Increased Efficiency In Finite Element Computations Through Template Metaprogramming

Karl Rupp
Christian Doppler Laboratory for Reliability Issues in Microelectronics
at the Institute for Microelectronics, TU Wien
Gußhausstraße 27–29/E360, A-1040 Wien, Austria
rupp@iue.tuwien.ac.at

Keywords: Template Metaprogramming, Finite Elements, Symbolic Integration,

Abstract

In the area of scientific computing, abstraction was long said to be achievable only in exchange for run time efficiency. With the rise of template metaprogramming [1] in C++ in the 1990s, run time efficiency comparable to hand tuned code could be achieved for isolated operations such as matrix-vector multiplication [2]. In more complex scenarios such as large finite element simulation packages, traditional object oriented programming is used for most abstractions, so the resulting code usually suffers from reduced run time efficiency. We have applied rigorous template metaprogramming to both the mesh handling and the mathematical algorithms acting on top, and obtain a high level of abstraction at a run time efficiency comparable to that of hand-tuned code. Since the weak formulation of the underlying mathematical problem is directly transferred to code, the code effectively meets the abstraction of the mathematical description, including an eventual independence from the underlying spatial dimension.

1. INTRODUCTION AND OVERVIEW

The mathematical description of the finite element method (FEM) is usually independent from the dimension of the problem domain Ω, provided that this holds true for the underlying (system of) partial differential equations (PDEs). Thus, one can obtain a high level of abstraction in FEM codes only if this holds true for the domain handling code (such as iteration over cells and subcells) as well. An abstraction of the underlying geometry is, if at all, in current FEM software packages typically achieved by the use of standard object oriented programming. The drawback of this approach is a significant decrease of run time efficiency due to type dispatches. Moreover, the code is often not flexible enough to selectively add or remove required functionality for a particular problem, hence a lot of CPU time and memory might be wasted for unnecessary calculations or data in such cases. Our approach to a flexible and fast domain management using template metaprogramming in C++ is presented in Sec. 2. It avoids unnecessary dispatches at run time and allows a flexible selection of the functionality needed for a particular problem at compile time.

In order to decouple algorithms acting on a certain geometry, a general way to store and access quantities on domain elements is necessary. Our approach works for quantities of arbitrary types and is discussed in Sec. 3.

The first step towards FEM discretizations is to derive the weak formulation, which is for second order PDEs obtained by multiplication with a test function v, integration over the whole domain and integration by parts. The generic procedure then is to select a suitable basis for a finite dimensional space of test functions and similarly for trial functions, which ultimately leads to a system of linear equations that can be solved by either direct or iterative solvers. More precisely, if the weak formulation can be written in the abstract form

$$\text{Find } u \in U \text{ s.t.} \quad a(u,v) = L(v) \quad \forall v \in V \tag{1}$$

with (in case of linear PDEs) bilinear form $a(\cdot,\cdot)$ and linear form $L(\cdot)$, the resulting system matrix $\mathbf{S}$ is given by

$$\mathbf{S} = (S_{ij})_{i,j=1}^{N}, \quad S_{i,j} = a(\varphi_j, \psi_i) \ , \tag{2}$$

where φ_j and ψ_i are a basis for the finite-dimensional spaces of trial and test functions respectively. A generic finite element implementation must therefore be able to evaluate bilinear forms for varying arguments. Moreover, the bilinear form is to be supplied by the user, thus the specification must be as easy and as convenient as possible. Our approach accomplishes this via extensive template metaprogramming and is discussed in Sec. 4.

By the use of metaprogramming, the full information about the weak formulation is then available at compile time. This allows to precompute so-called local element matrices over a reference cell already during the compilation process as is presented in Sec. 5.

Type dispatches in standard object oriented programming are carried out at run time. By the use of template metaprogramming, such dispatches are already resolved at compile time, hence the resulting code is expected to be faster at the cost of longer compilation times. We quantify this increase in compilation times and compare execution times of our new approach with existing finite element packages and a hand-tuned reference implementation in Sec. 6.

