Basic Ideas

Colsamm is a C++ library, that provides the mechanism and methods for the efficient and user-friendly computation of local stiffness matrices.

Contents

1 Mathematical Introduction

The finite element method (FEM) is a mechanism for discretizing partial differential equations (PDE). Thereby, the PDE specific domain gets divided by an approximating grid that divides the domain into many elements. In order to result a finite problem out of the infinite PDE, one chooses finite base functions on this elements, and achieves so-called finite elements. By transforming the PDE to its weak formulation one can use the finite elements to receive a discretized version of differential operators, that achieve a sparsely filled matrix. However, the complex mathematical theory behind these simple ideas bases on the Sobolev spaces, in order to ensure the existence and uniqueness of the resulted solution.

2 Goals of Colsamm

The goal of **Colsamm** project was to build a C++-library for computing the local stiffness matrices resulting by the FE discretization of differential operators. The operator overloading in C++ should provide a user-friendly interface, in order to realize an implementation, that is very close to the weak formulation of the PDE. Additionally, since this very fine grain application will be very frequently used while assembling global stiffness matrices, the library has to be very efficient. The building of the stiffness matrices may not be the long taking part while solving a PDE. Nevertheless, an additional goal was the flexibility in the use of **Colsamm**. It ought to work in one dimensional problems up to three dimensions, realizing several finite element types, including higher order and user-defined basis functions, iso-parametric elements, boundary elements, a complete set of operators for the easy coding of the weak formulations of any PDEs.

3 A Simple Example

First, we want to present a simple example, demonstrating the steps for using **Colsamm**. Even if it may be boring for some readers, we want to start with the two dimensional Poisson problem on a domain Ω, that is discretize by a triangle grid:

$$\Delta u(x, y) = f$$

with some boundary conditions, e.g. Dirichlet boundary 0. This then leads to a weak formulation that look like

$$\int_\Omega \nabla u(x, y) \nabla v(x, y) \mathbf{d}(x, y) = \int_\Omega f(x, y) v(x, y) \mathbf{d}(x, y)$$

for any $v \in V$, the space of testing functions. This weak formulation becomes on the triangulation Ω_h:

$$= \int_{\Omega_h} f(x,y) v_h(x,y) \mathbf{d}(x,y)$$

with the finite functions $v_h \in V_h$. Restricting u and f in the same space V_h, we receive the two operators:

$$a(v,w) = \int_{\Omega_h} \nabla v(x,y) \nabla w(x,y) \mathbf{d}(x,y)$$

and

$$h(v,w) = \int_{\Omega_h} v(x,y) w(x,y) \mathbf{d}(x,y)$$

Thus we start our program by including the **Colsamm** header file. Since **Colsamm** is a C++ template library, there is no object file that can be linked. All compilation of **Colsamm** has to be done every time we have changings within the **Colsamm**-dependent files.

```
#include "Colsamm.h"
```

The whole **Colsamm** code is embedded into the namespace `_COLSAMM_`. Since we want to spare the explicit namespace before every method, we just introduce the namespace into any existing one:

```
using namespace ::_COLSAMM_;
```

As next step we initialize our finite element type, in this case a triangle. All predefined finite elements within **Colsamm** are encapsulated in an additional namespace called `ELEMENTS`. The predefined elements have an implementation concerning linear basis functions. Thus, we define our element:

```
ELEMENTS::Triangle my_element;
```

Additionally, we have to define some variables, that will contain the coordinates of the vertices of the triangles and matrices, that will contain the local stencils computed by **Colsamm**:

```
std::vector < double > corners(6,0.);
std::vector < std::vector < double > > stencil_a, stencil_h;
```

After these initialization steps, we can start with the computation of the local stencils for the operators for a and h. Therefore, we iterate over all faces of the discretized domain. We store the coordinates into `corners` and pass them via

```
my_element ( corners ) ;
```

As soon we have initialized the element with the actual set of the coordinates of the vertices, we can use **Colsamm** for computing the stencils on the actual element:

```
stencil_a = my_element.integrate (grad(v_()) * grad(w_())) ;
stencil_h = my_element.integrate ( v_() * w_() ) ;
```

Thereby denote `v_()` and `w_()` the set of basis functions on this element. Adding the entries from these local stiffness matrices to the suitable position within the global discretization matrices and restarting with a new set of coordinates assembles the discretization of the operators a and h on the grid Ω_h.

