

SyFi Tutorial

Kent-Andre Mardal

January 9, 2006

Contents

1	Introduction	2
2	Software	3
2.1	License	3
2.2	Installation	4
2.3	Python Support	4
3	A Finite Element	5
3.1	Basic Concepts	5
3.1.1	Polygons	5
3.2	Polynomial Spaces	11
3.2.1	Bernstein Polynomials	12
3.3	A Finite Element	13
3.4	Degrees of Freedom	15
3.5	Some Examples of Finite Elements	18
3.5.1	The Lagrangian Element	18
3.5.2	The Crouizex-Raviart Element	19
3.5.3	The P_0 Element	20
3.5.4	The Discontinuous Lagrangian Element	20
4	Mixed Finite Elements	23
4.1	The Taylor–Hood and the $\mathbb{P}_n^d - \mathbb{P}_{n-2}$ Elements for the Stokes problem	23
4.2	The Mixed Crouizex-Raviart Element for the Stokes problem	24
4.3	The Mixed Raviart-Thomas Element for the Poisson Problem on Mixed Form	24
5	Computing Element Matrices	27
5.1	A Poisson Problem	27
5.2	A Poisson Problem on Mixed Form	30
5.3	A Stokes Problem	31
5.4	A Nonlinear Convection Diffusion Problem	32

1 Introduction

The finite element package SyFi is a C++ library built on top of the symbolic math library GiNaC (www.ginac.de). The name SyFi stands for Symbolic Finite elements. The package provides polygonal domains, polynomial spaces, and degrees of freedom as symbolic expressions that are easily manipulated. This makes it easy to define and use finite elements.

All the test examples described in this tutorial can be found in the directory `tests`. The reader is of course encouraged to run the examples along with the reading.

Before we start to describe SyFi, we need to briefly review the basic concepts in GiNaC. GiNaC is an open source C++ library for symbolic mathematics, which has a strong support for polynomials. The basic structure in GiNaC is an `ex`, which may contain either a number, a symbol, a function, a list of expressions, etc. (see `simple.cpp`):

```
ex pi = 3.14;
ex x = symbol("x");
ex f = cos(x);
ex list = lst(pi,x,f);
```

Hence, `ex` is a quite general class, and it is the cornerstone of GiNaC. It has a lot of functionality, for instance differentiation and integration (see `simple2.cpp`),

```
// initialization (f = x^2 + y^2)
ex f = x*x + y*y;

// differentiation (dfdx = df/dx = 2x)
ex dfdx = f.diff(x,1);

// integration (intf = 1/3*y^2, integral of f(x,y) from x=0 to x=1)
ex intf = integral(x,0,1,f);
```

GiNaC also has a structure for lists of expressions, `lst`, with the function `nops()` which returns the size of the list, and `operator [int i]` or the function `op(int i)` which returns the $(i + 1)$ 'th element¹.

We recommend the reader to glance through the GiNaC documentation before proceeding with this tutorial. GiNaC provides all the basic tools for manipulation of polynomials, such as differentiation and integration with respect to one variable. Our goal with the SyFi package is to extend GiNaC with higher level constructs such as differentiation with respect to several variables (e.g., ∇), integration of functions over polygonal domains, and polynomial spaces. All of which are basic ingredients in the finite element method.

Our motivation behind this project is threefold. First, we wish to make advanced finite element methods more readily available. We want to do this by implementing a variety of finite elements and functions for computing element matrices. Second, in our experience developing and debugging codes for finite element methods is hard. Having the basis functions and the weak form

¹It is standard C/C++ syntax that `operator [int i]` should return the $(i + 1)$ 'th element.

as symbolic expressions, and being able to manipulate them may be extremely valuable. For instance, being able to differentiate the weak form to compute the Jacobian in the case of a nonlinear PDE, eliminates a lot of handwriting and coding. Third, having the symbolic expressions it is reasonable that it is possible to generate efficient code by traversing the expressions as three structures (which GiNaC has support for). Alternatively, we can simply evaluate the basis functions, their derivatives, etc. in quadrature points and generate C++ code for these expressions. GiNaC has basic tools for code generation.

To try to motivate the reader, we also show an example where we compute the element matrix for the weak form of the Poisson equation, i.e.,

$$A_{ij} = \int_T \nabla N_i N_j dx.$$

We remark that the following example is an attempt to make an appetizer. All concepts will be carefully described during the tutorial.

```
void compute_element_matrix(Polygon& T, int order) {
  std::map<std::pair<int,int>, ex> A;           // matrix of expression
  std::pair<int,int> index;                    // index in matrix
  LagrangeFE fe;                              // Lagrangian element of any order
  fe.set(order);                              // set the order
  fe.set(domain);                             // set the polygon
  fe.compute_basis_functions();               // compute the basis functions
  for (int i=1; i<= fe.nbf() ; i++) {
    index.first = i;
    for (int j=1; j<= fe.nbf() ; j++) {
      index.second = j;
      ex nabla = inner(grad(fe.N(i)), grad(fe.N(j))); // compute the integrands
      ex Aij = T.integrate(nabla);                 // compute the weak form
      A[index] = Aij;                             // update element matrix
    }
  }
}
```

In the above example, everything is computed symbolically. Even the polygon may be an abstract polygon, e.g., specified as a triangle with vertices \mathbf{x}_0 , \mathbf{x}_1 , and \mathbf{x}_2 , where the vertices are symbols and not concrete points. Notice also, that we usually use STL containers to store our datastructure. This leads to the somewhat unfamiliar notation `A[index]` instead of `A[i,j]`.

Finally, we have to warn the reader: This project is still within its initial phase. Only a few elements have been implemented.

2 Software

2.1 License

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY-

ITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

In the case where the GNU licence does not fit your need. Contact the author at `kent-and@simula.no`.

2.2 Installation

Dependencies SyFi is a C++ library and therefore a C++ compiler is needed. At present the library has only been tested with the GNU C++ compiler. The `configure` script is a shell script made by the tools Automake and Autoconf. Hence, you can run this script with, e.g., the GNU Bourne-again shell. Finally, SyFi relies on the C++ library GiNaC.

Configuration and Installation As mention earlier, the configuration, build and installation scripts are all made by the Autoconf and Automake tools. Hence, to configure, build and install the package, simply execute the commands,

```
bash >./configure
bash >make
bash >make install
```

If this does not work, it is most likely because GiNaC is not properly installed on your system. Check if you have the script `ginac-config` in your path.

Reporting Bugs/Submitting Patches At present, there are no mailing-lists associated with this package. Therefore, all bug reports etc. should be directed directly to `kent-and@simula.no`.

In case, you want to contribute code, please create a patch with `diff`,

```
bash >diff -u --new-file --recursive SyFi SyFi-mod > SyFi-<name>-<date>.patch
```

Here `<name>` should be replaced with your name and `<date>` should be replaced with the current date.

2.3 Python Support

SyFi now comes with Python support. The Python module is made by using the tool SWIG (<http://www.swig.org>). In addition, one should also install Swiginac (<http://swiginac.berlios.de/>), which is a Python interface to GiNaC created by using SWIG. More about the usage of the Python interface can be found in Section 6.

3 A Finite Element

3.1 Basic Concepts

To keep the abstractions clear we briefly review the general definition of a finite element, see e.g., Brenner and Scott [1] or Ciarlet [3].

Definition 3.1 (A Finite Element) *A finite element consists of,*

1. *A polygonal domain, T .*
2. *A polynomial space, V .*
3. *A set of degrees of freedom (linear forms), $L_i : V \rightarrow \mathbb{R}$, for $i = 1, \dots, n$, where $n = \dim(V)$, that determines V uniquely.*

Furthermore, to determine a basis in V , $\{v_i\}_{i=1}^n$, we form the linear system of equations,

$$L_i(v_j) = \delta_{ij}, \quad (1)$$

and solve it.

Example 3.1 (Linear Lagrangian element on the reference triangle) *In this example we describe how the linear Lagrangian element is defined on the reference triangle. Let T be the unit triangle with vertices $(0, 0)$, $(1, 0)$, and $(0, 1)$. Furthermore, the polynomial space V consists of linear polynomials, i.e., polynomials on the form $a + bx + cy$. The degrees of freedom for a linear Lagrangian element are simply the pointvalues at the vertices, \mathbf{x}_i , $L_i(v_j) = v_j(\mathbf{x}_i)$. The degrees of freedom and (1) determined a_j , b_j , and c_j for each basis function v_j . For instance v_1 , which is on the form $a_1 + b_1x + c_1y$, is determined by,*

$$L_i(v_1) = v_1(\mathbf{x}_i) = \delta_{i1},$$

or written out as a system of linear equations,

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ b_1 \\ c_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Hence,

$$v_1 = 1 - x - y.$$

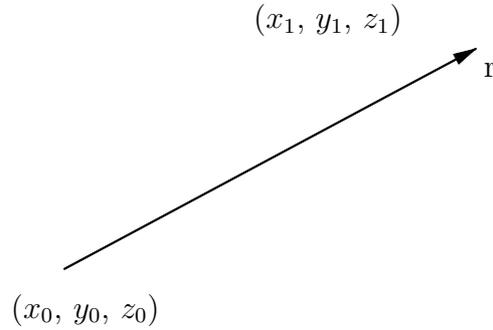
The functions v_2 and v_3 can be determined similarly.

In the next sections we will go detailed through polygons, polynomial spaces and degrees of freedom, and the corresponding software components.

3.1.1 Polygons

In the finite element method we need the concept of simple polygons to define integration, polynomial spaces etc. The basic polygons are lines, triangles, and tetrahedra (rectangles and boxes have not been implemented yet). These basic components will be briefly described in this section.

Figure 1: A line.



Line A line segment, L , between two points $\mathbf{x}_0 = [x_0, y_0, z_0]$ and $\mathbf{x}_1 = [x_1, y_1, z_1]$ in 3D is defined as, see also Figure 3.1.1,

$$x = x_0 + a t, \quad (2)$$

$$y = y_0 + b t, \quad (3)$$

$$z = z_0 + c t, \quad (4)$$

$$t \in [0, 1], \quad (5)$$

where $a = x_1 - x_0$, $b = y_1 - y_0$, and $c = z_1 - z_0$.

Integration of a function $f(x, y, z)$ along the line segment L is performed by substitution,

$$\int_L f(x, y, z) dx dy dz = \int_0^1 f(x(t), y(t), z(t)) D dt, \quad (6)$$

where $D = \sqrt{a^2 + b^2 + c^2}$.

