FEMEngine: finite element method C++ code based on functional and template metaprogramming

Alexey Gurin*, Alexey Baykin*, Timofey Polyansky†, Anton Krivtsov§

*Laboratory of Digital and Intelligent Hydrocarbon Production Systems
Lavrentyev’s Institute of Hydrodynamics of the SB RAS, Novosibirsk, Russia
Email: aleksej.gurin00@gmail.com
†Department of Physics
Novosibirsk State University, Novosibirsk, Russia
§Higher School of Theoretical Mechanics
Peter the Great St. Petersburg Polytechnic University (SPbPU), St. Petersburg, Russia

Abstract—The paper discusses the problems of a finite element method programming. Modern C++ functional programming and template metaprogramming approach to finite element analysis is presented. This approach simplifies the implementation of an effective assembly of the stiffness matrix for a problem defined by a weak form. This method is tested by a solution of the Poisson equation on an unstructured 3D tetrahedral mesh using FEM C++ library FEMEngine developed by authors. The function which calculates the matrix is generated by higher order functions during the compilation stage. The performance of the computation is analyzed by studying of a disassembled code and by comparison with the popular open source FEM software.

I. INTRODUCTION

The finite element method or finite element analysis is well suited to be implemented in a code. The algorithms of matrix assembly for any practical purpose are well known. However, the programming of a generic FEM implementation which can solve a wide variety of cases defined by weak formulations is a challenging task. It is difficult to achieve a good calculation performance, simplicity of the problem setup process and maintainable, well-structured code.

There are a large number of open source software for finite element analysis. A few examples of such software are FreeFem++, MFEM and FEniCS Project. FreeFem++ [1] solves partial differential equations in a weak form provided by user in text .edp file. This file must be written in a similar to C++ intermediate language. Although FreeFem++ is a simple yet versatile tool for FEA, its performance during matrix assembly is low for complex equations. MFEM [2] is a C++ library designed in an object-oriented approach. A solver programmed with MFEM can be compiled as an executable or as a library and linked to other software. The performance of MFEM is high, but the weak form implementation in the code is not straightforward, in comparison with FreeFem++. An engineer needs to learn a complex interface of the library to solve physical problems. The implementation of finite elements in the FEniCS Project [3] is devoid of these problems. A user sets a physical problem with a simple and straightforward Python interface, efficiently solves the problem and get a way to embed the solver in another software. This features are achieved by the metaprogramming technique. The FEniCS Form Compiler, written in Python as a part of the FEniCS Project, generates and compile an efficient C++ code for a matrix assembly on the fly. This form of metaprogramming is universal, but very difficult to implement and maintain.

During the last decade, the C++ language has been greatly improved. Three new standards are introduced: C++11, C++14, C++17. The ability to program in a functional paradigm has been added and the template metaprogramming technique has been simplified. In the Lavrentyev’s Institute of Hydrodynamics of the SB RAS, a library named “FEMEngine” has been developed by using the new C++ programming methods. The library combines the ability to set the equation through a weak form and the high performance of matrix assembly.

II. MATHEMATICAL FORMULATION

A convenient way to set a problem in FEM software is to provide a weak formulation, a boundary conditions, a domain geometry and a discretization parameters. This information should be enough for the software to provide the solution. For the sake of simplicity let us consider the Dirichlet problem (1), (2) for the Laplace equation

\[ \Delta T = 0 \quad \text{in} \ \Omega, \]

\[ T|_{\partial \Omega} = T_{\partial \Omega}, \]

and in a weak form

\[ \int_{\Omega} \nabla T \cdot \nabla \psi \, d\Omega = 0. \]

The physical domain \( \Omega \), over which the solution is sought, is discretized into the elements \( \Omega_k, k = 1, \ldots, N \) with N nodes on each element. The element is defined by the set of functions \( \phi_i(X) \) which form the interpolation polynomial over the element. The quantity T could be represented locally as

\[ T(X) = \sum_{i=1}^{N} T_i \phi_i. \]
const expr auto phi1 = [](
    std::tuple<double, double> r
    ){
        auto [xi, eta] = r;
        return -eta - xi + 1.0;
    }

... std::tuple phi{phi1, phi2, phi3};

Fig. 1. Lambdas as containers of interpolation functions

template<class F>
struct function_traits :
    std::false_type {}
;

template<class retType, class C, class ... Args>
struct function_traits<retType (C::*)(Args...) const> :
    std::true_type {
        typedef retType ret;
        using args = std::tuple<Args...>;
    };

Fig. 2. Function traits class

where $T_i$ are values in the element nodes. After replacing of
the unknown function $T$ with the polynomial and the
test function $\psi$ with the basis interpolation functions on
the element we obtain the expression for the element stiffness
matrix

$$M_{ij}^E = \int_{\Omega_E} \nabla \phi_i \cdot \nabla \phi_j \mathrm{d} \Omega, \quad i = 1, \ldots, N; \quad j = 1, \ldots, N. \quad (4)$$

While it is possible to integrate exactly in a simple case, in
general the numerical integration with a quadrature is used.
To simplify the calculation of the matrix, the element
functions are defined over the special canonical element and
the coordinate transformation to the physical element is carried
out.

