Symbolic Integration At Compile Time In Finite Element Methods

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

ABSTRACT

In most existing software packages for the finite element method it is not possible to supply the weak formulation of the problem of interest in a compact form, which was in the early days of programming due to the low abstraction capabilities of available programming languages. With the advent of pure object-oriented programming, abstraction was long said to be achievable only in trade-off with run time efficiency. In this work we show that it is possible to obtain both a high level of abstraction and good run time efficiency by the use of template metaprogramming in C++. We focus on a mathematical expressions engine, by which element matrices are computed during compile time and by which the weak formulation can be specified in a single line of code. A comparison of system matrix assembly times of existing finite element software shows that the template metaprogramming approach is up to an order of magnitude faster than traditional software designs.

Categories and Subject Descriptors

D.1.m [**Programming Techniques**]: Miscellaneous; G.1.4 [**Numerical Analysis**]: Quadrature and Numerical Differentiation—*Automatic differentiation*; G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*Finite element methods*; G.4 [**Mathematical Software**]: Efficiency; I.1.3 [**Symbolic and Algebraic Manipulation**]: Languages and Systems—*Special-purpose algebraic systems*

Keywords

Template Metaprogramming, Symbolic Integration, Finite Element Methods, C++

1. INTRODUCTION

The level of abstraction in most software packages dealing with the finite element method (FEM) is low, mainly because for a long time programming languages could not provide facilities for a higher level of abstraction and thus low level programming approaches are extensively documented in the literature. With the advent of pure object-oriented programming, abstraction was long said to be achievable only in trade-off with run time efficiency, which is again one of the major aims of scientific software. In order to still achieve a reasonably high level of abstraction and good run time efficiency, program generators have been designed to parse higher level descriptions and to generate the required source code, which is then compiled into an executable form. Examples for such an approach are freeFEM++ [6] or DOLPHIN [5].

The introduction of another layer in the compilation process is in fact not very satisfactory: On the one hand, an input file syntax needs to be specified and parsed correctly, and on the other hand, proper source-code should generated for all semantically valid inputs. Moreover, it gets much harder to access or manipulate objects at source code level, because any modifications in the higher level input file causes another precompilation run, potentially producing entirely different source code. Thus, it pays off to avoid any additional external precompilation, and instead provide higher-level components directly at source code level. To the author's knowledge, the highest level of abstraction for FEM at source code level has so far been achieved by `Sundance` [10], which heavily relies on object oriented programming to raise the level of abstraction directly at source code level while reducing run time penalties to a minimum.

In this work we present a compile time engine for mathematical expressions obtained through template metaprogramming [1], so that the level of abstraction at source code level effectively meets that of the underlying mathematical description. Additionally, due to dispatches at compile time, any penalties due to virtual dispatches at run time are avoided. Since both the weak formulation of the underlying mathematical problem and the test and trial functions are available at compile time, we evaluate local element matrices symbolically by the compiler, so any unnecessary numerical integration at run time is avoided. Our approach only relies on facilities provided with standard-conforming C++ compilers to generate the appropriate code that finally enters the executable, thus no further external dependencies have to be fulfilled.

As driving example throughout this work we consider the Poisson equation

$$-\Delta u = 1 \tag{1}$$

in a domain Ω with, say, homogeneous Dirichlet boundary conditions. However, the techniques presented in the following can also be applied to more general problems with additional flux terms, a more complicated right hand side or different types of boundary conditions. The weak formulation of (1) is to find u in a suitable trial space such that

$$a(u, v) := \int_\Omega \nabla u \nabla v \, \mathrm{d}\mathbf{x} = \int_\Omega v \, \mathrm{d}\mathbf{x} =: L(v) \qquad (2)$$

for all test functions v in a certain test space. After discretization, the resulting system matrix $\mathbf{S}$ is given by

$$\mathbf{S} = (S_{i,j})_{i,j=1}^N, \quad S_{i,j} = a(\varphi_j, \psi_i) \ , \qquad (3)$$

where φ_j and ψ_i are the trial and test functions from the trial and test spaces respectively [3,13]. $\mathbf{S}$ is typically sparse due to the local support of the chosen basis functions.

