Degrees of freedom

Credits for the logo of Morgana: Paola Infantino
(https://paolainfantino.com)

The authors wish to thank L. Barbareschi, for her valuable contribution and suggestions.

Contents

1 Morgana Kernel

This folder contains all the common utilities classes of for *Morgana*, in particular it contains the *staticAssert.hpp* class that makes it possible to verify the correctness of the use of the template structure during the compilation.

```
template<bool> class staticAssert;
template<>     class staticAssert<true>
{
public:
static const bool returnValue = true;
};
```

The class is rather simple and accepts only a template argument which can be $true$ or $false$. The class has only one specialization for the $true$ value and this means that, if the $false$ parameter is used, this results in a compilation error. The following example gives a hint of how the class can be used:

```
static const UInt A = 1;
static const UInt B = 1;
staticAssert<A == B> P; //This returns an error
```

2 Morgana Dofs

2.1 Introduction

Every single type used in *Morgana* is included in this folder. It includes all degrees of freedom and all entities to represent, for instance, the geometry. In particular, to date, *Morgana* implements the following degrees of freedom ($dofs$):

- *UInt* : the unsigned integer, a wrapper of *uint*, it is used for all indexes;
- *Real*: it is the wrapper of *double* and represents a scalar number;
- *point2d*: it represents the coordinates of a two dimensional point. It is used, for instance, to describe a vector of a two dimensional field;

- *point3d*: it is a three dimensional point so it can represent a three dimensional vector field but it is also largely used in the section $morganaGeometry$ to represent the coordinates of points in a mesh;
- *tensor2d*: the two dimensional tensor with four components;
- *tensor3d*: the three dimensional tensor with nine components. It can be used, for instance, to represent a stress field;
- *stateVector*: a dynamically sized vector (its size can be changed at run time). It is a wrapper of the dense vector contained in the package *epetra* of *trilinos* [4];
- *stateMatrix*: a dynamically sized matrix derived from the dense matrix of *trilinos-epetra*;
- *staticVector*: statically sized vector. This means its size is chosen using a template argument and cannot be modified at run time;
- *typeComplex*: the implementation of a complex number.

These $dofs$ cover all the current needs, however some more can be created and we will include some guidelines to create them. $dofs$ can be split into two categories:

- *dynamic dofs* ($stateVector$ and $stateMatrix$): they have a variable length that can be modified at run time;
- *static dofs* (all the other $dofs$): their size is known at compilation time.

We stress that the static degrees of freedom can be largely optimized simply due to the fact that the compiler knows their dimension, for instance the exchange of data between processes is quicker using the type $staticVector$ **in place of** $stateVector$**. We also point out that not all the** $Morgana$ **packages accept the** *dynamic* $dofs$**, for instance the finite element package** $morganaFiniteElement$ **only accepts** *static* $dofs$**.**
The $dofs$ implements the following functionalities:

- parallel communication using the data serialization utility of the third party package $boost - mpi$;
- algebraic operations such as sum, difference, product and division by a scalar value. Due to efficiency reasons these procedure are implemented using the *expression template* technique, see [1] for an introduction;
- multiplication by other degrees of freedom such as the scalar product of vectors and the product of a vector and a tensor. These functions are handled by a collection of classes in the folder $morganaDofs$ identified by the suffix *traits*;

- order functions, every dof must implement a series of functions that guarantee that they can be ordered in a *stl* list, see [3]. **The ordering scheme must guarantee that, given two degrees of freedom P_1 and P_2 with $P_1 \leq P_2$, then $P1 \leq (1-a)P_1 + aP_2 \leq P_2$ where a is a real number chosen in the interval $[0, 1]$;**

- print to screen functions.