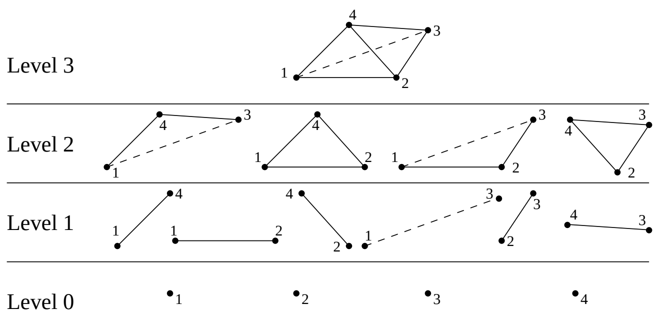


Figure 1. Topological decomposition of a tetrahedron

2. DOMAIN MANAGEMENT

For the implementation of algorithms for arbitrary spatial dimensions, a clean and general way to access mesh related quantities is necessary. Our approach is to first break cells down into topological levels [3] and then to use configuration classes that specify the desired implementation for each level separately. The subelements at lower topological levels of a tetrahedron are shown in Fig. 1. Elements on level zero are called *vertices*, on level one they are called *edges*, whereas elements of maximum topological level are called *cells* and elements with codimension one are called *facets*. A *domain* corresponds to the full problem domain and consists of at least one *segment*, which is the container for all elements located therein.

Our aim is to provide a simple means of customizing the domain management for the topological needs of a certain algorithm. The performance critical path in such algorithms is in many cases related to the iteration over domain elements of different levels, hence this functionality must be provided in a generic way at minimum computational costs. Consider for example a tetrahedral mesh and an algorithm that is a-priori known to iterate over edges only. In such a case it does not make any sense to explicitly store facets in memory, whereas this would make perfect sense for a different algorithm that needs to iterate over facets. To be able to handle both scenarios at minimum costs, each topological level can be configured separately via type definitions.

Following the ideas of *policy classes* and *tagging* [4], we define small classes that indicate or realize a specific behavior. The concept can be applied directly to polyhedral shapes, but for the sake of clarity we restrict ourselves to simplex elements in the following. The first step is the introduction of simple tag classes that hold the topological level (dimension) of the element:

```
struct VertexTag
  { enum { TopoLevel = 0 }; };
struct LineTag
  { enum { TopoLevel = 1 }; };
struct TriangleTag
  { enum { TopoLevel = 2 }; };
...
```

The next step is the specification of tags for the handling of each topological level of a cell, were we introduce the following two models:

- `TopoLevelFullHandling` indicates a full storage of elements on that topological level. For example, in a tetrahedron this tag specified for the facets means that all facet triangles are set up at initialization and stored within the cell.
- If `TopoLevelNoHandling` is used, the cell does not care about elements on that topology level.

A configuration class for each element type configures the desired implementation:

```
//declaration:
template <typename ElementTag_,
          long level>
struct TopologyLevel;

// topological description of
// a tetrahedron’s vertices
template <>
struct TopologyLevel<TetrahedronTag, 0>
{ typedef PointTag        ElementTag;
  typedef TopoLevelFullHandling
                          HandlingTag;

  enum{ ElementNum = 4 };  //4 vertices
};

// topological description of
// a tetrahedron’s edges
template <>
struct TopologyLevel<TetrahedronTag, 1>
{ typedef LineTag         ElementTag;
  typedef TopoLevelNoHandling
                          HandlingTag;

  enum{ ElementNum = 6 };  //6 edges
};

// similar for other elements and levels
```

The snippet above shows the configuration for a tetrahedron that stores its vertices, but does not store its edges. The actual realization of the storage scheme for subelements, say for triangular facets, are then again specified by additional partial specializations `TopologyLevel<`

TriangleTag, level> for each topological level level. This allows rather complex scenarios. Suppose an algorithm needs to iterate over all edges of all facets in the domain. Clearly, there is no need to store the edges on the cell, instead it is sufficient to store edges on the facets. Such behavior can be configured easily with the above configuration method. This way all the requirements for mesh handling can be encoded into a class hierarchy that can be evaluated at compile time and the compiler can then select the appropriate implementations.

