SyFi Tutorial

Kent-Andre Mardal

June 13, 2006

Contents

1 Introduction **2**

2 Software **4**
- 2.1 License . . . 4
- 2.2 Installation . . . 4
- 2.3 Python Support . . . 5

3 A Finite Element **5**
- 3.1 Basic Concepts . . . 5
- 3.2 Polygons . . . 6
  - 3.2.1 Line . . . 6
  - 3.2.2 Triangle . . . 7
  - 3.2.3 Tetrahedron . . . 9
  - 3.2.4 Rectangle . . . 11
  - 3.2.5 Box . . . 13
- 3.3 Polynomial Spaces . . . 14
  - 3.3.1 Bernstein Polynomials . . . 16
  - 3.3.2 Legendre Polynomials . . . 17
  - 3.3.3 Homogeneous Polynomials . . . 18
- 3.4 A Finite Element . . . 18
- 3.5 Degrees of Freedom . . . 20

4 Some Examples of Finite Elements **25**
- 4.1 Finite Elements in H^1 . . . 25
  - 4.1.1 The Lagrangian Element . . . 25
  - 4.1.2 The Crouizex-Raviart Element . . . 26
- 4.2 Finite Elements in L^2 . . . 28
  - 4.2.1 The P_0 Element . . . 28
  - 4.2.2 The Discontinuous Lagrangian Element . . . 28
- 4.3 Finite Elements in $H(div)$ . . . 30
  - 4.3.1 The Raviart-Thomas Element . . . 30
- 4.4 Finite Elements in $H(curl)$ . . . 33
  - 4.4.1 The Nedelec Element . . . 33

5 Mixed Finite Elements **35**
- 5.1 The Taylor–Hood and the $\mathbb{P}_n^d - \mathbb{P}_{n-2}$ Elements for the Stokes problem . . . 35
- 5.2 The Mixed Crouizex-Raviart Element for the Stokes problem . . 36
- 5.3 The Mixed Raviart-Thomas Element for the Poisson Problem on Mixed Form . . . 36

6 Computing Element Matrices **37**
- 6.1 A Poisson Problem . . . 37
- 6.2 A Poisson Problem on Mixed Form . . . 40
- 6.3 A Stokes Problem . . . 41
- 6.4 A Nonlinear Convection Diffusion Problem . . . 42

7 Python Support **44**

8 Code Generation **46**
- 8.1 Basic Tools . . . 47
- 8.2 Debugging . . . 50

1 Introduction

The finite element package SyFi is a C++ library built on top of the symbolic math library GiNaC [2]. 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'th element.

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 employ GiNaC, but also to provide 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 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 and employing GiNaCs tools for code generation, we are able to write efficient and directly compilable C++ code for the computation of element matrices etc.

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=0; i< fe.nbf(); i++) {
    index.first = i;
    for (int j=0; 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 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 [4]. In addition, one should also install Swiginac [5], which is a Python interface to GiNaC created by using SWIG. More about the usage of the Python interface can be found in Section 7.

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 [7] or Ciarlet [9].

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), $L_i : V \to \mathbb{R}$, for $i = 1, \ldots, n$, where $n = \dim(V)$, that determines V uniquely.*

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

$$L_i(N_j) = \delta_{ij}, \tag{1}$$

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 $N(x,y) = a + bx + cy$. The degrees of freedom for a linear Lagrangian element are simply the pointvalues at the vertices, $\mathbf{x}_i$, $L_i(N_j) = N_j(\mathbf{x}_i)$. The degrees of freedom and (1) determined a_j, b_j, and c_j for each basis function N_j. For instance N_1, which is on the form $a_1 + b_1x + c_1y$, is determined by,*

$$L_i(N_1) = N_1(\mathbf{x}_i) = \delta_{i1},$$

or written out as a system of linear equations,

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

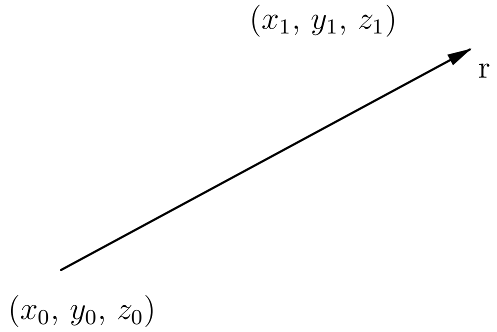
Hence,

$$N_1 = 1 - x - y.$$

The functions N_2 and N_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.