2.2 Dofs structure

In Figure 1 we have depicted a scheme representing the $morganaDofs$ package. The class $morganaTypes$ contains some definitions:

```
#ifndef MORGANATYPES_HPP
#define MORGANATYPES_HPP

typedef double       Real;
typedef unsigned int UInt;
typedef int          Int;

static const Real geoToll = 1e-10;

enum morganaTypes{typeReal, typePoint2d, typePoint3d, typeTensor2d,
typeTensor3d, typeStateVector, typeStateMatrix, typeStaticVector, typeComplex };

#endif
```

To be more precise we rename the variables *double*, *unisgned int* and *int* so that they can be quickly substituted with higher precision versions. As we had already pointed out, the parameter *geoToll*, represents the absolute geometric tolerance: coordinates differing, in absolute value, less than this tolerance are considered equal. The *enum* type $morganaTypes$ associates to each type of $dofs$ an integer number, for instance, the class *point2d* is associated to 1. This association will be useful to check the correctness of template structures and during the debugging phase.

Along with the implementation of the $dofs$, three *trait* classes are present (for the use and implementation of trait classes please see [1]):

- *traitsBasic*: completes some basic information about $dofs$ types;

- *traitsMultiply*: handles the product of various types;

- *traitsMpiOptimization*: manages the optimization of *static* $dofs$ types.

These latter classes will be discussed in detail in Section 2.4. We end this general presentation by describing the $typesInterface.hpp$ class which completes the expression template mechanism used to implement the sum and difference of $dofs$ of the same type and, at the meantime, it provides a general interface for the $morganaDof$ package. In other words it is sufficient to include the $typesInterface.hpp$ file to access all the dof types.

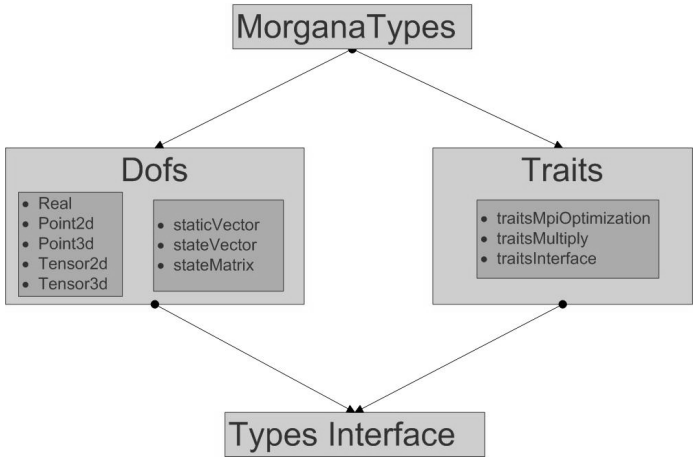

Figure 1: Structure of the *morganaDofs* package.

2.3 An example: point3d

Now we discuss in detail the implementation of the *point3d* class as an example of the implementation of *dofs* classes. The class has a long preamble that is used to implement the expression template structure for fundamental algebraic functions:

```
//-----------------------------------------------------------
// EXPRESSIONS
//-----------------------------------------------------------
//The sum operation
template<typename LHS, typename RHS>
struct plusPoint3d
{
static Real applyX(const LHS & left, const RHS & right)
{ return(left.getX() + right.getX()); }
static Real applyY(const LHS & left, const RHS & right)
{ return(left.getY() + right.getY()); }
static Real applyZ(const LHS & left, const RHS & right)
{ return(left.getZ() + right.getZ()); }
};

//The difference operation
template<typename LHS, typename RHS>
struct minusPoint3d
{
static Real applyX(const LHS & left, const RHS & right)
{ return(left.getX() - right.getX()); }
static Real applyY(const LHS & left, const RHS & right)
{ return(left.getY() - right.getY()); }
static Real applyZ(const LHS & left, const RHS & right)
{ return(left.getZ() - right.getZ()); }
};

//The multiplication operation
template<typename LHS>
struct multPoint3d
{
static Real applyX(const LHS & left, const Real & right)
{ return(left.getX() * right); }
static Real applyY(const LHS & left, const Real & right)
```

```
{ return(left.getY() * right); }
static Real applyZ(const LHS & left, const Real & right)
{ return(left.getZ() * right); }
};

//The multiplication operation
template<typename LHS>
struct divPoint3d
{
static Real applyX(const LHS & left, const Real & right)
{ return(left.getX() / right); }
static Real applyY(const LHS & left, const Real & right)
{ return(left.getY() / right); }
static Real applyZ(const LHS & left, const Real & right)
{ return(left.getZ() / right); }
};

//The difference operation
template<typename LHS, typename RHS>
struct rotPoint3d
{
static Real applyX(const LHS & left, const RHS & right)
{ return(left.getY() * right.getZ() - right.getY() * left.getZ()); }
static Real applyY(const LHS & left, const RHS & right)
{ return(left.getZ() * right.getX() - right.getZ() * left.getX()); }
static Real applyZ(const LHS & left, const RHS & right)
{ return(left.getX() * right.getY() - right.getX() * left.getY()); }
};

//The expression template
template<typename LHS, typename RHS, typename OP>
struct expressionPoint3d
{
static const morganaTypes myType = typePoint3d;

const LHS & left;
const RHS & right;

expressionPoint3d(const LHS & Left,
const RHS & Right) : left(Left), right(Right) { }

Real getX() const { return(OP::applyX(left,right)); }
Real getY() const { return(OP::applyY(left,right)); }
Real getZ() const { return(OP::applyZ(left,right)); }
};



//-------------------------------------------------------------
// OPERATORS
//-------------------------------------------------------------
//The rotor operator
template<typename LHS, typename RHS>
expressionPoint3d<LHS,RHS,rotPoint3d<LHS,RHS> >
operator^(const LHS & left, const RHS & right)
{
return(expressionPoint3d<LHS,RHS,rotPoint3d<LHS,RHS> >
(left,right));
}
```

