An overview of some new features of deal.II 9.0

Mauro Bardelloni, Nicola Giuliani, Luca Heltai[1],
Andrea Mola, Dirk Peschka, Alberto Sartori,

[1]SISSA - International School for Advanced Studies

27 July 2018, SISSA - Trieste

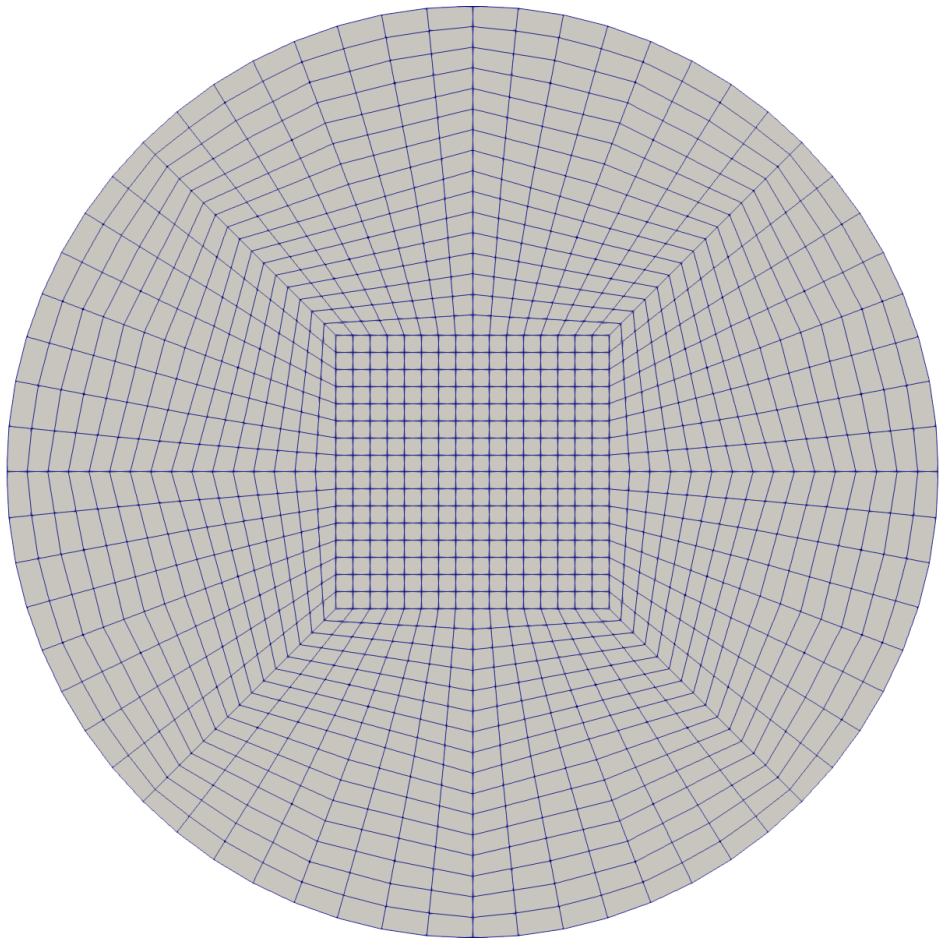
Outline

1. Improved support for curved geometries
2. Non-standard quadrature rules
3. User-defined run-time parameters
4. GMSH
5. NanoFlann
6. SUNDIALS

Curved geometries

1. Automatic Manifolds

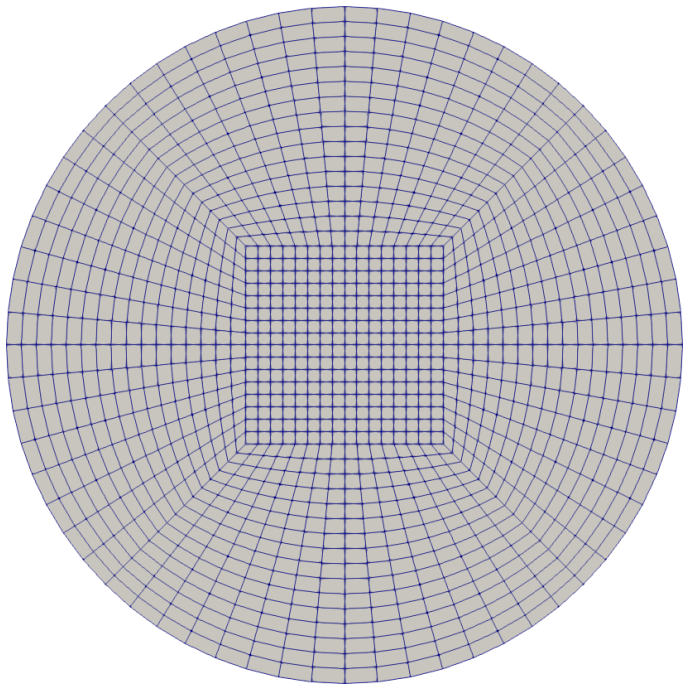
```
1 GridGenerator::hyper_ball (triangulation);
2 triangulation.refine_global (4);
```