Figure 1: A line.



3.2 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, tetrahedra, and orthogonal rectangles and boxes. These basic components will be briefly described in this section.

3.2.1 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 3.2.1,

$$
\begin{aligned}
x &= x_0 + a\,t, & (2)\\
y &= y_0 + b\,t, & (3)\\
z &= z_0 + c\,t, & (4)\\
t &\in [0, 1], & (5)
\end{aligned}
$$

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,

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

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`),

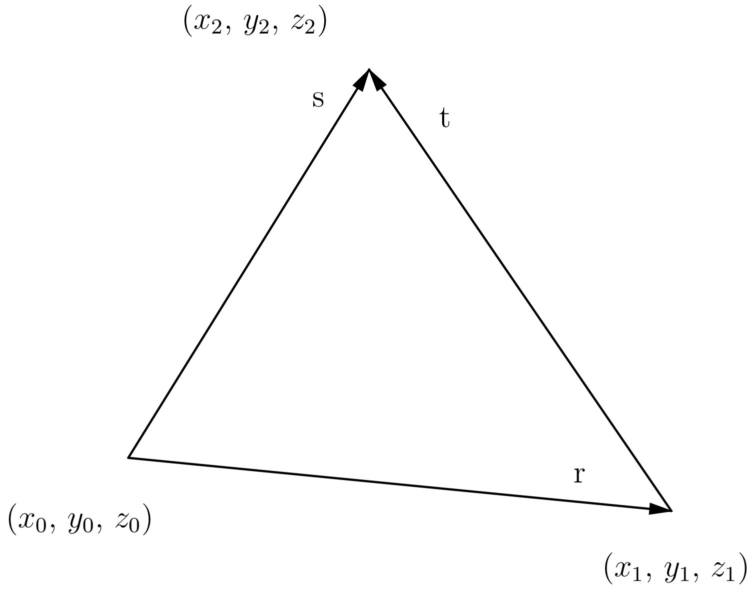
```
  ex p0 = lst(0.0,0.0,0.0);
  ex p1 = lst(1.0,1.0,1.0);

  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.

3.2.2 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

Figure 2: Triangle



as

$$
\begin{aligned}
x &= x_0 + ar + bs, &(7)\\
y &= y_0 + cr + ds, &(8)\\
z &= z_0 + er + fs, &(9)\\
s &\in [0, 1-r], &(10)\\
r &\in [0, 1], &(11)
\end{aligned}
$$

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

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");
```

3.2.3 Tetrahedron

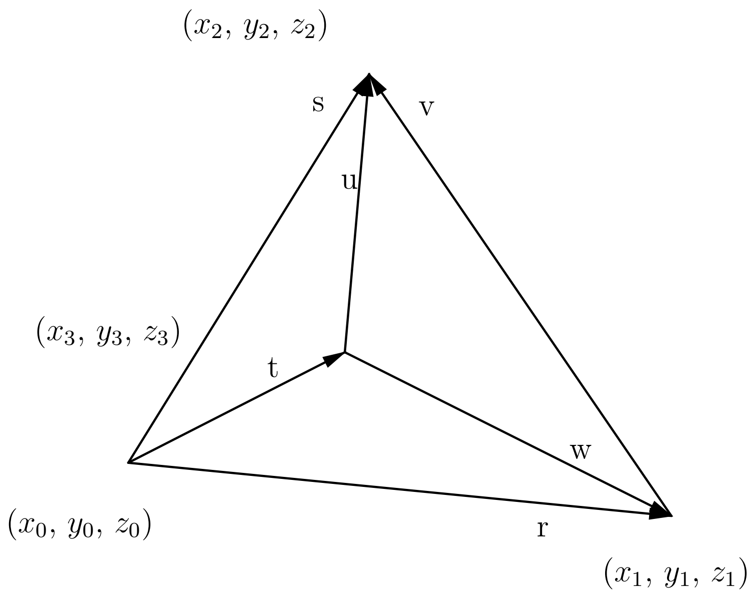
A tetrahedron is defined by four points $\mathbf{x}_0$, $\mathbf{x}_1$, $\mathbf{x}_2$, and $\mathbf{x}_3$. Associated with a tetrahedron are four triangles and six lines. The convention used so far is that

- 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,

$$
\begin{aligned}
x &= x_0 + ar + bs + ct, && (12)\\
y &= y_0 + dr + es + ft, && (13)\\
z &= z_0 + gr + hs + kt, && (14)\\
t &\in [0, 1 - r - s], && (15)\\
s &\in [0, 1 - r], && (16)\\
r &\in [0, 1], && (17)
\end{aligned}
$$

Figure 3: A tetrahedron.



