

hpGEM – A Software Framework for Discontinuous Galerkin Finite Element Methods

LARS PESCH, ALEXANDER BELL, HENK SOLLIE, VIJAYA R. AMBATI,
ONNO BOKHOVE, and JAAP J.W. VAN DER VEGT

hpGEM, a novel framework for the implementation of discontinuous Galerkin finite element methods (FEMs), is described. We present data structures and methods that are common for many (discontinuous) FEMs and show how we have implemented the components as an object-oriented framework. This framework facilitates and accelerates the implementation of finite element programs, the assessment of algorithms, and their application to real-world problems. The article documents the status of the framework, exemplifies aspects of its philosophy and design, and demonstrates the feasibility of the approach with several application examples.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications–C++; G.1.8 [**Numerical Analysis**]: Partial Differential Equations–Finite element methods; G.4 [**Mathematical Software**]: Algorithm design and analysis

General Terms: Algorithms, Design

Additional Key Words and Phrases: Discontinuous Galerkin methods, PDE, unstructured mesh, object-oriented programming

1. INTRODUCTION

Finite element methods are a field of active research in applied mathematics. In recent years, new and rapid developments have taken place, in particular in the field of discontinuous Galerkin (DG) methods. This type of finite element method (FEM) is different from “classical” ones in that functions are allowed to be discontinuous across the boundaries between elements. The advantages of the resulting discretizations include the possibility to use basis functions with different polynomial order in different elements (p -adaptation), the ease of incorporating mesh refinement (h -adaptation) and unstructured meshes, and the increase in locality of the discretization, which is of particular interest for parallel computing. More information about discontinuous Galerkin methods, their properties and their advantages over classical FEMs and other methods can be found in Section 2. For the moment we can return to our consideration of FEMs in general.

Typically, the design of finite element (FE) algorithms starts with the formal definition of the method by deriving a weak formulation of the system of partial differential equations and choosing basis functions to discretize the function spaces. Properties of the method can then be examined and, given satisfying results, the next step will be to use it for solving the

Authors’ address: University of Twente, Dept. of Applied Mathematics, P.O. Box 217, 7500 AE Enschede, The Netherlands.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2006 ACM 0098-3500/2006/1200-0001 \$5.00

target problem numerically. For that purpose, the developed algorithm has to be translated into a computer program. A correct implementation not only gives a numerical solution but can also be used to determine additional properties of the algorithm, like approximation orders, iterative convergence rates, and computational costs.

The definition and analysis of a FEM is a complicated exercise and relies on the mathematical skills of the developer. In this article, we address the following step, the implementation as a computer program, which we consider an equally complex task based on our experience with diverse mathematical problems, algorithms, and programming languages. Additional complications arise from recent developments on the computational side, which allow the numerical solution of many real-world problems on increasingly powerful computing systems. On the other hand, the utilization of such systems poses additional requirements on the implementation—parallelization is a keyword hinting at the added complexity. At this point it is tenable that the transformation of a mathematical model into a capable computer code is a task that goes far beyond the abilities of a single, say mathematically trained, person. Apart from the skill constraint, developments are also limited by the amount of work that an individual can accomplish on the timescale of a typical research project. The logical consequence is that efforts have to be joined to reach the forefront of current developments in applied mathematics.

A related difficulty regarding the efficiency of the software development process is to maintain productivity over more than a single project. Having invested in software design and development as described above, the effort would be wasted if there was no possibility to reuse the result—i.e. the software artifacts—for applying new algorithms to the same or other problems. For that reason it is important that software is built in a modular and extensible way, representing general concepts and separating the application-related items from abstract mathematical parts, and those in turn from underlying computer-scientific details. It is therefore a prerequisite for our work to decompose FEMs into recurring components and tasks, and additional parts, which are specific to individual methods.

The foundation of the work presented here is that those parts that are the same in many FEM implementations constitute a relatively large fraction. Examples are the representation of the geometric mesh of the domain, the mathematical definition of the finite elements (shape, basis functions, degrees of freedom), and the assembly and solution of systems of equations. The crux is that typically these parts are ‘reinvented’ and reimplemented by individuals when starting from scratch, incurring a large overhead in development time and—more importantly—the potential of introducing coding errors. That is where our current work comes in: it provides well-tested data structures and methods on which the specific FE application can build, thus cutting short the implementation time and reducing the danger of introducing mistakes.

Naturally, the decomposition of FEMs will also reveal tasks for which highly developed, tested and established solutions are available, e.g. from computer science (data containers and search algorithms) or numerical linear algebra (solvers for systems of equations). It is not so much the question of *whether* to use them but rather *how* to do it. Here the benefit of our software environment is that it provides access to various packages, which enhance its functionality and capabilities.

Given hpGEM, the finite element software framework we describe here [hpGEM], it remains to the applying scientist—starting from a correct mathematical formulation of the discrete problem—to a) assemble the provided components in a correct and efficient way,

and b) add whatever is special or unique to a considered problem or algorithm. Obviously additions to the framework are possible when sufficient generality and usefulness have been shown.

This article is structured as follows: a brief introduction to DG FEMs, their properties and differences to other FEMs is presented in Section 2. In Section 3, we will detail the requirements we formulated for the software framework and compare these with some existing FE packages. Topics like the choice of the programming model, the implementation language and software engineering aspects are covered in Section 4. Section 5 contains several examples of how a software framework can be designed to reflect the generic character of finite element algorithm components. Some sample applications to solve physical problems are presented in Section 6. In Section 7, we summarize our work, draw conclusions and give an overview of future activities.

2. DISCONTINUOUS GALERKIN FINITE ELEMENT METHODS

Finite element methods constitute a theoretically well-founded approach to discretize systems of partial differential equations (PDEs) and are used to solve numerous problems arising from applications in physics and technology. A key feature of many problems are phenomena which occur at very different length scales but strongly interact with each other. Examples are thin boundary layers and moving interfaces in fluid flow, chemical reactions at surfaces, and flames in combustion. A simulation technique is required with adjustable accuracy to efficiently capture the relevant scales in these multi-scale problems. Adaptation can either locally refine the mesh (*h*-refinement) or adjust the polynomial order of the solution representation (*p*-refinement). The combination of these techniques is called *hp*-adaptation and is a promising approach to obtain highly efficient numerical schemes for a large variety of multi-scale problems.

The use of *hp*-adaptation is, however, non-trivial to implement in standard finite element methods. In these methods, serious problems occur when two or more elements connect to a face of another element after local mesh refinement (hanging nodes) or if the polynomial basis functions in two connecting elements are of different order after *p*-adaptation.¹ As mentioned in the introduction, discontinuous Galerkin finite element methods pose no extra problems regarding *hp*-adaptation² and achieve higher order accuracy even on unstructured meshes. This is possible because the support of each basis function contains only one element, cf. Figure 1, and a coupling to the neighboring elements exists only in the weak sense, as we shall see later. The locality of the basis functions not only allows to vary the number of basis functions (the approximation order) individually for each element, but also reduces the coupling in the discretization: information is exchanged exclusively between neighboring elements by fluxes (hyperbolic PDEs) or other face-based operators (elliptic PDEs). This is beneficial for *hp*-adaptation as well as for parallel computing because it minimizes the amount of data communication. General surveys of discontinuous Galerkin finite element methods can be found in [Cockburn 1999; Cockburn et al. 2000; Cockburn and Shu 2001; Arnold et al. 2002]. In the remainder of this section we will give briefly introduce a DG method for conservation laws.

We consider a non-linear scalar conservation law, which is a simplified model for a large

¹Different zones have to be connected by special elements complicating the software.

²*hp*-adaptation is, in fact, so central to the development of both discontinuous Galerkin methods and our software framework that *hp* became part of the name hpGEM.

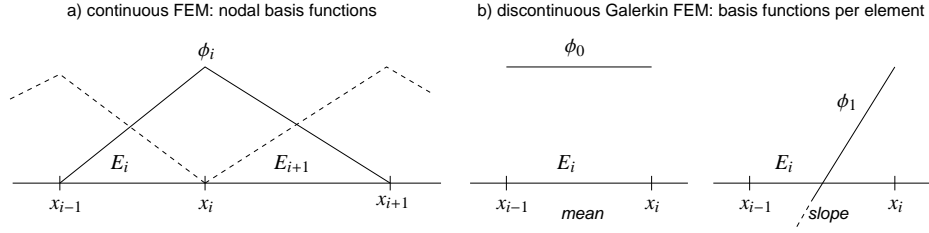


Fig. 1. Nodal basis functions of continuous FEMs (left) and element-based basis functions of DG methods (right). For continuous methods the shown basis (one degree of freedom in every mesh node) is the lowest possible approximation order. For the discontinuous Galerkin method, a linear representation per element E_i can be obtained by using the two basis functions for the element mean and slope, ϕ_0 and ϕ_1 , respectively. This representation could be reduced by using just ϕ_0 for the element mean, corresponding to a finite volume scheme.

