Research Article

Finite Element Assembly Using an Embedded Domain Specific Language

Bart Janssens,¹ Tamas Banyai,² Karim Limam,³ and Walter Bosschaerts¹

¹*Department of Mechanics, Royal Military Academy, Avenue de Renaissance 30, 1000 Brussels, Belgium*

²*von Karman Institute for Fluid Dynamics, Chaussée de Waterloo 72, 1640 Rhode-Saint-Genèse, Belgium*

³*LaSIE, La Rochelle University, Avenue Michel Crépeau, 17042 La Rochelle Cedex 1, France*

Correspondence should be addressed to Bart Janssens; bart@bartjanssens.org

Received 18 November 2013; Revised 22 October 2014; Accepted 19 January 2015

Academic Editor: Bormin Huang

Copyright © 2015 Bart Janssens et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

In finite element methods, numerical simulation of the problem requires the generation of a linear system based on an integral form of a problem. Using C++ meta-programming techniques, a method is developed that allows writing code that stays close to the mathematical formulation. We explain the specifics of our method, which relies on the Boost.Proto framework to simplify the evaluation of our language. Some practical examples are elaborated, together with an analysis of the performance. The abstraction overhead is quantified using benchmarks.

1. Introduction

The application of the finite element method (FEM) requires the discretization of the integral form of the partial differential equations that govern the physics, thus transforming the problem into a set of algebraic equations that can be solved numerically. In essence, the resulting numerical model only depends on the governing equations and the set of basis and test functions. From the point of view of a model developer, it would therefore be ideal to only need to specify these parameters in a concise form that closely resembles the mathematical formulation of the problem, without sacrificing computational efficiency.

The current work is part of Coolfluid 3 [1], a C++ framework intended primarily for the solution of fluid dynamics problems and available as open source under the LGPL v3 license. The system is designed around a Component class that can provide a dynamic interface through Python scripting and a GUI. Model developers will typically write a set of components to build a solver for a given set of equations, using functionality provided by our framework and external libraries. Problem-dependent settings such as the mesh, boundary conditions, and model parameters can then be controlled through Python or a GUI, though these

aspects are beyond the scope of the current paper. In this context, we provide an Embedded Domain Specific Language (EDSL) that can be used to implement finite element solver components. It uses a notation similar to the mathematical formulation of the weak form of a problem, allowing the programmer to focus on the physics rather than coding details. We assume a typical finite element workflow, where a global linear system is assembled from element matrix contributions. Our language can also describe boundary conditions and field arithmetic operations.

All language elements consist of standard C++ code, so they easily embed into the components of our framework. An extension mechanism is available, allowing any developer to implement his own language elements without touching the library code. The implementation uses the Boost.Proto [2] framework, which builds on template meta programming techniques for generating efficient code. The actual performance overhead will be quantified and compared to the global solution time for some applications.

The automatic generation of code based on an intuitive specification of the physical problem is of course a feature that is highly desirable for any numerical simulation framework. One example of previous work providing such functionality is OpenFOAM [3], a C++ library primarily intended for fluid

mechanics. It allows for the easy expression of differential equations by providing an embedded tensor manipulation language, specific to finite volume discretizations. The FEniCS project [4] provides an easy and efficient method for developing finite element discretizations. The approach differs from ours in that the language is embedded into Python and compiled using a just-in-time compiler. It also offers the option of automated analytical evaluation of integrals [5]. Another example is the FEEL++ project [6], which also provides a domain specific language embedded into C++ using expression templates.

Notwithstanding the existence of all these excellent projects, we decided to implement our own framework for tight integration with the Coolfluid data structures and to support our workflow of building user-configurable components. The implementation relies on expression templates, just like FEEL++, but we use Boost.Proto to handle the expression template generation and to define a grammar for our language. This results in simpler code and has a limited impact on runtime performance, as we will show. Even though our language is not as feature-complete as the more established FEEL++ and FEniCS projects, we do believe that the easy integration of user-defined extensions into the language is a unique feature. The purpose of the current paper is to present how Proto can be used to construct a language for finite element modeling—based on a simple example building a mass matrix—and to show the capabilities and performance of the language we developed. Aspects that are new compared to previous work in the field are the use of Proto as a new way to construct the language and the possibility to add user-defined terminals to the language.

This paper is structured as follows. In Section 2 the mathematical formulation of the integral form will be laid out in order to clearly define the class of problems that is supported. Next, in Section 3 the mechanics for constructing and interpreting the EDSL are explained in detail, based on a simple application. Section 4 contains some application examples and Section 5 a performance analysis. Finally, in Section 6 we present our conclusions and suggest future work.

2. Finite Element Discretization

In this section, we introduce the notation used for finite element discretizations, starting from the Poisson problem as an example. A more generic and much more thorough introduction to the finite element method is given in, for example, [7]. Considering a domain Ω with boundary Γ , the differential equations describing the Poisson problem are

$$\nabla^2 f + g = 0 \quad \text{over } \Omega, \quad (1)$$

$$f = f_0 \quad \text{over } \Gamma. \quad (2)$$

To transform these continuous equations into a discrete problem, the unknown f is interpolated using a linear

combination of n shape functions $N(\mathbf{x})$ with the unknown coefficients f_i :

$$f \approx \sum_{i=1}^n N_i(\mathbf{x}) f_i = \tilde{f}. \quad (3)$$

The shape functions depend only on the spatial coordinates $\mathbf{x}$ and to simplify the notation we will from here on just write N instead of $N(\mathbf{x})$. The shape functions have a local support and the interpolation happens on a per-element basis, so n is equal to the number of nodes in an element. The interpolation can also be written in vector form as $\tilde{f} = \mathbf{N}_f \mathbf{f}_e$. Here, $\mathbf{N}_f$ is a row vector with the shape functions associated with unknown f and $\mathbf{f}_e$ is the column vector of the unknown coefficients for the element.

To obtain a linear system for the discrete problem, we need to multiply the equations with a weighting function and integrate over the domain. Using the Galerkin method (i.e., the weighting functions are the shape functions) as well as integrating per-element yields the following linear system for the Poisson problem, using the weak formulation here:

$$\sum_{e=1}^{n_\Omega} \underbrace{\int_{\Omega_e} \nabla \mathbf{N}_f^T \nabla \mathbf{N}_f \mathbf{f}_e \, d\Omega_e}_{A_{ff}} = \sum_{e=1}^{n_\Omega} \underbrace{\int_{\Omega_e} \mathbf{N}_g^T \tilde{g} \, d\Omega_e}_{\mathbf{a}_f}. \quad (4)$$

Due to the local support of the shape functions, the integral is written as a sum of integrals over all the elements. Each element integral results in an element matrix A_{ff} on the left hand side and an element vector $\mathbf{a}_f$ on the right hand side. Summing up the contributions of all elements that share a given node assembles a global linear system that has the total number of nodes as dimension.

The indices f are useful for problems with multiple unknowns. In this case, each unknown can be associated with its own shape function, and the element matrices and vectors are built up of blocks associated with each variable and equation.

The matrix assembly procedure is the same for all problems; only the values of the element matrices depend on the initial equation (1). This observation drives the current work: we will present a convenient language to write out the element equations and automate the steps that are common for all finite element discretizations. The code for the Poisson problem is presented in Listing 1, where lines (17) and (18) map directly to the element matrix and vector, as defined in (4), and lines (20) and (21) represent the assembly into the global system. Lines (17) and (18) form the core of our language, and the techniques for implementing this will be explained in detail in the next section. The remainder of Listing 1 will be further explained in Section 4.1.

From this simple example, the basic requirements for our language can be identified: provide access to the shape functions and their derivatives, as well as nodal values of the variables; control the evaluation of integrals; and allow the application of boundary conditions. Finally, we must be able to express the assembly into global matrices and vectors. The embedding of such a language into C++ is the main topic of the remainder of this paper.

FIGURE 1: The structure of the program, showing the three different levels of the implementation.

3. Construction of the Language

Our language consists of three levels of implementation, as shown in Figure 1. The top level contains the language itself, which consists of keywords that the user combines into an expression. Because the language is embedded, only valid C++ expressions are allowed, but we can define the interpretation freely via operator overloading. This is the job of the algorithm implementation layer. Here, the language is parsed and appropriate actions—called semantic actions in language grammar terms—are linked to each part of the input. Some of the actions may result in calls to a shape function library or matrix operations, so the final layer contains all external libraries needed to execute these.

The remainder of this section will focus on each layer in more detail, using a simple stand-alone example. This allows us to easily demonstrate the techniques used in Coolfluid 3 while avoiding the need to explain the details of our mesh structure and linear algebra interface, which are beyond the scope of this paper. The example will result in a program that builds the mass matrix for a finite element problem with a single scalar variable. It is a stand-alone program that depends only on the Boost libraries and on the Eigen library [8] for matrix operations.

The mass matrix M is assembled from the outer product of the weight and shape functions; that is,

$$M = \sum_{n_{\Omega}} \int_{\Omega_e} \mathbf{N}^T \mathbf{N} d\Omega_e. \quad (5)$$

The shape function has no index referring to a specific variable here, since we only consider a single, unspecified scalar variable. We want to ensure that the following code evaluates (5):

```
for_each_element(mesh, element_quadrature
(M += transpose(N) * N));
```

(6)

here, `for_each_element` represents a generic function, taking the finite element mesh structure as first argument and a Proto expression as second argument. The expression consists here of a call to `element_quadrature` to evaluate

FIGURE 2: The expression tree for line (19) in Listing 2. The function call operator is denoted by ().

the integral over an element and the `+=` operator should be interpreted as an assembly into the mass matrix M . The outer product of the shape function vectors maps directly to `transpose(N) * N`. In what follows, we show how we can evaluate this expression.