The final domain configuration is again supplied by type definitions. For example, for a two-dimensional mesh consisting of triangles and double precision arithmetic, one defines

```
struct TriangleDomainConfig
{
  typedef double             CoordType;
  typedef TwoDimensionsTag   DimensionTag;
  typedef TriangleTag        CellTag;
  //several other type definitions here
};
```

An object of the domain type modeling the specified behavior above is then obtained by

```
domain<NewDomainTesting>  myDomain;
```

The automatically deduced types of all elements in the domain are obtained from a helper class DomainTypes:

```
typedef DomainTypes<TriangleDomainConfig>
            ::VertexType   VertexType;
typedef DomainTypes<TriangleDomainConfig>
            ::EdgeType     EdgeType;
//and so on
```

Iteration over different topological levels is implemented according to the iterator concept. This allows to iterate over arbitrary topological levels either globally over the whole segment or locally over sublevels of a topological (sub)element. The type retrieval is provided by a type container IteratorTypes, that takes as first template argument the type of the element *on* which iteration is to be carried out and as second argument the topological level of elements *over* which iteration is carried out. For example, in order to iterate over all edges of a cell cell of type CellType, one writes

```
typedef IteratorTypes
         <CellType,
          1>::ResultType     EdgeIterator;

for (EdgeIterator
   eit  = cell.getLevelIteratorBegin<1>();
   eit != cell.getLevelIteratorEnd<1>();
   ++eit)
{  /* do something */  }
```

This code is independent from the underlying dimension of the CellType. Since the topological level (in the above snippet equal to 1) is provided as template parameter, the compiler is able to eliminate all indirections at compile time and generate a very efficient executable. This is in contrast to traditional object-oriented programming, where the use of a single iterator class would lead to excessive run time dispatches in order to provide a similar functionality.

3. QUANTITY STORAGE

Since the domain management was not tailored to a particular algorithm, a convenient and fast way of storing and accessing quantities of arbitrary type is the key for the application of any kind of algorithms. Our approach is to provide a class QuantityManager, from which all topological elements are derived. The basic implementation of QuantityManager consists of two nested maps, where the first map uses the element addresses as keys and the second map accesses the desired quantity based on objects of user-defined key classes. Restrictions on the quantity types and the key types are very low and coincide with those of std::map from the C++ standard template library (STL). For example, storing and accessing quantities of type **double** and **bool** with keys of type **char** and **long** looks in the C++ code like

```
// 'element' is an arbitrary
// element of the domain
double quan1 = 23.0;
bool quan2 = true;
long key = 42;

// store quantities:
element.storeQuantity(key, quan1);
element.storeQuantity('c', quan2);

// access quantities:
double data1 =
   element.retrieveQuantity<double>(key);
bool data2 =
   element.retrieveQuantity<bool>('c');
```

Even though this quantity storage scheme is very general, its performance is not necessarily sufficient for high performance applications. The reason for that are the two nested maps, which allow access to data with access times of order $O(\log N + \log K)$, where N is the number of elements in the segment and K is the number of different keys of the same type for the quantity to be accessed.

If required, one can eliminate the logarithmic dependence on K by using different key types for quantities with different meaning. Such a scheme renders the second map obsolete, such that a single map with the element's address as key and the quantity as data parameterized by the type of the supplied key class is sufficient. The previous code snippet would then start with

```
KeyClassForPurposeA key1;
KeyClassForPurposeB key2;

// store quantities:
element.storeQuantity(key1, quan1);
element.storeQuantity(key2, quan2);
```

Even if `quan1` and `quan2` are of the same type, the correct quantities are stored because the type of the keys differ and hence the compiler generates two distinct codes. This way the run time dispatch based on *objects* of a certain key class is shifted to a compile time dispatch based on the *type* of the keys. However, this requires that these keys are already known at compile time, which is usually true at least for the performance critical path of mathematical algorithms operating on a mesh. To use this type based dispatch, the corresponding key types have to be registered at the `QuantityManager`.

For several performance critical tasks the now reduced access times of size $O(\log N)$ are still too large, especially if the number of elements N in fine meshes becomes very large. For such cases the `QuantityManager` allows to assign unique identifiers (within objects of the same time) of type **`long`** in the range $0, \ldots N-1$. These identifiers need to be manually assigned by the user, which is usually a simple task if these identifiers are already supplied by the mesh input file. The benefit is that instead of a map with the element's addresses as key it is then possible to use a vector that holds the quantities with the index given by the element identifier. The drawback of this solution is that extra care has to be taken to stay within the size of the vector, which can be controlled by

```
KeyClassForPurposeA key1;

// prior to any other use of
// that particular key type for
// every(!) quantity type used later on:
element.reserveQuantity<double>( key1 );
element.reserveQuantity<bool>( key1 );

// store and access as usual here.
```

Nevertheless, this way all quantity access times are reduced to $O(1)$ provided that only a single key object is used for each key type as discussed above.

