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

Gurin A.M.[1], Baykin A.N.[1], Polyansky T.A.[2], Krivtsov A.M.[3,4]

[1]Lavrentyev Institute of Hydrodynamics of SB RAS
[2]Novosibirsk State University
[3]Peter the Great St. Petersburg Polytechnic University (SPbPU)
[4]Institute for Problems in Mechanical Engineering

Finite Element Method

Laplace equation

$$\Delta T = 0 \quad \in \Omega\,; \quad T|_{\partial\Omega} = T_{\partial\Omega}$$

Weak form

$$\int_{\Omega} \nabla T \cdot \nabla \psi \, d\Omega = 0$$

Interpolation

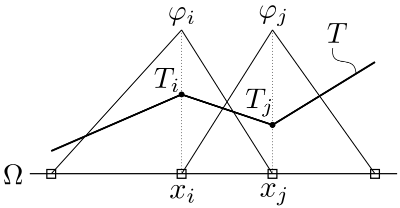
$$T(\mathbf{X}) = \sum_{i=1}^{N_p} T_i \varphi_i$$

System of linear equations

$$\sum_{i=1}^{N_p} T_i \int_{\Omega} \nabla \varphi_i \cdot \nabla \varphi_j \, d\Omega = 0$$

$j = 1, ..., N_p\,;\ N_p$ – number of nodes

The finite element method is widely used to solve systems of partial differential equations that are represented in the weak formulation



Shape functions

Stiffness matrix assembly

System of linear equations

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

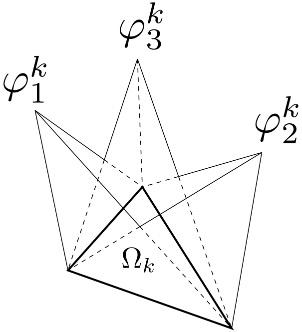
Assemble of global stiffness matrix

$$M_{ij}^k = \int\limits_{\Omega_k} \nabla\varphi_i \cdot \nabla\varphi_j \, d\Omega$$

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

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

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



Shape functions on triangle element

Shape functions on a 2D triangular element

Shape functions on a canonical element

$$\varphi_0(\xi,\eta) = -\eta - \xi + 1$$

$$\varphi_1(\xi,\eta) = \xi$$

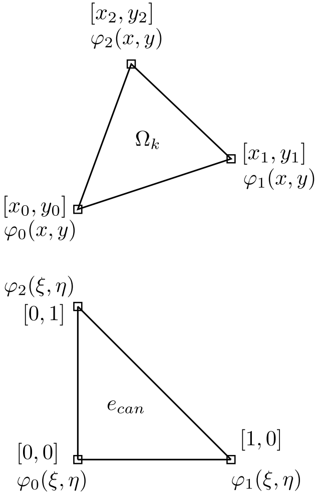
$$\varphi_2(\xi,\eta) = \eta$$

Integral calculation over a canonical element

$$\int_{\Omega_k} \varphi_i(x,y)\varphi_j(x,y)dxdy$$

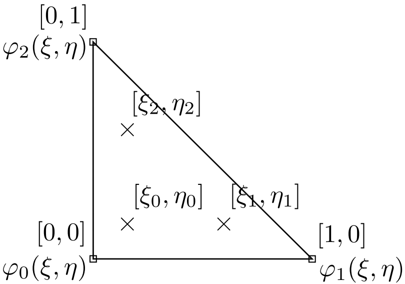
$$\int_{e_{can}} \varphi_i(\xi,\eta)\varphi_j(\xi,\eta)|J_k|d\xi d\eta$$

$$\int \varphi_i\varphi_j \to \begin{pmatrix} \int \varphi_0\varphi_0 & \int \varphi_0\varphi_1 & \int \varphi_0\varphi_2 \\ \int \varphi_1\varphi_0 & \int \varphi_1\varphi_1 & \int \varphi_1\varphi_2 \\ \int \varphi_2\varphi_0 & \int \varphi_2\varphi_1 & \int \varphi_2\varphi_2 \end{pmatrix}$$