Software Component: Line The class `Line` implements a general line. It is defined as follows (see `Polygon.h`):

```
class Line : public Polygon {
ex a_;
ex b_;
public:
Line() {}
Line(ex x0, ex x1, string subscript = ""); // x0_ and x1_ are points
~Line(){}

virtual int no_vertices();
virtual ex vertex(int i);
virtual ex repr(ex t);
virtual string str();
virtual ex integrate(ex f);
}
```

Most of the functions in this class are self-explanatory. However, the function `repr` deserves special attention. The function `repr` returns the mathematical definition of a line. To be precise, this function returns a list of expressions (`lst`), where the items are the items in (2)-(5) (see also the example below).

The basic usage of a line is as follows (see `line_ex1.cpp`),

```
ex p0 = lst(0.0,0.0,0.0);
ex p1 = lst(1.0,1.0,1.0);

Line line(p0,p1);

// show usage of repr
symbol t("t");
ex l_repr = line.repr(t);
cout <<"l_repr " <<l_repr<<endl;
EQUAL_OR_DIE(l_repr, "{x==t,y==t,z==t,{t,0,1}}");

for (int i=0; i< l_repr.nops(); i++) {
    cout <<"l_repr.op(" <<i<<"): " <<l_repr.op(i)<<endl;
}

// compute the integral of a function along the line
ex f = x*x + y*y*y + z;
ex intf = line.integrate(f);
cout <<"intf " <<intf<<endl;
EQUAL_OR_DIE(intf, "13/12");
```

The function `EQUAL_OR_DIE` compares the string representation of the expression with an expected expression represented as a character array. If the string representation of the expression and the character array are not equal the program dies, and this tells the programmer that the test faulted. The reason for the use of this function is that our test examples also serve as regression tests for the package.

Triangle A triangle is defined in terms of three points \mathbf{x}_0 , \mathbf{x}_1 , and \mathbf{x}_2 . Associated with each triangle are three lines; the first line is between the points \mathbf{x}_0 and \mathbf{x}_1 , the second line is between the points \mathbf{x}_0 and \mathbf{x}_2 , and the third line is between the points \mathbf{x}_1 and \mathbf{x}_2 . This is shown in Figure 2. The triangle can be represented as

$$x = x_0 + ar + bs, \quad (7)$$

$$y = y_0 + cr + ds, \quad (8)$$

$$z = z_0 + er + fs, \quad (9)$$

$$s \in [0, 1 - r], \quad (10)$$

$$r \in [0, 1], \quad (11)$$

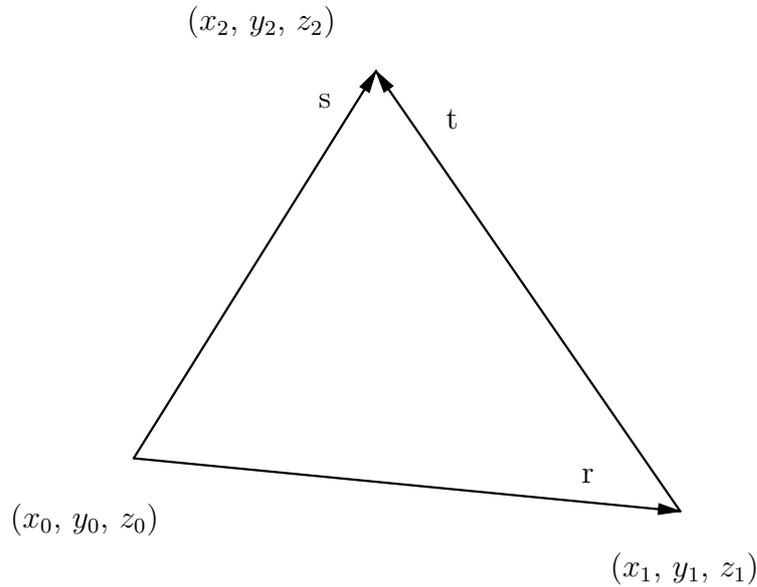
where $(a, c, e) = (x_1 - x_0, y_1 - y_0, z_1 - z_0)$ and $(b, d, f) = (x_2 - x_0, y_2 - y_0, z_2 - z_0)$.

Integration is performed by substitution,

$$\int_T f(x, y, z) dx dy dz = \int_0^1 \int_0^{1-r} f(x, y, z) D ds dr,$$

where $D = \sqrt{(cf - de)^2 + (af - be)^2 + (ad - bc)^2}$.

Figure 2: Triangle



Software Component: Triangle The class `Triangle` implements a general triangle. It is defined as follows (see `Polygon.h`):

```
class Triangle : public Polygon {
public:
    Triangle(ex x0, ex x1, ex x1, string subscript = "");
    Triangle() {}
    ~Triangle(){}

    virtual int no_vertices();
    virtual ex vertex(int i);
    virtual Line line(int i);
    virtual ex repr();
    virtual string str();
    virtual ex integrate(ex f);
};
```

Here the function `repr` returns a list with the items (7)-(11). In addition to the functions also found in `Line`, `Triangle` has a function `line(int i)`, returning a line.

The basic usage of a triangle is as follows (see `triangle_ex1.cpp`),

```
ex p0 = lst(0.0,0.0,1.0);
ex p1 = lst(1.0,0.0,1.0);
ex p2 = lst(0.0,1.0,1.0);

Triangle triangle(p0,p1,p2);

ex repr = triangle.repr();
cout <<"t.repr "<<repr<<endl;
EQUAL_OR_DIE(repr, "{x==r,y==s,z==1.0,{r,0,1},{s,0,1-r}}");
```

```

ex f = x*y*z;
ex intf = triangle.integrate(f);
cout <<"intf "<<intf<<endl;
EQUAL_OR_DIE(intf, "1/24");

```

Tetrahedron A tetrahedron is defined by four points \mathbf{x}_0 , \mathbf{x}_1 , \mathbf{x}_2 , and \mathbf{x}_3 . Associated with a tetrahedron are four triangles and six lines. The convention used so far is that

- the first line connects \mathbf{x}_0 and \mathbf{x}_1 ,
- the second line connects \mathbf{x}_0 and \mathbf{x}_2 ,
- the third line connects \mathbf{x}_0 and \mathbf{x}_3 ,
- the fourth line connects \mathbf{x}_1 and \mathbf{x}_2 ,
- the fifth line connects \mathbf{x}_1 and \mathbf{x}_3 ,
- the sixth line connects \mathbf{x}_2 and \mathbf{x}_3 .

The i 'th triangle has the vertices $\mathbf{x}_{i\%4}$, $\mathbf{x}_{(i+1)\%4}$, and $\mathbf{x}_{(i+2)\%4}$, where $\%$ is the modulus operator. The tetrahedron can be represented as, see also Figure 3,

$$x = x_0 + ar + bs + ct, \quad (12)$$

$$y = y_0 + dr + es + ft, \quad (13)$$

$$z = z_0 + gr + hs + kt, \quad (14)$$

$$t \in [0, 1 - r - s], \quad (15)$$

$$s \in [0, 1 - r], \quad (16)$$

$$r \in [0, 1], \quad (17)$$

where $(a, d, g) = (x_1 - x_0, y_1 - y_0, z_1 - z_0)$, $(b, e, h) = (x_2 - x_0, y_2 - y_0, z_2 - z_0)$, and $(c, f, k) = (x_3 - x_0, y_3 - y_0, z_3 - z_0)$.

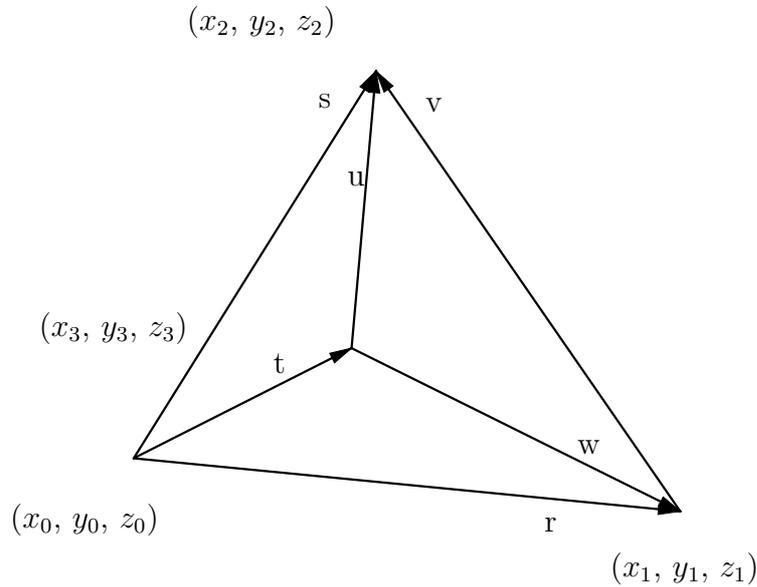
As earlier, integration is performed with substitution,

$$\int_T f(x, y, z) dx dy dz = \int_0^1 \int_0^{1-r} \int_0^{1-r-s} f(x(r, s, t), y(r, s, t), z(r, s, t)) D dt ds dr,$$

where D is the determinant of,

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & k \end{bmatrix}.$$

Figure 3: A tetrahedron.



Software Component: Tetrahedron The class `Tetrahedron` implements a general tetrahedron. It is defined as follows (see `Polygon.h`):

```
class Tetrahedron : public Polygon {
public:
    Tetrahedron(string subscript) {}
    Tetrahedron(ex x0, ex x1, ex x1, ex x2, string subscript = "");
    ~Tetrahedron(){}

    virtual int no_vertices();
    virtual ex vertex(int i);
    virtual Line line(int i);
    virtual Triangle triangle(int i);
    virtual ex repr();
    virtual string str();
    virtual ex integrate(ex f);
};
```

The function `repr` returns a list representing (12)–(17). In addition to the usual functions it has the functions `line(int i)` and `triangle(int i)` for returning the i 'th line and the i 'th triangle, respectively.

