Extending DUNE: The \texttt{dune-xt} modules

Tobias Leibner\textsuperscript{1}, René Milk\textsuperscript{1}, and Felix Schindler\textsuperscript{1}

\textsuperscript{1}Institute for Computational and Applied Mathematics, University of Münster, Orleans-Ring 10, D-48149 Münster

Received: February 6th, 2016; final revision: October 11th, 2016; published: March 6th, 2017.

Abstract: We present our effort to extend and complement the core modules of the Distributed and Unified Numerics Environment DUNE (http://dune-project.org) by a well tested and structured collection of utilities and concepts. We describe key elements of our four modules \texttt{dune-xt-common}, \texttt{dune-xt-grid}, \texttt{dune-xt-la} and \texttt{dune-xt-functions}, which aim at further enabling the programming of generic algorithms within DUNE as well as adding an extra layer of usability and convenience.

1 Introduction

Over the past decades, numerical approximations of solutions of partial differential equations (PDEs) have become increasingly important throughout almost all areas of the natural sciences. At the same time, research efforts in the field of numerical analysis, computer science and approximation theory have led to adaptive algorithms to produce such approximations in an efficient and accurate manner: by now, for a large variety of PDEs advanced approximation techniques are available, such as wavelet methods [27], spectral methods [14], radial basis functions [8] and in particular grid-based discretization techniques such as Finite Volume and continuous or discontinuous Galerkin methods [10].

Focusing on the latter class, there exist a variety of open source and freely available PDE software frameworks with a strong scientific background, such as deal.ii [2], DUNE [4, 3], Feel++ [26], FreeFem++ [16], Fenics [20] and libMesh [19]. We consider the C++-based Distributed and Unified Numerics Environment (DUNE), which consists of the core modules \texttt{dune-common, dune-geometry, dune-grid, dune-istl}\textsuperscript{1} and \texttt{dune-localfunctions}\textsuperscript{1}, complemented by the discretization frameworks \texttt{dune-fem}\textsuperscript{2} [9, 12], \texttt{dune-gdt}\textsuperscript{2} and \texttt{dune-pdelab}\textsuperscript{5}.

DUNE yields highly efficient programs, is fairly well documented and has a large developer and user base with a strong background in numerical analysis and scientific computing. However, since it is mainly targeted at researchers and centered around code efficiency, it has a steep learning curve and lacks some convenience features. In addition, despite enabling flexible numerical

\footnotesize{\textsuperscript{1}All available under http://dune-project.org.}
\footnotesize{\textsuperscript{2}https://github.com/dune-community/dune-gdt}
schemes and providing an exceptionally powerful and generic grid concept, it does not allow for generic algorithms at all times, as detailed further below. To remedy this situation we propose a family of DUNE extension modules.

This paper is organized as follows: Section 2 gives a brief overview of dune-xt and presents our testing framework which is used in all modules. The remaining sections each give an in-depth discussion of one of the DUNE modules which make up dune-xt.

2 The dune-xt project

The dune-xt project aims to provide an extra layer of usability and convenience to existing DUNE modules and at the same time to enable programming of generic algorithms, in particular in the context of discretization frameworks. It consists of the four modules dune-xt-common, dune-xt-grid, dune-xt-la and dune-xt-functions (which we refer to as dune-xt), which are detailed throughout this paper.

We complement our discussion with extensive code listings and examples, which are not necessarily meant to be directly usable (for instance, we shall omit the main() in C++ code and frequently drop the Dune:: namespace qualifier to improve readability). In addition we provide references to code locations, where “dune/foo/bar.hh” denotes the location of a file within the dune-foo module and “dune/xt/foo/bar.hh” denotes the location of a file within the dune-xt-foo module. However, we do not provide individual references for elements of the C++ standard template library, which are prefixed by std::.

Since the main goal of dune-xt is to provide a solid infrastructure, it does not ship examples for the usage of all of its parts. Thus, we often refer to the generic discretization toolbox dune-gdt for examples and motivation.

2.1 Code availability

The dune-xt modules are open source software and freely available on GitHub: https://github.com/dune-community/. They are mainly developed by T. Leibner, R. Milk and F. Schindler with contributions from A. Buhr, S. Girke, S. Kaulmann, B. Verfürth and K. Weber, amounting to roughly 43,000 lines of code at the time of writing. All modules are dual licensed under a BSD 2-Clause License or any GPL-2.0+ License with linking exception, thus being license-compatible to DUNE. We refer to the supplementary material, which contains a snapshot of the dune-xt-super module which bundles together all required dependencies and the dune-xt modules described in this paper.

2.2 Testing

Testing code and exposing it to as many different circumstances as possible is an extremely important part of software development, especially given the templated nature of DUNE and the large amount of provided grid managers. In order to actually take a burden off the developer, tests have to be executed automatically in a reliable manner on each published code change, should be carried out in as many different configurations as possible, and reports of test failure have to be delivered to the developer responsible for the change. In this section we describe the test procedure that is used in dune-xt.

---

3We refer to https://en.cppreference.com/ instead.
4https://github.com/dune-community/dune-gdt
5See Section 2.2 on why we cannot make use of https://gitlab.dune-project.org/.
6http://opensource.org/licenses/BSD-2-Clause
7http://opensource.org/licenses/gpl-license
8http://www.dune-project.org/license.html
9https://github.com/dune-community/dune-xt-super
For testing, we use the Google C++ Testing Framework\(^{10}\) (Gtest) in combination with the DUNE module dune-testtools\(^{11}\), introduced in [18]. Further, we use Travis CI\(^{12}\) to automatically run tests on each push to the code repository.

Gtest is a testing library for C++ code of the xUnit family \(^{22}\). It provides assertion macros that make it very easy to write tests. Assertions can be non-fatal, such that the test will continue to run and output failure information if an assertion fails. This allows for the detection of several faults in one test cycle. Gtest uses fixtures, “recipes” for parametrization, setup and teardown of tests, to make tests run independently of each other. Fixture classes allow to use the same configuration for several different tests and to share code between tests, minimizing the effort needed for writing and maintaining tests. Still, objects for each test are created only for that specific test and not shared between tests, which prevents hard-to-reproduce failures due to interacting tests.

We often want to run the same test for several related (template) classes. For this purpose, Gtest supports type parameterized test fixtures. The desired types have to be collected in the testing::Types struct and passed to a test macro that automatically runs the test for all types. However, constructing a large Types struct with permutations or products of type tuples is inconvenient. Moreover, Gtest currently only supports up to 50 types per test\(^{13}\) which is frequently exceeded in dune-xt. Consider the following class from dune-xt-functions, which represents a scalar-, vector-, or matrix-valued constant function \(f: \mathbb{R}^d \rightarrow \mathbb{R}^{r \times rC}\):

```
1 template <class E, class D, size_t d, class R, size_t r, size_t rC>
2 class ConstantFunction;
```

and suppose we want to test the ConstantFunction class for all combinations of dimensions \(d, r, rC \in \{1, 2, 3\}\) and domain and range field types \(D, R \in \{\text{float, double}\}\). This adds up to \(2^3 \cdot 3^3 = 108\) different specialized ConstantFunction classes that would have to be manually written in the test code. Each time we want to add another field type or higher dimensions, a lot of specializations have to be manually added. In addition, macros are necessary to circumvent the 50 types limit of Gtest complicating the test code even more.