In other words all these classes along with other ones, that will be introduced later on, allow to implement the sum or the subtraction of $3d$ points in an intuitive and efficient manner such as $P_1 + P_2$ where P_1 and P_2 are two points. In general, for each degree of freedom, the following algebraic operations are implemented:

- $+$ sum of two $dofs$;
- $-$ subtraction between two $dofs$;
- $*$ multiplication of a dof by a scalar value **and not viceversa**;
- / division of a dof by a scalar value **and not viceversa**.

The *point3d* class, in particular, implements also the vector product $\wedge$. Therefore it is possible to express the vector product in this way: $P_3 = P_1 \wedge P_2$ where vector P_3 will contain the output. **Be careful not to use an expression like $P_1 = P_1 \wedge P_2$, this may produce some wrong results since the components of P_1 are updated one by one and therefore the computation of the second component would use an updated first component of vector P_1.**
Then we pass to describe the parallel support for the *point3d* class, i.e.:

```
class point3d
{
/*! @name Parallel support */ //@{
private:
friend class boost::serialization::access;

template<class ARK>
void serialize(ARK & ar, const unsigned int version);
//@}


/*! @name Internal data */ //@{
public:
static const morganaTypes myType = typePoint3d;
```

The declaration *boost* :: *serialization* :: *access* associates this type with the *boost* serialization library used for parallel communication. The function *serialize* is necessary to reduce each dof to a string of characters, i.e. a type that *mpi* can handle directly. The variable *myType* is an *enum* variable that contains a number associated with the class according to what shown in the file *morganaTypes.hpp*. Each dof has a variable *myType* linked to its own code number.
Then we pass to the description of the constructors:

```
/*! @name Constructors */ //@{
public:
/*! Constructor */
point3d(Real xx=0.0, Real yy=0.0, Real zz=0.0);

/*! Constructor */
template<typename LHS, typename RHS, typename OP>
point3d(const expressionPoint3d<LHS,RHS,OP> & Expression);

/*! Copy constructor */
point3d(const point3d & V);

/*! Destructor */
virtual ~point3d();
//@}
```