III. MULTIPLICATION OF FUNCTIONS

In C++11 lambda functions were introduced. In combina-
tion with std::tuple container, lambdas can be used to contain
the element interpolation functions as shown in figure 1.
There std::tuple is a container that can accumulate objects
of different types and the compiler can inline its content. A
lambda function in C++ is implemented as the anonymous
functor with operator().

The number of lambda arguments and their types can
be obtained with the type traits struct shown in figure 2.
The "args" template alias inside of the function_traits struct
contains the type std::tuple<Args...> where "Args..." is
the template parameter pack containing types of the function
arguments. With the function traits, a higher order function
can be obtained for multiplication of functions as shown
in figure 3.

template<class F1, class F2, class ... Args>
auto multiply( F1 f1, F2 f2, std::tuple<Args...> )
{
    return [=]( Args ... args ){
        return f1(args...) * f2(args...);
    }
}

template<class F1, class F2>
auto multiply( F1 f1, F2 f2 )
{
    return multiply( f1, f2,
        typename function_traits<
            decltype(&F1::operator())>::args{} );
}

Fig. 3. Multiplication of functions

This function expects two functions $f_1$, $f_2$ with the same
arguments. The function traits class finds out the types and
the number of arguments of the first function. Then the high
order function returns a lambda with the arguments same as
in $f_1$, $f_2$ functions. This lambda calls functions $f_1$, $f_2$, then
passes arguments and returns the multiplication of their results.
Assuming that the vector is represented by a tuple and the
tensor is represented by a tuple of tuples, the function shown
in figure 4 straightforwardly implements the generic outer
product of vectors

$$F_{ij} = \phi_i \phi_j, \quad i = 1, \ldots, N; \quad j = 1, \ldots, N. \quad (5)$$

In this implementation the tupleBinaryOp function performs
a generic binary operation on tuples as schematically presented
in figure 5.

Now if the interpolation functions of an element are rep-
resented by lambdas, the tensorProd function can be used to
obtain the part of element stiffness matrix expression (5) under
integral.

IV. INTEGRATION OF A FUNCTION OVER ELEMENT

In FEM implementations usually the integrals of type (4)
are calculated using the coordinate transformation and moving
to the integral over a canonical element with fixed geometry.
Thus, the integral over an arbitrary element on a physical
domain can be calculated by applying the quadrature formula

$$\int_{\Omega_k} f \, \mathrm{d} \Omega = \sum_{i=1}^{N} f(\vec{r}_i) \cdot w_i |J_k|,$$

where $\vec{r}_i$, $w_i$ is the nodes and the weights, respectively,
of the quadrature on the canonical element and $|J_k|$ is the
determinant of Jacobi matrix of coordinate mapping from
canonical element to the real element $\Omega_k$ in the physical
domain. This quadrature can be applied to a function through
the partial application technique as shown in figure 6.
template <class ... Args1,
        class ... Args2,
        size_t ... Is>
auto tensorProd (std::tuple<
    Args1 ...>
t1,
    std::tuple<
    Args2 ...>
t2,
    std::index_sequence<
    Is ...>
)
{
    return std::make_tuple(
        tupleBinaryOp(
            t1,
            std::get<
                Is
            >(t2),
            std::multiplies{}
        )....
    ) ;
}

template <class ... Args1,
        class ... Args2>
auto tensorProd (std::tuple<
    Args1 ...>
t1,
    std::tuple<
    Args2 ...>
t2)
{
    return tensorProd (t1, t2, std::make_index_sequence<
        sizeof (Args2)>{});
}

Fig. 4. Outer product of vectors

v1 ... v6 - scalars
t1 = [v1, v2, v3]
t2 = [v4, v5, v6]
tupleBinaryOp (t1, t2) -> [v1*v4, v2*v5, v3*v6]
tupleBinaryOp (t1, v4) -> [v1*v4, v2*v4, v3*v4]

Fig. 5. Scheme of tupleBinaryOp function

\[
f(r) = [r1, r2, r3] \\
w = [w1, w2, w3] \\
\]

\[
\begin{align*}
\text{integrate}(f, rc, w) &\rightarrow \{ \\
\text{f\_int}(|J|) &\rightarrow ( f(r1) \ast w1 + \\
                        f(r2) \ast w2 + \\
                        f(r3) \ast w3 ) \ast |J| \}
\end{align*}
\]

