An Application Framework for deal.II

Guido Kanschat

`kanschat@dealii.org`

Universität Heidelberg

Building a Complex Application (My Way)

1. Write program for a linear, stationary PDE of similar structure
2. Add a nonlinear residual and use the linear program inside a quasi-Newton method
3. Write a timestepping scheme around this and use the previous solver in each timestep
4. Use this as black box in a multiple shooting optimization
5. Perform mesh adaption for efficient accuracy program

Some Observations

- Outer solvers rely on inner solvers
- Inner solvers must be controlled by outer solvers
- Remeshing should be controllable from any point in this hierarchy
- Not all kinds of solvers implement all functions

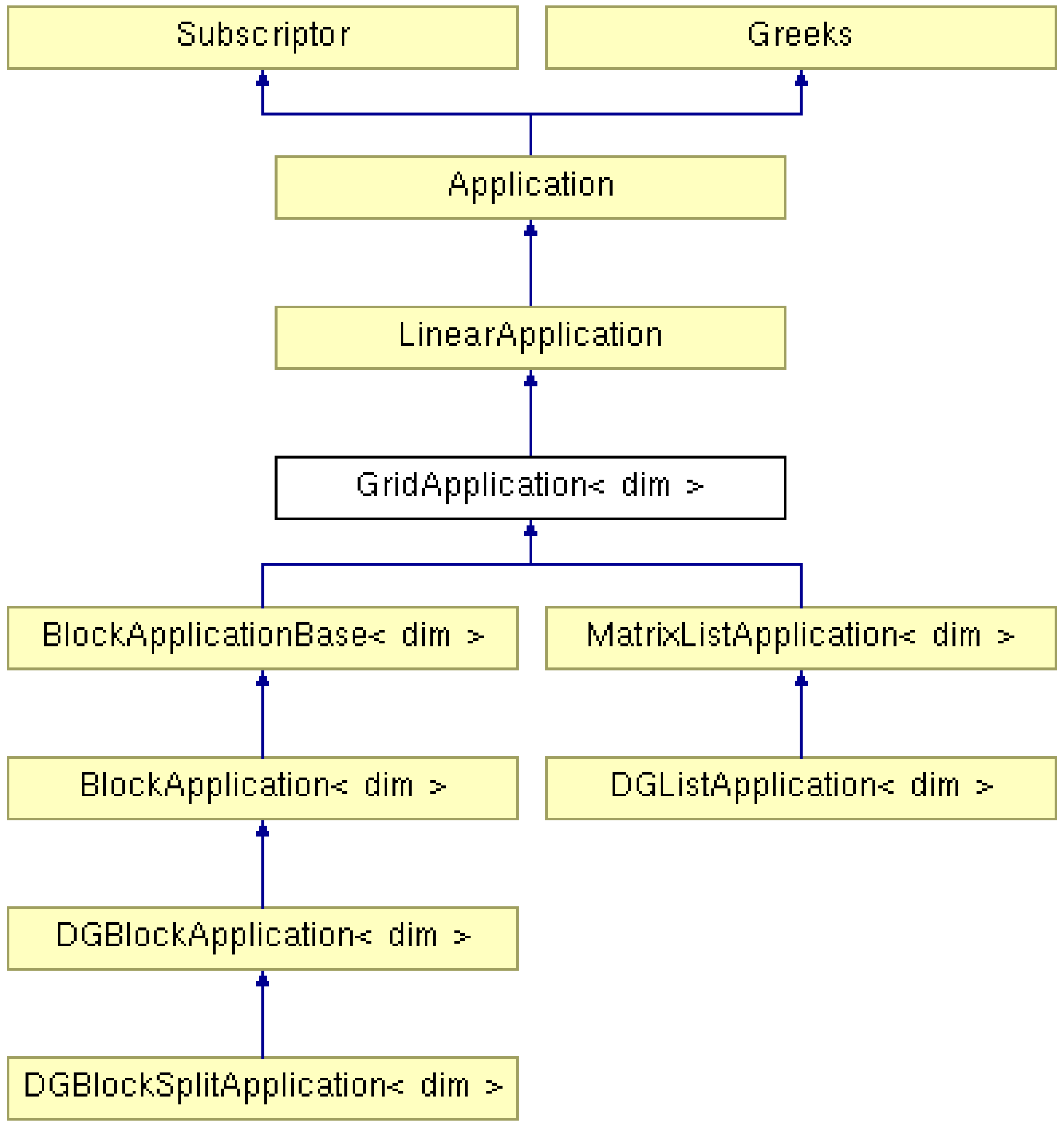
The Abstract Application Class (Public Interface)

```
class Application : public Subscriptor
{
  void   solve(...);
  double residual(...);

  void   remesh();
  void   assemble(...);

  void   evaluate(...);
  double estimate(...);
};
```

The Function remesh

```
void GridApp::remesh(
  Vector* previous_nonlinear = 0,
  Vector* previous_timestep = 0)
{
  remesh_prepare();     // UserApp
  remesh_grid();        // GridApp
  remesh_post();        // UserApp
  remesh_dofs();        // GridApp
  remesh_matrix(previous_nonlinear,
                previous_time);
  if (multigrid)
    {
      remesh_mg_dofs();
      remesh_mg_matrix();
    }
}
```

