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

Gurin A.M.\textsuperscript{1}, Baykin A.N.\textsuperscript{1}, Polyansky T.A.\textsuperscript{2}, Krivtsov A.M.\textsuperscript{3,4}

\textsuperscript{1}Lavrentyev Institute of Hydrodynamics of SB RAS
\textsuperscript{2}Novosibirsk State University
\textsuperscript{3}Peter the Great St. Petersburg Polytechnic University (SPbPU)
\textsuperscript{4}Institute for Problems in Mechanical Engineering
The finite element method is widely used to solve systems of partial differential equations that are represented in the weak formulation.
Stiffness matrix assembly

System of linear equations

$$[K]\{T\} = 0$$

Assemble of global stiffness matrix

$$M^k_{ij} = \int_{\Omega_k} \nabla \varphi_i \cdot \nabla \varphi_j \, d\Omega$$

$$i = 1, ..., e_p, \quad j = 1, ..., e_p \quad k = 1, ..., N$$

$$[K] = [A]^T \text{diag}([M^1], ..., [M^N])[A]$$

- $N$ – number of elements
- $e_p$ – number of nodes on element
- $[A]$ – transformation matrix from local to global

Shape functions on triangle element

$\varphi^k_1$, $\varphi^k_2$, $\varphi^k_3$
Shape functions on a 2D triangular element

**Shape functions on a canonical element**

\[
\varphi_0(\xi, \eta) = -\eta - \xi + 1 \\
\varphi_1(\xi, \eta) = \xi \\
\varphi_2(\xi, \eta) = \eta
\]

**Integral calculation over a canonical element**

\[
\int_{\Omega_k} \varphi_i(x,y) \varphi_j(x,y) \, dx \, dy \\
\int_{e_{can}} \varphi_i(\xi, \eta) \varphi_j(\xi, \eta) |J_k| \, d\xi \, d\eta \\
\int \varphi_i \varphi_j \rightarrow \begin{pmatrix}
\int \varphi_0 \varphi_0 & \int \varphi_0 \varphi_1 & \int \varphi_0 \varphi_2 \\
\int \varphi_1 \varphi_0 & \int \varphi_1 \varphi_1 & \int \varphi_1 \varphi_2 \\
\int \varphi_2 \varphi_0 & \int \varphi_2 \varphi_1 & \int \varphi_2 \varphi_2 
\end{pmatrix}
\]
Numerical quadrature

\[ \int_{e_{can}} \varphi(x, \eta) \, dx \, d\eta = \sum_{i=1}^{n_g} \varphi(\xi_i, \eta_i) \omega_i \]
C++ standards

- C++11
  - lambda functions
  - move semantics (rvalue references)
  - constexpr
  - initializer lists
  - type inference (auto keyword)
  - uniform initialization
  - variadic templates
  - tuples
  - type traits
  - static_assert

- C++14
  - function return type deduction
  - generic lambdas
  - tuple addressing via type

- C++17
  - Structured bindings
  - constexpr if
  - fold expressions
C++ standards

• C++11
  • lambda functions
  • move semantics (rvalue references)
  • constexpr
  • initializer lists
  • type inference (auto keyword)
  • uniform initialization
  • variadic templates
  • tuples
  • type traits
  • static_assert

• C++14
  • function return type deduction
  • generic lambdas
  • tuple addressing via type

• C++17
  • Structured bindings
  • constexpr if
  • fold expressions
Lambda functions

- Shape functions are implemented in the code as lambda functions.
- Functions receive tuples consisting of the local coordinates $\xi, \eta$ as input.
- The lambda function is stored in a static variable inside of the element class.

```cpp
constexpr auto phi1 = [](std::tuple<double, double> r)
{
    auto [xi, eta] = r;
    return -eta - xi + 1.0;
};
...
std::tuple phi{phi1, phi2, phi3};
```
Function traits

The function_traits class can determine lambda function arguments and return type at compile time.

Argument types are contained in type of type alias “Args”

Template argument <Args...> can contain any number of types.
Multiplication of functions

The higher order function “multiply” expects for input 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.

