Polynomials

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 Introduction

The evaluation of polynomials is a fundamental aspect of a finite element code since this package has a major impact on the efficiency of the algorithm. In fact polynomials define both the finite element fields and the geometrical transformations that map the actual geometrical elements to the reference ones. Therefore here we seek to implement an efficient way to evaluate polynomials and their derivatives.
Morgana has two main polynomial evaluation systems: a *static* one and a *dynamic* one. The first one has better performances while the second one is more flexible. The *static* evaluation system uses extensively the expression template technique and template recursions: for an introduction regarding these techniques see [1]. From a user perspective the package is rather simple. There are manly two interface classes: the *polyStatic* one and the *polyDynamic* one. They are very similar and they provide the access to all the evaluation functions of the *static* and *dynamic* branch respectively. Polynomials and their derivatives are evaluated in a given point $\vec{y} = (x, y, z)$ of the space, in general the polynomials are thought to be dependent upon three spatial coordinates.
The main difference between the *static* and *dynamic* implementation lies in the way the class is initialized with respect to the polynomial to be used. In the *static* case the class is template with respect to the type of polynomial. In the file *polyCards.h* all types of polynomial implemented can be found and in section 3 we will provide an outline of all the possibilities. The static polynomial types have the following syntax: $baseName|degree\_dymension\_progrLetter$. For instance $P0\_1d\_A$ specifies that this polynomial is of type $P0$ (it denotes the polynomials of degree at most 0) in one dimension. The letter A is a progressive letter that specifies the various polynomials considered: $P0$ polynomials in the interval $[0, 1]$ have only one base, for this reason there is only one polynomial

indicated with the letter A (there is no other polynomial indicated as $P0\_1d\_B$). If we consider the $P1$ basis then there exist two polynomials named $P1\_1d\_A$ and $P1\_1d\_B$: the first one corresponds to the polynomial $1 - x$ and the second one to x. Every element in the $P1$ space in the interval $[0, 1]$ can be expressed as the linear combination of these two bases, see [3].
Several other flags *baseName* are used in *morgana*:

- Pr represents the polynomials of degree at most r, in one, two or three dimensions. The number r defines the degree of the polynomial (For instance $P0$, $P1$, etc). They are mainly used to represent finite element functions on simplex elements such as triangles and tetrahedrons;
- Qr in two or three dimensions represents the polynomial of degree at most r in each coordinate (for instance $Q0$, $Q1$, etc). They are used to define finite element fields on quadrilateral or hexahedral elements;
- D dual grid finite elements are used only for some specific implementations of the finite volume method;
- DG is the discontinuous Galarking finite element basis;
- RT i the Raviart-Thomas finite element basis, see [2];
- $P1BP1$ are the linear $3d$ finite element fields with a piece-wise linear bubble, also known as mini-elements;
- CR is the Crouzier-Raviart finite element basis.

The dynamic polynomials, on the contrary, can be defined at run time and are, currently, much less used in *morgana*. It is sufficient to provide an instance of the class *polyDynamicCard* which provides the c_k, Px_k, Py_k and Pz_k coefficients so that the polynomial can be defined as:

$$\sum_{k=1}^{N_p} c_k x^{Px_k} y^{Py_k} z^{Pz_k}. \tag{1}$$

where N_p is the number of monomials necessary to define the polynomial.
Let us now give an overview of the classes of this package: the definitions of the static polynomial are contained in the *polyCards.h* and *polyCards.cpp* files, *polyPow.hpp* provides most of the static template recursion algorithms to evaluate the polynomial and *polyStatic.hpp* is the final interface. The dynamic polynomial are based on the *polyDynamicSubCard.h*, which defines a monomial. The collection of several monomials defines a polynomial *polyDynamicCard.h* and the interface is *polyDynamic.h*. Finally there are also some classes devoted to the implementation of the Gauss-Lobatto polynomials, see *glRule.h* and *glBase.h*.