4. EXPRESSION ENGINE

The domain management and the quantity storage scheme presented in the previous sections are independent from any FEM related requirements. However, since a superset of functionality required by FEM on the domain level is provided, the mathematical framework can now be built on top. The ingredients consist of a compile time representation of polynomials that are used as test and trial functions on the reference element, placeholders for these polynomials in the bilinear form $a(\cdot,\cdot)$, the linear functional $L(\cdot)$ in the weak formulation (1) of the underlying PDE, and the compile time representations of integrals.

Our implementation strongly relies on *expression templates* and uses a tree structure to handle mathematical expressions:

```
template <typename ScalarType,
          typename LHS,
          typename RHS,
          typename OP >
class Expression;
```

`ScalarType` denotes the underlying scalar type used for the arithmetic operations, `LHS` and `RHS` are the left and right hand side operands and `OP` encodes the type of the arithmetic operation. After adding appropriate template metafunctions one is then able to manipulate or even evaluate polynomials for fractional arguments at compile time. Moreover, one can eliminate trivial operations such as multiplications by one or addition of zeros directly within the syntax trees. While C++ is a procedural programming language, template metafunctions follow a functional programming paradigm and their implementation requires a different way of thinking. However, due to the recursive tree structure of the mathematical expressions the implementation of many template metafunctions is considerably more compact than procedural equivalents.

Placeholders for functions in the weak formulation have to distinguish between trial and test functions and have to take the order of differentiation into account. The first is achieved via a scalar template parameter, and the latter by nested differentiation tag classes:

```
basisfun<1>      // no differentiation
basisfun<1, diff<0> >         // d/dx
basisfun<1, diff<1> >         // d/dy
basisfun<1, diff<0,
                  diff<0> > > // d^2/dx^2
```

The representation of integrals in the weak formulation is driven by two tag classes that indicate the integration domain:

```
struct Omega {};

template <long id>
struct Gamma {};
```

where the first tag refers to integration over the whole segment and the latter to integration over (parts of) the boundary of the segment. The free template parameter `id` allows to distinguish between several not necessarily disjoint subregions of the boundary.

The final integral meta class follows the spirit of the previously introduced `Expression` class:

```
template <typename IntDomain,
          typename Integrand,
          typename IntTag>
struct IntegrationType;
```

IntDomain is one of the two tag classes, Integrand is an expression that encodes the integrand and IntTag is used to specify the desired integration method.

To demonstrate the flexibility achieved after all the above mentioned components are joined, let us consider the weak form

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x = \int_\Omega v \, \mathrm{d}x \quad \forall v \in V \, , \tag{3}$$

which is derived from the Poisson equation $-\Delta u = 1$. Transferred to code, (3) reads

```
assemble<FEMConfig>(segment, matrix, rhs,
    integral<Omega>( grad_u * grad_v ) =
    integral<Omega>( v ) );
```

The weak formulation can clearly be seen in the second and third line. grad_u and grad_v are both of type basisfun with suitable differentiation tags, integral is a convenience member function that generates the correct IntegrationType, and the assignment operator was suitably overloaded. The template parameter FEMConfig specifies the desired FEM related attributes such as the spaces of trial and test functions:

```
struct FEMConfig
{
  typedef ScalarTag       ResultDimension;
  typedef QuadraticBasisfunctionTag
                          TestSpace;
  typedef QuadraticBasisfunctionTag
                          TrialSpace;
  // further type definitions here
};
```

In this way, the specification of details of a particular finite element scheme is separated from the core of linear or linearized finite element iteration schemes, which is to loop over all functions from the test and trial spaces and to generate the system of linear equations from evaluations of the weak formulation at each such function pair.