3.1. The Language Layer. Using the Boost.Proto [2] library, expressions are constructed from so-called “terminals.” These are C++ objects that are instances of the Proto terminal class, which overloads all of the C++ operators. Combining different terminals using operators builds an expression that can be evaluated later. This expression is analogous to the expression templates used in, for example, FEEL++ [6]. Listing 2 presents a minimal program that allows compiling the expression for (5). Line (19) contains the expression, built from the terminals defined at lines (11) to (13). Because all operations are defined on a Proto terminal, the code compiles. The result of line (19) is an expression tree, retaining the concrete type of each element in the expression. This is the same kind of data structure that we encounter in any expression template library, but it is generated automatically by Proto here.

Figure 2 shows the expression tree corresponding to line (19). Leaf nodes in this tree correspond to Proto terminals.

```

(1) // The unknown function. The first template argument is a constant to distinguish each
    variable at compile time
(2) FieldVariable<0, ScalarField> f("f", solution_tag());
(3) // The source term, to be set at runtime using an initial condition
(4) FieldVariable<1, ScalarField> g("g", "source_term");
(5)
(6) // Action handling the assembly
(7) Handle<ProtoAction> assembly = create_component<ProtoAction>("Assembly");
(8) // Set the expression
(9) assembly->set_expression(elements_expression
(10) (
(11) mesh::LagrangeP1::CellTypes(), // All first order Lagrange elements
(12) group
(13) (
(14)     _A = _0, _a = _0, // The element matrix and RHS vector, initialized to 0
(15)     element_quadrature // Integration over the element
(16)     (
(17)         _A(f) += transpose(nabla(f)) * nabla(f),
(18)         _a[f] += transpose(N(g))*g
(19)     ),
(20)     system_matrix += _A, // Assemble into the global linear system
(21)     system_rhs += _a
(22) )
(23) ));

```

LISTING 1: Implementation of the assembly procedure for the Poisson problem.

```

(1) #include <boost/proto/proto.hpp>
(2)
(3) using namespace boost;
(4)
(5) // Different types to distinguish terminals at compile time
(6) struct shape_func_tag {};
(7) struct element_quadrature_tag {};
(8) struct transpose_tag {};
(9)
(10) // Some statically created terminals
(11) proto::terminal<shape_func_tag >::type const N = {};
(12) proto::terminal<element_quadrature_tag >::type const element_quadrature = {};
(13) proto::terminal<transpose_tag >::type const transpose = {};
(14)
(15) int main(void)
(16) {
(17)     double* M;
(18)     // This is a proto expression:
(19)     element_quadrature(M+= transpose(N)*N);
(20)     return 0;
(21) }

```

LISTING 2: Generating expressions from Boost.Proto terminals.

Each function call operator (denoted by `()`) has two child nodes: the terminal representing the function itself (e.g., `transpose`) and the argument to the function. The terminal that represents the function is used to link it with the correct code of the function, based on the unique type of that terminal. This implies that functions can easily be renamed

by creating a new terminal of the same type. The binary operators `+=` and `*` also have two children, corresponding to the left and right hand side.

So far, the expression on line (19) does nothing. The parsing and the implementation of the appropriate actions to evaluate this are the subject of the next section.

```

(1) struct fem_grammar:
(2) // Match the following rules in order:
(3) proto::or_
(4) <
(5) // Evaluate shape functions using the eval_shape_func transform
(6) proto::when
(7) <
(8)     proto::terminal<shape_func_tag>,
(9)     eval_shape_func(proto::_data)
(10) >,
(11) // Evaluate transpose expressions using eval_transpose
(12) proto::when
(13) <
(14)     proto::function<proto::terminal<transpose_tag>, fem_grammar >,
(15)     eval_transpose(fem_grammar(proto::_child1))
(16) >,
(17) // Evaluate element_quadrature using eval_element_quadrature
(18) proto::when
(19) <
(20)     proto::function< proto::terminal<element_quadrature_tag>, proto::plus_assign<proto::
(21)         terminal<Eigen::MatrixXd>, fem_grammar>>,
(22)     eval_element_quadrature(proto::_value(proto::_left(proto::_child1)),
(23)         proto::_right(proto::_child1), proto::_data)
(24) >,
(25) // On any other expression: perform the default C++ action
(26) proto::_default<fem_grammar>
(27) >
(28) {
(29) }
(30) };

```

LISTING 3: Grammar capable of parsing a simple expression for interpolating coordinates using a shape function.

3.2. The Algorithm Implementation Layer. The algorithm implementation layer in Figure 1 takes care of the interpretation of expressions and the execution of the associated actions. In this section, we take a top-down approach for evaluating our example expression. To this end, we first define a class—grammar in Proto terminology—capable of parsing the expression. The class defined in Listing 3 describes the language that we use in our example expression. The grammar is defined entirely in a templated type from which `fem_grammar` derives. This notation is actually the embedded domain specific language provided by Proto itself for defining a grammar. The `or_` construct allows us to list a series of rules (each a Proto grammar in itself) that are checked in sequence. The `when` statement first describes what kind of syntax to expect, followed by a second argument—the semantic action—that describes how to evaluate the expression matching the `when` clause. In the case of terminals, we can use the type to distinguish them. The grammar can be used to check the validity of an expression at compile time, using the `proto::matches` metafunction, which does not evaluate the expression but simply checks if it matches a given grammar, taking into account only the `when` clauses. This functionality can be used to generate compilation errors that can help in finding errors in expressions, but this is not further discussed in the current work.

The main use of the grammars is the actual evaluation of the expressions. To this end, we can use `fem_grammar` as a C++ functor on an expression, in which case the grammar acts as a Proto “transform” (i.e., a kind of functor), evaluating the expression by executing the semantic actions embedded into the grammar. We will now describe each rule in the grammar in detail.

First, at line (6), shape function expressions are matched and evaluated. In (6), shape functions appear as the terminalN. The grammar matches this using `proto::terminal` with the appropriate type on line (8). The second argument in the `when` clause (line (9)) performs the evaluation. All the grammar code is inside template parameters, so everything is actually a type. Line (9) is then not really a function call, but the type of a function that returns something of the type `eval_shape_func` and takes `proto::_data` as argument. This function is never really defined, but Proto uses the type to call an `eval_shape_func` functor and pass its arguments. The argument consists of context data here—the “data” block from Figure 1—and will be discussed in detail later.

The implementation of the `eval_shape_func` functor is given in Listing 4. To be usable in a Proto grammar, a functor must inherit the `proto::callable` class. Lines (4) to (13) are used by Proto to compute the result type of the functor.


```

(1) struct eval_shape_func: proto::callable
(2) {
(3)     // C++ result_of declaration
(4)     template<typename Signature>
(5)     struct result;
(6)
(7)     // C++ result_of implementation
(8)     template<class ThisT, typename DataT>
(9)     struct result<ThisT(DataT)>
(10)    {
(11)        typedef const typename
(12)            boost::remove_reference<DataT>::type::element_t::shape_func_t& type;
(13)    };
(14)
(15) template<typename DataT>
(16) const typename DataT::element_t::shape_func_t& operator()(DataT& data) const
(17) {
(18)     // Return a reference to the result, stored in data
(19)     return data.shape_func;
(20) }
(21) };

```

LISTING 4: Functor to evaluate the shape functions.

```

(1) struct eval_transpose: proto::callable
(2) {
(3)     template<typename MatT>
(4)     Eigen::Transpose<MatT> operator()(MatT& mat) const
(5)     {
(6)         return mat.transpose();
(7)     }
(8) };

```

LISTING 5: Functor to evaluate the matrix transpose.

In C++11, it is possible to obtain this automatically, so this part of the code can be omitted. The function itself is defined on lines (15)–(20) and on line (19) we return the shape value functions, which are just cached in the context data. As will be seen later, this cached value is actually computed in the `element_quadrature` function.

The next grammar rule, on line (12) in Listing 3, matches and evaluates the matrix transpose. In the matching rule on line (14), it is prescribed that a function call using a terminal of type `transpose_tag` and with as argument an expression that matches `fem_grammar` is the correct form of a transpose expression. To evaluate the argument, we make a recursive call to `fem_grammar` itself (using `_child1` to isolate the function argument in the expression) and apply the `eval_transpose` functor to the result. Listing 5 shows the relevant code for evaluating the transpose. It is a functor that takes any matrix type from the Eigen library [8] as argument and calls the transpose function on line (6).

The rule describing the integration over an element starts on line (18) in Listing 3. On line (20), we impose that only

expressions of the `plus_assign` type, that is, `+=`, can be used as an argument to the `element_quadrature` function. This expression is then picked apart on line (21) using the `_left` and `_right` Proto operations, where the value of the left hand side (an Eigen matrix representing the global system matrix), the right hand side expression, and the context data are passed to the `eval_element_quadrature` functor. This functor is defined in Listing 6. On line (13), the numerical integration is performed by looping over the Gauss points defined for the element. First, the shape function values and Jacobian determinant are computed and stored in the context data (lines (15) and (16)). Next, the element matrix is updated with the value of the expression at the Gauss point, multiplied with the appropriate weight and the Jacobian determinant. The expression itself is evaluated using the call:

`fem_grammar()(expr, 0, data).` (7)

Here, our grammar is default-constructed (i.e., the first `()`) and then called using three arguments: the expression, the state (0 here), and the context data. The state is similar to

```

(1) struct eval_element_quadrature: proto::callable
(2) {
(3)     template<typename ExprT, typename DataT>
(4)     void operator()(Eigen::MatrixXd& mat, const ExprT& expr, DataT& data) const
(5)     {
(6)         // Temporary storage for the result from the RHS
(7)         Eigen::Matrix<double, DataT::element_t::nb_nodes, DataT::element_t::nb_nodes> rhs_result
(8)         ;
(9)         rhs_result.setZero();
(10)        // Loop over the gauss points
(11)        const typename DataT::element_t::gauss_points_t gauss_points = DataT::element_t::
(12)            gauss_points();
(13)        const typename DataT::element_t::gauss_weights_t gauss_weights = DataT::element_t::
(14)            gauss_weights();
(15)        for(int i = 0; i != DataT::element_t::nb_gauss_points; ++i)
(16)        {
(17)            data.shape_func = DataT::element_t::shape_function(gauss_points.row(i));
(18)            data.det_jacobian = DataT::element_t::jacobian_determinant(gauss_points.row(i), data.
(19)                coord_mat);
(20)            rhs_result += gauss_weights[i] * data.det_jacobian * fem_grammar()(expr, 0, data);
(21)        }
(22)        // Place the result in the global matrix
(23)        for(int i = 0; i != DataT::element_t::nb_nodes; ++i)
(24)            for(int j = 0; j != DataT::element_t::nb_nodes; ++j)
(25)                mat(data.node_indices[i], data.node_indices[j]) += rhs_result(i,j);
(26)    };

```

LISTING 6: Functor to perform numerical integration over an element and assemble the result into the global matrix.

the data, but we chose not to use it in the present work and just pass an integer. By supplying the context data, any shape function evaluations that occur in `expr` will use the correct value at the current Gauss point as computed on line (15). This also means that the shape function is evaluated only once, no matter how many times it occurs in the expression.

