In General

main() template

```
#include <config.h>

#include <dune/common/parallel/mpihelper.hh>

int main(int argc, char **argv)
{
  Dune::MPIHelper::instance(argc, argv);

  // your code goes here
}
```

.cc-file template

```
#include <config.h>

// your code and includes go here
```

.hh-file template

```
// For a header that is included like
// #include <dune/module/dir/header-name.hh>
#ifndef DUNE_MODULE_DIR_HEADER_NAME_HH
#define DUNE_MODULE_DIR_HEADER_NAME_HH

// your code and includes go here
// do not #include <config.h>

#endif // DUNE_MODULE_DIR_HEADER_NAME_HH
```

dune-common

In the following, `r` is of type `R`, which may be a scalar real type, e.g. `double` or `float`. `k` is of type `K`, which may be may be `R` or `std::complex<R>`.

```
template<class K, int size> class FieldVector;
```

```
#include <dune/common/fvector.hh>

FieldVector<K, 2> x = { 0, 1 }; // $x_0 := 0, x_1 := 1$
FieldVector<K, 2> y(k);          // $x_i := k \quad \forall i$

assert(i < x.dim()); // get number of entries
k = x[i]; x[i] = k;  // access/assign entry
for(const auto &entry : x)
  k += entry;          // access each entry
for(auto &entry : x)
  entry = k;           // modify each entry

x += y; x -= y; // $x := x+y,\ x := x-y$
x *= k; x /= k; // $x := kx,\ x := k^{-1}x$
k = x * y;      // $k := x^T y = x \cdot y = \sum_i x_i y_i$
k = x.dot(y);   // $k := x^H y = \bar{x} \cdot y = \sum_i \bar{x}_i y_i$

r = x.one_norm();      // $r := \sum_i |x_i|$
r = x.two_norm();      // $r := \sqrt{\sum_i |x_i|^2}$
r = x.infinity_norm(); // $r := \max_i\{|x_i|\}$
```

```
template<class K, int rows, int cols>
class FieldMatrix;
```

```
#include <dune/common/fmatrix.hh>

FieldMatrix<K, 2, 2> S =
{ { 0, 1 },                      // $S_{00} := 0, S_{01} := 1$
  { 2, 3 } };                    // $S_{10} := 2, S_{11} := 3$
FieldMatrix<K, 2, 2> Q(k); // $Q_{ij} := k \quad \forall i,j$

assert(i < S.rows()); // get number of rows
assert(j < S.cols()); // get number of columns
k = S[i][j];          // access entry
S[i][j] = k;          // assign entry
for(const auto &row : S)
  for(const auto &entry : row)
    k += entry;       // access each entry
for(auto &row : S)
  for(auto &entry : row)
    entry = k;        // modify each entry

auto L = Q.leftmultiplyany(S);  // $L := SQ$
auto R = Q.rightmultiplyany(S); // $R := QS$
Q.leftmultiply(S);              // $S := SQ$
Q.rightmultiply(S);             // $S := QS$
S += Q; S -= Q; // $S := S+Q,\ S := S-Q$
S.axpy(k, Q);   // $S := S+kQ$
S *= k; S /= k; // $S := kS,\ S := k^{-1}S$
S.invert();     // $S := S^{-1}$

r = S.frobenius_norm(); // $r := \sqrt{\sum_{ij} |S_{ij}|^2}$
r = S.infinity_norm();  // $r := \max_i\{\sum_j |S_{ij}|\}$
k = S.determinant();    // $k := \det S$

Q.mv     (x, y); // $y := Qx$
Q.mtv    (x, y); // $y := Q^T x$
Q.umv    (x, y); // $y := y + Qx$
Q.umtv   (x, y); // $y := y + Q^T x$
Q.umhv   (x, y); // $y := y + Q^H x$
Q.usmv  (k, x, y); // $y := y + kQx$
Q.usmtv (k, x, y); // $y := y + kQ^T x$
Q.usmhv (k, x, y); // $y := y + kQ^H x$
Q.solve  (x, y); // find $x$ such that $Qx = y$
```

```
#define DUNE_THROW(ExceptionType, message)
```

```
#include <dune/common/exceptions.hh>

if(i > limit)
  DUNE_THROW(Exception, "Error: i > limit ("
             << i << " > " << limit << ")");
```

```
template<class T> std::string className();
template<class T> std::string className(T& t);
```

```
#include <dune/common/classname.hh>

template<class Vector>
void printTypes(const Vector &v) {
  std::cerr << "Info: Vector type is "
            << className<Vector>()
            << ", entry type is "
            << className(v[0]) << std::endl;
}
```