2 Definition of static polynomials

In this section we will take a look inside the *polyCards.h* and *polyCards.cpp* files where all the static polynomials are defined. **Some other polynomials can be added in principle but this can be not very straightforward if polynomials have many terms. We also stress that the use of a recursive template mechanism to efficiently evaluate polynomials generates a significant burden in the compilation phase. It is not practical to evaluate polynomials in degree higher than 2.**
We now consider a line of *polyCards.h* which defines one dimensional linear finite elements:

```
/*! Polynomial P1 - 1D */
template<Int N> struct trunk_P1_1d_A;

template<>      struct trunk_P1_1d_A<1>
{static SInt Px = 0; static SInt Py = 0; static SInt Pz = 0;
static SReal C =  1.0; };

template<>      struct trunk_P1_1d_A<2>
{typedef trunk_P1_1d_A<1> SON;
static SInt Px = 1; static SInt Py = 0; static SInt Pz = 0;
static SReal C = -1.0; };

struct P1_1d_A {static const UInt S = 2;
typedef trunk_P1_1d_A<2> ROOT; };
```

This represents the definition of the first basis of the polynomial $\mathbb{P}^1$ on the interval $[0, 1]$ i.e. it represents the function $1-x$. This polynomial is divided into the sum of two monomials that are 1 and $-x$. Each of these is represented in its canonical form (1) by specifying its coefficients. In this case we have $N_p = 2$, the first term 1 is defined by $c_1 = 1$, $Px_1 = 0$, $Py_1 = 0$, $Pz_1 = 0$, while the second one, $-x$, is associated to $c_2 = -1$, $Px_2 = 1$, $Py_2 = 0$, $Pz_2 = 0$. The first row of the above mentioned code i.e.

```
template<Int N> struct trunk_P1_1d_A;
```

is a $forward\ declaration$. The template parameter $N = 1, 2$ identifies the monomials of the polynomial, in this case we have two: 1 and $-x$. The subsequent row contains the definition of each monomial

```
template<>      struct trunk_P1_1d_A<1>
{static SInt Px = 0; static SInt Py = 0; static SInt Pz = 0;
static SReal C =  1.0; };
```

the part of the code

```
trunk_P1_1d_A<1>
```

means that we are specifying the first monomial i.e. 1. The code fragment

```
static SInt Px = 0; static SInt Py = 0; static SInt Pz = 0
```

contains the values of the coefficients Px, Py, Pz. And, finally the corresponding code fragment

```
static SReal C =  1.0;
```

in the file *polyCards.cpp* contains the c term. The second row

```
template<> struct trunk_P1_1d_A<2>
{typedef trunk_P1_1d_A<1> SON; static SInt Px = 1; static SInt Py = 0; static SInt Pz = 0; static SReal C; };
```

contains all the information relative to the second monomial, i.e. $-x$. The structure is quite similar except for the fact that the term

```
typedef trunk_P1_1d_A<1> SON;
```

has been added. This recalls the previous template specialization of the class. In other words every term recalls the previous so that, starting from the last one, it is possible to go up in the list. This is the main tool used to get a recursive evaluation of monomials. Then we pass to the last row:

```
struct P1_1d_A {static const UInt S = 2;
typedef trunk_P1_1d_A<2> ROOT; };
```

which contains the main information of the polynomial, in particular

```
static const UInt S = 2
```

states that the polynomial has two terms and

```
typedef trunk_P1_1d_A<2> ROOT
```

defines the last monomial.

3 List of static polynomials

Several different polynomial bases are present in *morgana* here we introduce a brief list:

- $P0\_1d$ (1 basis) : Lagrangian basis on the interval $[0, 1]$ of degree 0. Basis flag $P0\_1d\_A$;
- $P1\_1d$ (2 bases) : Lagrangian basis on the interval $[0, 1]$ of degree 1. Basis flags $P1\_1d\_A$, $P1\_1d\_B$;
- $P2\_1d$ (3 bases) : Lagrangian basis on the interval $[0, 1]$ of degree 2. Basis flags $P2\_1d\_A$, $P2\_1d\_B$, $P2\_1d\_C$;
- $D0\_1d$ (1 basis) : piece-wise constant polynomial on the dual grid of the interval $[0, 1]$, Basis flag $D0\_1d\_A$;
- $DG0\_1d$ (1 basis) : discontinuous Galerkin piece-wise constant basis on the interval $[0, 1]$. It is equal to $P0\_1d$. Basis flag $DG0\_1d\_A$;
- $DG1\_1d$ (2 bases) : Legendre hierarchical basis on the interval $[0, 1]$. Basis flags $DG1\_1d\_A$, $DG1\_1d\_B$;
- $P0\_2d$ (1 basis) : Lagrangian basis on the reference triangle of degree 0. Basis flag $P0\_2d\_A$;
- $P1\_2d$ (3 bases) : Lagrangian basis on the reference triangle of degree 1. Basis flags $P1\_2d\_A$, $P1\_2d\_B$, $P1\_2d\_C$;
- $P2\_2d$ (6 bases) : Lagrangian basis on the reference triangle of degree 2. Basis flags $P2\_2d\_A$, $P2\_2d\_B$, $P2\_2d\_C$, $P2\_2d\_D$, $P2\_2d\_E$, $P2\_2d\_F$;