4 Working with Colsamm

4.1 Predefined Finite Elements

Similar to the triangle finite element in the above example, **Colsamm** implements several types of finite elements, concerning linear basis functions. Each of these predefined finite elements are provided with three template parameters. The first one addresses the integration accuracy of the Gaussian quadrature formula. The user can choose out of:

- `Gauss1`, exact integration for polynomials of order 1
- `Gauss2`, exact integration for polynomials of order 3
- `Gauss3`, exact integration for polynomials of order 5

As standard, we choose `Gauss2`, which is enough for mostly all computations concerning linear basis functions. The second template parameter represents the underlying variable type, `double` or `complex<double>`, set to `double` by default. At least the third template parameter steers the data structure that will be used for the stencils. It can be chosen of

- `STLVector`, the STL vector of the type set by template parameter two, or
- `Array`, a array of type parameter two.

The dimension of the computed stencil depends on the problem, i.e. the sets of ansatz and testing functions underlying the integrand expression. Finally, we list the different predefined element types:

- in 1D:
  - `_Interval_<Gauss1/2/3,double/complex,STLVector/Array>`
  - `Interval` $\widehat{=}$ `_Interval_<Gauss2,double,STLVector>`
- in 2D:
  - `_Triangle_<Gauss1/2/3,double/complex,STLVector/Array>`
  - `Triangle` $\widehat{=}$ `_Triangle_<Gauss2,double,STLVector>`
  - `_Quadrangle_<Gauss1/2/3,double/complex,STLVector/Array>`
  - `Quadrangle` $\widehat{=}$ `_Quadrangle_<Gauss2,double,STLVector>`
  - `_Edge2D_<Gauss1/2/3,double/complex,STLVector/Array>`
- in 3D:
  - `_Tetrahedron_<Gauss1/2/3,double/complex,STLVector/Array>`
  - `Tetrahedron` $\widehat{=}$ `_Tetrahedron_<Gauss2,double,STLVector>`
  - `_Hexahedron_<Gauss1/2/3,double/complex,STLVector/Array>`
  - `Hexahedron` $\widehat{=}$ `_Hexahedron_<Gauss2,double,STLVector>`
  - `_Cuboid_<Gauss1/2/3,double/complex,STLVector/Array>`
  - `Cuboid` $\widehat{=}$ `_Cuboid_<Gauss2,double,STLVector>`
  - `_Pyramid_<Gauss1/2/3,double/complex,STLVector/Array>`

– `Pyramid` $\widehat{=}$ `_Pyramid_<Gauss2,double,STLVector>`

– `_Prism_<Gauss1/2/3,double/complex,STLVector/Array>`

– `Prism` $\widehat{=}$ `_Prism_<Gauss2,double,STLVector>`

– `_Triangle3D_<Gauss1/2/3,double/complex,STLVector/Array>`

– `_Quadrangle3D_<Gauss1/2/3,double/complex,STLVector/Array>`

4.2 Integrand Operations

The weak formulation is a sum of integrals over finite elements with their corresponding ansatz and testing functions. Since these functions can be elements of different sets of functions, we provide three types:

- `v_()` ansatz function for the discretized variable out of set 1. Alternatively, one can define and use a variable of type `Ansatz_Function`.
- `w_()` testing function of the PDE element of set 1, as well, or using a variable of type `Testing_Function`.
- `p_()` testing function of the PDE, but element of a different set 2, or initialize a variable of type `Mixed_Function`.

Since **Colsamm** handles computations of local stiffness matrices from one dimensional PDEs up to three dimensions, we provide operators to realize the gradient and the partial derivatives:

- `grad(.)` the gradient vector of a two or three dimensional variable
- `grad2(.)` the gradient vector of a two dimensional variable, only
- `d_dx(.)`, `d_dy(.)`, `d_dz(.)` the partial derivative in the chosen direction.