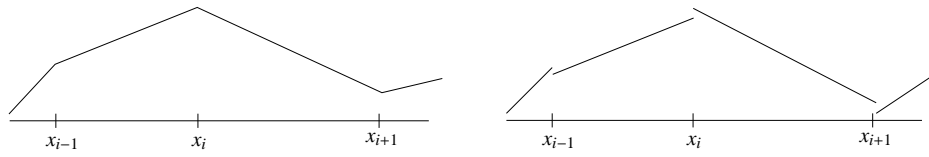


Fig. 2. Representation of a function by the above bases for continuous (left) and discontinuous (right) elements.

class of physical problems and contains many of their characteristic features. The scalar conservation law in a spatial domain $\Omega \subset \mathbb{R}^2$ and at time $t \in [t_s, t_e]$ is

$$\frac{\partial u}{\partial t} + \nabla \cdot \mathcal{F}(u) = 0, \text{ in } (t_s, t_e) \times \Omega, \quad (1)$$

with initial condition $u(t = t_s, \mathbf{x}) = u_0(\mathbf{x})$, $\forall \mathbf{x} = (x, y)^\top \in \Omega$, and boundary condition $u(t, \mathbf{x}) = g(\mathbf{x})$ at $\Gamma \subset \partial\Omega$ for all $t \in [t_s, t_e]$. Here, u is a conserved scalar quantity, \mathcal{F} the flux function, g a prescribed boundary function, and $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y})^\top$, where a superscript $(\cdot)^\top$ denotes the transpose of a vector. Examples for flux functions \mathcal{F} are the (linear) advection flux, $\mathcal{F}(u) = au$, and the flux for the inviscid Burgers equation, $\mathcal{F}(u) = u^2/2$.

The DG discretization is obtained by the following steps. First, we define a discrete space V_h without continuity requirement at element boundaries. Basis functions for V_h can be chosen such that they are nonzero only in one element rather than in all elements connected to a node, as would be the case for the nodal basis functions of classical FEMs. In Figure 1, we contrast nodal basis functions of a continuous FEM with the element-wise defined polynomial basis functions ϕ_i of a discontinuous Galerkin method in one dimension. Possible representations of a function in the discrete spaces spanned by these sets of basis functions are given in Figure 2.

Next, we multiply Equation 1 with an arbitrary test function v , integrate over each element, and sum over all elements. Finally, the solution u and the test function v are restricted to the space V_h , and denoted by u_h and v_h , respectively. The weak formulation of (1) is then given by: Find a $u_h \in V_h$, such that for all $v_h \in V_h$ the following relation is satisfied:

$$\sum_{i=1}^{N_e} \left(\frac{\partial}{\partial t} \int_{E_i} u_h v_h dE - \int_{E_i} \nabla v_h \cdot \mathcal{F}(u_h) dE + \int_{\partial E_i} v_h \mathbf{n} \cdot \hat{\mathcal{F}}(u_h^-, u_h^+) ds \right) = 0. \quad (2)$$

Here, E_i denotes the i^{th} element in a finite element mesh with N_e elements. The boundary of

each element E_i is denoted ∂E_i with \mathbf{n} the exterior pointing normal vector at this surface. Since the discrete function u_h is not required to be continuous at the element faces we have to deal with the fact that u_h , and consequently also the flux $\mathcal{F}(u_h)$ in the integral over the element boundaries, is multi-valued. We call the limit of u_h from the interior u_h^- , whereas the exterior limit is denoted u_h^+ ; the latter is either based on the u_h in a neighboring element or on the boundary data g . By introducing a consistent, antisymmetric numerical flux $\hat{\mathcal{F}}(u_h^-, u_h^+)$, the discretization is made conservative. When possible, the numerical flux can be used to give a physical meaning to the multivaluedness of u_h across the element boundary, for instance by using the (approximate) solution of a Riemann problem, an initial value problem with two constant states, see e.g. [Toro 1999].

The discrete solution u_h is expanded on element E_i in terms of basis functions ϕ_m^i as

$$u_h|_{E_i}(t, \mathbf{x}) = \sum_{m=0}^p \hat{u}_m^i(t) \phi_m^i(\mathbf{x}), \quad (3)$$

with the expansion coefficients \hat{u}_m^i and basis functions ϕ_m^i . Here, p is the number of basis functions in an element and may vary for different elements. The semidiscrete problem consists of a set of ordinary differential equations for the expansion coefficients \hat{u}_m^i in each element, which can be solved using a wide variety of time integration methods.

To reveal a number of building blocks for the implementation of a discontinuous Galerkin FEM, we also make use of faces F_j , $j \in \{1, \dots, N_f\}$, which are the objects of codimension one that bound the elements. Note that in Equation 2 integrals over the boundaries of elements are evaluated twice for each internal face. The two integrals differ only in the sign of the normal vector \mathbf{n} , so that we can combine the integrands as $v_h^L \mathbf{n} \cdot \hat{\mathcal{F}} + v_h^R (-\mathbf{n}) \cdot \hat{\mathcal{F}} = (v_h^L - v_h^R) \mathbf{n} \cdot \hat{\mathcal{F}}$, where superscripts $(\cdot)^L$ and $(\cdot)^R$ denote values taken from the elements on the (arbitrarily assigned) left and right side of the face, respectively. Hence

$$\sum_{i=1}^{N_e} \left(\sum_{m=0}^p \frac{\partial \hat{u}_m^i}{\partial t} \int_{E_i} \phi_k^{i'} \phi_m^i dE - \int_{E_i} \nabla \phi_k^{i'} \cdot \mathcal{F}(u_h) dE \right) + \sum_{j=1}^{N_f} \int_{F_j} ((\phi_k^i)^L - (\phi_k^i)^R) \mathbf{n} \cdot \hat{\mathcal{F}}(u_h^L, u_h^R) ds = 0, \quad (4)$$

for all test functions $\phi_k^{i'}$ with $i' \in \{1, \dots, N_e\}$ the element on which the test function is nonzero and k the index of the basis function on the element $E_{i'}$. We note the different mathematical entities in (4) and give some remarks about their realization in hpGEM:

—Sums over all elements or faces.

The finite element mesh is represented in hpGEM by an interface to container classes.

These provide access to the elements and faces through iterators which are compatible to those of the C++ Standard Template Library (STL), cf. Section 5.1.

—Element and face integrals.

Integrals are the basic units of the FE formulation and their computation is one of the services provided by our software framework, see Section 5.2.

—Systems of equations.

Equation 4 constitutes a nonlinear system of equations for the expansion coefficients \hat{u}_m^i .

Various linear and nonlinear solvers are used by and accessible within the framework.

Integrals and loops over internal faces distinguish discontinuous from continuous FEMs, which do not exhibit these constituents. The resulting discretization (4) is purely element-based in the sense that the only coupling between the degrees of freedom on neighboring

elements is through the numerical flux $\hat{\mathcal{F}}(u_h^-, u_h^+)$. This allows to use different polynomial order for u_h in the two elements connected by the face F_j .

The above formulation has been the foundation for our previous work on the Euler equations [van der Vegt and van der Ven 1998; 2002; van der Ven and van der Vegt 2002; van der Ven et al. 2003] and shallow water equations [Bokhove 2005]. Discontinuous Galerkin methods can also be applied to elliptic equations, an example can be found in [van der Vegt and Tomar 2005]. Combining the hyperbolic and elliptic parts allows, for instance, to numerically solve mixed-type equations like (nonlinear) advection-diffusion [Sudirham et al. 2006; Bokhove et al. 2005] and the Navier-Stokes equations [van der Ven et al. 2005; Klaij et al. 2006a; 2006b].

3. REQUIREMENTS ANALYSIS AND RELATED WORK

Looking at the finite element software landscape there are many existing tools, which we cannot review extensively here. From code fragments developed by single users to commercial systems there is a wide spectrum of artifacts available that may allow to implement a solution scheme. To establish why we have started another development we consider the essential requirements for our software; these include:

—Ability to use discontinuous Galerkin methods.