- $Q0\_2d$ (1 basis) : Lagrangian basis on the reference triangle of degree 0. Basis flag $Q0\_2d\_A$;
- $Q1\_2d$ (4 bases) : Lagrangian basis on the reference triangle of degree 1 in each direction. Basis flags $Q1\_2d\_A$, $Q1\_2d\_B$, $Q1\_2d\_C$, $Q1\_2d\_D$;
- $Q2\_2d$ (9 bases) : Lagrangian basis on the reference triangle of degree 2 in each direction. Basis flags $Q2\_2d\_A$, $Q2\_2d\_B$, $Q2\_2d\_C$, $Q2\_2d\_D$, $Q2\_2d\_E$, $Q2\_2d\_F$, $Q2\_2d\_G$, $Q2\_2d\_H$, $Q2\_2d\_I$;
- $D0\_2d$ (3 bases) : piece-wise constant polynomial on the dual grid of the reference triangle. Basis flags $D0\_2d\_A$, $D0\_2d\_B$, $D0\_2d\_C$;
- $RT0\_2d$ ($3x3$ bases) : Raviart-Thomas basis, three functions each defined by three components. Basis flags $RT0\_2d\_Ax$, $RT0\_2d\_Ay$, $RT0\_2d\_Az$, $RT0\_2d\_Bx$, $RT0\_2d\_By$, $RT0\_2d\_Bz$, $RT0\_2d\_Cx$, $RT0\_2d\_Cy$, $RT0\_2d\_Cz$;
- $DG0\_2d$ (1 basis) : discontinuous Galerkin piece-wise constant basis on the reference triangle. Basis flag $DG0\_2d\_A$;
- $DG1\_2d$ (3 bases) : Legendre hierarchical basis on the reference triangle. Basis flags $DG1\_2d\_A$, $DG1\_2d\_B$, $DG1\_2d\_C$;
- $P0\_3d$ (1 basis) : Lagrangian basis on the reference tetrahedron of degree 0. Basis flag $P0\_3d\_A$;
- $P1\_3d$ (4 bases) : Lagrangian basis on the reference tetrahedron of degree 1. Basis flags $P1\_3d\_A$, $P1\_3d\_B$, $P1\_3d\_C$, $P1\_3d\_D$;
- $P2\_3d$ (9 bases) : Lagrangian basis on the reference tetrahedron of degree 2. Basis flags $P2\_3d\_A$, $P2\_3d\_B$, $P2\_3d\_C$, $P2\_3d\_D$, $P2\_3d\_E$, $P2\_3d\_F$, $P2\_3d\_G$, $P2\_3d\_H$, $P2\_3d\_I$, $P2\_3d\_L$;
- $Q0\_3d$ (1 basis) : Lagrangian basis on the reference tetrahedron of degree 0. Basis flag $Q0\_3d\_A$;
- $Q1\_3d$ (8 bases) : Lagrangian basis on the reference tetrahedron of degree 1 in each direction. Basis flags $Q1\_3d\_A$, $Q1\_3d\_B$, $Q1\_3d\_C$, $Q1\_3d\_D$, $Q1\_3d\_E$, $Q1\_3d\_F$, $Q1\_3d\_G$, $Q1\_3d\_H$;
- $Q2\_3d$ (27 bases) : Lagrangian basis on the reference tetrahedron of degree 2 in each direction. Basis flags $Q2\_3d\_AA$, $Q2\_3d\_AB$, $Q2\_3d\_AC$, $Q2\_3d\_AD$, $Q2\_3d\_AE$, $Q2\_3d\_AF$, $Q2\_3d\_AG$, $Q2\_3d\_AH$, $Q2\_3d\_AI$, $Q2\_3d\_BA$, $Q2\_3d\_BB$, $Q2\_3d\_BC$, $Q2\_3d\_BD$, $Q2\_3d\_BE$, $Q2\_3d\_BF$, $Q2\_3d\_BG$, $Q2\_3d\_BH$, $Q2\_3d\_BI$, $Q2\_3d\_CA$, $Q2\_3d\_CB$, $Q2\_3d\_CC$, $Q2\_3d\_CD$, $Q2\_3d\_CE$, $Q2\_3d\_CF$, $Q2\_3d\_CG$, $Q2\_3d\_CH$, $Q2\_3d\_CI$;
- $D0\_3d$ (4 bases) : piece-wise constant polynomial on the dual grid of the reference tetrahedron. Basis flags $D0\_3d\_A$, $D0\_3d\_B$, $D0\_3d\_C$, $D0\_3d\_D$;