Its basic usage is as follows (see `tetrahedron_ex1.cpp`),

```
ex p0 = lst(0.0,0.0,0.0);
ex p1 = lst(1.0,0.0,0.0);
ex p2 = lst(0.0,1.0,0.0);
ex p3 = lst(0.0,0.0,1.0);
```

```

Tetrahedron tetrahedron(p0,p1,p2,p3);

ex repr = tetrahedron.repr();
cout <<"t.repr " <<repr<<endl;
EQUAL_OR_DIE(repr, "{x==r,y==s,z==t,{r,0,1},{s,0,1-r},{t,0,1-s-r}}");

ex f = x*y*z;
ex intf = tetrahedron.integrate(f);
EQUAL_OR_DIE(intf, "1/720");

```

3.2 Polynomial Spaces

The space of polynomials of degree less or equal to n , \mathbb{P}_n , plays a fundamental role in the construction of finite elements. There are many ways to represent this polynomial space. The perhaps visually nicest representation is having it spanned by the basis (in 1D) $\{1, x, x^2, \dots, x^n\}$. This representation is not suitable for polynomials of high degree².

In 1D, \mathbb{P}_n is spanned by functions on the form

$$v = a_0 + a_1x + \dots + a_nx^n = \sum_{i=0}^n a_i x^i \quad (18)$$

In 2D on triangles, \mathbb{P}_n is spanned by functions on the form:

$$v = \sum_{i,j=0}^{i+j \leq n} a_{ij} x^i y^j \quad (19)$$

In 2D on quadrilaterals, \mathbb{P}_n is spanned by functions on the form:

$$v = \sum_{i,j=0}^{i,j \leq n} a_{ij} x^i y^j \quad (20)$$

The corresponding polynomial spaces in 3D are completely analogous.

Software Component: Polynomial Space The following functions generate symbolic expressions for the above polynomial spaces (18), (19), and (20), their corresponding polynomial spaces in 3D and their vector counterparts.

```

// generates a polynomial of any order on a line, a triangle, or a tetrahedron
ex pol(int order, int nsd, const string a);

// generates a vector polynomial of any order on a line, a triangle or a tetrahedron
lst polv(int order, int nsd, const string a);

// generates a polynomial of any order on a square or a box
ex polb(int order, int nsd, const string a);

// generates a vector polynomial of any order on a square or a box
lst polbv(int order, int nsd, const string a);

```

The function `pol` returns a list with the following 3 items,

²In that case, one should use the Bernstein polynomials.

1. The polynomial, e.g., $a_0 + a_1x + \dots + a_nx^n$ in 1D.
2. A list of variables, e.g., $\{a_0, a_1, \dots, a_n\}$ in 1D.
3. A list containing the basis, e.g., $\{1, x, \dots, x^n\}$ in 1D.

The functions `polb`, `polv`, and `polbv` return lists that are completely analogous.

These abstract polynomials (or polynomial spaces) can be easily manipulated, e.g., (see also `pol.cpp`),

```
int order = 2;
int nsd   = 2;

ex polynom_space = pol(order,nsd, "a");
cout <<"polynom_space "<<polynom_space<<endl;

ex p = polynom_space.op(0);
cout <<"polynom p = "<<p<<endl;
EQUAL_OR_DIE(p, "y^2*a5+x^2*a3+a2*y+y*x*a4+a0+a1*x");

ex dpdx = diff(p,x);
cout <<"dpdx = "<<dpdx<<endl;
EQUAL_OR_DIE(dpdx, "y*a4+a1+2*x*a3");

Triangle triangle(1st(0,0), 1st(1,0), 1st(0,1));
ex intp = triangle.integrate(p);
cout <<"integral of p over reference triangle = "<<intp<<endl;
EQUAL_OR_DIE(intp, "1/6*a2+1/6*a1+1/12*a5+1/2*a0+1/24*a4+1/12*a3");
```

3.2.1 Bernstein Polynomials

Another basis for \mathbb{P}_n is the Bernstein polynomials. This basis is much better suited for polynomials of high degree. Moreover, these polynomials can be easily expressed in barycentric coordinates, which makes them easy to adapt to, e.g., faces of polygons³ etc.

In 1D, the polynomial basis is on the form,

$$B_{i,n} = \binom{n}{i} x^i (1-x)^{n-i}, \quad i = 0, \dots, n.$$

And with this basis, \mathbb{P}_n can be spanned by functions on the form,

$$v = a_0 B_{0,n} + a_1 B_{1,n} + \dots + a_n B_{n,n}.$$

One reason for the widespread use of these polynomials is that they adapt easily to general triangles and tetrahedra, by using barycentric coordinates. Let b_1 , b_2 , and b_3 be the barycentric coordinates for the triangle shown in Figure 2. Then the basis is on the form,

$$B_{i,j,k,n} = \frac{n!}{i!j!k!} b_1^i b_2^j b_3^k, \quad \text{for } i + j + k = n.$$

³This will be used in the definition of the Raviart-Thomas element.

and \mathbb{P}_n is spanned by functions on the form,

$$v = \sum_{i+j+k=n} a_{i,j,k} B_{i,j,k,n}.$$

The Bernstein polynomials in 3D are completely analogous.

Software Components The following functions generates symbolic expressions for \mathbb{P}^n using the Bernstein basis,

```
// polynom of arbitrary order on a line, a triangle,
// or a tetrahedron using the Bernstein basis
ex bernstein(int order, Polygon& p, const string a);

// vector polynom of arbitrary order on a line, a triangle,
// or a tetrahedron using the Bernstein basis
lst bernsteinv(int order, Polygon& p, const string a);
```

These functions return lists that are analogous to the lists made by the functions `pol` and `polv` described on page 11.

As described earlier, GiNaC has the tools for manipulating these polynomial spaces, (see also `pol.cpp`)

```
ex polynom_space2 = bernstein(order,triangle, "a");
ex p2 = polynom_space2.op(0);
cout <<"polynom p2 = "<<p2<<endl;
EQUAL_OR_DIE(p2, "y^2*a0+2*(1-y-x)*x*a4+2*(1-y-x)*a3*y+(1-y-x)^2*a5+2*a1*y*x+a2*x^2");

ex dp2dx = diff(p2,x);
cout <<"dp2dx = "<<dp2dx<<endl;
EQUAL_OR_DIE(dp2dx, "2*a1*y+2*(-1+y+x)*a5+2*a2*x+2*(1-y-x)*a4-2*a3*y-2*x*a4");

ex intp2 = triangle.integrate(p2);
cout <<"integral of p2 over reference triangle = "<<intp2<<endl;
EQUAL_OR_DIE(intp2, "1/12*a3+1/12*a2+1/12*a1+1/12*a5+1/12*a0+1/12*a4");
```

3.3 A Finite Element

Before we start describing how to construct a finite element based on the Definition 3.1, we will concentrate on the *usage* of a finite element. A finite element has only two interesting components, the basis functions $\{N_i\}$ and the corresponding degrees of freedom $\{L_i\}$. The basis functions (and their derivatives) are used to compute the element matrices and the element vectors, while the degrees of freedom are used to define the mapping between the element matrices/vectors and the global matrix/vector. As we see in the following, the observation that only these two components are needed leads us to a minimalistic definition of a finite element in our software tools.

Software Component: Finite Element Due to the powerful expression class in GiNaC, `ex`, our base class for the finite elements can be very small. Both the basis function N_i and the corresponding degree of freedom L_i can be well represented as an `ex`. Hence, the following definition of a finite element is suitable,

```

class FE {
public:
    FE() {}
    ~FE() {}

    virtual void set(Polygon& p);           // Set polygonal domain
    virtual Polygon& getPolygon();         // Get polygonal domain
    virtual ex N(int i);                   // The i'th basis function
    virtual ex dof(int i);                 // The i'th degree of freedom
    virtual int nbf();                      // The number of basis functions/
                                           // degrees of freedom
};

```

The usage of a finite element is as follows (see `fe_ex1.cpp`),

```

ex Ni;
ex gradNi;
ex dofi;
for (int i=1; i<= fe.nbf() ; i++) {
    Ni = fe.N(i);
    gradNi = grad(Ni);
    dofi = fe.dof(i);
    cout <<"The basis function, N("<i<<"="<<Ni<<endl;
    cout <<"The gradient of N("<i<<"="<<gradNi<<endl;
    cout <<"The corresponding dof, L("<i<<"="<<dofi<<endl;
}

```

When `fe_ex1` is runned it produces the following output:

```

The basis function, N(1)=2*y^2-y
The gradient of N(1)=[[0],[-1+4*y]]
The corresponding dof, L(1)={0,1}
The basis function, N(2)=4*y*x
The gradient of N(2)=[[4*y],[4*x]]
The corresponding dof, L(2)={1/2,1/2}
The basis function, N(3)=2*x^2-x
The gradient of N(3)=[[1+4*x],[0]]
The corresponding dof, L(3)={1,0}
The basis function, N(4)=-4*y*x-4*y^2+4*y
The gradient of N(4)=[[4*y],[4-8*y-4*x]]
The corresponding dof, L(4)={0,1/2}
The basis function, N(5)=-4*y*x-4*x^2+4*x
The gradient of N(5)=[[4-4*y-8*x],[-4*x]]
The corresponding dof, L(5)={1/2,0}
The basis function, N(6)=1+4*y*x+2*x^2+2*y^2-3*y-3*x
The gradient of N(6)=[[3+4*y+4*x],[-3+4*y+4*x]]
The corresponding dof, L(6)={0,0}

```

The computation of the element matrix for a Poisson problem is as follows (see `fe_ex2.cpp`),

```

Triangle T(1st(0,0), 1st(1,0), 1st(0,1), "t");
int order = 2;

std::map<std::pair<int,int>, ex> A;
std::pair<int,int> index;
LagrangeFE fe;
fe.set(order);
fe.set(T);
fe.compute_basis_functions();
for (int i=1; i<= fe.nbf() ; i++) {
    index.first = i;
    for (int j=1; j<= fe.nbf() ; j++) {
        index.second = j;
        ex nabla = inner(grad(fe.N(i)), grad(fe.N(j)));
        ex Aij = T.integrate(nabla);
    }
}

```

```

    A[index] = Aij;
  }
}

```

Here, we have used the class `LagrangeFE`, which is a subclass of `FE`, that implements Lagrangian elements of arbitrary order. The construction of this element is described later in Section 3.5.1.