These methods are the focus of our research. From Section 2 it follows that some aspects make DG methods different from continuous finite element methods. In particular the use of face-based data structures for the computation of (numerical) fluxes in hyperbolic systems like (4) is different from classical FEMs. On the other hand, some of these differences can be taken advantage of, in particular in the form of increased concurrency in the computation of the element-based discretization.

—General types of elements in \mathbb{R}^d , $d = 1, \dots, 4$.

Another advantage of DG methods is their flexibility with respect to the geometric shape of the elements. For two-dimensional meshes, mixtures of triangles and quadrilaterals can be as easily accommodated as three-dimensional meshes with tetrahedra, pyramids, triangular prisms, and hexahedra. For so-called space-time FEMs, we have to extend the elements with an extra dimension for time, so for three-dimensional space the elements become four-dimensional.

—Dimension-independence of the code.

For many FE algorithms the mathematical formulation is independent of the dimension of the problem. To a large degree this property can be conserved by the software environment. Application codes developed in a two-dimensional setting can frequently be used for three-dimensional computations by changing a (C++-template-) parameter, if the user's code allows this, too.

—Easy and fast generation of applications.

When testing an algorithm the first question is how long it will take to correctly implement it. Hence it is important that provided components are easy to use and well-documented.

—Parallelism handled internally.

While parallel computing is of prime importance for many relevant physical applications, it is a corollary of the previous item that the exploit of parallel computation strategies should be as far as possible handled internally, so that a user program can remain

largely unchanged for serial and parallel processing. To what extent this can be realized and where the user has to intervene will be described in a follow-up article.

—Enforcement of quality standards.

To guarantee a correct and extensible FE framework, we rely on documentation, both in-code and external, a set of unit tests and separate test applications.

—Access to external software for common tasks.

Because we concentrate on FE-related research we make use of existing components for the pre- and post-processing steps, e.g. mesh generation and visualization. Also for tasks like linear algebra, existing high-quality solutions can be employed by the framework or made accessible to the user.

The above items constitute the most important requirements, yet the result of our search for a matching candidate was that no available solution satisfied our needs. We briefly discuss this conclusion by comparing a few available software packages based on object-oriented development to the above list; the findings are typical for many FEM software artifacts.

Discontinuous Galerkin methods, being a relatively recent topic, are not implemented by most FE packages and their special aspects (e.g. face data structures) are not taken into account. Packages which seem to support discontinuous Galerkin methods are DEAL.II [Bangerth et al.] and DOLFIN [Hoffman and Logg 2002]. However, the former works only on n -cubes, i.e. elements are lines, quadrilaterals, and hexahedra in dimension 1, 2, and 3, respectively. The latter, just like other packages, e.g. ALBERTA [Schmidt and Siebert], restricts itself to simplices, so that meshes consist of triangular or tetrahedral elements. Mixed meshes are rarely accommodated by existing FE software, possibly because of the complications when using continuous FEMs. We could not find any package that offers support for space-time discretizations, elements are only implemented for dimensions one to three. Within these bounds, codes based on DEAL.II or ALBERTA can be dimension-independent for meshes based exclusively on n -cubes or simplices, respectively.

Different philosophies become apparent when it comes to the application program interface (API) available to the users of a software framework. In some packages, high-level access is provided to a degree that modules allow the solution of pre-programmed equations by choosing the geometry, suitable initial and boundary conditions [Hoffman and Logg 2002].

The access to external software in the mentioned packages is—if at all provided—frequently hidden by extensive interfaces. This has the advantage of unified access to different packages, which however comes at high development cost and possibly restricts the range of usable options of the integrated packages. Remarkably, some FE packages rely entirely on their own developments regarding linear algebra and equation solvers.

Efforts regarding the quality measures mentioned previously vary; online documentation for the abovementioned packages is available, sometimes also detailed tutorials, e.g. for DEAL.II, which also seems to be most advanced regarding its test suite.

Apparently, failure to comply with the first two items from our list above lead us to reject many available FE packages. The other topics are more debatable, but collectively can be of equal importance. We will show applications of mixed meshes and dimension independent code in the examples in Section 6, which belong to the sample programs helping to learn using hpGEM. Software handling issues will be touched upon in Section 4.3. Regarding the quality and accessibility of the framework, our available documentation is

important, but what we have found even more valuable is that a test-suite is built up with the code. Currently more than 150 unit tests check the correct working of individual classes, procedures, and collaborative tasks (like computing normal vectors, evaluating integrals). The unit tests can be re-run at any time, so that changes and additions to the framework can immediately be assessed for correctness.

Whenever a complex task (e.g. linear equation solving) is served well by an available external solution we consider using that one rather than to reimplement such features. The latter does not seem viable to us for reasons of development overhead and lack of expertise; we appreciate the efforts of many groups to provide free, quality, high-performance software.

Regarding the API, it is not the goal of hpGEM to be a ‘solver’. Rather we intend to serve numerical scientists by providing the concepts that occur in a worked-out weak form, e.g. as identified for Equation 4 in Section 2. Concrete problems are solved by sample applications, but the actual hpGEM framework concentrates on the general building blocks. We believe that this allows us to be least restrictive and most widely applicable.

4. BASIC APPROACH

When dealing with FEMs, an advantageous aspect of their structure is the way in which mathematical and geometrical concepts emerge and work together. It will be beneficial in several ways to preserve this structure when creating a finite element software environment: first, the accessibility of the software framework for users with mainly mathematical background knowledge is improved; having worked out a FE formulation in mathematical terms, the translation into computer code is simplified if the same concepts are used to build up the code as for the analytical formulation. Second, software artifacts representing the general concepts of finite element algorithms will have a high potential of reuse. A typical one-off FE application code tries to use as many simplifications as the structure of the problem at hand allows. Consequently, the resulting code does not reflect the abstract mathematical concepts any more. The latter, however, are the general building blocks for many formulations and hence providing them will increase the re-usability. The clear-cut entities in FEMs can be preserved well by an object-oriented (OO) programming model.

4.1 Object-Oriented Development and Programming

The first approaches to object-oriented modeling of systems date back to the eighties. In the mid-nineties, Rumbaugh’s *Object Modeling Technique* (OMT) [Rumbaugh et al. 1991] and Booch’s *Booch Method* (BM) [Booch 1994] were combined resulting in the development of the Unified Modeling Language (UML), which was accepted as a standard of the Object Management Group in 1997. In 2005, UML evolved to an international standard as ISO 19501.

We consider object-oriented development, comprising the modeling and programming of a system, as the most important advance in software engineering during the last two decades, clearly going one step further than object-oriented programming alone.

During the development of hpGEM we use the UML to represent the results from individual phases of the BM: conceptualization (definition of core requirements of system), analysis (creation of a model of the system capturing important properties regarding the functionality), design (modeling the system architecture in detail: subsystems, first “objects” and their interaction), evolution (translation of into a programming language, documentation), and maintenance (post-production phase: adaptation, extension, bug-fixes).

The first two phases have already been addressed in Section 3. Note that the usage of an object-oriented development model does not require the usage of an object-oriented implementation language, although such a language clearly simplifies the implementation.

A disadvantage of the object-oriented approach lies in the runtime overhead that may be incurred by the introduction of higher-level software constructs. Representing the general concepts of FEMs within an object-oriented language reduces the possibilities to apply structural, problem-dependent optimization and introduces overhead for the object handling. However this effect on the runtime has to be contrasted with the greater ease of familiarization and development with the software framework. We experience that the time savings thanks to faster application development outweigh the runtime overhead.

4.2 Programming Language

Having agreed on the development model, the programming language is the next item to select. In recent years, with the growing acceptance of object-oriented techniques also in the scientific programming world, C++ has been the language of choice for many developments. Reasons that suggest committing to this language include:

- C++ is not only a full-fledged object-oriented language, but also supports programming in procedural style and with elementary constructs, like loops. The latter fact is particularly beneficial for scientific computing where the nature and amount of data requires such constructs and their execution at close to machine speed. Furthermore the overhead to get users without experience in object-oriented programming started with hpGEM is reduced by the availability of these more traditional concepts.
- C++ enforces strong type checking, which leads to an unambiguous API and enables to find many logical errors at compile time.
- Aspects of generic programming are included in C++, most notably in the form of templates. In the scientific computing context, templates are also used to improve runtime performance, cf. Veldhuizen [1995] for classical examples. We will come back to the usage of templates at the end of this section.
- C++ is a widely available, standardized, well-known, current language. These properties make the language a solid foundation to build on: standard-compliant code is guaranteed to run on a wide range of platforms with several available compilers. Its widespread use, not only in scientific computing, means that support for C++ will not cease on the foreseeable timescale of our project, that there are qualified scientists to contribute to the project, and vice versa that expertise gained in the project is of wider relevance as well. Last but not least, C++ is one of the most supported languages when it comes to techniques and tools for software engineering. These can significantly contribute to the success of a project; the ways in which we make use of such tools will be described in Section 4.3.