To circumvent these problems, we use the dune-testtools module which provides tools for system testing in DUNE. Among other features, dune-testtools makes it easy to create tests with dynamic and static variations. The configuration is done by providing a meta ini file that contains the possible variations (see Listing 1).

```
[...static]
FUNCTIONTYPE = ConstantFunction<{E}, {D}, {d}, {R}, {r}, {rC}>
```

The expand command tells dune-testtools to create all possible values of the corresponding key, the curved brackets make dune-testtools paste the content of the embraced variable. FUNCTIONTYPE can be used like a preprocessor define in the associated C++ test code. Provided the meta ini file and the C++ test file, dune-testtools creates 108 executables which each test a single specialization of

---

\(^{10}\)https://github.com/google/googletest
\(^{11}\)https://gitlab.dune-project.org/quality/dune-testtools
\(^{12}\)https://travis-ci.org/
\(^{13}\)https://github.com/google/googletest/blob/master/googletest/include/gtest/internal/gtest-type-util.h
Figure 1: Travis integration with Github gives visual feedback for automatic tests (failing: left, succeeding: right), in particular to keep the workload for library developers at a minimum when handling code changes and contributions.

ConstrantFunction. Adding another field type or higher dimensions is as simple as adding it to the meta ini file without modifying the test code at all. In addition, dynamic variations, i.e. parameters that are provided at runtime and not at compile time, can easily be added in a similar way, see [18].

Besides the reduced code complexity, this setup also makes it easier to debug tests as one can see at a glance which types are failing and debug only the associated tests. Creating anexecutable per tested type instead of testing all types in one executable using the Types struct greatly reduces memory consumption during compilation. As a downside we see increased total compile times. Depending on the available build infrastructure this can be offset by the greater available parallelism in building the test, resulting in more but smaller binaries.

Writing the tests is only the first step. Even the most comprehensive test suite is useless without getting regularly executed. We therefore continuously test each module on every update of the module’s code repository using the Travis CI infrastructure (Travis)\(^\text{14}\), whose use is free of charge for open source projects. Relying on Google’s Compute Engine\(^\text{15}\) and following the setup specified in the module’s .travis.yml file Travis provides us with a highly flexible environment to run our test suites in. Using Travis’ “build matrix”\(^\text{16}\) feature our test suites are run with a diverse setup of available DUNE modules, from the minimal set of hard dependencies specified in each dune.module file to the full set of suggested ones, as well as different compilers (currently gcc 4.9 and 5.3 and clang 3.7). Travis offers tight integration with Github\(^\text{17}\) pull requests (see Figure 1) and branches, each of which gets automatically tested whenever new commits get added, with visual on-page and email feedback. This is one of the main reasons why we host our modules on Github. In our opinion the prompt and automatic feedback on the validity of code contributions, with reasonably high coverage, is of enormous value to the development process.

3 The dune-xt-common module

Just like dune-common, we provide a common infrastructure in dune-xt-common with minimal dependencies, which is used throughout the other modules.

3.1 Improved handling of dense containers

dune-common contains the FieldVector\(^\text{18}\) and FieldMatrix\(^\text{19}\) classes, which model small dense vectors and matrices of fixed size (in particular used for coordinates, affine reference maps and function evaluations). These containers are implemented to provide maximum performance, but lack some convenience features. In particular the lack of certain operators and constructors make it difficult

\(^\text{14}\)https://travis-ci.org/dune-community
\(^\text{15}\)https://cloud.google.com/compute/
\(^\text{16}\)https://docs.travis-ci.com/user/customizing-the-build/#Build-Matrix
\(^\text{17}\)https://github.com
\(^\text{18}\)dune/common/fvector.hh
\(^\text{19}\)dune/common/fmatrix.hh
to write generic code. Consider, for instance, a generic string conversion utility (assuming $T$ is a matrix type):

```cpp
template <class T>
static inline T from_string(const std::string ss, 
const size_t rows = 0, 
const size_t cols = 0)
{
    T result(rows, cols); // <- does not compile for FieldMatrix
    // ... fill result from ss
    return result;
}
```

The above example does compile if $T$ is a DynamicMatrix, but not if $T$ is a FieldMatrix, which makes it extremely difficult to write generic code. And while it is clear that constructing a FieldMatrix of fixed size $M \times N$ is not sensible for other values of rows and cols, the construction in line 6 should be possible for rows = $M$ and cols = $N$ (throwing an appropriate exception otherwise).

In order to allow for generic algorithms we provide templated VectorAbstraction and MatrixAbstraction classes in dune/xt/common/{matrix,vector}.hh (along with specialization for all sensible vector and matrix classes), which allow for generic creation of and access to matrices and vectors. Additionally, we provide is_vector and is_matrix traits, which allow to rewrite the above example:

```cpp
#include <dune/xt/common/matrix.hh>
#include <dune/xt/common/type_traits.hh>

using namespace Dune::XT::Common;

template <class T>
static inline typename std::enable_if<is_matrix<T>::value , T>::value 
from_string(const std::string ss, 
const size_t rows = 0, const size_t cols = 0)
{
    auto result = MatrixAbstraction<T>::create(rows, cols);
    // ... fill result from ss using MatrixAbstraction< T >::set_entry(...) 
    return result;
}
```

It is thus possible to use from_string with matrices of different type:

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

auto fmat = from_string<Dune::FieldMatrix<double, 2, 2>>("[1. 2.; 3. 4.]");
auto dmat = from_string<Dune::DynamicMatrix<double>>("[1. 2.; 3. 4.]");
```

In particular, one can use it with any custom matrix implementation by providing a specialization of MatrixAbstraction within the user code:

```cpp
class CustomMatrix { /* user provided matrix implementation */ }; 
struct MatrixAbstraction<CustomMatrix> { /* implement specialization */ }; 
auto cmat = from_string<CustomMatrix>("[1. 2.; 3. 4.");
```

---

$dune/common/dynmatrix.hh$

© by the authors, 2017

Archive of Numerical Software 5(1), 2017
Based on these abstractions we provide many generic implementations in dune-xt. For instance, we provide an extension of the floatCmp\(^{21}\) mechanism from dune-common (see Section 3.4) for any combination of vectors (which allows for the same syntax as its counterpart in dune-common, see Section 3.4):

\begin{verbatim}
C++ code
#include <dune/common/dynvector.hh>
#include <dune/xt/common/float_cmp.hh>

std::vector<double> svector({1., 1.});
Dune::DynamicVector<double> dvector(2, 1.);

Dune::XT::Common::FloatCmp::eq(svector, dvector);
\end{verbatim}

### 3.2 Improved string handling and Configuration

As already hinted at above, we provide the string conversion utilities

\begin{verbatim}
C++ code
#include <dune/xt/common/string.hh>

template<typename T>
static inline T from_string(const std::string ss,
                            const size_t size = 0,
                            const size_t cols = 0);

template<typename T>
static inline std::string to_string(const T &ss);
\end{verbatim}

in dune/xt/common/string.hh. These can be used with any basic type as well as with all matrices and vectors supported by the abstractions from Section 3.1. We use standard notation for vectors ("[1 2]") and matrices ("[1 2; 3 4]"), see the previous paragraph for examples. The from_string function takes optional arguments which determine the size of the resulting container (for containers of dynamic size), where 0 means automatic detection (the default).

Based on these string conversion utilities, we provide an extension of dune-common’s ParameterTree\(^{22}\) in dune/xt/common/configuration.hh: the Configuration class. The Configuration is derived from ParameterTree and can thus be used in all places where a ParameterTree is expected. While it also adds an additional layer of checks (in particular regarding provided defaults), report and serialization facilities, one of its main features is to allow the user to extract any type that is supported by the string conversion facilities. Given a sample configuration file in .ini format (for example the default configuration required to create a cubic grid using the factory methods discussed below),

\begin{verbatim}
Code Listing 2: Contents of the cube_gridprovider_default_config() in dune-xt-grid
1 2 3 4 5 6
type = xt.grid.gridprovider.cube
lower_left = [0 0 0 0]
upper_right = [1 1 1 1]
num_elements = [8 8 8 8]
num_refinements = 0
overlap = [1 1 1 1]
\end{verbatim}

we can query the resulting Configuration object config for the types supported by the ParameterTree,

\begin{verbatim}
C++ code
auto num_refinements = config.get<int>("num_refinements");
\end{verbatim}

as well as for all types supported by our string conversion utilities (including custom matrix and vector types as explained in the previous paragraph):

\(^{21}\)dune/common/float_cmp.hh
\(^{22}\)dune/common/parametertree.hh
Extending DUNE: dune-xt

The above is valid for all $0 \leq \text{dimDomain} \leq 4$, due to the automatic size detection of `from_string`.

### 3.3 Timings

Getting information about how long certain portions of an application take to run is crucial to guide optimization efforts. It can also be useful to estimate the entire loop execution time from the first couple of loops and of course for benchmarking parts of or an entire application for different parameters. Mature and generally simple to use tools exist, both commercial (e.g., Intel VTune\textsuperscript{23}, Allinea Map\textsuperscript{24}) and free (e.g., VagrdinCallgrind\textsuperscript{25}, Oprofile\textsuperscript{26}), that allow very fine-grained performance analysis of a given code, down to function or even instruction level. It is however very hard or impossible to measure custom, arbitrary sections of code with these tools and to present those measurements in an easily understood format. The dune-common module provides a straightforward implementation of a Timer\textsuperscript{27} class that relies on `std::system_clock` to measure expired walltime. This is enough for simple needs, but we extend on this concept with the Timings and related classes in dune/xt/common/timings.hh, because measuring user and system time can be greatly insightful and having a centralized managing facility with its own output capabilities is much easier for the user. The global Timings instance keeps a map of named sections and associated timingData objects. The user can start/stop sections manually or use a guard object to start a timing at object construction and stop it when the guard object goes out of scope:

#### Code Listing 3: Timing example

```cpp
auto lower_left = config.get<FieldVector<double, dimDomain>>("lower_left");
```

Internally TimingData expands on Timer by measuring wall, system and user time between start and stop concurrently using a boost::timer::cpu_timer\textsuperscript{28}. Timings has a couple of output_\* functions that all write data in the Comma Separated Values (CSV) format and which distinguish themselves from one another in which measures are produced and whether averages, minima and maxima are calculated over MPI-ranks. We chose the CSV format for our output because it enables post processing and analysis by a wide variety of tools, e.g. Pandas\textsuperscript{21} or matplotlib\textsuperscript{17}.

#### Code Listing 4: example.csv

```text
threads,ranks,sec_avg_usr,sec_max_usr,sec_avg_wall,sec_max_wall,sec_avg_sys,sec_max_sys,sec_inner_avg_usr,sec_inner_max_usr,sec_inner_avg_wall,sec_inner_max_wall,sec_inner_avg_sys,sec_inner_max_sys
8,1,2050,2050,2050,2050,0,0,2050,2050,2050,2050,0,0
```

\textsuperscript{23}https://software.intel.com/en-us/intel-vtune-amplifier-xe

\textsuperscript{24}http://www.allinea.com/products/map

\textsuperscript{25}http://valgrind.org/info/tools.html#callgrind

\textsuperscript{26}http://oprofile.sourceforge.net/

\textsuperscript{27}dune/common/timer.hh

\textsuperscript{28}http://www.boost.org/doc/libs/1_58_0/libs/timer/doc/cpu_timers.html#Class-cpu_timer

© by the authors, 2017

Archive of Numerical Software 5(1), 2017
3.4 Extending Float Comparison

Floating point operations are not always exact, so using `operator==` to compare primitive floating point types may yield unwanted results. To avoid this problem, `dune-common` provides functions `eq`, `ne`, `gt`, `lt`, `ge` and `le` in the namespace `Dune::FloatCmp` to approximately compare floating point numbers. There are three variants of approximate comparisons (`absolute`, `relativeWeak` and `relativeStrong`) implemented in `dune-common`. The `absolute` compare style checks two floating point numbers `a` and `b` for equality by

\[ |a - b| \leq \epsilon_{\text{abs}}, \]

where \(\epsilon_{\text{abs}}\) is a specified tolerance parameter. This works well for numbers that are neither too small nor too large, but two numbers with absolute values lesser than \(\epsilon_{\text{abs}}\) will always be specified as equal while for large numbers the gap between two adjacent floats may become greater than \(\epsilon_{\text{abs}}\) rendering the result the same as with `==`. The compare styles `relativeWeak`

\[ |a - b| \leq \epsilon_{\text{rel}} \max(|a|, |b|) \]

and `relativeStrong`

\[ |a - b| \leq \epsilon_{\text{rel}} \min(|a|, |b|) \]

use a tolerance parameter that is scaled with the numbers and thus are suitable for a wide range of numbers. However, problems occur if one of the numbers is zero. Thus, we provide a fourth compare style `numpy` in `dune/xt/common/float_cmp.hh` that is equivalent to the implementation in `numpy.isclose`\(^\text{29}\). Here, both an absolute tolerance \(\epsilon_{\text{abs}}\) and a relative tolerance \(\epsilon_{\text{rel}}\) are chosen and the numbers `a`, `b` are considered equal if

\[ |a - b| \leq \epsilon_{\text{abs}} + \epsilon_{\text{rel}} |b|. \]

This allows for an absolute comparison (with a sufficiently small \(\epsilon_{\text{abs}}\) near 0) and a relative comparison elsewhere and is thus the default compare style in `dune-xt-common`. Note that this comparison is not symmetric with respect to `a`, `b`, so `XT::Common::FloatCmp::eq(a, b)` may not be the same as `XT::Common::FloatCmp::eq(b, a)` in some cases. The `absolute`, `relativeWeak` and `relativeStrong` compare styles are also supported by `dune-xt-common` and can be chosen by providing a template parameter to the comparison functions.

Often we rather want to compare vectors of floating point numbers instead of scalar values. This could be done by looping over the components of the vectors and comparing them one by one, which is inconvenient for the user. `dune-xt-common` has built-in support for vector comparisons, i.e. one can compare any two vectors as long as there are `VectorAbstractions` (see Section 3.1) available for both vector types. In particular, `std::complex` and vectors containing `std::complex` values are supported.

4 The dune-xt-grid module

The dune-xt-grid module builds on `dune-grid` and `dune-xt-common` to provide powerful concepts to improve performance (such as the `EntityInlevelSearch` and the `Walker`) or to allow for generic algorithms (such as the `GridProvider`).

4.1 Searching the grid

A typical task within a PDE solver is to prolong discrete function data from one discrete function space onto another one (that is usually associated with a finer grid). Without a direct, one-to-one identification mapping of degrees of freedom (DoF) available (think of different discrete functions stemming from `dune-fem` and `dune-pdelab`), this requires identifying those entities in the source space where the source function should be evaluated in quadrature points defined

\(^\text{29}\)http://docs.scipy.org/doc/numpy-dev/reference/generated/numpy.isclose.html
with respect to the range space. With the `EntityInlevelSearch` we provide a means to that end in
`dune/xt/grid/search.hh` for arbitrary grids, as long as the range grid covers a subset of the physical
domain of the source grid.