- $P1BP1\_3d$ ($4+4$ bases) : $P1$ basis functions plus a piece-wise $P1$ bubble, also known as mini-elements. Basis flags $P1BP1\_3d\_A$, $P1BP1\_3d\_B$, $P1BP1\_3d\_C$, $P1BP1\_3d\_D$, $P1BP1\_3d\_E$, $P1BP1\_3d\_F$, $P1BP1\_3d\_G$, $P1BP1\_3d\_H$;

- $RT0\_3d$ ($4x3$ bases) : Raviart-Thomas basis, four functions each defined by three components. Basis flags $RT0\_3d\_Ax$, $RT0\_3d\_Ay$, $RT0\_3d\_Az$, $RT0\_3d\_Bx$, $RT0\_3d\_By$, $RT0\_3d\_Bz$, $RT0\_3d\_Cx$, $RT0\_3d\_Cy$, $RT0\_3d\_Cz$, $RT0\_3d\_Dx$, $RT0\_3d\_Dy$, $RT0\_3d\_Dz$;

- $CR1\_3d$ (4 bases) : Cruizer-Raviarth basis. Basis flags $CR1\_3d\_A$, $CR1\_3d\_B$, $CR1\_3d\_C$, $CR1\_3d\_D$.

4 Static evaluation of polynomials

In this section we describe a recursive template mechanism to evaluate statically the polynomials. **This section can be skipped since it describes the internal evaluation mechanism.** The evaluation mechanism is composed by a rather large set of small classes (see *polyPow.hpp*): they implement a recursive iteration scheme based on an integer defined as N. The recursion scheme is broken by a template specialization for $N = 1$. The first class we are going to describe computes the factorial of a given integer number:

```
template<UInt D>
struct polyFactorial
{
static Real eval(const UInt & N)
{return(polyFactorial<D-1>::eval(N) * (N - D + 1) ); }
};

template<>
struct polyFactorial<0>
{
static Real eval(const UInt & N)
{
assert(N == N);
return(1.0);
}
};
```

The mechanism is rather simple: let us assume, for instance, that the function $polyFactorial < 2 >:: eval(2)$ is called. This function computes the factorial of 2. This function evaluates $polyFactorial < 1 >:: eval(N) * 1$ and calls $polyFactorial < 1 >:: eval(N)$, thus we get $polyFactorial < 0 >:: eval(N)*(2)*1$. Since there exist a template specialization $polyFactorial < 0 >$ we get that the chain is terminated and we get the final result i.e. 2.
The factorial evaluation function is used to compute the derivatives. In fact the k-th derivative of x^n is equal to

$$\frac{d^k x^n}{dx^k} = n(n-1)\cdots(n-k)x^{n-k} \tag{2}$$

if $k \leq n$ and 0 otherwise. The term $n(n-1)\cdots(n-k)$ represents the first k terms of the factorial of a number. In the same way it is possible to create

some recursions to evaluate the powers of a real number, for instance x^n. This is achieved by the *polyPow* class

```
template<UInt N>
struct polyPow
{
static Real eval(const Real & x)
{ return(polyPow<N-1>::eval(x) * x); }
};
```

We have also implemented a static switch function that returns different values in function of the templates used. In particular the *polySwitch* class returns the n-th power of x if a given logic flag is true, otherwise it returns zero:

```
template<bool,Int N> class polySwitch;

template<Int N>
struct polySwitch<true,N>
{
static Real eval(const Real & x)
{ return(polyPow<N>::eval(x)); }
};

template<Int N>
struct polySwitch<false,N>
{
static Real eval(const Real & x)
{ return(0.0); }
};
```