A possible alternative in the language decision would have been Fortran 90, which is still widely used for scientific computing. However, Fortran cannot match with C++ on the above list of qualities. In the versions for which compilers are currently available, Fortran incompletely supports object-oriented programming and lacks some more evolved object-oriented features of C++ (e.g. multiple inheritance). The basic data type of Fortran (on which it is admittedly outperforming most other languages) is the array, but less structured

types as they are readily provided by the C++-Standard Template Library (STL) containers are not offered, and generic programming support as through C++-templates is not present in the language. Obviously it would not be necessary to limit oneself to a single programming language, as linking object files generated from several languages is possible. While we may use external software packages programmed in other languages, for the development of hpGEM itself, we have chosen to restrict ourselves to C++, as doing otherwise would potentially decrease portability and increase complexity for the developers.

Of course, choosing C++ means making a compromise that includes disadvantages, too. The absence of many services and data structures (like arrays) from the language is obstructive and has to be compensated by libraries. This fact together with the inclusion model for header files, leads to increased compilation time overhead. With the advent of relevant libraries and the increased computer speed these matters tend to become less influential, but that cannot be said for the compile times of code that relies heavily on templates. As mentioned above, templates are used frequently not only to achieve generality through static polymorphism but also to improve performance through compile time code expansion. While the C++ standard includes explicit instantiation as a means to avoid the overhead of the inclusion model for template definitions, this can most frequently not be used for the purposes above because the template arguments are not determined by the library but rather by the user's code. As a simple example, the class template

```
template <DimType dim> class PhysSpacePoint<dim>;
```

is suitable for explicit instantiation since we know that for all our applications the parameter `dim`, which gives the dimension of the physical problem space, is restricted to the range $1, \dots, 4$. On the other hand, a construct like

```
template <class UserData> class DataOnElements;
```

cannot be instantiated at the build time of the hpGEM library, as its argument, namely the class that includes all the data that is stored for each element, is completely problem dependent and hence given by the user.

The latter case is, unfortunately, far more typical of our framework's code, which relies heavily on templates; furthermore, experience from large projects [Vandevoorde and Joutsittis 2003] suggests that explicit instantiation becomes hard to manage, so that up to now we have refrained from using it and put up with the longer compilation times.

An aspect that should be mentioned—though it concerns the object-oriented approach rather than the choice of C++ as the programming language—is that thinking in an object-oriented way is a skill that requires learning and experience. It often takes considerable time to adapt new users to this programming and thinking style as it is fundamentally different from traditional programming paradigms frequently taught to students, even if C++ is used for that purpose. Hence the education of students and researchers with regard to C++, object-oriented software development, and hpGEM itself plays an important role for the future prospects of the framework.

4.3 Software Engineering Aspects

Having decided to apply an object-oriented software development approach we looked for (software) tools to support in the object-oriented process, cf. Section 4. We will focus on

the tools used in the design and evolution phases of the object-oriented macro process in the following paragraphs.

Design. We have used UML [Booch et al. 1999] class, object, and sequence diagrams to specify the requirements and capabilities of the framework. The diagrams also serve as basis for the documentation, cf. Section 5.3 for an example.

Evolution. During the evolution phase we continue to use UML tools to reflect changes and extensions to the design. Further, we rely on the classical version management [CVS] and build [GNU make] tools. To guarantee portability and code quality we build hpGEM using Gnu and Intel C++ compilers for 32 and 64 bit architectures. Documentation is an integral part of the hpGEM development; in addition to an introductory document for new users, the source code is documented using the Doxygen documentation generator [Doxygen] which generates either online-browsable documentation in HTML, or an offline reference manual in \LaTeX from documented source files. To ensure that components of the hpGEM framework still meet their specification in the course of ongoing development, we test individual components, mostly on class level but also the collaboration between classes, by applying so called unit tests. To support the development of unit tests we use the CppUnit library [CppUnit].

5. DESIGN EXAMPLES

5.1 Mesh Interface

To apply a FE method to solve a set of PDEs, the domain Ω is discretized with a mesh that subdivides Ω into a number of simple shapes of suitably small size. On these elements, basis functions and other mathematical entities are defined, cf. Section 2. The setup of such a mesh is itself a complex task, but this step is hardly connected to the equations solved or the FE method. hpGEM encapsulates mesh setup as a complete service provided by the framework. Within a few commands that concern the origin and type, the user obtains the mesh and the framework is ready to work with it. Similar services are provided by other FE packages, as described in Section 3, but hpGEM differs from them in that it takes dimension-independent programming to the full, even on unstructured meshes. Based on the value of the dimension parameter `dim`, the commands

```
Mesh<dim> theMesh;
CentaurMeshFileReader<dim> Reader("mesh.hyb", theMesh);
```

will read the file `mesh.hyb` generated by the Centaur mesh generator [Centaursoft 2005], in particular unstructured meshes with mixtures of the possible element geometries: triangles and quadrilaterals in two-dimensional meshes and tetrahedra, pyramids, triangular prisms, and hexahedra in three-dimensional meshes. The elements and faces, including the appropriate geometry description, consisting of a reference geometry, the physical coordinates and a corresponding mapping, are generated automatically by suitable factory methods, cf. Gamma et al. [1994]. Note that the boundary faces are typically provided by the mesh generator. On the other hand, the internal faces, which are required by DG FEMs, are not. Hence hpGEM provides a general algorithm to generate them.

All information ends up in the `Mesh` class, whose most prominent task is to provide STL-compatible iterators to the element and face containers. Since most meshes in practice are unstructured (i.e. there is no global rule to identify neighbors; hence neighborhood

information has to be stored locally, for DG algorithms typically on the face) we do not provide other access than iterators. Consequently, a rectangular mesh on the unit square generated with the hpGEM class `RectangularMeshGenerator` by

```
PhysSpacePoint<2> p1; p1[0] = p1[1] = 0.; // lower left
PhysSpacePoint<2> p2; p2[0] = p2[1] = 1.; // upper right
unsigned int nrOfEl[] = { 8, 8 } ; // 8x8 elements
RectangularMeshGenerator<2> rmg(p1, p2, nrOfEl, theMesh);
```

will *not* have access by element coordinates (i, j) . Here again we prefer the generality of the approach with iterators: by exchanging the few code lines shown above for generating a rectangular mesh with those used earlier for reading a file from an (unstructured) mesh generator, the iterator-based program can work on any mesh in any dimension, while the version with indices or other direct access cannot.

Further services implemented by hpGEM as mesh processing steps are methods to

- (1) apply transformations to the nodes of a mesh,
- (2) make a mesh periodic,
- (3) add a (time) dimension to the given space coordinates.

The first two enable to adapt meshes to special geometries, e.g. a mesh on a cube can be used to discretize a torus. The last feature is needed for the so-called space-time DG methods, in which the space and time discretization are done in one step (rather than separately discretizing space and time), see for example Section 6.1 and [van der Vegt and van der Ven 2002]. The communication between all mesh generation-related classes takes place through abstract interfaces, so that the above tools may be combined in a chain.

To conclude, the philosophy of hpGEM is to provide front-ends to mesh sources, in particular (commercial) mesh generators, encapsulate the geometry information generation, and provide simple but universal means for unstructured mesh usage.

5.2 Integration

Due to the nature of finite element methods, their fine-grain building blocks are integrals over elements and faces, cf. Section 2, in particular Equation 4. Integration is linked to several other concept areas, e.g. the different reference geometries in a mesh (like triangles and quadrilaterals in 2d), algorithm-specific details (like the actual choice of an integration rule depending on various factors specific to the method), and the variety of mathematical objects to be integrated (e.g. scalars, vector fields). Therefore it has been one of the complicated design challenges, with the flexibility of the integration interface being a prerequisite for usability in various contexts.

