Modern C++ approaches to FEM modelling

Gurin A.M., Baykin A.N.

Lavrentyev Institute of Hydrodynamics of SB RAS

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

```
int main() {
   auto increment = []( int& val ){ val++; };
   int v = 0;
   increment( v );
   std::cout << v << '\n';
}
```

```
auto create_function( int& val ) {
    return [&val](){ val++; };
}

int main() {
    int v = 0;
    auto increment = create_function(v);
    increment();
    std::cout << v << '\n';
}
```

- Lambda function is an anonymous function which can be stored in variable
- Lambda function can be returned from ordinary function or from another lambda function
- Lambda function can capture external variable

Variadic templates

```cpp
template< class ... types >
struct Container {
   std::tuple<types ...> values;

   void initialize_values( std::tuple<types ...> v ) {
       values = v;
   }
};

int main() {
   Container< double, std::string, int > c;
   c.initialize_values( std::make_tuple( 5.5, "some_string", 7 ) );
}
```

- Variadic template has parameter pack that can accept any number of types
- Access pattern to parameter pack had to be known at compile time

Fold expressions

```
template< class ... types >
auto reduce_parameter_pack( types ... t ){
   return ( t + ... );
}
```

- Fold expression simplifies reduction operation on parameter pack
- Fold expression can be used with most binary operations

Newton method. Solver.

```
template< class F, class Der >
struct Newton {
   F func;
   Der der;
   Newton( F func, Der der ) : func(func), der(der) {}

   double solve( double x0, double tol, int N ) {
       double XOld = x0;
       double XNew = 0;
       for( int i = 0 ; i < N ; i++ ) {
           XNew = XOld - func(XOld) / der(XOld);
           if( fabs( func(XNew) ) < tol ) break;
           XOld = XNew;
       }
       return XNew;
   }
};
```

Newton method. Solver.

```
template< class F, class Der >
struct Newton {
   F func;
   Der der;
   Newton( F func, Der der ) : func(func), der(der) {}

   double solve( double x0, double tol, int N ) {
       double XOld = x0;
       double XNew = 0;
       for( int i = 0 ; i < N ; i++ ) {
           XNew = XOld - func(XOld) / der(XOld);
           if( fabs( func(XNew) ) < tol ) break;
           XOld = XNew;
       }
       return XNew;
   }
};
```

Newton method. Main function.

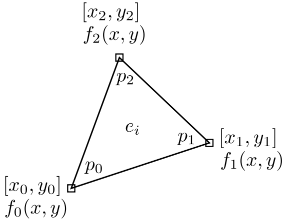
```
template<class F>
auto generateDerivativeCentral( F func, double hx ) {
    return [=](double x){
        return ( func( x + hx*0.5 ) - func(x - hx*0.5) ) / hx;
    };
}

int main()
{
    double a = 1;
    double U0 = 1;
    auto funcToSolve = [=]( double x ){
        return ctg( sqrt( 2.0 * a * U0 * ( 1.0 - x ) ) )
                - sqrt( 1.0 / x - 1.0 );
    };
    auto derivative = generateDerivativeCentral( funcToSolve, 0.0001 );
    Newton solver( funcToSolve, derivative );
    auto solution = solver.solve( 0.8, 0.0001, 100 );

    std::cout << "solution = " << solution << "\n";
}
```

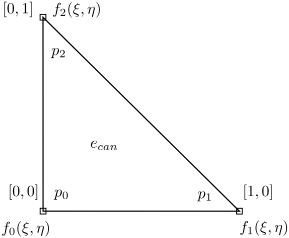
Newton method. Disassembly.

```
.LCPI1_0:
        .quad   4603614670303855196     # double 0.60390034734889353
main:                                   # @main
        push    rax
        mov     edi, offset std::cout
        mov     esi, offset .L.str
        mov     edx, 11
        call    std::basic_ostream<char, std::char_traits<char> >& st
        movsd   xmm0, qword ptr [rip + .LCPI1_0] # xmm0 = mem[0],zero
        mov     edi, offset std::cout
        call    std::basic_ostream<char, std::char_traits<char> >& st
        mov     esi, offset .L.str.1
        mov     edx, 1
        mov     rdi, rax
        call    std::basic_ostream<char, std::char_traits<char> >& st
        xor     eax, eax
        pop     rcx
        ret
```

- Disassembly of executable compiled with Clang 8.0

Finite element method. Basic formulation.



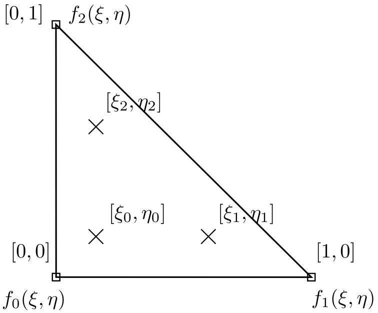


$$\int_{e_i} f_i(x,y) f_j(x,y)\, dx dy$$

$$\int_{e_{can}} f_i(\xi,\eta) f_j(\xi,\eta) |J_{2D}|\, d\xi d\eta$$

$$\int f_i f_j \rightarrow \begin{pmatrix} \int f_0 f_0 & \int f_0 f_1 & \int f_0 f_2 \\ \int f_1 f_0 & \int f_1 f_1 & \int f_1 f_2 \\ \int f_2 f_0 & \int f_2 f_1 & \int f_2 f_2 \end{pmatrix}$$

Finite element method. 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

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

Finite element method. Algorithm.