This is a standard definition of the constructors except for the fact that we have implemented also a constructor that accepts the expression template *expressionPoint3d*. This is part of the mentioned expression template mechanism we will describe in Section 2.5.
Then the part

```
/*! @name Set functions */ //@{
public:
/*! Set all components of the point */
void set(const Real & xx, const Real & yy, const Real & zz);

/*! Set the X component */
void setX(const Real & xx);

/*! Set the Y component */
void setY(const Real & yy);

/*! Set the Y component */
void setZ(const Real & zz);

/*! Set the i-th component */
void setI(const UInt & i, const Real & val);

/*! Set the Id*/
void setId(const UInt & Id);
//@}


/*! @name Get functions */ //@{
public:
/*! Get the X component */
inline Real getX();
inline const Real & getX() const;

/*! Get the Y component */
inline Real getY();
inline const Real & getY() const;

/*! Get the Z component */
inline Real getZ();
inline const Real & getZ() const;

/*! Get the i-th component */
inline Real getI(UInt i);
inline const Real & getI(const UInt & i) const;

/*! Get the Id */
UInt getId() const;
//@}
```

implements the classic access and setting functions for point coordinates. The definition of operators

```
/*! @name Operators */ //@{
public:
/*! Access operator */
Real operator()(const UInt & i);

/*! Access operator */
const Real & operator()(const UInt & i) const;

/*! The equality operator */
point3d & operator=(const point3d & V);

/*! The equality operator for expressions */
template<typename LHS, typename RHS, typename OP>
void operator=(const expressionPoint3d<LHS,RHS,OP> & Expression);

/*! Sum operator */
void operator+=(const point3d & V);

/*! Subtract operator */
void operator-=(const point3d & V);
```

```
/*! Product operator */
void operator*=(const Real & a);

/*! Division operator */
void operator/=(const Real & a);

/*! Scalar product */
Real operator*(const point3d &V) const;

/*! Curl operator */
void rotor(const point3d & Px, const point3d & Py, const point3d & Pz);
//@}
```

contains both the mandatory and the optional operators. In particular each $dofs$ class must implement the following operators

- $=$: assign operator;
- $+=$: adds an instance V of the class to the current class;
- $-=$: subtracts an instance V of the class from the current class;
- $*=$: multiplies the class by a scalar factor a;
- $/=$: divides the class by a scalar factor a.

All other operators are optional. The ordering functions

```
/*! @name Ordinal functions */ //@{
public:
/*! Less operator: a vector is "less" than another if its
first coordinate is less than the other vector one.
If the first component value is equal the second one
is considered and so on */
bool operator<(const point3d & V) const;

/*! Inequality operator: two vectors are equal if their
components are equal to the geometric tolerance */
bool operator!=(const point3d & V) const;
//@}
```

make it possible to create an ordered list of points. Given two points P_1 and P_2 if the first one has an x coordinate smaller than the corresponding x coordinate of P_2 then we set $P_1 < P_2$. If P_1 has a coordinate x greater than the corresponding coordinate of P_2 then $P_1 > P_2$. If two points have the same x coordinate within the tolerance *geoToll* defined in *morganaTypes.hpp*, then we pass to compare the y coordinate and so on. If all the coordinates of the two points differ less than the *geoToll* tolerance, then they are considered equal. The remaining part of the class

```
/*! @name Combinatorial functions */ //@{
public:
point3d combinationGS(const point3d & N) const;
//@}


/*! @name Other functions */ //@{
public:
/*! The norm of the R3 vector */
Real norm2() const;

/*! Static norm function */
static Real norm2(const point3d & P);
```

```
/*! Static dot function */
static Real dot(const point3d & P1, const point3d & P2);

/*! Clearing function */
void clear();

/*! Outstream operator */
friend ostream & operator<<(ostream & f, const point3d & P);
//@}
```

implements the output operator << ad some other optional functions.

2.4 Traits classes

The trait classes define some other information of the $dofs$ classes and how they interacts with other parts of the code. The first class we are going to introduce is the $traitsBasic$ one. This class owns different specializations depending on the considered dof, **only the static degrees of freedom are here included**. Let us first introduce the specialization for the $Real$ type

```
/*! Real trait */
template<> class traitsBasic<Real>
{
public:
static const UInt numI = 1;
static const UInt numJ = 1;
static const morganaTypes myType = typeReal;

public:
traitsBasic();
Real getZero() const;
Real getUnity() const;
Real getUnityIJ(const UInt & i, const UInt & j) const;
Real getIJ(const UInt & i, const UInt & j, const Real & V);

public:
void setIJ(const UInt & i, const UInt & j,
const Real & value, Real & V);
void setZero(Real & V);

public:
Real norm(const Real & A);
};
```

