMatrixd`), `phynexis.fields.CSRMatrixi` **Header**: `src/fields/containers/csr_matrix.hpp`

Description

`CSRMatrix` is a compressed sparse row (CSR) matrix for linear algebra operations. Each row stores `(column, value)` pairs sorted by column index. The class supports dynamic element insertion, row-wise queries, matrix–matrix and matrix–vector multiplication, and transpose.

Two concrete types are exposed:

Python Type	Value Type	Index Type
`CSRMatrixd` / `CSRMatrix`	`double`	`Int64`
`CSRMatrixi`	`Int64`	`Int64`

Constructors

`CSRMatrix()`

Creates an empty 0×0 matrix.

`CSRMatrix(num_rows, num_cols)`

Creates a matrix with the given dimensions. All entries are initially zero (not stored).

Parameters:

Parameter	Type	Description
`num_rows`	`int`	Number of rows
`num_cols`	`int`	Number of columns

Example:

```python
import phynexis
m = phynexis.fields.CSRMatrix(3, 3)
print(m.get_num_rows(), m.get_num_cols())
```

Output:

```text
3 3
```

Methods

`resize(num_rows, num_cols)`

Resizes the matrix. Existing data is preserved where possible.

Parameters:

Parameter	Type	Description
`num_rows`	`int`	New row count
`num_cols`	`int`	New column count

`set_element(row, col, value)`

Sets a single element. If the element already exists, its value is overwritten.

Parameters:

Parameter	Type	Description
`row`	`int`	Row index
`col`	`int`	Column index
`value`	`float` / `int`	Value to set

Example:

```python
import phynexis
m = phynexis.fields.CSRMatrix(3, 3)
m.set_element(0, 0, 1.0)
m.set_element(1, 1, 2.0)
m.set_element(2, 2, 3.0)
print(m.get_element(0, 0), m.get_element(1, 1))
```

Output:

```text
1.0 2.0
```

`get_element(row, col)`

Returns the value at `(row, col)`. Returns `0.0` if the element is not explicitly stored.

Parameters:

Parameter	Type	Description
`row`	`int`	Row index
`col`	`int`	Column index

Returns: `float` / `int`

`has_element(row, col)`

Returns `True` if the element is explicitly stored (non-zero and inserted).

Parameters:

Parameter	Type	Description
`row`	`int`	Row index
`col`	`int`	Column index

Returns: `bool`

`set_row(row, cols, vals)`

Replaces the entire row with the given column indices and values.

Parameters:

Parameter	Type	Description
`row`	`int`	Row index
`cols`	`list[int]`	Column indices
`vals`	`list[float]`	Corresponding values

Example:

```python
import phynexis
m = phynexis.fields.CSRMatrix(3, 3)
m.set_row(0, [0, 2], [1.0, 3.0])
print(m.get_row(0))
print(m.get_row_indices(0))
```

Output:

```text
[1.0, 3.0]
[0, 2]
```

`get_row(row)`

Returns the non-zero values in the given row.

Parameters:

Parameter	Type	Description
`row`	`int`	Row index

Returns: `list[float]` / `list[int]`

`get_row_indices(row)`

Returns the column indices of non-zero values in the given row.

Parameters:

Parameter	Type	Description
`row`	`int`	Row index

Returns: `list[int]`

`get_row_size(row)`

Returns the number of non-zero elements in the given row.

Parameters:

Parameter	Type	Description
`row`	`int`	Row index

Returns: `int`

`set_elements(rows, cols, vals)`

Batch-sets multiple elements.

Parameters:

Parameter	Type	Description
`rows`	`list[int]`	Row indices
`cols`	`list[int]`	Column indices
`vals`	`list[float]`	Values

Example:

```python
import phynexis
m = phynexis.fields.CSRMatrix(2, 2)
m.set_elements([0, 1], [1, 0], [5.0, 7.0])
print(m.get_row(0), m.get_row(1))
```

Output:

```text
[5.0] [7.0]
```

`add_elements(rows, cols, vals)`

Adds values to existing elements. If an element does not exist, it is created.

Parameters:

Parameter	Type	Description
`rows`	`list[int]`	Row indices
`cols`	`list[int]`	Column indices
`vals`	`list[float]`	Values to add

Example:

```python
import phynexis
m = phynexis.fields.CSRMatrix(2, 2)
m.set_elements([0], [0], [10.0])
m.add_elements([0], [0], [5.0])
print(m.get_element(0, 0))
```

Output:

```text
15.0
```

`multiply(other)`

Matrix–matrix multiplication. Returns a new `CSRMatrix`.

Parameters:

Parameter	Type	Description
`other`	`CSRMatrix`	Right-hand side matrix

Returns: `CSRMatrix`

`multiply(vector)`

Matrix–vector multiplication.

Parameters:

Parameter	Type	Description
`vector`	`list[float]`	Dense vector (length must match column count)

Returns: `list[float]`

Example:

```python
import phynexis
m = phynexis.fields.CSRMatrix(2, 2)
m.set_elements([0, 1], [0, 1], [2.0, 3.0])
v = m.multiply([1.0, 1.0])
print(v)
```

Output:

```text
[2.0, 3.0]
```

`transpose()`

Returns the transpose of the matrix.

Returns: `CSRMatrix`

Example:

```python
import phynexis
m = phynexis.fields.CSRMatrix(2, 3)
m.set_element(0, 1, 5.0)
mt = m.transpose()
print(mt.get_num_rows(), mt.get_num_cols())
print(mt.get_element(1, 0))
```

Output:

```text
3 2
5.0
```

`get_num_rows()` / `get_num_cols()`

Returns the matrix dimensions.

Returns: `int`

`get_total_elements()`

Returns the total number of stored elements (including duplicates if any).

Returns: `int`

`get_nnz()`

Returns the number of non-zero elements.

Returns: `int`

`get_sparsity()`

Returns the sparsity ratio (`1 - nnz / (rows * cols)`).

Returns: `float`

`get_memory_usage()`

Returns the estimated memory usage in bytes.

Returns: `int`

`optimize_memory()`

Reclaims unused capacity in all rows.

`size()`

Returns the total number of stored elements. Same as `get_total_elements()`.

Returns: `int`

`empty()`

Returns `True` if the matrix contains no stored elements.

Returns: `bool`

Unexposed C++ API

The following C++ APIs are not exposed via pybind:

- `clear()` — removes all elements
- `row(i)` — returns a row view (`CSRMatrixRowView`)
- `get_row_data_ptr(i)` — returns raw row data pointer

Known Issues

`clear()` not exposed

The `clear()` method is not bound to Python. To clear a matrix, create a new instance or use `resize(0, 0)` as a workaround.

Status: noted

Field Views

C++: `phynexis::fields::{FieldViewBase, ScalarFieldView, Vec3FieldView, Vec4FieldView, Vec6FieldView, VecNFieldView, VecXFieldView}` **Python**: `phynexis.fields` **Headers**: `src/fields/views/*.hpp`

Field views are **non-owning wrappers** over existing scalar/vector fields. They provide a unified, type-safe interface for reading and writing field data without copying memory. Views are commonly used for SOA (Structure of Arrays) access patterns — e.g., treating three scalar fields `x`, `y`, `z` as a single vector field.

All view types inherit from `FieldViewBase`.

Type Summary

Python Class	C++ Type	Element / Component Type
`FieldViewBase`	`FieldViewBase`	— (abstract)
`ScalarFieldView` (alias `FieldView`)	`ScalarFieldView<Double>`	`float`
`Int32FieldView` (alias `Idx32FieldView`)	`ScalarFieldView<Int32>`	`int` (32-bit)
`Int64FieldView` (alias `Idx64FieldView`)	`ScalarFieldView<Int64>`	`int` (64-bit)
`BoolFieldView`	`ScalarFieldView<bool>`	`bool`
`Vec3dField`	`Vec3FieldView<Double>`	`float`
`Vec3iField`	`Vec3FieldView<Int32>`	`int` (32-bit)
`Vec4dField` (alias `QuaternionField`)	`Vec4FieldView<Double>`	`float`
`Vec6dField`	`Vec6FieldView<Double>`	`float`
`VecNdField`	`VecNFieldView<Double>`	`float`
`VecXdFieldView`	`VecXFieldView<Double>`	`float`
`VecXiFieldView`	`VecXFieldView<Int32>`	`int` (32-bit)

Common API (FieldViewBase)

All views share the following methods.

`is_valid()`

Returns `True` if the view references valid (non-null) underlying fields.

`__bool__()`

Same as `is_valid()`. Allows `if view:` syntax.

`size()` / `__len__()`

Returns the number of elements (rows) in the view.

`empty()`

Returns `True` if `size() == 0`.

`get_name(delimiter)`

Returns the base name of the view. For compound views (Vec3, Vec4, etc.), this strips the component suffix using the given delimiter.

Parameters:

Parameter	Type	Default	Description
`delimiter`	`str`	`"."`	Delimiter used to split component names

Known issue: The `delimiter` default is not automatically supplied from Python. Pass it explicitly: `view.get_name(".")`.

`get_field_names()`

Returns a `list[str]` of the underlying field names.

ScalarFieldView

Single-field view. Wraps one `Field<T>` and provides direct element access.

Constructors

Signature	Description
`ScalarFieldView()`	Empty (invalid) view.
`ScalarFieldView(field)`	View over a `ScalarField` (or `FieldBase`).

Example:

```python
import phynexis
F = phynexis.fields

fm = F.FieldManager()
h = fm.create_field(F.FieldMeta("temp", F.FieldType.Scalar, F.ValueType.Double))
f = fm.get_field(h)

view = F.ScalarFieldView(f)
print(view.size(), view.is_valid())  # 0 True

# Element access
view.resize(3)
view[0] = 1.0
view[1] = 2.0
print(view[0], view[1])  # 1.0 2.0
```

`field_ptr()`

Returns the underlying `Field<T>*` pointer.

`field_ref()`

Returns a reference to the underlying field. Same as dereferencing `field_ptr()`.

`data()`

Returns a pointer to the raw data buffer. **Unsafe** from Python — use `__getitem__` / `__setitem__` instead.

`__getitem__(idx)` / `__setitem__(idx, value)`

Direct element read/write. No bounds checking; use with caution.

`__getattr__(name)`

Forwards unknown attribute access to the underlying field object. Allows calling field methods through the view.

Vec3FieldView

View over three scalar fields interpreted as `(x, y, z)` components.

Constructors

Signature	Description
`Vec3dField(x, y, z)`	View over three `ScalarField` (or `FieldBase`) objects.
`Vec3iField(x, y, z)`	View over three `Int32Field` objects.

Example:

```python
import phynexis
F = phynexis.fields

fm = F.FieldManager()
hx = fm.create_field(F.FieldMeta("pos.x", F.FieldType.Scalar, F.ValueType.Double))
hy = fm.create_field(F.FieldMeta("pos.y", F.FieldType.Scalar, F.ValueType.Double))
hz = fm.create_field(F.FieldMeta("pos.z", F.FieldType.Scalar, F.ValueType.Double))

x = fm.get_field(hx)
y = fm.get_field(hy)
z = fm.get_field(hz)

for f in [x, y, z]:
    f.resize(3)

v3 = F.Vec3dField(x, y, z)
print(v3.size())         # 3
print(v3.get_name(".")) # "pos"

# Component access (returns ScalarFieldView)
print(v3.x().size())     # 3

# Indexed component access (returns scalar value)
print(v3.x(0))           # 0.0 (default-initialized)

# __getitem__ returns Vec3d (Double version only)
v3.x()[0] = 1.0
v3.y()[0] = 2.0
v3.z()[0] = 3.0
print(v3[0])             # Vec3d(1.0, 2.0, 3.0)
```

Component Access

Method	Returns	Description
`x()`	`ScalarFieldView`	View over the x-component field
`y()`	`ScalarFieldView`	View over the y-component field
`z()`	`ScalarFieldView`	View over the z-component field
`x(i)`	`float` / `int`	Value of x at index `i`
`y(i)`	`float` / `int`	Value of y at index `i`
`z(i)`	`float` / `int`	Value of z at index `i`

Vec4FieldView

View over four scalar fields interpreted as `(w, x, y, z)` components. Exposed as `Vec4dField` and aliased as `QuaternionField`.

Constructors

Signature	Description
`Vec4dField(w, x, y, z)`	View over four `ScalarField` (or `FieldBase`) objects.

Component Access

Method	Returns	Description
`w()`	`ScalarFieldView`	View over the w-component field
`x()`	`ScalarFieldView`	View over the x-component field
`y()`	`ScalarFieldView`	View over the y-component field
`z()`	`ScalarFieldView`	View over the z-component field
`w(i)`	`float`	Value of w at index `i`
`x(i)`	`float`	Value of x at index `i`
`y(i)`	`float`	Value of y at index `i`
`z(i)`	`float`	Value of z at index `i`

Vec6FieldView

View over six scalar fields interpreted as symmetric tensor components `(xx, yy, zz, xy, xz, yz)`.

Constructors

Signature	Description
`Vec6dField(xx, yy, zz, xy, xz, yz)`	View over six `ScalarField` (or `FieldBase`) objects.

Component Access

Method	Returns	Description
`xx()`	`ScalarFieldView`	View over the xx-component field
`yy()`	`ScalarFieldView`	View over the yy-component field
`zz()`	`ScalarFieldView`	View over the zz-component field
`xy()`	`ScalarFieldView`	View over the xy-component field
`xz()`	`ScalarFieldView`	View over the xz-component field
`yz()`	`ScalarFieldView`	View over the yz-component field

Note: Indexed component access (`xx(i)`, etc.) is **not bound** in Python.

VecNFieldView

View over N scalar fields of arbitrary dimension. Unlike Vec3/Vec4/Vec6, the dimension is determined at runtime by the number of fields passed.

Constructors

Signature	Description
`VecNdField()`	Empty (invalid) view.

Note: Constructors taking a field list (`VecX<FieldBase*>`) are **not bound** in Python because `VecX<FieldBase*>` is not exposed. Create views through `FieldManager` layouts or C++ APIs instead.

`__getitem__(index)`

Returns a `ScalarFieldView` over the field at the given index.

VecXFieldView

View over a `VecXField<T>` (ragged/jagged array field). Provides row-level access.

Constructors

Signature	Description
`VecXdFieldView(field)`	View over a `VecXdField`.
`VecXiFieldView(field)`	View over a `VecXiField`.

`__call__(i, j)`

Access element at row `i`, column `j`. Returns a mutable reference.

```python
vxf = F.VecXdField(3)
vxf.push_back(0, 1.0)
vxf.push_back(0, 2.0)

view = F.VecXdFieldView(vxf)
print(view(0, 0))  # 1.0
print(view(0, 1))  # 2.0
```

`row_size(i)`

Returns the number of elements in row `i`.

`row_capacity(i)`

Returns the allocated capacity of row `i`.

`get_total_elements()`

Returns the total number of stored elements across all rows.

`__getitem__(i)`

Returns a `VecXFieldRowView<T>` for row `i`.

Known Issues

`get_name()` requires explicit delimiter argument

The C++ default argument `delimiter = "."` is not propagated through pybind11. Always pass a delimiter string:

```python
name = view.get_name(".")  # OK
name = view.get_name()     # TypeError
```

Status: noted

`LinkedFieldRowView.data()` may segfault

Iterating a `LinkedFieldRowView` returned from `LinkedFieldView.get_targets()` or `get_sources()` can crash. The underlying C++ `RowView` is a proxy type whose lifetime rules are not fully captured by the Python binding.

Workaround: Access individual elements by index instead of calling `.data()`.

Status: noted

`VecNFieldView` list constructor not exposed

`VecNdField(fields: list[FieldBase])` is not available because `VecX<FieldBase*>` is not registered with pybind11.

Workaround: Use default constructor; practical construction requires C++-side `BoundLayout` or `FieldManager` integration.

Status: noted

`Vec6FieldView` indexed accessors not bound

`xx(i)`, `yy(i)`, etc. are not exposed in Python. Use the component view (`v6.xx()`) and then index the `ScalarFieldView`.

Workaround:

```python
v6.xx()[i]  # instead of v6.xx(i)
```

Status: noted

LinkedFieldView

C++: `phynexis::fields::LinkedFieldView` **Python**: `phynexis.fields.LinkedFieldView` **Header**: `src/fields/views/linked_field_view.hpp`

Bidirectional many-to-many link view over two `VecXField<Int64>` storages (`a_to_b` and `b_to_a`). Each row stores a list of linked indices. `LinkedFieldView` itself does not allocate index storage; it borrows the two `VecXField` objects and provides thread-safe `link()` / `unlink()` operations.

`LinkedFieldView` also supports **attached data**: `VecXField<T>` objects can be registered via `attach()`. When a link is created or removed, the attached fields are updated in sync (append on link, swap-remove on unlink).

Related Types

Python Class	C++ Type	Description
`LinkedFieldView`	`LinkedFieldView`	Bidirectional link manager
`LinkedFieldRowView`	`LinkedFieldRowView`	Read-only view of one link row
`ILinkedFieldAttachedData`	`ILinkedFieldAttachedData`	Abstract base for attached data adapters
`VecXFieldAttachedAdapterDouble`	`VecXFieldAttachedAdapter<Double>`	Adapter for `VecXdField` attached data
`VecXFieldAttachedAdapterInt32`	`VecXFieldAttachedAdapter<Int32>`	Adapter for `VecXiField` attached data

LinkedFieldView

Constructors

Signature	Description
`LinkedFieldView()`	Empty (invalid) view.
`LinkedFieldView(a_to_b, b_to_a)`	Links over two `VecXlField` instances.

The constructor takes **pointers** (kept alive by pybind11) and stores them internally.

Example:

```python
import phynexis
F = phynexis.fields

# Two VecXlField storages: 3 targets (A) x 3 sources (B)
a2b = F.VecXlField(3)
b2a = F.VecXlField(3)

lfv = F.LinkedFieldView(a2b, b2a)
print(lfv.size_targets(), lfv.size_sources())  # 3 3

# Create links
lfv.link(0, 1)
lfv.link(0, 2)
lfv.link(1, 0)

# Query targets of A=0
row = lfv.get_targets(0)
print(row.size())       # 2
print(row[0], row[1])   # 1 2

# Query sources of B=1
row2 = lfv.get_sources(1)
print(row2[0])          # 0
```

`link(a, b)`

Creates a bidirectional link between `a` (target index) and `b` (source index). Thread-safe.

`unlink(a, b)`

Removes the bidirectional link between `a` and `b`. Uses swap-remove on the internal rows, so link order is not preserved. Thread-safe.

`clear_links()`

Removes all links and clears attached data rows.

`size_targets()` / `size_sources()`

Returns the number of rows in `a_to_b` / `b_to_a` respectively.

`get_targets(a)` / `get_sources(b)`

Returns a `LinkedFieldRowView` containing the linked indices for target `a` / source `b`.

`a_to_b_ptr()` / `b_to_a_ptr()`

Returns the underlying `VecXlField` pointers.

`attach(field)`

Attaches a `VecXdField` or `VecXiField` to be kept in sync with links. On `link(a, b)`, a new element is appended to row `a` of the attached field. On `unlink(a, b)`, the element at the same index is swap-removed.

Supported field types: `VecXdField`, `VecXiField`

`attached_count()`

Returns the number of attached data fields.

LinkedFieldRowView

Read-only view of a single row in a `LinkedFieldView`. Created by `get_targets()` or `get_sources()`.

`size()` / `__len__()`

Returns the number of linked indices in this row.

`__getitem__(i)`

Returns the linked index at position `i`.