```
C++ code

template <class GridViewType, int codim = 0>
class EntityInlevelSearch : public EntitySearchBase<GridViewType>
{
  // not all types shown
  public:
    EntityInlevelSearch(const GridViewType& gridview)
      : gridview_(gridview), it_last_(gridview_.template begin<codim>())
    {}
    
    template <class PointContainerType>
    EntityVectorType operator()(const PointContainerType& points);
  
  private:
    const GridViewType gridview_; 
    IteratorType it_last_; 
};
```

It is modeled after the `HierarchicSearch` in `dune-grid`, with the extension that instead of finding
one `Entity` for one point at a time we allow searching for a sequence of points (via `operator()`)
and return a sequence of entities accordingly. As another improvement the implementation
remembers the last search position on the source grid across search calls. This greatly improves
the performance over the naïve implementation of restarting the search for every `operator()` in case
of equivalent grids.

### 4.2 Providing generic access to grid views and grid parts

There exists an unfortunate disagreement between `dune-grid` and `dune-fem`, whether grid views
or grid parts are to be used to model a collection of grid elements, which makes it hard to implement
generic algorithms. Think of some code which needs to create a `LevelGridView` or `LevelGridPart`,
depending on the further use for a discrete function space implemented via `dune-pdelab` or `dune-fem`:

```
C++ code

#include <dune/fem/gridpart/levelgridpart.hh>

template <class GridType>
void create_level(GridType& grid, const int lv) {
  // either ?
  auto level_view = grid.levelGridView(lv);
  // or ?
  Dune::Fem::LevelGridPart<GridType> level_part(grid, lv);
  // ...
}
```

In `dune/xt/grid/gridprovider.hh` we thus provide the `GridProvider`, the purpose of which is to hold a
the grid object and to allow for a generic creation of grid views and grid parts to ultimately allow
for generic code despite the above mentioned disagreement:

```
C++ code

template <class GridType>
class GridProvider
{
  // not all methods and types shown ...

  #dune/grid/utility/hierarchicsearch.hh
```

© by the authors, 2017  Archive of Numerical Software 5(1), 2017
public:
  std::shared_ptr/GridType& grid_ptr();
  GridType& grid();

  template <Layers layer_type, Backends backend>
  typename Layer[layer_type, backend>::Type layer(const int lv = 0);
};

Together with the Layers and Backends enum classes\(^{31}\), this allows to write generic code by passing a grid provider along with the required tag:

```cpp
define <dune/xt/grid/provider.hh>
using namespace Dune::XT::Grid;

template <class GridType, Backends backend>
define create_level(GridProvider<GridType>& grid_provider, const int lv) {
  auto level_part_or_view = grid_provider.layer<Layers::level, backend>(lv);
  // ...
}
```

We provide several factory methods to create a GridProvider (for instance make_cube_grid, which creates grids of rectangular domains and make_dgf_grid and make_gmsh_grid, which either create grids from the respective definition file). In addition, we provide a means to select a factory at runtime using the GridProviderFactory struct, given the type of the grid \(G\) and a configuration object \(config\) (for instance using the default one for make_cube_grid, see Listing 2),

```cpp
define include <dune/xt/gridprovider.hh>
#define using namespace Dune::XT::Grid;

auto grid_provider = GridProviderFactory\<G\>::create("xt.grid.gridprovider.cube", config);
```

which is particularly useful in conjunction with configuration files or Python bindings.

### 4.3 Identification of domain boundaries

PDE solvers need to prescribe the solution’s behaviour on the boundary of the computational domain. In general, there may exist a large number of different boundary conditions in any given problem, and therefore any PDE solver needs to provide a way to identify parts of that domain boundary and categorize them. Unfortunately, dune-grid does not provide such a mechanism.

This problem is tackled in dune-pdelab, for instance, by deriving from Dune::TypeTree::LeafNode\(^{32}\) and implementing the following methods:

```cpp
virtual bool isDirichlet(const IntersectionGeometryType& intersection_geometry, 
 const FieldVector<D, d - 1>& coord) const;

virtual bool isNeumann(const IntersectionGeometryType& intersection_geometry, 
 const FieldVector<D, d - 1>& coord) const;
```

\(^{31}\)dune/xt/grid/layers.hh

\(^{32}\)dune/typetree/leafnode.hh
This approach is however limited to the boundary types defined in dune-pdelab and the user has no means to extend this approach to other boundary type (one cannot just add a isCustomBoundary() method to the interface).

We therefore propose a more flexible mechanism that is based on mapping the boundary category to types derived from BoundaryType.

```
c++ code

```class BoundaryType
{
    protected:
        virtual std::string id() const = 0;
    public:
        virtual bool operator==(const BoundaryType& other) const
        {
            return id() == other.id();
        }
    }

class DirichletBoundary : public BoundaryType
{
    protected:
        virtual std::string id() const override final
        {
            return "dirichlet boundary";
        }
}
class NeumannBoundary : public BoundaryType { /*...*/};
class RobinBoundary : public BoundaryType { /*...*/};
```

For a given grid view (and its type of intersection), access to the type of the boundary is granted by the BoundaryInfo class concept

```
c++ code

```template <class IntersectionType>
class BoundaryInfo
{
    public:
        virtual BoundaryType& type(const IntersectionType& intersection) const = 0;
};
```

based on which a PDE solver library (such as dune-gdt) can provide generic algorithms which act only on parts of the domain boundary by checking

```
c++ code

```if (boundary_info.type(intersection) == DirichletBoundary()) {
    // ...
} else if (boundary_info.type(intersection) == NeumannBoundary()) {
    // ....
} else
    // ...
```

In addition, this concept can be easily extended by the user by simply deriving from BoundaryType. As implementations of the BoundaryInfo concept, we currently provide:

- AllDirichlet and AllNeumann, the purpose of which is self-explanatory.

---

33dune/xt/grid/boundaryinfo/types.hh

© by the authors, 2017 Archive of Numerical Software 5(1), 2017
• NormalBased, which allows to identify boundary intersections by the direction of their outward pointing normal.

In order to allow for problem definition classes to define domain boundaries independently of the type of the grid, we also provide the `BoundaryInfoFactory`. User classes can hold a complete description of one of the boundary infos from above in a `Configuration` instance. Given the type of the grid (and thus the type of an intersection), one is then able to create an instance of one of the implementations of the `BoundaryInfo` concept of correct type, as required:

```cpp
#include <dune/grid/yaspgrid.hh>
#include <dune/grid/sgrid.hh>
#include <dune/xt/common/configuration.hh>
#include <dune/xt/grid/boundaryinfo.hh>

using namespace Dune::XT::Common;
using namespace Dune::XT::Grid;

class Problem {
public:
    Configuration boundary_info_cfg() {
        Configuration config;
        config["type"] = "xt.grid.boundaryinfo.normalbased";
        config["default"] = "dirichlet";
        config["neumann.0"] = "[ 1. 0.]";
        config["neumann.1"] = "[-1. 0.]";
        return config;
    }
};

typedef typename Dune::YaspGrid<2>::LeafIntersection YI;
typedef typename Dune::SGrid<2, 2>::LeafIntersection SI;

Problem problem;
auto boundary_info_y = BoundaryInfoFactory<YI>::create(problem.boundary_info_cfg());
auto boundary_info_s = BoundaryInfoFactory<SI>::create(problem.boundary_info_cfg());
```

The type of `boundary_info_y`, for instance, is `std::unique_ptr<BoundaryInfo<YI>>`. Given a rectangular domain in $\mathbb{R}^2$, it models a Neumann boundary left and right and a Dirichlet boundary everywhere else.