While some of the integrals may be computable exactly (when the integrand is of simple analytical form), usually they are evaluated with numerical quadrature. In hpGEM, integration is based on the application of quadrature rules, i.e. a weighted sum of integrand evaluations at a set of points on the reference shape is computed, cf. Figure 3. The function call to compute an element integral takes a reference to the element over which to integrate, (optionally) the Gauss integration rule to be used, the function to integrate and a reference for the result storage:

```
integrateOverElement(element, quadRule, integrand, result);
```

$$\int_{E_i} f(\mathbf{x}) d\tau = \int_{\hat{E}_i} f(G_i(\hat{\mathbf{x}})) |Jac_{\hat{\mathbf{x}}} G_i(\hat{\mathbf{x}})| d\hat{\tau} \approx \sum_{p=1}^{N_p} \alpha_p f(G_i(\hat{\mathbf{x}}_p)) |Jac_{\hat{\mathbf{x}}} G_i(\hat{\mathbf{x}}_p)|$$

Fig. 3. The integral of a function f over the element E_i is transformed to the appropriate reference element \hat{E}_i through the mapping G_i . For the reference geometry, numerical integration rules are available as a weighted sum of function values at a set of points $\hat{\mathbf{x}}_p$ with weights α_p . The transformations are taken care of by the integration framework of hpGEM, and so is the computation of the Jacobian of the mapping, $Jac_{\hat{\mathbf{x}}} G_i$.

The transformation between reference and physical space takes place automatically by also evaluating the Jacobian of the transformation. Quadrature rules of orders up to at least 7 for all supported reference geometries (cf. Section 5.1) have been implemented based on Stroud [1971]. Since every integration rule is for a fixed dimension, they can be compiled once into a library, unlike C++ class templates, whose definition has to be available at compile time of the application. In fact, also the code for the integration rules is generated only for the preparation of the library. The code generation uses a simple description format, in which for example the first order integration rule on the $[-1; 1]$ reference line is given as

```
GaussRuleFromPoints C1d(
    "Cn1_1_1",           // name; cf. Stroud (1971)
    "centroid formula 1d", // explanation
    "ReferenceLine",    // reference geometry
    1);                 // order
C1d.addIntegrationPoint("2.0 [ 0.0 ]"); // weight&coordinate
```

Product rules can be constructed simply by giving the two lower-dimensional input rules. The code generation makes it easy to extend the set of integration rules without having to understand the details of how the rules are implemented. This includes that each rule is a singleton [Gamma et al. 1994] which guarantees that there will be only one instance of each rule in a program; further, each rule registers itself with the reference geometry it belongs to. That way, each reference geometry knows which rules are available to integrate on it and allows the user to apply selection criteria to find an appropriate rule. For that purpose, different possibilities exist, see Figure 4. A choice can be made, for example, based on the name of the rule or the approximation order it offers. The latter is particularly interesting as it allows to require that all integrals have to be computed with at least a certain order of accuracy. On the other hand the user can also decide on an element-by-element basis which rule to use; this feature is required for FEMs with p -refinement, i.e. the approximation order of the FE space is allowed to vary on different elements.

A crucial aspect of the integration routine is that the types to be integrated and the result type are templated. The condition on the integrand function is that it maps from a reference geometry to some linear space S , i.e. $\varphi : \hat{E} \rightarrow S$. In terms of C++ that means the function evaluation $s = \varphi(\hat{\mathbf{x}})$ for $s \in S$ and at the reference shape coordinate $\hat{\mathbf{x}} \in \hat{E}$ can be evaluated in the function call syntax as `phi(const RefSpacePoint<dim>& xi, S& s)`, i.e., it is either a function with this prototype or a functor with a corresponding `operator()`.

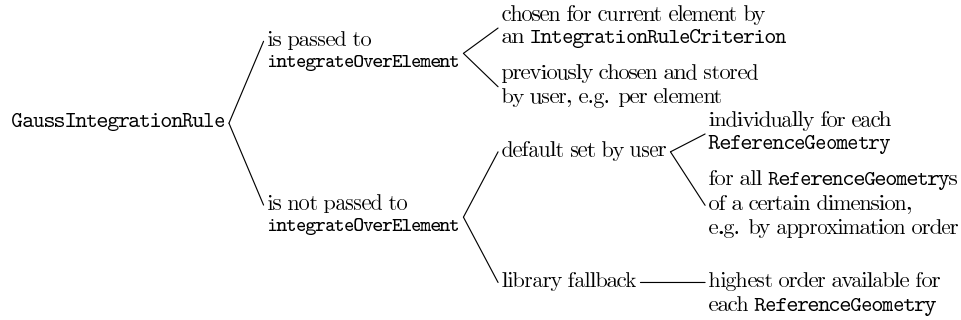


Fig. 4. Different ways to choose the numerical integration rule for computing an integral.

Via a traits system [Myers 1995] the result type of the integral is deduced from the input types. Using this system, also various products can be formed and integrated. For instance for the projection of a physical space function $f : \mathbb{R}^d \rightarrow \mathbb{R}$, $\mathbf{x} \mapsto f(\mathbf{x})$, onto a set of basis functions $\{\phi_j, j = 1 \dots n\}$ the integral $b_j = \int_E f(\mathbf{x}) \phi_j(\mathbf{x}) dE$ on the element E is computed as

```

integrateOverElement (E, integrationRule,
  scalarProduct<d>(E.transform2RefElement (f), phi (j)),
  b (j));

```

where `integrationRule` denotes a pointer to an explicitly chosen integration rule. The `transform2RefElement (f)` command wraps the user's function, which expects physical space coordinates, so that it can be queried at reference space points by the procedure `integrateOverElement`.

If the integrated function is not provided by hpGEM and its type is not included in the predefined traits, the user can extend these rules by template specialization. Similar functionality as described above for elements is also available for face integrals.

5.3 Fluxes

Another entity that was identified in the weak form (4) of the hyperbolic problem described in Section 2 is the flux function \mathcal{F} and its numerical counterpart $\hat{\mathcal{F}}$. The occurrence of \mathcal{F} in an element integral of the integrand $\nabla \phi_j \cdot \mathcal{F}$ is typical for conservation laws. The functional dependence of $\mathcal{F}(U)$ on the state variables U depends on the equation being solved and has to be specified by the user. For some equations and numerical fluxes, implementations are provided by hpGEM ready to use. But even for a flux that is not provided, the structure of the flux class family allows to implement additions to the framework quickly and in a reusable way.

The abstract base class `NumericalFlux` is derived from to fix the type of the state variable or vector, and the flux result, cf. Figure 5. Using the Euler flux as an example, they are both $(\text{dim}+1)$ -dimensional vectors when the space-time dimension is `dim`. Implementations of different numerical fluxes are in turn derived from this class, `NumericalEulerFlux`, so that they are polymorphic and can be exchanged without affecting user code. A possible implementation is the HLLC flux, cf. [Toro 1999]. Note that this is the only fully implemented class in Figure 5. The computation of fluxes also involves taking care of

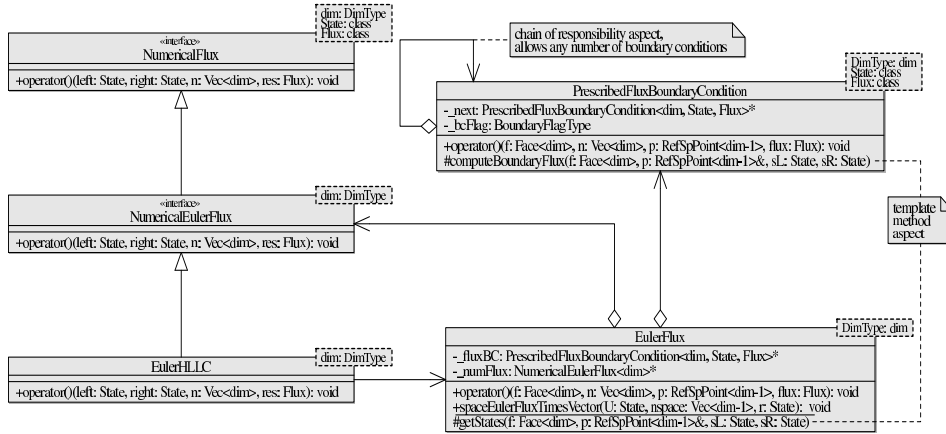


Fig. 5. Implementation perspective UML diagram of flux-related classes for the Euler equations in hpGEM. `spaceEulerFluxTimesVector` is the analytical flux function used for the $\nabla\phi_j \cdot \mathcal{F}$ term.

different boundary conditions: `EulerFlux` combines a `NumericalEulerFlux` with a number of implementations of `PrescribedFluxBoundaryCondition`, which treat different types of boundary conditions possible for the system (e.g. sub-/supersonic in-/outflow, slip). Several implementations of `PrescribedFluxBoundaryCondition` can be used for the different possible boundary conditions, by joining them in a chain of responsibility, cf. [Gamma et al. 1994]. Lastly, it should be noted that `EulerFlux` is still a template method [Gamma et al. 1994] since the `getStates` function, cf. Figure 5, must be implemented by the user. This function extracts the state at a given point; as hpGEM does not prescribe how and which data is stored per element, the task of evaluating states is referred to the user. The rest of the flux calculation does not depend on this detail, it just works with the given state. The described relationships and partitioning of responsibilities allow to vary every aspect of the flux computation independently.