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,

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

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");
```

3.2.4 Rectangle

The rectangles currently supported by SyFi are orthogonal. Such a rectangle is defined in terms of two points $\mathbf{x}_0$ and $\mathbf{x}_1$, as shown in Figure 4.

The rectangle can be represented as

$$\begin{aligned}
x &= x_0 + ar, && (18)\\
y &= y_0 + bs, && (19)\\
z &= z_0 + ct, && (20)\\
r &\in [0,1], && (21)\\
s &\in [0,1], && (22)\\
t &\in [0,1], && (23)
\end{aligned}$$

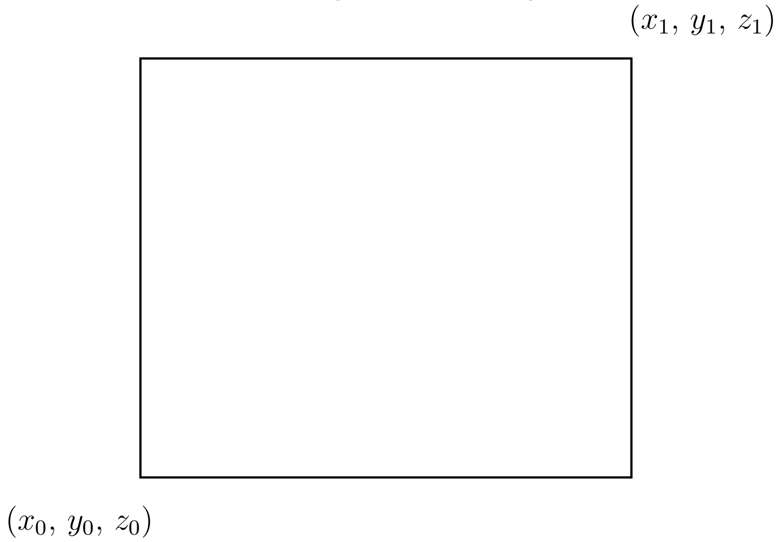
where $a = x_1 - x_0$, $b = y_1 - y_0$, and $c = z_1 - z_0$. Notice that either a, b, or c needs to be zero, or else (18)-(23) defines a box (which will be described later).

As earlier, integration is performed with substitution,

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

where $D = ab$ if $c = 0$, $D = bc$, if $a = 0$, and $D = ac$, if $b = 0$.

Figure 4: A rectangle.



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

```
class Rectangle : public Polygon {
  public:
  Rectangle(GiNaC::ex p0, GiNaC::ex p1, string subscript = "");
  Rectangle() {}
  virtual ~Rectangle(){}

  virtual int no_vertices();
  virtual GiNaC::ex vertex(int i);
  virtual Line line(int i);
  virtual GiNaC::ex repr(Repr_format format = SUBS_PERFORMED);
  virtual string str();
  virtual GiNaC::ex integrate(GiNaC::ex f);
};
```

As described with the previous polygons, the function `repr` returns a list with the items (18)-(23). The basic usage of the rectangle is as follows (see `rectangle_ex1.cpp`),

```
  ex f = x*y;

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

  Rectangle rectangle(p0,p1);

  ex repr = rectangle.repr();
  cout <<"s.repr "<<repr<<endl;

  ex intf = rectangle.integrate(f);
  cout <<"intf "<<intf<<endl;

  ex f2 = (x+1)*y*z;
  p0 = lst(0.0,0.0,1.0);
```

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

Rectangle rectangle2(p0,p1);

ex repr2 = rectangle2.repr();
cout <<"s2.repr "<<repr2<<endl;

ex intf2 = rectangle2.integrate(f2);
cout <<"intf2 "<<intf2<<endl;
```

3.2.5 Box

Currently, SyFi only supports orthogonal boxes (as was also the case with rectangles). Such a box is defined in terms of two points $\mathbf{x}_0$ and $\mathbf{x}_1$, as can be seen in Figure 5. The box can be represented as

$$
\begin{aligned}
x &= x_0 + ar, && (24)\\
y &= y_0 + bs, && (25)\\
z &= z_0 + ct, && (26)\\
r &\in [0,1], && (27)\\
s &\in [0,1], && (28)\\
t &\in [0,1], && (29)
\end{aligned}
$$