Curved geometries

2. Transfinite manifold

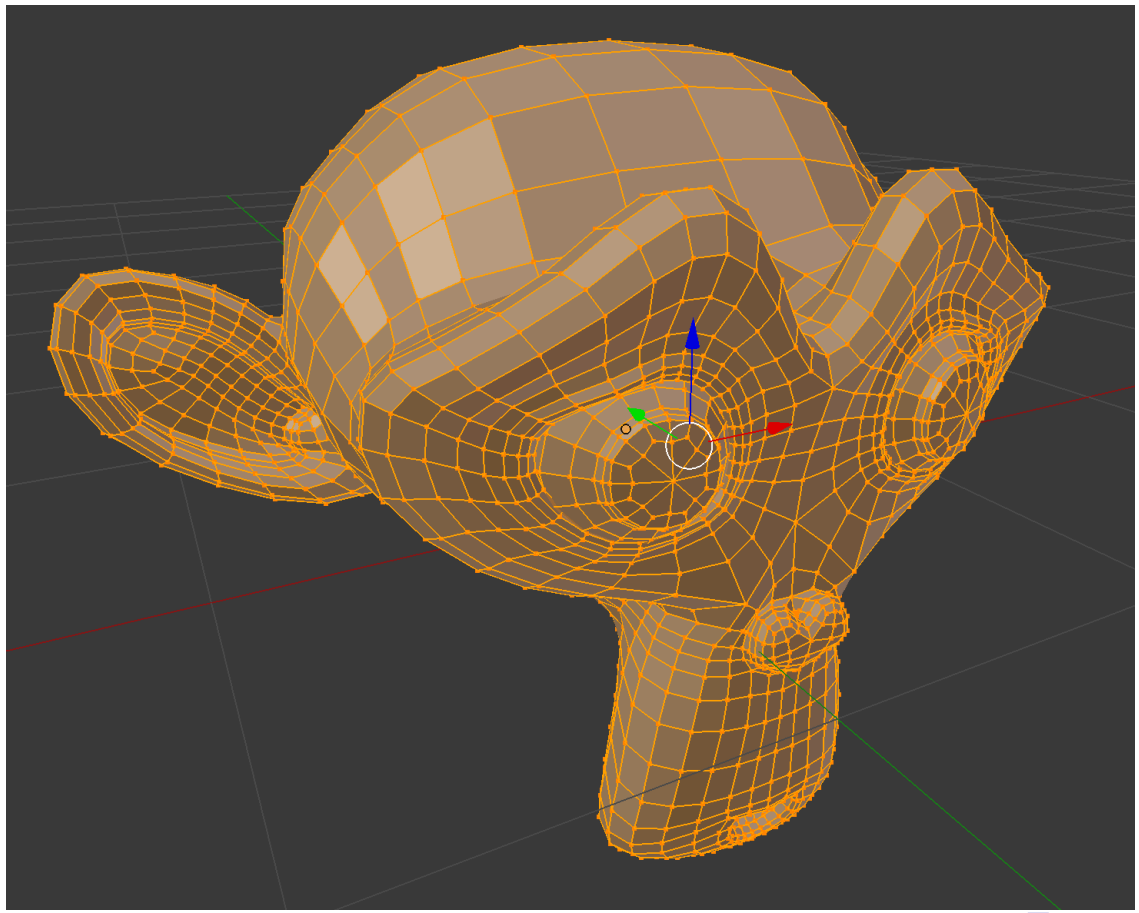
```
1 TransfiniteInterpolationManifold<dim> inner_manifold;
2
3 triangulation.set_all_manifold_ids(1);
4 triangulation.set_all_manifold_ids_on_boundary(0);
5 inner_manifold.initialize(triangulation);
6 triangulation.set_manifold (1, inner_manifold);
```



Curved geometries

3. Support for external libraries (Assimp, see Nicola's talk)

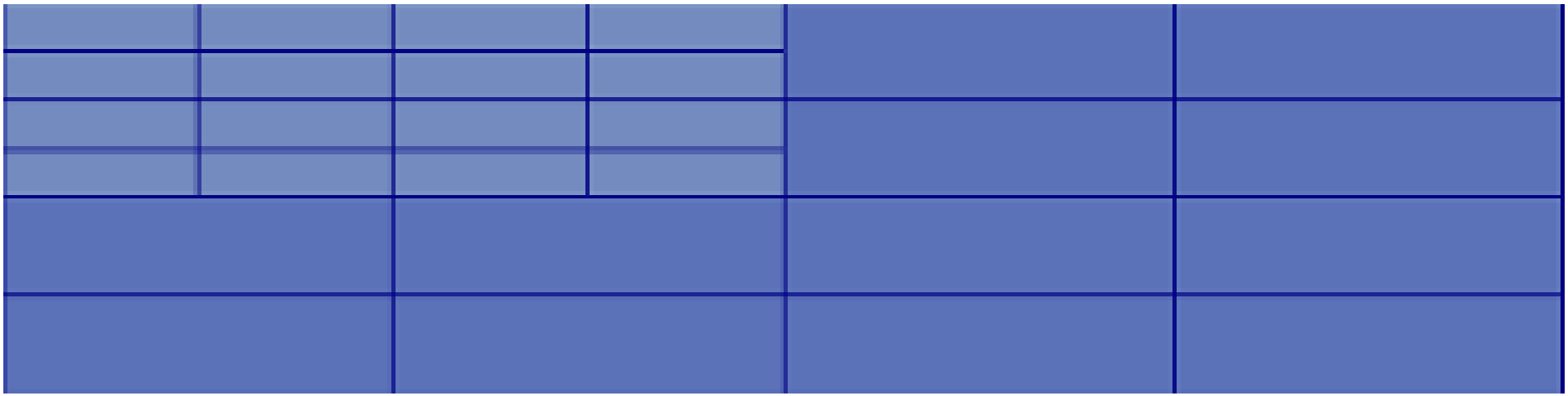
```
1 GridIn<dim> gi;
2 gi.attach_triangulation(tria);
3 gi.read_assimp("input_file.obj");
```