In this example, the element quadrature function also places the result in the global system matrix (i.e., `M` in (6)). This happens on lines (21)–(23), where the global node index for every entry in the element matrix is obtained from the context data and the corresponding global matrix entry is updated.

The final rule in the grammar (line (24) in Listing 3) is a fall-back to the default C++ behavior. It tells Proto to perform the default C++ action when none of the earlier rules are matched, recursively calling `fem_grammar` to interpret any subtrees. In the case of our example expression, it ensures that the matrix product is evaluated using the operators defined in the Eigen library, after evaluating the left and right operand using `fem_grammar`.

The definition of the context data used by the functors is given in Listing 7. It holds the knowledge about the concrete element type and the finite element mesh that we use. We are completely free to define this type as we see fit, since Proto passes it along as a template parameter. In our example, we provide storage for results such as the shape function

values and the node coordinates. The data changes for each element and is updated by the `set_element` function that is called by the element looping algorithm (still to be discussed). This function updates the node coordinates and the mapping between local element node index and global node index. The entire class is templated on the element type, which allows using high-performance fixed size matrices from the Eigen library.

Using predefined transforms from the Proto framework and user-defined function objects allows for the creation of grammars which can call themselves transforms recursively. This concise domain specific language describing grammars is a key feature of Proto: it makes it possible to indicate how to evaluate an expression without resorting to complicated metaprogramming techniques, resulting in code that is easier to read and maintain. The complexity of the grammar and the expressions it can parse is limited only by the time and memory available for the compilation.

To obtain a working code, we still need to implement the loop over the elements and generate the data that is used in expression evaluation. The data is templated on the concrete element type, but as can be seen from Listing 14 the mesh data lacks this information at compile-time. This is logical, since in a real application the user loads the mesh, so the software cannot know what concrete type is used at compile time. To select the element at run time, we need to

```

(1) template<typename ElementT>
(2) struct dsl_data
(3) {
(4)     // Required by Eigen to store fixed-size data
(5)     EIGEN_MAKE_ALIGNED_OPERATOR_NEW
(6)     // The concrete element type
(7)     typedef ElementT element_t;
(8)
(9)     // Construct using the mesh
(10)    dsl_data(const fem::mesh_data& d): mesh_data(d)
(11)    {
(12)    }
(13)
(14)    // Set the current element
(15)    void set_element(const int e)
(16)    {
(17)        for(int i = 0; i != element_t::nb_nodes; ++i)
(18)        {
(19)            const int node_idx = mesh_data.connectivity[e][i];
(20)            node_indices[i] = node_idx;
(21)            for(int j = 0; j != element_t::dimension; ++j)
(22)                coord_mat(i,j) = mesh_data.coordinates[node_idx][j];
(23)        }
(24)    }
(25)
(26)    // Reference to the mesh
(27)    const fem::mesh_data& mesh_data;
(28)    // Storage for the coordinates of the current element nodes
(29)    typename ElementT::coord_mat_t coord_mat;
(30)    // Global indices of the nodes of the current element
(31)    int node_indices[element_t::nb_nodes];
(32)    // Value of the last shape function computation
(33)    typename element_t::shape_func_t shape_func;
(34)    // Value of the last Jacobian determinant computation
(35)    double det_jacobian;
(36)};

```

LISTING 7: Data passed to the functors.

generate code for all the element types that are supported by the solver. In this example, we will support both 1D line elements and 2D triangle elements. The difference in the mesh data lies in the number of columns in the coordinates and connectivity tables, so we can use that to determine which mesh we are using. By checking for each supported element type if the mesh matches, we can execute the correct code at runtime. We use the MPL functor defined in Listing 8 to match the correct element type to the mesh. The functor is executed for each item in a list of allowed elements (line (37)), resulting in cleaner code than a long list of if-else statements. Once the element type check at line (13) passes, the data can be constructed and we can start the loop, updating the data for each element at line (23). The actual expression is executed using the grammar at line (24), in the same way as in the previously discussed element integration functor (Listing 6, line (17)). The expression itself remains a template parameter, allowing us to write any kind of Proto expression in a call to `for_each_element` and keeping all compile-time

information. Both of these properties are key advantages over a similar system that could be set up using virtual functions. By generating code for a list of predefined element types, the technique can be seen as a special case of generative programming [9], where the code to generate is defined through the MPL vector of element types.

In Coolfluid 3, the process is more complicated, since we can have different shape functions for the geometry and each variable that appears in an expression. This means we must now generate code for every possible combination of shape functions. We chose to organize the data so that each unknown that appears in an expression has its own data structure, aggregated into a global data structure for the expression. This approach makes it possible to have a unified data structure supporting expressions with an arbitrary number of variables. Another complication is the possibility to mix different element types in one mesh, for example, triangles and quadrilaterals. We solve this by organizing the mesh into sets that contain only one element type and then


```

(1) /// MPL functor to loop over elements
(2) template<typename ExprT>
(3) struct element_looper
(4) {
(5)     element_looper(const fem::mesh_data& m, const ExprT& e): mesh(m), expr(e)
(6)     {
(7)     }
(8)
(9)     template < typename ElemT >
(10)    void operator()(ElemT) const
(11)    {
(12)        // Bail out if the shape function doesn't match
(13)        if(ElemT::dimension != mesh.coordinates.shape()[1] || ElemT::nb_nodes != mesh.
            connectivity.shape()[1])
(14)            return;
(15)
(16)        // Construct helper data
(17)        dsl_data<ElemT> data(mesh);
(18)
(19)        // Evaluate for all elements
(20)        const int nb_elems = mesh.connectivity.size();
(21)        for(int i = 0; i != nb_elems; ++i)
(22)        {
(23)            data.set_element(i);
(24)            fem_grammar()(expr, 0, data);
(25)        }
(26)    }
(27)
(28)    const fem::mesh_data& mesh;
(29)    const ExprT& expr;
(30) };
(31)
(32) /// Execute the given expression for every element in the mesh
(33) template<typename ExprT>
(34) void for_each_element(const fem::mesh_data& mesh, const ExprT& expr)
(35) {
(36)     // Allowed element types
(37)     typedef mpl::vector2<fem::line1d, fem::triag2d> element_types;
(38)     mpl::for_each<element_types>(element_looper<ExprT>(mesh, expr));
(39) }

```

LISTING 8: The element looping algorithm.

deal with each set in turn. The complete algorithm to create the data and loop over a set of elements of the same type is as in Algorithm 1.

At this point we might wonder what happens if we try to evaluate an expression that makes no sense (e.g., $N + \text{transpose}$). Unfortunately, this often results in a very long list of compiler messages (430 lines for the example), exposing Proto implementation details and long template types. In the current example, it would be easy to fix using compile-time error messages, but for a more complex language grammar this is difficult and we have not yet implemented a satisfactory error handling system. We do intend to handle some common cases, and already errors from Eigen indicating incompatible matrix expressions come

through, but they are often obscured by a great number of other errors.

3.3. External Libraries. The example code uses the simplest possible data structures: the mesh simply contains arrays for the coordinates and the element nodes. The shape functions are simplified to only provide the functionality needed for the examples and they also include a fixed set of Gauss points for numerical integration. Listings 13 and 14 show the code for a first order 1D line element of the Lagrange family and the mesh data structure, respectively.

In Coolfluid 3, we provide our own mesh data structure, shape function library, and numerical integration framework. For operations at the element level, the Eigen [8] library is used, since it provides highly optimized routines for small,

```

(1) template<typename T> struct user_op {};
(2) struct my_callable {};
(3) // A terminal typed using the above structs
(4) proto::terminal< user_op<my_callable>>::type const my_op={};
(5) // Can be used as a function
(6) my_op(1,2,3);

```

LISTING 9: Definition of a terminal allowing user extension.

```

(1) proto::when
(2) <
(3) proto::function< proto::terminal< user_op<proto::>>, proto::vararg<proto::>>,
(4) evaluate_user_op(proto::function< proto::, proto::vararg<fem_grammar>>)
(5) >

```

LISTING 10: Extension of `fem_grammar` to add user-defined operations.

dense matrices. Finally, sparse matrix operations—required for the solution of the linear system arising from the element-wise assembly—are performed using the Trilinos [10] library.