Fig. 6. Scheme of integrate function

\[
\phi = [\phi_1(r), \phi_2(r), \phi_3(r)] \\
f(T) \rightarrow T^2 \\
\text{interpolate}(f, \phi) \rightarrow \{ \\
\text{f\_interp}(r, [T1, T2, T3]) \rightarrow f(T1*\phi_1(r) + \\
                              T2*\phi_2(r) + \\
                              T3*\phi_3(r)) \}
\]

Fig. 7. Scheme of interpolate function

The high order function integrate requires the \( f \) function, the quadrature nodes \( rc \) and the weights \( w \). It returns the function which expects Jacobian and returns the integral of \( f \) function calculated with the provided quadrature.

V. INTERPOLATION OVER AN ELEMENT

In practical applications there is a need to solve equations with nonlinear coefficients. For example, the expression

\[
\int_0^1 T^2 \nabla T \cdot \nabla \psi d\Omega
\]

contains the nonlinear coefficient \( T^2 \), where \( T \) is the finite element function taken from the previous nonlinear iteration. The nonlinear coefficient under the integral sign must be interpolated to the quadrature nodes inside the integration procedure. The interpolation can be easily performed with the expression (3). In order to calculate the integral, \( T^2 \) is interpolated to the quadrature nodes and then squared to obtain the nonlinear coefficient value. The high order function interpolate presented in figure 7 creates the interpolation function for an arbitrary nonlinear coefficient.

The interpolate function receives the \( f \) function which represents the nonlinear coefficient and returns the function which expects coordinates on the canonical element and values of \( T \) in the element nodes.

VI. LOCAL STIFFNESS MATRIX ASSEMBLY

The above mentioned methods could be used to generate the function for the matrix assembly. Assume that the matrix need to be calculated from the expression

\[
\int_0^1 T\phi_i\phi_j d\Omega, \quad i = 1, \ldots, N, \quad j = 1, \ldots, N
\]

on the 1st order triangle element. So

\[
\phi = [\phi_1, \phi_2, \phi_3]
\]

is the interpolation functions on the element. The full algorithm of matrix assembly is described in figure 8.

By usage of the tensorProd function, 3x3 local matrix of the functions representing \( \phi_i\phi_j \) is obtained. Then the interpolation function for the coefficient is created by the call to the interpolate function and each function in the matrix are multiplied by it. After the multiplication, the integrate function is applied to each element of the matrix. The matrix of functions elementMatrixF exactly calculates the element matrix if the Jacobian and the \( T \) node values are provided.
The efficiency of the method

The most important feature of the described methodology is the generalization of FEM. The high order functions described in the previous sections accept functions with any number of arguments. The same implementation of the functions tensorProd, interpolate, integrate is used for 1D, 2D, 3D FEM elements of any order.

The proposed methods are less error prone than a conventional manual programming of matrix assembly for every task. This feature is achieved due to the compile time error checks provided by the C++ compiler and custom checks based on a static assert function.

The entities like the tuple and the lambda function used in section 3 are inlined by the C++ compiler. When the function matrix elementMatrixF is called in a code, the compiler does not generate a huge tree of calls to lambdas and tuples, it inlines and optimizes the code to calculate the matrix elements. The efficiency of the optimization was tested on a simple case where the integral

\[
\int_{\Gamma_k} T \phi_i \phi_j d\Omega; \quad i = 1, 2, j = 1, 2
\]

calculated with the Gauss quadrature of the first order. The values of the T coefficient are inputted by a user through the std::cin input stream. It is necessary, because if the values of T are hardcoded in the test program, the compiler calculates the matrix values during the compilation stage and eliminates all code except the code related to output in a terminal. This test code was compiled by the gcc 7.4 compiler with the -O2 fast compilation flag and the resulted binary file was disassembled by the radare2 end examined. In figure 9 a part of the main function disassembly is shown. The resulting binary file was disassembled by the radare2 end examined. In figure 9 a part of the main function disassembly is shown. This code exists between the user input and the terminal output inside the C++ nested calls to the tensorProd, the interpolate, the integrate functions and after the compilation it consists only of the arithmetic and copy operations. The instructions 2, 3 receive the user input for T1, T2 values on the element. On the instructions 4, 6, 14, 15 the program uses four double-precision floating point values which is determined during the optimization phase. These values represent the part of the integrated functions that the compiler can calculate. On the instructions 8–16 the actual matrix calculation takes place. There is no function calls except to the input/output streams. Thus, the compiler eliminates all nested calls and optimizes the arithmetics as best as it can.