`data()`

Returns the raw row data as a Python `list`. **Known issue:** This method may segfault due to lifetime issues with the underlying proxy pointer.

Workaround: Iterate using `__getitem__` or `for i in range(row.size()): row[i]`.

Attached Data Adapters

`ILinkedFieldAttachedData` is the abstract base class for attached data. Two concrete adapters are exposed:

`VecXFieldAttachedAdapterDouble(field)`

Wraps a `VecXdField` for use as attached data. The field must have the same row layout as the `a_to_b` storage.

`VecXFieldAttachedAdapterInt32(field)`

Wraps a `VecXiField` for use as attached data.

Example:

```python
# Create a VecXdField with matching row layout
weights = F.VecXdField(3)
weights.push_back(0, 1.0)
weights.push_back(1, 2.0)

# Attach to LinkedFieldView
lfv.attach(weights)
print(lfv.attached_count())  # 1

# Now link() will append a default value to row 'a' of weights
lfv.link(0, 2)
```

Known Issues

`LinkedFieldRowView.data()` segfault

Calling `.data()` on a `LinkedFieldRowView` may crash. The underlying C++ `RowView` holds a raw pointer to `VecXField` row data; the Python binding does not extend the parent `LinkedFieldView` lifetime.

Workaround: Access elements by index.

Status: noted

`LinkedFieldView` default constructor not bound

`LinkedFieldView()` (no-arg default constructor) is **not exposed** in Python.

Workaround: Always construct with two `VecXlField` arguments.

Status: noted

`get_name()` requires explicit delimiter

Same as other field views: `view.get_name(".")` works; `view.get_name()` raises `TypeError`.

Status: noted

Operators

Python: `phynexis.fields` **Headers**: `src/fields/operators/*.hpp`

Free functions for element-wise math, reductions, prefix sums, and sorting on `Field` objects. All functions operate on `phynexis.fields` types and return new fields or scalar values.

Math

Element-wise mathematical operations. All functions accept `ScalarField` and return a new `ScalarField` of the same size.

Function	Signature	Description
sqrt	sqrt(field)	Square root
sqr	sqr(field)	Square (x * x)
sin	sin(field)	Sine
cos	cos(field)	Cosine
exp	exp(field)	Exponential (e^x)
abs	abs(field)	Absolute value
log	log(field)	Natural logarithm
pow	pow(field, exponent)	Power (x ** exponent)
tan	tan(field)	Tangent
floor	floor(field)	Floor
ceil	ceil(field)	Ceiling
round	round(field)	Round to nearest integer
fabs	fabs(field)	Floating-point absolute value
atan	atan(field)	Arc tangent
sinh	sinh(field)	Hyperbolic sine
max	max(a, b) / max(field, value)	Element-wise maximum
min	min(a, b) / min(field, value)	Element-wise minimum

Example:

```python
import phynexis
F = phynexis.fields

fm = F.FieldManager()
h = fm.create_field(F.FieldMeta("x", F.FieldType.Scalar, F.ValueType.Double))
f = fm.get_field(h)
f.resize(3)
f[0] = 1.0
f[1] = 4.0
f[2] = 9.0

print(F.sqrt(f)[0])   # 1.0
print(F.sqrt(f)[1])   # 2.0
print(F.sqr(f)[0])    # 1.0
print(F.pow(f, 0.5)[1])  # 2.0
```

Factory Functions

Function	Signature	Description
zeros	zeros(size)	ScalarField of zeros
ones	ones(size)	ScalarField of ones
linspace	linspace(start, end, num)	num evenly spaced values
arange	arange(start, end, step=1)	Range with step
fill	fill(field, value)	Fill field with a scalar value

Example:

```python
z = F.zeros(5)
o = F.ones(3)
ls = F.linspace(0.0, 1.0, 5)   # [0.0, 0.25, 0.5, 0.75, 1.0]
```

```
ar = F.arange(0.0, 5.0, 1.0)   # [0.0, 1.0, 2.0, 3.0, 4.0]
```

Vector Operations

`dot(a, b)`

Element-wise dot product of two `Vec3dField` objects. Returns a `ScalarField`.

Example:

```python
a = F.Vec3dField(ax, ay, az)
b = F.Vec3dField(bx, by, bz)
d = F.dot(a, b)  # ScalarField of a[i] . b[i]
```

`cross(a, b)`

Element-wise cross product of two `Vec3dField` objects.

Returns: `_Vec3dFieldCrossResult` — call `.get()` to obtain the `Vec3dField`.

Example:

```python
c = F.cross(a, b)
cv = c.get()
print(cv.x(0), cv.y(0), cv.z(0))
```

Known issue: Calling `.get()` may produce a segfault in some cases due to lifetime issues with the internal `shared_ptr` fields.

Reduction

Aggregate operations that return a single scalar value.

Function	Signature	Description
`sum`	`sum(field)`	Sum of all elements
`mean`	`mean(field)`	Arithmetic mean
`variance`	`variance(field)`	Population variance
`std_dev`	`std_dev(field)`	Standard deviation
`max_element`	`max_element(field)`	Maximum value
`min_element`	`min_element(field)`	Minimum value
`any`	`any(bool_field)`	`True` if any element is `True`
`all`	`all(bool_field)`	`True` if all elements are `True`
`count`	`count(bool_field)`	Count of `True` elements

Example:

```python
print(F.sum(f))          # 14.0
print(F.mean(f))         # 4.666666666666667
print(F.max_element(f)) # 9.0
print(F.min_element(f)) # 1.0
```

Prefix Sum

Cumulative sum operations.

`PrefixSumMode` enum

Value	Description
`PrefixSumMode.Inclusive`	Include current element in sum
`PrefixSumMode.Exclusive`	Exclude current element in sum

Functions

Function	Signature	Description
`prefix_sum_inclusive`	`prefix_sum_inclusive(field)`	Inclusive scan (returns new field)
`prefix_sum_exclusive`	`prefix_sum_exclusive(field)`	Exclusive scan (returns new field)
`prefix_sum`	`prefix_sum(field, mode=Inclusive)`	Generic scan (returns new field)
`prefix_sum_serial`	`prefix_sum_serial(field, mode=Inclusive)`	Single-threaded scan
`prefix_sum_inplace_inclusive`	`prefix_sum_inplace_inclusive(field)`	In-place inclusive scan
`prefix_sum_inplace_exclusive`	`prefix_sum_inplace_exclusive(field)`	In-place exclusive scan
`prefix_sum_inplace`	`prefix_sum_inplace(field, mode=Inclusive)`	Generic in-place scan

Example:

```python
f.resize(5)
for i in range(5):
    f[i] = float(i + 1)  # [1, 2, 3, 4, 5]

ps = F.prefix_sum_inclusive(f)
print([ps[i] for i in range(5)])  # [1, 3, 6, 10, 15]

ps2 = F.prefix_sum_exclusive(f)
print([ps2[i] for i in range(5)])  # [0, 1, 3, 6, 10]
```

Sort

`sort(field)`

In-place sort of a `ScalarField` in ascending order.

`sort(keys, values)`

Sort `keys` and reorder `values` accordingly (key-value sort).

`sort_by_value(keys, values)`

Sort `keys` by the corresponding `values` (indirect sort).

Supported types: `ScalarField`, `Int32Field`, `Int64Field` for both keys and values.

Example:

```python
k = F.Int32Field(5)
v = F.Int32Field(5)
for i in range(5):
    k[i] = 4 - i
    v[i] = i

```

```
F.sort_by_value(k, v)
print([k[i] for i in range(5)])  # [4, 3, 2, 1, 0] (k permuted so v is sorted)
```

Radix Sort

High-performance integer sort using radix sort algorithm.

`radix_sort(keys, values)`

Sort integer `keys` and reorder `values` accordingly.

`radix_sort_by_value(keys, values)`

Sort `keys` by integer `values`.

Supported types: `Int32Field`, `Int64Field` for both keys and values.

Example:

```python
k = F.Int32Field(5)
v = F.Int32Field(5)
for i in range(5):
    k[i] = 4 - i
    v[i] = i

F.radix_sort(k, v)
print([k[i] for i in range(5)])  # [0, 1, 2, 3, 4]
```

Known Issues

`cross().get()` may segfault

The `cross()` function returns a `_Vec3dFieldCrossResult` object. Calling `.get()` on it can crash in some environments due to `shared_ptr` lifetime management between pybind11 and the internal `Vec3Field` view.

Workaround: Use manual component-wise cross product if stability is required.

Status: noted

I/O and Utilities

Python: `phynexis.fields` **Headers**: `src/fields/utils/*.hpp`

Free functions for field serialization, MPI communication, and console output. These are utility functions that complement the core field data structures.

Console Output

`field_to_string(field)`

Returns a human-readable string representation of a scalar field.

Supported types: `ScalarField`, `Int32Field`, `Int64Field`

Example:

```python
import phynexis
F = phynexis.fields

fm = F.FieldManager()
h = fm.create_field(F.FieldMeta("rho", F.FieldType.Scalar, F.ValueType.Double))
f = fm.get_field(h)
f.resize(3)
f[0] = 1.0
f[1] = 2.0
f[2] = 3.0

print(F.field_to_string(f))
# ' name="rho" size=3 [1, 2, 3]'
```

`vecx_field_to_string(field)`

Returns a string representation of a `VecXdField` or `VecXiField`.

Supported types: `VecXdField`, `VecXiField`

Field I/O

VTK I/O

`write_vtk(field, path, file)`

Writes a scalar field to a VTK file.

Parameters:

Parameter	Type	Description
`field`	`ScalarField` / `Int32Field` / `Int64Field` / `BoolField`	Field to write
`path`	`str`	Directory path
`file`	`str`	Filename

`read_vtk(field, path, file)`

Reads a VTK file into an existing field.

Parameters:

Parameter	Type	Description
`field`	`ScalarField` / `Int32Field` / `Int64Field` / `BoolField`	Field to populate
`path`	`str`	Directory path
`file`	`str`	Filename

Note: VTK I/O requires the field to be part of a `FieldManager` with associated coordinate data. Standalone fields without point positions are not writable.

Example (read_vtk):

```python
f2 = F.ScalarField()
F.read_vtk(f2, "/tmp/data", "field.vtk")
```

Binary I/O

`write_binary(field, path, file)`

Writes a field to a binary file.

`read_binary(field, path, file)`

Reads a binary file into an existing field.

Example:

```python
F.write_binary(f, "/tmp/data", "field.bin")

f2 = F.ScalarField()
F.read_binary(f2, "/tmp/data", "field.bin")
print([f2[i] for i in range(f2.size())])
```

JSON I/O

`field_to_json(field)`

Converts a field to a Python `dict`.

Returns: `dict` with keys `name`, `size`, `data`

Example:

```python
j = F.field_to_json(f)
print(j)
# {'name': 'rho', 'size': 3, 'data': [1.0, 2.0, 3.0]}
```

`field_from_json(field, json)`

Populates a field from a Python `dict`.

Parameters:

Parameter	Type	Description
`field`	`ScalarField` / `Int32Field` / `Int64Field` / `BoolField`	Field to populate
`json`	`dict`	Dictionary with `name`, `size`, `data` keys

Example:

```python
f2 = F.ScalarField(3)
F.field_from_json(f2, {'name': 'copy', 'size': 3, 'data': [10.0, 20.0, 30.0]})
print(f2[0])  # 10.0
```

Supported types for all I/O operations: `ScalarField`, `Int32Field`, `Int64Field`, `BoolField`

Manager I/O

`pack_manager(manager, indices)`

Serializes a subset of fields from a `FieldManager` into a `bytes` buffer.

Parameters:

Parameter	Type	Description
`manager`	`FieldManager`	Source manager
`indices`	`list[int]`	Element indices to pack

Returns: `bytes`

`unpack_manager(data, manager, target_indices)`

Deserializes a `bytes` buffer into a `FieldManager` at the specified target indices.

Parameters:

Parameter	Type	Description
`data`	`bytes`	Buffer from `pack_manager`
`manager`	`FieldManager`	Target manager
`target_indices`	`list[int]`	Target element indices

Example:

```python
fm = F.FieldManager()
# ... populate fm ...

# Pack elements 0 and 2
blob = F.pack_manager(fm, [0, 2])

# Unpack into a new manager at indices 10 and 11
fm2 = F.FieldManager()
F.unpack_manager(blob, fm2, [10, 11])
```

VecXField I/O

`pack_vecx_field(field, indices)`

Packs a subset of elements from a `VecXdField` or `VecXiField` into a `bytes` buffer.

Parameters:

Parameter	Type	Description
`field`	`VecXdField` / `VecXiField`	Source field
`indices`	`list[int]`	Row indices to pack

Returns: `bytes`

`unpack_vecx_field(field, data, target_indices)`

Unpacks a `bytes` buffer into a `VecXdField` or `VecXiField` at target rows.

Parameters:

Parameter	Type	Description
`field`	`VecXdField` / `VecXiField`	Target field
`data`	`bytes`	Buffer from `pack_vecx_field`
`target_indices`	`list[int]`	Target row indices

Example:

```python
vf = F.VecXdField(5)
vf.push_back(0, 1.0)
vf.push_back(0, 2.0)

blob = F.pack_vecx_field(vf, [0])

```

```
vf2 = F.VecXdField(5)
F.unpack_vecx_field(vf2, blob, [3])
print(vf2.get_row(3))  # [1.0, 2.0]
```

MPI Utilities

Note: These functions require `mpi4py` and an MPI environment.

`gather_data(local, global)`

Gathers data from all MPI ranks into a single `global` field on the root rank.

Parameters:

Parameter	Type	Description
`local`	`ScalarField` / `Int32Field`	Local data on each rank
`global`	`ScalarField` / `Int32Field`	Global buffer on root (pre-sized)

Supported types: `ScalarField`, `Int32Field`

`scatter_data(global, local, root, counts, displs)`

Scatters data from a `global` field on the root rank to `local` fields on all ranks.

Parameters:

Parameter	Type	Description
`global`	`ScalarField` / `Int32Field`	Global data on root
`local`	`ScalarField` / `Int32Field`	Local buffer on each rank (pre-sized)
`root`	`int`	Root rank
`counts`	`list[int]`	Number of elements per rank
`displs`	`list[int]`	Displacement offsets per rank

Supported types: `ScalarField`, `Int32Field`

Example:

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Gather: each rank has 2 elements
local = F.ScalarField(2)
local[0] = float(rank * 2)
local[1] = float(rank * 2 + 1)

if rank == 0:
    global_field = F.ScalarField(comm.Get_size() * 2)
else:
    global_field = F.ScalarField()

F.gather_data(local, global_field)
```

Known Issues

`field_to_json` / `field_from_json` require pybind11_json

If `pybind11_json` is not included in the binding compilation, `field_to_json` raises `TypeError: Unregistered type: nlohmann::json...`.

Status: fixed (added `#include <pybind11_json/pybind11_json.hpp>` to `pyfield_io.cpp`)

phynexis.parsim

Python: `phynexis.parsim` **pybind module**: `pyparsim` **Header**: `src/parsim/parsim.hpp`

Particle simulation framework. Provides node/edge/hyperedge graph structures, spatial indexing, operator-based simulation loop, and model factories for contact mechanics.

Module Status

Class / Group	Status	Doc
Schemas & BoundLayouts	done	schemas
Graph (NodeSet, EdgeSet, HyperEdgeSet, ComputationalGraph)	done	graph
Simulator, Context, OperatorSystem	done	simulator
Models & Utils	done	models-utils
Operators	done	operators
Views	done	views

Submodules

- **schema** — `NodeKinematicsSchema`, `NodeShapeSchema`, `NodeInertiaSchema`, `NodeBoundsSchema`, `NodeNodeLinkSchema`, `NodeGridLinkSchema`, `NodeDomainLinkSchema`
- **graph** — `ComputationalGraph`, `NodeSet`, `EdgeSet`, `HyperEdgeSet`, `SetBase`, `ShapeStore`
- **simulator** — `Simulator`, `Context`, `OperatorSystem`, `RuntimeState`, `DomainSystem`
- **models** — `RepulsionModel`, `LinearRepulsionModel`, `RepulsionModelFactory`
- **utils** — `NodeGenerator`, `FieldSampler`, `SpatialIndex`, `UniformGridIndex`
- **operators** — `phases`, `integration`, `forces`, `interaction`, `output` submodules

Quick Start

```python
import phynexis
pm = phynexis.parsim

# Create a graph with nodes
g = pm.ComputationalGraph()
ns = g.nodes()

# Add a node (requires schema + bound layout)
schema = pm.NodeKinematicsSchema()
layout = pm.BoundLayoutNodeKinematicsSchema.make_bound(ns.field_manager(), schema)
```

```
ns.set_view(layout)

# Initialize node storage
ns.initialize()
print(ns.size())  # 0
```

See also

- Python API overview
- fields module — field data structures used by parsim

Schemas

Python: `phynexis.parsim` **Headers**: `src/parsim/schema/*.hpp`

Schema definitions for particle node properties. Each schema describes a set of field slots (position, velocity, radius, etc.) that are allocated in a `FieldManager` and bound to a `NodeSet` via a `BoundLayout`.

All schemas inherit from `phynexis.fields.FieldSchema`.

Schema Types

Python Class	Description	Slots
`NodeKinematicsSchema`	Position, velocity, acceleration, orientation	`pos`, `vel`, `acc`, `quat`
`NodeShapeSchema`	Shape ID and radius	`shape_id`, `radius`
`NodeInertiaSchema`	Mass and inertia tensor	`mass`, `inertia`
`NodeBoundsSchema`	AABB bounding box	`min_bound`, `max_bound`
`NodeNodeLinkSchema`	Node-to-node connectivity	`target`, `link_type`
`NodeGridLinkSchema`	Grid-cell linkage	`cell_id`, `grid_level`
`NodeDomainLinkSchema`	MPI domain linkage	`domain_id`, `rank`

Constructors

All schemas have a default constructor:

```python
schema = pm.NodeKinematicsSchema()
```

`with_prefix(prefix)` (NodeNodeLinkSchema only)

Returns a copy of the schema with the given slot name prefix. Static method.