### 4.4 Periodic Gridviews

Periodic boundary conditions are frequently used to model an infinite domain. To apply periodic boundary conditions in the context of Finite Volume methods, intersections on the periodic boundary have to be linked to the intersection on the opposite side of the (finite) grid. For Finite Element methods, the indices of connected degrees of freedom (DoFs) on the boundary have to be identified and suitable constraints applied. In any case, the implementation can be quite time-consuming and cumbersome if there is no support from the underlying grid manager.

Support for periodic boundary conditions in DUNE is varying from grid to grid. `YaspGrid` supports periodic boundaries using the parallel DUNE grid interface which requires the user to run a parallel program. Several grid managers including `dune-alugrid` [1] and `SPGrid` [25] support periodicity by gluing together edges of the unit cube [25]. While this allows to obtain the neighboring entity of an intersection on the periodic boundary, DoFs still have to be modified by hand.

Regarding the uneven support for periodic boundary conditions in DUNE, we provide the `PeriodicGridView` class in `dune/xt/grid/periodic_gridview.hh` which is derived from a given arbitrary `Dune::GridView`. As long as the GridView models an axis-parallel hyperrectangle with conforming
Extending DUNE: dune-xt

faces, it is enough to replace a GridView by the corresponding PeriodicGridView in existing code to apply periodic boundary conditions (see Listing 5). This is internally achieved by replacing the IndexSet and iterators and the corresponding methods (begin, end, ibegin, iend, size and indexSet) of the GridView by periodic variants, utilizing the search capabilities from Section 4.1. The remaining methods are forwarded to the underlying GridView. Periodic directions can be specified by supplying a std::bitset (see Listing 5). By default, all directions are made periodic.

The PeriodicIntersectionIterator that can be obtained by the ibegin and iend methods returns a PeriodicIntersection when dereferenced. The PeriodicIntersection behaves exactly like the non-periodic Intersection except that it returns neighbor()== true and an outside() entity on the periodic boundary. The outside() entity is the entity adjacent to the periodically equivalent intersection, i.e., the intersection at the same position on the opposite side of the domain. The indexSet() method returns a PeriodicIndexSet which assigns the same index to entities that are periodically equivalent. Hence, the PeriodicIndexSet is usually smaller than the corresponding IndexSet. The begin and end methods of PeriodicGridView return a PeriodicIterator which visits only one of several periodically equivalent entities.

Note that there existed a similar but independently developed class PeriodicGridPart in dune-fem that contained a periodic index set but no periodic intersections. Unfortunately, the PeriodicGridPart was removed in the dune-fem release 1.2.

4.5 Walking the grid

Since we are considering grid-based numerical methods, we frequently need to iterate over the elements $t \in \tau_h$ of a grid view. In order to minimize the amount of required grid iterations, we want to be able to carry out $N$ operations on each grid element, as opposed to walking the entire grid $N$ times. We thus provide in dune/xt/grid/walker.hh the templated walker class, working with any GridView (from dune-grid) or GridPart (from dune-fem):

```cpp
template <class GridViewType>
class Walker
{
    // not all methods and types shown ...

public:
    void add(Functor::Codim0<GridViewType>& functor/*, ... */);
    void add(Functor::Codim1<GridViewType>& functor/*, ... */);
    void add(Functor::Codim0And1<GridViewType>& functor/*, ... */);
    void walk(const bool use_tbb = false);
};
```

The user can add an arbitrary amount of functors to the Walker, all of which are then locally executed on each grid element. Each functor is derived from one of the virtual interfaces Functor::Codim0, Functor::Codim1 or Functor::Codim0And1, for instance

```cpp
template <class GridViewType>
class Codim0
{
    typedef typename XT::Grid::Entity<GridViewType>::Type EntityType;
};
```
where `prepare` (and `finalize`) are called before (and after) iterating over the grid, while `apply_local` is called on each entity of the grid. The user might note here the usage of traits from `dune/grid/entity,intersection.hh` to extract required information from a `GridView` or `GridPart` in a generic way. Each `add` method of the `Walker` accepts an additional argument which allows to select the elements and faces the functor will be applied on. For instance, in the context of a discontinuous Galerkin discretization in `dune-gdt` we want to apply local coupling operators on all inner faces of the grid and local boundary operators on all Dirichlet faces of the grid. Presuming we were given suitable implementations of these local operators as functors and a `BoundaryInfo` object in the sense of the previous paragraph, the following would realize just that:

```
#include <dune/xt/grid/walker.hh>
using namespace Dune::XT::Grid;
Walker<GV> walker(grid_view);
walker.add(coupling_operator, new ApplyOn::InnerIntersectionsPrimally<GV>());
walker.add(boundary_operator, new ApplyOn::DirichletIntersections<GV>(boundary_info));
// add more, if required...
walker.walk();
```

The use of the `new` keyword will not create memory leaks here as the pointer is wrapped into a `std::unique_ptr` in the `add` method of the walker. Note that the `walk` method allows to switch between a serial and a shared memory parallel iteration over the grid, at runtime (via the `use_tbb` switch). In particular, the user only has to provide implementations of the functors and need not deal with any parallelization issues (or different types of grid walkers, depending on the parallelization paradigm).

### 5 The `dune-xt-la` module

As discussed (Section 3.1), `dune-common` provides small dense vectors and matrices which are, e.g., used for coordinates. In addition, any PDE solver requires large vectors and (usually sparse) matrices to represent assembled functionals and operators stemming from the underlying PDE. Linear algebra containers are a performance critical aspect of any discretization framework and one usually does not want to bet on a single horse: there exists external backends which are well suited for serial and shared memory parallel computations (such as `EIGEN`[15]) while others are more suited for distributed memory parallel computations (such as `dune-istl`). Neither is fitting for every purpose and we thus require a means to exchange the implementation of matrices and vectors depending on the circumstances. This calls for abstract interfaces for containers and linear solvers, which we provide within the `dune-xt-la` module.