3.4. User Defined Terminals. If we want to extend the language with new functionality, this currently requires modification to the grammar (Listing 3). Since this is part of the programming framework, it is not something that is supposed to be modified by the user of the language. To provide more flexibility, we allow users to define new terminals that can be used in expressions without modifying the grammar. To the best of our knowledge, this is a unique feature for this kind of embedded language. We will show here how it works, by providing an overview of how this could be added to our example code. We return to the definition of terminals, tagged using empty classes as in Listing 9. We created a terminal with the template class `user_op` as type. Its use as a function (line (6)) can be matched using the grammar:

```

proto::function<proto::terminal< user_
op<proto::>>,
proto::vararg<proto::>>

```

Here, `proto::` is a wildcard that matches any type and `vararg` allows the function to have any number of arguments. This kind of construct is exactly what we need to make `my_op` a user defined terminal, though we still have to provide a way to evaluate the expression. Adding a `when` clause as in Listing 10 yields the code that we need to add to `fem_grammar` in Listing 3. This uses `fem_grammar` to evaluate all function arguments and then calls the `evaluate_user_op` functor to evaluate the call (not listed here for the sake of brevity). In `evaluate_user_op`, we use the template argument to `user_op` (`my_callable` here) as a functor to evaluate the operator. This allows the user to define how the terminal should be evaluated without ever

needing to touch the core grammar, by simply implementing the correct function call operator in the type that is passed to `user_op`.

In Coolfluid 3, this technique is used to implement specialized code for certain solvers, directly setting entries of the element matrix when analytical expressions are known. Some core functions, such as the gradient and divergence operations, are also defined this way. Listing 11 shows the implementation for the divergence operation. On line (9), we have the function call implementation, where `var` is expanded into the data associated with the unknown for which we want to compute the divergence. This data is then used to access the gradient matrix (line (12)), much like we computed the shape function value in our previous example (i.e., line (15) in Listing 6). The terminal is statically created on line (30), using the `MakeSFOp` metafunction to avoid a long Proto type name. We can easily overload the function signature using additional function call operators. This is used on line (24) to provide an alternative divergence call that does not take a mapped coordinate as an argument but evaluates the divergence at the current quadrature point instead.

3.5. Integration into a Framework. So far, we have focused on building a small language, with the `for_each_element` looping function as entry point. In Coolfluid 3, our EDSL is used to implement reusable solver components, so before showing the concrete examples in Section 4 we have to present a few concepts about our framework and how it interfaces with our EDSL.

Each class in Coolfluid 3 is derived from the `Component` base class, which offers infrastructure to set options at run time and holds an arbitrary number of child components, thus creating a tree. A particular subclass `Action` implements the Command pattern [11], and we derive a class from that for working with expressions, called `ProtoAction`. Expressions can then be added to

```

Require: Mesh with field data for  $n$  variables
Require: A compile-time list of  $m$  shape functions  $N_i$  that may be used
Require: An expression  $E$ 
Ensure: Construction of context data of the correct type
  for all shape functions  $N_i$  do
    if  $N_i$  matches the geometry shape function then
      set geometry shape function:  $N_g = N_i$ 
    for all variables  $V$  do
      for all  $N_j$  compatible with  $N_g$  do
        if  $N_j$  matches the variable shape function then
          set  $N_V = N_j$ 
        end if
      end for
    end for
    create context data  $d$  using known  $N_g$  and  $N_V$  ( $\forall V$ )
    for all elements  $e$  do
      execute grammar  $(E, d)$ 
    end for
  end if
end for

```

ALGORITHM 1: Element looping algorithm.

```

(1) // Operator definition
(2) struct DivOp
(3) {
(4)   // The result is a scalar
(5)   typedef Real result_type;
(6)
(7)   // Return the divergence of unknown var, computed at mapped_coords
(8)   template<typename VarT>
(9)   Real operator()(const VarT& var, const typename VarT::MappedCoordsT& mapped_coords)
(10)  {
(11)    // Get the gradient matrix
(12)    const typename VarT::GradientT& nabla = var.nabla(mapped_coords);
(13)    Real result = 0.;
(14)    // Apply each component and return the result
(15)    for(int i = 0; i != VarT::EtypeT::dimensionality; ++i)
(16)    {
(17)      result += nabla.row(i) * var.value().col(i);
(18)    }
(19)    return result;
(20)  }
(21)
(22)  // Divergence at the current quadrature point
(23)  template<typename VarT>
(24)  Real operator()(const VarT& var);
(25)};
(26)
(27) // The terminal to use in the language
(28) // divergence(v, xi): compute the divergence at any mapped coordinate xi
(29) // divergence(v): compute the divergence at current quadrature point
(30) static MakeSFOp<DivOp>::type const divergence = {};

```

LISTING 11: Definition of the Coofluid 3 gradient operation, using a user-defined terminal.

```

(1) // Terminal that refers to the temperature
(2) FieldVariable<0, ScalarField> T("T", "temperature_field");
(3) // Create a new action, as a child component of parent
(4) Handle<ProtoAction> action = parent.create_component<ProtoAction>("Action");
(5) // Set an expression to loop over nodes
(6) action->set_expression(nodes.expression(T = 288.));
(7) // Execute the action
(8) action->execute();

```

LISTING 12: Loop over nodes, setting a temperature field.

an object tree by creating a `ProtoAction` and setting its expression using the `set_expression` function. The framework executes it transparently just like any other `Action`. Expressions in Coolfluid 3 can loop over either elements or nodes, using different grammars. The node expressions are primarily used for setting boundary and initial conditions and to update the solution.

We also need a way of accessing the mesh and the unknowns, which we do by making use of `FieldVariable` terminals. A basic action for looping over nodes could be added to a parent component as in Listing 12, in this case setting the temperature to 288 K. Here, the temperature is defined on line (2), indicating that it is stored as variable `T` in the field `temperature_field`. This seems redundant here, but fields can store more than one variable. Furthermore, the parser needs to be able to distinguish each variable at compile time, which is why each variable has a unique number in the first template argument. Each distinct variable that is used in the same expression must have a different number. The second template argument specifies whether we are dealing with a scalar or a vector. We then create the expression component and set its expression on line (6), choosing an expression that loops over nodes and simply assigns the value 288 to the temperature field here. Boundary and initial conditions are provided in a similar fashion, but there the component is parametrized on the field and variable names, so it can be reused in any solver.

3.6. Compatibility with Matrix Expression Templates. At the element level, matrix calculations are delegated to the Eigen library, which uses its own expression templates. As it turns out, such libraries cause problems when embedded into a Proto expression tree. A library like Eigen will, in the case of complex operations, create temporary matrices on the fly. When a temporary object is passed on through the functors that evaluate our expressions we return references to these temporary objects, resulting in memory errors. The problem is common to all modern matrix expression template libraries, and Iglberger et al. [12] review some cases where these temporaries might appear. To handle this issue, we preprocess the expressions using a special grammar: whenever a matrix product is found, the multiplication expression is modified to store a temporary matrix—allocated only once—that can hold the result of the multiplication. Any matrix multiplication result is then stored in the Proto expression

itself and a reference to it is returned during matrix product evaluation. The preprocessing happens by calling an additional functor in the element looping function, so the whole process is transparent to the user of the expressions and allows writing matrix arithmetic of arbitrary complexity.

4. Application Examples

In this section we work out a few examples to illustrate the mapping between the mathematical formulation and the code for different problems.

4.1. Poisson Problem. The Poisson problem was already used as introductory example in Section 2. Here, we provide some more details on the code from Listing 1 that was skipped in the introduction.

First, there is the declaration of the unknown `f` and the source term `g` on lines (2) and (4), respectively. This follows the same mechanism as in Listing 12, using a distinct number to distinguish the variables at compile time. On line (7), we create an `Action` component to hold the expression. Finally, the expression is set on line (9). The `elements_expression` function indicates that the loop will happen over the elements and is comparable to the `for_each_element` function in the simplified example. On line (11), the applicable element types are chosen (all elements from the first order Lagrange family here). The actual expression starts on line (12), with a call to `group` to combine multiple expressions into one. This is an advantage if multiple steps are required in the assembly, since they can be combined into one loop over the elements. The assembly makes use of an element matrix `_A` and element vector `_a`, which are set to zero on line (14).

An expression of the form `_A(f)` returns only the rows of the element matrix that refer to the equation for f (i.e., all rows in this case). If we pass a second variable as argument, the returned matrix only contains the columns that refer to it, so `_A(f, f)` would return a block that only contains the f contributions from the f equation (again, all rows and columns in this case). This notation is convenient in the presence of multiple variables to select only the relevant entries in an element matrix. The size of the element matrix is also determined by looking for blocks like this, so here we only use `_A(f)` to indicate that the element matrix contains

```

(1) /// 1D Line shape function
(2) struct line1d
(3) {
(4)     static const int nb_nodes = 2; // Number of nodes
(5)     static const int dimension = 1;
(6)
(7)     // Type of the mapped coordinates
(8)     typedef Eigen::Matrix<double, 1, dimension> coord_t;
(9)     // Type of the shape function vector
(10)    typedef Eigen::Matrix<double, 1, nb_nodes> shape_func_t;
(11)    // Type of the coordinates matrix
(12)    typedef Eigen::Matrix<double, nb_nodes, dimension> coord_mat_t;
(13)
(14)    // Compute the shape function vector at mapped coordinate c
(15)    static shape_func_t shape_function(const coord_t& c)
(16)    {
(17)        const double xi = c[0];
(18)        shape_func_t result;
(19)        result[0] = 0.5*(1.-xi);
(20)        result[1] = 0.5*(1.+xi);
(21)        return result;
(22)    }
(23)
(24)    // Compute the jacobian determinant
(25)    static double jacobian_determinant(const coord_t& mapped_coord, const coord_mat_t&
        node_coords)
(26)    {
(27)        return 0.5*(node_coords[1] - node_coords[0]);
(28)    }
(29)
(30)    static const int nb_gauss_points = 2;
(31)    // Type of the matrix with the Gauss points
(32)    typedef Eigen::Matrix<double, nb_gauss_points, 1> gauss_points_t;
(33)    // The Gauss points for the current shape function (definition omitted)
(34)    static const gauss_points_t gauss_points();
(35)    // Type for the weights
(36)    typedef Eigen::Matrix<double, nb_gauss_points, 1> gauss_weights_t;
(37)    // The Gauss weights (definition omitted)
(38)    static const gauss_weights_t gauss_weights();
(39) };

```

LISTING 13: Element type for first order line elements of the Lagrange family.

only a single equation for f . The same applies to the element right hand side vector `_a` using square brackets.