These operators concerning derivatives cannot occur in a nested manner. They have to be directly applied on a ansatz or testing function. Additionally, one can arrange any sequency of those combined with the

- standard binary operators `+ , * , - ,/`.
- standard unary operator `-` .

Further, these expressions can contain constants, double or complex. However, since the underlying Fast Expression Template implementation works only with types during the evaluation, all constants have to be wrapped in an constant class and provided with an expression-unique enumeration. Therfore, we provide the two container for constants

- `Double_<.>(.)` and `Complex_<.>(.)`.
- Example: `Double_<1>(2.)` or `Double_<2> aConst(-3.)`.

The template has to be an integer, that has to be unique for the expression within the constant it is used. This encapsulating and enumeration is a kind of strange in first sight, but it has an deep impact on the performance of the expression template implementations. The same principle underlies the introduction of user-defined functions, where one can wrap one to three dimensional

functions in the corresponding Fast Expression Template class, by using the constructor `func<.>(.)`. Herein the value within the brackets has to be again an enumerating integer, and the parenthesis take the functions of the following types:

- `double f(double)`, `complex f(complex)`
- `double f(double,double)`, `complex f(complex,complex)`
- `double f(double,double,double)`, `complex f(complex,complex,complex)`

Further, we have operators for conjugate complex values (`conj(.)`), computing the square root (`Sqrt(.)`), calculating the power of values (`Pow<.>(.,.)`), the last one again taking a unique template integer. In a similar manner polynomials are introduced, taking the exponent as template parameter:

- `x_<exponent>()`
- `y_<exponent>()`
- `z_<exponent>()`

Additionally, one can define vectors and matrices of these operands. Therefore we have the operator `&` to separate the entries of a vector and the entries in a row of a matrix. `|` is the marker for the end of a matrix line. Vectors can be added subtracted and multiplied to other vectors and to matrices. Vectors and matrices can be defined by using the `Define_Integrand(.) name` macro, that stores the type of the data in the parenthesis into the variable name.

```
Define_Integrand ( d_dx (v_()) & d_dy (v_()) & d_dz (v_()) ) grad_v;
```

would define the gradient vector of ansatz functions as variable `grad_v`.

Further, there is are operators denoting the normal vector `N_()` and the unit normal`N_Unit()` on a surface. These vectors always correspond to the surface limited by the first n points passed to the finite element. Means, in the case of a tetrahedron, a three dimensional triangle with the first three vertices as corners.

The evaluation of the integrals is accomplished by the `integrate` member function that is provided with two interfaces:

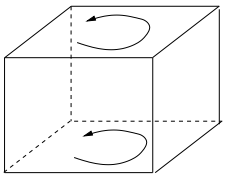
```
template < class Integrand >
const Predefined_Stencil_TYPE&
 integrate ( const Integrand& a_ );

t*emplate < typename StencilType, class Integrand >
void
 integrate ( StencilType& stencil, const Integrand& a_ );
```

The first version takes just the integrand as parameter and returns a value, vector, matrix, or tensor, dependent on the data structure defined while initializing the finite element. The set of predefined elements have a nested STL vector as underlying data structure, i.e. in these cases, the data type is built of STL vectors. The dimension of the result depends on the integrand passed. Integrand contains

- no basis function parts (i.e. `v_,w_,p_` or variables of the corresponding types) the result will be a single value.
- one basis function part, then the result will be a STL vector of the corresponding amount of basis functions.
- two different basis function parts, thus the returned variable will be a STL vector of STL vectors, each size corresponding to the ansatz or testing basis function set, respectively.
- three different types of basis function types, then the resulting variable has the type of a three dimensional STL vector.

4.3 Passing vertices to Colsamm

The parenthesis operator builds the interface for passing the vertices of the actual finite element to **Colsamm**. The passed data structure has to be one-dimensional, the components accessible via the square brackets operator (`[]`). Within, the components of each vertex are listed successively, i.e. starting with the components of vertex 1, followed by the components of the second vertex, and so on. The vertices them self have to be listed anticlockwise, and in the case of hexahedrons or cuboids, one has first to pass the bottom quadrangle anticlockwise and then the top quadrangle, starting with that vertex lying above the first one, in the same manner.