3.4 Degrees of Freedom

As we have seen earlier, for each element e , we have a local set of degrees of freedom $\{L_i^e\}$, which in general are linear forms on the polynomial space. Degrees of freedom and linear forms are quite general concepts, but the reader not familiar with this general definition can think of them for instance as nodal values at vertices, i.e.,

$$L_i(v) = v(\mathbf{x}_i).$$

Another example is the integral of v over an edge (or a face), e_i , of the polygon,

$$L_i(v) = \int_{e_i} v ds.$$

The most important thing with the degrees of freedom is that they provide a mapping from the local degree of freedom, L_i^e , on a given element, e , to the global degree of freedom, L_j , which in turn provides the mapping between the element matrices/vectors and the global matrix/vector⁴. Hence, we have the following mapping,

$$(e, i) \rightarrow L_i^e \rightarrow L_j \rightarrow j. \quad (21)$$

Here e , i , and j are integers, while L_i^e and L_j are degrees of freedom (or linear forms). Additionally, given a global degree of freedom we have a mapping to the local degrees of freedom,

$$j \rightarrow L_j \rightarrow \{L_{i(e)}^e\}_{e \in E(j)} \rightarrow \{(e, i(e))\}_{e \in E(j)}. \quad (22)$$

Here $E(j)$ is the set of elements sharing the degree of freedom L_j .

Software Component: Degrees of Freedom A degree of freedom, local or global, is well represented as an `ex` (in fact `ex` is more general than a linear form). Hence, to implement proper tools for degrees of freedom, we only need to provide the mappings (21) and (22). The class `Dof` provides these mappings,

```

class Dof {
protected:
  int counter;
  // the structures loc2dof, dof2index, and doc2loc are completely dynamic
  // they are all initialized and updated by insert_dof(int e, int d, ex dof)

```

⁴The degrees of freedom is also used to define a basis for the polynomial space, as we will see later.

```

// (int e, int i) -> ex Li
map<pair<int,int>, ex>          loc2dof;
// (ex Lj) -> int j
map<ex,int,ex_is_less>        dof2index;
// (int j) -> ex Lj
map<int,ex>                    index2dof;
// (ex Lj) -> vector< pair<el, i1>, .. pair<en, in> >
map <ex, vector<pair<int,int> >,ex_is_less > dof2loc;

public:
Dof() { counter = 0; }
~Dof() {}
int insert_dof(int e, int j, ex Lj); // to update the internal structures

// helper functions when the dofs have been set (in Dof)
// These do not modify the internal structure
int glob_dof(int e, int j);
int glob_dof(ex Lj);
ex glob_dof(int j);
int size();
vector<pair<int, int> > glob2loc(int j);
};

```

Here, the function `int insert_dof(int e, int i, ex Li)` creates the various mappings between the local dof L_i , in element e , and the global dof L_j . This is the only function for initializing the mappings. After the mappings have been initialized, they can be used as follows,

- `int glob_dof(int e, int i)` is the mapping $(e, i) \rightarrow j$,
- `int glob_dof(ex Lj)` is the mapping $L_j \rightarrow j$,
- `ex glob_dof(int j)` is the mapping $j \rightarrow L_j$,
- `vector<pair<int, int> > glob2loc(int j)` is the mapping $j \rightarrow \{(e, i(e))\}$.

The following code shows how to make two finite elements, assign their local degrees of freedom to the global set of degrees of freedom in `Dof`, and print out the local degrees of freedom associated with each global degree of freedom (see also `dof_ex.cpp`):

```

Dof dof;

Triangle t1(1st(0,0), 1st(1,0), 1st(0,1));
Triangle t2(1st(1,1), 1st(1,0), 1st(0,1));

// Create a finite element and corresponding
// degrees of freedom on the first triangle
int order = 2;
LagrangeFE fe;
fe.set(order);
fe.set(t1);
fe.compute_basis_functions();
for (int i=1; i<= fe.nbf(); i++) {
    cout <<"fe.dof("<<i<<"= "<<fe.dof(i)<<endl;
    // insert local dof in global set of dofs
    dof.insert_dof(1,i, fe.dof(i));
}

// Create a finite element and corresponding
// degrees of freedom on the second triangle
fe.set(t2);

```

```

fe.compute_basis_functions();
for (int i=1; i<= fe.nbf() ; i++) {
  cout <<"fe.dof("<<i<<")= "<<fe.dof(i)<<endl;
  // insert local dof in global set of dofs
  dof.insert_dof(2,i, fe.dof(i));
}

// Print out the global degrees of freedom an their
// corresponding local degrees of freedom
vector<pair<int,int> > vec;
pair<int,int> index;
ex exdof;
for (int i=1; i<= dof.size(); i++) {
  exdof = dof.glob_dof(i);
  vec = dof.glob2loc(i);
  cout <<"global dof " <<i<<" dof "<<exdof<<endl;
  for (int j=0; j<vec.size(); j++) {
    index = vec[j];
    cout <<" element "<<index.first<<" local dof "<<index.second<<endl;
  }
}

```

In the previous example, the reader that also runs the companion code will notice that the degrees of freedom in `LagrangeFE` are not linear forms on polynomial spaces, i.e.,

$$L_i(v) = v(\mathbf{x}_i).$$

They are instead represented as points, \mathbf{x}_i , which is the usual way to represent these degrees of freedom in finite element software (because of their obvious simplicity compared to linear forms on polynomial spaces). Hence, the degrees of freedom in `LagrangeFE` are actually implemented in the standard fashion. However, the tools we have described are far more general than conventional finite element codes. Still the tools are equally simple to use, due to the powerful expression class `ex` in GiNaC.

Our next example concerns degrees of freedom which are line integrals over the edges of triangles. Let T be a triangle with the edges e_i , $i \in [1, 3]$. The degree of freedom associated with e_i is then simply,

$$L_i(v) = \int_{e_i} v ds.$$

As our next example shows, such degrees of freedom can be implemented equally easy as the point values shown in the previous example (see `dof_ex2.cpp`):

```

Dof dof;

// create two triangles
Triangle t1(1st(0,0), 1st(1,0), 1st(0,1));
Triangle t2(1st(1,1), 1st(1,0), 1st(0,1));

// create the polynomial space
ex Nj = pol(1,2,"a");
cout <<"Nj " <<Nj<<endl;
Line line;
ex dofi;

// dofs on first triangle
for (int i=1; i<= 3; i++) {
  line = t1.line(i); // pick out the i'th line

```

```

    dofi = line.integrate(Nj); // create the dof which is a line integral
    dof.insert_dof(1,i, dofi); // insert local dof in global set of dofs
}

// dofs on second triangle
for (int i=1; i<= 3; i++) {
    line = t2.line(i); // pick out the i'th line
    dofi = line.integrate(Nj); // create the dof which is a line integral
    dof.insert_dof(2,i, dofi); // insert local dof in global set of dofs
}

```

3.5 Some Examples of Finite Elements

3.5.1 The Lagrangian Element

Finally, we will describe the construction of a Lagrangian element on a 2D triangle. As we saw in Section 3.2, the polynomial space \mathbb{P}_n in 2D is on the form

$$v = \sum_{i,j=0}^{i+j \leq n} a_{ij} x^i y^j.$$

Hence, to determine the basis functions $\{v_k\}$ we simply represented them in abstract form,

$$v_k = \sum_{i,j=0}^{i+j \leq n} a_{ij}^k x^i y^j.$$

Then the coefficients $\{a_{ij}^k\}$ are to be determined by the $(n+1)(n+2)/2$ degrees of freedom that are the nodal values at the the points \mathbf{x}_i , i.e.,

$$L_i(v_k) = v_k(\mathbf{x}_i).$$

Hence, we need a set of $(n+1)(n+2)/2$ nodal points to determine the coefficients $\{a_{ij}^k\}$ for each basis function. We have chosen to use the Bezier ordinates. Then it is simply a matter of solving the linear system

$$L_i(v_k) = v_k(\mathbf{x}_i) = \delta_{ik},$$

for each basis function v_k .

Software Component: The Lagrangian Element The Lagrangian element of arbitrary order in 2D and 3D, is implemented in `LagrangeFE.cpp`. The following code is taken from `fe_ex3.cpp`.

```

Triangle t(1st(0,0), 1st(1,0), 1st(0,1));
int order = 2; //second order elements
ex polynom;
1st variables;

// the polynomial spaces on the form:
// first item: a0 + a1*x + a2*y + a3*x^2 + a4*x*y ... the polynomial
// second item: a0, a1, a2, ... the coefficients
// third item 1, x, y, x^2 the basis
// Could also do:

```

```

// GiNaC::ex polynom_space = bernstein(order, t, "a");
ex polynom_space = pol(order, 2, "a");
ex polynom = polynom_space.op(0);

// the variables a0,a1,a2 ..
variables = ex_to<lst>(polynom_space.op(1));

ex Nj;
// The Bezier ordinates in which the basis function should be either 0 or 1
lst points = bezier_ordinates(t,order);

// Loop over all basis functions Nj and all points.
// Each basis function Nj is determined by a set of linear equations:
//   Nj(xi) = dirac(i,j)
// This system of equations is then solved by lsolve
for (int j=1; j <= points.nops(); j++) {
  lst equations;
  int i=0;
  for (int i=1; i<= points.nops() ; i++ ) {
    // The point xi
    ex point = points.op(i-1);
    // The equation Nj(x) = dirac(i,j)
    ex eq = polynom == dirac(i,j);
    // Substitute x = xi and y = yi and appended the equation
    // to the list of equations
    equations.append(eq.subs(lst(x == point.op(0) , y == point.op(1))));
  }

  // We solve the linear system
  ex subs = lsolve(equations, variables);
  // Substitute to get the Nj
  Nj = polynom.subs(subs);
  cout <<"Nj " <<Nj<<endl;
}

```

In this example the degrees of freedom are very simple. It is only a matter of evaluating the function v_k in the point \mathbf{x}_i (which in GiNaC is performed by substitution). Later we will see that more advanced degrees of freedom are readily available since we have stored the degrees of freedom as a set of `exes`.

3.5.2 The Crouizex-Raviart Element

The Crouizex-Raviart element is the nonconforming equivalent of linear continuous Lagrangian elements. The degrees of freedom are the values at the midpoint of the sides, i.e.,

$$L_i(v) = v(x_{m(e_i)}),$$

where $x_{m(e_i)}$ is the midpoint on the edge, e_i . An equivalent definition of the degrees of freedom is,

$$L_i(v) = \int_{e_i} v ds,$$

This is the definition we will use.