The element integral fills these matrices on lines (17) and (18), in a similar fashion to the simple example. The assembly into the global system is a separate step and occurs on lines (20) and (21). The `system_matrix` and `system_rhs` terminals keep track of wrapper objects for a Trilinos matrix and vector, using a generic linear solver API that is part of Coolfluid. They are initialized in a base class that provides the functionality for working with expressions and linear systems.

The code in Listing 1 builds a component that assembles the linear system. At run time, it can be combined with other components to add initial and boundary conditions and to solve the linear system, all of which can be controlled from a Python script.

4.2. Navier-Stokes Equations Using Chorin's Method. The Navier-Stokes equations for incompressible flow with velocity vector $\mathbf{u}$, pressure p , and kinematic viscosity ν are

$$\nabla \cdot \mathbf{u} = 0, \quad (8)$$

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} + \frac{\nabla p}{\rho} - \nu \nabla^2 \mathbf{u} = \mathbf{0}. \quad (9)$$

We will first solve this system of equations using Chorin's method [13], since this will allow us to easily compare performance later on with the same example from the FEniCS project. Chorin proposed an iterative time stepping scheme, computing first an auxiliary velocity $\mathbf{u}^{\text{aux}}$, followed by the pressure and finally the corrected velocity at the new time

```

(1) /// Unstructured mesh data
(2) struct mesh_data
(3) {
(4)     // Construct using a number of nodes and element properties
(5)     mesh_data(const int nb_nodes, const int nb_elems, const int nb_nodes_per_elem, const int
        dimension):
(6)         coordinates(boost::extents[nb_nodes][dimension]),
(7)         connectivity(boost::extents[nb_elems][nb_nodes_per_elem])
(8)     {
(9)     }
(10)
(11) // Global coordinates array
(12) boost::multi_array<double, 2> coordinates;
(13) // Global connectivity array
(14) boost::multi_array<int, 2> connectivity;
(15)};

```

LISTING 14: The mesh data structure.

step. The weak form (written again for a single element) for the $\mathbf{u}^{\text{aux}}$ equation is

$$\begin{aligned}
 & \left(\frac{1}{\Delta t} \underbrace{\int_{\Omega_e} \mathbf{N}_u^T \mathbf{N}_u d\Omega_e}_{T_{u_i u_i}} + \nu \underbrace{\int_{\Omega_e} \nabla \mathbf{N}_u^T \nabla \mathbf{N}_u d\Omega_e}_{A_{u_i u_i}} \right) (\mathbf{u}_e^{\text{aux}})_i \\
 &= \frac{1}{\Delta t} \int_{\Omega_e} \mathbf{N}_u^T \tilde{\mathbf{u}}_i^n d\Omega_e \\
 & - \int_{\Omega_e} \mathbf{N}_u^T \tilde{\mathbf{u}}^n \nabla \mathbf{N}_u (\mathbf{u}_e^n)_i d\Omega_e = \mathbf{a}_{u_i}.
 \end{aligned} \tag{10}$$

The superscript n indicates the known solution at time step n . The index i represents the i th component of the velocity here, which is a scalar in the case of the interpolated value $\tilde{u}_i^n$ and a vector of values for each node of the element for the unknowns $(\mathbf{u}_e^{\text{aux}})_i$. This means that, for each value of i , we insert a square block with dimension equal to the number of nodes into the element matrix and a corresponding segment into the element right hand side vector. We split up the element matrix as indicated by the braces, where $T_{u_i u_i}$ indicates the block corresponding to the rows and columns of component i of the velocity. The code to build this linear system is presented in Listing 15.

To get a stable solution, we will interpolate the velocity and pressure using second and first order shape functions, respectively (the Taylor-Hood element). If we restrict the solver to triangles, we can obtain this using the typedef on line (2). The actual shape function is chosen at run time, but we enforce the use of second order for the velocity here through the third constructor argument for the velocity variable on line (6). We use separate actions for the system matrix and right hand side assembly, since in this case the matrix coefficients do not change with time, so we can run the assembly procedure only once. The definition of the element matrices A and T is provided on lines (19) and (20). The Laplacian is written exactly the same as in the Poisson

problem (Section 4.1), but a new aspect is the introduction of the index $_i$ when indexing into the element matrix. It is automatically expanded into a loop over the number of physical dimensions, addressing the diagonal blocks of the element matrices as defined in (10). The same applies for the mass matrix on line (20), and both matrices are combined and assembled into the global system on line (22). The `auxiliary_1ss` component provides access to the terminals related to the linear system and the time. Here, `invdt` returns a reference to the inverse of the time step, which is set by the user running the simulation.

In the right hand side expression on line (34), the index $_i$ is used again, using `u[_i]` to get each component of the interpolated velocity vector $\tilde{\mathbf{u}}$. The last term represents the advection, and it requires access to a single component of the nodal values vector $\mathbf{u}_e$. We store the nodal values for a vector variable as a matrix, with each column corresponding to one physical component of the vector. This matrix is obtained using the `nodal_values` function while individual columns can be addressed using `_col`. The notation is a bit verbose, but if the user determines this to be a problem it is easy to introduce a user defined terminal for the advection operation.

The next step in Chorin's algorithm calculates the pressure, through the following Poisson problem:

$$\int_{\Omega_e} \nabla \mathbf{N}_u^T \nabla \mathbf{N}_u d\Omega_e \mathbf{p}_e^{n+1} = -\frac{1}{\Delta t} \int_{\Omega_e} \mathbf{N}_p^T (\nabla \mathbf{N}_p)_i d\Omega_e (\mathbf{u}_e^{\text{aux}})_i. \tag{11}$$

Listing 16 shows the code for this system, again using two assembly actions. For the right hand side assembly on line (22), we used the `divergence` function that was defined in Listing 11. We could also write this in terms of the nodal values matrix like this:

```

element_quadrature(_a[p]
+= transpose(N(p))
*nabla(u][_i]*_col(nodal_values(u),_i))

```



```

(1) // Allow a mix of first and second order shape functions
(2) typedef boost::mpl::vector2<mesh::LagrangeP1::Triag2D, mesh::LagrangeP2::Triag2D>
    LagrangeP1P2;
(3) // Kinematic viscosity, as a user-configurable constant:
(4) PhysicsConstant nu("kinematic_viscosity");
(5) // The velocity
(6) FieldVariable<0, VectorField> u("u", "navier_stokes_u.velocity", "cf3.mesh.LagrangeP2");
(7) // LSSActionUnsteady links with the linear algebra backend and the time tracking
(8) Handle<LSSActionUnsteady> auxiliary_lss =
(9)   create_component<LSSActionUnsteady>("AuxiliaryLSS");
(10) // The matrix assembly
(11) Handle<ProtoAction> auxiliary_mat_assembly =
(12)   auxiliary_lss->create_component<ProtoAction>("MatrixAssembly");
(13) auxiliary_mat_assembly->set_expression(elements_expression(LagrangeP1P2(),
(14) group
(15) (
(16)   A(u) = _0, T(u) = _0,
(17)   element_quadrature
(18)   (
(19)     A(u[_i], u[_i]) += transpose(nabla(u))*nabla(u),
(20)     T(u[_i], u[_i]) += transpose(N(u))*N(u)
(21)   ),
(22)   auxiliary_lss->system_matrix += auxiliary_lss->invdt()*_T + nu*_A
(23)   ));
(24)
(25) // RHS assembly
(26) Handle<ProtoAction> auxiliary_rhs_assembly =
(27)   auxiliary_lss->create_component<ProtoAction>("RHSAssembly");
(28) auxiliary_rhs_assembly->set_expression(elements_expression(LagrangeP1P2(),
(29) group
(30) (
(31)   a[u] = _0,
(32)   element_quadrature
(33)   (
(34)     a[u[_i]] += auxiliary_lss->invdt() * transpose(N(u))*u[_i] -
(35)                 transpose(N(u))*(u*nabla(u))*_col(nodal_values(u), _i)
(36)   ),
(37)   auxiliary_lss->system_rhs += a
(38)   ));

```

LISTING 15: Code to build the linear system for $\mathbf{u}^{\text{aux}}$ in Chorin's method.

On line (23), we call the `lit` function on `invdt`. This is actually a Proto function that constructs a terminal in-place, and it is needed to delay the evaluation of the minus sign, which would otherwise be evaluated right away by C++, resulting in the storage of a copy of the negative inverse timestep at expression creation, rather than a reference to the current value.

The final step of the algorithm updates the velocity, using the gradient of the newly calculated pressure:

$$\begin{aligned}
 & \int_{\Omega_e} \mathbf{N}_u^T \mathbf{N}_u d\Omega_e (\mathbf{u}_e^{n+1})_i \\
 &= \int_{\Omega_e} \mathbf{N}_u^T \tilde{\mathbf{u}}_i^{\text{aux}} d\Omega_e - \Delta t \int_{\Omega_e} \mathbf{N}_u^T (\nabla \mathbf{N}_p)_i \mathbf{P}_e^{n+1} d\Omega_e.
 \end{aligned} \tag{12}$$

The system matrix is the mass matrix, so as seen in Listing 17 we assemble it in its own action. Note the similarity here with

the stand-alone example equation (6). The gradient function on line (18) is defined using the user defined function mechanism, and just as is the case with the divergence it can be written using nodal values as well:

```

transpose(N(u))
*(u[_i] - lit(correction_lss->dt()))
*(nabla(p)[_i]*nodal_values(p))[0])

```

The implementation of Chorin's method shows how different systems can be combined to solve a problem with multiple unknowns, each interpolated using a different shape function. The coding of the assembly procedure remains concise and follows the structure of the mathematical equations.