Grid regularization

1. Remove hanging nodes

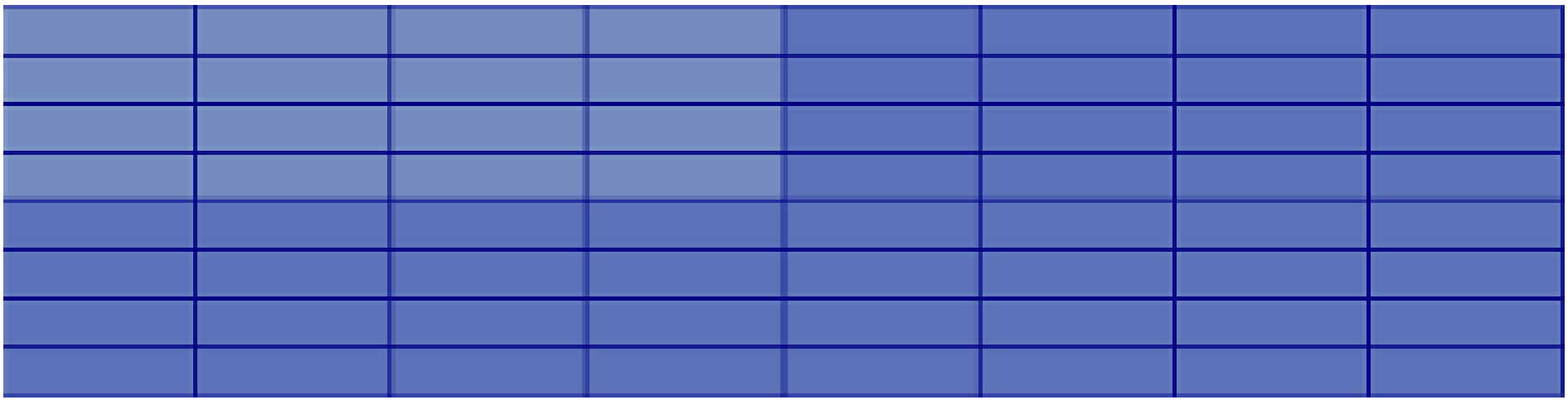
```
1 remove_hanging_nodes (tria, true /* isotropic */);
```



Grid regularization

1. Remove hanging nodes

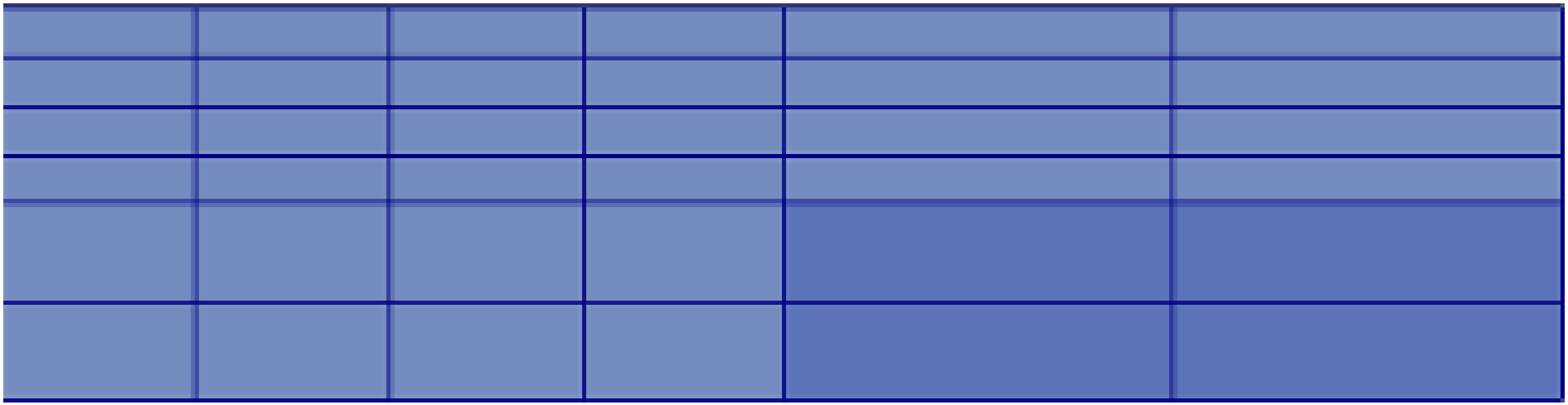
```
1 remove_hanging_nodes (tria, true /* isotropic */);
```