#### 5.1 Generic linear algebra containers

We chose a combination of static and dynamic inheritance for these interfaces, allowing for virtual function calls that act on the whole container (such as `scal`) while using the “Curiously recurring template patterns” (CRTP, see [11]) paradigm for methods that are called frequently (such as access to individual elements in loops), allowing the compiler to optimize performance critical calls (for instance by inlining). We provide a thread-safe helper class `CRTPInterface` in `dune/xt/common/crtp.hh` along with thread-safe `CHECK_...` macros for debugging, as the tools provided in `dune/common/bartonmannifcheck.hh` may not work properly in a shared memory parallel program.
All matrices and vectors are derived from `ContainerInterface`:

```
C++ code

// not all methods and types shown ...
public:
  typedef typename Traits::derived_type derived_type;

// Sample CRTP implementation:
inline void scal(const ScalarType& alpha)
{
  as_imp() from CRTPInterface performs a static_cast into derived_type.
  Thus, scal() of the derived class is called.
  CHECK_AND_CALL_CRTP(this->as_imp().scal(alpha));
}
inline void axpy(const ScalarType& alpha, const derived_type& xx);
inline derived_type copy() const;

// Default implementation:
// could be overridden by any derived class.
virtual derived_type& operator*=(const ScalarType& alpha)
{
  scal(alpha);
  return this->as_imp();
}
```

The `ContainerInterface` enforces just enough functionality to assemble a linear combination of matrices or vectors, which is frequently required in the context of model reduction (compare [24]). Given an affine decomposition of a parametric matrix or vector $B$, we need to assemble \[ \sum_{q=0}^{t-1} \theta_q B_q \] for given component containers $B_q$ and scalar coefficients $\theta_q$. The following generic code will work for any matrix or vector type $C$ derived from `ContainerInterface`:

```cpp
#include <dune/xt/la/container/container-interface.hh>
using namespace Dune::XT::LA;

template <class C>
typename std::enable_if<is_container<C>::value, C>::type assemble_lincomb(const std::vector<C>& components, const std::vector<double>& coefficients)
{
  auto result = components[0].copy();
  result *= coefficients[0];
  for (size_t qq = 1; qq < components.size(); ++qq)
    result.axpy(coefficients[qq], components[qq]);
  return result;
}
```

All containers in `dune-xt-la` are implemented with “copy-on-write” (COW) [23, 190-194]. Implementing data sharing among entities with data duplication only occurring if an entity tries to modify the referenced data is a well established and common technique in Computer Science. This pattern (also sometimes denoted as “lazy copy”) typically incurs minimal runtime overhead for the required reference counting for the great benefit of being able to pass around copies of an entity without immediate expensive memory copies. Since all container implementations in `dune-xt-la` are proxy classes that forward operations to the respective backend instances, we

---

34 `dune/xt/la/container/container-interface.hh`

35 Note the use of the `is_container` traits in line 6 that we provide in `dune/la/container/container-interface.hh`. Together with `std::enable_if`, this checks that $C$ is derived from `ContainerInterface` at compile time.
insert the COW logic into the backend call, which is then used throughout the class to access the backend (in addition to allowing the user to directly access the underlying backend):

```
C++ code
1 BackendType& backend()
2 {
3    ensure_uniqueness();
4    return *backend_;  // backend_
5 }
```

The backend_ is simply a std::shared_ptr<BackendType> on which the ensure_uniqueness method can query the current status by a call to unique and perform the deep copy, if required. Note that due to COW and move semantics for all matrices and vectors in dune-xt-la, the only deep copy in Listing 5.1 is actually done in line 11 (neither in line 10 nor in line 14).

Based on ContainerInterface we provide the VectorInterface\(^\text{36}\) for dense vectors and the MatrixInterface\(^\text{37}\) for dense and sparse matrices. Each derived vector class has to implement the methods size, add_to_entry, set_entry and get_entry_ref, which allow to access and change individual entries of the vector. The interface provides default implementations for all relevant mathematical operators, support for range-based for loops and many useful methods, such as dot, mean, standard_deviation and l2_norm, just to name a few.

Each derived matrix class has to implement rows, cols, add_to_entry, set_entry and get_entry to allow for access to individual entries, mw for matrix/vector multiplication and clear_row, clear_col, unit_row and unit_col, which are required in the context of solving PDEs with Dirichlet Constraints or pure Neumann problems. Every matrix implementation (even a dense one) is constructible from the same type of sparsity pattern (which we provide in dune/xt/la/container/pattern.hh) and the access methods ..._entry are only required to work on entries that are contained in the pattern. The interface provides several mathematical operators, norms and a means to obtained a pruned matrix (where all entries close to zero are removed from the pattern).

We also provide several vector and matrix implementations:

- The CommonDenseVector and CommonDenseMatrix in dune/xt/la/container/common.hh, based on the DynamicVector and DynamicMatrix from dune-common. These are always available.

- The EigenDenseVector, EigenMappedDenseVector, EigenDenseMatrix and EigenRowMajorSparseMatrix in dune/xt/la/container/eigen.hh, based on the EIGEN package (if available). The EigenMappedDenseVector allows to wrap an existing double* array and the EigenRowMajorSparseMatrix allows to wrap existing matrices in standard CSR format, which allows to wrap other container implementation (for instance in the context of Python bindings).

- The IstlDenseVector and IstlRowMajorSparseMatrix in dune/xt/la/container/istl.hh, based on dune-istl (if available).

To allow for generic algorithms we also provide the Backends enum class along with the default_backend, default_sparse_backend and default_dense_backend defines (which are set depending on the build configuration). These can be used in other libraries and user code to obtain suitable containers independently of the current build configuration.

Consider, for instance, an \(L^2\) projection of a function \(f : \Omega \to \mathbb{R}\) onto a discontinuous Galerkin discrete function space defined on a grid view \(\tau_h\) modeling \(\Omega\). Given local basis functions \(\psi^i_j\),

\(^\text{36}\)dune/xt/la/container/vector-interface.hh
\(^\text{37}\)dune/xt/la/container/matrix-interface.hh
Extending DUNE:  
dune-xt

0 ≤ i < l of such a space on each entity t ∈ τₜ, the local DoF vector \( f^{l}_h \) of the projected discrete function is given as the solution of the linear system

\[
L^{21}_t \cdot f^{l}_h = l^{t}_h \quad \text{with} \quad (L^{21}_t)_{ij} := \int_t \varphi_i \varphi_j^t \quad \text{and} \quad (l^t_h) := \int_t f \varphi_j^t. \tag{1}
\]

The problem of assembling and solving (1) is a typical situation within any PDE solver. In dune-gdt, for instance, (1) is roughly assembled as follows, given a local basis and an appropriate quadrature:

```cpp
#include <dune/xt/la/container.hh>

using namespace Dune::XT::LA;

typedef typename Container<double, default_dense_backend>::MatrixType LocalMatrixType;

typedef typename Container<double, default_dense_backend>::VectorType LocalVectorType;

// ... on each grid element
LocalMatrixType local_matrix(local_basis.size(), local_basis.size(), 0.);
LocalVectorType local_vector(local_basis.size(), 0.);
LocalVectorType local_DoFs(local_basis.size(), 0.);

// ... at each quadrature point, given evaluations of the local basis as basis_values
// and of the source function as source_value
for (size_t ii = 0; ii < local_basis.size(); ++ii) {
    local_vector[ii] += integration_element * quadrature_weight
                         * (source_value * basis_values[ii]);
    for (size_t jj = 0; jj < local_basis.size(); ++jj) {
        local_matrix.add_to_entry(ii, jj,
                                   integration_element * quadrature_weight
                                   * (basis_values[ii] * basis_values[jj]));
    }
}
```

Note that we neither have to manually specify the correctly matching matrix and vector types nor to include the correct headers (which depend on the current build configuration). The Container traits together with any of the default... defines always yields appropriate available types (lines 1, 5, 6).

### 5.2 Runtime selectable solvers

In order to determine the local DoF vector in the above example, we need to solve the algebraic problem: find \( \text{local}_\text{DoFs} \), such that

\[
\text{local}_\text{matrix} \cdot \text{local}_\text{DoFs} = \text{local}_\text{vector}.
\]

In addition to such small, dense problems we also require the inversion of large (sparse) system and product matrices. For interesting large and real-world problems, however, there are few linear solvers available which can be used as a black box (if at all). Most problems require a careful choice and detailed configuration of the correct linear solver. We thus require access to linear solvers which can be used in a generic way but also exchanged and configured at runtime.