4.3. PSPG/SUPG Stabilized Incompressible Navier-Stokes. As an alternative to the use of the Taylor-Hood element used

```

(1) // The pressure field, using the default first order shape function
(2) FieldVariable<1, ScalarField> p("Pressure", pressure_lss->solution_tag());
(3) // The linear system manager
(4) Handle<LSSActionUnsteady> pressure_lss = create_component<LSSActionUnsteady>("PressureLSS");
(5) // The assembly action
(6) Handle<ProtoAction> pressure_mat_assembly =
(7) pressure_lss->create_component<ProtoAction>("MatrixAssembly");
(8) pressure_mat_assembly->set_expression(elements_expression(LagrangeP1(),
(9) group
(10) (
(11) _A(p) = _0,
(12) element.quadrature(_A(p) += transpose(nabla(p))*nabla(p)),
(13) pressure_lss->system_matrix += _A
(14) ));
(15)
(16) Handle<ProtoAction> pressure_rhs_assembly =
(17) pressure_lss->create_component<ProtoAction>("RHSAssembly");
(18) pressure_rhs_assembly->set_expression(elements_expression(LagrangeP1P2(),
(19) group
(20) (
(21) _a[p] = _0,
(22) element.quadrature(_a[p] += transpose(N(p))*divergence(u)),
(23) pressure_lss->system_rhs += -lit(pressure_lss->invdt())*_a
(24) ));

```

LISTING 16: Pressure Poisson problem for Chorin's method.

in the previous example, we can use equal interpolation order for the pressure and velocity, provided that we add a stabilization term to the continuity equation. We follow the method presented in [14], which adds PSPG and SUPG stabilization, as well as a bulk-viscosity term. When using Crank-Nicolson time stepping, the method is second order accurate in both time and space. We also start from the skew symmetric momentum equation, yielding the following form to replace (9):

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} + \frac{\mathbf{u}(\nabla \cdot \mathbf{u})}{2} + \frac{\nabla p}{\rho} - \nu \nabla^2 \mathbf{u} = \mathbf{0}. \quad (13)$$

In the absence of stabilization, the discretization of the skew symmetric form preserves the kinetic energy [15]. The weak form of the equations for a single element, after discretization in time, can be written as

$$\left(\frac{1}{\Delta t} T + \theta A \right) \Delta \mathbf{x}_e = -A \mathbf{x}_e^n. \quad (14)$$

We applied a theta scheme here for the time discretization, solving for the difference between two time steps, $\Delta \mathbf{x}_e$. The vector of unknowns $\mathbf{x}_e^n$ is arranged in blocks for each unknown, that is, first the pressures for all nodes, then the velocities in the x direction, and so on. This results in a

corresponding blocked structure for the element matrices T and A , where the blocks are given by

$$A_{pu_i} = \int_{\Omega_e} \left(\left(\mathbf{N}_p + \frac{\tau_{\text{PSPG}} \tilde{\mathbf{u}}_{\text{adv}} \nabla \mathbf{N}_p}{2} \right)^T (\nabla \mathbf{N}_u)_i \right. \\ \left. + \tau_{\text{PSPG}} (\nabla \mathbf{N}_p)_i^T \tilde{\mathbf{u}}_{\text{adv}} \nabla \mathbf{N}_u \right) d\Omega_e,$$

$$A_{pp} = \int_{\Omega_e} \tau_{\text{PSPG}} \frac{1}{\rho} \nabla \mathbf{N}_p^T \nabla \mathbf{N}_p d\Omega_e,$$

$$A_{u_i u_j} = \int_{\Omega_e} \left(\tau_{\text{BULK}} (\nabla \mathbf{N}_u)_i + \frac{1}{2} (\tilde{\mathbf{u}}_{\text{adv}})_i (\mathbf{N}_u + \tau_{\text{SUPG}} \tilde{\mathbf{u}}_{\text{adv}} \nabla \mathbf{N}_u) \right)^T \\ \cdot (\nabla \mathbf{N}_u)_j d\Omega_e,$$

$$A_{u_i u_i} \\ = \int_{\Omega_e} \left(\nu \nabla \mathbf{N}_u^T \nabla \mathbf{N}_u + (\mathbf{N}_u + \tau_{\text{SUPG}} \tilde{\mathbf{u}}_{\text{adv}} \nabla \mathbf{N}_u)^T \tilde{\mathbf{u}}_{\text{adv}} \nabla \mathbf{N}_u \right) d\Omega_e \\ + A_{u_i u_i},$$

$$A_{u_i p} = \int_{\Omega_e} \frac{1}{\rho} (\mathbf{N}_u + \tau_{\text{SUPG}} \tilde{\mathbf{u}}_{\text{adv}} \nabla \mathbf{N}_u)^T (\nabla \mathbf{N}_p)_i d\Omega_e,$$

$$T_{pu_i} = \int_{\Omega_e} \tau_{\text{PSPG}} (\nabla \mathbf{N}_p)_i^T \mathbf{N}_u d\Omega_e,$$

$$T_{u_i u_i} = \int_{\Omega_e} (\mathbf{N}_u + \tau_{\text{SUPG}} \tilde{\mathbf{u}}_{\text{adv}} \nabla \mathbf{N}_u)^T \mathbf{N}_u d\Omega_e.$$

(15)

```

(1) Handle<LSSActionUnsteady> correction_lss = create_component<LSSActionUnsteady>("
    CorrectionLSS");
(2)
(3) Handle<ProtoAction> correction_matrix_assembly = correction_lss->create_component<
    ProtoAction>("MatrixAssembly");
(4) correction_matrix_assembly->set_expression(elements_expression(LagrangeP1P2(),
(5) group
(6) (
(7)   _A(u) = _0,
(8)   element_quadrature(_A(u[_i], u[_i]) += transpose(N(u))*N(u)),
(9)   correction_lss->system_matrix += _A
(10) ));
(11)
(12) Handle<ProtoAction> correction_rhs_assembly = correction_lss->create_component<ProtoAction>("
    RHSAssembly");
(13) correction_rhs_assembly->set_expression(elements_expression(LagrangeP1P2(),
(14) group
(15) (
(16)   _a[u] = _0,
(17)   element_quadrature(_a[u[_i]] +=
(18)     transpose(N(u))*(u[_i] - lit(correction_lss->dt()) * gradient(p)[_i])),
(19)   correction_lss->system_rhs += _a
(20) ));

```

LISTING 17: The code for the correction step in Chorin's method.

Each stabilization term is multiplied with a corresponding coefficient τ . Splitting the equation into blocks helps in managing the added complexity due to the stabilization and skew-symmetric terms. Block A_{pp} , for example, represents the pressure Laplacian that arises from the PSPG stabilization. The SUPG terms in blocks $A_{u_i u_i}$, $A_{u_i u_j}$, and $T_{u_i u_i}$ are all written as a modification to the weighting function, adding more weight to upstream nodes. The bulk viscosity and skew symmetric terms in the momentum equation fill the off-diagonal blocks, as indicated by the use of both indices i and j . Equation (14) is assembled into a single linear system using the elements expression defined in Listing 18 (showing only the part relevant to the assembly itself).

Since only one linear system is involved, the code is integrated into the framework in the same way as in Listing 1. The value of the coefficients τ depends on local element properties [16]. We calculate them using a user-defined terminal `compute_tau`, passing a reference to a `double` for each coefficient (line (7)). On line (13), we use indices `_i` and `_j` to create a nested loop over the dimension of the problem, filling all $A_{u_i u_j}$ blocks. The right hand side from (14) is built by applying the element matrix A to the current element unknowns $\mathbf{x}_e$, represented by `_x`. On line (22), we divide by θ to avoid the use of the θ scheme on the continuity equation, improving stability. Finally, on line (23) we write the system matrix as in the left hand side of (14).

5. Performance Analysis

In this section we discuss the results of some performance tests, using the application examples from the previous

TABLE 1: System characteristics for the performance tests.

	Poisson and Chorin	PSPG/SUPG (per node)
CPU(s)	Intel i7-2600	Two Intel Xeon E5520
RAM	16 GB	24 GB
Operating system	Fedora 18	CentOS 6.2
Compiler	GCC 4.7.2	GCC 4.8.0
Trilinos version	11.4.1	11.2.3

section. Table 1 lists our system characteristics. The PSPG/SUPG tests were run on a cluster with 28 nodes, connected using 1 Gb Ethernet. To avoid any difficulties in installing DOLFIN on the cluster, the tests comparing our results with DOLFIN were run on a separate desktop computer.

5.1. Poisson Problem. Our first performance test concerns the Poisson problem. The element equations (4) can easily be calculated analytically when using linear shape functions, allowing a comparison between manually coded versions, a code using a virtual function interface to the shape functions and code generated using our language. Additionally, we compare with a specialized user-defined terminal containing the manually coded version and with DOLFIN [4] from the FEniCS project. Problem (4) is completed with boundary conditions and a source term identical to the Poisson demo case from FEniCS:

$$\begin{aligned}
 f &= -6 \quad \text{over } \Omega, \\
 u &= 1 + x^2 + 2y^2 \quad \text{over } \Gamma.
 \end{aligned} \tag{16}$$

```

(1) assembly->set_expression(elements_expression
(2) (
(3)   AllElementsT(),
(4)   group
(5)   (
(6)     _A = _0, _T = _0,
(7)     compute_tau(u, nu_eff, u_ref, lit(tau_ps), lit(tau_su), lit(tau_bulk)),
(8)     element_quadrature
(9)     (
(10)      _A(p, u[_i]) += transpose(N(p) + tau_ps*u_adv*nabla(p)*0.5) * nabla(u)[_i]
(11)                      + tau_ps * transpose(nabla(p)[_i]) * u_adv*nabla(u),
(12)      _A(p, p) += tau_ps * transpose(nabla(p)) * nabla(p) / rho,
(13)      _A(u[_i], u[_j]) += transpose((tau_bulk + 1/3*nu_eff)*nabla(u)[_i]
(14)                                + 0.5*u_adv[_i]*(N(u) + tau_su*u_adv*nabla(u))) * nabla(u)[_j],
(15)      _A(u[_i], u[_i]) += nu_eff * transpose(nabla(u)) * nabla(u)
(16)                        + transpose(N(u) + tau_su*u_adv*nabla(u)) * u_adv*nabla(u),
(17)      _A(u[_i], p) += transpose(N(u) + tau_su*u_adv*nabla(u)) * nabla(p)[_i] / rho,
(18)      _T(p, u[_i]) += tau_ps * transpose(nabla(p)[_i]) * N(u),
(19)      _T(u[_i], u[_i]) += transpose(N(u) + tau_su*u_adv*nabla(u)) * N(u)
(20)    ),
(21)    system_rhs += -_A * _x,
(22)    _A(p) = _A(p) / theta,
(23)    system_matrix += invdt() * _T + theta * _A
(24)  ));

```

LISTING 18: The assembly of the PSPG/SUPG stabilized incompressible Navier-Stokes equations.