```python
schema = pm.NodeNodeLinkSchema.with_prefix("bond_")
```

BoundLayout

A `BoundLayout` binds a schema to a concrete `FieldManager`, allocating the actual fields.

Python Class	Schema
`BoundLayoutNodeKinematicsSchema`	`NodeKinematicsSchema`
`BoundLayoutNodeShapeSchema`	`NodeShapeSchema`
`BoundLayoutNodeInertiaSchema`	`NodeInertiaSchema`
`BoundLayoutNodeBoundsSchema`	`NodeBoundsSchema`
`BoundLayoutNodeNodeLinkSchema`	`NodeNodeLinkSchema`
`BoundLayoutNodeGridLinkSchema`	`NodeGridLinkSchema`
`BoundLayoutNodeDomainLinkSchema`	`NodeDomainLinkSchema`
`BoundLayoutSetSchema`	Generic set schema

Factory Methods

`make_bound(manager, schema)`

Static factory. Creates a bound layout and allocates fields in the given `FieldManager`.

Parameters:

Parameter	Type	Description
`manager`	`FieldManager`	Target field manager
`schema`	`*Schema`	Schema instance

Returns: `BoundLayout*` (concrete subclass)

Example:

```python
import phynexis
pm = phynexis.parsim
F = phynexis.fields

fm = F.FieldManager()
schema = pm.NodeKinematicsSchema()
layout = pm.BoundLayoutNodeKinematicsSchema.make_bound(fm, schema)
print(layout.is_valid())  # True
```

Instance Methods

Method	Description
`bind(manager)`	(Re-)bind to a different field manager
`refresh()`	Refresh field handles from manager
`is_valid()`	Returns `True` if all slots are bound
`manager()`	Returns the bound `FieldManager`
`handles()`	Returns a list of `FieldHandle` objects
`fields()`	Returns a list of `FieldBase*` objects
`slot_fields(slot)`	Returns fields for a named slot
`slot_fields_count(slot)`	Returns number of fields in a slot
`slot_fields_offset(slot)`	Returns offset of a slot in the handle list

Example:

```python
# Access fields via the layout
fields = layout.fields()
print(len(fields))

# Access a specific slot
```

```
pos_fields = layout.slot_fields("pos")
print(len(pos_fields))
```

Known Issues

`fields()` returns `FieldBase*` without RTTI resolution

`layout.fields()` returns a list of `FieldBase*` pointers. The concrete subclass (e.g., `ScalarField`) is not resolved via RTTI in the current binding.

Workaround: Use `layout.handles()` to get `FieldHandle` objects, then retrieve fields via `manager.get_field(handle)`.

Status: noted

Graph

Python: `phynexis.parsim` **Headers**: `src/parsim/graph/*.hpp`

Graph data structures for particle simulations. `ComputationalGraph` is the top-level container holding `NodeSet`, `EdgeSet`, `HyperEdgeSet`, and `ShapeStore`.

ComputationalGraph

Top-level graph container.

Constructors

Signature	Description
`ComputationalGraph()`	Empty graph.

Methods

Method	Returns	Description
`nodes()`	`NodeSet`	The node set
`edges()`	`EdgeSet`	The edge set
`hyper_edges()`	`HyperEdgeSet`	The hyperedge set
`shape_store()`	`ShapeStore`	The shape store
`node_count()`	`int`	Number of nodes
`edge_count()`	`int`	Number of edges
`hyper_edge_count()`	`int`	Number of hyperedges
`clear()`	—	Clear all sets
`is_empty()`	`bool`	True if all sets are empty
`save_to(path, file, opt)`	—	Save graph to file
`load_from(path, file, opt)`	—	Load graph from file
`inspect(filepath)`	—	Inspect file metadata

Example:

```python
import phynexis
pm = phynexis.parsim

g = pm.ComputationalGraph()
print(g.is_empty())  # True

ns = g.nodes()
print(type(ns).__name__)  # NodeSet
```

SetBase

Abstract base class for `NodeSet`, `EdgeSet`, `HyperEdgeSet`.

Methods

Method	Returns	Description
`ensure_fields()`	—	Ensure all schema fields exist
`schema()`	`FieldSchema`	The set's schema
`field_manager()`	`FieldManager`	Underlying field manager
`set_view(layout)`	—	Bind a `BoundLayout` to this set
`count()` / `size()`	`int`	Number of elements
`empty()`	`bool`	True if size == 0
`clear()`	—	Remove all elements
`emplace()`	`int`	Add a new element (returns ID)
`emplace_with_id(id)`	—	Add a new element with given ID
`erase(id)`	—	Remove element by ID
`allocate_id()`	`int`	Allocate a new unique ID
`save_to(path, file, opt)`	—	Save set data
`load_from(path, file, opt)`	—	Load set data
`inspect(filepath)`	—	Inspect file metadata

NodeSet

Particle/node storage.

Constructors

Signature	Description
`NodeSet()`	Empty node set.

Methods

Method	Description
`add_node(id, data)`	Add a node with JSON data
`add_nodes(data_list)`	Batch add nodes from list of JSON
`remove_node(id)`	Remove node by ID
`remove_nodes(ids)`	Batch remove nodes
`has_node(id)`	Check if node exists
`initialize()`	Initialize internal storage
`print_info()`	Print debug info
`inspect_fields()`	Inspect field layout
`views()`	Return node views

Example:

```python
import phynexis
pm = phynexis.parsim
F = phynexis.fields

ns = pm.NodeSet()

# Bind a schema layout
fm = F.FieldManager()
schema = pm.NodeKinematicsSchema()
layout = pm.BoundLayoutNodeKinematicsSchema.make_bound(fm, schema)
ns.set_view(layout)
ns.initialize()

# Add nodes
ns.add_node(0, {"pos": [0.0, 0.0, 0.0]})
ns.add_node(1, {"pos": [1.0, 0.0, 0.0]})
print(ns.size())  # 2
```

EdgeSet

Pairwise edge/link storage.

Constructors

Signature	Description
`EdgeSet()`	Empty edge set.

Methods

Method	Description
`initialize()`	Initialize internal storage
`print_info()`	Print debug info
`views()`	Return edge views

Note: No direct `add_edge` / `remove_edge` methods are exposed. Use `SetBase.emplace()` / `erase()` for bulk operations.

HyperEdgeSet

Many-to-many hyperedge storage.

Constructors

Signature	Description
`HyperEdgeSet()`	Empty hyperedge set.

Methods

Method	Description
`initialize()`	Initialize internal storage
`print_info()`	Print debug info
`views()`	Return hyperedge views

ShapeStore

Stores geometric shapes with stable handles.

Constructors

Signature	Description
`ShapeStore()`	Empty store.

Methods

Method	Returns	Description
`insert(shape)`	`int`	Insert shape clone, returns auto-allocated ID
`insert_with_id(id, shape)`	`int`	Insert with specified ID
`id_at(handle)`	`int`	Get ID from handle (-1 if invalid)
`find(id)`	`ShapeStoreHandle`	Find handle by ID
`get(handle)`	`ShapeStoreEntry`	Get entry by handle
`erase(id)`	—	Erase shape by ID
`contains(id)`	`bool`	Check if ID exists
`valid(handle)`	`bool`	Check if handle is valid
`set_label(id, label)`	—	Set label
`set_need_sync(id, need_sync)`	—	Set sync flag
`get_label_of(id)`	`str`	Get label
`initialize_mpi()`	—	Initialize MPI support
`sync()`	—	Sync across MPI ranks
`get_shape_ids()`	`list[int]`	All shape IDs
`size()`	`int`	Number of shapes
`to_json()`	`dict`	Export all shapes as JSON
`save_to(path, file, opt)`	—	Save to file
`load_from(path, file, opt)`	—	Load from file
`inspect(filepath)`	—	Inspect file metadata

Example:

```python
import phynexis
pm = phynexis.parsim
U = phynexis.utils

ss = pm.ShapeStore()
sphere = U.shape.Sphere(0.5)
sid = ss.insert(sphere)
print(sid)  # 0

entry = ss.get(ss.find(sid))
print(entry.label)  # ''

ss.set_label(sid, "particle")
print(ss.get_label_of(sid))  # "particle"
```

ShapeStoreEntry

Read-only entry returned by `ShapeStore.get()`.

Attribute	Type	Description
`shape`	`Shape`	The stored shape
`label`	`str`	User-defined label
`need_sync`	`bool`	MPI sync flag

ShapeStoreHandle

Lightweight handle with `index` and `epoch` fields.

Views

NodeViewById / NodeViewByIndex

Class	Description
`NodeViewById`	View nodes by stable ID
`NodeViewByIndex`	View nodes by contiguous index

EdgeViewById / EdgeViewByIndex

Class	Description
`EdgeViewById`	View edges by stable ID
`EdgeViewByIndex`	View edges by contiguous index

HyperEdgeViewById / HyperEdgeViewByIndex

Class	Description
`HyperEdgeViewById`	View hyperedges by stable ID
`HyperEdgeViewByIndex`	View hyperedges by contiguous index

PropertiesView

Generic property accessor view for nodes/edges.

Known Issues

EdgeSet / HyperEdgeSet lack add/remove convenience methods

Only `initialize()`, `print_info()`, and `views()` are exposed. Use `SetBase.emplace()` / `erase()` for modifications.

Status: noted

ShapeStoreHandle uses `module_local()`

`ShapeStoreHandle` is registered with `pybind11::module_local()` to avoid conflict with `FieldHandle` from `phynexis.fields`. Cross-module type checking (e.g., `isinstance(handle, FieldHandle)`) will not work.

Status: noted

Simulator

Python: `phynexis.parsim` **Headers**: `src/parsim/simulator/*.hpp`, `src/parsim/context/*.hpp`

Simulation driver, context management, and operator system.

Simulator

Main simulation driver.

Constructors

Signature	Description
`Simulator()`	Default simulator.

Methods

Method	Returns	Description
`step()`	—	Advance one time step
`run()`	—	Run until end condition
`context()`	`Context`	Simulation context
`operators()`	`OperatorSystem`	Operator system
`config()`	`SimulationConfig`	Current configuration
`save_to(path, file, opt)`	—	Save state
`load_from(path, file, opt)`	—	Load state

Example:

```python
import phynexis
pm = phynexis.parsim

sim = pm.Simulator()
ctx = sim.context()
print(type(ctx).__name__)  # Context
```

SimulationConfig

Configuration struct.

Attribute	Type	Description
`time_step`	`float`	Simulation time step
`end_time`	`float`	Simulation end time
`max_steps`	`int`	Maximum number of steps
`enable_gravity`	`bool`	Enable gravity
`gravity`	`Vec3d`	Gravity vector
`enable_logging`	`bool`	Enable logging

Context

Central simulation context managing all resources.

Methods

Method	Returns	Description
`initialize()`	—	Initialize context
`finalize()`	—	Finalize context
`graph()`	`ComputationalGraph`	The computational graph
`domain()`	`DomainSystem`	Domain decomposition system
`state()`	`RuntimeState`	Runtime state
`spatial_index()`	`SpatialIndex`	Spatial index (may be None)
`ensure_spatial_index()`	—	Ensure spatial index is created
`find_resource(name)`	`tuple`	Find resource handle `(index, epoch)`
`resource(name)`	`object`	Get typed resource (4 hardcoded keys)
`save_to(path, file, opt)`	—	Save context
`load_from(path, file, opt)`	—	Load context

Resource keys: - `"computational_graph"` → `ComputationalGraph` - `"domain_system"` → `DomainSystem` - `"runtime_state"` → `RuntimeState` - `"uniform_grid_index"` → `UniformGridIndex`

Example:

```python
ctx = pm.Context()
ctx.initialize()
g = ctx.graph()
print(g.is_empty())  # True
```

OperatorSystem

Container for simulation operators.

Methods

Method	Description
`insert(op)`	Insert an operator
`insert_with_label(label, op)`	Insert with a label
`get(label)`	Get operator by label
`get(handle)`	Get operator by handle
`erase(label)`	Remove operator by label
`enable(handle)`	Enable operator
`disable(handle)`	Disable operator
`size()`	Number of operators
`empty()`	True if no operators
`clear()`	Remove all operators

Known issue: `BaseOperator` is not bound in Python, so `insert()` is only usable with C++-created operators. The `operators` submodule exposes concrete operators but their insertion API is incomplete.

Status: noted

OperatorSystemEntry

Metadata for an operator.

Attribute	Type	Description
`op`	`handle`	Operator handle
`label`	`str`	Operator label

RuntimeState

Time and step tracking.

Methods

Method	Returns	Description
`step()`	`int`	Current step
`time()`	`float`	Current time
`time_step()`	`float`	Time step size
`set_step(v)`	—	Set step
`set_time(v)`	—	Set time
`set_time_step(v)`	—	Set time step
`advance(dt)`	—	Advance time by dt
`history()`	—	Get history recorder

DomainSystem

Spatial domain decomposition for MPI.

Methods

Method	Returns	Description
`initialize()`	—	Initialize domain decomposition
`config()`	`DomainSystemConfig`	Configuration
`subdomain_count()`	`int`	Number of subdomains
`domain_bound()`	`BoundingBox`	Global domain bounds
`self_subdomain_id()`	`int`	Local subdomain ID
`self_subdomain_bound()`	`BoundingBox`	Local subdomain bounds
`get_overlapped_subdomain(position)`	`int`	Subdomain overlapping position
`is_judge_domain(bbox_a, bbox_b)`	`bool`	Whether this rank judges bbox pair

DomainSystemConfig

Attribute	Type	Description
`min_bound`	`Vec3d`	Domain minimum corner
`max_bound`	`Vec3d`	Domain maximum corner
`subdivisions`	`Vec3i`	Subdivision counts per axis

SpatialIndex

Abstract base for spatial indexing.

Methods

Method	Returns	Description
`build(nodes)`	—	Build index from NodeSet
`query(position, radius)`	`list[int]`	Query nodes within radius
`update()`	—	Update index

UniformGridIndex

Uniform grid spatial index.

Methods

Method	Returns	Description
`build(nodes)`	—	Build index from NodeSet
`query(position, radius)`	`list[int]`	Query nodes within radius
`update()`	—	Update index
`config()`	`UniformGridIndexConfig`	Current config

UniformGridIndexConfig

Attribute	Type	Description
`min_bound`	`Vec3d`	Grid minimum corner
`max_bound`	`Vec3d`	Grid maximum corner
`grid_size`	`Vec3d`	Cell size
`update_mode`	`str`	Update strategy

ResourceRegistry / ContextResource

Resource factory pattern for extensible context resources.

ResourceRegistry

Method	Description
`register_factory(name, factory)`	Register a resource factory
`create(name)`	Create a resource by name

ContextResource

Base class for context-managed resources.

Known Issues

`OperatorSystem.insert()` unusable from Python

`BaseOperator` is not exposed in Python bindings, so operators created in Python cannot be inserted into `OperatorSystem`.

Workaround: Use C++-side operator creation or the `Simulator` factory methods.

Status: noted

`Context.resource()` only supports 4 hardcoded keys

Extensible resources fall through to an exception.

Status: noted

operators

C++: `phynexis::parsim::operators` **Python**: `phynexis.parsim.operators` **Header**: `src/parsim/operators/*.hpp`

Simulation operators that are registered with `OperatorSystem` and execute during specific simulation phases. Each operator defines `order()` (execution priority) and `enabled()` state.

Base Classes

`BaseOperator`

Abstract base class for all simulation operators.

Methods:

Method	Returns	Description
`name()`	`str`	Operator name
`order()`	`int`	Execution priority (lower = earlier)
`set_order(n)`	—	Set execution priority
`enabled()`	`bool`	Whether operator is active
`set_enabled(b)`	—	Enable or disable operator
`timings()`	—	Print timing statistics
`print_info()`	—	Print operator info

Phase Identifiers

Phase constants define when operators execute in the simulation loop.

Constant	Value	Description
`PreStep`	0	Before step begins
`PreBoundary`	1	Before boundary conditions
`PostBoundary`	2	After boundary conditions
`PreInteraction`	3	Before interaction detection
`PostInteraction`	4	After interaction detection
`PreForces`	5	Before force computation
`PostForces`	6	After force computation
`PreIntegration`	7	Before integration
`PostIntegration`	8	After integration
`PostStep`	9	After step completes

Example:

```python
import phynexis
ops = phynexis.parsim.operators

print("Phase IDs:", ops.PreStep, ops.PreForces, ops.PostStep)
```

Output:

```text
Phase IDs: 0 5 9
```

Integration Operators (`operators.integration`)

`IntegratorOperator(method=Verlet)`

Time integration operator using Euler or Verlet scheme.

Constructor:

Parameter	Type	Default	Description
`method`	`IntegratorMethod`	`Verlet`	Integration scheme

Methods:

Method	Returns	Description
`initialize(nodes)`	—	Bind to a `NodeSet`
`set_time_step(dt)`	—	Set time step size
`time_step()`	`float`	Current time step
`config()`	`IntegratorOperatorConfig`	Current configuration
`set_config(cfg)`	—	Set configuration

`IntegratorMethod`

Value	Description
`Euler`	Forward Euler integration
`Verlet`	Velocity Verlet integration (default)

`IntegratorOperatorConfig`

Property	Type	Access	Description
`method`	`IntegratorMethod`	read/write	Integration method

Example:

```python
import phynexis
ops = phynexis.parsim.operators

# Velocity Verlet (default)
io = ops.integration.IntegratorOperator()
io.set_time_step(0.001)
print("method:", io.config().method)
print("dt:", io.time_step())

# Euler
io2 = ops.integration.IntegratorOperator(
    ops.integration.IntegratorMethod.Euler)
print("Euler method:", io2.config().method)
```

Output:

```text
method: IntegratorMethod.Verlet
dt: 0.001
Euler method: IntegratorMethod.Euler
```

Force Operators (`operators.forces`)

`Gravity(label="gravity")`

Gravitational force operator. Applies gravity to all nodes.

Constructor:

Parameter	Type	Default	Description
`label`	`str`	—	Operator label
`gx, gy, gz`	`float`	—	Gravity components
`gravity_vector`	`Vec3d`	—	Gravity as vector (requires label)

Properties:

Property	Type	Access	Description
`gravity_vector`	`Vec3d`	read/write	Gravity vector (default `(0,0,-9.81)`)

`Damping(label, coefficient)`

Viscous damping operator. Applies damping proportional to velocity.

Constructor:

Parameter	Type	Default	Description
`label`	`str`	—	Operator label
`coefficient`	`float`	—	Damping coefficient

Properties:

Property	Type	Access	Description
`coefficient`	`float`	read/write	Damping coefficient

`RepulsionForce`

Contact repulsion force operator. Requires a `RepulsionModel` to define the force law.

Methods:

Method	Returns	Description
`initialize(nodes, edges)`	—	Bind to `NodeSet` and `EdgeSet`
`set_model(model)`	—	Set repulsion model (cloned)
`set_model_by_name(name)`	—	Set model by name from factory
`model()`	`RepulsionModel`	Get current model (reference)

Example:

```python
import phynexis
ops = phynexis.parsim.operators

# Gravity
g = ops.forces.Gravity()
print("gravity:", g.name(), g.gravity_vector)

# Damping
d = ops.forces.Damping("viscous", 0.1)
print("damping:", d.name(), d.coefficient)
```

Output:

```text
gravity: gravity Vec3d(0, 0, -9.81)
damping: damping 0.1
```

Interaction Operators (`operators.interaction`)

`InteractionUpdate(label="")`

Updates interactions (edges/hyperedges) based on current node positions.

`AdjacencyBuilder`

Builds adjacency structures using spatial index.

Methods:

Method	Returns	Description
`initialize(spatial_index, nodes, domain_system)`	—	Initialize with spatial index
`config()`	`AdjacencyBuilderConfig`	Current config
`set_config(cfg)`	—	Set config

`AdjacencyBuilderConfig` properties:

Property	Type	Default	Description
`max_dirty_ratio`	`float`	0.3	Max dirty rebuild ratio
`max_invalid_ratio`	`float`	0.3	Max invalid rebuild ratio

`UniformGridIndexBuilder`

Builds the uniform grid spatial index.

Methods:

Method	Returns	Description
`initialize(uniform_grid_index, nodes)`	—	Initialize with grid index and nodes

Output Operators (`operators.output`)

`UnifiedOutput(output_dir="output/", format=VTK, frequency=100)`

Simulation output writer supporting VTK and HDF5 formats.

Constructor:

Parameter	Type	Default	Description
`output_dir`	`str`	`"output/"`	Output directory
`format`	`OutputFormat`	`VTK`	File format
`frequency`	`int`	100	Output frequency (steps)

Methods:

Method	Returns	Description
`initialize(graph, state)`	—	Bind to graph and runtime state
`config()`	`UnifiedOutputConfig`	Current configuration

`OutputFormat`

Value	Description
`VTK`	VTK file format
`HDF5`	HDF5 file format

`OutputLogStyle`

Value	Description
`Summary`	Compact log output
`Verbose`	Detailed log output
`Silence`	No log output

`OutputMode`

Value	Description
`ByStep`	Output every N steps
`ByTime`	Output every N time units

UnifiedOutputConfig

Property	Type	Access	Description
format	OutputFormat	read/write	Output file format
output_dir	str	read/write	Output directory
frequency	int	read/write	Output frequency
output_mode	OutputMode	read/write	Output mode (step/time based)
time_interval	float	read/write	Time interval for ByTime mode
time_stamp_adjustable	bool	read/write	Adjustable timestamps
edges	bool	read/write	Output edges
hyper_edges	bool	read/write	Output hyperedges
shapes	bool	read/write	Output shapes
log_style	OutputLogStyle	read/write	Log verbosity

Example:

```python
import phynexis
ops = phynexis.parsim.operators

# VTK output (default)
uo = ops.output.UnifiedOutput("sim_output/", ops.output.OutputFormat.VTK)
print("output:", uo.name())

cfg = uo.config()
print("format:", cfg.format)
print("frequency:", cfg.frequency)
```

Output:

```text
output: unified_output
format: OutputFormat.VTK
frequency: 100
```

Models and Utils

Python: `phynexis.parsim` **Headers**: `src/parsim/models/*.hpp`, `src/parsim/utils/*.hpp`

Contact models, node generation utilities, and spatial indexing helpers.

RepulsionModel

Base class for contact repulsion models.

Methods

Method	Returns	Description
name()	str	Model name
clone()	RepulsionModel*	Deep copy

LinearRepulsionModel

Simple linear spring repulsion.

Constructors

Signature	Description
`LinearRepulsionModel()`	Default stiffness.
`LinearRepulsionModel(stiffness)`	Specify stiffness.

Methods

Method	Returns	Description
`stiffness()`	`float`	Current stiffness
`set_stiffness(v)`	—	Set stiffness

Example:

```python
import phynexis
pm = phynexis.parsim

model = pm.LinearRepulsionModel(1000.0)
print(model.stiffness())  # 1000.0
print(model.name())       # "LinearRepulsionModel"

clone = model.clone()
print(type(clone).__name__)  # LinearRepulsionModel
```

RepulsionModelFactory

Factory for creating repulsion models by name.

Static Methods

Method	Returns	Description
`create_model(name)`	`RepulsionModel*`	Create model by name
`get_available_models()`	`list[str]`	List registered model names
`is_registered(name)`	`bool`	Check if name is registered
`print_info()`	—	Print factory info

Example:

```python
print(pm.RepulsionModelFactory.get_available_models())
model = pm.RepulsionModelFactory.create_model("LinearRepulsionModel")
```

NodeGenerator

Generates node property sets from field samplers.

Constructors

Signature	Description
`NodeGenerator()`	Empty generator.

Methods

Method	Description
`add_field(slot, sampler)`	Add a field slot with sampler
`add_field(name, sampler)`	Add Vec3 field by name (spatial sampler)
`add_field(name, dist)`	Add scalar field by name (distribution)
`add_field(name, generator)`	Add field by callable (double/Vec3/JSON)
`erase_field(name)`	Remove a field slot
`contains(name)`	Check if slot exists
`slots()`	Get all slots
`num_slots()`	Number of slots
`slot(i)` / `slot(name)`	Get slot by index or name
`generate_one()`	Generate one node's properties (JSON)
`generate(count)`	Generate `count` nodes (list of JSON)
`clear()`	Remove all slots
`print_slots()`	Print slot info to stderr

Example:

```python
import phynexis
pm = phynexis.parsim
U = phynexis.utils

gen = pm.NodeGenerator()

# Add a scalar field with fixed value
gen.add_field("radius", lambda: 0.5)

# Add a Vec3 field with random position
sampler = U.sampling.spatial.UniformBoxSampler(
    U.Vec3d(0, 0, 0), U.Vec3d(1, 1, 1)
)
gen.add_field("pos", sampler)

# Generate 3 nodes
nodes = gen.generate(3)
print(len(nodes))  # 3
print(nodes[0])    # {'radius': 0.5, 'pos': [x, y, z]}
```

FieldSampler

Factory for creating field samplers used by `NodeGenerator`.

Static Methods

Method	Returns	Description
`make_scalar(dist)`	`FieldSampler`	From distribution
`make_scalar(generator)`	`FieldSampler`	From callable `() -> double`
`make_scalar(value)`	`FieldSampler`	From constant value
`make_vec3(sampler)`	`FieldSampler`	From spatial sampler
`make_vec3(generator)`	`FieldSampler`	From callable `() -> Vec3d`
`make_vec4(sampler)`	`FieldSampler`	From orientation sampler
`make_json(generator)`	`FieldSampler`	From callable `() -> dict`

Node Property Utils

Free functions for node property manipulation.

`apply_density(view, density)`

Applies density to node mass/inertia fields.

`assign_shape(view, shape_id)`

Assigns a shape ID to nodes.

`update_node_bounds(nodes, skin_factor=0.05, margin_factor=0.1)`

Updates AABB bounds for all nodes.

Example:

```python
ns = pm.NodeSet()
# ... setup and initialize ...

pm.update_node_bounds(ns, 0.05, 0.1)
```

SpatialIndex / UniformGridIndex

See simulator for full documentation.

Quick example:

```python
idx = pm.UniformGridIndex()
config = idx.config()
config.min_bound = U.Vec3d(0, 0, 0)
config.max_bound = U.Vec3d(10, 10, 10)
config.grid_size = U.Vec3d(1, 1, 1)

idx.build(ns)
result = idx.query(U.Vec3d(5, 5, 5), 2.0)
print(result)  # list of node indices within radius 2.0
```

Known Issues

`NodeGenerator.generate()` returns `list[dict]` not `VecX[json]`

The binding wraps `VecX<json>` to `py::list` for compatibility.

Status: noted

views

C++: `phynexis::parsim::{NodeView,EdgeView,HyperEdgeView,PropertiesView}` **Python**: `phynexis.parsim.{NodeViewByIndex`
Header: `src/parsim/views/*.hpp`

View classes provide indexed or ID-based access to graph elements and their properties. Each view wraps a stable reference to the underlying set and supports conversion between index-based and ID-based access.

Node Views

`NodeViewByIndex(dense_index, nodes)`

View a node by its contiguous (dense) index.

`NodeViewById(node_id, nodes)`

View a node by its stable integer ID.

Constructor Parameters:

Parameter	Type	Description
`dense_index` / `node_id`	`int`	Position or ID in the node set
`nodes`	`NodeSet`	The node set (kept alive by the view)

Methods:

Method	Returns	Description
`refresh()`	—	Re-validate the view after set mutations
`is_valid()`	`bool`	Whether the view points to a live node
`id()`	`int`	Stable node ID
`dense_index()`	`int`	Contiguous dense index
`epoch()`	`int`	Set modification epoch
`nodes()`	`NodeSet`	The underlying node set (reference)
`to_by_id()`	`NodeViewById`	Convert to ID-based view
`to_by_index()`	`NodeViewByIndex`	Convert to index-based view
`properties()`	`PropertiesView`	Property accessor for this node

Cross-conversion constructors allow creating an index view from an ID view and vice versa.

Example:

```python
import phynexis
P = phynexis.parsim

# Create a node set
g = P.ComputationalGraph()
ns = g.nodes()
# ... initialize and populate ...

# Create views
```

```python
by_index = P.NodeViewByIndex(0, ns)
print("valid:", by_index.is_valid())
print("dense_index:", by_index.dense_index())

# Convert to ID-based view
by_id = P.NodeViewById(by_index.id(), ns)
print("by_id valid:", by_id.is_valid())
```

Output:

```text
valid: False
dense_index: 0
by_id valid: False
```

Edge Views

`EdgeViewByIndex(dense_index, edges)`

View an edge by its dense index.

`EdgeViewById(edge_id, edges)`

View an edge by its stable ID.

Constructor Parameters:

Parameter	Type	Description
`dense_index` / `edge_id`	`int`	Position or ID in the edge set
`edges`	`EdgeSet`	The edge set (kept alive by the view)

Methods:

Method	Returns	Description
`refresh()`	—	Re-validate after set mutations
`is_valid()`	`bool`	Whether the view is live
`edge_id()`	`int`	Stable edge ID
`dense_index()`	`int`	Contiguous dense index
`source_node_id()`	`int`	Source node ID
`target_node_id()`	`int`	Target node ID
`edge_type()`	`int`	Edge type tag
`set_edge_type(type)`	—	Set edge type
`edges()`	`EdgeSet`	The underlying edge set
`to_by_id()` / `to_by_index()`	—	Convert between access modes
`properties()`	`PropertiesView`	Property accessor

HyperEdge Views

`HyperEdgeViewByIndex(dense_index, hyper_edges)`

View a hyperedge by dense index.

`HyperEdgeViewById(hyper_edge_id, hyper_edges)`

View a hyperedge by stable ID.

Methods:

Method	Returns	Description
`refresh()`	—	Re-validate after set mutations
`is_valid()`	`bool`	Whether the view is live
`hyper_edge_id()`	`int`	Stable hyperedge ID
`dense_index()`	`int`	Contiguous dense index
`hyper_edges()`	`HyperEdgeSet`	The underlying set
`to_by_id()` / `to_by_index()`	—	Convert between access modes
`properties()`	`PropertiesView`	Property accessor
`to_json()`	`dict`	Serialize to JSON
`from_json(j)`	—	Deserialize from JSON

PropertiesView

Generic property accessor for node/edge/hyperedge views. Provides dict-style access to field values and field manager introspection.

Methods:

Method	Returns	Description
`field_manager()`	`FieldManager`	Get the underlying field manager
`dense_index()`	`int`	Current dense index
`is_valid()`	`bool`	Whether the view is valid
`to_json()`	`dict`	Serialize all properties to JSON
`from_json(j)`	—	Deserialize all properties from JSON
`print_info()`	—	Print schema information

Dict-style access:

Method	Description
`pv["name"]`	Get property value by name
`pv["name"] = value`	Set property value by name

Example:

```python
import phynexis
P = phynexis.parsim

# Access properties through node view
g = P.ComputationalGraph()
ns = g.nodes()
nvi = P.NodeViewByIndex(0, ns)
pv = nvi.properties()

print("valid:", pv.is_valid())
print("field_manager:", type(pv.field_manager()).__name__)
```

Output:

```text
valid: False
```

```
field_manager: FieldManager
```

phynexis.workflow

Workflow automation module providing a graph-based execution framework for building and running procedural pipelines. Supports node registration, method introspection, and JSON-driven graph execution.

Import

```python
import phynexis

# Bind locally
from phynexis import workflow
g = workflow.Graph()
print(type(g).__name__)
```

Module Overview

Class / Function	Description
PortInfo	Port descriptor for method signature I/O
MethodSignature	Method signature description
WorkflowObject	Abstract base class for workflow-enabled objects
Node	Node representing an object instance in a workflow graph
MethodCall	Method invocation within a node
Graph	Workflow graph managing nodes and method calls
Executor	Graph executor that processes calls in topological order
Registry	Global registry for workflow node types
`register_type()`	Register a Python class as a workflow node type
`run_graph_from_json()`	Run a workflow from a JSON string

Free Functions

Function	Description
`register_type(name, pycls)`	Register a Python class as a workflow node type
`run_graph_from_json(json_str)`	Parse JSON, build graph, execute, return outputs

PortInfo / MethodSignature

C++: `phynexis::workflow::PortInfo` / `phynexis::workflow::MethodSignature` **Python**: `phynexis.workflow.PortInfo` / `phynexis.workflow.MethodSignature` **Header**: `src/workflow/types.hpp`

Description

`PortInfo` describes a single input or output port of a workflow method. `MethodSignature` aggregates multiple port descriptors with a method name and description to form a complete method signature for introspection and GUI discovery.

PortInfo

`PortInfo()`

Default constructor — creates an empty port descriptor with all fields default-initialized.

Properties

Property	Type	Access	Description
`name`	`str`	read/write	Port name (e.g. `"position"`, `"velocity"`)
`valueType`	`str`	read/write	Value type string (e.g. `"Vec3d"`, `"double"`)
`optional`	`bool`	read/write	Whether this port is optional
`defaultValue`	`json`	read/write	Default value when optional and not connected
`description`	`str`	read/write	Human-readable port description

Example:

```python
import phynexis

# Create and configure a port descriptor
port = phynexis.workflow.PortInfo()
port.name = "position"
port.valueType = "Vec3d"
port.optional = False
port.description = "Particle position vector"

print(f"Port: {port.name} ({port.valueType})")
print(f"Optional: {port.optional}")
print(f"Description: {port.description}")
```

Output:

```text
Port: position (Vec3d)
Optional: False
Description: Particle position vector
```

MethodSignature

`MethodSignature()`

Default constructor — creates an empty method signature.

Properties

Property	Type	Access	Description
`methodName`	`str`	read/write	Method name (e.g. `"set_position"`)
`inputs`	`VecX[PortInfo]`	read/write	Input port descriptors
`outputs`	`VecX[PortInfo]`	read/write	Output port descriptors
`description`	`str`	read/write	Human-readable method description

Direct access to `inputs` and `outputs` requires a `VecX<PortInfo>` type converter that may not be registered in all build configurations. If you encounter `TypeError: Unregistered type`, the internal storage uses `VecX<PortInfo>` which needs a type caster. As a workaround, these fields can still be set through the graph API.

Example:

```python
import phynexis

# Build a method signature manually
sig = phynexis.workflow.MethodSignature()
sig.methodName = "set_position"
sig.description = "Sets the particle position"

# Create input port
input_port = phynexis.workflow.PortInfo()
input_port.name = "position"
input_port.valueType = "Vec3d"
input_port.description = "New position value"

print(f"Method: {sig.methodName}")
print(f"Description: {sig.description}")
print(f"Input port: {input_port.name} ({input_port.valueType})")
```

Output:

```text
Method: set_position
Description: Sets the particle position
Input port: position (Vec3d)
```

Deprecated Reference (phynexis v0)

The files in this directory are legacy reference documentation from the early phynexis Python API. They are preserved for historical reference but may be **out of date**.

Known issues with these documents:

- Class names may not match the actual Python API (e.g., `Voronoi` -> `Tessellation`/`Diagram`)
- Method signatures may have changed
- Examples are not validated against the current build

For material that tracks the current bindings, use the **Python API overview** and the in-site module folders **utils** and **fields**. Other `phynexis.*` submodules (for example `netdem`, `cfddem`) may ship in wheels before detailed pages exist here—use the bundled **API PDF** and your editor's introspection, plus the narrative **manual** for workflows.

ContactModel

This class represents a contact model.

Constructor

`ContactModel(label='')`
Creates a new instance of the `ContactModel` class.

- `label` (string, optional): A string label for the contact model. Default is an empty string.

Attributes

`label`
A string label for the contact model.

Methods

`SetProperty(json_data)`
Set the properties of the contact model from a JSON object.

- `json_data` (JSON object): A JSON object containing the properties to set.

`SetProperty(prop_name, prop_value)`
Set a specific property of the contact model.

- `prop_name` (string): The name of the property to set.
- `prop_value` (float): The value to set for the property.

LinearSpring

This class represents a linear spring.

Constructor

`LinearSpring()`
Creates a new instance of the LinearSpring class.

`LinearSpring(stiffness, damping, rest_length, contact_radius)`
Creates a new instance of the LinearSpring class with specific parameters.

- `stiffness` (float): The spring stiffness.
- `damping` (float): The spring damping.
- `rest_length` (float): The rest length of the spring.
- `contact_radius` (float): The contact radius of the spring.

ParallelBond

This class represents a parallel bond contact model.

Constructor

`ParallelBond()`
Creates a new instance of the ParallelBond class.

`ParallelBond(kn, kt, gamma, t0)`
Creates a new instance of the ParallelBond class with specified parameters.

- `kn` (float): Normal contact stiffness.
- `kt` (float): Tangential contact stiffness.
- `gamma` (float): Viscoelastic parameter.
- `t0` (float): Yield strength.

Inheritance

`ParallelBond` inherits from `ContactModel`

ContactSolverSettings Class

This class represents settings for the contact solver.

Attributes

`solver_type` (SolverType)
The type of contact solver to use. Default is SolverType.

`gjk_use_erosion` (bool)
Whether to use erosion for the GJK contact solver.

`gjk_erosion_ratio_initial` (float)
The initial erosion ratio for the GJK contact solver.

`gjk_erosion_ratio_increment` (float)
The increment of erosion ratio for the GJK contact solver.

`sdf_potential_type` (int)
The type of potential function to use for the SDF contact solver.

SolverType Enumeration

This enumeration represents the type of contact solver.

`SolverType.gjk`: The GJK contact solver.

`SolverType.sdf`: The SDF contact solver.

`SolverType.automatic`: The automatic contact solver.

ContactSolverFactory

This class is responsible for creating instances of contact solvers based on settings.

Attributes

settings (ContactSolverSettings): The settings for the contact solver.

Methods

init(): Constructs a new instance of the ContactSolverFactory class with default settings.

init(factory: ContactSolverFactory): Constructs a new instance of the ContactSolverFactory class with settings copied from factory.

copy(): Copies the settings of the contact solver factory to a new instance of the ContactSolverFactory class.

DEMSolver

This class represents a discrete element method (DEM) solver.

Constructor

`DEMSolver()`
Creates a new instance of the `DEMSolver` class.

Attributes

`timestep`
The timestep used for the simulation.

`contact_solver_factory`
The contact solver factory used to create contact solvers for the simulation.

Enums

`CyclePoint`
An enumeration representing the different cycle points in the simulation.

- `pre`: The pre-collision cycle point.
- `mid_0`: The first mid-collision cycle point.
- `mid_1`: The second mid-collision cycle point.
- `mid_2`: The third mid-collision cycle point.
- `mid_3`: The fourth mid-collision cycle point.
- `mid_4`: The fifth mid-collision cycle point.
- `post`: The post-collision cycle point.

CellManager

This class represents a cell manager.

Constructor

`CellManager()`
Creates a new instance of the CellManager class.

Methods

`SetSpacing(spacing)`
Set the spacing of the cell manager.

- `spacing` (float): The value of the spacing to set.

Domain

This class represents a computational domain.

Constructor

`Domain()`
Creates a new instance of the Domain class.

Attributes

`cell_manager`
An instance of the CellManager class.

`self_rank`
The rank of the current process.

`total_rank`
The total number of processes.

`bound_min`
The minimum bound of the domain.

`bound_max`
The maximum bound of the domain.

Methods

`SetBound(bound_min, bound_max)`
Set the bounds of the domain.

- `bound_min` (Vec3): The minimum bound of the domain.
- `bound_max` (Vec3): The maximum bound of the domain.

`SetCellSpacing(spacing)`
Set the spacing between cells.

- `spacing` (Vec3): The spacing between cells.

`IsJudgeDomain(particle1, particle2)`
Check if two particles belong to the same domain.

- `particle1` (Particle): The first particle.
- `particle2` (Particle): The second particle.

`IsJudgeDomain(particle, wall)`
Check if a particle and a wall belong to the same domain.

- `particle` (Particle): The particle.
- `wall` (Wall): The wall.

`IsBelongToDomain(particle)`
Check if a particle belongs to the domain.

- `particle` (Particle): The particle.

`IsBelongToDomain(particle_data)`
Check if a particle data belongs to the domain.

- `particle_data` (ParticleData): The particle data.

DomainManager

This class manages the domains in a distributed computing environment.

Constructor

`DomainManager()`
Creates a new instance of the DomainManager class.

Methods

`SetBound(bound_min, bound_max)`
Sets the bounding box of the domains.

- `bound_min` (tuple of floats): The minimum coordinates of the bounding box.
- `bound_max` (tuple of floats): The maximum coordinates of the bounding box.

`SetDecomposition(num_subdomains)`
Sets the number of subdomains for decomposition.

- `num_subdomains` (int): The number of subdomains to decompose the domain into.

`SetCellSpacing(spacing)`
Sets the cell spacing for the subdomains.

- `spacing` (float): The cell spacing for the subdomains.

`GetGhostSubDomains()`
Returns a list of ghost subdomains.

`GetSelfGhostSubDomain()`
Returns the ghost subdomain for the current process.

`GetSelfSubDomain()`
Returns the subdomain for the current process.

TetMesh

This class represents a tetrahedral mesh.

Constructor

`TetMesh()`
Creates a new instance of the TetMesh class.

`TetMesh(nodes, elements, scale)`
Creates a new instance of the TetMesh class from the given nodes and elements, with the specified scale.

- `nodes` (list of Vec3d): The list of nodes.
- `elements` (list of Vec3i): The list of elements.
- `scale` (float): The scaling factor.

Attribution

`nodes`
The list of nodes.

`elements`
The list of elements.

bound_facets
The list of boundary facets.

bound_edges
The list of boundary edges.

bound_nodes
The list of boundary nodes.

surface_nodes
The list of surface nodes.

surface_facets
The list of surface facets.

surface_node_linked_boundaries
The list of surface node linked boundaries.

Methods

GetSurfaceSTL()
Returns the surface mesh in STL format.

SaveAsVTK(filename)
Saves the mesh in VTK format to the specified file.

- `filename` (str): The name of the file to save to.

Init()
Initializes the mesh.

Membrane

This class represents a membrane in a simulation.

Constructor

__init__(radius, height)
Creates a new instance of the PyMembrane class.

- `radius` (float): The radius of the membrane.
- `height` (float): The height of the membrane.

__init__(radius, height, mesh_size)
Creates a new instance of the PyMembrane class.

- `radius` (float): The radius of the membrane.
- `height` (float): The height of the membrane.
- `mesh_size` (float): The size of the mesh.

__init__(radius, height, mesh_size, neo_k, neo_mu, density)
Creates a new instance of the PyMembrane class.

- `radius` (float): The radius of the membrane.
- `height` (float): The height of the membrane.
- `mesh_size` (float): The size of the mesh.
- `neo_k` (float): The Neo-Hookean constant of the membrane.
- `neo_mu` (float): The Neo-Hookean coefficient of the membrane.
- `density` (float): The density of the membrane.

Attributes

`radius` (float):
The radius of the membrane.

`height` (float):
The height of the membrane.

`mesh_size` (float):
The size of the mesh.

`center` (tuple of floats):
The center coordinates of the membrane.

- `neo_k` (float): The Neo-Hookean constant of the membrane.
- `neo_mu` (float): The Neo-Hookean coefficient of the membrane.
- `density` (float): The density of the membrane.
- `thickness` (float): The thickness of the membrane.
- `damp_coef` (float): The damping coefficient of the membrane.
- `timestep` (float): The timestep of the membrane.
- `nodes` (list of tuples of floats): The coordinates of the nodes in the membrane.
- `elements` (list of tuples of ints): The connectivity of the elements in the membrane.
- `elemental_stress` (list of tuples of floats): The elemental stress of the membrane.
- `nodal_vols` (list of floats): The nodal volumes of the membrane.
- `nodal_vels` (list of tuples of floats): The nodal velocities of the membrane.
- `bc_facet_pressure` (list of floats): The boundary condition pressure of the membrane.
- `bc_facet_forces` (list of tuples of floats): The boundary condition forces of the membrane.
- `bc_nodal_vels` (list of tuples of floats):

The boundary condition nodal velocities of the membrane.

Methods

`Remesh()`
Remeshes the membrane.

`SetBCNodalVelocity(nodal_vels)`
Sets the boundary condition nodal velocities of the membrane.

- `nodal_vels` (list of tuples of floats): The boundary condition nodal velocities of the membrane.

`Init()`
Initializes the membrane.

`Solve()`
Solves the membrane.

`SaveAsVTK(filename: str)`
Saves the membrane as a VTK file.

- `filename` (str): The filename of the VTK file.

ModifierManager

This class manages a collection of modifiers that can be enabled or disabled.

Constructor

`ModifierManager()`
Creates a new instance of the ModifierManager class.

Methods

`Insert(modifier)`
Inserts a modifier into the manager.

- `modifier` (Modifier): The modifier to insert.

`Enable(modifier)`
Enables a modifier.

- `modifier` (Modifier): The modifier to enable.

`Enable(name)`
Enables a modifier by name.

- `name` (string): The name of the modifier to enable.

`Disable(modifier)`
Disables a modifier.

- `modifier` (Modifier): The modifier to disable.

`Disable(name)`
Disables a modifier by name.

- `name` (string): The name of the modifier to disable.

Modifier

This class represents a modifier.

Constructor

`Modifier()`
Creates a new instance of the Modifier class.

Attributes

`label`
A string label for the modifier.

`cycle_point`
An integer representing the cycle point at which the modifier will be executed.

`sim`
A pointer to the simulation object associated with the modifier.

`scene`
A pointer to the scene object associated with the modifier.

`update_with_scene`
A boolean indicating whether the modifier should update with the scene.

`enable_logging`
A boolean indicating whether logging should be enabled for the modifier.

Methods

`Clone()`
Creates a new instance of the modifier.

`Init()`
Initializes the modifier.

`Enable()`
Enables the modifier.

`Disable()`
Disables the modifier.

`Execute()`
Executes the modifier.

`Update()`
Updates the modifier.

BreakageAnalysisPD

This class represents a breakage analysis for particles in a DEM simulation.

Constructor

`BreakageAnalysisPD()`
Creates a new instance of the BreakageAnalysisPD class.

Attributes

`particle_id_list`
A list of particle IDs to include in the breakage analysis.

`use_particles_in_scene`
A boolean indicating whether to use all particles in the scene or only the particles in the particle_id_list.

`pd_dem_coupler`
An instance of the PdDemCoupler class used to couple the breakage analysis with the DEM simulation.

Methods

`SetRootPath(root_path)`
Sets the root path for the breakage analysis output files.

- `root_path` (string): The path to the root directory for the output files.

`SetExecuteByTime(start_time, end_time, time_step)`
Sets the time range and time step for the breakage analysis execution.

- `start_time` (float): The start time for the breakage analysis.
- `end_time` (float): The end time for the breakage analysis.
- `time_step` (float): The time step for the breakage analysis.

`SetExecuteByCycles(start_cycle, end_cycle, cycle_step)`
Sets the cycle range and cycle step for the breakage analysis execution.

- `start_cycle` (int): The start cycle for the breakage analysis.
- `end_cycle` (int): The end cycle for the breakage analysis.
- `cycle_step` (int): The cycle step for the breakage analysis.

`SetParticlesFromScene()`
Sets the particles to include in the breakage analysis based on the particles in the current DEM simulation scene.

`SetParticles(particles)`
Sets the particles to include in the breakage analysis.

- `particles` (list of ints or initializer list of ints): The list of particle IDs to include in the breakage analysis.

`Cast()`
Returns a reference to the current instance of the BreakageAnalysisPD class as a Modifier object.

DataDumper

This class is responsible for dumping data from the simulation to file.

Constructor

`DataDumper()`
Creates a new instance of the DataDumper class.

Attributes

`dump_particle_info`
A boolean flag indicating whether particle information should be dumped.

`dump_wall_info`
A boolean flag indicating whether wall information should be dumped.

`dump_contact_info`
A boolean flag indicating whether contact information should be dumped.

`dump_shape_info`
A boolean flag indicating whether shape information should be dumped.

`dump_mesh`
A boolean flag indicating whether mesh information should be dumped.

`dump_reconstructed`
A boolean flag indicating whether reconstructed information should be dumped.

`time_stamp_adjustable`
A boolean flag indicating whether the time stamp is adjustable.

Methods

`SetRootPath(root_path)`
Sets the root path where the data will be dumped.

- `root_path` (string): The root path where the data will be dumped.

`SetSaveByTime(save_by_time, time_interval)`
Sets whether the data should be saved based on a time interval.

- `save_by_time` (bool): A boolean flag indicating whether the data should be saved based on a time interval.
- `time_interval` (float): The time interval for saving the data.

`SetSaveByCycles(save_by_cycles, cycle_interval)`
Sets whether the data should be saved based on a cycle interval.

- `save_by_cycles` (bool): A boolean flag indicating whether the data should be saved based on a cycle interval.
- `cycle_interval` (int): The cycle interval for saving the data.

`ClearHistories()`
Clears the history of data.

`SaveParticleInfoAsVTK(particles, file_name)`
Saves the particle information as a VTK file.

- `particles` (list): A list of particles.
- `file_name` (string): The name of the file to save.

`SaveParticleMeshAsVTK(particles, file_name)`
Saves the particle mesh information as a VTK file.

- `particles` (list): A list of particles.
- `file_name` (string): The name of the file to save.

`SaveShapeInfoAsSTL()`
Saves the shape information as an STL file.

`SaveShapeInfoAsJson()`
Saves the shape information as a JSON file.

`Print()`
Prints the current configuration of the data dumper.

`Cast()`
Casts the Modifier instance to a DataDumper instance.

MembraneWall

This class represents a wall in a membrane system.

Constructor

The MembraneWall class has multiple constructors to create a new instance of the class. Each constructor takes a different set of parameters.

`MembraneWall()`
Default constructor with no arguments.

`MembraneWall(double)`
Constructor with a single argument, the thickness of the wall.

`MembraneWall(double, double)`
Constructor with two arguments, the thickness and the bending modulus of the wall.

`MembraneWall(double, double, double)`
Constructor with three arguments, the thickness, the bending modulus, and the stretching modulus of the wall.

`MembraneWall(double, double, double, double, double, double)`
Constructor with six arguments, the thickness, the bending modulus, the stretching modulus, the repulsion strength, the friction coefficient, and the adhesion energy of the wall.

Attributes

`enable_deformation`
A boolean flag indicating whether the wall is deformable or not.

`dump_info`
A boolean flag indicating whether to dump information about the wall.

`facing_outside`
A boolean flag indicating whether the wall is facing outward or inward.

`wall_list`
A list of walls.

Methods

`SetRootPath(path)`
Set the root path for the wall.

`SetSaveByTime(interval)`
Set the time interval for saving the wall.

`SetSaveByCycles(num)`
Set the number of cycles between saving the wall.

`Init()`
Initialize the wall.

`SetDimensions(nx, ny, nz)`
Set the dimensions of the wall.

`SetPressure(pressure)`
Set the pressure on the wall.

`Cast()`
Cast the wall as a MembraneWall object.

ParticleGroup

This class represents a group of particles.

Constructor

`ParticleGroup()`
Creates a new instance of the ParticleGroup class.

Attributes

`particle_list`
A reference to the list of particles in the group.

Methods

`Add(id)`
Adds a particle to the group.

- `id` (int): The ID of the particle to add.

`Remove(id)`
Removes a particle from the group.

- `id` (int): The ID of the particle to remove.

`Add(id_list)`
Adds multiple particles to the group.

- `id_list` (list of int): A list of particle IDs to add.

`Remove(id_list)`
Removes multiple particles from the group.

- `id_list` (list of int): A list of particle IDs to remove.

`SetVelocity(v)`
Sets the velocity of all particles in the group.

- `v` (Vec3d): The velocity vector to set.

`SetSpin(w)`
Sets the spin of all particles in the group.

- `w` (Vec3d): The spin vector to set.

`Clear()`
Removes all particles from the group.

`Cast()`
Casts the Modifier object to a ParticleGroup object.

ParticleMotionControl

This class represents a modifier that controls the motion of particles.

Constructor

`ParticleMotionControl()`
Creates a new instance of the ParticleMotionControl class.

Methods

`SetFixed(fixed)`
Set the particle to be fixed.

- `fixed` (bool): A boolean value indicating whether the particle should be fixed or not.

`SetLinearVelocity(velocity)`
Set the linear velocity of the particle.

- `velocity` (Vec3d): A 3D vector representing the linear velocity of the particle.

`SetSinVelocity(velocity, frequency)`
Set the sinusoidal velocity of the particle.

- `velocity` (Vec3d): A 3D vector representing the amplitude of the sinusoidal velocity of the particle.
- `frequency` (float): The frequency of the sinusoidal velocity.

`SyncToAllProcessors()`
Synchronize the state of the particle with all processors.

`Clear()`
Clear the state of the particle.

Inheritance

This class inherits from the Modifier class.

WallGroup

This class represents a group of walls in the simulation.

Inheritance

WallGroup inherits from the Modifier class.

Constructor

`WallGroup()`
Creates a new instance of the WallGroup class.

Attributes

`wall_list`
A reference to the list of walls in the group.

Methods

`Add(wall_id)`
Add a wall to the group by ID.

- `wall_id` (int): The ID of the wall to add.

`Remove(wall_id)`
Remove a wall from the group by ID.

- `wall_id` (int): The ID of the wall to remove.

`Add(wall_ids)`
Add a list of walls to the group by their IDs.

- `wall_ids` (list of int): The IDs of the walls to add.

`Remove(wall_ids)`
Remove a list of walls from the group by their IDs.

- `wall_ids` (list of int): The IDs of the walls to remove.

`SetVelocity(vel)`
Set the velocity of all walls in the group.

- `vel` (Vec3d): The velocity to set.

`SetSpin(spin)`
Set the spin of all walls in the group.

- `spin` (Vec3d): The spin to set.

`Clear()`
Remove all walls from the group.

`Cast()`
Cast the WallGroup object to a Modifier object.

WallDispControl

This class represents a modifier that controls the displacement of walls.

Constructor

`WallDispControl()`
Creates a new instance of the WallDispControl class.

Attributes

`vel`
The velocity of the walls.

spin
The spin of the walls.

Methods

SetVelocity(vel)
Sets the velocity of the walls.

- `vel` (Vec3d): The velocity to set.

SetSpin(spin)
Sets the spin of the walls.

- `spin` (Vec3d): The spin to set.

SetWalls(wall_ids)
Sets the walls that this modifier will act on.

`wall_ids` (`VecXT<int>` or list of int): The IDs of the walls to act on.

Cast()
Casts the Modifier base class to a WallDispControl object.

__init__()
The Python constructor for the WallDispControl class.

WallMotionIntegrator

This class represents a wall motion integrator.

Constructor

WallMotionIntegrator()
Creates a new instance of the WallMotionIntegrator class.

Attributes

mass
The mass of the wall.

moi_principal
The principal moments of inertia of the wall.

enable_translation
A boolean indicating whether or not translation of the wall is enabled.

enable_rotation
A boolean indicating whether or not rotation of the wall is enabled.

Methods

Cast()
Returns a pointer to the WallMotionIntegrator object.

SetMass(mass)
Sets the mass of the wall.

- `mass` (float): The mass to set for the wall.

`SetMomentOfInertia(principal_moments)`
Sets the principal moments of inertia of the wall.

- `principal_moments` (Vec3d): A 3D vector containing the principal moments of inertia.