In the following we assume that the system matrix $\mathbf{S}$ is fully set up prior to solving the resulting system of linear equations. Since typically iterative solvers are used for the solution of the linear system, it is in principle sufficient to provide matrix-vector multiplications and never set up the full system matrix. Our approach is also suitable for such a configuration, but for better comparison with other software packages in Sec. 5 and Sec. 6 we consider the case that $\mathbf{S}$ is set up explicitly.

According to (2) and (3), a generic finite element implementation must be able to evaluate bilinear forms for varying function arguments. Moreover, since the bilinear form is to be supplied by the user, its specification should be as easy and as convenient as possible. Consequently, we start with the discussion of higher-level components for the specification of the weak formulation in Sec. 2. The specification and manipulation of test and trial functions is outlined in Sec. 3. In Sec. 4 the higher-level components are joined in order to compute local element matrices at compile time. The influence on compilation times and execution times is quantified in Sec. 5 and Sec. 6 respectively.

2. EXPRESSION ENGINE

We have implemented high level components for the compile time representation of mathematical expressions in the style of *expression templates* [11,12]. The concept of syntax trees often used at run time was adapted to handle operators and operands at compile time:

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

`ScalarType` denotes the underlying integral 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. In the following we refer to this combination of expression templates and syntax trees as *expression trees.*

As we have seen in (3), the system matrix is build from plugging trial and test functions into the bilinear form. Consequently, we start with the introduction of placeholders for functions in the weak formulation, which have to distinguish between trial and test functions.

```
template <long num, typename diff_tag>
struct basisfun;
```

If the template parameter `num` is one, the class is a placeholder for a test function, otherwise it is a placeholder for a trial function. The template parameter `diff_tag` allows to specify derivatives of function for which `basisfun` is a placeholder. The differentiation tag `diff_tag` can also be nested:

```
// placeholder for a test function v
basisfun<1>
basisfun<1, diff<0> >      // for dv/dx
basisfun<1, diff<1> >      // for dv/dy

//placeholder for d^2 v/dx^2
basisfun<1, diff<0,
                  diff<0> >
         >
```

The latter allows for example to deal with PDEs of fourth order.

A key ingredient in weak formulations are integrations over the full domain, the full boundary or parts of the boundary. Our compile time representation of integrals in the weak formulation is driven by two tag classes [2] that indicate the desired integration domain:

```
struct Omega {};

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

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, where for example Neumann fluxes are prescribed.

The meta class representing integrals takes three template arguments: The domain of integration, the integrand and the type of integration (symbolical or numerical):

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

`IntDomain` is one of the two tag classes `Omega` and `Gamma`, `Integrand` is an expression that encodes the integrand, typically of type `Expression`, and `IntTag` is used to select the desired integration method.

After suitable overloads of arithmetic operators acting on `basisfun`, we are ready to specify weak formulations in mnemonic form directly in code. Let us again consider the weak form given in (2). Transferred to code, it reads in two spatial dimensions

```
basisfun<1>  v;
basisfun<1, diff<0> >  v_x;
basisfun<1, diff<1> >  v_y;
basisfun<2, diff<0> >  u_x;
basisfun<2, diff<1> >  u_y;