This class is used along with *polyPosPow*

```
template<Int N>
struct polyPosPow
{
static Real eval(const Real & x)
{ return(polySwitch< N>=0 , N>::eval(x)); }
}
```

This latter class returns the power of a number only if the integer of the power is positive, otherwise it returns zero. This is very useful when the derivatives of polynomials are evaluated, in fact the derivative of x^n, with $n \in \mathbb{N}$, is equal to nx^{n-1} only if $n > 0$, otherwise it vanishes.

```
template<typename POLYCARD, UInt S> class polyEvalIter;

template<typename POLYCARD, UInt S>
struct polyEvalIter
{
typedef typename POLYCARD::SON SON;

static Real eval(const point3d & P)
{
return(polyEvalIter<SON,S-1>::eval(P) +
POLYCARD::C *
polyPosPow<POLYCARD::Px>::eval(P.getX()) *
polyPosPow<POLYCARD::Py>::eval(P.getY()) *
polyPosPow<POLYCARD::Pz>::eval(P.getZ()) );
}
};

template<typename POLYCARD>
struct polyEvalIter<POLYCARD,1>
{
static Real eval(const point3d & P)
{
```

```
return(POLYCARD::C *
polyPosPow<POLYCARD::Px>::eval(P.getX()) *
polyPosPow<POLYCARD::Py>::eval(P.getY()) *
polyPosPow<POLYCARD::Pz>::eval(P.getZ()) );
}
};
```

The *PolyEvalIter* class is the heart of the static evaluation system: the class sets the integer S to the number of monomials in a given polynomial. For instance $1 - x$ has two monomials 1 and $-x$, therefore $S = 2$. The class calls itself by setting $S = S - 1$. The $S = 0$ specialization terminates the process. The evaluation of derivatives is performed with the *polyDerIter* class that is very similar to what we have already seen. The parameters $UInt\ dx$, $UInt\ dy$, $UInt\ dz$ define the derivative order with respect to the x, y and z coordinates.

```
template<typename POLYCARD, UInt dx, UInt dy, UInt dz, UInt S> class polyDerIter;

template<typename POLYCARD, UInt dx, UInt dy, UInt dz, UInt S>
struct polyDerIter
{
typedef typename POLYCARD::SON SON;

static Real eval(const point3d & P)
{
return(polyDerIter<SON,dx,dy,dz,S-1>::eval(P) +
POLYCARD::C *
polyFactorial<dx>::eval(UInt(POLYCARD::Px)) * polyPosPow<POLYCARD::Px - dx>::eval(P.getX()) *
polyFactorial<dy>::eval(UInt(POLYCARD::Py)) * polyPosPow<POLYCARD::Py - dy>::eval(P.getY()) *
polyFactorial<dz>::eval(UInt(POLYCARD::Pz)) * polyPosPow<POLYCARD::Pz - dz>::eval(P.getZ()) );
}
};

template<typename POLYCARD, UInt dx, UInt dy, UInt dz>
struct polyDerIter<POLYCARD,dx,dy,dz,1>
{
static Real eval(const point3d & P)
{
return(POLYCARD::C *
polyFactorial<dx>::eval(UInt(POLYCARD::Px)) * polyPosPow<POLYCARD::Px - dx>::eval(P.getX()) *
polyFactorial<dy>::eval(UInt(POLYCARD::Py)) * polyPosPow<POLYCARD::Py - dy>::eval(P.getY()) *
polyFactorial<dz>::eval(UInt(POLYCARD::Pz)) * polyPosPow<POLYCARD::Pz - dz>::eval(P.getZ()) );
}
};
```

5 Interface of static evaluation

All the static evaluation systems can be accessed through the *polyStatic* class which loads all the relevant information regarding the specific polynomial declaring the *POLY* type as listed in Section 3.

```
template<typename POLY>
class polyStatic
{
public:
typedef typename POLY::ROOT ROOT;

/*! @name Constructors */ //@{
public:
polyStatic();
//@}
```