`SetTranslationEnabled(enabled)`
Sets whether or not translation of the wall is enabled.

- `enabled` (bool): Whether or not translation of the wall is enabled.

`SetRotationEnabled(enabled)`
Sets whether or not rotation of the wall is enabled.

- `enabled` (bool): Whether or not rotation of the wall is enabled.

WallServoControl

This class represents a wall servo control.

Constructor

`WallServoControl(kn, area)`
Creates a new instance of the WallServoControl class.

- `kn` (float): The spring constant.
- `area` (float): The area of the wall.

Attributes

`kn`
The spring constant of the wall servo control.

`area`
The area of the wall servo control.

`target_pressure`
The target pressure of the wall servo control.

`vel_max`
The maximum velocity of the wall servo control.

`study_rate`
The study rate of the wall servo control.

`tol`
The tolerance of the wall servo control.

`enable_warning`
The flag to enable warnings of the wall servo control.

`enable_auto_area`
The flag to enable automatic area of the wall servo control.

`achieved`
The achieved flag of the wall servo control.

Methods

`SetWalls(walls)`
Sets the walls for the wall servo control.

- `walls` (list or array of integers): The walls to set.

`AddWall(wall)`
Adds a wall to the wall servo control.

- `wall` (integer): The wall to add.

`Cast()`
Casts the modifier to a WallServoControl object.

MPIManager

This class provides an interface for MPI-based parallelism.

Constructor

`MPIManager()`
Creates a new instance of the MPIManager class.

Methods

`GetSelfRank()`
Returns the rank of the current process.

`GetTotalRank()`
Returns the total number of processes.

`SyncShapeToAllProcessors(shape)`
Synchronizes the shape of an array among all the processes.

- `shape` (numpy array): A numpy array that represents the shape to synchronize.

`SyncDataAmongProcessors(data)`
Synchronizes the data of an array among all the processes.

- `data` (numpy array): A numpy array that represents the data to synchronize.

Note: This method can be called with either an array of integers or an array of doubles.

PyNetDEM

PyNetDEM is a Python interface for NetDEM.

Modules

PyNetDEM is composed of several modules that can be imported separately.

- `pynetdem.utils`: Contains utility functions used across all other modules.
- `pynetdem.fem`: Contains functions related to Finite Element Method.
- `pynetdem.dem`: Contains functions related to Discrete Element Method.
- `pynetdem.domain`: Contains functions related to domain creation and modification.
- `pynetdem.peridigm`: Contains functions related to PeriDEM.
- `pynetdem.shape`: Contains functions related to shape creation and modification.
- `pynetdem.scene`: Contains functions related to scene creation and modification.
- `pynetdem.modifier`: Contains functions related to modifier creation and modification.
- `pynetdem.mpi`: Contains functions related to MPI management.
- `pynetdem.simulation`: Contains functions related to simulation creation and management.

Peridigm

DomainSplittor

This class represents a domain splittor.

Constructor

`DomainSplittor()`
Creates a new instance of the DomainSplittor class.

Methods

`InitFromSTL(filename, num_parts)`
Initializes the domain splittor from an STL file.

- `filename` (string): The name of the STL file to read.
- `num_parts` (int): The number of parts to split the domain into.

LevelSetSplittor

This class represents a level set splittor, which is a type of domain splittor.

Constructor

`LevelSetSplittor()`
Creates a new instance of the LevelSetSplittor class.

Methods

`InitFromDistanceMap(filename)`
Initializes the level set splittor from a distance map file.

- `filename` (string): The name of the distance map file to read.

`InitFromDistanceMap (x_min, x_max, y_min, y_max, z_min, z_max, num_parts, distance_map)`
Initializes the level set splittor from a distance map.

- `x_min` (double): The minimum x coordinate of the domain.
- `x_max` (double): The maximum x coordinate of the domain.
- `y_min` (double): The minimum y coordinate of the domain.
- `y_max` (double): The maximum y coordinate of the domain.
- `z_min` (double): The minimum z coordinate of the domain.
- `z_max` (double): The maximum z coordinate of the domain.
- `num_parts` (int): The number of parts to split the domain into.
- `distance_map` (`VecXT<double>`): A vector containing the distance map data.

`InitFromSTL(stl_model, num_parts)`
Initializes the level set splittor from an STL model.

- `stl_model` (STLModel): The STL model to initialize from.
- `num_parts` (int): The number of parts to split the domain into.

`GetPeriDigmNodes()`
Returns the Peridigm node data for the domain.

`MakePorosity(radius)`
Adds porosity to the domain.

- `radius` (double): The radius of the porosity.

`GetSTLModel()`
Returns the STL model for the domain.

`GetSTLModel(part_ids)`
Returns the STL model for a specific set of domain parts.

- `part_ids` (`VecXT<int>`): A vector containing the IDs of the domain parts to include in the model.

TetMeshSplittor

This class represents a tetrahedral mesh splittor.

Constructor

`TetMeshSplittor()`
Creates a new instance of the TetMeshSplittor class.

Methods

`InitFromSTL(stl_model, num_partitions)`
Initialize the tetrahedral mesh splittor from an STL model.

- `stl_model` (STLModel): The STL model to use for initialization.
- `num_partitions` (int): The number of partitions to split the mesh into.

`GetPeriDigmNodes()`
Get the Peridigm nodes.

`MakePorosity()`
Make porosity in the mesh.

`GetSTLModel()`
Get the STL model.

Returns a `VecXT<int>` object representing the STL model.

`GetSTLModel(node_id)`
Get the STL model.

- `node_id` (`VecXT<int>`): The node ID to get the STL model for.

Returns a `VecXT<int>` object representing the STL model for the specified node ID.

PeriDigmDiscretization

This class represents a discretization of a PeriDigm model.

Constructor

`PeriDigmDiscretization()`
Creates a new instance of the PeriDigmDiscretization class.

Attributes

`type`
An enumeration that specifies the type of the discretization. Possible values are:

level_set
a level set discretization

tetmesh
a tetrahedral mesh discretization

domain_splittor
The domain splittor used to split the domain into blocks.

nodes
The nodes of the discretization.

node_block_indices
The block indices of the nodes.

node_vols
The volumes of the nodes.

Methods

InitFromSTL(stl_file_path, num_partitions)
Initialize the discretization from an STL file.

- `stl_file_path` (string): The path to the STL file.
- `num_partitions` (int): The number of partitions to split the domain into.

InitFromSTL(stl_model, num_partitions)
Initialize the discretization from an STL model.

- `stl_model` (STLModel): The STL model to use.
- `num_partitions` (int): The number of partitions to split the domain into.

InitFromDistanceMap(distance_map_file_path, num_partitions)
Initialize the discretization from a distance map file.

- `distance_map_file_path` (string): The path to the distance map file.
- `num_partitions` (int): The number of partitions to split the domain into.

InitFromGrid(grid_file_path, num_partitions)
Initialize the discretization from a grid file.

- `grid_file_path` (string): The path to the grid file.
- `num_partitions` (int): The number of partitions to split the domain into.

MakePorosity(porosity)
Add porosity to the discretization.

- `porosity` (float): The porosity to add.

WriteInputFile(node_file_path)
Write the input file of the discretization.

- `node_file_path` (string): The path to write the node file.

GetNodeSize()
Get the size of the nodes in the discretization.

PeriDigmDiscretization

This class represents a PeriDigm discretization.

Constructor

`PeriDigmDiscretization()`
Creates a new instance of the PeriDigmDiscretization class.

Enum

`Type`
An enum class for different types of PeriDigm discretization. The available values are:

- `level_set`
- `tetmesh`

Attributes

`type`
A Type value representing the type of PeriDigm discretization.

`domain_splittor`
The domain splitter for the PeriDigm discretization.

`nodes`
A vector of Node objects representing the nodes of the PeriDigm discretization.

`node_block_indices`
A vector of int values representing the block indices of the nodes.

`node_vols`
A vector of double values representing the volumes of the nodes.

Methods

`InitFromSTL(filename, numBlocks)`
Initialize the PeriDigm discretization from an STL file.

- `filename` (string): The filename of the STL file to load.
- `numBlocks` (int): The number of blocks to create.

`InitFromSTL(stl_model, numBlocks)`
Initialize the PeriDigm discretization from an STL model.

- `stl_model` (STLModel): The STL model to use for initialization.
- `numBlocks` (int): The number of blocks to create.

`InitFromDistanceMap(distanceMap, numBlocks)`
Initialize the PeriDigm discretization from a distance map.

- `distanceMap` (DistanceMap): The distance map to use for initialization.
- `numBlocks` (int): The number of blocks to create.

`InitFromGrid(grid, numBlocks)`
Initialize the PeriDigm discretization from a grid.

- `grid` (Grid): The grid to use for initialization.
- `numBlocks` (int): The number of blocks to create.

`MakePorosity(porosity)`
Set the porosity of the PeriDigm discretization.

- `porosity` (double): The porosity to set.

`WriteInputFile(filename)`
Write the PeriDigm discretization to a file.

- `filename` (string): The filename to write to.

`GetNodeSize()`
Return the size of the nodes in the PeriDigm discretization.

PeriDigmMaterial

This class represents a material model in PeriDigm.

Constructor

`PeriDigmMaterial()`
Creates a new instance of the PeriDigmMaterial class.

Attributes

`type`
An enum value that represents the type of material. The possible values are:

- `Elastic`

`density`
The density of the material.

`youngs_modulus`
The Young's modulus of the material.

`poissons_ratio`
The Poisson's ratio of the material.

Methods

`WriteInputFile(filename)`
Writes the material properties to an input file.

- `filename` (string): The name of the file to write to.

PeriDigmDamageModel

This class represents a damage model in the PeriDigm library.

Constructor

`PeriDigmDamageModel()`
Creates a new instance of the PeriDigmDamageModel class.

Attributes

`type`
A string representing the type of the damage model.

`critical_stretch`
A double representing the critical stretch for the damage model.

Methods

`InitFromEnergyReleaseRate(energy_release_rate)`
Initializes the damage model based on the energy release rate.

- `energy_release_rate` (double): The energy release rate.

`GetStretchFromEnergyReleaseRate(energy_release_rate)`
Returns the stretch based on the energy release rate.

- `energy_release_rate` (double): The energy release rate.

`WriteInputFile(file_name)`
Writes the input file for the damage model.

- `file_name` (string): The name of the input file to write.

PeriDigmBlock

This class represents a block in a PeriDigm simulation.

Constructor

`PeriDigmBlock()`
Creates a new instance of the PeriDigmBlock class.

Attributes

`node_indices`
A list of the node indices that make up the block.

`material_id`
An integer ID for the material that the block is composed of.

`damage_model_id`
An integer ID for the damage model used to simulate damage in the block.

`horizon`
The horizon used in the simulation.

Methods

`WriteInputFile(file_name)`
Writes the block information to an input file.

- `file_name` (string): The name of the file to write to.

PeriDigmBoundaryCondition

This class represents a boundary condition in the PeriDigm code.

Constructor

`PeriDigmBoundaryCondition()`
Creates a new instance of the PeriDigmBoundaryCondition class.

Attributes

`type`
An enum that specifies the type of boundary condition. The possible values are:

`Prescribed_Displacement`
The boundary condition is specified by a prescribed displacement.

`Body_Force`
The boundary condition is specified by a body force.

`node_indices`
A list of integers that specify the indices of the nodes that the boundary condition applies to.

`dim_activated`
A list of integers that specify the dimensions that the boundary condition is activated in. The possible values are 0, 1, and 2, which correspond to the x, y, and z directions, respectively.

`disp_rate`
A float that specifies the rate of displacement for the boundary condition.

`loading_rate`
A float that specifies the rate of loading for the boundary condition.

`disp`
A float that specifies the displacement for the boundary condition.

`loading`
A float that specifies the loading for the boundary condition.

`mech_time`
A float that specifies the mechanical time for the boundary condition.

Methods

`InsertNode(node_index)`
Insert a node into the list of nodes that the boundary condition applies to.

- `node_index` (int): The index of the node to insert.

`SetActivatedDimensions(dimensions)`
Set the dimensions that the boundary condition is activated in.

- `dimensions` (list of int): A list of integers that specify the dimensions that the boundary condition is activated in.

`SetByDisplacementRate(rate)`
Set the boundary condition by a prescribed displacement rate.

- `rate` (float): The rate of displacement to use.

`SetByLoadingRate(rate)`
Set the boundary condition by a prescribed loading rate.

- `rate` (float): The rate of loading to use.

`SetByUltimateDisplacement(displacement)`
Set the boundary condition by a prescribed ultimate displacement.

- `displacement` (float): The ultimate displacement to use.

`SetByUltimateLoading(loading)`
Set the boundary condition by a prescribed ultimate loading.

- `loading` (float): The ultimate loading to use.

`WriteInputFile(filename)`
Write the boundary condition to an input file.

- `filename` (string): The name of the file to write to.

PeriDigmSettings

This class represents the settings for the PeriDigm code.

Constructor

`PeriDigmSettings()`
Creates a new instance of the PeriDigmSettings class.

Attributes

`result_dir`
A string representing the directory where the simulation results are stored.

`peridigm_exe`
A string representing the path to the PeriDigm executable.

`horizon_factor`
A float representing the factor used to determine the horizon for each particle.

`use_auto_timestep`
A boolean indicating whether to use an automatic time step or not.

`timestep`
A float representing the time step size to be used.

`timestep_factor`
A float representing the factor used to determine the time step size for each particle.

`mech_time`
A float representing the total mechanical time of the simulation.

`loading_radius_factor`
A float representing the factor used to determine the loading radius for each particle.

`constrain_radius_factor`
A float representing the factor used to determine the constrain radius for each particle.

`output_freqency`
An integer representing the frequency of output data.

Methods

`WriteInputFile(input_file_path)`
Writes the input file for the PeriDigm code.

- `input_file_path` (string): The path to the input file to write.

DEMFragment

This class represents a discrete element method (DEM) fragment.

Constructor

`DEMFragment()`
Creates a new instance of the DEMFragment class.

Attributes

`shape_type`
An integer that represents the shape type of the fragment.

`sphere_size`
A float that represents the sphere size of the fragment.

`stl_model`
A STLModel object that represents the STL model of the fragment.

`vel`
A Vec3d object that represents the velocity of the fragment.

`spin`
A Vec3d object that represents the spin of the fragment.

Methods

InitLevelSet(level_set)

Applies a boundary force to the fragment based on the given level set.

- `level_set` (LevelSet): A level set that represents the boundary.

`ResolverOverlap(other_frag)`
Resolves the overlap between this fragment and another fragment.

- `other_frag` (DEMFragment): The other fragment to resolve the overlap with.

`ReInitSTLModel()`
Re-initializes the STL model of the fragment.

ParticleStrengthParameters

This class represents the strength parameters of a particle.

Constructor

`ParticleStrengthParameters()`
Creates a new instance of the ParticleStrengthParameters class.

Attributes

`ref_size`
The reference size of the particle.

`ref_energy_release_rate`
The reference energy release rate of the particle.

`weibull_modulus`
The Weibull modulus of the particle.

`weibull_coef_a`
The Weibull coefficient A of the particle.

weibull_coef_b
The Weibull coefficient B of the particle.

min_breakable_size
The minimum breakable size of the particle.

Methods

GetEnergyReleaseRate(size)
Calculates the energy release rate of the particle for a given size.

- `size` (float): The size of the particle.

GetEnergyReleaseRate(size, strength)
Calculates the energy release rate of the particle for a given size and strength.

- `size` (float): The size of the particle.
- `strength` (float): The strength of the particle.

PeriDigmSimulator

This class represents a PeriDigm simulator.

Constructor

PeriDigmSimulator()
Creates a new instance of the PeriDigmSimulator class.

Attributes

discretization
The discretization used in the simulation.

materials
The materials used in the simulation.

damage_models
The damage models used in the simulation.

blocks
The blocks used in the simulation.

boundary_conditions
The boundary conditions used in the simulation.

settings
The settings used in the simulation.

Methods

Clear()
Clears all data associated with the simulator.

InsertMaterial(material)
Inserts a material into the simulator.

- `material` (Material): The material to insert.

InsertDamageModel(damage_model)
Inserts a damage model into the simulator.

- `damage_model` (DamageModel): The damage model to insert.

`InsertBlock(block)`
Inserts a block into the simulator.

- `block` (Block): The block to insert.

`InsertBoundaryCondition(boundary_condition)`
Inserts a boundary condition into the simulator.

- `boundary_condition` (BoundaryCondition): The boundary condition to insert.

`InitDefaultSetup()`
Initializes the simulator with default settings.

`InitAutoTimestep()`
Initializes the simulator with automatic timestep settings.

`WriteNodeFile()`
Writes the node file for the simulator.

`WriteNodeSetFile()`
Writes the node set file for the simulator.

`WriteInputFile()`
Writes the input file for the simulator.

`Solve()`
Runs the simulation.

`SetUpResultDirectory()`
Sets up the result directory for the simulation.

`CleanUpResultDirectory()`
Cleans up the result directory for the simulation.

PeriDigmDEMCoupler

This class represents a coupler between the Peridynamics and DEM models.

Constructor

`PeriDigmDEMCoupler()`
Creates a new instance of the PeriDigmDEMCoupler class.

Attributes

`particle`
The Peridynamics particle system.

`pd_sim`
The Peridynamics simulation.

`base_dir`
The base directory for output files.

`sub_dir_index`
The sub-directory index for output files.

`mesh_res`
The resolution of the DEM mesh.

node_size_ave
The average size of the DEM mesh nodes.

mech_time
The mechanical time step size.

surface_stl
The path to the STL file representing the surface of the simulation domain.

boundary_force_nodes
The nodes on the boundary where forces are applied.

boundary_force_node_vols
The volumes associated with the boundary force nodes.

boundary_force_values
The values of the boundary forces.

unbalanced_force_nodes
The nodes where unbalanced forces are applied.

unbalanced_force_values
The values of the unbalanced forces.

use_customized_loading_rate
A flag indicating whether to use a customized loading rate.

loading_rate
The loading rate.

loading_steps
The number of loading steps.

is_broken
A flag indicating whether the material is broken.

damage_fraction_limit
The damage fraction limit.

fragment_vol_limit
The fragment volume limit.

ignore_fines
A flag indicating whether to ignore fines.

use_alpha_shape
A flag indicating whether to use alpha shape.

fragment_alpha
The alpha value for fragmenting.

strength_params
The strength parameters.

material_params
The material parameters.

Methods

Init()
Initializes the DEM coupler.

ApplyBoundaryForce()
Applies the boundary forces.

`Solve()`
Solves the DEM coupler.

`CheckBreakage()`
Checks for breakage.

`GetFragments()`
Gets the fragments.

DEMObjectPool

This class represents a pool of DEM objects.

Constructor

The DEMObjectPool class does not have a constructor.

Methods

`init()`
Initialize the DEMObjectPool.

`GetParticle()`
Get a reference to a particle object in the pool.

- Returns: A reference to a Particle object.

`GetContactPP()`
Get a reference to a ContactPP object in the pool.

- Returns: A reference to a ContactPP object.

`GetContactPW()`
Get a reference to a ContactPW object in the pool.

- Returns: A reference to a ContactPW object.

`Clone(contact_pp)`
Create a copy of a ContactPP object in the pool.

`contact_pp` (ContactPP const*): A pointer to the ContactPP object to clone.

- Returns: A reference to a new ContactPP object.

`Clone(contact_pw)`
Create a copy of a ContactPW object in the pool.

`contact_pw` (ContactPW const*): A pointer to the ContactPW object to clone.

- Returns:A reference to a new ContactPW object.

Scene

This class represents a simulation scene.

Constructor

`Scene()`
Creates a new instance of the Scene class.

Attributes

gravity_coef
A float representing the gravitational acceleration coefficient.

contact_model_map
A dictionary mapping contact model labels to their corresponding ContactModel objects.

bond_model_table
A list of bond model names.

collision_model_table
A list of collision model names.

particle_list
A list of Particle objects.

particle_proxy_list
A list of ParticleProxy objects.

particle_ghost_list
A list of ParticleGhost objects.

wall_list
A list of Wall objects.

wall_ghost_list
A list of WallGhost objects.

particle_map
A dictionary mapping particle IDs to their corresponding Particle objects.

shape_map
A dictionary mapping shape IDs to their corresponding Shape objects.

local_shape_list
A list of local shape IDs.

Methods

InsertContactModel(contact_model)