6. APPLICATION EXAMPLES

The implementation of the hpGEM framework goes hand in hand with the development of sample applications. This ensures that the developed parts meet the requirements and furthermore are usable and beneficial from the practical viewpoint of scientists applying the provided means.

In the following, we present three of these sample programs, each treating a different mathematical problem. We start with a time-dependent hyperbolic PDE system which is solved on an unstructured mesh, cf. Section 6.1. The different stages of the program are specified and the role of hpGEM in their application is sketched. Verification results and some numerical solutions are given. Section 6.2 focuses on an elliptic PDE. In contrast to DG methods for hyperbolic problems, those for elliptic equations require solving a global system of equations. We show how hpGEM enables the user to assemble this system. The Poisson equation is solved in 2d on a mixed mesh with triangles and quadrilaterals. Finally, in Section 6.3, we treat an interface tracking problem for which the mesh has to be locally refined.

6.1 A Hyperbolic Problem: the Shallow Water Equations

We consider the space-time DG discretization of Ambati and Bokhove [2006a; 2006b] for the rotating shallow water equations (SWE). The depth-averaged shallow water equations consist of the flow field $\mathbf{U} = (h, hu, hv)^T$ with the depth-averaged velocity field $\mathbf{u} = (u(t, x, y), v(t, x, y))^T$, and flux tensor

$$\mathcal{F}(\mathbf{U}) = \begin{pmatrix} h & hu & hv \\ hu & hu^2 + gh^2/2 & huv \\ hv & huv & hv^2 + gh^2/2 \end{pmatrix}, \quad (5)$$

where $h(t, x, y)$ is the depth and g the gravitational acceleration. Beyond the hyperbolic system considered in Section 2, we have the source vector $\mathbf{S} = (0, f, -f)$ with the Coriolis parameter f . The discretization typically includes a numerical flux and a dissipation operator combined with a discontinuity indicator, resulting into a non-linear algebraic system. We employ a semi-implicit five stage Runge-Kutta method to solve this system.

The outline of the space-time DG algorithm implementation using hpGEM is:

- (1) We use the `CentaurMeshFileReader<dim>` to read a spatial mesh data file from Centaur; subsequently we generate space-time elements and faces by transforming with a `SpaceTimeMeshInterpreter<dim-1>`.
- (2) The data on the space-time elements is initialized and we loop over elements and faces with the element and face iterators provided by hpGEM. Integrals are computed using the functions `integrateOverElement` and `integrateOverFace` to obtain the nonlinear algebraic system per element.
- (3) We solve the system of nonlinear equations and write the solution output to a data file using a `TecplotDiscontinuousSolutionWriter<dim>`. Using Tecplot [Tecplot], we can immediately visualize our numerical solutions, cf. Figures 6 and 7.

From the user side, we mainly provide the flux tensor, source vector and numerical flux of the shallow water equations to build our numerical implementation. The numerical scheme is verified for its order of accuracy by considering a symmetric two-dimensional nonlinear exact solution of shallow water equations given in Section 4.1 of Ambati and Bokhove [2006b]. Figure 6(a) shows the numerical solution of the water depth $h(t, x, y)$ from $t = 0$ to 0.3 and Figure 6(b) documents the second order accuracy of the method using linear polynomials.

Further, we consider a Poincaré wave solution of the nonlinear SWE in a circular basin (as given by Ambati and Bokhove [2006a]), showing the ability of the package to deal with an unstructured mesh. Instead of simulating only one Poincaré mode, we include an additional Poincaré mode with the same amplitude but different frequency. These two Poincaré modes are simulated for one time period of the corresponding linear solution for the fast moving mode with high amplitude. The Poincaré modes not only disperse at different speeds but, because of the high initial amplitudes, also tend to break due to nonlinearity. This can be seen in Figure 7(b) and (c) as the contour lines come close to each other.

6.2 An Elliptic Problem: the Poisson Equation

In this section we consider the Poisson equation $-\nabla^2\phi = f$ on $\Omega \subset \mathbb{R}^{\dim}$ for the unknown ϕ , with Dirichlet boundary conditions and source term f in either two or three dimensions.

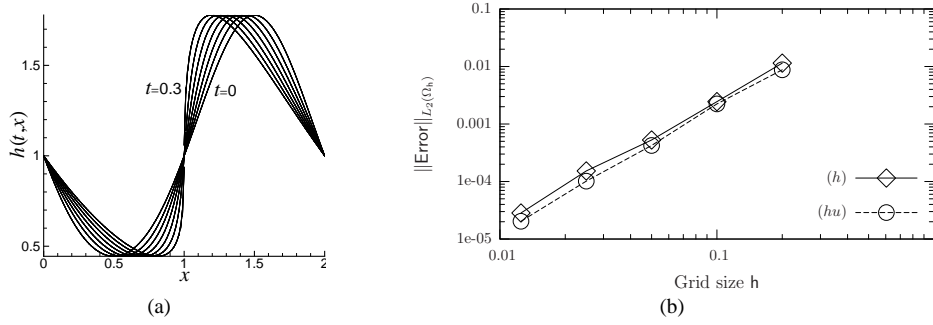


Fig. 6. a) Numerical solution of the SWE with 160 elements and b) L_2 error vs. grid size at $t = 0.2$ with an order of accuracy 2.2 for h and hu . Based on the analytical solution for the Burgers equation.

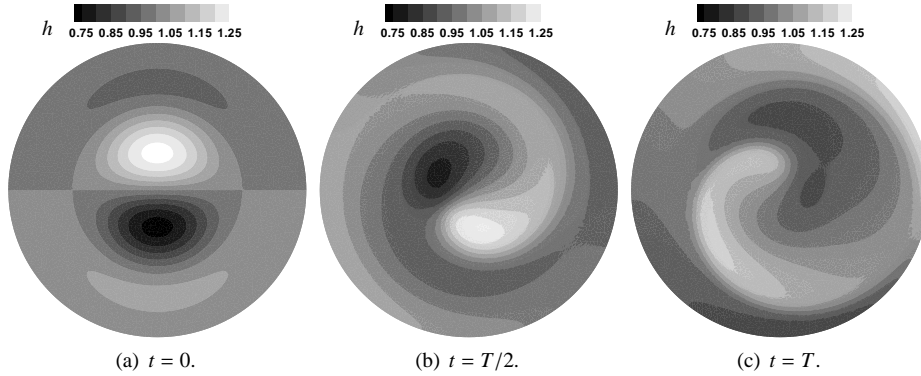


Fig. 7. Numerical solution for $h(t, x, y)$ for two Poincaré modes in a circular basin at time levels $t = 0, T/2$ and T , with T being the time period of the fast moving linear Poincaré mode.