5. INTEGRATION AT COMPILE TIME

Since the mesh is unknown at compile time, evaluations of the weak form (3) have to be carried out over each cell of the mesh at run time. The standard procedure is to evaluate the transformed weak formulation on a reference element and to transform the result according to the location and orientation of the respective element. This procedure is well described in the literature and makes use of so-called local element matrices [5]. The local element matrix $\mathbf{A}_\mathrm{e}(T)$ for a cell T is typically a linear combination of matrices $\mathbf{A}_k(T_\mathrm{ref})$ precomputed on a reference cell T_ref, thus

$$\mathbf{A}_\mathrm{e}(T) = \sum_{k=0}^{K} \alpha_k(T) \mathbf{A}_k(T_\mathrm{ref}) \, , \tag{4}$$

where K and the dimensions and entries of $\mathbf{A}_k(T_\mathrm{ref})$ depend on the spatial dimension, the underlying (system of) PDEs and the chosen set of basis functions. The scalars $\alpha_k(T)$ are the transformation coefficients from the reference cell T_ref to the cell T. While many FEM implementations use hard-coded element matrices, we use the fact that both the weak formulation and the test and trial functions are available at compile time in order to compute these local element matrices during the compilation. At present a compile time integration is supported for simplex cells only, because in that case the Jacobian of the transformation is a scalar and can be pulled out of the resulting integrals.

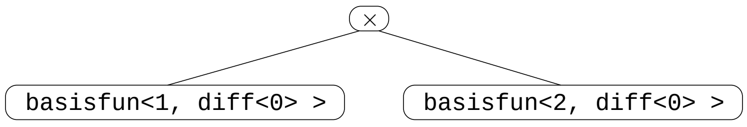
The transformation of integrals in the weak formulations such as (3) typically requires the transformation of derivatives according to the chain rule. Thus, this transformation also needs to be applied to the template expression tree as illustrated in Fig. 2 for the case of a product of two derivatives in two dimensions. The class dt_dx<i,j> is used to represent the entries of the Jacobian matrix. Since such a transformation is independent from the set of trial and test functions, it has to be carried out only once, keeping the workload for the compiler low. After expansion of the products and rearrangement, the weak formulation is recast into a form that directly leads to local element matrices as in (4). In a compile time loop the test and trial functions defined on the reference element are then substituted in pairs into this recast weak formulation and the resulting integrals are evaluated symbolically. This evaluation has to be carried out for each pair separately, thus a compile time integration cannot be applied to large sets of test and trial functions without excessive compilation times.

The resulting local integrals consists of summands of the form

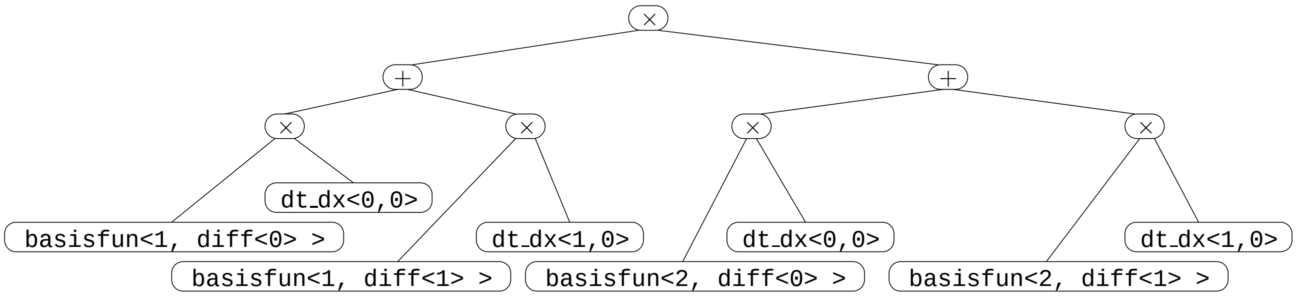
$$I_\beta = \int_{S_n} \left(\prod_{i=0}^{n-1} \xi_i^{\beta_i} \right) \left(1 - \sum_{i=0}^{n-1} \xi_i \right)^{\beta_n} \mathrm{d}\xi \, , \tag{5}$$

where S_n denotes the n-dimensional reference simplex with corners at $(0,0,\ldots,0)$, $(1,0,\ldots,0)$, $(0,1,0,\ldots,0)$, ..., $(0,0,\ldots,1)$, ξ_i denotes the i-th local variable and the multi-index $\beta = (\beta_i), \ i = 0,\ldots,n$ contains suitable exponents for the chosen test and trial spaces. The natural approach then is to carry out an iterated symbolic integration. Consider the integral

$$\int_0^1 \int_0^{1-x} xy^2 \, \mathrm{d}y \, \mathrm{d}x \tag{6}$$



(a) Initial expression tree.



(b) Expression tree after transformation.