In dune/xt/la/solver.hh we provide such solvers via the solver class:

```cpp
template <class MatrixType>
class Solver {
    // simplified variant
    public:
```
We provide specializations of the solver class for all matrix implementations derived from MatrixInterface (see the previous paragraph). To continue the example from the previous paragraph, this allows to determine the local DoF vector of the $L^2$ projection:

```
#include <dune/xt/common/exceptions.hh>
#include <dune/xt/la/solver.hh>

try {
    XT::LA::Solver<LocalMatrixType>(local_matrix).apply(local_vector , local_DoFs);
} catch (XT::Exceptions::linear_solver_failed& ee) {
    DUNE_THROW(Exceptions::projection_error ,
               "L2 projection failed because a local matrix could not be inverted!\n\n" << "This was the original error: " << ee.what());
}
```

The above example shows a typical situation within the library code of dune-gdt: we need to solve a small dense system for provided matrices and vectors of unknown type. We can do so by instantiating a solver and calling the black-box variant of apply (line 5). This apply variant is default implemented by calling

```
apply(rhs , solution , types()[0]);
```

where types() always returns a (non-empty) list of available linear solvers for the given matrix type, in descending priority (meaning the first is supposed to “work best”). For instance, if local_matrix was an EigenDenseMatrix, a call to types() would reveal the following available linear solvers:

```
{"lu.partialpivot", "qr.householder", "llt", "ldlt", "qr.colpivhouseholder",
 "qr.fullpivhouseholder", "lu.fullpiv"}
```

On the other hand, if the matrix was an EigenRowMajorSparseMatrix, a call to types() would yield

```
{"bicgstab.ilut", "lu.sparse", "llt.simplicial", "ldlt.simplicial",
 "bicgstab.diagonal", "bicgstab.identity", "qr.sparse",
 "cg.diagonal.lower", "cg.diagonal.upper", "cg.identity.lower",
 "cg.identity.upper"}
```

Given a (large sparse) system_matrix (for instance stemming from a discretized Laplace operator) and rhs vector, we can solve the corresponding linear system using a specific solver by calling

```
#include <dune/xt/la/solver.hh>
```
Extending DUNE: dune-xt

```cpp
XT::LA::Solver<SystemMatrixType> linear_solver(system_matrix);
linear_solver.apply(rhs, solution, "ldlt.simplicial");
```

The above call to apply is default implemented by calling

```cpp
code
apply(rhs, solution, options(type));
```

where options(type) always returns a configuration object with appropriate options for the selected type. With type = "ldlt.simplicial", for instance, we are implicitly using the following options (which are the default for "ldlt.simplicial"):

```
1 type = ldlt.simplicial
2 post_check_solves_system = 1e-5
3 check_for_inf_nan = 1
4 pre_check_symmetry = 1e-8
```

All implemented solvers default to checking whether the computed solution does actually solve the linear system and provide additional sanity checks. We make extensive use of exceptions if any check is violated, which allows to recover from undesirable situations in library code (as shown in the example above). We also provide our own implementation of the `DUNE_THROW` macro in dune/xt/common/exceptions.hh which replaces the macro from dune-common and can be used with any Dune::Exception. Apart from a different formatting (and colorized output), it also provides information of interest in distributed memory parallel computations.

The "ldlt.simplicial" solver, for instance, does only work for symmetric matrices and we thus check the matrix for symmetry beforehand (which can be disabled by setting "pre_check_symmetry" to 0). Each type of linear solver provides its own options, which allow the user to fine-tune the linear solver to his needs. For instance, the iterative "bicgstab" solver with "ilut" preconditioning accepts the following options (in addition to "post_check_solves_system" and "check_for_inf_nan", which are always supported):

```cpp
code
#include <dune/xt/la/solver.hh>

XT::LA::Solver<SystemMatrixType> linear_solver(system_matrix);
auto options = linear_solver.options("bicgstab.ilut");
options["max_iter"] = 1000;
options["precision"] = "1e-14";
options["preconditioner.fill_factor"] = 10;
options["preconditioner.drop_tol"] = "1e-4";
linear_solver.apply(rhs, solution, options);
```

The actually implemented variant of Solver in dune-xt-la also takes a communicator as an optional argument, to allow for distributed parallel linear solvers. We provide several implementations of such parallel solvers based on dune-istl. A linear solver for IstlRowMajorSparseMatrix, for instance, supports the following types(), most of which can be readily used in distributed parallel environments:

```cpp
code
{ 
  #if !HAVE_MPI && HAVE_SUPERLU
  "superlu",
  #endif
  "bicgstab.amg.ilu0", "bicgstab.amg.ssor", "bicgstab.ilut",
  "bicgstab.ssor", "bicgstab"
  #if HAVE_UMFPACK
    , "umfpack"
```
6 The dune-xt-functions module: Local and localizable functions

A correct interpretation of data functions and a correct handling of analytical and discrete functions is an important part of every PDE solver. Given a domain $\Omega \subset \mathbb{R}^d$ and a grid view $\tau_h$ of this domain, one is usually not interested in considering a function $f : \Omega \rightarrow \mathbb{R}$, but rather its local function $f^i := f \circ \Phi^i$ with respect to a grid element $t \in \tau_h$, where $\Phi^i : t \rightarrow \hat{t}$ denotes the reference mapping from the respective reference element $\hat{t}$ to the actual element $t$, as provided by dune-grid. For simplicity’s sake we only present the scalar case here, but the same concepts apply to vector- and matrix-valued functions, as implemented in dune-xt-functions. This allows to compute integrals in terms of quadratures from dune-geometry and shape functions from dune-localfunctions defined on the reference elements of a grid. In addition, one is interested in “localized derivatives” of $f$, for instance its localized gradient $\nabla f^i := \nabla f \circ \Phi^i$. While the localized gradient of a function does not coincide with the gradient of a local function it is an important tool to preserve the structure of an integrand, when transformed to the reference elements for integration. Using the chain rule, we obtain for the latter $\nabla f^i = \nabla f \circ \Phi^i = (\nabla f \circ \Phi^i) \nabla \Phi^i$, while the following holds for the former: $\nabla f^i = \nabla f \circ \Phi^i = (\nabla f \circ \Phi^i)(\nabla \Phi^i)^{-1} = \nabla f^i (\nabla \Phi^i)^{-1}$.

Consider for instance the following integral arising in the weak formulation of a Laplace operator, which is transformed to the reference element $\hat{t}$, where $a$, $\varphi$ and $\psi$ denote scalar functions and $\Lambda_t^2 := \det(\nabla \Phi^i \nabla \Phi^j)$:

$$\int_{\hat{t}} (a \nabla \psi) \cdot \nabla \varphi = \int_{\hat{t}} \Lambda_{\hat{t}} (a \circ \Phi^i)(\nabla \psi \circ \Phi^i) : (\nabla \varphi \circ \Phi^i).$$

Using the definition of a local function and a localized gradient, the above is equivalent to

$$\ldots = \int_{\hat{t}} \Lambda_{\hat{t}} (a^\prime \nabla \psi^\prime) : \nabla \varphi^\prime,$$

which nicely preserves the structure of the original integrand.