dune-geometry

```
class GeometryType;
```

```
#include <dune/geometry/type.hh>

GeometryType gt;
gt.makeVertex();      gt.makeLine();
gt.makeTriangle();    gt.makeQuadrilateral();
gt.makeTetrahdron();  gt.makePyramid();
gt.makePrism();       gt.makeHexahedron();
gt.makeSimplex(2); // same as makeTriangle()
gt.makeCube(3);    // same as makeHexahedron()
// for each makeShape() there is an isShape()
assert(gt.isHexahedron());
assert(gt.isCube());     // ignore dimension
assert(gt.dim() == 3); // check dimension
```

Concept Geometry

```
using Geo = ...; Geo geo;

using ctype = Geo::ctype;
int ldim = Geo::mydimension;    // local dim
int gdim = Geo::coorddimension; // global dim

Geo::LocalCoordinate  xl; // $\hat{x} \in \text{ctype}^{\text{ldim}}$
Geo::GlobalCoordinate x;  // $x \in \text{ctype}^{\text{gdim}}$
x  = geo.global(xl);      // $x := g(\hat{x})$
xl = geo.local(x);        // $\hat{x} := g^{-1}(x)$

// $J^{-T} \in \text{ctype}^{\text{gdim}\times\text{ldim}}$, $J_{ij} := \partial g_i/\partial\hat{x}_j$, $\mu := \sqrt{|\det J^T J|}$
Geo::JacobianInverseTransposed JInvT =
  geo.jacobianInverseTransposed(xl);
ctype mu = geo.integrationElement(xl);

GeometryType gt = geo.type(); // shape
assert(i < geo.corners()); // count corners
x = geo.corner(i);         // access corner
x = geo.center();          // roughly
ctype v = geo.volume();    // in global coords
```

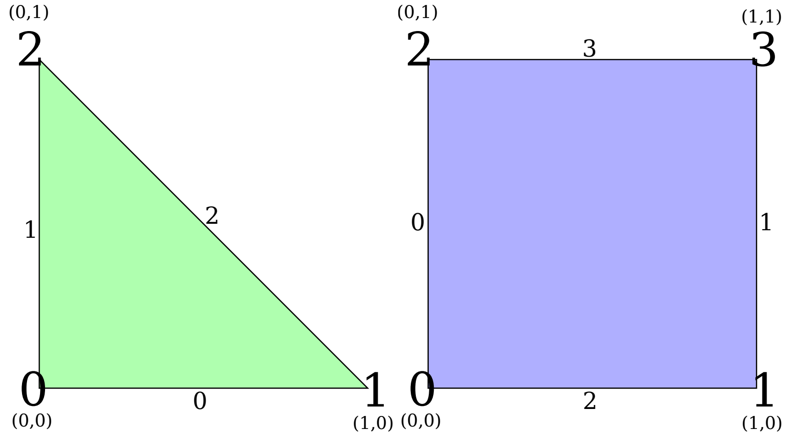
```
template<class ctype, int dim>
class ReferenceElements;
```