Grid regularization

1. Remove hanging nodes

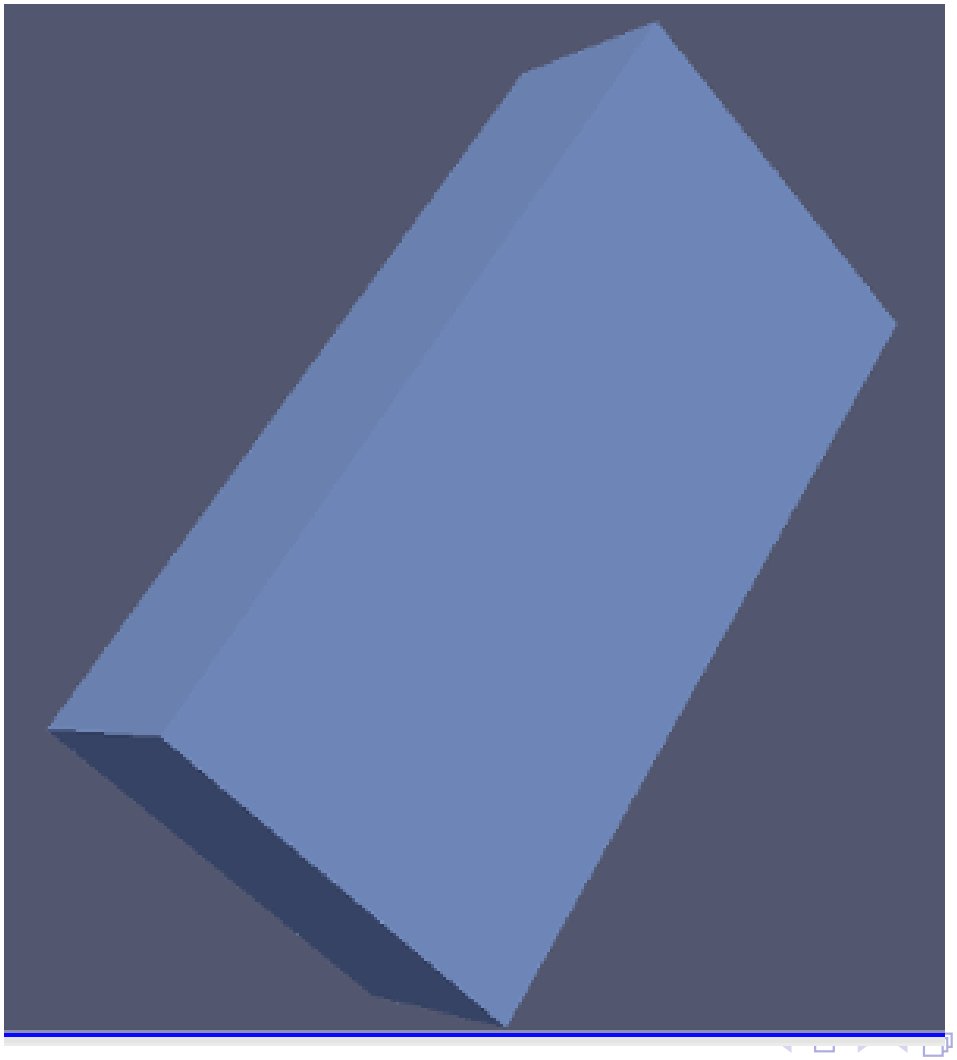
```
1 remove_hanging_nodes (tria, false /* isotropic */);
```