The class contains some important parameters such as $numI$ and $numJ$ that outline the dimensions of the type. In other words every type can be seen as a sort of matrix: the scalar type has one row $numI = 1$ and one column $numJ = 1$. For instance the type $point3d$ has three rows $numI = 3$ and one column $numJ = 1$ and $tensor2d$ has two rows $numI = 2$ and two columns $numJ = 2$. It is also associated a flag $myType = typeReal$ that outlines the progressive $enum$ flag of the class.
The $traitsBasic$ has one constructor and several methods:

- $getZero$: returns the null element such that all its components are equal to zero;
- $getUnity$: returns an element whose entries are equal to one;
- $getUnityIJ$: returns a null element whose i-th and j-th entry is equal to one;

- $getIJ$: given an instance V of the class the method gets the component corresponding to the i-th row and j-th column;
- $setIJ$: sets the component at the i-th row and j-th column to the value specified by V;
- $setZero$: given an instance of the class it sets all its components to zero;
- $norm$: it computes the magnitude of the dof. For the $Real$ type it is equivalent to the absolute value, for the vectors, such as $point3d$, is the length and for the tensors it is the two norm, see [5].

The $traitsMpiOptimization.hpp$ contains a definition of all the $dofs$ $static$ classes whose send and receive process of mpi can be optimized:

```
namespace boost
{
namespace mpi
{
template<>
struct is_mpi_datatype<point3d> : mpl::true_{ };

template<>
struct is_mpi_datatype<point2d> : mpl::true_{ };

template<>
struct is_mpi_datatype<tensor3d> : mpl::true_{ };

template<>
struct is_mpi_datatype<tensor2d> : mpl::true_{ };

template<>
struct is_mpi_datatype<komplex> : mpl::true_{ };

template<>
template<size_t N>
struct is_mpi_datatype<staticVector<N> > : mpl::true_{ };
}
}
```

Finally we comment the $traitsMultiply.hpp$ class that enables the multiplication of different $dofs$. By now three different types of multiplications have been implemented:

- *direct product* : the product without index saturation, see [6]. We will describe this kind of product in detail just below;
- *vector product* : the classical vector product either in two or three dimensions;
- *scalar product* : the scalar product.

The notes on the $traitsMultiply.hpp$ file give an outline of what kind of multiplications are allowed:

```
<li> Real         * Real         = Real
<li> Real         * point2d      = point2d
<li> Real         * point3d      = point3d
<li> Real         * tensor2d     = tensor2d
<li> Real         * tensor3d     = tensor3d
```

```
<li> Real         * stateVector  = stateVector
<li> Real         * stateMatrix  = stateMatrix
<li> Real         * staticVector = staticVector
<li> Real         * komplex      = komplex
<li> point2d      * Real         = point2d
<li> point2d      * point2d      = tensor2d
<li> point3d      * Real         = point3d
<li> point3d      * point3d      = tensor3d
<li> tensor2d     * Real         = tensor2d
<li> tensor3d     * Real         = tensor3d
<li> stateVector  * Real         = stateVector
<li> stateMatrix  * Real         = stateMatrix
<li> staticVector * Real         = staticVector
<li> komplex      * Real         = komplex
<li> komplex      * komplex      = komplex
```

For instance the product of two points gives rise to a tensor: the point has only one index, so the multiplication of two points gives an object with two indexes, i.e. a tensor. This function is very useful when implementing the $morganaFiniteElement$ package as the code can automatically handle fields defined by different species of finite elements and $dofs$. We will describe in detail this in the $morganaFiniteElement$ manual. Then the other two types of products are defined as

```
Cross product A
<ol>
<li> point2d  * point2d  = point3d  (E_{i j k} a_j     b_k     )
<li> point3d  * point3d  = point3d  (E_{i j k} a_j     b_k     )
</ol>

Scalar product A
<ol>
<li> Real      * Real      = Real
<li> point2d   * point2d   = Real   ( a_k     b_k)
<li> point3d   * point3d   = Real   ( a_k     b_k)
</ol>
```