The higher order function returns a lambda with the same arguments as in functions \( f_1 \), \( f_2 \).

```cpp
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{} );
```
Simplified pseudocode of the higher order function “multiply”

```cpp
f1(args...)  
f2(args...)  
multiply( f1, f2 ) -> {  
    fMulitp(args...) -> f1(args...) * f2(args...)  
}
```

- C++ variadic template metaprogramming code is too complex and contains too much boilerplate code
- Here, the functional concept of “multiply” function is represented in simple pseudocode
- Function implements the mathematical operation \( f_1(X) \cdot f_2(X) \)
Pseudocode of the higher order function “cartesian product”

phi = [ p1(xi, eta),
       p2(xi, eta),
       p3(xi, eta) ]

tensorProd( phi, phi ) ->
   [[p1 * p1, p1 * p2, p1 * p3],
    [p2 * p1, p2 * p2, p2 * p3],
    [p3 * p1, p3 * p2, p3 * p3]]

- The “tensorProd” function takes two tuples of functions and returns a matrix of functions represented by a tuple of tuples.
- Implements the mathematical operation $\phi \otimes \phi$.
Pseudocode of the higher order function “integrate”

The “integrate” function takes as input a function to be integrated and a numerical quadrature (nodes and weights)

Returns a function which calculates the integral if a Jacobian is provided

Implements the mathematical operation \( \int_{\Omega_k} f(\vec{r}) d\Omega \)

```plaintext
f(r)
rc = [r1, r2, r3]
w = [w1, w2, w3]

integrate(f, rc, w) -> {
    f_int(|J|) -> ( f(r1) * w1 +
                          f(r2) * w2 +
                          f(r3) * w3 ) * |J| }
```
Treatment of a nonlinear coefficient

\[ \int_\Omega T^2 \nabla T \cdot \nabla \psi \, d\Omega = 0 \]

\[ \sum_{i=1}^{N_p} T_i \int_\Omega \left( \sum_{k=1}^{N_p} T_k^{old} \varphi_k \right)^2 \nabla \varphi_i \cdot \nabla \varphi_j \, d\Omega = 0 \]

- The “interpolate” function takes as input a nonlinear coefficient to be interpolated and a shape functions.
- Returns a function which calculates nonlinear coefficient at the “r” coordinate of the canonical element if the values of unknowns in the coefficient are provided.
- Implements the mathematical operation:

\[ \left( \sum_{k=1}^{N_p} T_k^{old} \varphi_k \right)^2 \]
Local matrix generation algorithm

\[ \int_{\Omega} T^2 \varphi_i \cdot \varphi_j \, d\Omega \]

- The higher order functions generate the matrix of functions “elementMatrixF”
- This matrix of functions generate local stiffness matrix if called with Jacobian and pack of T values \([T1, T2, T3]\)

```python
rc = [r1, r2, r3]
w = [w1, w2, w3]

phi = [phi1(r), phi2(r), phi3(r)]

phiOuterPhi = tensorProd(phi, phi);

coeffF = f(T) -> T^2

coeffInterp = interpolate(coeffF, phi);

elementMatrixF = integrate(phiOuterPhi*coeffInterp, rc, w)
```
Disassembly of code

\[ \int_{\Omega} T \varphi_i \cdot \varphi_j \, d\Omega ; \quad i = 1, 2, j = 1, 2 \]

- Disassembly of the code which calculates local stiffness matrix and output it to the standard output stream is presented on the slide.
- There are no function calls and class instances such as a “tuple” in this code.
- C++ compiler efficiently optimized the code generated by the methods described on previous slides.
Test on solution of Poisson equation

\[
\int_\Omega \nabla T \cdot \nabla \varphi \, d\Omega = -12x - 12y - 12z
\]

```cpp
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();```
Comparison with FEniCS and FreeFEM++

- Tetrahedral mesh 41x41x41 nodes, 384000 elements
- Same mesh for all solvers

<table>
<thead>
<tr>
<th>Solver</th>
<th>Calculation of $[K]$, s</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEniCS</td>
<td>0.21</td>
</tr>
<tr>
<td>FEMEngine</td>
<td>1.07</td>
</tr>
<tr>
<td>FreeFem++</td>
<td>8.32</td>
</tr>
</tbody>
</table>
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.
FEMEngine source code will be released soon under an open source license. If you are interested to try it, then write to this email: "aleksej.gurin00@gmail.com" and we will send you a link to the repository when it’s ready.