VIII. TEST FOR THE SOLUTION OF POISSON EQUATION

To demonstrate the applicability of this method to solve the problems on the real meshes, the solution of the Poisson equation (8) is performed via the in-house code FemEngine and compared to the widely known FEniCS and FreeFEM++ open source codes. The expression at the right hand side of eq. (6) is chosen in such way that formula (9) describes the exact solution. The boundary conditions are defined by the expression (10).

\[
f(x, y, z) = 12(x^2 + y^2 + z^2),
\]

\[
\Delta T = f(x, y, z),
\]

\[
\int_{\Omega} \nabla T \cdot \nabla \psi d\Omega = - \int_{\Omega} f(x, y, z) \psi d\Omega,
\]

\[
T(x, y) = x^4 + y^4 + z^4,
\]

\[
T(x, y)|_{\partial \Omega} = x^4 + y^4 + z^4.
\]

This mathematical problem is translated into the C++ FemEngine code presented in figure 10.
auto Tf = FCL::f(T);
auto gradT = grad(T);
auto gradTMul = scalarMul(gradT, gradT);
auto integratedGradTMul = integrate(gradTMul, Quadrature3D::GaussOrder3{});

auto interpFunc = interpolate(Tf, double x, double y, double z) {
    return -12.0 * x * x - 12.0 * y * y - 12.0 * z * z;
};

auto rhsFunc = Tf * interpFunc;
auto rhsFuncIntegr = integrate(rhsFunc, Quadrature3D::GaussOrder3{});
auto rhs = LinearForm(rhsFuncIntegr, T, x, y, z);

EquationFEM eq(P1Space, mesh, std::move(solver));
Eq.addToGlobalMatrix(integratedGradTMul);
eq.addToGlobalRHSVector(rhs);
eq.solve();

Fig. 10. FEMEngine code for the Poisson equation.

Fig. 11. Poisson equation solution in YZ plane, x = 0.5.

For testing purpose the 1x1x1 sized cubic computational domain is discretized by the 40x40x40 computational mesh. The calculations are performed on Intel i7-6700K CPU. The FEMEngine calculates the global stiffness matrix in 1.07 s, whereas the FreeFEM++ spends 8.32 s. The FEniCS uses a just-in-time compilation method, so first start of the program takes 3.41 s to compile functions and assemble the matrix. All subsequent launches take only 0.21 s for the matrix assembly. The numerical results obtained from the FEMEngine and the FreeFEM++ on the same mesh are matching perfectly if the quadrature and the linear system solver are identical. The FEniCS solution is different by 0.03% as expected because the different linear system solver is used. The aforementioned numerical solution is shown in figure 11.

FreeFEM++ parses the ".edp" file, which contains a problem formulation, and assembles the matrix in runtime. Codes that can solve a wide class of problems defined by the input received during the runtime generally rely on a time-expensive dynamic polymorphism and a complex branching logic. FEniCS uses the just-in-time compilation method together with a C code generation by Python. An optimized code for a problem solution are compiled into a dynamic library during the program runtime and linked immediately. This method is immensely difficult to implement and maintain, but its performance and versatility are one of the highest. FEniCS assembles the matrix 40 times faster than FreeFEM++ for the problem. FEMEngine is based on the C++ templates, so the program is compiled for a single task being solved. FEMEngine demonstrates about 8 times faster than FreeFEM++ and 5 times slower than FEniCS matrix assembly operations, that is a room for improvement. The templated code for generation of the local stiffness matrix relies on the optimization provided by the C++ compiler which is already quite efficient and difficult to improve. On the other hand, the speed-up of the code which takes the local matrices and assembles them into a global matrix will be the goal of the authors further work.

IX. CONCLUSIONS

The C++ template metaprogramming library for finite element analysis FEMEngine is developed.

The template metaprogramming along with the functional approach has a great potential for the finite element code development. These programming techniques make it possible to write a reliable, generic and efficient code.

The matrix construction time between the FEMEngine and the FreeFEM++ is compared and 8 times advantage is achieved. The comparison with FEniCS FEM code shows that there is a potential to optimize the bottlenecks of the current matrix assembly algorithm.

ACKNOWLEDGMENT

This work was supported by Ministry of Science and Higher Education of the Russian Federation within the framework of the Federal Program "Research and development in priority areas for the development of the scientific and technological complex of Russia for 2014 – 2020" (activity 1.2), grant No. 14.575.21.0146 of September 26, 2017, unique identifier: RFMEFI57517X0146. The industrial partner of the grant is LLC "Gazprom-neft Science & Technology Centre".

REFERENCES