Inserts a contact model into the scene.

- `contact_model` (ContactModel): The contact model to insert.

SetNumberOfMaterials(num_materials)
Sets the number of materials in the scene.

- `num_materials` (int): The number of materials.

SetCollisionModel(i, j, contact_model)
Sets the collision model between two materials.

- `i` (int): The index of the first material.
- `j` (int): The index of the second material.
- `contact_model` (ContactModel): The contact model to use for the collision.

InsertShape(shape)
Inserts a shape into the scene.

- `shape` (Shape): The shape to insert.

InsertShape(shapes)
Inserts multiple shapes into the scene.

- `shapes` (list of Shape): The shapes to insert.

InsertParticle(particle)
Inserts a particle into the scene.

- `particle` (Particle): The particle to insert.

InsertParticle(particles)
Inserts multiple particles into the scene.

- `particles` (list of Particle): The particles to insert.

InsertParticle(bonded_spheres)
Inserts a bonded sphere into the scene.

- `bonded_spheres` (BondedSpheres): The bonded sphere to insert.

InsertParticle(bonded_spheres_list)
Inserts multiple bonded spheres into the scene.

- `bonded_spheres_list` (list of BondedSpheres): The bonded spheres to insert.

InsertParticle(bonded_voronois)
Inserts a bonded Voronoi into the scene.

- `bonded_voronois` (BondedVoronois): The bonded Voronoi to insert.

InsertParticle(bonded_voronois_list)
Inserts multiple bonded Voronois into the scene.

- `bonded_voronois_list` (list of BondedVoronois): The bonded Voronois to insert.

InsertWall(wall)
Inserts a wall into the scene.

- `wall` (Wall): The wall to insert.

InsertWall(walls)
Inserts multiple walls into the scene.

- `walls` (list of Wall): The walls to insert.

RemoveShape(shape)
Removes a shape from the scene.

- `shape` (Shape): The shape to remove.

RemoveParticle(particle)
Removes a particle from the scene.

- `particle` (Particle): The particle to remove.

RemoveParticle(index)
Removes a particle from the scene by its index.

- `index` (int): The index of the particle to remove.

RemoveWall(wall)
Removes a wall from the scene.

- `wall` (Wall): The wall to remove.

RemoveWall(index)
Removes a wall from the scene by its index.

- `index` (int): The index of the wall to remove.

InsertContactModel(contact_model)
Inserts a contact model into the scene.

- `contact_model` (ContactModel): The contact model to insert.

`GetShapes()`

Returns a reference to the vector of shapes in the scene.

`InScene(shape)`

Checks if a shape is in the scene.

- `shape` (Shape): The shape to check.

`InScene(contact_model)`

Checks if a contact model is in the scene.

- `contact_model` (ContactModel): The contact model to check.

`SetNumberOfMaterials(num_materials)`

Sets the number of materials in the scene.

- `num_materials` (int): The number of materials.

`SetBondModel(i, j, contact_model)`

Sets the bond model between two particles.

- `i` (int): The index of the first particle.
- `j` (int): The index of the second particle.
- `contact_model` (ContactModel): The contact model to use for the bond.

`SetBondModel(i, j, contact_model_label)`

Sets the bond model between two particles.

- `i` (int): The index of the first particle.
- `j` (int): The index of the second particle.
- `contact_model_label` (str): The label of the contact model to use for the bond.

`SetCollisionModel(i, j, contact_model)`

Sets the collision model between two particles.

- `i` (int): The index of the first particle.
- `j` (int): The index of the second particle.
- `contact_model` (ContactModel): The contact model to use for the collision.

`SetCollisionModel(i, j, contact_model_label)`

Sets the collision model between two particles.

- `i` (int): The index of the first particle.
- `j` (int): The index of the second particle.
- `contact_model_label` (str): The label of the contact model to use for the collision.

`SetGravity(gravity)`

Sets the gravity vector for the scene.

- `gravity` (Vec3): The gravity vector.

`GetBondModel(particle1, particle2)`

Gets the bond model between two particles.

- `particle1` (Particle): The first particle.
- `particle2` (Particle): The second particle.

`GetBondModel(particle, wall)`

Gets the bond model between a particle and a wall.

- `particle` (Particle): The particle.
- `wall` (Wall): The wall.

`GetCollisionModel(particle1, particle2)`

Returns a reference to the collision model between two particles.

- `particle1` (Particle): The first particle.
- `particle2` (Particle): The second particle.

`GetCollisionModel(particle, wall)`
Returns a reference to the collision model between a particle and a wall.

- `particle` (Particle): The particle.
- `wall` (Wall): The wall.

`AutoReadRestart()`
Automatically reads restart files and sets up the simulation accordingly.

`ReadRestartShapes()`
Reads the shape data from a restart file.

`ReadRestartParticles()`
Reads the particle data from a restart file.

`ReadRestartWalls()`
Reads the wall data from a restart file.

`ReadRestartContacts()`
Reads the contact data from a restart file.

`GetContactPPs()`
Returns a list of all particle-particle contacts in the simulation.

`GetContactPWs()`
Returns a list of all particle-wall contacts in the simulation.

`ClearShapes()`
Removes all shapes from the scene.

`ClearParticles()`
Removes all particles from the scene.

`ClearWalls()`
Removes all walls from the scene.

`ClearContactModels()`
Removes all contact models from the scene.

`ClearContacts()`
Removes all contacts from the scene.

`FindParticle(particle_id)`
Finds a particle in the simulation by its ID.

- `particle_id` (int): The ID of the particle.

`FindWall(wall_id)`
Finds a wall in the simulation by its ID.

- `wall_id` (int): The ID of the wall.

PackGenerator

This class represents a pack generator.

Constructor

`PackGenerator()`
Creates a new instance of the PackGenerator class.

Static Methods

`GetGridPack()`
Generates a pack using a grid-based algorithm.

GetGridPack(xmin, xmax, ymin, ymax, zmin, zmax, nx, ny, nz, shapes): Generates a pack using a grid-based algorithm with the given dimensions and number of cells in each direction, using the provided shapes.

GetGridPack(xmin, xmax, ymin, ymax, zmin, zmax, nx, ny, nz, shape_vec): Generates a pack using a grid-based algorithm with the given dimensions and number of cells in each direction, using the provided vector of shapes.

GetGridPack(xmin, xmax, ymin, ymax, zmin, zmax, nx, ny, nz, bonded_spheres): Generates a pack using a grid-based algorithm with the given dimensions and number of cells in each direction, using the provided bonded spheres.

GetGridPack(xmin, xmax, ymin, ymax, zmin, zmax, nx, ny, nz, bonded_voronois): Generates a pack using a grid-based algorithm with the given dimensions and number of cells in each direction, using the provided bonded Voronois.

GetVoronoiPack Generates a pack using a Voronoi-based algorithm.

GetVoronoiPack(xmin, xmax, ymin, ymax, zmin, zmax, n, shapes): Generates a pack using a Voronoi-based algorithm with the given dimensions and number of cells in each direction, using the provided shapes.

GetVoronoiPack(xmin, xmax, ymin, ymax, zmin, zmax, n, shape_vec): Generates a pack using a Voronoi-based algorithm with the given dimensions and number of cells in each direction, using the provided vector of shapes.

GetVoronoiPack(stl_model, n, shape): Generates a pack using a Voronoi-based algorithm with the given STL model and number of cells in each direction, using the provided shape.

GetVoronoiPack(stl_model, n, shape_vec): Generates a pack using a Voronoi-based algorithm with the given STL model and number of cells in each direction, using the provided vector of shapes.

Particle

This class represents a particle in a simulation.

Constructor

`Particle()`
Creates a new instance of the Particle class.

Attributes

`id`
The ID of the particle.

`shape`
A reference to the shape of the particle.

`bound_min`
The minimum bound of the particle.

`bound_max`
The maximum bound of the particle.

`margin`
The margin of the particle.

`bound_disp`
The displacement of the particle.

`material_type`
The type of material of the particle.

density
The density of the particle.

mass
The mass of the particle.

moi_principal
The moment of inertia of the particle.

damp_global
The global damping of the particle.

pos
The position of the particle.

quaternion
The quaternion of the particle.

vel
The velocity of the particle.

spin
The spin of the particle.

vel_m0p5
The velocity of the particle divided by 0.5.

spin_principal
The principal spin of the particle.

force
The force acting on the particle.

moment
The moment acting on the particle.

dynamic_properties
The dynamic properties of the particle.

enable_rotation
Whether the particle is allowed to rotate.

enable_bound_aabb
Whether the particle is bound by an AABB.

need_update_linked_list
Whether the particle's linked list needs to be updated.

linked_cell_list
The linked cell list of the particle.

linked_particle_list
The linked particle list of the particle.

linked_wall_list
The linked wall list of the particle.

contact_pp_lookup_table
The lookup table for particle-particle contact.

contact_pw_lookup_table
The lookup table for particle-wall contact.

is_on_edge
Whether the particle is on an edge.

need_send_out
Whether the particle needs to be sent out.

linked_domain_list
The linked domain list of the particle.

need_update_stl_model
Whether the particle's STL model needs to be updated.

stl_model
The STL model of the particle.

Wall

This class represents a wall object.

Constructor

Wall()
Creates a new instance of the Wall class.

Wall(shape)
Creates a new instance of the Wall class with a specified shape.

- `shape` (Shape): The shape of the wall.

Attributes

id
An integer ID for the wall.

label
A string label for the wall.

shape
The shape of the wall.

material_type
The material type of the wall.

enable_rotation
A boolean indicating whether the wall is allowed to rotate.

enable_bound_aabb
A boolean indicating whether the wall has an axis-aligned bounding box.

bound_min
The minimum position of the bounding box.

bound_max
The maximum position of the bounding box.

bound_disp
The displacement of the bounding box.

pos
The position of the wall.

quaternion
The quaternion orientation of the wall.

force
The force acting on the wall.

moment
The moment acting on the wall.

vel
The velocity of the wall.

spin
The spin of the wall.

vel_spin
The velocity spin of the wall.

dynamic_properties
A map containing the dynamic properties of the wall.

need_update_linked_list
A boolean indicating whether the wall's linked list needs to be updated.

linked_cell_list
A list of linked cells.

linked_particle_list
A list of linked particles.

contact_pw_lookup_table
A lookup table for pairwise contacts.

need_update_stl_model
A boolean indicating whether the wall's STL model needs to be updated.

stl_model
The STL model of the wall.

Methods

SetShape(shape)
Sets the shape of the wall.

- `shape` (Shape): The shape of the wall.

SetPosition(pos)
Sets the position of the wall.

- `pos` (Vec3d): The position of the wall.

SetRodrigues(rot)
Sets the Rodrigues rotation of the wall.

- `rot` (Vec3d): The Rodrigues rotation of the wall.

SetQuaternion(quat)
Sets the quaternion orientation of the wall.

- `quat` (Vec4d): The quaternion orientation of the wall.

SetVelocity(vel)
Sets the velocity of the wall.

- `vel` (Vec3d): The velocity of the wall.

SetSpin(spin)
Sets the spin of the wall.

- `spin` (Vec3d): The spin of the wall.

SetVelocitySpin(vel, spin)

Sets the velocity and spin of the wall.

- `vel` (Vec3d): The velocity of the wall.
- `spin` (Vec3d): The spin of the wall.

GetVelocity()

Return: the velocity of the wall.

SetDynamicProperty(prop_name, prop_value)

Sets a dynamic property of the wall.

- `prop_name` (string): The name of the dynamic property to set.
- `prop_value` (float): The value to set for the dynamic property.

GetDynamicProperty(prop_name)

Returns the value of a dynamic property of the wall.

- `prop_name` (string): The name of the dynamic property.

AddForce(force)

Adds a force to the wall.

- `force` (Vec3d): The force to add.

AddMoment(moment)

Adds a moment to the wall.

- `moment` (Vec3d): The moment to add.

AddForce(f)

Add a force to the wall.

- `f` (Vec3d): The force to add.

AddMoment(m)

Add a moment to the wall.

- `m` (Vec3d): The moment to add.

AddForceAtomic(f)

Add an atomic force to the wall.

- `f` (Vec3d): The atomic force to add.

AddMomentAtomic(m)

Add an atomic moment to the wall.

- `m` (Vec3d): The atomic moment to add.

ClearForce()

Clear the total force on the wall.

ClearMoment()

Clear the total moment on the wall.

ApplyContactForce(contact_pw, contact_force)

Apply a contact force to the wall.

- `contact_pw` : The contact point on the wall.
- `contact_force` : The contact force to apply.

UpdateContactForce(contact_pw, contact_force)

Update the contact force on the wall.

- `contact_pw` (Vec3d): The contact point on the wall.
- `contact_force` (Vec3d): The new contact force.

UpdateMotion()

Update the motion of the wall.

- `t` (double, optional): The time step. Default is 0.

`UpdateMotion(pos, quat, t)`
Update the motion of the wall.

- `pos` (Vec3d): The position of the wall.
- `quat` (Vec4d): The quaternion of the wall.
- `t` (double, optional): The time step. Default is 0.

`UpdateMotion(pos, euler, t)`
Update the motion of the wall.

- `pos` (Vec3d): The position of the wall.
- `euler` (Vec3d): The Euler angles of the wall.
- `t` (double, optional): The time step. Default is 0.

`UpdateBound()`
Update the bounding box of the wall.

`ClearLinkedCells()`
Clear the linked cells of the wall.

`ClearLinkedNeighs()`
Clear the linked neighbors of the wall.

`BuildContactLookUpTable()`
Build the contact lookup table of the wall.

`ClearContactLookUpTable()`
Clear the contact lookup table of the wall.

`UpdateLinkedCells()`
Update the linked cells of the wall.

`UpdateLinkedNeighs()`
Update the linked neighbors of the wall.

`GetContactPWs()`
Get the contact points and weights of the wall.

`UpdateSTLModel()`
Update the STL model of the wall.

`SaveAsVTK(file_name)`
Save the wall as a VTK file.

- `file_name` (string): The name of the file to save.

`Print()`
Print the wall.

`FindLinked()`
Find the linked objects of the wall.

`FindContactRef(contact_pw)`
Find the contact reference object of the wall.

WallBoxPlane

Constructor

`WallBoxPlane(x, y, z, w, h, d)`

Creates a new instance of the WallBoxPlane class with the following parameters:

- `x` (float): x coordinate of the center of the box
- `y` (float): y coordinate of the center of the box
- `z` (float): z coordinate of the center of the box
- `w` (float): width of the box
- `h` (float): height of the box
- `d` (float): depth of the box

Methods

`GetShapes()`

Returns a list of shape objects of the wall.

`GetWalls()`

Returns a list of wall objects of the wall.

`ImportToScene()`

Imports the wall object to the simulation scene.

WallBoxPlate

These classes represent wall objects in the simulation scene, with a box shape and a plane or a plate surface.

Constructor

`WallBoxPlate(x, y, z, w, h, d)`

Creates a new instance of the WallBoxPlate class with the following parameters:

- `x` (float): x coordinate of the center of the box
- `y` (float): y coordinate of the center of the box
- `z` (float): z coordinate of the center of the box
- `w` (float): width of the box
- `h` (float): height of the box
- `d` (float): depth of the box

Methods

`GetShapes()`

Returns a list of shape objects of the wall.

`GetWalls()`

Returns a list of wall objects of the wall.

`ImportToScene()`

Imports the wall object to the simulation scene.

BondedSpheres

This function initializes a PyBind11 module for the BondedSpheres class.

Attributes

`sphere`
A Sphere object representing the bonded sphere.

`particle_list`
A list of Particle objects.

`contact_list`
A list of Contact objects.

`bond_pair_list`
A list of bonded particle pairs.

`bond_model`
A reference to the BondModel object used to calculate bonds.

Methods

`init()`
The default constructor.

`init(BondedSpheres)`
A copy constructor that initializes a BondedSpheres object from another BondedSpheres object.

`SetBondModel()`
Sets the bond model used to calculate bonds.

`Translate()`
Translates the BondedSpheres object.

`RotateByRodrigues()`
Rotates the BondedSpheres object using Rodrigues' rotation formula.

`GetCentroid()`
Returns the centroid of the BondedSpheres object.

`InitFromSTL()`
Initializes the BondedSpheres object from an STL file.

- `filename` (string): The path to the STL file.
- `radius` (float): The radius of the sphere.

`InitFromGrid()`
Initializes the BondedSpheres object from a grid.

`MakePorosity()`
Creates porosity in the BondedSpheres object.

`InitBonds()`
Initializes the bonded pairs in the BondedSpheres object.

`ImportToScene()`
Imports the BondedSpheres object to a scene.

BondedVoronois

This class represents a bonded Voronois object.

Constructor

`BondedVoronois()`
Creates a new instance of the BondedVoronois class.

Attributes

`trimesh_list`
A list of trimeshes.

`particle_list`
A list of particles.

`contact_list`
A list of contacts.

`bond_pair_list`
A list of bond pairs.

`cvt_max_iters`
The maximum number of iterations for the CVT algorithm.

`cvt_tol`
The tolerance for the CVT algorithm.

`bond_model`
The bond model used by the BondedVoronois object.

Methods

`SetBondModel(bond_model)`
Set the bond model used by the BondedVoronois object.

- `bond_model` (BondModel): The bond model to set.

`Translate(x, y, z)`
Translate the BondedVoronois object.

- `x` (float): The amount to translate in the x direction.
- `y` (float): The amount to translate in the y direction.
- `z` (float): The amount to translate in the z direction.

`RotateByRodrigues(theta, u)`
Rotate the BondedVoronois object by the Rodrigues formula.

- `theta` (float): The angle of rotation.
- `u` (array-like): The axis of rotation.

GetCentroid()
Get the centroid of the BondedVoronois object.

`InitFromSTL(stl_file, label='')`
Initialize the BondedVoronois object from an STL file.

- `stl_file` (string): The path to the STL file.
- `label` (string, optional): A label for the object. Default is an empty string.

`MakePorosity(porosity)`
Make the BondedVoronois object porous.

- `porosity` (float): The desired porosity.

`InitBonds()`
Initialize the bonds of the BondedVoronois object.

`RefreshPointers()`
Refresh the pointers of the BondedVoronois object.

`SaveAsVTK(filename)`
Save the BondedVoronois object as a VTK file.

- `filename` (string): The name of the VTK file.

`ImportToScene(scene)`
Import the BondedVoronois object to a scene.

- `scene` (Scene): The scene to import to.

ContactPP

This class represents a contact model between two particles.

Constructor

`ContactPP()`
Creates a new instance of the ContactPP class.

`ContactPP(particle_1, particle_2)`
Creates a new instance of the ContactPP class with the given particles.

- `particle_1` (Particle* const): Pointer to the first particle.
- `particle_2` (Particle* const): Pointer to the second particle.

Attributes

`particle_1`
Pointer to the first particle involved in the contact.

`particle_2`
Pointer to the second particle involved in the contact.

`bond_entries`
A vector of bond entries for the contact.

`collision_entries`
A vector of collision entries for the contact.

`active`
A boolean value indicating whether the contact is active or not.

`dynamic_properties`
A dictionary of dynamic properties for the contact.

Methods

`SetBondModel(bond_model)`
Set the bond model for the contact.

- `bond_model` (BondModel*): Pointer to the bond model.

`SetCollisionModel(collision_model)`
Set the collision model for the contact.

- `collision_model` (CollisionModel*): Pointer to the collision model.

`EvaluateForceMoment()`
Evaluate the force and moment for the contact.

`ApplyToParticle()`
Apply the contact force and moment to the particles.

`ApplyToParticle1()`
Apply the contact force and moment to the first particle.

`ApplyToParticle2()`
Apply the contact force and moment to the second particle.

`IsActive()`
Return a boolean indicating whether the contact is active or not.

`Clear()`
Clear the contact data.

`Print()`
Print the contact data.

ContactPW

This class represents a contact between a particle and a wall.

Constructor

`ContactPW(particle, wall)`
Creates a new instance of the `ContactPW` class with a given particle and wall.

- `particle` (Particle object): The particle involved in the contact.
- `wall` (Wall object): The wall involved in the contact.

Attributes

`particle`
The particle involved in the contact.

`wall`
The wall involved in the contact.

`bond_model`
The bond model used for the contact.

`collision_model`
The collision model used for the contact.

`bond_entries`
A list of bond entries for the contact.

`collision_entries`
A list of collision entries for the contact.

`active`
A boolean flag indicating whether the contact is currently active.

`dynamic_properties`
A dictionary of dynamic properties associated with the contact.

Methods

`SetBondModel(bond_model)`
Set the bond model for the contact.

- `bond_model` (BondModel object): The bond model to use for the contact.

`SetCollisionModel(collision_model)`
Set the collision model for the contact.

- `collision_model` (CollisionModel object): The collision model to use for the contact.

`EvaluateForceMoment()`
Calculate and return the force and moment of the contact.

`ApplyToParticle()`
Apply the contact to the particle involved in the contact.

`ApplyToWall()`
Apply the contact to the wall involved in the contact.

`IsActive()`
Return True if the contact is currently active, False otherwise.

`Clear()`
Clear all bond and collision entries for the contact.

`Print()`
Print information about the contact.

Shape

This class represents a shape.

Constructor

`Shape()`
Creates a new instance of the Shape class.

Attributes

`id`
An integer ID for the shape.

`label`
A string label for the shape.