We use the discontinuous Galerkin discretization presented by van der Vegt and Tomar [2005] which is based on the primal formulation developed in [Arnold et al. 2002]. In Algorithm 1 we present the (only slightly simplified) C++ main program of this application. Line 1 determines the dimension we are handling. In fact, it suffices to change this single line of the program to switch from a 2D to a 3D problem. The variable `myMesh`, holding the topological and geometrical information of the mesh is declared and filled in line 3. Here `StdRect` creates a Cartesian 8×8 mesh on the unit square. Replacing this line by a call to the Centaur mesh-file reader is the only required change if one wants to solve the Poisson equation for a Centaur-generated mesh. The global number of degrees of freedom is calculated in line 4 (the dots represent some parameters omitted here for brevity). Using this number we create the variable `UserData` that allows the user to store arbitrary data on each single element in line 5. In line 6 we declare the global matrix (`A`) and source- and solution-vectors (`b` and `x`, respectively). These variables are passed to the `GlobalLapackMatrixSorter` in line 7; this class is responsible for assembling the global matrix from the individual element and face contributions. It handles the mapping from local degrees of freedom per element to the global matrix entries internally. At this point

the setup phase of our hpGEM application is completed. Lines 8–9 do the actual work of assembling the global system by computing the element and face contributions. This is done using the `assembleElementContributions` and `assembleFaceContributions` calls, which dispatch the task to compute the local contribution on an element or face to the user-written functors `calcElCont` and `calcFaceCont`. The basic design of such a functor will be discussed below. The global system is solved in line 10 using the `gesv` routine from LAPACK, thereafter the computed solution is stored on the elements in line 11 and finally line 12 produces a Tecplot file containing the solution. We omit a discussion of the latter steps here.

```

A      1.
1. const DimType dim = 2; // problem dimension
2. int main() {
3.   Mesh<dim> myMesh; StdRect(myMesh, 8);
4.   const unsigned int nDof = countGlobalNrOfDOF(...);
5.   DataOnElements<ElementData> UserData(myMesh.NoEl());
6.   GlobMat A(nDof, nDof); GlobVec x(nDof), b(nDof);
7.   GlobalLapackMatrixSorter<ElementIDType> sorter(A, x, b);
8.   assembleElementContributions(myMesh,
           calcElCont<dim>(UserData),
           sorter);
9.   assembleFaceContributions(myMesh,
           calcFaceCont<dim>(UserData),
           sorter);
10.  lapack::gesv (A, b); x=b; // solve system using LAPACK
11.  sortSolutionBack(...); // return b to element data
12.  tecWriter.write(...); // write data for visualization
13.  return 0; }

```

Exemplarily for the user-written part of the construction of the global system we discuss the computation of the right-hand side contribution $\int_E \phi_i \cdot f(x, y, z) dE$. This contribution is calculated in the `calcElCont` functor, whose `operator()` is called by the function `assembleElementContributions` (see line 8 in Algorithm 1). The essential parts of its implementation are shown in Algorithm 2. The parameters of this functor are a reference to the element `el` for which the contribution is computed and an `ElementLocalSystemAcceptor` (in Alg. 1 the `GlobalLapackMatrixSorter`), which organizes the local contributions into a global system. In line 2 we omit the calculation and assembly of the left hand side contribution. The vector `G`, required to store the local contribution, is declared and initialized in lines 3–4, thereafter we loop over all local degrees of freedom (line 5) and compute the element integral given above in line 6, where `G_source()` is another functor implementing the source term. Note that the call to the integration routine does not depend on the dimension or the type of the element. The computed integration result is added to the global system in line 7.

```

A      2.
1. void operator()(const Element<dim>& el,
           ElementLocalSystemAcceptor<ElementIDType>& eAcceptor) {
2. ...

```

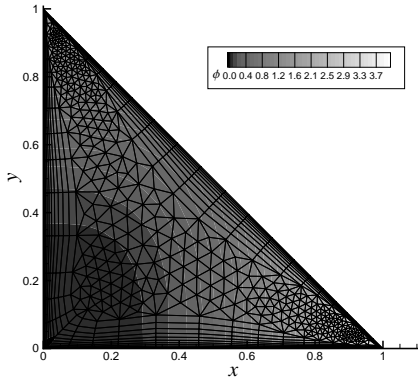


Fig. 8. Contour plot of the numerical solution of the Poisson equation on a triangle in 2d, for the source term $f = -24x - 12y$ and Dirichlet boundary conditions given by the exact solution $\phi(x, y) = 4x^3 + 2y^3$.

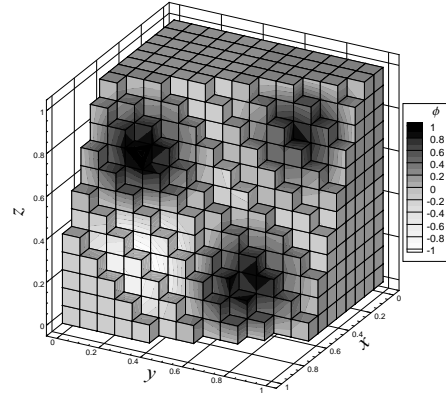


Fig. 9. Cross section of the numerical solution of the Poisson equation with $f = 12\pi^2 \sin(2\pi x) \sin(2\pi y) \sin(2\pi z)$ and Dirichlet boundary condition $\phi|_{\partial\Omega} = 0$ on the unit cube tessellated with 12^3 elements. The exact solution is $\phi(x, y, z) = \sin(2\pi x) \sin(2\pi y) \sin(2\pi z)$.

```

3. GlobalAssembly::LocalRHSType G;
4. G.resize(1NoDOF); G.clear();
5. for (int i = 0; i < 1NoDOF; ++i) {
6.   integrateOverElement(el,
       scalarProduct<dim>(BasisExpServer::basisFunction(i),
       el.transform2RefElement(G.source())), G(i));
7.   eAcceptor.assembleElementLocalRHS(el.id(), G, add); }

```

A numerical result computed on a two-dimensional mixed mesh for a triangle is shown in Figure 8. As explained above, after minimal changes the program can be used for computations in 3d, for which an example result is given in Figure 9.

6.3 An Interface Tracking Method

Using hpGEM, a new method for solving two-fluid flow problems in 2D space-time was implemented [Sollie et al. 2006]. The method combines the Cartesian cut-cell method and the level set method with the discontinuous Galerkin finite element method of van der Vegt and van der Ven [2002]. Like in the Cartesian cut-cell method, a background mesh is used, which is refined around the interface in such a way that all the elements in the refined mesh contain only one fluid. The element refinement is based on the interface position, which in turn is defined by the zero-level of the level set function. The level set function is advected with the interface velocity. We show an example of the dynamic mesh refinement for two approaching interfaces, based on the conservation of mass equation

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho a(x))}{\partial x} = 0, \quad \text{with initial condition } \rho(0, x) = \rho_0(x) = \begin{cases} 1.0 & \text{for } |x| \leq 2.5 \\ 2.0 & \text{for } |x| > 2.5, \end{cases}$$

$a(x)$ a given velocity and $\rho(t, x)$ the density, for which extrapolating boundary conditions are used. At the interface, special extrapolating interface conditions are prescribed and

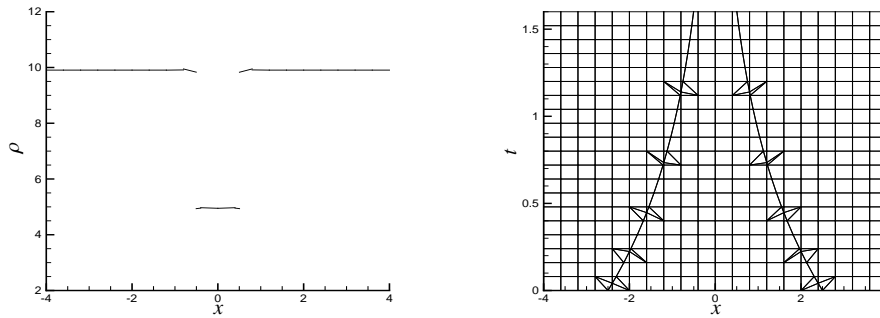


Fig. 10. Numerical solution at time $t = 1.6$ (left) and the space-time mesh using 20 elements (right).

the space-time interface flux is defined separately for the left and the right side. For the velocity $a(x) = -\alpha x$ was taken, with $\alpha = 1.0$ to ensure that the characteristics converge to $x = 0.0$. The exact solution is $\rho(t, x) = \rho_0(xe^{\alpha t})e^{\alpha t}$. To deal with the two interfaces, two level set functions and corresponding extension velocities $a(x)$ are used. The spatial domain is $[-4; 4]$ and time runs from $t = 0.0$ to $t = 1.6$. The numerical solution and the space-time mesh are shown in Figure 10. Due to the combination of mesh refinement and a suitably chosen numerical flux, the solution remains sharp at the interface and does not show many oscillations or dissipation. A small deviation occurs in the interface position, caused by errors in the level set function and by the piecewise linear approximation of the interface path in the mesh. In the code, hpGEM does all operations except for the two-fluid refinement parts, which are specific for this application.