Grid regularization

1. Remove anisotropy

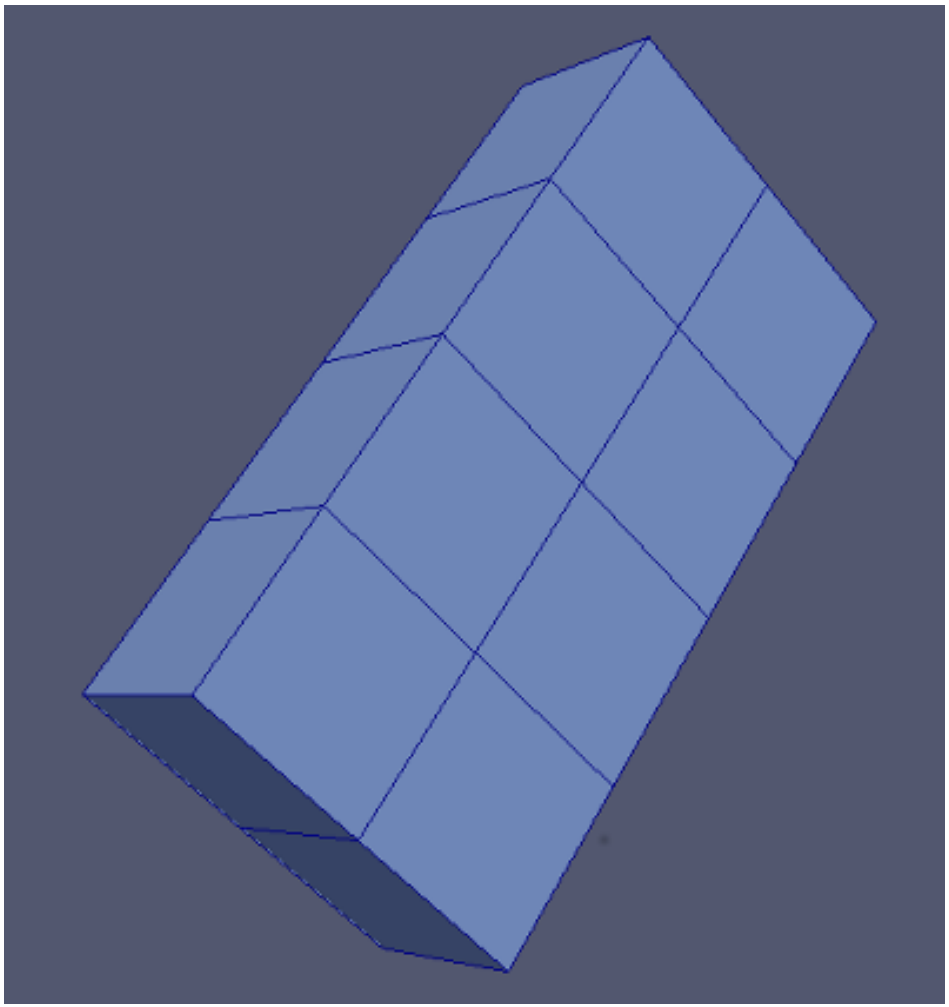
```
1 remove_anisotropy (tria);
```



Grid regularization

1. Remove anisotropy

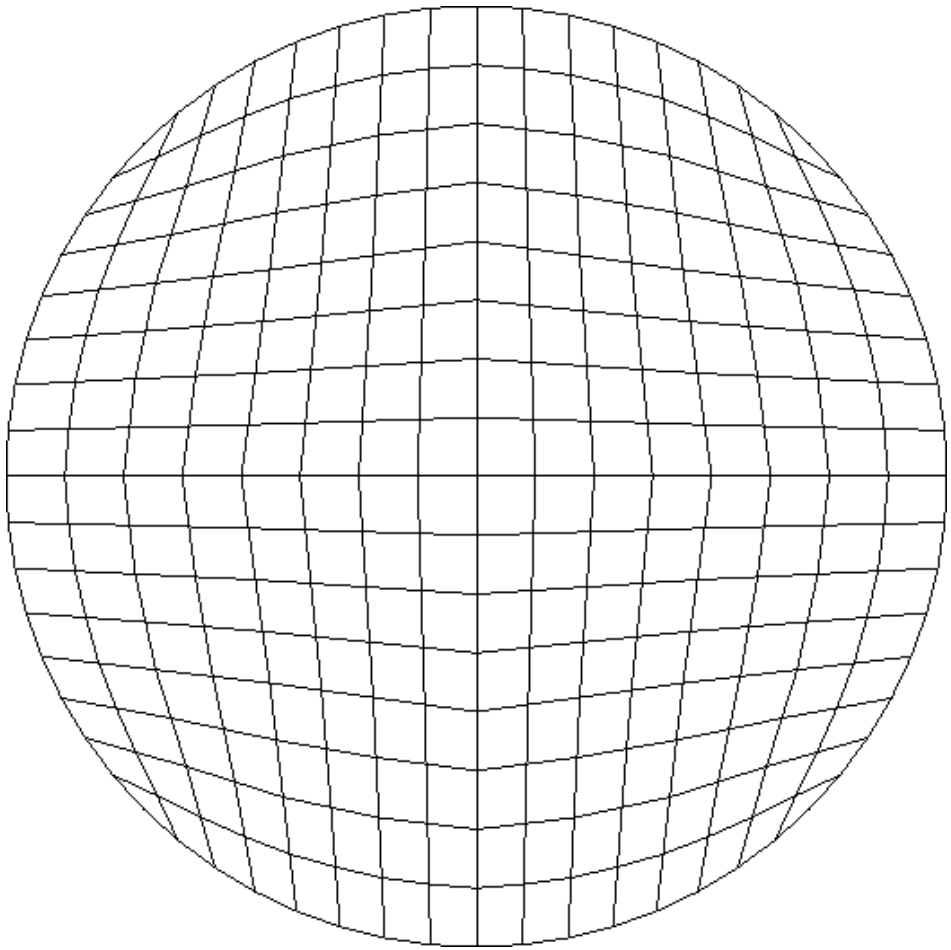
```
1 remove_anisotropy (tria);
```



Grid regularization

1. Regularize corner cells

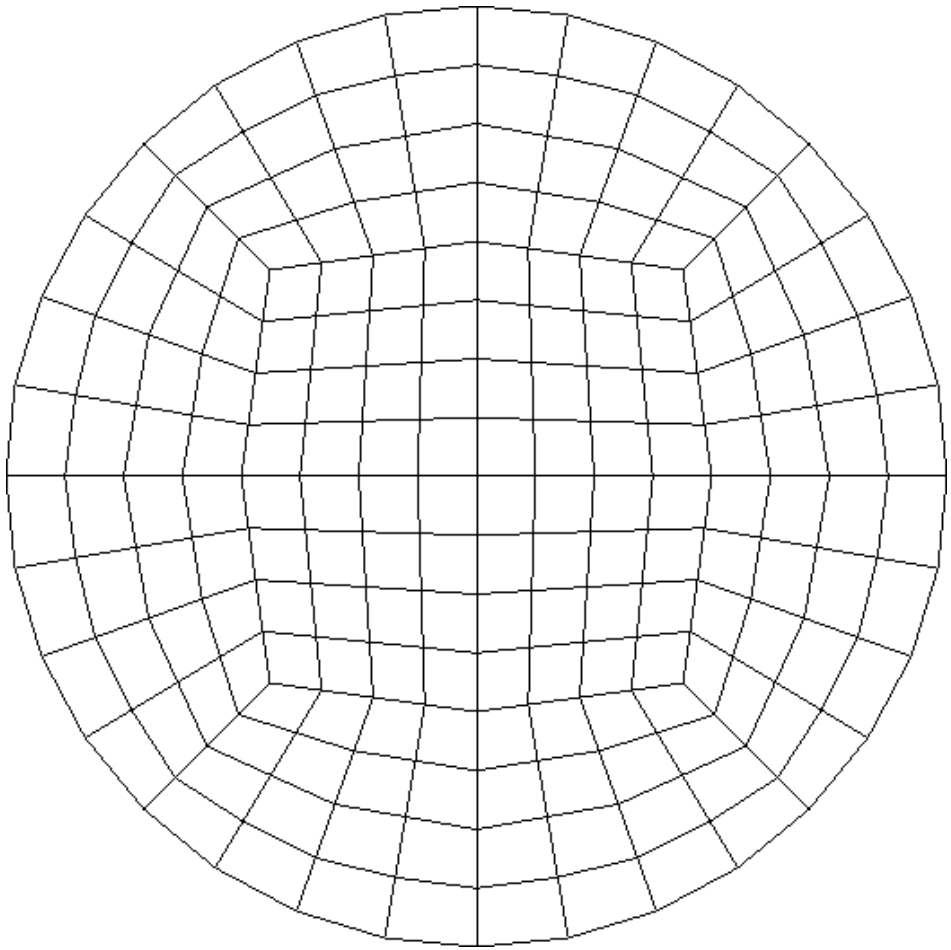
```
1 regularize_corner_cells (tria);
```