where E_{ijk} is the Ricci tensor see [6]. The $traitsMultiply$ class has two templates and, depending on the couple of $dofs$ considered, each couple has its own specialization, let us take a look to an example:

```
template<> class traitsMultiply<Real,Real>
{
public:
typedef Real DIRECTTYPE;
typedef Real SCALARTYPEA;

traitsMultiply() { };
static DIRECTTYPE  multiply(const Real & A, const Real & B);
static SCALARTYPEA scalarProductA(const Real & A, const Real & B);
};
```

In this case the class defines only the direct product with the function $multiply$ and the scalar product with the function $scalarProductA$. Each function returns a particular type that is either $DIRECTTYPE$ or $SCALARTYPEA$ defined just above with a couple of typedefs directives. In this case they are both equal to the $Real$ type.

2.5 Hints about the expression template

In this section we would like to give a rough idea of the methods we have used to implement the algebraic operations using the expression template mechanism. **This part of the manual it is not strictly necessary to use the**

functionalities of $Morgana$, so the reader can skip this paragraph. The expression template is an advanced programming technique of $C++$ capable of transferring to the compiler an amount of the tasks that are normally performed at run time. This technique, if properly used, may fundamentally reduce the computational cost at the expense of a slightly longer compilation time. We also stress that the time needed to compile the code can be reduced using the $-j$ option of *make* which exploits the possibility of using more than one core to get the work done. The expression template technique is rather involved so we underline that there is a number of examples on-line, see, for instance, [1].
We would like to compute the expression $point3d\ P_3 = P_1 + P_2$ in an efficient manner: the operator overloading technique, see [2], does not provide an efficient way to do so. In this latter case we would have defined the operator $operator + (const\ point3d\ P)$ in the class $point3d$. The compiler would have found that the instance P_1 of the class $point3d$ implements the + operator. So the P_2 point would have passed to that operator and the function $operator + (const\ point3d\ P)$ would have returned a temporary object whose existence is not directly clear from the code. This temporary object represents the sum $P_1 + P_2$ and would have been passed to the copy constructor of the class P_3 that would have been initialized with components equal to the sum $P_1 + P_2$. This technique is effective and correct although it creates and destroys a temporary instance of the class and this is particularly un-effective due to the fact that this operation might be repeated several times along the code.
To avoid the creation of temporary objects we have defined some proper sum and subtraction functions just above the class $point3d$. As the compiler does not find any overloading of the + operator in the $point3d$ class it looks for a function that is found in the file $typesInterface.hpp$:

```
//The sum operator
template<typename LHS, typename RHS>
typename plusTrait<LHS,RHS,LHS::myType>::EXPRESSION
operator+(const LHS  left, const RHS  right)
{
typedef typename plusTrait<LHS,RHS,LHS::myType>::EXPRESSION  EXPRESSION;
assert(staticAssert<LHS::myType == RHS::myType>::returnValue);
return(EXPRESSION(left,right));
}
```

The compiler then tries to adapt this function to the types involved so it sets $LHS = point3d$ and $RHS = point3d$. Therefore the function described above reduces to

```
//The sum operator
typename plusTrait<point3d,point3d,point3d::myType>::EXPRESSION
operator+(const point3d left, const point3d right)
{
typedef plusTrait<point3d,point3d,point3d::myType>::EXPRESSION
EXPRESSION;

assert(staticAssert<point3d::myType ==
point3d::myType>::returnValue);

return(EXPRESSION(left,right));
}
```

The constant $point3d :: myType$ is an *enum* integer defined in $morganaTypes$ therefore the *assert* functions checks statically that we are going to sum two equivalent types, in this case two $point3d$. The function $plusTrait < point3d, point3d, point3d :: myType > (left, right)$ returns $return(EXPRESSION(left, right))$, therefore it looks for this latter function and this is found in the file $typesInterface.hpp$:

```
//Point3d trait
```

```
template<typename LHS, typename RHS>
class plusTrait<LHS,RHS,typePoint3d>
{
public:
typedef expressionPoint3d<LHS,RHS,plusPoint3d<LHS,RHS> >
EXPRESSION;
};
```