However, this ordering depends on the manner the formula concerning the mapping is built. While implementing user-defined elements and generating suitable mappings, one can steer the numbering of the vertices by adapting the formula of the mapping.

5 User-defined Elements

As very feature of **Colsamm** it is very simple to define new finite element types. The user has therefore just to implement its own finite element by deriving from the class `_Domain_` with specific template parameters. Additionally, one has to define a mapping (if there is no suitable one already defined) and list the basis functions that will be used.

5.1 Introducing Mappings

Since one of the goals of **Colsamm** was to combine user-friendliness with efficiency, we use the Fast Expression Template technique, that computes the results of expressions by building a expression dependent object which is first evaluated when the assignment operator is called. Fast Expression Templates work just on the types of the objects, without any internal member variables.

As consequence, all operations with constants have to be packed into a suitable class, and since the constant will be stored as a static variable of this type, the class types have to be enumerated by a template integer. This was already mentioned while presenting the operators concerning the integrands above.

The same mechanism has to be applied while defining a mapping. We provide therefore a macro called `Define_Element_Transformation` which starts a typedef of the passed formula. For instance, the definition of the mapping concerning a tetrahedron is implemented as follows:

```
Define_Element_Transformation
       (
         P_0() +
         ( P_1() - P_0()) * _U() +
         ( P_2() - P_0()) * _V() +
         ( P_3() - P_0()) * _W()
       )
         Tetrahedron_Transformation;
```

Herein, the `P_*()` denote the vertices of the element, beginning with zero. `_U()`, `_V()` and `_W()` are standing for the different spatial directions of the mapping formula, i.e.

$$T(u, v, w) = P_0 + (P_1 - P_0)u + (P_2 - P_0)v + (P_3 - P_0)w$$

Since **Colsamm** uses the same mechanism for defining element mappings, there are several predefined transformations within the library (listed in `Elements.h`)

- `Hexahedron_Transformation`
- `Cuboid_Transformation`
- `Tetrahedron_Transformation`
- `Pyramid_Transformation`
- `Prism_Transformation`
- `Triangle_Transformation`
- `Quadrangle_Transformation`
- `Interval_Transformation`

5.2 Introducing Elements

The declaration of a new finite element is accomplished by the implementation of a new class, that is derived from the class `_Domain_` with the element specific template parameters. These parameters are:

```
_Domain_ < \
(1.) number of vertices,
(2.) dimension D1 / D2 / D3 ,
(3.) mapping from reference to a general element,
(4.) interior / boundary / boundary_with_derivatives.
```

```
(5.) number of basis functions of set 1,
(6.) number of basis functions of set 2,
(7.) geometry of reference element,
(8.) Gaussian quadrature formula, Gauss1 / Gauss2 / Gauss3
(9.) double / complex<double>,
(10.)STLVector / Array  >
```

1. `number of vertices` is a positive integer, that denotes the number of the vertices of the element. In case of a tetrahedron, this would be 4.

2. `dimension` is an enumeration type which can have the values

   - `D1` a one dimensional element
   - `D2` a two dimensional element
   - `D3` a three dimensional element

3. `mapping` is a type, defined by the mechanism presented in the previous section. This type contains the mapping formula from a reference element to a general element.

4. attach the suitable computations to the mapping.

   - `interior` this element is an interior element. The first derivatives are computed as well as the Jacobian determinant.
   - `boundary` this element lies at the boundary of the domain. We have the mapping values with the surface normal
   - `boundary_with_derivatives` similar to the boundary element before, however, here we additionally compute the derivatives

5. `number of basis function of set 1`, amount of basis functions of set 1. The `Ansatz_Functions` (`v_`) and the `Testing_Functions` (`w_`) are taken out of this set. At this point, only the number of basis functions in set 1 has to specified. The definition of the basis functions them self will be accomplished in the constructor of the element.