Grid regularization

1. Regularize corner cells

```
1 regularize_corner_cells (tria);
```



Quadrature rules

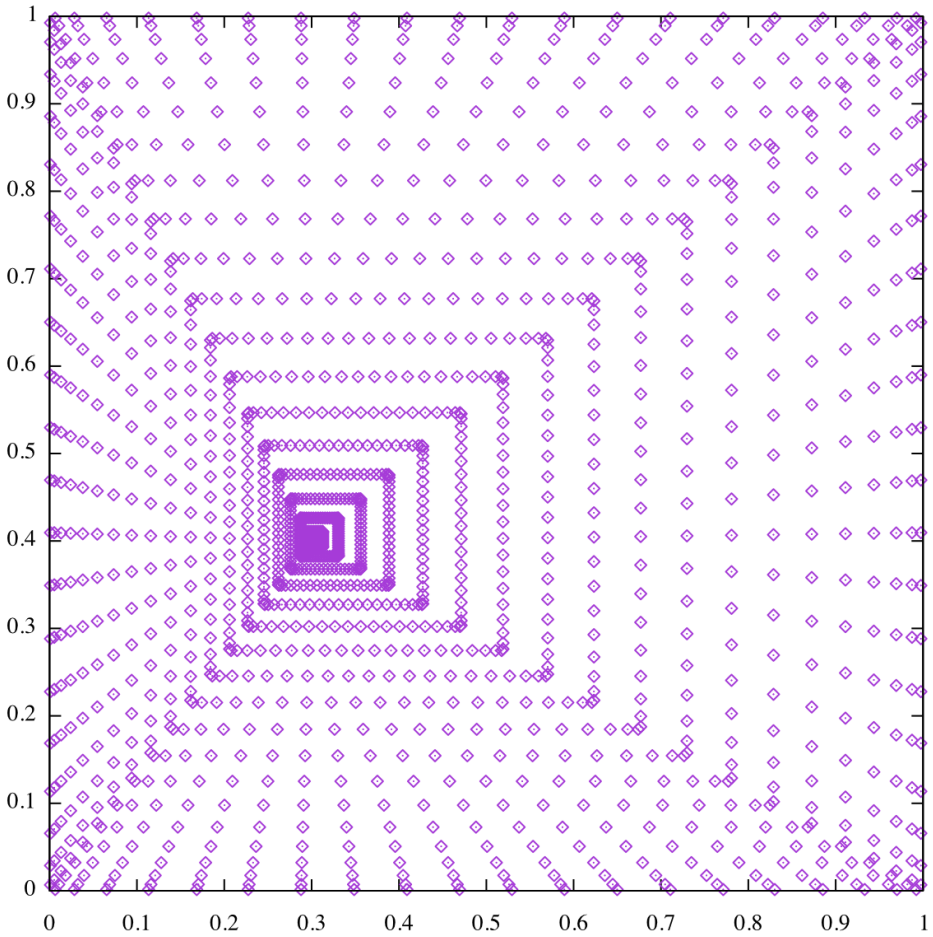
- `QSimplex`: chop a quadrature, only keep points on reference simplices
- `QTrianglePolar`, `QDuffy`, `QTelles`: simplex quadratures, with zero jacobian in $(0, 0)$
- `QSplit`: using a `QSimplex`, cover the reference quad or hex with simplices (with consistent simplex vertex on split point)

Essential for singular integration (BEM)

Example of split quadrature

Works well for singular integration

```
1  Qsplit<2> quad(QDuffy(20), Point<2>(.4, .4));
```



User parameters

Simplify definition and usage

```
1 double kappa = 1.0;
2 std::vector<std::pair<types::boundary_id, Point<dim>>
      special_points;
3 ParameterHandler prm;
4
5 prm.add_parameter("The famous kappa constant", kappa);
6 prm.add_parameter("Some special boundary points",
7                   special_points);
```

Parameter aware classes

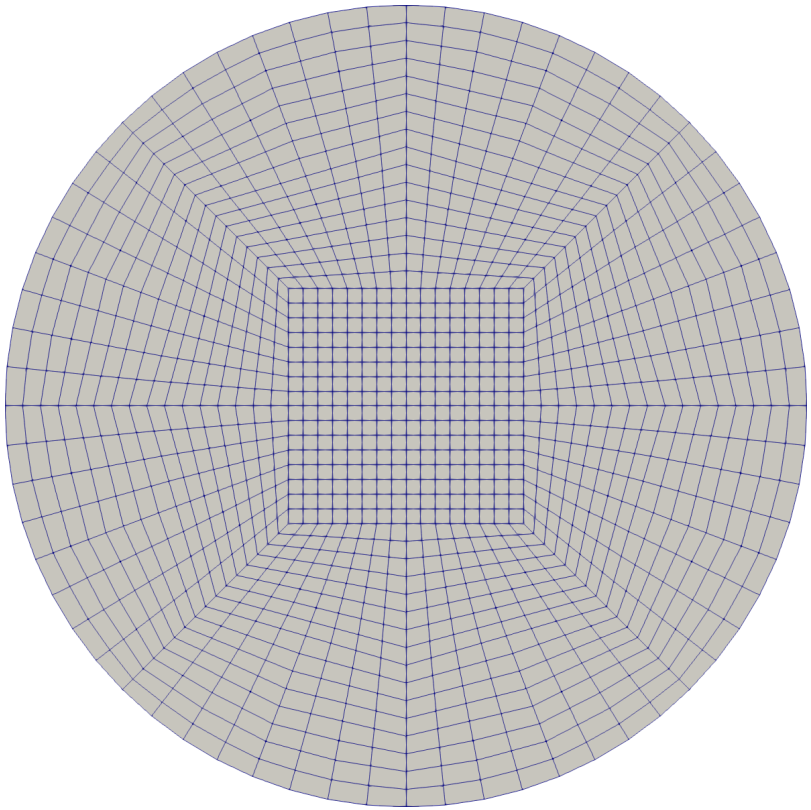
```
1  // This is your own class, derived from ParameterAcceptor
2  class MyClass : public ParameterAcceptor
3  {
4    // Section names also accepts "/" separated strings
5    MyClass()
6      : ParameterAcceptor("Some class name")
7    {
8      add_parameter("A param", member_var);
9    }
0  private:
1    std::vector<unsigned int> member_var;
2  };
3  int main()
4  {
5    // Create your object BEFORE
6    MyClass my_class;
7    // With this call, all derived classes will have their
8    // parameters initialized
9    ParameterAcceptor::initialize("file.prm");
0  }
```