//the weak formulation:
integral<Omega>( u_x*v_x + u_y*v_y )
 = integral<Omega>( v );
```

Since the gradient in the weak formulation has to be adjusted whenever the spatial dimension of the underlying simulation domain changes, a convenience class `gradient` was introduced, which represents the mathematical object of a gradient in dependence of the spatial dimension. In principle, `gradient` can be generalized to act on arbitrary arguments and not just on basis functions. Summing up, the full assembly instruction for the weak formulation in (2) on a mesh

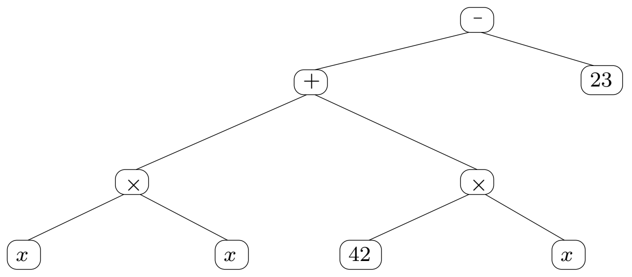


Figure 1: Compile time expression tree for the polynomial $x^2 + 42x - 23$.

object `segment` for a matrix `matrix` and a load vector `rhs` can now be written in a single statement in the mnemonic form

```
basisfun<1>   v;
gradient<1, dim>   grad_v;
gradient<2, dim>   grad_u;

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

The template parameter `FEMConfig` is a container of type definitions and specifies all 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. The benefit of this decoupling is that the only necessary change in the code when switching from quadratic to, say, cubic test and trial functions is to modify the two type definitions in `FEMConfig`, all other code remains unchanged.

Another advantage of separate configuration classes such as `FEMConfig` is that one could even switch between different families of discretization schemes. For example, a finite volume discretization could be indicated in another configuation class, e.g. `FVMConfig`. The end-user has to change only one line of code then, while totally different code is generated by the compiler.

The configuration class `FEMConfig` does not contain any information about the spatial dimension and other mesh-related parameters, thus the configuration is effectively independent of the underlying spatial dimension and fully decoupled from any mesh handling. By a highly flexible and clean interface to any mesh-related manipulations we have even managed to use the same code base for arbitrary spatial dimensions, but the discussion of such a domain management is beyond the scope of this paper.

3. POLYNOMIALS AT COMPILE TIME

With the specification of the weak formulation in the previous section, we now proceed with the discussion of test and trial spaces. Typically, these spaces consist of piecewise polynomials defined on a reference element and transformed to the physical elements in space. Therefore, we have implemented compile time representations of polynomials by reusing the `Expression` class defined in the previous section. A placeholder class `var` for variables like x or y was introduced, taking one integer parameter specifying the index of the unknown the placeholder represents. With suitable overloads of arithmetic and evaluation operators, polynomials can finally be defined and evaluated as

```
double result;
var<0> x;

//the polynomial x^2 + 42 x - 23
//evaluated at 0.5:
result = (x * x + 42 * x - 23)(0.5);

var<1> y;

//the polynomial x^2 - xy + y
//evaluated at (4.2, 1.3):
result = (x * x - x * y + y)(4.2, 1.3);
```

If polynomials are to be evaluated at real-valued arguments, the above code is the best one can get: The polynomials are encoded via template classes at compile time, while the evaluation is carried out at run time. This restriction to evaluation at run time is due to the fact that the present C++ standard does not allow floating point template parameters [8]. However, if the evaluation arguments are known to be integers (and also known at compile time), polynomials can directly be evaluated at compile time using template metaprogramming:

```
template <long arg, typename P>
double evaluate(P const & polynomial)
{
  return typename EVAL_POLYNOMIAL<P, arg>::
      ResultType()();
}

void main()
{
  double result;
  var<0> x;

  //the polynomial x^2 + 42 x - 23
  //evaluated at 1:
  result = evaluate<1>(x*x + 42*x - 23);
}
```

In contrast to the first code snippet, the expression tree of the polynomial is evaluated by the compiler in the metafunction `EVAL_POLYNOMIAL`. All occurrences of the tag class `var<0>` represented by `x` in the compile time expression tree are replaced with a wrapper for the scalar value 1, then the resulting expression tree is simplified by removing trivial operations, performing integer operations and the like. In the end, the return statement in the function `evaluate` is optimized by the compiler to `return 20.0;`.