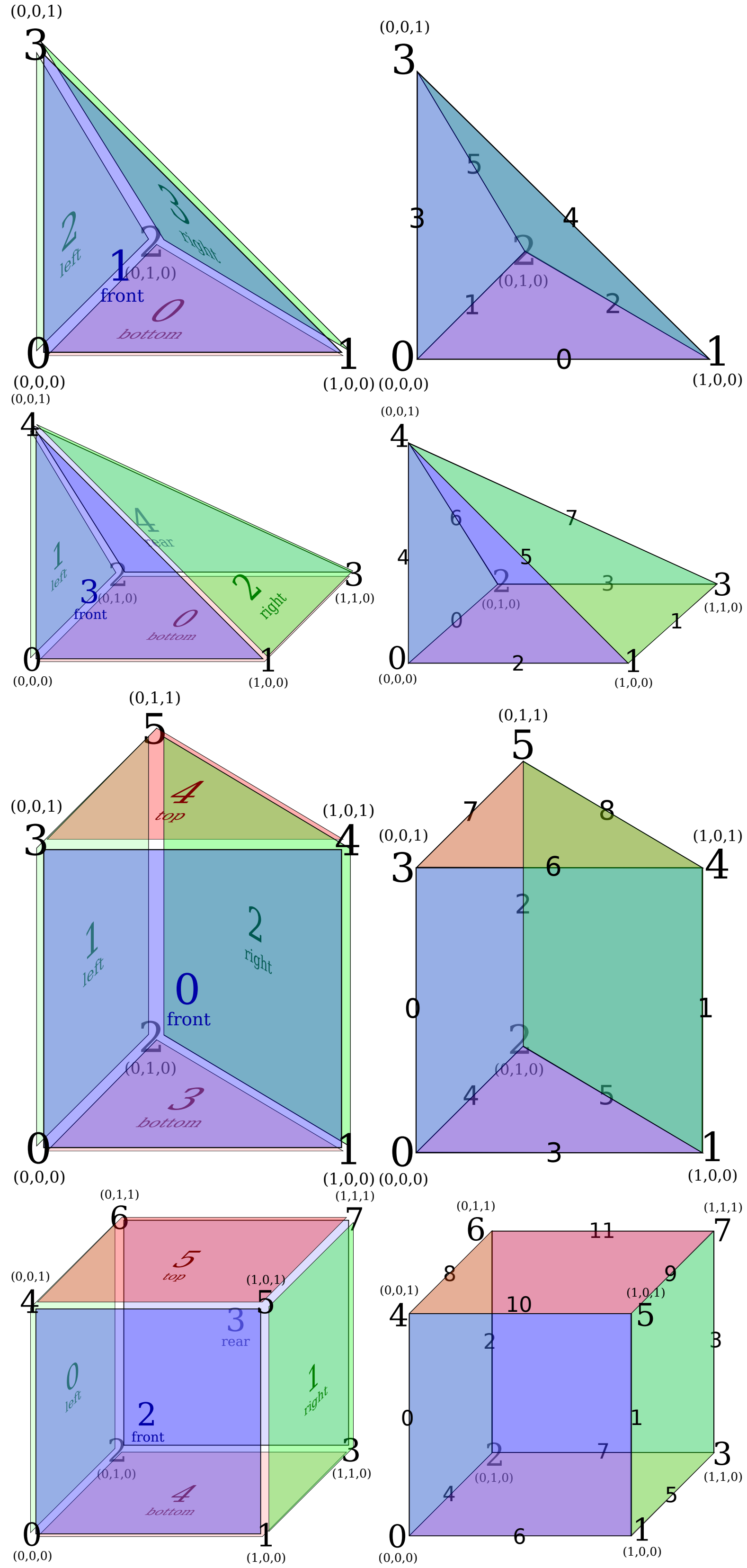
```
#include <dune/geometry/referenceelements.hh>

using Factory = ReferenceElements<ctype, 3>;
const auto &refTet = Factory::simplex();
const auto &refHex = Factory::cube();
GeometryType gt; gt.makePrism();
const auto &ref = Factory::general(gt);

// Info about ref itself
gt = ref.type();
ctype v = ref.volume();
ref.size(c); // count subentities of codim c

// Info about subentity (i,c)
gt = ref.type(i,c);
// position of barycenter
FieldVector<ctype, 3> x = ref.position(i,c)
// count sub-subentities of codim cc
ref.size(i,c, cc);
// transform number of sub-subentity to ref
ref.subEntity(i,c, ii,cc);
```






```
template<class ctype, int dim> class QuadratureRules;
```

```
#include <dune/geometry/quadraturerules.hh>

K f(const FieldVector<ctype, dim> &x);
int p; // max polynomial order of f

K result = 0;
GeometryType gt; gt.makeSimplex(dim);
for(const auto &qp :
    QuadratureRules<ctype, dim>::rule(gt, p))
  result += qp.weight() * f(qp.position());
// now result contains the integral of f()
// over the reference-simplex of dimension dim
```