TABLE 2: Linear system assembly times (wall clock time and timing relative to Manual) for the Poisson problem on the unit square, using first order triangle shape functions on a 1000×1000 grid.

	Dummy matrix		Epetra matrix	
	Wall clock (s)	Relative	Wall clock (s)	Relative
Proto	0.32	5.93	0.61	1.79
Proto specialized	0.069	1.27	0.35	1.03
Manual	0.054	1	0.34	1
Virtual	2.82	52.2	3.18	9.35
DOLFIN	0.31	5.74	1.13	3.32

When using linear shape functions, the solution for the discrete system captures the analytical solution up to machine precision at the nodes. As an illustration, the code for the specialized user-defined terminal is presented in Listing 19. The terminal `assemble_triags` can then be used to directly assemble the linear system as shown on line (41). The manually coded version uses the same algorithm, but here we also loop over elements directly, avoiding the use of Proto entirely.

We first run a test on the unit square, divided into 1000 parts in both the x and y direction. Each square cell is divided into two triangles and first order shape functions from the Lagrange family are used. Table 2 summarizes the results, with labeling as follows: “Proto” is the code using our EDSL (see Listing 1); “Proto specialized” is the user-defined terminal from Listing 19; “Manual” is the manually coded assembly loop; “Virtual” is the code using the virtual function interface to the shape functions as it is available in Coolfluid

3; and finally “DOLFIN” is the code generated by the FEniCS project demo. All timings represent the average of 10 assembly runs. Given the large problem size, 10 runs are representative and the variation between subsequent averaged runs was less than 2%. Due to the simplicity of the Poisson problem, the insertion into the global sparse matrix structure can actually be more expensive than the evaluation of the element integrals. We run the benchmark using both the Trilinos backend (using an Epetra CRS matrix) and a “dummy” matrix—not storing any data—to properly time the assembly procedure. As seen from Table 2, the overhead of the matrix insertion is about 0.3 s in Coolfluid 3 and 0.8 s in DOLFIN, that is, at least of the order of the time it takes to compute the element matrix itself. When comparing the timings for the dummy matrix, we see that the generic Proto code—which uses second order Gauss quadrature—is more than 5 times slower than the manually coded version. The difference between the specialized and the manual versions is much smaller. Since the specialized code still uses the Proto element looping mechanism, we can conclude that its inherent overhead is small. We confirmed this by profiling the assembly with `gperftools` (<http://gperftools.googlecode.com/>), generating the call graphs shown in Figure 3. Each graph starts in the `execute` method of the relevant Action class. On the left, the generic Proto code is seen to be mostly inlined into the element loop, with only 15% of the time spent in calls to other functions (mostly the shape functions). The large absolute execution time (numbers next to the arrows) is due to the extra matrix operations involved in the second order quadrature. For the specialized function (middle

FIGURE 3: Call graphs of optimized code for the Poisson element matrix computation, from left to right: the generic Proto code, the Proto code with user-defined terminal, and the manually coded version. The percentages indicate the time spent in each function relative to the total execution time. The numbers next to the arrows indicate the absolute execution time, expressed in number of “ticks.”

TABLE 3: Linear system assembly times (wall clock time and timings relative to the default Proto implementation) for the Poisson problem on the unit cube, using first order tetrahedron or hexahedron shape functions on a $100 \times 100 \times 100$ grid.

	Dummy matrix		Epetra matrix	
	Wall clock (s)	Relative	Wall clock (s)	Relative
Proto, default	3.29	1	5.51	1
Proto, hexahedra	4.60	1.40	5.65	1.03
Proto, 1st order	0.81	0.25	2.64	0.48
DOLFIN	1.22	0.37	5.05	0.92

graph), some operations related to index book-keeping are performed in the loop function, while the rest is executed in the user-defined `PoissonTriagAssembly` operator. Finally, in the manual version on the right, all operations are in the specific implementation of the execute function, resulting in 10 fewer ticks for the overall execution time.

The virtual function implementation performs much worse than any other method. Like Proto, it allows writing generic code for all elements but does so by using virtual function calls and dynamically allocated matrices. This makes the method much slower than the Proto code. A separate test comparing dynamically and statically sized matrices (size 4×4) from the Eigen library shows a slowdown by a factor of about 8 for dynamic matrices when running matrix-matrix and matrix-vector multiplications, reinforcing the results from Table 2.

In DOLFIN, the right hand side and the matrix are computed using separate assembly loops. The presented timing is the total time spent on assembly, divided by the number of assemblies executed. For this particular case, Proto and DOLFIN result in the same performance. This is surprising, since the more advanced integration routines in DOLFIN detect here that first order quadrature is sufficient, while our Proto code assumes the worst case and applies second order integration.

The above observation leads us to perform the test in 3D, using a unit cube with 100 segments in each direction and built up of tetrahedra. We only compare the Proto and

DOLFIN versions, since this code can be applied in 3D without modification. The results are shown in Table 3. The effect of the quadrature order is obvious here, with our second order quadrature being almost three times slower than the first order quadrature performed by DOLFIN. To check if this is the only effect, we temporarily forced our code to use first order quadrature, that is, using one quadrature point instead of four. The speedup is as expected, but we do emphasize that it is only obtained after modification of the integration code: we do not have a method for determining the integration order based on the terms appearing in the equations. Instead, our integration system assumes there is a mass matrix term in the equation and proceeds to choose the integration order based on the shape function. If the performance penalty is significant, as is mostly the case with simple problems, it is possible to use a user-defined terminal to override the integration method, or even to avoid numerical integration altogether.

We also include some results for hexahedral elements, where the second order quadrature is necessary. The element matrix dimension is also doubled, resulting in longer computation times for the matrix operations. We see that this is compensated here when inserting into the sparse matrix, due to the lower number of elements (each hexahedron represents 6 tetrahedra).

5.2. Chorin’s Method. In Chorin’s method, there are a total of 6 different assembly loops to be run: one for each system matrix, and one for each right hand side. Even though the matrices only need to be assembled once for the simulation, we present timings here for comparison purposes. Table 4 summarizes the results. We see that, except for the auxiliary matrix assembly, DOLFIN is faster every time when the “dummy” matrix is used, with a very large discrepancy for the correction matrix assembly. This is due to an optimization in DOLFIN, which recognizes the coefficients of the element matrix can easily be precomputed, avoiding quadrature. Our code does not allow this optimization to happen automatically, although it is of course possible to add a user-defined terminal. In the auxiliary matrix, the same term appears, but here it is divided by Δt , causing DOLFIN to apply quadrature.

```

(1) // Specialized code for triangles
(2) struct PoissonTriagAssembly
(3) {
(4)     typedef void result_type;
(5)     // Functor that takes: source term f, Linear system lss, element matrix and vector acc
(6)     template <typename FT, typename LSST>
(7)     void operator()(const FT& f, LSST& lss, math::LSS::BlockAccumulator& acc) const
(8)     {
(9)         typedef mesh::LagrangeP1::Triag2D ElementT;
(10)        // Get the coordinates of the element nodes
(11)        const ElementT::NodesT& nodes = f.support().nodes();
(12)
(13)        // Compute normals
(14)        ElementT::NodesT normals;
(15)        normals(0, XX) = nodes(1, YY) - nodes(2, YY);
(16)        // ...repetitive code omitted
(17)        // Jacobian determinant
(18)        const Real det_jac = normals(2, YY)*normals(1, XX) - normals(1, YY)*normals(2, XX);
(19)        const Real c = 1. / (2.*det_jac);
(20)
(21)        // Indices of the nodes of the current element
(22)        acc.neighbour_indices(f.support().element_connectivity());
(23)
(24)        for(UInt i = 0; i != 3; ++i)
(25)            for(UInt j = 0; j != 3; ++j)
(26)                acc.mat(i, j) = c * (normals(i, XX)*normals(j, XX) + normals(i, YY)*normals(j, YY));
(27)
(28)        // Get the values of the source term
(29)        const Real f0 = f.value()[0];
(30)        // ...f1 and f2
(31)        acc.rhs[0] = (2*f0 + f1 + f2);
(32)        // ...acc.rhs[1] and acc.rhs[2]
(33)        acc.rhs *= det_jac/24.;
(34)        lss.matrix().add_values(acc);
(35)        lss.rhs().add_rhs_values(acc);
(36)    }
(37) };
(38) // Create an terminal that can be used as a function in a proto expression
(39) static MakeSF0p <PoissonTriagAssembly>::type const assemble_triags = {};
(40) // Usage example:
(41) assembly->set_expression(elements.expression
(42) (
(43)     boost::mpl::vector1< mesh::LagrangeP1::Triag2D>(),
(44)     assemble_triags(f, system_matrix, m_block_accumulator)
(45) ));

```

LISTING 19: Code for the specialized user-defined terminal for the Poisson problem, valid for linear shape functions over a triangle.