In principle it is also possible to allow rational arguments, but due the limited range of integers one is soon confronted with overflows. For example, evaluation of the polynomial x^4 at the fractional 121/1000 leads to a denominator 10^{12}

and consequently an overflow. An alternative is to emulate floating point arithmetic at compile time, but the compiler performance was already reported to be atrocious due to the heavy manipulation work [9]. However, direct manipulation of the syntax trees such as replacing all occurrences of `y` with `z` is rather cheap, which is the key ingredient for the remainder of this section.

3.1 Symbolic Differentiation

For the assembly of the system matrix of our model problem (3), derivatives of basis functions (polynomials) are required. In earlier days, these derivatives were computed on the reference element by hand and the result was spread over relevant code lines. Thanks to template metaprogramming and the expression trees introduced in the previous section, the compiler can now compute the required derivatives. All that is left then is to specify the test and trial functions on the reference element.

The differentiation of polynomials is in fact very similar to evaluation. Instead of replacing the placeholder for the unknown with a scalar, we replace the unknown with its derivative, taking the basic rules of differentiation into account:

$$\begin{aligned}
(f+g)' &= f' + g' \ , \\
(f-g)' &= f' - g' \ , \\
(fg)' &= f'g + fg' \ , \\
(f/g)' &= (f'g - fg')/g^2 \ , \\
\partial x_i / \partial x_j &= \delta_{ij}
\end{aligned}$$

as well as the fact that derivatives of scalars vanish. Thanks to the functional paradigm of template metaprogramming, the implementation of the metafunction for differentiation is a direct, recursive application of these basic rules. Since the result of the differentiation operation is again an expression tree, we can directly apply the evaluation facilities shown above:

```
double result;
var<0> x;

//derivative of x^2 + 42 x - 23
//evaluated at 1 during compile time:
result = evaluate<1>(differentiate<0>(x*x +
    42*x - 23));

var<1> y;

//derivative w.r.t. y of x^2 - xy + y
//evaluated at (4.2, 1.3) during run time:
result = differentiate<1>(x * x - x * y + y)
    (4.2, 1.3);
```

The template argument of `differentiate` denotes the variable as defined by `var`. In the above snippet, `0` corresponds to a differentiation with respect to `x`, while `1` indicates differentiation with respect to `y`. The implementation of the function `differentiate` is similar to that of `evaluate`: It is a convenience wrapper for a call to the metafunction `DIFFERENTIATE`.

The application of the basic rules of differentiation at compile time may introduce several trivial operations such as multiplications by zero or unity into the compile time expression tree. For compile time evaluations, this is not an issue, but run time evaluations suffer from reduced performance. Consequently, every compile time manipulation is followed by an optimization step, eliminating trivial operations.

3.2 Symbolic Integration

Similar to evaluation and differentiation, the antiderivative of polynomials can also be obtained from compile time manipulations of the underlying expression tree. For the case that the integration bounds are integers, it is also possible to evaluate the antiderivatives at the bounds, hence we are able to compute definite integrals with integer bounds at compile time. Such integer bounds are typically the case for FEM, where reference elements are usually chosen to have corners at points with integer coordinates. For example, the integral

$$\int_0^1 x^2 + 42x - 23 \, \mathrm{d}x \tag{4}$$

can be evaluated at compile time by

```
double result;
var<0> x;

result = integrate<0,1>(x*x + 42*x - 23);
```

The function `integrate` is a convenience wrapper for the metafunction `INTEGRATE_POLYNOMIAL` and implemented similarly to the function `evaluate` defined previously. Moreover, also nested integration even in case that integral bounds depend on other integration variables have been implemented, which is needed for FEM in higher dimensions. Let us in the following consider the integral

$$\int_0^1 \int_0^{1-x} x(1-x-y)^2 \, \mathrm{d}y \, \mathrm{d}x \ , \tag{5}$$

which naturally turns up in higher-order FEM in two spatial dimensions if the reference element is chosen at points $(0,0)$, $(1,0)$ and $(0,1)$.