- Perform function multiplication to obtain 3x3 matrix of functions

$$g_{i,j}(\xi,\eta) = f_i(\xi,\eta) f_j(\xi,\eta)$$

$$G = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \end{pmatrix} \begin{pmatrix} f_0 & f_1 & f_2 \end{pmatrix} = \begin{pmatrix} f_0 f_0 & f_0 f_1 & f_0 f_2 \\ f_1 f_0 & f_1 f_1 & f_1 f_2 \\ f_2 f_0 & f_2 f_1 & f_2 f_2 \end{pmatrix} = \begin{pmatrix} g_{0,0} & g_{0,1} & g_{0,2} \\ g_{1,0} & g_{1,1} & g_{1,2} \\ g_{2,0} & g_{2,1} & g_{2,2} \end{pmatrix}$$

- Through currying apply quadrature to a functions

$$q_{i,j}(|J_{2D}|) = (g_{i,j}(\xi_0,\eta_0)\omega_0 + g_{i,j}(\xi_1,\eta_1)\omega_1 + g_{i,j}(\xi_2,\eta_2)\omega_2)|J_{2D}|$$

$$Q = \left(q_{i,j}(|J_{2D}|)\right) i = 0...2; j = 0...2$$

- Use Q to obtain local matrix by providing Jacobian of element in loop over all elements
- Assemble global matrix and solve with linear system solver

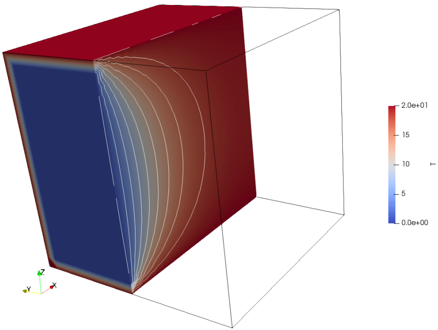
Finite element method. Function multiplication.

```
template<class T>
struct traits {};

template<class R, class T, class ... args>
struct traits< R (T::*)( args ... ) const > {

};

template<class A, class B, class R, class T, class ... args>
auto multiplyFunctions(A f1, B f2, traits< R (T::*)( args ... ) const >) {
   return [=]( args ... v ){ return f1( v... ) * f2( v... ); };
}

template<class A, class B>
auto multiplyFunctions(A f1, B f2) {
   return multiplyFunctions( f1, f2, traits< decltype(&A::operator()) >{} );
}
```

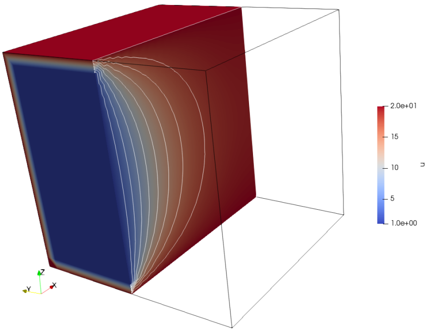
Finite element method. Tensor multiplication.

```
template<class ...Args1, class ...Args2, size_t ...Is>
auto tensorProduct(std::tuple<Args1...> t1,
                   std::tuple<Args2...> t2,
                   std::index_sequence<Is...>) {
   return std::make_tuple(
       tupleBinaryOperation( t1,
                             std::get<Is>(t2),
                             std::multiplies{} ) ...
       );
}

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

Laplace equation. Comparison with FreeFem++.



FemEngine



FreeFem++

- Matrix assemble time of Freefem++ 0.297 s of our code 0.104 s

Laplace equation. Comparison of codes.

```
CompoundFEMSpace dofs;
FEMSpace P1Space( mesh, dofs );

FieldFEM< Element3DTetra1Order, 0 > T = P1Space.createField< Element3DTetra1Order,
0 >( "T" );

auto gradT = grad(T);
auto gradTMul = scalarMul( gradT, gradT );
auto integratedGradTMul = integrate( gradTMul, Quadrature3DTri::GaussOrder3{} );

EquationFEM eq( P1Space, mesh, std::move( solver ) );

eq.addBoundaryCondition( "top",
                        T.getInnerRepresentation(),
                        [](double x, double y, double z){return 1.0;} );
eq.addBoundaryCondition( "sides",
                        T.getInnerRepresentation(),
                        [](double x, double y, double z){return 20.0;} );

eq.addToGlobalMatrix( integratedGradTMul );
eq.solve();
```

FemEngine

```
varf lap( u, uu, solver=sparsesolver,tgv=ttgv) =
    int3d(Th, mpirank)(
        dx(u) * dx(uu) + dy(u) * dy(uu) + dz(u) * dz(uu)
    ) + on(labelToSetBC, u = 1)+ on(2, u = 20);

varf rhsForm(u, uu, solver=sparsesolver,tgv=ttgv) = on(labelToSetBC, u = 1) +
                                                     on(2, u = 20);

matrix A = lap(Vh,Vh,solver=sparsesolver,tgv=ttgv);

real [int] b = rhsForm(0, Vh,tgv = ttgv);
real[int] rinfo(40);
int[int] info(40);

set(A, solver = sparsesolver,tgv = ttgv ,master=-1, rinfo=rinfo, info=info);

u[] = A^-1*b ;
```

FreeFem++

Conclusions

- Last decade evolution of C++ has added number of features especially useful for FEM modelling
- Lambda functions and operation on them are effectively optimizable by modern C++ compilers
- FEM engine were implemented with its core in functional paradigm on C++17