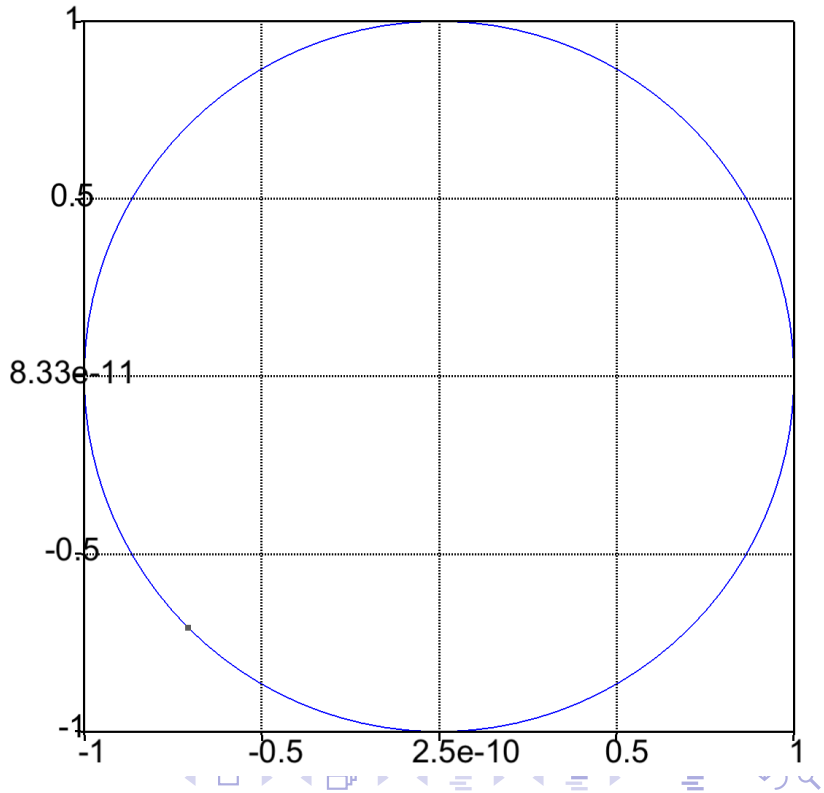
Parameter aware classes

```
1  // If you have a member function, you could use
2  // ParameterAcceptorProxy
3
4  int main()
5  {
6    MyClass my_class;
7
8    ParameterAcceptorProxy<Functions::ParsedFunction<2> >
9      fun("Some function");
0
1
2    // my_class parameters and fun parameters gets initialized
3    ParameterAcceptor::initialize("file.prm");
4  }
```

GMSH

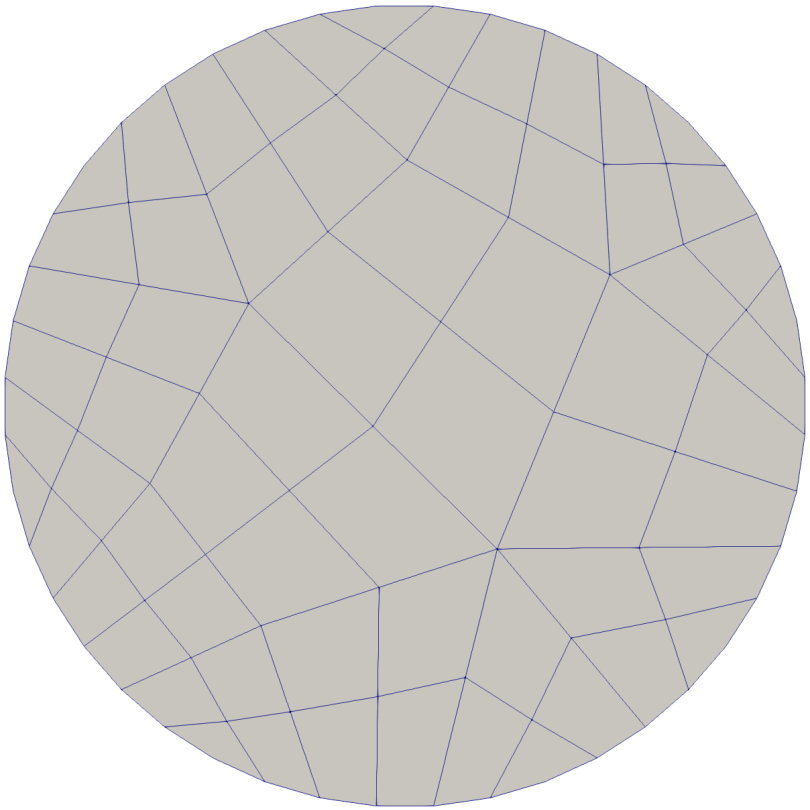
```
1 // Remesh triangulations
2 auto c =
3  OpenCASCADE::create_curves_from_triangulation_boundary(tria);
4 // OpenCASCADE::write_IGES(curves[0], "boundary.iges");
5 Gmsh::create_triangulation_from_boundary_curve(c[0], tria2);
6 GridTools::regularize_corner_cells(tria2);
```