Our first approach was to carry out the full iterated integration. Each integration consists of the following steps:

- Expand the integrand until it is given as a sum of products of monomials.
- Integrate each summand separately.
- Determine the power of the integration variable in each summand.
- Replace the integration variable by the antiderivative.
- Subtract the term resulting from substituted upper bounds from the term resulting from substituted lower bounds.

Each step was implemented in a separate metafunction. The final iterated integration routine adds these separate metafunctions together and provides the desired functionality. However, one has to expect that the number of summands in the integrand explodes as the number of integrations increases, especially in the case that integral bounds depend on other integration variables.

To minimize compiler load for the integration over an n-simplex S_n with vertices located at $(0,0,\ldots,0)$, $(1,0,\ldots,0)$, $\ldots$, $(0,\ldots,0,1)$, as it is needed for FEM using triangular ($n=2$) or tetrahedral ($n=3$) elements, we have first derived the following formula:

$$\int_{S_n} \Big(\prod_{i=0}^{n-1} \xi_i^{\alpha_i}\Big)\Big(1 - \sum_{i=0}^{n-1} \xi_i\Big)^{\alpha_n} \,\mathrm{d}\xi \\ = \frac{\alpha_0!\,\alpha_1! \cdots \alpha_n!}{(\alpha_0 + \alpha_1 + \ldots + \alpha_n + n)!}. \tag{6}$$

This formula allows to avoid any costly iterated integrations, therefore it is sufficient to bring the integrand into the canonical form

$$\sum_k \left(\prod_{i=0}^{n-1} \xi_i^{\alpha_{i,k}}\right)\left(1 - \sum_{i=0}^{n-1} \xi_i\right)^{\alpha_{n,k}} \tag{7}$$

and integrate each summand separately. However, one has to bear in mind that the costly iterated integration is avoided at the cost of fixing the reference element.

Similar to differentiation, an optimization of the transformed expression tree is carried out as a final step. This results in a single rational number for each integral over the reference element and is in terms of efficiency comparable to hard-coding that particular value.

4. COMPUTATION OF ELEMENT MATRICES AT COMPILE TIME

Since the mesh is unknown at compile time, evaluations of the weak form (2) 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 [13]. The local element matrix $\mathbf{A}(T)$ for a cell T is typically a linear combination of matrices $\mathbf{A}_k(T_{\mathrm{ref}})$ precomputed on a reference element T_{ref}, thus

$$\mathbf{A}_{\mathrm{e}}(T) = \sum_{k=0}^{K} \alpha_k(T)\mathbf{A}_k(T_{\mathrm{ref}}) \ , \tag{8}$$

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. 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 (2) 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 of the mapping. Since such a transformation is independent from the set of trial and test functions, it has to be carried out only once during compilation, 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 (8). 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 as described in section 3.2. This evaluation has to be carried out for each pair of test and trial functions separately, thus a compile time integration cannot be applied to large sets of test and trial functions without excessive compilation times.

To circumvent the restriction to small sets of test and trial functions for symbolic integration at compile time, our implementation also supports numerical integration. A switch from symbolic to numeric integration is available within the code for the weak formulation:

```
// symbolic integration
integral<Omega>( grad_u * grad_v,
                 AnalyticIntegrationTag() )

// numerical integration, first order
integral<Omega>( grad_u * grad_v,
                 LinearIntegrationTag() )

// default: numerical integration, seventh
    order
integral<Omega>( grad_u * grad_v )
```

This allows to use several integration rules during the assembly: For integrands which are known to be very smooth, a low order quadrature rule can be assigned, while high order quadrature rules can be applied to less regular integrands.