Inheritance of GridApp

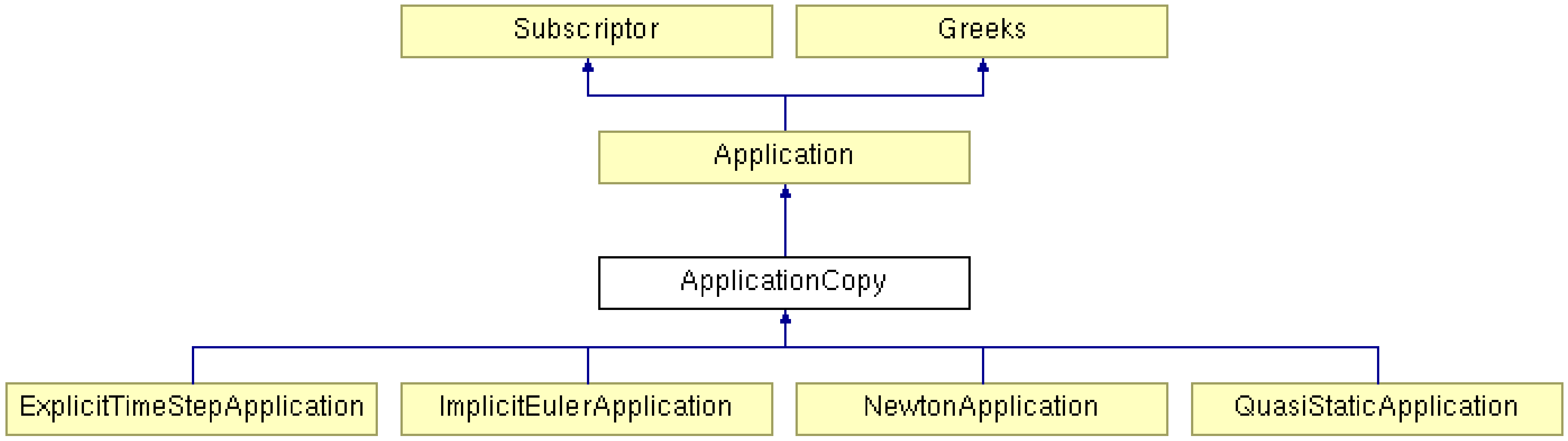


The function `remesh_matrix`

- DGBlockApp
  - initialize BlockSparsityPattern
  - reinit matrix
- DGListApp
  - initialize SparsityPatterns
  - reinit matrices

The class ApplicationCopy

- Contains a pointer to an Application
- Implements all pure functions of Application by forwarding them to the applicaton pointed to



Example: Newton's Method

```
NewtonApplication newton(&my_app);
```

- Compute residual (`my_app->residual`)
- Assemble matrix (`my_app->assemble`)
  - optionally, if last residual too large
- Solve system (`my_app->solve`)
- Add solution to iterate

✘ `assemble`, `remesh`?

Example: Backward Euler Method

- Choose start vector
- Assemble matrix (`my_app->assemble`)
  - optionally, in each step
  - consider mass matrix term
- Assemble right hand side in each step
- Solve system in each step (`my_app->solve`)

✘ `assemble`, `remesh`?

Function arguments

```
void solve(
  Vector& start,
  const Vector& rhs,
  const Vector* previous_nonlinear = 0,
  const Vector* previous_timestep = 0);

void assemble(
  const Vector* previous_nonlinear = 0,
  const Vector* previous_timestep = 0);
```

Assembling

- Implemented in classes
  - DGBlockApp
  - DGBlockAppSplit
  - DGListApp
  - ...
- Relies on assembling routines on cells, faces.

Remarks on cell assembling

Example: Stokes equations

- Modularize
  - Compute matrix by base element
- Compute the Laplacian only once
- Compute either divergence or gradient
  - Use BlockMatrixArray to combine matrices

MatrixShop

- Functions for assembling always similar
- Provide these for standard operators
  - (scaled) Laplacian
  - Advection
  - Divergence
  - Elasticity
- UserApp only combines these

Cell assembling Stokes I (u,v)

```
FullMatrix<double> M(fe_u.dofs_per_cell);
laplacian_scaled_cell(M, fe_u, data.diffusion);
if (with_advection)
  grw_cell(M, u, Du, fe_u);
if (time_step != 0.)
  mass_cell(M, fe_u, 1./time_step);
A.add(M, 1., ustart, ustart);
```

Cell assembling Stokes II (u,q)

```
if (fe_u.get_fe().is_primitive())
  M.reinit(fe_p.dofs_per_cell, dim*fe_u.dofs_per
else
  M.reinit(fe_p.dofs_per_cell, fe_u.dofs_per_cel

div_cell(M, fe_u, fe_p);
A.add(M, -1., pstart, ustart);
```

```
MatrixShop<dim>::laplacian_cell (FullMatrix<double>& M,
  const FEValuesBase<dim>& fe, const double factor)
{
  const unsigned int n_dofs = fe.dofs_per_cell;
  for (unsigned k=0;k<fe.n_quadrature_points;++k)
    {
      const double dx = fe.JxW(k);
      for (unsigned i=0;i<n_dofs;++i)
        {
          for (unsigned j=0;j<n_dofs;++j)
            {
              if (fe.get_fe().is_primitive())
                M(i,j) += dx * factor *
                          (fe.shape_grad(j,k) * fe.shape_grad(i,k));
              else
                for (unsigned int d=0;d<dim;++d)
                  M(i,j) += dx * factor *
                            (fe.shape_grad_component(j,k,d)
                             * fe.shape_grad_component(i,k,d));
            }
        }
    }
}
```