Triangle element

Numerical quadrature



i	ξ_i	η_i	ω_i
0	1/6	1/6	1/6
1	2/3	1/6	1/6
2	1/6	2/3	1/6

$$\int_{e_{can}} \varphi(\xi,\eta)d\xi d\eta = \sum_{i=1}^{n_g} \varphi(\xi_i,\eta_i)\omega_i$$

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

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

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

Function traits

```
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...>;
};
```

- 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" `std::tuple<Args...>`
- Template argument `<Args...>` can contain any number of types

Multiplication of functions

```
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{} );
}
```

- The higher order function "multiply" expects for input two functions f1, f2 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 f1, f2

Simplified pseudocode of the higher order function “multiply”

```
f1(args...)
f2(args...)

multiply( f1, f2 ) -> {
   fMultip(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 $f1(X) \cdot f2(X)$

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 $\varphi \otimes \varphi$

Pseudocode of the higher order function "integrate"

```
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| }
```

- 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 $\int_{\Omega_k} f(\vec{r})d\Omega$

Treatment of a nonlinear coefficient

$$\int_{\Omega} T^2 \nabla T \cdot \nabla \psi \, d\Omega = 0$$

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

```
phi = [ phi1(r), phi2(r), phi3(r) ]
f( T ) -> T^2

interpolate( f, phi ) -> {
f_interp( r, [T1, T2, T3] )
   -> f( T1*phi1(r) +
         T2*phi2(r) +
         T3*phi3(r) )
}
```

- 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

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

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

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

phiOuterPhi = tensorProd( phi, phi );

coefF = f(T) -> T^2
coefInterp = interpolate( coefF, phi );

elementMatrixF = integrate( phiOuterPhi*
                            coefInterp,
                            rc,
                            w )
```

- 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]

Disassembly of code

```
call sym std::istream
movsd xmm4, qword [rsp + 0x18]
movsd xmm0, qword [0x00000e10]
movapd xmm3, xmm2
movsd xmm1, qword [0x00000e08]
mulsd xmm2, xmm0
movapd xmm5, xmm4
mulsd xmm3, xmm1
mulsd xmm1, xmm4
mulsd xmm5, xmm0
movsd xmm4, qword [0x00000e28]
addsd xmm1, xmm2
addsd xmm3, xmm5
movapd xmm0, xmm1
movapd xmm2, xmm3
movapd xmm5, xmm1
addsd xmm0, xmm3
mulsd xmm3, xmm4
mulsd xmm5, xmm4
mulsd xmm0, qword [0x00000e18]
movsd qword [rsp], xmm0
movsd xmm0, qword [0x00000e20]
mulsd xmm1, xmm0
mulsd xmm2, xmm0
addsd xmm3, xmm1
addsd xmm2, xmm5
movapd xmm0, xmm3
call sym std::ostream
```

$$\int\limits_{\Omega} T\varphi_i \cdot \varphi_j \, d\Omega\,; \quad i = 1, 2,\ j = 1, 2$$

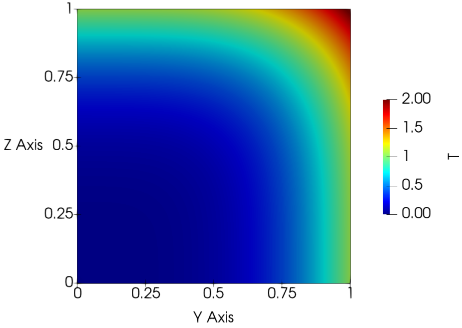
- Disassembly of the code which calculates local stiffness matrix and output it to the standard output stream is presented on the slide
- There are no function calls and class instances such as a "tuple" in this code
- C++ compiler efficiently optimized the code generated by the methods described on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

Test on solution of Poisson equation

$$\int_{\Omega} \nabla T \cdot \nabla \varphi \, d\Omega = -12x - 12y - 12z$$

```
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

Solver	Calculation of $[K]$, s
FEniCS	0.21
FEMEngine	1.07
FreeFem++	8.32

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.

Future release of the source code

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 its ready.