Figure 2. Transformation of the expression tree representing $\partial u/\partial x_0 \times \partial v/\partial x_0$ to a two-dimensional reference cell.

with cubic integrand over the reference triangle. After integration with respect to y one obtains

$$\frac{1}{3}\int_0^1 x(1-x)^3\,\mathrm{d}x\,. \tag{7}$$

In order to carry out the remaining integration, one first has to expand the cubic term $(1-x)^3$. Such a procedure is too costly for a compiler, especially if this has to be repeated for many different integrands. To circumvent these problems associated with iterated integration we have derived the following analytic formula

$$I_\beta = \frac{\beta_0!\,\beta_1!\cdots\beta_n!}{(\beta_0+\beta_1+\ldots+\beta_n+n)!}\,, \tag{8}$$

which greatly reduces the compilation effort, because now it is sufficient for the compiler to bring all integrals into the canonical form (5).

6. BENCHMARK RESULTS

We have compared compilation and execution times for the assembly of the FEM system matrix for the Poisson equation with weak formulation as in (3) for different polynomial degrees of the trial and test spaces. In all our test cases the test space was chosen equal to the trial space and simplex cells were used. The compilation was carried out using gcc 4.3.2 on a machine with a Core 2 Quad 9550 CPU.

As can be seen in Tab. 1, symbolic integration at compile time leads to reasonable compilation times in one and two dimensions. In three dimensions one cannot go beyond cubic basis polynomials for the trial and test spaces without excessive compilation times. The reason is that there are already 20 different cubic test (and trial) functions, so the compiler has to compute 400 entries for each local element matrix. In case of a polynomial basis of degree four, 35 basis functions require to compute 1225 entries in each local element matrix, which is for current compilers too much to handle in a reasonable amount of time. Additionally, for more complicated weak formulations, compilation times are further increased due to a larger number of terms in the transformed weak formulation. Nevertheless, due to the often complicated computational domains in real-world applications it is in most cases sufficient to be able to cope with basis polynomials up to third order.

	1D	2D	3D
Linear	5s, 321MB	5s, 329MB	7s, 371MB
Quadratic	5s, 324MB	8s, 375MB	36s, 698MB
Cubic	6s, 326MB	12s, 457MB	424s, 1896MB
Quartic	7s, 328MB	35s, 760MB	-
Quintic	7s, 330MB	148s, 1230MB	-

Table 1. Compilation times and compiler memory consumption for several polynomial degrees of the test and trial functions with symbolic integration at compile time in different dimensions.

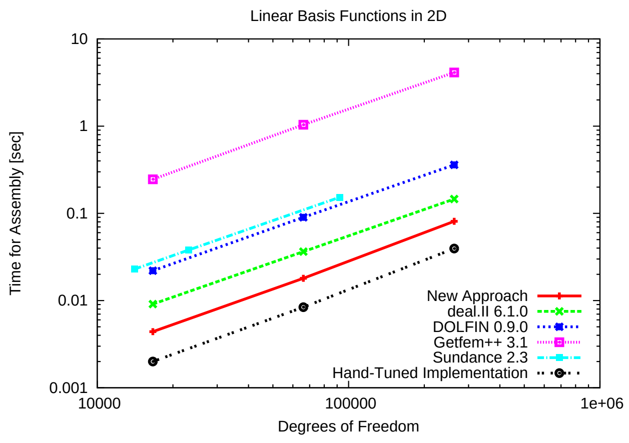
As a benchmark for the run time efficiency of our approach, we compared the full assembly process for the system matrix resulting from the weak formulation (3) of the Poisson equation with the freely available FEM packages `deal.ii` [6], `DOLFIN` [7], `Getfem++` [8] and `Sundance` [9]. Since the system matrix is stored with a sparse matrix format that differs in each package, we eliminated matrix access times by redirecting calls to a dummy matrix object with almost zero access times. Moreover, we added a hand-tuned reference implementation in `C` (tailored to that particular problem only) for linear and quadratic test and trial functions to the benchmark.