The Proto-generated code is currently suboptimal for the assemblies of the right hand sides. This is due to some missed chances for matrix reuse: the advection operation in (10), for example, is calculated once for every component. While this effect is significant when we eliminate the influence of the linear system, it is much less apparent when looking at the results for Epetra matrices and vectors in the last column of Table 4. This leads us to conclude that our performance level is adequate for practical use and the element matrix and vector calculations will not be a dominant factor in the total solution time.

5.3. Channel Flow Simulation. In the last performance test, we take a look at a practical example, using the PSPG/SUPG stabilized Navier-Stokes formulation from Listing 18. The test problem is the flow between two infinite flat plates, that is, a 3D channel flow with two periodic directions. We initialize the flow using a laminar solution with centerline Reynolds number of 11250 with respect to the channel half-height. We apply periodic boundary conditions in the stream- and span-wise directions and a no-slip condition at the walls. The average timings for the first 100 timesteps (initial Courant number: 0.57) are presented in Table 5. We ran the test using

TABLE 4: Assembly times for each step in Chorin’s method, compared between our Proto expressions and DOLFIN. Relative is the DOLFIN timing in multiples of the Proto timing. Wall times are in seconds.

	Dummy matrix			Epetra matrix		
	Proto	DOLFIN		Proto	DOLFIN	
	Wall	Wall	Relative	Wall	Wall	Relative
Aux. matrix	4.33	10.15	2.34	7.17	17.98	2.51
p matrix	0.28	0.19	0.67	0.53	0.75	1.42
Corr. matrix	2.38	0.22	0.09	5.23	8.09	1.55
Aux. RHS	3.12	1.17	0.375	3.18	2.32	0.73
p RHS	0.85	0.40	0.47	0.86	0.85	0.99
Corr. RHS	1.23	0.35	0.28	1.26	1.59	1.26

TABLE 5: Assembly and solution times for the coupled PSPG/SUPG stabilized Navier-Stokes equations (Listing 18) on a 3D channel flow with 128 hexahedra (tetrahedralized in the tetra cases) in each direction.

# CPU	Element	Assembly	Solution	solution/assembly
32	Hexa	8.97 s	90.90 s	10.14
	Tetra	7.69 s	73.06 s	9.51
	Tetra specialized	2.73 s	70.95 s	25.99
64	Hexa	4.89 s	48.33 s	9.88
	Tetra	4.14 s	40.59 s	9.81
	Tetra specialized	1.45 s	40.15 s	27.67
128	Hexa	3.05 s	32.91 s	10.47
	Tetra	2.58 s	54.53 s	21.13
	Tetra specialized	0.99 s	46.32 s	46.70

hexahedra and tetrahedra, where the test on tetrahedra also used a specialized code wrapped into a user-defined terminal (“Tetra specialized” in the table). The linear system was solved using the Belos Block GMRES method from Trilinos, preconditioned using ML algebraic multigrid. We tweaked the settings to obtain the fastest possible solution time. We see that the solution of the system takes about 10 times as long as its assembly using our EDSL. This shows that, even for the relatively complicated assembly expressions of (15), our language can be used to assemble the system efficiently. Any further optimization should first focus on the linear system solution before the assembly will become a bottleneck. In this context, it should be noted that the solution of the linear system does not scale as well as the assembly. This may be related to the relatively slow communication (1 Gb Ethernet) between the nodes.

The user-defined code for tetrahedra results in a further speedup factor of 2.5. In this case, the code was reused from a previous version of the solver, written only for tetrahedra. A domain specific language can also assist in developing hand-tuned code, however: using the language we can first easily specify the generic formulation and then check the element matrices of manually coded optimizations against the automatically generated matrices.

6. Conclusion and Future Work

We presented a domain specific language for the implementation of finite element solvers, embedded in C++. The language mirrors the mathematical notation faithfully, providing a

clean separation between numerics and equations. Our work is set apart from other work in this area by the use of the Boost.Proto library and the possibility to implement user defined terminals. Proto uses concise grammars to describe and extend the functionality of the language, as explained in detail using the stand-alone example. The addition of user defined terminals allows using hand-optimized code when possible, while staying within the automated framework for element looping.

We also analyzed the performance, demonstrating—in our opinion—acceptable abstraction overhead when compared to manual implementations and FEniCS. A large scale test with the PSPG/SUPG method for the incompressible Navier-Stokes equations showed that assembly took up to 10 % of the linear system solution time. This makes the assembly only the second largest consumer of CPU time by a large margin. It is our opinion that the sacrifice of some speed in the assembly is acceptable in view of the reduced turnaround time for model development.

Possible directions for future development include changes to the numerical integration framework, to better deduce the required quadrature order. On a more technical level, some parts of the code could be simplified by using new features of the C++11 standard, such as variadic templates and automatic return type deduction. Better error handling can also be looked into. One final interesting possibility is the investigation of expression optimization techniques. Using grammars, it is theoretically possible to scan the expressions for recurring matrix products and calculate each product only once, for example.

Appendix

Code Download Information

All of the code used in this work is available under open source licenses. Coolfluid 3 is licensed under the LGPL version 3 and available from <https://github.com/coolfluid/coolfluid3/>. Most benchmarks are in the `plugins/UFEM/test/demo` directory. The test comparing dynamically and statically sized Eigen matrices is at `test/math/p-test-eigen-vs-matrixt.cpp`. The code for the stand-alone example can be found in the repository <https://github.com/barche/eigen-proto/>, where `fem_example.cpp` contains the complete running program.

We also had to adapt DOLFIN, adding a “dummy” linear algebra backend to be able to measure without any overhead from the linear system backend. This code can be found at <https://bitbucket.org/barche/dolfin/>.

Finally, the code for benchmarks (including FEniCS tests) used in this paper is in the repository <https://github.com/barche/coolfluid3-benchmarks/>.

Nomenclature

$\mathbf{a}_{u_i}$:	Block of the element right hand side vector for the i th component of the vector variable $\mathbf{u}$
$A_{u_i u_j}$:	Block of the element matrix corresponding to rows for the i th component of vector variable $\mathbf{u}$ and the columns of the j th component
f :	Unknown in the Poisson problem
$\tilde{f}$:	Value of unknown f interpolated by shape functions
$\mathbf{f}_e$:	Vector of values at the element nodes for variable f
g :	Source term for the Poisson problem
$\mathbf{N}_f$:	Element shape function vector for variable f
n :	Number of nodes in an element
n_Ω :	Number of elements in the mesh
p :	Pressure
t :	Time
$\mathbf{u}$:	Velocity vector
Γ :	The problem domain boundary
ν :	Kinematic viscosity
Ω :	The problem domain.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The authors thank the Coolfluid 3 development team for the many hours of work and helpful discussions, without which this work would not have been possible. In particular they thank Tiago Quintino for laying out the basic framework;

Willem Deconinck for the work on the mesh structure; and Quentin Gasper for the work on the GUI.

References

- [1] T. Quintino, W. Deconinck, B. Janssens et al., “Cooluid 3,” 2012, <http://coolfluid.github.io/>.
- [2] E. Niebler, “Proto: a compiler construction toolkit for DSELs,” in *Proceedings of the Symposium on Library-Centric Software Design (LCSD '07)*, pp. 42–51, October 2007.
- [3] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby, “A tensorial approach to computational continuum mechanics using object-oriented techniques,” *Computers in Physics*, vol. 12, no. 6, pp. 620–631, 1998.
- [4] A. Logg and G. N. Wells, “DOLFIN: automated finite element computing,” *ACM Transactions on Mathematical Software*, vol. 37, no. 2, article 20, 2010.
- [5] M. S. Alnæs and K.-A. Mardal, “On the efficiency of symbolic computations combined with code generation for finite element methods,” *ACM Transactions on Mathematical Software*, vol. 37, no. 1, p. 1, 2010.
- [6] C. Prud'homme, “A domain specific embedded language in C++ for automatic differentiation, projection, integration and variational formulations,” *Scientific Programming*, vol. 14, no. 2, pp. 81–110, 2006.
- [7] O. Zienkiewicz and R. Taylor, *The Finite Element Method*, Butterworth-Heinemann, Oxford, UK, 2000.
- [8] G. Guennebaud, B. Jacob, M. Lenz et al., Eigen v3, 2010, <http://eigen.tuxfamily.org>.
- [9] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, Boston, Mass, USA, 2000.
- [10] M. A. Heroux and J. M. Willenbring, “Trilinos users guide,” Tech. Rep. SAND2003-2952, 2003.
- [11] R. Johnson, R. Helm, J. Vlissides, and E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Mass, USA, 1995.
- [12] K. Iglberger, G. Hager, J. Treibig, and U. Rüde, “Expression templates revisited: a performance analysis of current methodologies,” *SIAM Journal on Scientific Computing*, vol. 34, no. 2, pp. C42–C69, 2012.
- [13] A. J. Chorin, “Numerical solution of the Navier-Stokes equations,” *Mathematics of Computation*, vol. 22, no. 104, pp. 745–762, 1968.
- [14] T. Bányai, D. Vanden Abeele, and H. Deconinck, “A fast fullycoupledsolution algorithm for the unsteady incompressible Navier-Stokes equations,” in *Proceedings of the Conference on Modelling Fluid Flow (CMFF '06)*, Budapest, Hungary, 2006.
- [15] T. A. Zang, “On the rotation and skew-symmetric forms for incompressible flow simulations,” *Applied Numerical Mathematics*, vol. 7, no. 1, pp. 27–40, 1991.
- [16] T. Tezduyar and S. Sathe, “Stabilization parameters in SUPG and PSPG formulations,” *Journal of Computational and Applied Mechanics*, vol. 4, no. 1, pp. 71–88, 2003.