Software Component: The Crouzeix-Raviart Element The following code, which is from the file `CrouzeixRaviart.cpp`, shows how this element can

be defined in 2D. The definition of the element in 3D can also be found in this file.

```

Triangle triangle;

// create the polynomial space
ex polynom_space = bernstein(1, triangle, "a");
ex polynom = polynom_space.op(0);
ex variables = polynom_space.op(1);
ex basis = polynom_space.op(2);

// create the dofs
int counter = 0;
symbol t("t");
for (int i=1; i<= 3; i++) {
    Line line = triangle.line(i);
    ex dofi = line.integrate(polynom);
    dofs.insert(dofs.end(),dofi);
}

// solve the linear system to compute
// each of the basis functions
for (int i=1; i<= 3; i++) {
    lst equations;
    for (int j=1; j<= 3; j++) {
        equations.append(dofs[j-1] == dirac(i,j));
    }
    ex sub = lsolve(equations, variables);
    ex Ni = polynom.subs(sub);
    Ns.insert(Ns.end(),Ni);
}

```

This element can be used in a standard fashion, (see also `crouzeixraviart_ex.cpp`),

```

CrouzeixRaviart fe;
fe.set(1);
fe.set(p);
fe.compute_basis_functions();
for (int i=1; i<= fe.nbf() ; i++) {
    cout <<"fe.N("<<i<<"="<<fe.N(i)<<endl;
}

```

See also the Python implementation of this element in Section 6.

3.5.3 The P_0 Element

The P_0 element consists of piecewise constants, i.e.,

$$v|_T = 1,$$

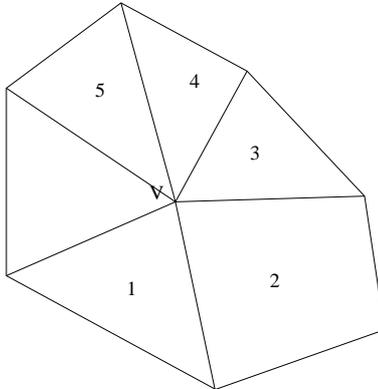
where T is the polygon. This element is discontinuous across elements.

Software Component: The P_0 Element The P_0 element is implemented in the class `P0`. The implementation is straightforward.

3.5.4 The Discontinuous Lagrangian Element

The discontinuous Lagrangian elements are similar to the continuous Lagrangian elements except for the fact that they are discontinuous. Hence, locally on the

Figure 4: Some triangles with the common vertex V .



polygon T , the basis functions are the same. The difference is that discontinuous Lagrangian elements are not continuous between elements.

To exemplify this we consider the continuous and the discontinuous linear Lagrangian elements in 2D. In Figure 4 we see that the triangles $1, \dots, 5$ all share the common vertex V . For continuous Lagrangian elements, this means that there will be one degree of freedom associated with V . On the other hand, for discontinuous Lagrangian elements, there will be one degree of freedom associated with V per triangle. Hence, in the concrete case depicted in Figure 4, there will be 5 degrees of freedom associated with V . Each degree of freedom is associated with a basis function which is 1 in V , 0 in the other vertices, and zero outside the triangle.

Software Component: The discontinuous Lagrangian element The implementation of the discontinuous Lagrangian element is really easy because this element is identical to the continuous Lagrangian element locally. Hence, the basis functions on each element is the same. We only need to modify the degrees of freedom.

The degrees of freedom for the discontinuous Lagrangian elements are such that for each element, each degree of freedom is new. Hence, none of degrees of freedom are shared among elements. It is fairly easy to implement this. Assume that the polygons in the mesh or the elements in the finite element space are numbered. Then the degree of freedom can be represented by both the vertex x_i and the element number e associated with the polygon T_e ,

$$L_i^e(v) = v|_{T_e}(x_i),$$

where $v|_{T_e}$ means the restriction of v to the polygon T_e . It is important to take the restriction to T_e since v is in general discontinuous in x_i .

We have implemented the discontinuous Lagrangian element as a subclass of the continuous Lagrangian element, with an additional integer parameter `element` which is the element number. The class declaration is as follows,

```

class DiscontinuousLagrangeFE : public LagrangeFE{
    int element;
public:
    DiscontinuousLagrangeFE();
    ~DiscontinuousLagrangeFE() {}

    virtual void set(int order);
    virtual void set_element_number(int element);
    virtual void set(Polygon& p);
    virtual void compute_basis_functions();
    virtual int nbf();
    virtual GiNaC::ex& N(int i);
    virtual GiNaC::ex& dof(int i);
};

```

Earlier, the degrees of freedom for continuous Lagrangian elements were represented as vertices or points (instead of linear forms), as is usual in finite element codes. We do the same simplification here, and store the degrees of freedom as (x_i, e) , where x_i is the vertex/point and e is the element number associated with T_e . This is implemented in the functions `compute_basis_functions`:

```

void DiscontinuousLagrangeFE:: compute_basis_functions() {
    LagrangeFE:: compute_basis_functions();
    for (int i=0; i< dofs.size(); i++) {
        dofs[i] = lst(dofs[i], element);
    }
}

```

The usage is standard (see `disconlagrange_ex.cpp`),

```

Dof dof;
// create two triangles
Triangle t1(lst(0,0), lst(1,0), lst(0,1));
Triangle t2(lst(1,1), lst(1,0), lst(0,1));

int order = 2;

DiscontinuousLagrangeFE fe;
fe.set(order);
fe.set(t1);
fe.set_element_number(1);
fe.compute_basis_functions();
usage(fe);
for (int i=1; i<= fe.nbf(); i++) {
    dof.insert_dof(1,i,fe.dof(i));
}

fe.set(t2);
fe.set_element_number(2);
fe.compute_basis_functions();
usage(fe);
for (int i=1; i<= fe.nbf(); i++) {
    dof.insert_dof(2,i,fe.dof(i));
}

// Print out the global degrees of freedom an their
// corresponding local degrees of freedom
vector<pair<int,int> > vec;
pair<int,int> index;
ex exdof;
for (int i=1; i<= dof.size(); i++) {
    exdof = dof.glob_dof(i);
    vec = dof.glob2loc(i);
    cout <<"global dof " <<i<<" dof "<<exdof<<endl;
    for (int j=0; j<vec.size(); j++) {

```

```

    index = vec[j];
    cout <<" element "<<index.first<<" local dof "<<index.second<<endl;
  }
}

```

When this program (`disconlagrange`) runs, it prints out 12 degrees of freedom in contrast to 9 which it would be for continuous Lagrangian elements.

4 Mixed Finite Elements

Mixed finite element methods typically refer to discretization methods for systems of PDEs where different finite elements are used for the different unknowns. For instance, in incompressible flow problems, one typically has (at least) two unknowns, the velocity \mathbf{v} and the pressure p . It is wellknown that the velocity elements should have higher order than the pressure elements. The reasons for this have been extensively studied the last 30 years, and we will not go into details on this here, see e.g., Brezzi and Fortin [2] and Girault and Raviart[4].

What we will do here is to describe mixed finite elements from the programmers point of view. In this setting, we simply refer to mixed elements as a collection of finite elements of different types on the same polygon. Several examples are listed in the following sections.

4.1 The Taylor–Hood and the $\mathbb{P}_n^d - \mathbb{P}_{n-2}$ Elements for the Stokes problem

The Taylor–Hood and the $\mathbb{P}_n^d - \mathbb{P}_{n-2}$ elements are mixed elements that are popular for incompressible flow. The elements for both the velocity and the pressure are of Lagrangian type, but have different order. The Taylor–Hood element on a polygon T is,

$$\mathbf{v}(T) \in \mathbb{P}_2^d \quad \text{and} \quad p(T) \in \mathbb{P}_1.$$

The $\mathbb{P}_n - \mathbb{P}_{n-2}$ element on a polygon T is,

$$\mathbf{v}(T) \in \mathbb{P}_n^d \quad \text{and} \quad p(T) \in \mathbb{P}_{n-2}, \quad n \geq 2.$$

For $n > 2$ the pressure element is of Lagrangian type, while for $n=2$ the pressure element is piecewise constant. These elements satisfy the Babuska-Brezzi condition.

The Taylor–Hood elements can be created as follows, (see also `taylorhood_ex.cpp`)

```

VectorLagrangeFE v_fe;
v_fe.set(2);
v_fe.set_size(2);
v_fe.set(domain);
v_fe.compute_basis_functions();

LagrangeFE p_fe;
p_fe.set(1);
p_fe.set(domain);
p_fe.compute_basis_functions();

```

The $\mathbb{P}_n^d - \mathbb{P}_{n-2}$ element can be made by changing the order of the elements with the `set` function.

4.2 The Mixed Crouizex-Raviart Element for the Stokes problem

The mixed Crouizex-Raviart element is a nonconforming linear element for the velocity and piecewise constant for the pressure. The Crouizex-Raviart element was described in Section 3.5.2, while the P_0 element was described in Section 3.5.3.

These elements can be made as follows (see also `crouzeixraviart_ex2.cpp`)

```
ReferenceTriangle domain;

VectorCrouzeixRaviart v_fe;
v_fe.set_size(2);
v_fe.set(domain);
v_fe.compute_basis_functions();

P0 p_fe;
p_fe.set(domain);
p_fe.compute_basis_functions();
```

4.3 The Mixed Raviart-Thomas Element for the Poisson Problem on Mixed Form

Velocity Element The family of Raviart-Thomas elements is popular when considering the mixed formulation of elliptic problems. In this case the polynomial space is not \mathbb{P}_n^d , but

$$\mathbb{P}_n^d + \mathbf{x}\mathbb{P}_n. \quad (23)$$

And the degrees of freedom are,

$$\int_{e_i} \mathbf{v} \cdot \mathbf{n} p_k ds, \quad \forall p_k \in \mathbb{P}_k(e_i), \quad (24)$$

$$\int_T \mathbf{v} \cdot \mathbf{p}_{k-1} dx, \quad \forall p_{k-1} \in \mathbb{P}_{k-1}^d(T), \quad (25)$$

where T is the polygon domain and e_i is its edges (in 2D) or faces (in 3D). Degrees of freedom which are integrals have been dealt with already for the Crouizex-Raviart element in Section 3.5.2. Hence, there are mainly two new concepts we need to deal with to implement this element. It is the polynomial space, which is on the form (23), and the polynomial spaces on faces or edges of the polygon, as in (24). Both concepts will be dealt with below.

Pressure Element The pressure element is discontinuous polynomials of degree n . The \mathbb{P}_0 element is described in Section 3.5.3, while the discontinuous \mathbb{P}_n element is described in Section 3.5.4.