From the results in two dimensions depicted in Fig. 3 one can see that for linear basis functions the approach presented in this paper is the fastest among the general purpose FEM packages, only beaten by the hand-tuned reference implementation. The performance differences are surprisingly large: The reference implementation is about a factor of 100 faster than the slowest package. Since only low computational effort is needed for the evaluation of the local element matrices, higher weight is put on the access to global degrees of freedom and to the iteration over domain elements. For quadratic basis functions our approach even achieves the performance of the hand tuned reference implementation, while the variation in execution times among the packages becomes smaller. For cubic basis functions, only `Sundance` is faster than our approach. This test case puts most emphasis on the fast and efficient computation of the entries of the local element matrices and less emphasis on domain iterations. Thus, the reason for the faster assembly by `Sundance` is most likely that the transformed expression tree after compile time integration can still be evaluated more efficiently, for example in a factorized form [10]. Moreover, the performance differences between the fastest and the slowest implementation drops to a factor of about 20, therefore the larger differences in the case of linear basis functions is most likely due to differences in domain iteration and quantity access efficiency.

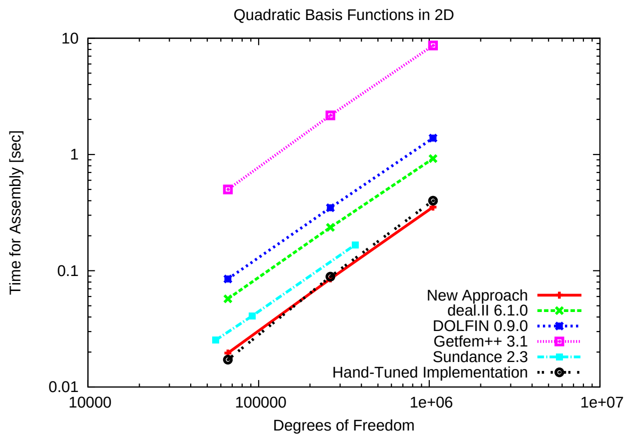
The results in three dimension as shown in Fig. 4 draw a similar picture: Our approach is in the case of linear basis functions only beaten by the hand-tuned reference implementation, `Sundance` is faster in the case of quadratic basis functions most likely thanks to a more efficient way of evaluating the integrated expressions. For cubic basis functions, our approach is the fastest among the remaining contestants, because `Sundance` does not provide an implementation for this test case.

With matrix access times included in the comparison, relative differences in execution times are smaller, though still significant. Our approach then has increased execution times of a factor of six to seven for the two-dimensional test cases and of a factor of about five, four and three for linear, quadratic and cubic basis functions in three dimensions respectively. Absolute increases in execution times of the same order of magnitude can also be observed at the other packages. It is interesting to note that our approach with matrix access times included is still faster than `Getfem++` without matrix access times.

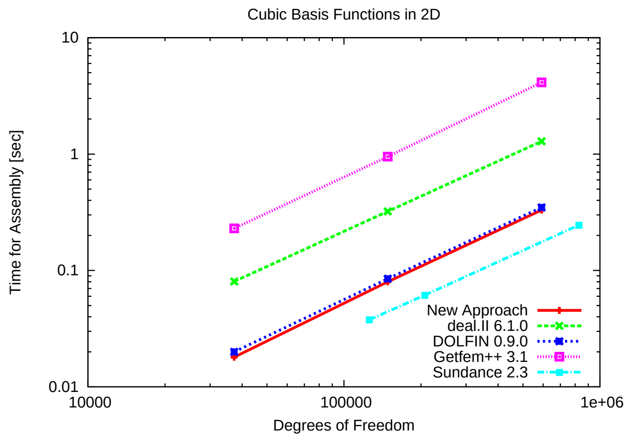
Once the system matrix and the right hand side vector are assembled, the system of linear equations still has to be solved. Especially for linear problems such as the Poisson equation used for our comparison here, the solution procedure typically takes considerably longer than the assembly process. For this reason one has to keep in mind that large dif-



(a) Linear test and trial functions.

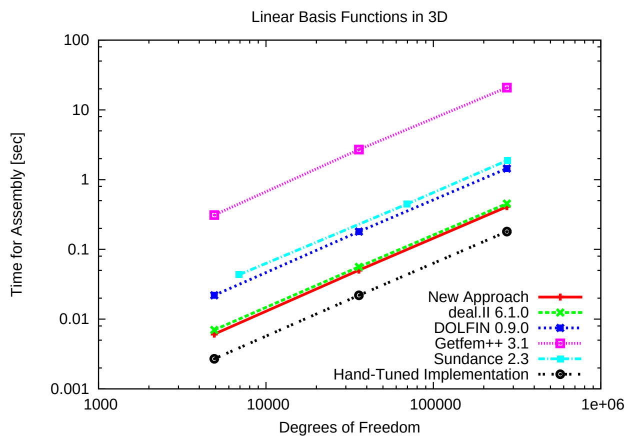


(b) Quadratic test and trial functions.