The polynomial can be evaluated by the following two functions

```
/*! @name Evaluation functions */ //@{
```

```
public:
Real evaluate(const point3d & X) const;
static Real evaluateStatic(const point3d & X);
//@}
```

which call the *polyEvalIter* class. The evaluation of gradients is performed by the following set of functions

```
/*! @name Gradient evaluation functions */ //@{
public:
static Real evaluateGradientX(const point3d & X);
static Real evaluateGradientY(const point3d & X);
static Real evaluateGradientZ(const point3d & X);
static point3d evaluateGradient(const point3d & X);
//@}
```

which calls the *polyDerIter* class. All the components of the Hessian matrix can be computed by

```
/*! @name Second derivative evaluation functions */ //@{
public:
static Real evaluateHessianXX(const point3d & X);
static Real evaluateHessianXY(const point3d & X);
static Real evaluateHessianXZ(const point3d & X);
static Real evaluateHessianYX(const point3d & X);
static Real evaluateHessianYY(const point3d & X);
static Real evaluateHessianYZ(const point3d & X);
static Real evaluateHessianZX(const point3d & X);
static Real evaluateHessianZY(const point3d & X);
static Real evaluateHessianZZ(const point3d & X);
static tensor3d evaluateHessian(const point3d & X);
//@}
```

and it is also possible to evaluate a generic derivative as:

```
/*! @name Generic evaluation function */ //@{
public:
template<UInt dx, UInt dy, UInt dz>
static Real evaluateDerivative(const point3d & X);
//@}
```

Finally the total degree of the polynomial is computed by:

```
/*! @name Other functions */ //@{
public:
static UInt getDegree();
//@}

};
```

6 Interface of dynamic evaluation

The dynamic evaluation of polynomials is based on the standard *pow* function for the evaluation of the powers of monomials. The interface is contained in the file *polyDynamic.h*: it is almost similar to the static one discussed in Section 5 except for the fact that it is not template. The polynomial is specified using the function *void setPolyDynamicCard*(*const polyDynamicCard Card*):

```
#ifndef POLYDYNAMIC_H_
#define POLYDYNAMIC_H_

#include "polyDynamicCard.h"

/*! The dynamic evaluation class.
The polynomial type is given setting the \c polyDynamicCard
```

```
wich contains all the information that define the polynomial. */
class polyDynamic
{
/*! @name Internal data */ //@{
public:
bool cardLoaded;
polyDynamicCard card;
//@}

/*! @name Constructor */ //@{
public:
polyDynamic();
//@}

/*! @name Internal functions */ //@{
public:
void setPolyDynamicCard(const polyDynamicCard & Card);
UInt factorial(const UInt & d, const UInt & p) const;
//@}
```

The class *polyDynamicCard* allows you to define the various monomials using the functions *setSlot* and *addSlot*:

```
/*! The container class that storage all
the information to define a polynomial. */
class polyDynamicCard
{
/*! @name Internal data */ //@{
public:
sVect<polyDynamicSubCard> subCards;
//@}

/*! @name Constructors */ //@{
public:
polyDynamicCard();
polyDynamicCard(const polyDynamicCard & card);
polyDynamicCard operator=(const polyDynamicCard & card);
void operator*=(const polyDynamicCard & card);
void operator*=(const Real & val);
void reorder();
void simplify();
//@}

/*! @name Set and get functions */ //@{
public:
void addSlot(const UInt & cx, const UInt & cy,
const UInt & cz, const Real & c);
void setSlot(const UInt & i, const UInt & cx,
const UInt & cy, const UInt & cz, const Real & c);
UInt getCx(const UInt & i) const;
UInt getCy(const UInt & i) const;
UInt getCz(const UInt & i) const;
Real getCoeff(const UInt & i) const;
UInt getNumCoeff() const;
//@}

/*! @name Outstream operator */ //@{
public:
friend ostream & operator<<(ostream & f,
const polyDynamicCard & G);
//@}
};
```

Every single *slot* defines a set of coefficients c, cx, cy, cz that represent a single monomial: they are called c_k, Px_k, Py_k, Pz_k in equation (1). The functions $reorder()$ and $simplify()$ reorder and simplify the polynomial considered whenever this is possible. In particular the second function

- identifies the monomials that have equal coefficients and it sums them (for instance it can perform the following simplification $x + x = 2x$);
- eliminates all the null terms.