TODO: integral over a geometry over a scalar
TODO: integral over a geometry over a gradient (incl piola)

dune-grid

```
Grid (YaspGrid, UGGrid, OneDGrid, GeometryGrid)
└─GridView (LevelGridView, LeafGridView)
  ├─IndexSet
  ├─Entity (elements, facets, edges, vertices)
  │ └─Geometry (entity to global)
  └─Intersection
    ├─Geometry (intersection to global)
    ├─Entity (inside/outside element/cell)
    └─Geometry (intersection to inside/outside)
```

Concept `Grid` – hierarchy of meshes

```
Grid g;

using ctype = Grid::ctype;
int dim  = Grid::dimension;
// think "surface grid"
int dimw = Grid::dimensionworld;

g.globalRefine(n); // add n levels
assert(g.maxLevel() > 0);
// all coarse/macro entities
auto levelView = g.levelGridView(0);
// all finest/leaf entities
auto leafView = g.leafGridView();
```

Concept `GridView` – one mesh from the hierarchy

```
GridView gv;

using Grid = GridView::Grid;
using ctype = GridView::ctype; // as on Grid
int dim  = GridView::dimension;
int dimw = GridView::dimensionworld;

const Grid &g       = gv.grid();
const auto &idxSet = gv.indexSet();
// count entities...
int n = gv.size(c);  // with codim c
int n = gv.size(gt); // with GeometryType gt

// iterate over entities in gv
for(const auto &elem   : elements(gv)) ...;
for(const auto &facet  : facets   (gv)) ...;
for(const auto &edge   : edges    (gv)) ...;
for(const auto &vertex : vertices(gv)) ...;
// iterate intersections of elem in gv
for(const auto &isect  :
      intersections(gv, elem)) ...;
```

Concept `IndexSet` – numbering within `GridView`s

Entities of different shape (`GeometryType`) are numbered separately. See `MultipleCodimMultipleGeomTypeMapper`.

```
const IndexSet &idxSet;

Entity e; // any codim
int i, c; // number/codim of subentity
idxSet.index(e);          // index of e in gv
idxSet.subIndex(e, i,c); // index of subentity
```

Concept `Entity<codim>` – elements, facets, edges, vertices

```
Entity e;

// all entities: mydim + codim == dim
// elements: codim == 0; facets:   codim == 1
// edges:     mydim == 1; vertices: mydim == 0
int codim = Entity::codimension;
int dim   = Entity::dimension; // as on Grid
int mydim = Entity::mydimension;

GeometryType gt = e.type(); // Shape
// the LevelGridView that e is part of
int l = e.level();

// transform mydimension -> dimensionworld
Entity::Geometry geo = e.geometry();
```

Concept `Intersection` – connectivity between elements

```
Intersection isect;

using ctype = Intersection::ctype;
// local coords  (== Grid::dimension - 1)
int mydim = Intersection::mydimension;
// global coords (== Grid::dimensionworld)
int dimw  = Intersection::dimensionworld;

GeometryType gt = isect.type(); // Shape

// transform intersection -> world
Intersection::Geometry geo = isect.geometry();
Intersection::LocalCoordinate  xl;
Intersection::GlobalCoordinate nu_u, nu_q;
// ||ν_u||_2 = 1,  ν_q := ν_u · geo.integrationElement(xl)
nu_u = isect.unitOuterNormal(xl);
nu_q = isect.integrationOuterNormal(xl);

using Element = Intersection::Entity;
using LGeo    = Intersection::LocalGeometry;

// inside element (always exists)
Element in = isect.inside();
// transform intersection -> inside
LGeo inGeo = isect.geometryInInside();
// index of subfacet of in that contains isect
int inIdx  = isect.indexInInside();

if(isect.neighbor()) { // check outside exists
  Element out = isect.outside();
  LGeo outGeo = isect.geometryInOutside();
  int outIdx  = isect.indexInOutside();
} // otherwise on domain boundary
```