Software Component: The Raviart-Thomas Velocity Element First, we described how to make the polynomial space (23). The polynomial spaces, $\mathbb{P}_n(T)$ and $\mathbb{P}_n^d(T)$ on a polygonal domain, can be made by the functions `bernstein` and `bernsteinv`, respectively. However, we can not just add the spaces $\mathbb{P}_n^d(T)$ and $\mathbf{x}\mathbb{P}_n(T)$ together. Because, some of the basis functions are the same in both space, while others are not. Consider for instance $\mathbb{P}_1^d(T)$, which has the basis functions,

$$\{(0, 1)^T, (1, 0)^T, (x, 0)^T, (0, x)^T, (y, 0)^T, (0, y)^T\}$$

while $\mathbf{x}\mathbb{P}_1(K)$ has the following basis functions

$$\{(x, 0)^T, (x^2, 0)^T, (xy)^T, (0, y)^T, (0, y^2)^T, (0, xy)^T\}.$$

Hence $(x, 0)^T$ and $(0, y)^T$ are common.

The way we solve this problem is that we create the two spaces $\mathbb{P}_n^d(T)$ and $\mathbf{x}\mathbb{P}_n(T)$ independently. We then have two polynomial spaces, each with two independent sets of variables (or degrees of freedom). The variables associated with a basis in $\mathbf{x}\mathbb{P}_n(T)$ which is also a basis in $\mathbb{P}_n^d(T)$ is then removed. This is done by removing all variables associated with basis functions that have degree less than $n - 1$ in \mathbb{P}_n from $\mathbf{x}\mathbb{P}_n(T)$. This is done as follows in 2D (both 2D and 3D elements of arbitrary order are implemented in `RaviartThomas.cpp`),

```
Triangle& triangle = (Triangle&)(*p);
lst equations;
lst variables;
ex polynom_space1 = bernstein(order-1, triangle, "a");
ex polynom1 = polynom_space1.op(0);
ex polynom1_vars = polynom_space1.op(1);
ex polynom1_basis = polynom_space1.op(2);

lst polynom_space2 = bernsteinv(order-1, triangle, "b");
ex polynom2 = polynom_space2.op(0).op(0);
ex polynom3 = polynom_space2.op(0).op(1);

lst pspace = lst( polynom2 + polynom1*x,
                 polynom3 + polynom1*y);

// remove multiple dofs
if ( order >= 2 ) {
  ex expanded_pol = expand(polynom1);
  for (int c1=0; c1<= order-2;c1++) {
    for (int c2=0; c2<= order-2;c2++) {
      for (int c3=0; c3<= order-2;c3++) {
        if ( c1 + c2 + c3 <= order - 2 ) {
          ex eq = expanded_pol.coeff(x,c1).coeff(y,c2).coeff(z,c3);
          if ( eq != numeric(0) ) {
            equations.append(eq == 0);
          }
        }
      }
    }
  }
}
```

Second, we described how to implement the degrees of freedom (24)-(25).

The degrees of freedom associated with the edges,

$$\int_{e_i} \mathbf{v} \cdot \mathbf{n} p_k ds, \forall p_k \in \mathbb{P}_k(e_i),$$

are implemented as follows (Notice that the polynomial space on the edges of the triangle is made by creating Bernstein polynomials in standard fashion).

```

ex bernstein_pol;

int counter = 0;
symbol t("t");
ex dofi;
// loop over all edges
for (int i=1; i<= 3; i++) {
  Line line = triangle.line(i);
  lst normal_vec = normal(triangle, i);
  bernstein_pol = bernstein(order-1, line, istr("a",i));
  ex basis_space = bernstein_pol.op(2);
  ex pspace_n = inner(pspace, normal_vec);

  // loop over all basis functions on current edge
  ex basis;
  for (int i=0; i< basis_space.nops(); i++) {
    counter++;
    basis = basis_space.op(i);
    ex integrand = pspace_n*basis;
    dofi = line.integrate(integrand);
    dofs.insert(dofs.end(), dofi);
    ex eq = dofi == numeric(0);
    equations.append(eq);
  }
}

```

The degrees of freedom associated with the whole triangle,

$$\int_T \mathbf{v} \cdot \mathbf{p}_{k-1} dx, \forall p_{k-1} \in \mathbb{P}_{k-1}^d(T),$$

is implemented as

```

// dofs related to the whole triangle
lst bernstein_polv;
if ( order > 1) {
  counter++;
  bernstein_polv = bernsteinv(order-2, triangle, "a");
  ex basis_space = bernstein_polv.op(2);
  for (int i=0; i< basis_space.nops(); i++) {
    lst basis = ex_to<lst>(basis_space.op(i));
    ex integrand = inner(pspace, basis);
    dofi = triangle.integrate(integrand);
    dofs.insert(dofs.end(), dofi);
    ex eq = dofi == numeric(0);
    equations.append(eq);
  }
}

```

In the above code we have formed the linear system,

$$L_i(v) = 0$$

To compute the different v_j we then produce different right hand sides corresponding to δ_{ij} and solve the system. How this is done can be seen in the `RaviartThomas.cpp`.

Software Components: The Raviart-Thomas Pressure Element The pressure elements have been described in Section 3.5.3 and Section 3.5.4.

5 Computing Element Matrices

Our next task is to compute element matrices. As earlier, everything will be done symbolically⁵. There are several reasons for doing the computations symbolically:

- Everything is exact (No floating point precision issues)!
- Differentiation of the weak form with respect to the variables is possible (Easy to compute the Jacobian for nonlinear PDEs).
- In case one uses integers and rational numbers as input (e.g., the vertices of the polygon) one gets rational numbers as output. This enables nice output.
- In case one uses symbols as input, one get symbols as output. Hence, one might actually compute an abstract element matrix, where each entry in the matrix is a function of the vertices of the polygon, $\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_n$, which are symbols. We will consider this in more detail later.
- Every step can be checked against analytic computations. We can even, as we will see, produce output in L^AT_EX format, for easy reading.

5.1 A Poisson Problem

The Poisson problem is on the form,

$$\begin{aligned} -\Delta u &= f, & \text{in } \Omega, \\ u &= h, & \text{on } \partial\Omega_E, \\ \frac{\partial u}{\partial n} &= g, & \text{on } \partial\Omega_N, \end{aligned}$$

where $\partial\Omega = \partial\Omega_E \cup \partial\Omega_N$.

The weak form of the Poisson problem is (as we have already used): Find $u \in V_h$ such that

$$a(u, v) = b(v), \quad \forall v \in V_0.$$

where,

$$\begin{aligned} a(u, v) &= \int_{\Omega} \nabla u \cdot \nabla v \, dx, \\ b(v) &= \int_{\Omega} f v \, dx + \int_{\Gamma_N} g v \, ds. \end{aligned}$$

⁵We will add numerical integration etc. later.

and $V_k = \{v \in H^1; v|_{\partial\Omega_E} = k\}$, for $k = 0, h$.

From this weak form we obtain the element matrix, see e.g., Brenner og Scott [1], Ciarlet [3], or Langtangen [5],

$$A_{ij} = a(N_i, N_j) = \int_T \nabla N_j \cdot \nabla N_i \, dx.$$

This is implemented in the function `compute_Poisson_element_matrix` in `ElementComputations.cpp`,

```
void compute_Poisson_element_matrix(
    FE& fe,
    Dof& dof,
    std::map<std::pair<int,int>, ex>& A)
{
    std::pair<int,int> index;

    // Insert the local degrees of freedom into the global Dof
    for (int i=1; i<= fe.nbf(); i++) {
        dof.insert_dof(1,i,fe.dof(i));
    }

    Polygon& domain = fe.getPolygon();

    // The term (grad u, grad v)
    for (int i=1; i<= fe.nbf(); i++) {
        index.first = dof.glob_dof(fe.dof(i)); // fetch the global dof for Ni
        for (int j=1; j<= fe.nbf(); j++) {
            index.second = dof.glob_dof(fe.dof(j)); // fetch the global dof for Nj
            ex nabla = inner(grad(fe.N(i)), // compute the integrand
                            grad(fe.N(j)));
            ex Aij = domain.integrate(nabla); // compute the integral
            A[index] += Aij; // add to global matrix
        }
    }
}
```

Notice that in this example, both the degrees of freedom `dof` and the matrix `A` are global.

This function can be used as follows (see `fe_ex4.cpp`),

```
//matrix in terms of rational numbers
int order = 1;
Triangle triangle(1st(0,0), 1st(1,0), 1st(0,1));
LagrangeFE fe;
fe.set(order);
fe.set(triangle);
fe.compute_basis_functions();

Dof dof;
std::map<std::pair<int,int>, ex> A;
compute_Poisson_element_matrix(fe, dof, A);
```

In the above example, the vertices were integers, therefore the entries in the matrix will be rational numbers. In the following example the vertices are symbols.

```
//matrix in terms of symbols
symbol x0("x0"), x1("x1"), x2("x2");
symbol y0("y0"), y1("y1"), y2("y2");
Triangle triangle2(1st(x0,y0), 1st(x1,y1), 1st(x2,y2));

LagrangeFE fe2;
```

```

fe2.set(order);
fe2.set(triangle2);
fe2.compute_basis_functions();

Dof dof2;
std::map<std::pair<int,int>, ex> A2;
compute_Poisson_element_matrix(fe2, dof2, A2);

```

In this case `A2` will contain expressions involving the vertices, (x_0, y_0) , (x_1, y_1) , (x_2, y_2) (we used a triangle above).