Among the many $plusTrait$ items defined in $typesInterface.hpp$ this latter function satisfies $point3d :: myType == typePoint3d$. Moreover this function implies that $EXPRESSION = expressionPoint3d < point3d, point3d, plusPoint3d < point3d, point3 >>$ and this equivalence is substituted in the plus operator that now reads as:

```
//The sum operator
typename expressionPoint3d<point3d,point3d,plusPoint3d<point3d,point3> >
operator+(const point3d left, const point3d right)
{
return(expressionPoint3d<point3d,point3d,plusPoint3d<point3d,point3d> >
(left,right));
}
```

In other words, the operator + gets two points P_1 and P_2 and returns an instance of $expressionPoint3d$ that, in turn, receives two points and the type of the operation to be performed. Now the compiler looks for $expressionPoint3d < point3d, point3d\ , plusPoint3d < point3d, point3d >>$ and it finds it in the $point3d.h$ file:

```
template<typename LHS, typename RHS, typename OP>
struct expressionPoint3d
{
static const morganaTypes myType = typePoint3d;

const LHS & left;
const RHS & right;

expressionPoint3d(const LHS & Left, const RHS & Right) :
left(Left), right(Right) { }

Real getX() const { return(OP::applyX(left,right)); }
Real getY() const { return(OP::applyY(left,right)); }
Real getZ() const { return(OP::applyZ(left,right)); }
};
```

where $LHS = point3d$, $RHS = point3d$ and $OP = plusPoint3d < point3d, point3d >$. This latter operator is defined just above in the $point3d.h$ header:

```
template<typename LHS, typename RHS>
struct plusPoint3d
{
static Real applyX(const LHS & left, const RHS & right)
{ return(left.getX() + right.getX()); }

static Real applyY(const LHS & left, const RHS & right)
{ return(left.getY() + right.getY()); }

static Real applyZ(const LHS & left, const RHS & right)
{ return(left.getZ() + right.getZ()); }
};
```

Therefore just substituting backward we have:

```
struct expressionPoint3d<point3d,point3d,plusPoint3d<point3d,point3d> >
{
static const morganaTypes myType = typePoint3d;

const point3d & left;
```

```
const point3d & right;

expressionPoint3d(const point3d Left, const point3d Right) :
left(Left), right(Right) { }

Real getX() const { return(left.getX() + right.getX()); }
Real getY() const { return(left.getY() + right.getY()); }
Real getZ() const { return(left.getZ() + right.getZ()); }
};
```

To sum up: the operator + gets two points P_1 and P_2 returning a class $expressionPoint3d < point3d, point3d, plusPoint3d < point3d, point3d >>$. This latter class has some methods capable of performing the sum of the point components. The work is almost completed: the compiler now looks for an implementation of the operator = and it finds it directly in the class *point3d*. In fact:

```
template<typename LHS, typename RHS, typename OP>
void
point3d::
operator=(const expressionPoint3d<LHS,RHS,OP> & Expression)
{
X[0] = Expression.getX();
X[1] = Expression.getY();
X[2] = Expression.getZ();
}
```

where $LHS = point3d$, $RHS = point3d$ and $OP = plusPoint3d < point3d, point3d >$. Since the functions $applyX$, $applyY$ and $applyZ$ in $plusPoint3d$ are static, no temporary instance of the expression class $expressionPoint3d$ is created: the components are summed without any major run time overhead.
In spite of the fact that the expression template mechanism may seems convoluted and non-intuitive it is extremely computationally effective. In the file $performaBench2.cpp$, we have tested the performances of our expression template system considering some very simple algebraic operations: i.e $P = (P+Q)/2-(P*2)$. The same operations are both performed using the expression system and by hand-computing them in the code operating on the point components. These operations are performed one billion times and the time is recorded using a standard $core-2$ laptop. The hand-coded version is the quickest with times lasting from 4 to 6 seconds. The expression template version requires some more seconds with runs lasting from 9 to 10 seconds. We have also implemented (this version is not available), for testing purposes, also the relevant algebraic operators directly in the class *point3d* by using the operator overloading technique: in this case the computational burden has risen to $67-78$ seconds.
In other words the expression template mechanism provides a good balance between computational and coding performances.