Once initialized it is possible to evaluate the polynomial using the following functions:

```
/*! @name Evaluate functions */ //@{
public:
Real evaluate(const point3d & X) const;
//@}

/*! @name Gradient evaluation functions */ //@{
public:
Real evaluateGradientX(const point3d & X) const;
Real evaluateGradientY(const point3d & X) const;
Real evaluateGradientZ(const point3d & X) const;
point3d evaluateGradient(const point3d & X) const;
//@}

/*! @name Second derivative evaluation functions */ //@{
public:
Real evaluateHessianXX(const point3d & X) const;
Real evaluateHessianXY(const point3d & X) const;
Real evaluateHessianXZ(const point3d & X) const;
Real evaluateHessianYX(const point3d & X) const;
Real evaluateHessianYY(const point3d & X) const;
Real evaluateHessianYZ(const point3d & X) const;
Real evaluateHessianZX(const point3d & X) const;
Real evaluateHessianZY(const point3d & X) const;
Real evaluateHessianZZ(const point3d & X) const;
tensor3d evaluateHessian(const point3d & X) const;
//@}

/*! @name Generic evaluation function */ //@{
public:
Real evaluateDerivative(const UInt & dx, const UInt & dy,
const UInt & dz, const point3d & X) const;
//@}

/*! @name Other functions */ //@{
public:
UInt getDegree() const;
//@}
};

#endif
```

that are very similar to the ones already seen in the *polyStatic* case.

7 Gauss-Lobatto support

The great versatility of dynamic polynomials can be exploited to create a wide variety of approximating polynomials. In particular it is possible to develop a number of classes that generate some proper *polyDynamicCard* instances. In this case, we discuss the generation of a set of Gauss-Lobatto (GL) polynomials:

```
/*! Spectral polynomial generator on structured meshes.
The parameters \c nx, \c ny and \c nz represent the polynomial degree.
The indices 1 < \c ix < nx + 1, 1 < \c iy < ny + 1 and
1 < \c iz < nz + 1 identify the base.  */
class glBase
```

```
{
public:
glBase();
polyDynamicCard getPolynomial(const UInt & nx,
const UInt & ny, const UInt & nz,
const UInt & ix, const UInt & iy, const UInt & iz);
};
```

This interface requires the degree in each spatial direction to generate a set of spectral bases. The indexes ix, iy and iz allow to extract the desired polynomial: some $(nx+1)(ny+1)(nz+1)$ polynomials are created and the indexes ix, iy, iz identify one of them. The GL polynomials are such that that they are equal to one in a GL node and they vanish in all the others GL nodes (see Figure 7 for a two dimensional example). The GL nodes are generated by

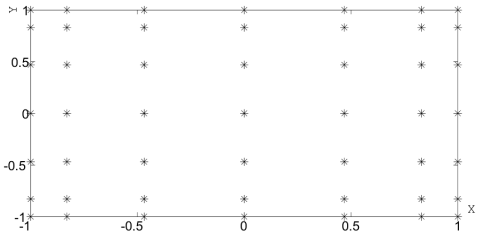

Figure 1: Representation of the Gauss-Lobatto nodes in two dimensions.

the class

```
/*! The generator of the Gauss-Lobatto nodes in one dimension.
The user provides the number of the nodes \c NN and the
algorithm generates both the nodes coordinates \c xgl
and the associated weights \c wgl. Since these data are obtained
numerically it is necessary to provide a convergence tolerance \c toll */
class glRule
{
public:
UInt N;

public:
glRule(const UInt & NN);
void compute(sVect<Real> & xgl, sVect<Real> & wgl,
const Real & toll) const;
};
```

that generates a list of nodes and integration weights in one dimension. The *toll* value is used to terminate the iterative loop used to generate the nodes.

8 Tutorials

In the */morgana/tests/morganaPolynomial* folder some tutorials can be found. The *polyStaticTestX.cpp* tutorials, with $X = 1, 2, 3, 4, 5$, check the correctness of some Lagrangian polynomials, the *staticEvaluationBench*1*.cpp* has been developed to get an idea of the computational time required by using several different techniques to compute the fourth power of a number. The files *polyDynamicCardTest*1*.cpp* and *dynamicEvaluation.cpp* test the correctness

of the dynamic polynomial evaluation system. Finally, the *glNodesTest*1.*cpp* and *glBaseTest*1.*cpp* check the implementation of the GL polynomials.

References

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

[2] A. Ern and J-L. Guermond. *Theory and practice of finite elements*. Pringer, 2004.

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