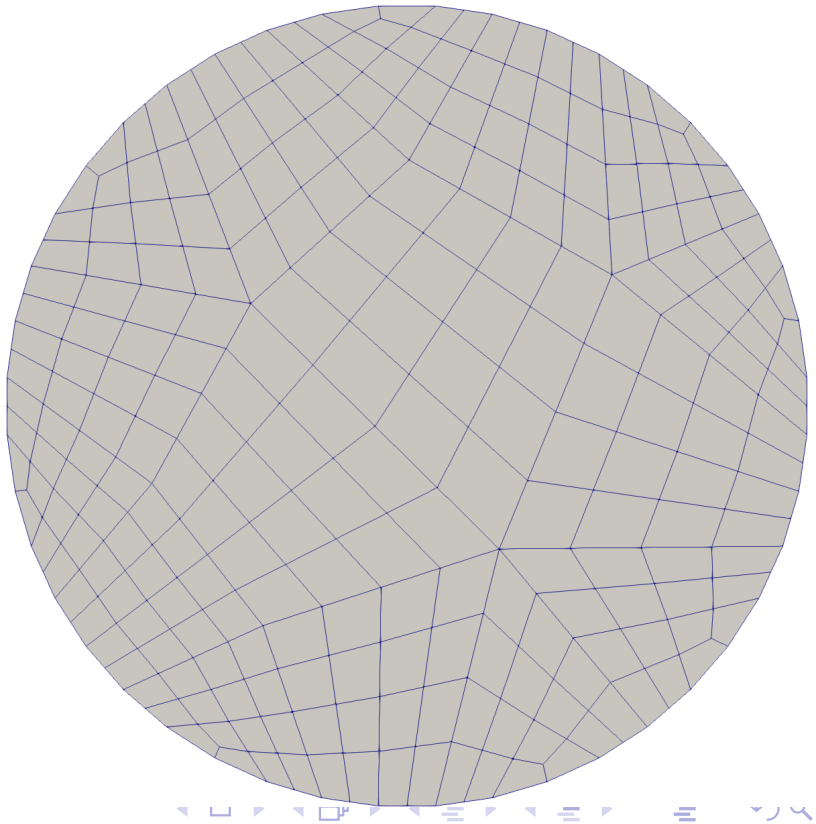




GMSH

```
1 // Remesh triangulations
2 auto c =
3  OpenCASCADE::create_curves_from_triangulation_boundary(tria);
4 // OpenCASCADE::write_IGES(curves[0], "boundary.iges");
5 Gmsh::create_triangulation_from_boundary_curve(c[0], tria2);
6 GridTools::regularize_corner_cells(tria2);
```





KDTree

```
1  // Find vertices in O(log(N))
2  Point<dim>  p;
3  KDTree<dim> tree(10, tria.get_vertices());
4  auto        v = tree.get_closest_points(p, 3);
5  auto        w = tree.get_points_within_ball(p, .1);
6
7  // Convenient output tools
8  std::cout << Patterns::Tools::to_string(v) << std::endl;
9  std::cout << Patterns::Tools::to_string(w) << std::endl;
```

Produces (cost $\log(n)$ each, with n vertices):

```
169:0.0798531, 60:0.0943704, 229:0.16151
60:0.0943704, 169:0.0798531
```

Two-speed odes:

$$M\dot{y} = f_E(t, y) + f_I(t, y)$$
$$y(t0) = y0$$

- f_E: “slow” scale (or complicated scale), integrated explicitly
- f_I: “fast” scale (“easy to integrate”), integrated implicitly

SUNDIALS

1. ARKODE - II (as an explicit integrator)

```
1  ParameterHandler prm;
2  SUNDIALS::ARKode<Vector<double>>::AdditionalData data;
3  data.add_parameters(prm); ... // Parse parameters
4
5  SUNDIALS::ARKode<Vector<double>> ode(data);
6
7  ode.reinit_vector = ... ;
8  ode.explicit_function = ... ;
9  ode.output_step = ... ;
0
1  Vector<double> y(2);
2  y[0] = 0;
3  y[1] = 1.0;
4
5  ode.solve_ode(y);
6  return 0;
```

SUNDIALS

1. ARKODE - II (as an explicit integrator)

```
1  ode.reinit_vector = [&](VectorType &v) { v.reinit(2); };
```

```
1  double kappa = 1.0;
2  ode.explicit_function =
3      [&](double, const VectorType &y, VectorType &ydot) ->
           int {
4      ydot[0] = y[1];
5      ydot[1] = -kappa * kappa * y[0];
6      return 0;
7    };
```

```
1  ode.output_step = [&](const double        t,
2                        const VectorType & sol,
3                        const unsigned int step_number) ->
                             int {
4      out << t << " " << sol[0] << " " << sol[1] << std::endl;
5      return 0;
6    };
```

SUNDIALS

1. ARKODE - III (as an explicit - implicit integrator)

```
...
ode.reinit_vector = ... ;
ode.explicit_function = ... ;
ode.implicit_function = ... ;
// If you don't have a jacobian, it will be approximated
// ode.solve_jacobian_system = ... ;
ode.output_step = ... ;

Vector<double> y(2);
y[0] = 0;
y[1] = 1.0;

ode.solve_ode(y);
return 0;
```

Differential algebraic equation

$$F(y, \dot{y}, t) = 0$$
$$y(t0) = y0$$
$$\dot{y}(t0) = y0$$

- $\partial_{\dot{y}} F$ may be non-invertible (think of Navier-Stokes)
- Solved using variable order, varial time step BDF methods

SUNDIALS

1. IDA - II

```
1  SUNDIALS::IDA<Vector<double>>::AdditionalData data;
2  ParameterHandler                              prm;
3  data.add_parameters(prm);
4  ... // Parse parameters
5
6  SUNDIALS::IDA<Vector<double>>  ida(data);
7
8  ida.reinit_vector = ... ;
9  ida.residual = ... ;
0  ida.setup_jacobian = ... ;
1  ida.solve_jacobian_system = ... ;
2  ida.output_step = ... ;
3
4  y[1]      = kappa;
5  y_dot[0] = kappa;
6  ida.solve_dae(y, y_dot);
```

Conclusions

- A lot of new external libraries
- We need many more tutorial programs...
- Should we support more?
- Should we support less?