7. CONCLUSIONS AND OUTLOOK

In this article we have introduced hpGEM, a general-purpose framework for the implementation of discontinuous Galerkin finite element discretizations. In contrast to several other finite element packages available, hpGEM is neither supposed to be a “solver” for a predefined set of equations nor is its focus to provide implementations of finite element algorithms. Rather it makes data structures and methods available that are general and frequently needed in the development and implementation of finite element application software. The framework provides means to define basis functions and expansions in terms of such bases but leaves decisions about which variables to store to the user. Consequently, hpGEM does not impose restrictions on the type or number of equations in the partial differential equation system. hpGEM handles general, unstructured meshes and all geometric transformations are handled internally. A convenient interface for the computation of integrals is provided.

Object-oriented programming techniques and design patterns are employed and lead to flexible, reusable software. C++ templates are used to support generic programming aspects, like dimension-independent programming, user-defined data classes, and function return types. They also allow to increase performance by template meta-programming techniques.

The framework does not implement any linear algebra solver capabilities but rather makes several existing packages available, which provide well-tested, efficient iterative

and direct solvers. The same holds for the necessary up- and downstream software, e.g. mesh generation and visualization tools.

As we have shown by means of several examples with different equations and numerical methods, a diverse range of applications has benefitted from being based on hpGEM. The ongoing and future developments will make it a more versatile tool. The geometric capabilities of hpGEM are currently extended by mesh refinement techniques as they were employed in the example in Section 6.3, to be used with general meshes. Apart from the more accurate geometric representation this also enables to apply multi-grid techniques, which will increase the efficiency of the solution process. Also we plan to expand on the linear algebra backends, so that a larger variety of sparse linear solvers—including parallel versions—are available to the framework user.

Another aspect of the future development of hpGEM is the broadening of its user community. While the development has so far been centralized in our research group, we intend to make it available to research collaborators as we are convinced this will benefit all users as well as hpGEM itself.

ACKNOWLEDGMENTS

L. Pesch acknowledges support by the Technologiestichting STW. O. Bokhove gratefully acknowledges support by a fellowship from The Royal Netherlands Academy of Arts and Sciences. J. J. W. van der Vegt acknowledges support through the national program BSIK in the ICT project BRICKS (<http://www.bsik-bricks.nl>), theme MSV1.

REFERENCES

- A , V. R. B , O. 2006a. Space-time discontinuous Galerkin discretizations of rotating shallow water equations on moving grids. *Submitted to J. Comput. Phys.*
- A , V. R. B , O. 2006b. Space-time discontinuous Galerkin finite element method for shallow water flows. *J. Comput. Appl. Math. (accepted)*.
- A , D. N., B , F., C , B., M , L. D. 2002. Unified analysis of discontinuous Galerkin methods for elliptic problems. *SIAM J. Numer. Anal.* 39, 5, 1749–1779.
- B , W., H , R., K , G. deal.II *Differential Equations Analysis Library, Technical Reference*. <http://www.dealii.org>
- B , O. 2005. Flooding and drying in finite-element Galerkin discretizations of shallow-water equations. Part I: One dimension. *J. Sci. Comput.* 22, 47–82.
- B , O., W , A. W., B , A. 2005. Magma flow through elastic-walled dikes. *Theor. Comput. Fluid Dyn.* 19, 261–286.
- B , G. 1994. *Object-oriented analysis and design with applications*, 2nd ed. The Benjamin/Cummings Publishing Company, Inc.
- B , G., R , J., J , I. 1999. *The Unified Modelling Language reference manual*. Addison-Wesley.
- C . 2005. CentaurTM Grid Generator. <http://www.centaurosoft.com/>
- C , B. 1999. *Lecture Notes in Computational Science and Engineering*. Vol. 9. Chapter Discontinuous Galerkin methods for convection-dominated problems, 69–224.
- C , B., K , G. E., S , C.-W., Eds. 2000. *Discontinuous Galerkin Methods. Theory, computation and applications*. Lecture Notes in Computational Science and Engineering, vol. 11. Springer, Berlin.
- C , B. S , C.-W. 2001. Runge-Kutta discontinuous Galerkin methods for convection-dominated problems. *J. Sci. Comput.* 16, 3 (Sept.), 173–261.
- CppUnit. C++ unit testing framework. <http://cppunit.sourceforge.net>
- CVS. CVS – Concurrent Versions System. <http://www.nongnu.org/cvs/>
- Doxygen. <http://www.stack.nl/~dimitri/doxygen/>

- Ganguly, E., Heston, R., Johnson, R., and Vignati, J. 1994. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- GNU make. <http://www.gnu.org/software/make/>
- Heston, J., and Loecherer, A. 2002. DOLFIN: Dynamic object oriented library for finite element computation. Preprint 2002-06, Department of Computational Mathematics, Chalmers University of Technology. April.
- Kang, C. M., and Vignati, J. J. W., and Vignati, H. 2006a. Pseudo-time stepping methods for space-time discontinuous Galerkin discretizations of the compressible Navier-Stokes equations. *J. Comput. Phys.* 219, 2, 622–643.
- Kang, C. M., Vignati, J. J. W., and Vignati, H. 2006b. Space-time discontinuous Galerkin method for the compressible Navier-Stokes equations. *J. Comput. Phys.* 217, 2, 589–611.
- hpGEM. The version discussed here is available from <http://wwwhome.math.utwente.nl/hpgemdev>
- Meyers, N. 1995. Traits: a new and useful template technique. *C++ Report* 7, 5 (June), 32–35.
- Rosen, J., Borner, M., Parnold, W., Ertl, F., and Lippman, W. 1991. *Object-oriented modeling and design*. Prentice Hall.
- Schwarz, A., and Sander, K. G. ALBERTA – An adaptive hierarchical finite element toolbox. <http://www.alberta-fem.de/>.
- Schwarz, W. E. H., Vignati, J. J. W., and Bode, O. 2006. A space-time discontinuous Galerkin finite element method for two-fluid problems. *To be submitted to J. Comput. Phys.*
- Singh, A. H. 1971. *Approximate Calculation of Multiple Integrals*. Prentice-Hall.
- Singh, J. J., Vignati, J. J. W., and Durrant, R. M. J. 2006. Space-time discontinuous Galerkin method for advection-diffusion problems on time-dependent domains. *Appl. Num. Math.* 56, 12, 1491–1518.
- Tecplot. <http://www.tecplot.com/>
- Tveit, E. F. 1999. *Riemann Solvers and Numerical Methods for Fluid Dynamics : A Practical Introduction*, 2nd ed. Springer.
- Vignati, J. J. W., and Tveit, S. K. 2005. Discontinuous Galerkin method for linear free-surface gravity waves. *J. Sci. Comput.* 22, 1, 531–567.
- Vignati, J. J. W., and Vignati, H. 1998. Discontinuous Galerkin finite element method with anisotropic local grid refinement for inviscid compressible flows. *J. Comput. Phys.* 141, 1, 46–77.
- Vignati, J. J. W., and Vignati, H. 2002. Space-time discontinuous Galerkin finite element method with dynamic grid motion for inviscid compressible flows: I. General formulation. *J. Comput. Phys.* 182, 2, 546–585.
- Vignati, H., Bode, O. J., Kang, C. M., and Vignati, J. J. W. 2005. Extension of the discontinuous Galerkin finite element method to viscous rotor flow simulations. In *Proceedings of the 31st European Rotorcraft Forum*. see also TW-Memorandum 1775, <http://www.math.utwente.nl/publications/>, Florence, Italy.
- Vignati, H., and Vignati, J. J. W. 2002. Space-time discontinuous Galerkin finite element method with dynamic grid motion for inviscid compressible flows. II. Efficient flux quadrature. *Comput. Methods Appl. Mech. Engrg.* 191, 41-42, 4747–4780.
- Vignati, H., Vignati, J. J. W., and Bode, E. G. 2003. Space-time discontinuous Galerkin finite element method for inviscid gas dynamics. *Comput. Fluid Solid Mech.* 1, 1181–1184.
- Vignati, D., and Johnson, N. M. 2003. *C++ Templates – The Complete Guide*. Addison-Wesley.
- Vignati, T. 1995. Using C++ template metaprograms. *C++ Report* 7, 4 (May), 36–43. Reprinted in *C++ Gems*, ed. Stanley Lippman.

Received?