It has to be emphasized that symbolic integration can only be applied in cases where coefficients in the weak formulation do not show a spatial dependence. For example, the weak form

$$\int_\Omega |\mathbf{x}|\nabla u \nabla v \ \mathrm{d}\mathbf{x} = \int_\Omega |\mathbf{x}| v \ \mathrm{d}\mathbf{x} \quad \forall v \in V \tag{9}$$

fails for symbolic integration at compile time due to $|\mathbf{x}|$ in the integrands. Nevertheless, in such a case one has to rely on numerical integration, unless the space dependent part is first projected or interpolated onto polynomials on the reference element.

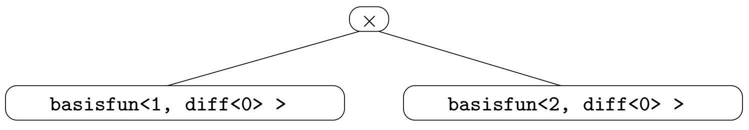
Hybrid approaches, where integrands without explicit spatial dependence are integrated at compile time and those with spatial dependence are integrated at run time, are also possible. However, they have larger compilation times due to the compile time integration, but hardly improve execution times because most time needs to be spent on the numerical integration anyway.

5. COMPILATION TIMES

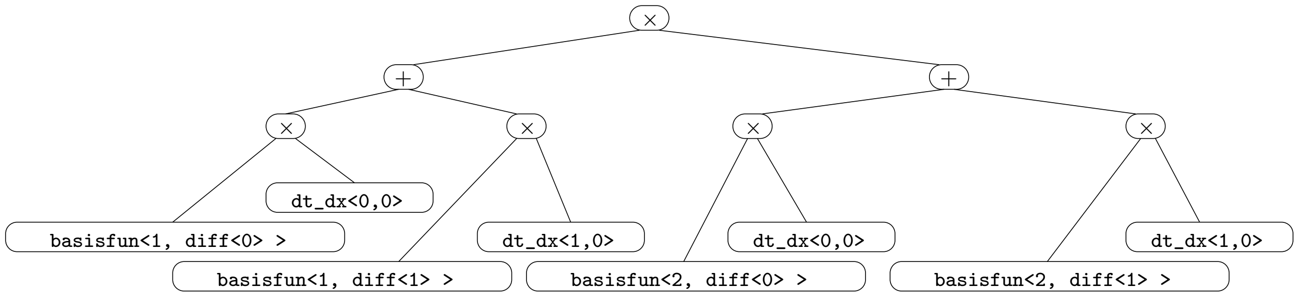
We have compared compilation for the assembly of the Poisson equation with weak formulation as in (2) for different polynomial degrees of the trial and test spaces. The benchmarks were carried out using `GCC 4.3.2` with optimization flag `-O3` on a machine with a Core 2 Quad 9550 CPU.

Compilation times for full iterated integration, i.e. integrating one variable after another for integrals as in (5), are shown in Tab. 1. In one dimension the numbers stay within an acceptable amount of two minutes. No iterated integrals have to be computed and the number of test and trial functions increases only linearly with the polynomial order. Nevertheless, more than one gigabyte of memory is required for test and trial functions of order five.

In two dimensions, full iterated integration works up to cubic polynomials, but fails to yield reasonable compilation times and memory requirements for polynomial orders larger



(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 element.

	1D	2D	3D
Linear	5s, 321MB	6s, 360MB	11s, 434MB
Quadratic	6s, 341MB	12s, 439MB	126s, 988MB
Cubic	7s, 363MB	384s, 1769MB	-
Quartic	15s, 442MB	-	-
Quintic	86s, 1112MB	-	-

Table 1: Compilation times and compiler memory consumption for several polynomial degrees of the test and trial functions with iterated symbolic integration at compile time in different dimensions. Dashes indicate that the compilation was aborted after ten minutes.

	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 2: Compilation times and compiler memory consumption for several polynomial degrees of the test and trial functions with formula-assisted symbolic integration at compile time in different dimensions.

than three. The reason for the breakdown is that the number of test and trial functions increases quadratically with the polynomial order and that the integrand gets considerably more complicated due to the polynomials terms.