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

As earlier, integration is performed with substitution,

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

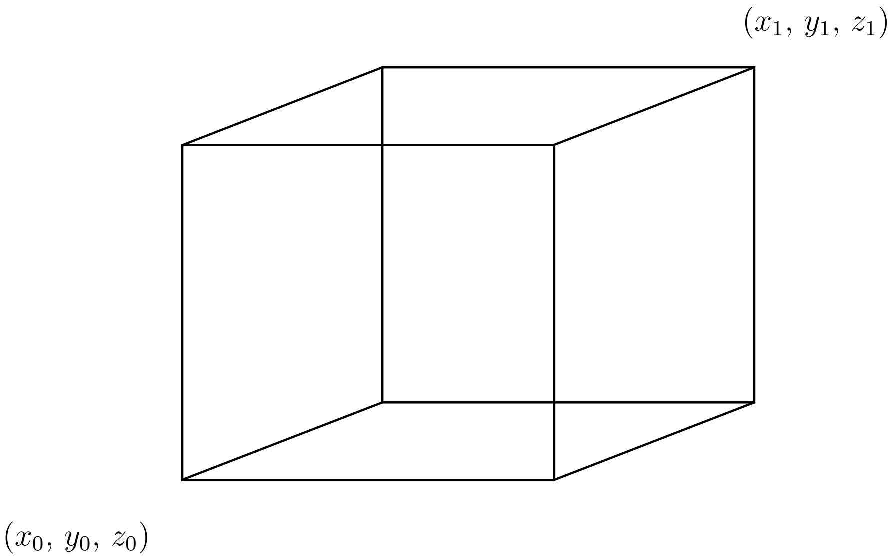
where $D = abc$.

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

```
class Box: public Polygon {
  public:
  Box(GiNaC::ex p0, GiNaC::ex p1, string subscript = "");
  Box(){}
  virtual ~Box(){}

  virtual int no_vertices();
  virtual GiNaC::ex vertex(int i);
  virtual Line line(int i);
  virtual GiNaC::ex repr(Repr_format format = SUBS_PERFORMED);
  virtual string str();
  virtual GiNaC::ex integrate(GiNaC::ex f);
};
```

The `repr` function returns a list of the definition of a orthogonal box in (24)-(29). A box can be used as follows (see `box_ex1.cpp`),

Figure 5: A Box.




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

Box box(p0,p1);

ex repr = box.repr();
cout <<"b.repr "<<repr<<endl;

ex intf = box.integrate(f);
cout <<"intf "<<intf<<endl;
```

Finally, we also mention that in addition to the above mentioned classes, `Line`, `Triangle`, `Tetrahedron`, `Rectangle`, and `Box`, we have implemented the corresponding reference geometries in the subclasses `ReferenceLine`, `ReferenceTriangle`, `ReferenceTetrahedron`, `ReferenceRectangle`, and `ReferenceBox`.

3.3 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) $\{1, x, x^2, \ldots, x^n\}$. This representation is not suitable for polynomials of high degree[1].

[1] In that case, one should use the Bernstein or Legendre polynomials.

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

$$v = a_0 + a_1 x + \ldots a_n x^n = \sum_{i=0}^{n} a_i x^i \tag{30}$$

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

$$v = \sum_{i,j=0}^{i+j<=n} a_{ij} x^i y^j \tag{31}$$

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

$$v = \sum_{i,j=0}^{i,j<=n} a_{ij} x^i y^j \tag{32}$$

The corresponding polynomial spaces in 3D are completely analogous.

Software Component: Polynomial Space The following functions generate symbolic expressions for the above polynomial spaces (30), (31), and (32), 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,

1. The polynom, e.g., $a_0 + a_1 x + \ldots + a_n x^n$ in 1D.
2. A list of variables, e.g., $\{a_0, a_1, \ldots, a_n\}$ in 1D.
3. A list containing the basis, e.g., $\{1, x, \ldots, x^n\}$ 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(lst(0,0), lst(1,0), lst(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");
```

3.3.1 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 polygons[2] etc.