The GiNaC library supports many different ways to print out the output. In the example below, we turn on L^AT_EX output with the command `cout <<latex;` before we print out `A2`.

```

cout <<"LaTeX format on output "<<endl;
cout <<latex;
print(A2);

```

This gives the following expression (compiled by latex) for $A[1, 1]$ (code for the other entries are also produced, but these are not shown here).

$$\begin{aligned}
A[1, 1] = & \frac{1}{2} \frac{x_0^2 |(-x_0 + x_1)(y_2 - y_0) - (-x_0 + x_2)(y_1 - y_0)|}{(-y_1 x_2 - x_0 y_2 + y_0 x_2 + y_2 x_1 + x_0 y_1 - y_0 x_1)^2} \\
& - \frac{y_1 y_0 |(-x_0 + x_1)(y_2 - y_0) - (-x_0 + x_2)(y_1 - y_0)|}{(-y_1 x_2 - x_0 y_2 + y_0 x_2 + y_2 x_1 + x_0 y_1 - y_0 x_1)^2} \\
& + \frac{1}{2} \frac{y_0^2 |(-x_0 + x_1)(y_2 - y_0) - (-x_0 + x_2)(y_1 - y_0)|}{(-y_1 x_2 - x_0 y_2 + y_0 x_2 + y_2 x_1 + x_0 y_1 - y_0 x_1)^2} \\
& - \frac{x_0 |(-x_0 + x_1)(y_2 - y_0) - (-x_0 + x_2)(y_1 - y_0)| x_1}{(-y_1 x_2 - x_0 y_2 + y_0 x_2 + y_2 x_1 + x_0 y_1 - y_0 x_1)^2} \\
& + \frac{1}{2} \frac{|(-x_0 + x_1)(y_2 - y_0) - (-x_0 + x_2)(y_1 - y_0)| x_1^2}{(-y_1 x_2 - x_0 y_2 + y_0 x_2 + y_2 x_1 + x_0 y_1 - y_0 x_1)^2} \\
& + \frac{1}{2} \frac{y_1^2 |(-x_0 + x_1)(y_2 - y_0) - (-x_0 + x_2)(y_1 - y_0)|}{(-y_1 x_2 - x_0 y_2 + y_0 x_2 + y_2 x_1 + x_0 y_1 - y_0 x_1)^2}
\end{aligned}$$

We can also print out C code,

```

cout <<"C code format on output "<<endl;
cout <<csrc;
print(A2);

```

Then the following code for $A[1, 1]$ is produced,

```

A[1,1]=(x0*x0)/pow(-y1*x2-x0*y2+y0*x2+y2*x1+x0*y1-y0*x1,2.0)
*fabs((-x0+x1)*(y2-y0)-(-x0+x2)*(y1-y0))/2.0
-y1/pow(-y1*x2-x0*y2+y0*x2+y2*x1+x0*y1-y0*x1,2.0)*y0
*fabs((-x0+x1)*(y2-y0)-(-x0+x2)*(y1-y0))
+1.0/pow(-y1*x2-x0*y2+y0*x2+y2*x1+x0*y1-y0*x1,2.0)*(y0*y0)
*fabs((-x0+x1)*(y2-y0)-(-x0+x2)*(y1-y0))/2.0
-x0/pow(-y1*x2-x0*y2+y0*x2+y2*x1+x0*y1-y0*x1,2.0)
*fabs((-x0+x1)*(y2-y0)-(-x0+x2)*(y1-y0))*x1
+1.0/pow(-y1*x2-x0*y2+y0*x2+y2*x1+x0*y1-y0*x1,2.0)*fabs((-x0+x1)*(y2-y0)
-(-x0+x2)*(y1-y0))*(x1*x1)/2.0+(y1*y1)
/pow(-y1*x2-x0*y2+y0*x2+y2*x1+x0*y1-y0*x1,2.0)
*fabs((-x0+x1)*(y2-y0)-(-x0+x2)*(y1-y0))/2.0

```

As is clear, these expressions can be rather large. GiNaC does not, by default, try to simplify these expressions. However, the above expressions is composed of smaller expressions that appear many times and it is possible to simplify these expressions fairly easy. For instance, the expression $(-y_1x_2 - x_0y_2 + y_0x_2 + y_2x_1 + x_0y_1 - y_0x_1)^2$ appears at least six times (and this is only in $A[1,1]$). Of course, this expression should be computed only once. It seems that GiNaC has powerful tools for expression three traversal that could enable generation of efficient code based on finding common sub-expressions, but we have not exploited these tools to a great extent yet. Some example code can be found in `check_visitor.cpp` in the sandbox.

5.2 A Poisson Problem on Mixed Form

The Poisson problem can also be written on mixed form,

$$\begin{aligned} \mathbf{u} - \nabla p &= 0, & \text{in } \Omega, \\ \nabla \cdot \mathbf{u} &= f, & \text{in } \Omega, \\ \mathbf{u} \cdot \mathbf{n} &= g, & \text{on } \partial\Omega_N, \\ p \mathbf{n} &= h \mathbf{n} & \text{on } \partial\Omega_E. \end{aligned}$$

Notice that essential boundary conditions for the Poisson problem on standard form become natural conditions for the Poisson problem on mixed form and vice versa.

The weak form of the Poisson problem on mixed form is: Find $\mathbf{u} \in \mathbf{V}_g, p \in Q$ such that

$$a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) = \mathbf{G}(\mathbf{v}), \quad \forall \mathbf{v} \in \mathbf{V}_0, \quad (26)$$

$$b(\mathbf{u}, q) = F(q), \quad \forall q \in Q, \quad (27)$$

where

$$a(\mathbf{u}, \mathbf{v}) = \int_{\Omega} \mathbf{u} \cdot \mathbf{v} \, dx, \quad (28)$$

$$b(\mathbf{u}, q) = \int_{\Omega} \nabla \cdot \mathbf{u} \, q \, dx, \quad (29)$$

$$F(q) = \int_{\Omega} f \, q \, dx, \quad (30)$$

$$\mathbf{G}(\mathbf{v}) = \int_{\Omega_E} h \mathbf{n} \cdot \mathbf{v} \, ds \quad (31)$$

$$\mathbf{V}_k = \{\mathbf{v} \in \mathbf{H}(\text{div}) : \mathbf{v} \cdot \mathbf{n}|_{\partial\Omega_N} = k\}, \quad k = 0, g.$$

$$\mathbf{H}(\text{div}) = \{\mathbf{v} \in \mathbf{L}^2 : \nabla \cdot \mathbf{v} \in L^2\},$$

$$Q = \begin{cases} L_0^2 & \text{if } \partial\Omega_N = \partial\Omega, \\ L^2 & \text{else.} \end{cases}$$

The function `compute_mixed_Poisson_element_matrix` in `ElementComputations.cpp` computes the element matrix for the mixed Poisson problem. We will not comment or list the code here because it is very similar to the code described in the next section. An example of use is in `mypoisson_ex.cpp`.

5.3 A Stokes Problem

The Stokes problem is on the form: Find \mathbf{u} and p such that

$$\begin{aligned} -\Delta \mathbf{u} + \nabla p &= \mathbf{f}, & \text{in } \Omega, \\ \nabla \cdot \mathbf{u} &= 0, & \text{in } \Omega, \\ \mathbf{u} &= \mathbf{g}, & \text{on } \partial\Omega_E, \\ \frac{\partial \mathbf{u}}{\partial \mathbf{n}} - p\mathbf{n} &= \mathbf{h}, & \text{on } \partial\Omega_N. \end{aligned}$$

The weak form for the Stokes problem is: Find $\mathbf{u} \in \mathbf{V}_g, p \in Q$ such that

$$\begin{aligned} a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) &= \mathbf{F}(\mathbf{v}), \quad \forall \mathbf{v} \in \mathbf{V}_0, \\ b(\mathbf{u}, q) &= 0, \quad \forall q \in Q, \end{aligned}$$

where

$$\begin{aligned} a(\mathbf{u}, \mathbf{v}) &= \int_{\Omega} \nabla \mathbf{u} : \nabla \mathbf{v} \, dx, \\ b(\mathbf{u}, q) &= - \int_{\Omega} \nabla \cdot \mathbf{u} \, q \, dx, \\ \mathbf{F}(q) &= \int_{\Omega} \mathbf{f} \cdot \mathbf{v} \, dx + \int_{\Omega_N} \mathbf{h} \cdot \mathbf{v} \, ds, \end{aligned}$$

$$\mathbf{V}_k = \{\mathbf{v} \in \mathbf{H}^1 : \mathbf{v}|_{\partial\Omega_E} = \mathbf{k}\}, \quad \mathbf{k} = \mathbf{0}, \mathbf{g},$$

$$Q = \begin{cases} L_0^2 & \text{if } \partial\Omega_E = \partial\Omega, \\ L^2 & \text{else.} \end{cases}$$

Notice that we have multiplied the equation for the mass conservation, $\nabla \cdot \mathbf{u} = 0$, with -1 to obtain symmetry.

The function `compute_Stokes_element_matrix` in `ElementComputations` implements the computation of an element matrix for the Stokes problem. The code is shown below.

```
void compute_Stokes_element_matrix(
    FE& v_fe,
    FE& p_fe,
    Dof& dof,
    std::map<std::pair<int,int>, ex>& A)
{
    std::pair<int,int> index;
    std::pair<int,int> index2;

    Polygon& domain = v_fe.getPolygon();

    // Insert the local degrees of freedom into the global Dof
```

```

for (int i=1; i<= v_fe.nbf(); i++) {
  dof.insert_dof(1,i,v_fe.dof(i));
}
for (int i=1; i<= p_fe.nbf(); i++) {
  dof.insert_dof(1,v_fe.nbf()+i,p_fe.dof(i));
}

// The term (grad u, grad v)
for (int i=1; i<= v_fe.nbf(); i++) {
  index.first = dof.glob_dof(v_fe.dof(i)); // fetch the global dof for v_i
  for (int j=1; j<= v_fe.nbf(); j++) {
    index.second = dof.glob_dof(v_fe.dof(j)); // fetch the global dof for v_j
    GiNaC::ex nabla = inner(grad(v_fe.N(i)),
                           grad(v_fe.N(j))); // compute the integrand
    GiNaC::ex Aij = domain.integrate(nabla); // compute the integral
    A[index] += Aij; // add to global matrix
  }
}

// The term (-div u, q)
for (int i=1; i<= p_fe.nbf(); i++) {
  index.first = dof.glob_dof(p_fe.dof(i)); // fetch the global dof for p_i
  for (int j=1; j<= v_fe.nbf(); j++) {
    index.second=dof.glob_dof(v_fe.dof(j)); // fetch the global dof for v_j
    ex divV= -p_fe.N(i)*div(v_fe.N(j)); // compute the integrand
    ex Aij = domain.integrate(divV); // compute the integral
    A[index] += Aij; // add to global matrix

    // Do not need to compute the term (grad(p),v), since the system is
    // symmetric. We simply set Aji = Aij
    index2.first = index.second;
    index2.second = index.first;
    A[index2] -= Aij;
  }
}
}