2.6 Hints for a new dof

We give also a few hints to create new $dofs$, although the one already implemented covers almost all the current modeling needs we have encountered. Here we use as the type name the $newType$ name:

- create in the file *morganaTypes.hpp* a new *enum* identifier for the type, i.e., for instance, *typeNewType*;
- create the classes *newType.h* and *newType.cpp*, the makefile will detect the presence of a new *.cpp* file and will automatically build the corresponding object (*.o*) file;
- add the row $friend\ class\ boost :: serialization :: access$; in the definition of the class and implement the function:

```
template<class ARK>
void serialize(ARK & ar, const unsigned int version);
```

- test the correctness of the implementation just passing an instance of the class between two processes. You can simply modify the file *point3dTest2.cpp* in the *tests* folder;
- implement the *set* and *get* functions to access all the data in the new *dof*;
- implement the functions $* =$, $/ =$, $+ =$ and $- =$;
- implement the operator $=$;
- implement the expression templates for the operations $+$, $-$, $*$ and $/$ in the *newType.h* file;
- implement the operator $=$ that gets, as an argument the expression template;
- add in the file *typesInterface.hpp* for each operation the new specialization for *newType* and test the correctness of the implementation;
- implement the operators $<$ and $! =$. Be careful that they must satisfy the rules discussed above. If necessary consider geometric tolerances;
- implement the *cout* operator;
- modify the file *traitsBasic.hpp* adding the relevant feature of *newType*;
- if the type is static just add it to the list in *traitsMpiOptimization.hpp* file;
- modify the *traitsMultiply.hpp* file considering all the multiplications that make sense with *newType*.

2.7 Tutorials

The code has some examples in the folder $tests/morganaDofs$, in particular:

- $dofsTest1.cpp$ checks some simple algebraic operations and the ordering of many $dofs$;
- $komplexTest1.cpp$ checks some simple algebraic operations and the ordering of the $komplex$ class which implements the complex numbers;
- $komplexTest2.cpp$ is an example of exchanging two complex numbers between two processes;
- $performaBench1.cpp$ and $performaBench2.cpp$ evaluate the performances of the expression template system;
- $point3dTest1.cpp$ checks some simple algebraic operations and the ordering of the $point3d$ class;
- $point3dTest2.cpp$ and $point3dTest3.cpp$ are examples of exchanging two $point3d$ between two processes;
- $point3dTest4.cpp$ checks a particular function of $point3d$;
- $stateMatrixTest2.cpp$ checks some simple algebraic operations and the ordering of the $stateMatrix$ class;
- $stateVectorTest1.cpp$ checks some simple algebraic operations and the ordering of the $stateVector$ class;
- $stateVectorTest2.cpp$ is an example of exchanging two $stateVector$ between two processes;
- $staticVectorTest1.cpp$ checks some simple algebraic operations and the ordering of the $staticVector$ class;
- $staticVectorTest2.cpp$ is an example of exchanging two $staticVector$ between two processes;
- $tensor3dTest2.cpp$ checks some simple algebraic operations and the ordering of the $tensor3d$ class;
- $traitsBasicTest1.cpp$ checks the correctness of the $traitsBasis$ class;
- $traitsMultiplyTest1.cpp$ checks the correctness of the $traitsMultiply$ class.

References

[1] Advanced C++ lessons: http://aszt.inf.elte.hu/ gsd/halado cpp/.

[2] Cplusplus reference: http://www.cplusplus.com/doc/tutorial/.

[3] Stl reference: http://www.sgi.com/tech/stl/.

[4] Trilinos project: http://trilinos.sandia.gov/.

[5] A. Quarteroni. *Matematica Numerica*. Springer, 2008.

[6] J. Synge and A. Schild. *Tensor Calculus*. Courier Dover Publications, 1878.