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

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

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,

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

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: Bernstein polynomials The following functions generate 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);
```

[2] This will be used in the definition of the Raviart-Thomas element.

These functions return lists that are analogous to the lists made by the functions pol and polv described on page 15.

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 = "<<intp<<endl;
EQUAL_OR_DIE(intp2, "1/12*a3+1/12*a2+1/12*a1+1/12*a5+1/12*a0+1/12*a4");
```

3.3.2 Legendre Polynomials

A popular polynomial basis for polygons that are either rectangles or boxes are the Legendre polynomials. This polynomial basis is also usable to represent polynomials of high degree. The basis is defined on the interval $[-1, 1]$, as

$$L_k(x) = \frac{1}{2^k k!}\frac{d^k}{dx^k}(x^2-1), \quad k = 0, 1, \dots,$$

A nice feature with these polynomials is that they are orthogonal with respect to the L_2 inner product, i.e.,

$$\int_{-1}^{1} L_k(x)L_l(x)\,dx = \begin{cases} \frac{2}{2k+1}, & k = l, \\ 0, & k \neq l, \end{cases}$$

The Legendre polynomials are extended to 2D and 3D simply by taking the tensor product,

$$L_{k,l,m}(x, y, z) = L_k(x)L_l(y)L_m(z).$$

and $\mathbb{P}^n$ is defined by functions on the form (in 3D),

$$v = \sum_{k,l,m=0}^{k,l,m<=n} a_{k,l,m}L_{k,l,m}.$$

Software Components: Legendre polynomials The following functions generate symbolic expressions for $\mathbb{P}^n$ using the Legendre basis,

```
// generates a Legendre polynom of arbitrary order
GiNaC::ex legendre(int order, int nsd, const string a);
// generates a Legendre vector polynom of arbitrary order
GiNaC::lst legendrev(int no_fields, int order, int nsd, const string a);
```

These functions return lists that are analogous to the lists made by the functions pol and polv described on page 15.

The following code demonstrates the use of the Legendre polynomials, and (when runned) that the basis is orthogonal (see also legendre.cpp).

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

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

ex p = polynom_space.op(0);
cout <<"polynom p = "<<p<<endl;

ex dpdx = diff(p,x);
cout <<"dpdx = "<<dpdx<<endl;

ex p0 = lst(-1,-1);
ex p1 = lst(1,1);

Rectangle rectangle(p0,p1) ;
ex basis = polynom_space.op(2);
for (int i=0; i< basis.nops(); i++) {
    cout <<"i "<<i<<endl;
    ex integrand = p*basis.op(i);
    ex ai = rectangle.integrate(integrand);
    cout <<"ai "<<ai<<endl;
}
```

3.3.3 Homogeneous Polynomials

Another set of polynomials which sometimes are useful are the set of homogeneous polynomials $\mathbb{H}^n$. These are polynomials where all terms have the same degree. $\mathbb{H}^n$ is in 2D spanned by polynomials on the form:

$$v = \sum_{\substack{i, j, \\ i+j=n}} a_{i,j,k} x^i y^j$$

Software Components: Homogeneous polynomials The following functions generate symbolic expressions for $\mathbb{H}^n$,

```
// generates a homogeneous polynom of arbitrary order
GiNaC::ex homogenous_pol(int order, int nsd, const string a);
// generates a homogenous vector polynom of arbitrary order
GiNaC::lst homogenous_polv(int no_fields, int order, int nsd, const string a);
```

The use of these polynomials are similar to the other polynomials described earlier.

3.4 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` where Lagrangian elements are used),

```
  ex Ni;
  ex gradNi;
  ex dofi;
  for (int i=0; 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 you run `fe_ex1`, 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(lst(0,0), lst(1,0), lst(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=0; i< fe.nbf(); i++) {
  index.first = i;
  for (int j=0; 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 4.1.1.

3.5 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, besides defining a basis for the polynomial space, 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. This mapping does in turn provide the mapping between the element matrices/vectors and the global matrix/vector. Hence, we have the following mapping,

$$(e, i) \to L_i^e \to L_j \to j. \tag{33}$$

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,

$$j \to L_j \to \{L_{i(e)}^e\}_{e\in E(j)} \to \{(e, i(e))\}_{e\in E(j)}. \tag{34}$$

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

Software Component: Degrees of Freedom Handler 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 handling the degrees of freedom, we only need to provide the mappings (33) and (34). We have implemented a class `Dof` which 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)