`template<int dim> class YaspGrid;`
Yet Another Structured Parallel Grid

Implements concept `Grid`.

```
#include <dune/grid/yaspgrid.hh>

// construct unit square [0,1]^2 with one element
YaspGrid<2> grid0({ 1, 1 }, { 1, 1 });

// construct cube [-1,1]^3 with 8 = 2^3 elements
YaspGrid<3> grid1({ -1, -1, -1 }, { 1, 1, 1 },
                  { 2, 2, 2 });
```

`template<class GridView> class VTKWriter;`
Generate files for `paraview`

```
#include <dune/grid/io/file/vtk/vtkwriter.hh>

GridView gv;
double f(FieldVector<ctype, dim> xg);

// for multiple possible GeometryTypes use
// MultipleCodimMultipleGeomTypeMapper instead
const auto &set = gv.indexSet();

// interpolate f to piecewise constants
std::vector<double> p0(gv.size(0))
for(const auto &e : elements(gv))
  p0[set.index(e)] = f(e.geometry().center());

// interpolate f to P1/Q1
std::vector<double> p1(gv.size(dim));
for(const auto &v : vertices(gv))
  p1[set.index(v)] = f(v.geometry().center());

// output the two interpolations of f
VTKWriter<GridView> writer(gv);
writer.addCellData(p0, "constants");
writer.addVertexData(p1, "linears");
writer.write("file_name_base");
```

dune-localfunctions

```
LocalFiniteElement
├─LocalBasis
├─LocalInterpolation
└─LocalCoefficients
```

Concept `LocalFiniteElement`

```
using LocalFiniteElement = ...;
LocalFiniteElement lfe;

GeometryType             gt = lfe.type();
unsigned shapeFunctionCount = lfe.size();

const auto &lb = lfe.localBasis();
const auto &li = lfe.localInterpolation();
const auto &lc = lfe.localCoefficients();
```

Concept `LocalBasis` – evaluate shape functions and derivatives

```
using LocalBasis = ...; LocalBasis lb;

unsigned shapeFunctionCount = lb.size();
unsigned p = lb.order(); // max polynom order

using T = LocalBasis::Traits;
unsigned dimD = T::dimDomain;
using DF      = T::DomainFieldType;
using Domain  = T::DomainType;    // DF^dimD
unsigned dimR = T::dimRange;
using RF      = T::RangeFieldType;
using Range   = T::RangeType;     // RF^dimR

Domain xl;                  // x̂
std::vector<Range> y;       // y[i][j]= y_j^(i)
// resizes y to fit; y_j^(i) := φ̂_j^(i)(x̂)  ∀i
lb.evaluateFunction(xl, y);

// only guaranteed for T::diffOrder > 0:
using Jacobian = T::JacobianType; // RF^(dimR×dimD)
std::vector<Jacobian> J; // J[i][j][k]= J_jk^(i)
// resizes J to fit; J_jk^(i) := (∂φ̂_j^(i)/∂x̂_k)|_x̂  ∀i,j,k
lb.evaluateJacobian(xl, J);
// for scalar bases (dimR==1): J[i][0]= ∇̂φ̂^(i)
```

Concept `LocalInterpolation` – interpolate into shape functions

TODO

Concept `LocalCoefficients` – DoF position database

TODO

`class LocalKey;` – DoF position info format

TODO

TODO: list of local finite elements

dune-istl

`BlockVector` `BCRSMatrix`
`MatrixAdapter` `Preconditioner`
list of preconditioners
`Solver` Interface `InverseOperatorResult`
list of solvers