For a given grid view $\tau_h$, we thus call a function localizable with respect to $\tau_h$, if there exists a local function with localized derivatives for each element of the grid view. These functions form the discontinuous space of locally polynomial functions with varying polynomial degree,

$$Q(\tau_h) := \left\{ q : \Omega \rightarrow \mathbb{R} \mid \forall t \in \tau_h \quad \exists k(t) \in \mathbb{N}, \text{ such that } q^i \in \mathbb{P}_{k(t)} \right\},$$

where $\mathbb{P}_{k(t)}$ denotes the set of polynomials $q : \hat{t} \rightarrow \mathbb{R}$ of maximal order $k$. Note that in the context of integration, the polynomial degree of the integrand is limited by the availability of suitable quadratures and always finite in practical computations. Thus, even functions such as $f(x) = \sin(x)$ can in practical computations only be represented by surrogates $f \approx f \in Q(\tau_h)$. Consequently, the space $Q(\tau_h)$ presents a common space for all functions, including analytically given data functions and, most importantly, discrete functions of any discretization framework.

This concept allows for a generic discretization framework such as dune-gdt, where all operators, functionals, projections and prolongations are implemented in terms of localizable functions. Thus, a Laplace operator which locally realizes (2) can be used to assemble a system or product matrix, where $\psi^i$ and $q^i$ are basis functions of a discrete function space, or to compute the norm of any combination of analytical or discrete functions, in which case $\psi^i$ and $q^i$ model the corresponding local functions. In dune-xt-functions, we provide interfaces and implementations to realize this concept of localizable functions and local functions (not to be confused with shape functions from dune-localfunctions) and for the remainder of this section we discuss the realization of these
Extending DUNE: dune-xt

concepts in dune-xt-functions. A similar effort has been undertaken in the dune-functions module [13], independently of our work. Since both efforts share the same mathematical basis it is planned to make dune-xt-functions compatible to dune-functions in future work.

Given a grid element $t \in \tau_h$, we model a collection of (scalar-, vector- or matrix-valued) local functions $\phi^t : \hat{t} \rightarrow \mathbb{R}^{rc}$, for $r, c \in \mathbb{N}$ (where $\hat{t}$ denotes the reference element associated with $t$) by the LocalfunctionSetInterface in dune/xt/functions/interfaces.hh:

```cpp
template <class EntityType, class DomainFieldType, size_t dimDomain, class RangeFieldType, size_t dimRange, size_t dimRangeCols = 1>
class LocalfunctionSetInterface
{
  // not all methods and types show ...
  public:
    virtual const EntityType& entity() const
    {
      return entity;
    }

    virtual size_t size() const = 0;
    virtual size_t order() const = 0;

    virtual void evaluate(const DomainType& xx, std::vector<RangeType>& ret) const = 0;
    virtual void jacobian(const DomainType& xx, std::vector<JacobianRangeType>& ret) const = 0;
};
```

The template parameter EntityType models the type of the grid element $t \in \tau_h$, the parameters DomainFieldType and dimDomain model $\mathbb{R}^d \supset t$ while RangeFieldType, dimRange and dimRangeCols model $\mathbb{R}^{rc}$.38

The resulting DomainType is a FieldVector<DomainFieldType, dimDomain>, while RangeType and JacobianRangeType are composed of FieldVector and FieldMatrix, depending on the dimensions. Each set of local functions has to report its polynomial order and size. The methods evaluate and jacobian expect vectors of size() for ret.

Note that we use a combination of static polymorphism, to fix the type of the grid and all dimension at compile time, and dynamic polymorphism, which allows to exchange functions of same grid and dimensions at runtime.

By implementing all operators and functionals in terms of LocalfunctionSetInterface, a discretization framework can realize the above claim of unified handling of analytical and discrete functions. This is for instance the case in dune-gdt, where the local bases of discrete function spaces are realized as implementations of LocalfunctionSetInterface. In addition, we also provide an interface for individual local functions, which can be used by data functions and local functions of discrete functions:

```cpp
template <class E, class D, size_t d, class R, size_t r, size_t rC>
class LocalfunctionInterface
{
  // not all methods and types shown ...
  public:
    virtual void evaluate(const DomainType& xx, RangeType& ret) const = 0;
    virtual void jacobian(const DomainType& xx, JacobianRangeType& ret) const = 0;

    virtual size_t size() const override final
    {
      return 1;
    }
};
```

As a shorthand, we write $<E, D, d, R, r, rC>$ instead of $<\text{EntityType}, \text{DomainField}, \text{dimDomain}, \text{RangeField}, \text{dimRange}, \text{dimRangeCols}>$.38

© by the authors, 2017

Archive of Numerical Software 5(1), 2017
Alongside, we also provide the following interface for localizable functions which mostly act as containers of local functions:

```cpp
1 template <class E, class D, size_t d, class R, size_t r, size_t rC>
2 class LocalizableFunctionInterface
3 {
4     // not all methods and types shown ...
5     public:
6         virtual std::string name() const;
7         virtual std::unique_ptr<LocalFunctionInterface<E, D, d, R, r, rC>>
8             local_function(const EntityInterface entity) const = 0;
9     }
10 }
```

Based on this interface we provide visualization and convenience operators, allowing for

```cpp
(f - f_h).visualize(grid_view, "difference");
```

where f may denote a localizable data function and f_h might denote a discrete function, if both are localizable with respect to the same grid view. Expressions such as f - f_h, f + f_h or f*f_h yield localizable functions via generically implemented local functions (if the dimensions allow it).

We also provide numerous implementations of LocalizableFunctionInterface in dune-xt-functions, most of which can also be created using the FunctionsFactory, given a Configuration:

- **CheckerboardFunction** in dune/xt/functions/checkerboard.hh models a piecewise constant function, the values of which are associated with an equidistant regular partition of a domain. Sample configuration for a function $\mathbb{R}^2 \to \mathbb{R}$:

  ```
  1  lower_left = [0. 0.]
  2  upper_right = [1. 1.]
  3  num_elements = [2 2]
  4  values = [1. 2. 3. 4.]
  ```

- **ConstantFunction** in dune/xt/functions/constant.hh models a constant function. Sample configuration for a function $\mathbb{R}^d \to \mathbb{R}^2 \times 2$, for any $d \in \mathbb{N}$, mapping to the unit matrix in $\mathbb{R}^2$:

  ```
  1  value = [1. 0.; 0. 1.]
  ```

- **ExpressionFunction** in dune/xt/functions/expression.hh models continuous functions, given an expression and order at runtime (expressions for gradients can be optionally provided). Sample configuration for $f : \mathbb{R}^2 \to \mathbb{R}^2$ given by $(x, y) \mapsto (x, \sin(y))$:

  ```
  1  variable = x
  2  order = 3
  3  expression = [x[0] sin(x[1])]
  4  gradient.0 = [1 0]
  5  gradient.1 = [0 cos(x[1])]
  ```

Note that the user has to provide the approximation order, resulting in $f$ being locally approximated as a third order polynomial on each grid element.

- **GlobalLambdaFunction** in dune/xt/functions/global.hh models continuous functions by evaluating a C++ lambda expression. Sample usage for $f : \mathbb{R}^2 \to \mathbb{R}$ given by $(x, y) \mapsto x$:

  ```
  GlobalLambdaFunction<e, d, 2, R, 1>([](DomainType x){ return x[0]; }), 1); // << local polynomial order
  ```

---

39 dune/xt/functions/factory.hh
- Spe10::Model1Function in dune/xt/functions/spe10.hh models the permeability field of the SPE10 model1 test case, given the appropriate data file\textsuperscript{40}.

References


\textsuperscript{40}Available at \url{http://www.spe.org/web/csp/datasets/set01.htm}