6. `number of basis functions of set 2`, amount on functions within the second set. The `Mixed_Functions` (`p_`) will be chosen out of this set. If the problem is not a mixed problem, this set stays empty and this template parameter is 0.

7. The `geometry of the reference element` has one of the enumeration values:

   - `cuboid`, $0 \leq x, y, z \leq 1$
   - `tetrahedron`, $0 \leq x, y, z \leq 1 \wedge 0 \leq x + y + z \leq 1$
   - `pyramid`, $0 \leq x, y, z \leq 1 \wedge 0 \leq \max\{x, y\} + z \leq 1$
   - `prism`, $0 \leq x, y, z \leq 1 \wedge 0 \leq x + y \leq 1$
   - `quadrangle`, $0 \leq x, y \leq 1$
   - `triangle`, $0 \leq x, y \leq 1 \wedge 0 \leq x + y \leq 1$
   - `interval`, $0 \leq x \leq 1$

The first vertex of these reference elements is always in 0. The other vertices lie always on the axis.

8. `Gaussian qudarature formula` on the reference element defined previously. Since the accuracy of the Gaussian quadrature formula depends in the amount of Gaussian points that are chosen for the evaluation of the integral, we provide three versions for the accuracy:

   - `Gauss1`, evaluation at one Gaussian point, the stress. This yields and exact integration for integrands that are polynomials up to grade 1.
   - `Gauss2`, evaluation at several Gaussian points. The integration is exact for polynomials up to grade 3.
   - `Gauss3` (not implemented for tetrahedrons and pyramids yet), integrates exact for polynomials of grade 5.

9. `double or complex<double>`: if one wants to compute a problem concerning complex integrands or complex basis functions, this template parameter has to be set to `complex<double>`. Otherwise this should be set to double. Unfortunately, the actual version does not support the `float` data type.

10. `STLVector or Array` steers the returning type of the integrate function.

    - `STLVector` is a STL vector of the type set by the previous parameter. Internally, we initialize a three dimensional STL vector type.
    - `Array` initializes internally a three dimensional array of the previously defined type.

    By default, this value is set to `STLVector`. The `integrate` function chooses the dimension of the result by analyzing the integrand to be computed.

Concluding, the declaration of the finite element addressing tetrahedrons looks like:

```
struct _Tetrahedron_
 : public _Domain_<4,
                   D3,
                   Tetrahedron_Transformation,
                   interior,
                   4,
                   0,
                   tetrahedron,
                   Gauss2,
                   double,
                   STLVector>
```

5.3 Introducing basis functions

Additionally to the template inheritance from the `_Domain_` class presented before, one has to implement a constructor of the new class, that initializes the basis functions. Since the evaluation of the basis functions within the reference

element has only to happen once, we decided to implement a worse performing classical Expression Template implementation, in order to increase the user-friendliness. For the initialization of the basis functions we provide two methods:

- `Set`, setting the basis functions for set 1. As parameter the method takes a `Basis_Function_Expr`, explained below. In order to specify the namespace of the called method, one has to code: `this->Set(...)`.
- `Set_P`, method for introducing basis functions into set 2.

As basis functions expression, we again implemented the standard operators `+,-,*,/` and introduced a way to specify monomials (`exponent` always denoting a non negative integer):

- `X_(exponent)`, defining $x^{exponent}$
- `Y_(exponent)`, defining $y^{exponent}$
- `Z_(exponent)`, defining $z^{exponent}$

The exponent is set to 1 by default. Additionally, we provide a mechanism for implementing exponential basis functions:

- `Exp(expression)`, defining $\exp^{expression}$
- `Sin(expression)`, defining $\sin(expression)$
- `Cos(expression)`, defining $\cos(expression)$

Thus, the constructor concerning the tetrahedron is implemented as:

```
_Tetrahedron_( )
   {
      this->Set ( 1. - X_(1) - Y_(1) - Z_(1) ) ;
      this->Set ( X_(1)  ) ;
      this->Set ( Y_(1)  ) ;
      this->Set ( Z_(1)  ) ;
   }
```