`shape_type`
An enumeration representing the type of the shape.

`shape_name`
A string name for the shape.

`size`
A Vec3d vector representing the size of the shape.

`volume`
A double representing the volume of the shape.

`inertia`
A Vec3d vector representing the inertia tensor of the shape.

`bound_sphere_radius`
A double representing the radius of the bounding sphere of the shape.

skin
A bool indicating whether the shape has a skin.

skin_factor
A double representing the skin factor of the shape.

bound_aabb_min
A Vec3d vector representing the minimum corner of the axis-aligned bounding box of the shape.

bound_aabb_max
A Vec3d vector representing the maximum corner of the axis-aligned bounding box of the shape.

use_node
A bool indicating whether the shape uses a node.

node_num
An integer representing the number of nodes used by the shape.

node_spacing
A double representing the spacing between nodes of the shape.

nodes
A vector of Vec3d vectors representing the nodes of the shape.

is_convex
A bool indicating whether the shape is convex.

use_customized_solver
A bool indicating whether the shape uses a customized solver.

Methods

PackJson()
Pack the shape data into a JSON object.

InitFromJson(json_data)
Initialize the shape from a JSON object.

- `json_data` (JSON object): A JSON object containing the data to initialize the shape.

InitFromJsonFile(json_file)
Initialize the shape from a JSON file.

- `json_file` (string): The path of the JSON file to load.

UpdateNodes()
Update the nodes of the shape.

UpdateShapeProperties()
Update the properties of the shape.

SetSize(size)
Set the size of the shape.

- `size` (Vec3d): The new size of the shape.

Clone()
Create a new instance of the shape that is a copy of the current shape.

GetSTLModel()
Get the STL model of the shape.

SaveAsVTK(filename)
Save the shape as a VTK file.

- `filename` (string): The path of the file to save.

`SaveAsSTL(filename)`
Save the shape as an STL file.

- `filename` (string): The path of the file to save.

`GetBoundAABB()`
Get the axis-aligned bounding box of the shape.

`SignedDistance(point)`
Compute the signed distance between a point and the surface of the shape.

- `point` (Vec3d): The point to compute the signed distance from.

`SurfacePoint(point)`
Find the surface point of the shape closest to a given point.

- `point` (Vec3d): The point to find the closest surface point to.

`Enclose(shapes)`
Create a new shape that encloses a set of shapes.

- `shapes` (vector of Shape objects): The set of shapes to enclose.

`Print()`
Print the properties of the shape to the console.

Sphere

Represents a sphere.

Constructor

`Sphere()`
Creates a new instance of the Sphere class.

`Sphere(radius: float)`
Creates a new instance of the Sphere class with the given radius.

- `radius` (float): The radius of the sphere.

PointSphere

Represents a point sphere.

Constructor

`PointSphere()`
Creates a new instance of the PointSphere class.

`PointSphere(radius: float)`
Creates a new instance of the PointSphere class with the given radius.

- `radius` (float): The radius of the point sphere.

Plane

Represents a plane.

Constructor

`Plane()`
Creates a new instance of the Plane class.

`Plane(center: Vec3d, dir_n: Vec3d, extent: float)`
Creates a new instance of the Plane class with the given center, normal vector, and extent.

- `center` (Vec3d): The center point of the plane.
- `dir_n` (Vec3d): The normal vector of the plane.
- `extent` (float): The extent of the plane.

Attributes

`center`
The center point of the plane.

`dir_n`
The normal vector of the plane.

`extent`
The extent of the plane.

Methods

`SetExtent(extent: float)`
Sets the extent of the plane.

- `extent` (float): The extent to set for the plane.

`SetCenter(center: Vec3d)`
Sets the center point of the plane.

- `center` (Vec3d): The center point to set for the plane.

`SetNormal(dir_n: Vec3d)`
Sets the normal vector of the plane.

- `dir_n` (Vec3d): The normal vector to set for the plane.

Triangle

Represents a triangle.

Constructor

`Triangle()`
Creates a new instance of the Triangle class.

`Triangle(v1: Vec3d, v2: Vec3d, v3: Vec3d)`
Creates a new instance of the Triangle class with the given vertices.

- `v1` (Vec3d): The first vertex of the triangle.
- `v2` (Vec3d): The second vertex of the triangle.
- `v3` (Vec3d): The third vertex of the triangle.

Attributes

`vertices`
The vertices of the triangle.

`dir_n`
The normal vector of the triangle.

Methods

`SetVertices(v1: Vec3d, v2: Vec3d, v3: Vec3d)`
Sets the vertices of the triangle.

- `v1` (Vec3d): The first vertex of the triangle.
- `v2` (Vec3d): The second vertex of the triangle.
- `v3` (Vec3d): The third vertex of the triangle.

Cylinder

This class represents a cylinder.

Constructor

`Cylinder()`
Creates a new instance of the Cylinder class with default values.

`Cylinder(radius, height)`
Creates a new instance of the Cylinder class with the given radius and height values.

Attributes

`radius`
The radius of the cylinder.

`height`
The height of the cylinder.

Methods

`Init()`
Initializes the cylinder with default values.

`Init(radius, height)`
Initializes the cylinder with the given radius and height values.

Ellipsoid

This class represents an ellipsoid.

Constructor

`Ellipsoid()`
Creates a new instance of the Ellipsoid class with default values.

`Ellipsoid(axis_a, axis_b, axis_c)`
Creates a new instance of the Ellipsoid class with the given axis_a, axis_b, and axis_c values.

Attributes

`axis_a`
The length of the semi-axis along the x-axis.

axis_b
The length of the semi-axis along the y-axis.

axis_c
The length of the semi-axis along the z-axis.

Methods

Init()
Initializes the ellipsoid with default values.

Init(axis_a, axis_b, axis_c)
Initializes the ellipsoid with the given axis_a, axis_b, and axis_c values.

SphericalHarmonics

This class represents a spherical harmonics shape.

Constructor

SphericalHarmonics()
Creates a new instance of the SphericalHarmonics class with default values.

SphericalHarmonics(degree)
Creates a new instance of the SphericalHarmonics class with the given degree value.

Attributes

degree
The degree of the spherical harmonics shape.

a_nm
The coefficients of the spherical harmonics shape.

Methods

Init()
Initializes the spherical harmonics shape with default values.

Init(degree)
Initializes the spherical harmonics shape with the given degree value.

InitFromSTL(file_path)
Initializes the spherical harmonics shape from an STL file located at file_path.

InitFromSTL(stl_model)
Initializes the spherical harmonics shape from an STLModel object stl_model.

PolySuperEllipsoid

This class represents a poly super ellipsoid shape.

Constructor

`PolySuperEllipsoid()`
Creates a new instance of the PolySuperEllipsoid class.

Attributes

`axis_a`
The axis a value of the poly super ellipsoid.

`axis_b`
The axis b value of the poly super ellipsoid.

`axis_c`
The axis c value of the poly super ellipsoid.

`order_ab`
The order ab value of the poly super ellipsoid.

`order_c`
The order c value of the poly super ellipsoid.

Methods

`Init(axis_a, axis_b, axis_c, order_ab, order_c)`
Initializes the poly super ellipsoid with the given parameters.

- `axis_a` (float): The axis a value to set.
- `axis_b` (float): The axis b value to set.
- `axis_c` (float): The axis c value to set.
- `order_ab` (float): The order ab value to set.
- `order_c` (float): The order c value to set.

PolySuperQuadrics

This class represents a poly superquadric shape.

Constructor

`PolySuperQuadrics`
(axis_a=1, axis_b=1, axis_c=1, order_a=2, order_b=2, order_c=2, x=0, y=0, z=0, qx=0, qy=0, qz=0)

Creates a new instance of the PolySuperQuadrics class.

- `axis_a` (float, optional): The scaling factor for the x axis. Default is 1.
- `axis_b` (float, optional): The scaling factor for the y axis. Default is 1.
- `axis_c` (float, optional): The scaling factor for the z axis. Default is 1.
- `order_a` (float, optional): The order of the polynomial for the x axis. Default is 2.
- `order_b` (float, optional): The order of the polynomial for the y axis. Default is 2.
- `order_c` (float, optional): The order of the polynomial for the z axis. Default is 2.
- `x` (float, optional): The x-coordinate of the position. Default is 0.
- `y` (float, optional): The y-coordinate of the position. Default is 0.
- `z` (float, optional): The z-coordinate of the position. Default is 0.
- `qx` (float, optional): The x-component of the rotation quaternion. Default is 0.
- `qy` (float, optional): The y-component of the rotation quaternion. Default is 0.
- `qz` (float, optional): The z-component of the rotation quaternion. Default is 0.

Attributes

`axis_a`
The scaling factor for the x axis.

`axis_b`
The scaling factor for the y axis.

`axis_c`
The scaling factor for the z axis.

`order_a`
The order of the polynomial for the x axis.

`order_b`
The order of the polynomial for the y axis.

`order_c`
The order of the polynomial for the z axis.

Methods

`Init()`
Initialize the poly superquadric shape.

`InitFromJSON(json_data)`
Initialize the poly superquadric shape from a JSON object.

- `json_data` (JSON object): A JSON object containing the parameters to set.

`Init(axis_a, axis_b, axis_c, order_a, order_b, order_c)`
Initialize the poly superquadric shape.

- `axis_a` (float): The scaling factor for the x axis.
- `axis_b` (float): The scaling factor for the y axis.
- `axis_c` (float): The scaling factor for the z axis.
- `order_a` (float): The order of the polynomial for the x axis.
- `order_b` (float): The order of the polynomial for the y axis.
- `order_c` (float): The order of the polynomial for the z axis.

LevelSet

This class represents a level set.

Constructor

`LevelSet()`
Creates a new instance of the LevelSet class.

Methods

`InitFromSTL(stl_file, sign)`
Initialize the level set from an STL file.

- `stl_file` (string): The name of the STL file.
- `sign` (int): The sign of the level set.

`InitFromSTL(stl_model, sign)`
Initialize the level set from an STL model.

- `stl_model` (STLModel): The STL model.

- `sign` (int): The sign of the level set.

`InitFromDistanceMap(distance_map)`
Initialize the level set from a distance map.

- `distance_map` : The distance map.

`SurfacePoint(point)`
Get the surface point of the level set.

- `point` (Vector3d): The point.

`AlignAxes()`
Align the axes of the level set.

TriMesh

This class represents a triangular mesh.

Constructor

`TriMesh()`
Creates a new instance of the TriMesh class.

Attributes

`vertices`
A list of vertices that make up the mesh.

`facets`
A list of facets that make up the mesh.

Methods

`Init()`
Initialize the triangular mesh.

`InitFromSTL(file_name)`
Initialize the triangular mesh from an STL file.

- `file_name` (string): The name of the STL file.

`InitFromOFF(file_name)`
Initialize the triangular mesh from an OFF file.

- `file_name` (string): The name of the OFF file.

`Decimate(num_faces)`
Decimate the mesh to reduce the number of faces.

- `num_faces` (int): The target number of faces for the decimated mesh.

`MakeConvex()`
Make the mesh convex.

`AlignAxes()`
Align the axes of the mesh.

InitPyShapeModule

This function initializes a Python module with various shape classes and functions. The following functions are called to add their respective classes to the module:

InitPyShape

InitPySphere

InitPyPointSphere

InitPyPlane

InitPyTriangle

InitPyCylinder

InitPyEllipsoid

InitPySphericalHarmonics

InitPyPolySuperEllipsoid

InitPyPolySuperQuadrics

InitPyLevelSet

InitPyTriMesh

Simulation

This class represents a simulation.

Attributes

domain_manager
The domain manager used in the simulation.

mpi_manager
The MPI manager used in the simulation.

modifier_manager
The modifier manager used in the simulation.

scene
The scene used in the simulation.

dem_solver
The DEM solver used in the simulation.

`mech_time`
The mechanical time used in the simulation.

`mech_cycles`
The mechanical cycles used in the simulation.

`enable_logging`
A flag indicating whether logging is enabled for the simulation.

Methods

`Run()`
Run the simulation.

`AutoReadRestart()`
Automatically read the restart file for the simulation.

`SetTimeAndCycles()` (time, cycles)
Set the mechanical time and cycles for the simulation.

time (float): The mechanical time to set. cycles (int): The mechanical cycles to set.

LevelSetFunction

This class represents a level set function.

Constructor

`LevelSetFunction()`
Creates a new instance of the LevelSetFunction class.

Attributes

`corner`
A vector representing the corner of the level set function.

`spacing`
A vector representing the spacing of the level set function.

`dim`
An integer representing the dimension of the level set function.

Methods

`SetCorner(corner)`
Set the corner of the level set function.

- `corner` (vector): A vector representing the corner to set.

`SetSpacing(spacing)`
Set the spacing of the level set function.

- `spacing` (vector): A vector representing the spacing to set.

`SetDimension(dim)`
Set the dimension of the level set function.

- `dim` (int): An integer representing the dimension to set.

`SignedDistance(pos)`
Compute the signed distance at a given position.

- `pos` (vector): A vector representing the position to compute the signed distance.

`GradientInterpolate(pos)`
Compute the gradient at a given position using interpolation.

- `pos` (vector): A vector representing the position to compute the gradient.

`GradientMinus(pos)`
Compute the gradient at a given position using the minus side.

- `pos` (vector): A vector representing the position to compute the gradient.

`GradientPlus(pos)`
Compute the gradient at a given position using the plus side.

- `pos` (vector): A vector representing the position to compute the gradient.

`Reinitialization()`
Reinitialize the level set function.

`Reinitialization(iterations, cfl)`
Reinitialize the level set function with the given number of iterations and CFL coefficient.

- `iterations` (int): The number of iterations to perform. Default is 1.
- `cfl` (double): The CFL coefficient to use. Default is 0.5.

STLReader

This class is used for reading STL files.

Constructor

`STLReader()`
Creates a new instance of the STLReader class.

Methods

`ReadFile(filename)`
Reads an STL file and returns the vertices and triangles.

- `filename` (string): The name of the STL file to read.

STLModel

This class represents a 3D model loaded from an STL file.

Constructor

`STLModel()`
Creates a new instance of the STLModel class.

`STLModel(vertices, facets)`
Creates a new instance of the STLModel class with the specified vertices and facets.

- `vertices` (list): A list of vertices in the model, where each vertex is a 3D vector. facets (list): A list of facets in the model, where each facet is a tuple of three vertex indices.

Attributes

`vertices`
A list of vertices in the model, where each vertex is a 3D vector.

`facets`
A list of facets in the model, where each facet is a tuple of three vertex indices.

Methods

`InitFromSTL(file_path)`
Loads an STL file and initializes the model from it.

- `file_path` (string): The path to the STL file.

`InitFromOFF(file_path)`
Loads an OFF file and initializes the model from it.

- `file_path` (string): The path to the OFF file.

`Translate(translation)`
Translates the model by the specified translation vector.

- `translation` (list): A 3D vector representing the translation.

`Rotate(rotation)`
Rotates the model by the specified rotation matrix.

- `rotation` (list): A 3x3 rotation matrix.

`CopyPose()`
Creates a copy of the model's current pose.

`CopyPoseDev()`
Creates a copy of the model's current pose on the device.

`SaveAsVTK(file_path)`
Saves the model as a VTK file.

- `file_path` (string): The path to the output VTK file.

`SaveAsSTL(file_path)`
Saves the model as an STL file.

- `file_path` (string): The path to the output STL file.

`RemoveUnreferencedVertices()`
Removes any vertices from the model that are not referenced by any facets.

`RemoveDuplicateVertices()`
Removes any duplicate vertices from the model.

`ReorientFacets()`
Reorients the facets in the model so that their normals point outward.

`Decimate(target_num_faces)`
Reduces the number of facets in the model using mesh decimation.

- `target_num_faces` (int): The desired number of facets in the decimated model.

`Standardize()`
Standardizes the model by centering it at the origin and scaling it to unit size.

`SetSize(size)`
Scales the model to the specified size.

- `size` (float): The desired size of the model.

`MakeConvex()`
Converts the model to a convex hull.

`SmoothMesh(smoothing_iterations)`
Smooths the mesh by averaging the position of each vertex with the positions of its neighbors.

- `smoothing_iterations` (int): The number of smoothing iterations to perform.

`MergeSTLModel(other_model)`
Merges the current model with another STLModel.

- `other_model` (STLModel): The model to merge with.

`GetTriangleStrips()`
Returns the model as a list of triangle strips.

`IsFaceOutside(facet_index, point)`
Checks whether a given point is outside the face with the specified index.

- `facet_index` (int): The index of the face to check.
- `point` (list): A 3D vector representing the point to check.

`IsConvex()`
Checks whether the model is convex.

WSCVTSampler

This class represents a sampler using the weighted stochastic collocation (WSC) method with the variable transformation technique (VTT).

Constructor

`WSCVTSampler(max_iters=1000, tol=1e-6)`
Creates a new instance of the WSCVTSampler class.

- `max_iters` (int, optional): The maximum number of iterations for the optimization algorithm. Default is 1000.
- `tol` (float, optional): The tolerance for the optimization algorithm. Default is 1e-6.

Attributes

`max_iters`
The maximum number of iterations for the optimization algorithm.

`tol`
The tolerance for the optimization algorithm.

Methods

`GetInstance()`
Returns a singleton instance of the WSCVTSampler class.

`Get(num_samples, new_random=False)`
Generates a set of samples using the WSC method with VTT.

- `num_samples` (int): The number of samples to generate.
- `new_random` (bool, optional): If True, generate a new set of random numbers. Default is False.

Voronoi

This class represents a Voronoi object.

Methods

`Solve(points, stl_model, bool_val)`
Computes Voronoi tessellation given a set of points and an STLModel.

- `points` (`VecXT<Vec3d>`): A vector of Vec3d points.
- `stl_model` (STLModel): An instance of the STLModel.
- `bool_val` (bool): A boolean value.

`Solve(stl_model, int_val1, int_val2, double_val, bool_val)`
Computes Voronoi tessellation given an STLModel, and specified parameters.

- `stl_model` (STLModel): An instance of the STLModel.
- `int_val1` (int): An integer value.
- `int_val2` (int): An integer value.
- `double_val` (double): A double value.
- `bool_val` (bool): A boolean value.

`SaveAsVTK()`
Saves the Voronoi tessellation as a VTK file.

Cork

This class provides methods for boolean operations on triangle meshes.

Constructor

`Cork()`
Creates a new instance of the Cork class.

Methods

`MeshIntersect(vertices1, triangles1, vertices2, triangles2, vertices_out=None, triangles_out=None, labels_out=None)`
Computes the intersection between two triangle meshes.

- `vertices1` (`VecXT<Vec3d>`): A vector of vertices defining the first mesh.
- `triangles1` (`VecXT<Vec3i>`): A vector of triangles defining the first mesh.
- `vertices2` (`VecXT<Vec3d>`): A vector of vertices defining the second mesh.
- `triangles2` (`VecXT<Vec3i>`): A vector of triangles defining the second mesh.
- `vertices_out` (`VecXT<Vec3d>`, optional): A vector to store the output vertices. If not provided, a new vector will be created.
- `triangles_out` (`VecXT<Vec3i>`, optional): A vector to store the output triangles. If not provided, a new vector will be created.
- `labels_out` (`VecXT<int>`, optional): A vector to store the output labels. If not provided, a new vector will be created.

`MeshUnion(vertices1, triangles1, vertices2, triangles2, vertices_out=None, triangles_out=None, labels_out=None)`
Computes the union of two triangle meshes.

- `vertices1` (`VecXT<Vec3d>`): A vector of vertices defining the first mesh.
- `triangles1` (`VecXT<Vec3i>`): A vector of triangles defining the first mesh.
- `vertices2` (`VecXT<Vec3d>`): A vector of vertices defining the second mesh.
- `triangles2` (`VecXT<Vec3i>`): A vector of triangles defining the second mesh.
- `vertices_out` (`VecXT<Vec3d>`, optional): A vector to store the output vertices. If not provided, a new vector will be created.
- `triangles_out` (`VecXT<Vec3i>`, optional): A vector to store the output triangles. If not provided, a new vector will be created.

- `labels_out` (`VecXT<int>`, optional): A vector to store the output labels. If not provided, a new vector will be created.

`MeshDifference(vertices1, triangles1, vertices2, triangles2, vertices_out=None, triangles_out=None, labels_out=None)`

Computes the difference between two triangle meshes.

- `vertices1` (`VecXT<Vec3d>`): A vector of vertices defining the first mesh.
- `triangles1` (`VecXT<Vec3i>`): A vector of triangles defining the first mesh.
- `vertices2` (`VecXT<Vec3d>`): A vector of vertices defining the second mesh.
- `triangles2` (`VecXT<Vec3i>`): A vector of triangles defining the second mesh.
- `vertices_out` (`VecXT<Vec3d>`, optional): A vector to store the output vertices. If not provided, a new vector will be created.
- `triangles_out` (`VecXT<Vec3i>`, optional): A vector to store the output triangles. If not provided, a new vector will be created.
- `labels_out` (`VecXT<int>`, optional): A vector to store the output labels. If not provided

---

OpenMP

This module provides bindings for `OpenMP`, a specification for parallel programming in C++.

Functions

`omp_get_max_threads() -> int`

Returns the maximum number of threads that can be used in parallel sections.

`omp_set_num_threads(num_threads: int) -> None`

Sets the number of threads to be used in parallel sections.

- `num_threads (int)`: The number of threads to be used.