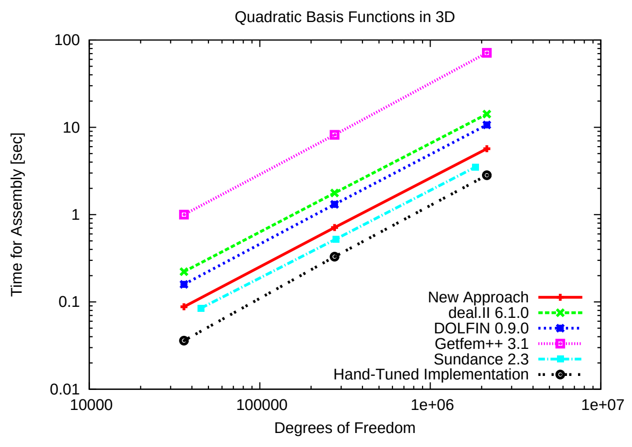


(c) Cubic test and trial functions.

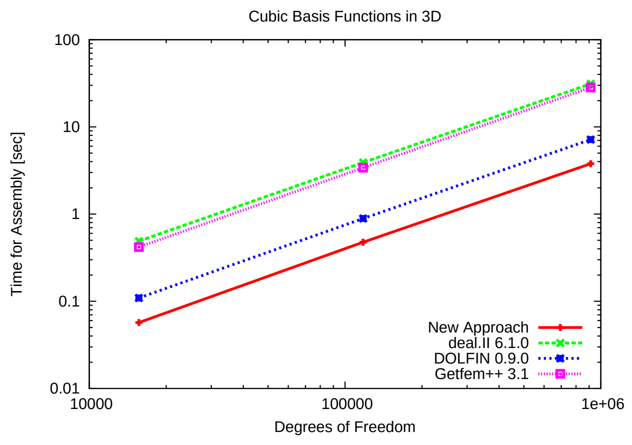
Figure 3. Run time comparison for the assembly of the stiffness matrix in two dimensions.



(a) Linear test and trial functions.



(b) Quadratic test and trial functions.



(c) Cubic test and trial functions.

Figure 4. Run time comparison for the assembly of the stiffness matrix in three dimensions.

ferences in assembly times such as those observed here lead to much smaller differences in the overall simulation time. The situation is likely to change for the assembly of nonlinear problems, where the assembly process has a much larger weight in the overall time budget, but we have postponed such a comparison to future work.

7. CONCLUSION

In this work we have shown that the use of template metaprogramming in C++ allows to increase both the level of abstraction from the implementation point of view and the run time efficiency of FEM implementations compared to traditional purely object-oriented approaches. By the strict orthogonalization of domain handling, quantity storage and algorithmics, clean interfaces of the individual components are found and the coupling is reduced to the absolute minimum. The only disadvantage of the approach are increased compilation times, which nevertheless still stay within a reasonable range for most problems.

REFERENCES

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, And Techniques From Boost And Beyond*. Addison-Wesley Professional (2004).

[2] T. Veldhuizen. “Expression Templates”. *C++ Report*, vol. 7, p.26-31 (1995).

[3] A. Logg. “Efficient Representation Of Computational Meshes”, *International Journal of Computational Science and Engineering*, vol. 4, no. 4, p.283-295 (2009).

[4] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley Longman Publishing Co., Inc. (2001)

[5] O. C. Zienkiewicz, and R. L. Taylor, *The Finite Element Method - Volume 1: The Basis*, Butterworth-Heinemann (2000)

[6] deal.II - Differential Equations Analysis Library. URL: `http://www.dealii.org/`.

[7] FEniCS project. URL: `http://www.fenics.org/`.

[8] Getfem++.
URL: `http://home.gna.org/getfem/`.

[9] Sundance. URL: `http://www.math.ttu.edu/˜klong/Sundance/html/`.

[10] R. C. Kirby., M. G. Knepley, A. Logg, and L. R. Scott. “Optimizing The Evaluation Of Finite Element Matrices”. *SIAM Journal on Scientific Computing*, vol. 27, no. 3, p.741-758 (2005).