In three dimensions, triple integrals have to be evaluated on the reference tetrahedron. This increased effort for the compiler leads to reasonable compilation times in the case of linear and quadratic test and trial functions only. Thus, full iterated symbolic integration of element matrices at compile time does not lead to reasonably short compilation times for polynomial order larger than two.

As can be seen in Tab. 2, symbolic integration at compile time using the derived formula (6) leads to reasonable compilation times in one and two dimensions for all test cases. 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 in three dimensions, so the compiler has to compute 400 entries for each local element matrix. In the 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 on current desktop computers too much to handle in a reasonable amount of time. A rough extrapolation estimates a compilation time of about 5000 seconds using eight gigabytes of memory for quartic polynomials in three dimensions. 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 many cases sufficient to be able to cope with basis polynomials up to third order.

Apart from compilation times there is another limiting factor for symbolic integration: The denominator in the term (6) produces an integer overflow at 13!, so in three space dimensions with $n = 3$, the criterion

$$\alpha_0 + \alpha_1 + \alpha_2 + \alpha_3 < 10 \tag{10}$$

has to be fulfilled. Since the sum of the exponents is roughly twice the polynomial degree of the test and trial functions, one cannot go far beyond degree four even if common factors in the fractional terms are cancelled.

Using numerical integration at run time, but no integration at compile time, the compiler load is much smaller and polynomial orders much larger than three in three dimensions can be handled within less then a minute of compilation times. The drawback of unnecessary numerical integration at run time can be circumvented by a suitable expression engine at run time, as it is implemented e.g. in `Sundance`.

	SI	NI, 1 Point	Exact NI
Linear	0.026	0.025	0.025
Quadratic	0.094	0.105	0.132
Cubic	0.36	0.65	2.17
Quartic	0.96	7.50	88.58
Quintic	1.7	35.9	462

Table 3: Comparison of assembly times (in seconds) for symbolic integration (SI) and numerical integration (NI) for different degrees of basis functions in two dimensions on a triangular mesh with 66049 **vertices.**

	SI	NI, 1 Point	Exact NI
Linear	0.0064	0.0069	0.0069
Quadratic	0.093	0.120	0.229
Cubic	0.47	0.65	2.82

Table 4: Comparison of assembly times (in seconds) for symbolic integration (SI) and numerical integration (NI) for different degrees of basis functions in three dimensions on a tetrahedral mesh with 4913 **vertices.**

6. EXECUTION TIMES

We have compared execution times for the assembly of the Poisson equation with weak formulation as in (2) 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 benchmarks were again carried out using `GCC 4.3.2` with optimization flag `-O3` on a machine with a Core 2 Quad 9550 CPU.

Matrix access times due to sparse matrix lookup times have been eliminated by redirecting all write operations to a fixed memory position, thus the measured times reflect the time needed to compute the matrix entries, the element transformation coefficients and the lookup times for the indices of the global system matrix. We have compared the symbolic integration with a numerical integration rule using one quadrature point and a quadrature rule with the minimum number of points needed to compute the respective integrals exactly. For polynomials of degree p, we have thus chosen a quadrature rule exact for polynomials up to degree $2p-2$, since according to (2) and (3) each integrand consists of a product of two derivatives of polynomials. The quadrature rule with only one integration point is used to compare the costs of a single evaluation of the integrand relative to other costs.

For a two-dimensional simulation domain with triangular elements, the results in Tab. 3 show that symbolic integration is very attractive for higher order methods. For linear basis functions, there is no notable difference between numerical and symbolic integration. For higher order polynomials we observe that even if only a single quadrature point is used, the increased effort needed to evaluate higher order polynomials leads to a severe difference in execution times of up to a factor of 20 for a quintic basis.