```

5.4 A Nonlinear Convection Diffusion Problem

Our next example concerns a nonlinear convection diffusion equation, where we compute the element matrix for the Jacobian typically arising in a Newton iteration. Let the PDE be,

$$(\mathbf{u} \cdot \nabla) \mathbf{u} - \Delta \mathbf{u} = \mathbf{f}, \quad \text{in } \Omega, \quad (32)$$

$$\mathbf{u} = \mathbf{g}, \quad \text{on } \partial\Omega. \quad (33)$$

This can be stated on weak form as: Find $\mathbf{u} \in \mathbf{V}_g$ such that

$$\mathbf{F}(\mathbf{u}, \mathbf{v}) = 0, \quad \forall \mathbf{v} \in \mathbf{V}_0,$$

where

$$\mathbf{F}(\mathbf{u}, \mathbf{v}) = \int_{\Omega} (\mathbf{u} \cdot \nabla \mathbf{u}) \cdot \mathbf{v} \, dx + \int_{\Omega} \nabla \mathbf{u} : \nabla \mathbf{v} \, dx - \int_{\Omega} \mathbf{f} \cdot \mathbf{v} \, dx$$

and

$$\mathbf{V}_k = \{\mathbf{v} \in \mathbf{H}^1 : \mathbf{v}|_{\partial\Omega} = \mathbf{k}\}, \quad \mathbf{k} = \mathbf{0}, \mathbf{g}.$$

The Jacobian is obtained by letting $\mathbf{u} = \hat{\mathbf{u}} = \sum_j u_j \mathbf{N}_j$, $\mathbf{v} = \mathbf{N}_i$ and differentiating \mathbf{F} with respect to u_j ,

$$J_{ij} = \frac{\partial F(\hat{\mathbf{u}}, \mathbf{N}_i)}{\partial u_j}.$$

This is precisely the way it is done with SyFi, (see also `nljacobian_ex.cpp`),

```
void compute_nlconvdiff_element_matrix(
    FE& fe,
    Dof& dof,
    std::map<std::pair<int,int>, ex>& A)
{
    std::pair<int,int> index;
    Polygon& domain = fe.getPolygon();

    // insert the local dofs into the global Dof object
    for (int i=1; i<= fe.nbf() ; i++) {
        dof.insert_dof(1,i,fe.dof(i));
    }

    // create the local U field: U = sum_k u_k N_k
    ex UU = matrix(2,1,lst(0,0));
    ex ujs = symbolic_matrix(1,fe.nbf(), "u");
    for (int k=1; k<= fe.nbf(); k++) {
        UU +=ujs.op(k-1)*fe.N(k);    // U += u_k N_k
    }

    //Get U represented as a matrix
    matrix U = ex_to<matrix>(UU.evalm());

    for (int i=1; i<= fe.nbf() ; i++) {
        index.first = dof.glob_dof(fe.dof(i));           // fetch global dof

        // First: the diffusion term in Fi
        ex gradU = grad(U);                               // compute the gradient
        ex Fi_diffusion = inner(gradU, grad(fe.N(i)));    // grad(U)*grad(Ni)

        // Second: the convection term in Fi
        ex Ut = U.transpose();                             // get the transposed of U
        ex UgradU = (Ut*gradU).evalm();                   // compute U*grad(U)
        ex Fi_convection = inner(UgradU, fe.N(i), true);  // compute U*grad(U)*Ni

        // add together terms for convection and diffusion
        ex Fi = Fi_convection + Fi_diffusion;

        // Loop over all uj and differentiate Fi with respect
        // to uj to get the Jacobian Jij
        for (int j=1; j<= fe.nbf() ; j++) {
            index.second = dof.glob_dof(fe.dof(j));       // fetch global dof
            symbol uj = ex_to<symbol>(ujs.op(j-1));       // cast uj to a symbol
            ex Jij = Fi.diff(uj,1);                       // differentiate Fi wrt. uj
            ex Aij = domain.integrate(Jij);                // intergrate the Jacobian Jij
            A[index] += Aij;                               // update the global matrix
        }
    }
}
```

Running the example `nljacobian_ex`, which employs second order continuous

Lagrangian elements, yields the following output for $A[1, 1]$,

$$A[1, 1] = \frac{1}{2} + \frac{2}{105}u_3 + \frac{2}{105}u_7 + \frac{1}{21}u_2\frac{13}{420}u_1 - \frac{1}{280}u_{11} \quad (34)$$

$$- \frac{1}{21}u_6 - \frac{1}{280}u_5\frac{1}{140}u_{10} + \frac{1}{210}u_9 - \frac{1}{140}u_4. \quad (35)$$

We have used GiNaC to generate the \LaTeX code, as described on Page 29.

6 Python Support

SyFi now comes with Python support. The SyFi Python modul is created by using the tool SWIG (<http://www.swig.org>). One should also install the Python interface to GiNaC called Swiginac (<http://swiginac.berlios.de/>).

The Python interface to SyFi is in its early stages, and has not yet been tested much. However, it appears to be quite easy to use in connection with Swiginac, as will be demonstrated.

First, a few remarks concerning GiNaC and Swiginac. In the C++ library GiNaC, one has the powerful object `ex` which is "typeless", i.e., it can be any GiNaC type, like e.g., `numeric`, `function`, `matrix`, or `lst`. On the other hand, Python is in itself typeless in the sense that a Python variable may refer to a Python object of any type. For this reason, the authors of Swiginac have chosen to let Python manage the "typelessness" and therefore Swiginac does not use objects of type `ex`. The following code shows how Swiginac can be used (see also `simple.py`),

```
from swiginac import *

x = symbol("x")
y = symbol("y")

f = sin(x)
print "f = ", f
dfdx = diff(f,x)
print "dfdx = ", dfdx
```

In Python, `x`, `y`, `f`, and `dfdx` are not declared to be of any type, but `x` and `y` will be `symbol` objects, while `f` and `dfdx` are `function` objects. In C++ all objects could have been declared to be of type `ex`. However, the underlying objects would be of type `symbol` or `function`, just as in Python.

Many SyFi functions require `ex` objects as input. Python objects created by Swiginac are usually not of type `ex`, instead they are Python wrappers around the underlying object. As described above, `x` is not of type `ex`, it is a Python wrapper on top of a `symbol` object. Fortunately, Swiginac has a function `toex` which can be used to create a Python `ex` object, which is a Python wrapper on top of a C++ `ex` object (which is on top of an underlying object such as e.g., `symbol`). Swiginac also has the function `eval` for accessing the underlying object of a Python object of type `ex`. Both `toex` and `eval` will be used frequently below. One can check the behavior of `toex` and `eval` as follows (see also `simple.py`),

```
print "type(x) ", type(x) # swiginac.symbol
print "type(x.eval()) ", type(x.eval()) # swiginac.symbolPtr
print "type(toex(x)) ", type(toex(x)) # swiginac.exPtr
print "type((toex(x)).eval()) ", type((toex(x)).eval()) # swiginac.exPtr
```

```

print "type(f) ", type(f) # swiginac.functionPtr
print "type(f.eval()) ", type(f.eval()) # swiginac.functionPtr
print "type(toex(f)) ", type(toex(f)) # swiginac.exPtr
print "type((toex(f)).eval()) ", type((toex(f)).eval()) # swiginac.functionPtr

```

SyFi classes and functions can be used in Python just as they are used in C++. The following example shows how to compute the element matrix for a Poisson problem using fourth order Lagrangian elements,

```

from swiginac import *
from SyFi import *

p0 = [0,0,0]; p1 = [1,0,0]; p2 = [0,1,0]
triangle = Triangle(toex(p0), toex(p1), toex(p2))

fe = LagrangeFE()
fe.set(4)
fe.set(triangle)
fe.compute_basis_functions()
print fe.nbf()
for i in range(1,fe.nbf()+1):
    for j in range(1,fe.nbf()+1):
        integrand = inner(grad(fe.N(i)),grad(fe.N(j)))
        Aij = triangle.integrate(integrand)
        print "A(%d,%d)=%%(i,j), Aij.eval()

```

Finally, we show a Python implementation of the Crouzeix-Raviart element (The C++ implementation can be found in the file `CrouzeixRaviart.cpp`). Notice that in this code we inherit the functions `ex N(int i)` and `ex dof(int i)` and the exvectors `Ns` and `dofs` from the C++ class `StandardFE`. Hence, thanks to SWIG, cross-language inheritance works, and we therefore only need to implement the function `compute_basis_functions`. The following example is implemented in `crouzeixraviart.py`.

```

from swiginac import *
from SyFi import *

x = cvar.x; y = cvar.y; z = cvar.z # fetch some global variables

class CrouzeixRaviart(StandardFE):
    """
    Python implementation of the Crouzeix-Raviart element.
    The corresponding C++ implementation is in the
    file CrouzeixRaviart.cpp.
    """

    def __init__(self):
        """ Constructor """
        StandardFE.__init__(self)

    def compute_basis_functions(self):
        """
        Compute the basis functions and degrees of freedom
        and put them in Ns and dofs, respectively.
        """
        polyspace = bernstein(1,triangle,"a")
        N = polyspace.eval()[0]
        variables = polyspace.eval()[1]

        for i in range(1,4):
            line = triangle.line(i)

```

```

        dofi = line.integrate(toex(N))
        self.dofs.append(dofi)

    for i in range(1,4):
        equations = []
        for j in range(1,4):
            equations.append(self.dofs[j-1].eval() == dirac(i,j) )
        sub = lsolve(equations, variables)
        Ni = N.subs(sub)
        self.Ns.append(toex(Ni));

p0 = [0,0,0]; p1 = [1,0,0]; p2 = [0,1,0];

triangle = Triangle(toex(p0), toex(p1), toex(p2))

fe = CrouzeixRaviart()
fe.set(triangle)
fe.compute_basis_functions()
for i in range(1,fe.nbf()+1):
    print "N(%d)      = %i,    fe.N(i).eval()"
    print "grad(N(%d)) = %i,    grad(fe.N(i)).eval()"
    print "dof(%d)     = %i,    fe.dof(i).eval()"

```

References

- [1] S. C. Brenner and L. R. Scott. *The mathematical theory of finite element methods*. Springer Verlag, 1994.
- [2] F. Brezzi and M. Fortin. *Mixed and hybrid finite element methods*. Springer Verlag, 1991.
- [3] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*. SIAM, 2002.
- [4] V. Girault and P.-A. Raviart. *Finite element methods for Navier–Stokes equations*. Springer Verlag, 1986.
- [5] H. P. Langtangen. *Computational Partial Differential Equations - Numerical Methods and Diffpack Programming*. Textbooks in Computational Science and Engineering. Springer, 2nd edition, 2003.