Similar results are obtained in three dimensions, c.f. Tab. 4. A noteable difference to the two-dimensional case is that symbolic integration leads to a slightly smaller execution times already in the case of linear polynomials. For higher order polynomials, the number of quadrature points increases as well as the effort needed for each evaluation, leading to much larger execution times compared to those obtained with symbolic integration. In the cubic case, the difference is already close to one order of magnitude.

	Linear	Quadratic	Cubic
Metaprog. Approach	0.052	0.74	3.78
deal.II 6.1.0 [4]	0.056	1.77	31.20
DOLPHIN 0.9.0 [5]	0.18	1.31	7.16
Getfem++ 3.1 [7]	2.73	8.21	28.37
Sundance 2.3 [10]	0.20	0.53	-
Hand-Tuned Ref.	0.022	0.33	-

Table 5: Execution times (in seconds) for the assembly of the system matrix for the Poisson problem. Linear and quadratic test and trial functions on a tetrahedral mesh with 35937 **vertices were compared. Matrix access times are not included.**

Additionally, we have compared assembly times of our symbolic integration approach with existing FEM software in the case of linear, quadratic and cubic basis polynomials in three dimensions. Again, we have eliminated matrix access times in order to emphasize assembly times. Due to the strongly varying software architectures among the packages, the measured execution times have to be taken with a grain of salt, since other components like for example mesh handling influence the result. The obtained results were compared with a hand-tuned reference implementation that should reflect the achievable performance. The selected packages differ significantly in their architecture: `deal.ii` requires the user to write large parts of the discretization herself. `DOLPHIN` relies on scripts from which C++ code is generated and therefore reflects the family of code generation approaches. `Getfem++` and `Sundance` allow to specify the weak formulation directly in code and parse it at run time.

As can be seen in Tab. 5, our approach leads to good run time efficiency only beaten by `Sundance` in the quadratic case. An interesting observation is the large spread between the execution times, which is more than one order of magnitude compared to the hand-tuned reference implementation. However, especially for simple linear PDEs the assembly times make up only a small amount of the total execution time, which also includes pre- and postprocessing steps and the solution of the resulting linear system. Therefore, differences in execution times for the full solution process show considerably smaller variation among the test candidates.

7. CONCLUSION

We have shown that the application of template metaprogramming together with its functional paradigm in C++ is very well suited for the representation of mathematical objects such as polynomials and operations such as integration or differentiation. The application to FEM allows an abstraction as high as the mathematical formulation so that the weak formulation can directly be transferred from paper to code. Unlike traditional object-oriented programming, template metaprogramming avoids unnecessary dispatches at run time, leading to excellent run time efficiency and short assembly times. Moreover, having the full weak formulation of the underlying mathematical problem available during compile time allows many other optimizations and manipulations at compile time that could have been achieved earlier only by separate, error-prone precompiler. The drawback of our template metaprogramming approach is the longer and memory demanding compilation process, which is still within reasonable limits up to cubic polynomials in three dimensions.

8. REFERENCES

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

[2] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[3] O. Axelsson and V. A. Barker. *Finite Element Solution of Boundary Value Problems: Theory and Computation.* Academic Press, Orlando, Fla., 1984.

[4] deal.II . Internet: `http://www.dealii.org/`.

[5] FEniCS project . Internet: `http://www.fenics.org/`.

[6] freeFEM++ . Internet: `http://www.freefem.org/`.

[7] Getfem++. Internet: `http://home.gna.org/getfem/`.

[8] ISO/IEC JTC1 SC22 WG21. The C++ Standard: ISO/IEC 14882:1998, 1998.

[9] E. Rosten. Floating Point Arithmetic in C++ Templates . Internet: `http://mi.eng.cam.ac.uk/~er258/code/fp_template.html`.

[10] Sundance 2.3. Internet: `http://www.math.ttu.edu/~klong/Sundance/html/`.

[11] D. Vandevoorde and N. M. Josuttis. *C++ Templates.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[12] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

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