  // (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<e1, 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 to be used when the dofs have been set.
  // 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);
  void clear();
};
```

Here, the function `int insert_dof(int e, int i, ex Li)` creates the various mappings between the local dof L_i^e, 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 Lagrangian elements, implemented by the class `LagrangeFE` (The description of `LagrangeFE` is postponed until Section `sec:fem:examples`), 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(lst(0,0), lst(1,0), lst(0,1));
  Triangle t2(lst(1,1), lst(1,0), lst(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=0; 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=0; 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(lst(0,0), lst(1,0), lst(0,1));
  Triangle t2(lst(1,1), lst(1,0), lst(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
}
```

Software Component: Degrees of Freedom Handler Template We will also describe an equally general degree of freedom handler which is not based on GiNaC, but which employs templates instead. This template class relies on two classes, the degree of freedom `D` and a comparison function. The rest is basically identical to the previously described `Dof`, except that we have added two boolean variables which can be used to turn off the computation of the global to local mapping in (34) and the $j \to N_j$ mapping. This class can be found in the header file `DofT.h`:

```
template <class D, class C>
class DofT {
  protected:
  bool create_index2dof, create_dof2loc;
  int counter;
  // the structures loc2dof, dof2index, and doc2loc  are completely dynamic
  // they are all initialized and updated by insert_dof(int e, int i, ex Li).

  // (int e, int i) ->  int j
  map<pair<int,int>, int>          loc2dof;
  // (ex Lj)    -> int j
  map<D,int,C>                     dof2index;
  typename map<D,int,C>:: iterator iter;

  // (int j)        -> ex Lj
  map<int,D>                     index2dof;
  // (ex j)    -> vector< pair<e1, i1>, ..  pair<en, in> >
  map <int, vector<pair<int,int> > >  dof2loc;

  public:
  DofT( bool create_index2dof_ = false, bool create_dof2loc_ = false ) {
    counter = -1;
    create_index2dof = create_index2dof_;
    create_dof2loc   = create_dof2loc;
  }
  ~DofT() {}
  int insert_dof(int e, int i, D Li);  // to update the internal structures

  // Helper functions to be used when the dofs have been set.
  // These do not modify the internal structure.
  int glob_dof(int e, int i);
  int glob_dof(D Lj);
  D  glob_dof(int j);
  int size();
  vector<pair<int, int> > glob2loc(int j);
  void clear();
};
```

The typical way to represent most common degrees of freedom is as points. Hence, we have implemented a simple point class `ptv` and its comparison function. The header file (see also `ptv.h`) is as follows:

```
class ptv {
private:
  int dim;
  double* v;
  static double tol;

public:
  ptv(int size_);
  ptv(int size_, double* v_);
  ptv(const ptv& p);
  ptv();

  virtual ~ptv();

  const int size() const;

  const double& operator [] (int i) const;
  double& operator [] (int i);
  ptv& operator = (const ptv& p);

  bool is_less(const ptv& p) const;

};

struct ptv_is_less : public std::binary_function<ptv, ptv, bool> {
bool operator() (const ptv &lh, const ptv &rh) const { return lh.is_less(rh);  }
};

std::ostream & operator<< ( std::ostream& os, const ptv& p);
```

The `ptv` class simply contain an array of doubles with variable size. The comparison function should check whether a point $x \in \mathbb{R}^n$ is less than $y \in \mathbb{R}^m$, which is not necessarily obvious how to do. For instance, which is the smallest of $x_1 = (1, 0) \ \in \ \mathbb{R}^2$, $x_2 = (0, 1) \ \in \mathbb{R}^2$ and $x_3 = (0, 0, 1) \ \in \mathbb{R}^3$? There are many possible ways to compare points. The convention we have chosen so far is to first check the size of the points. Hence, $x < y$, where $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$, if $n < m$. If $n = m$, then $x < y$ if $x_0 < y_0$. If $x_0 = y_0$, then $x < y$ if $x_1 < y_1$ and we continue in this fashion, if $x_j = y_j, 0 \leq j < i$ then $x < y$ if $x_i < y_j$. Notice that this comparison operator only affects the ordering of the degrees of freedom internally in the STL `map` structure. But it might be that other comparison conventions will speed up the search and insert routines in `map`.

Finally, we remark that the `ptv` class and `DofT` can be used also for degrees of freedom associated with lines, edges, faces or general polygons. For instance the edge of a 2D triangle, between the points $\mathbf{x}_0 = (x_0, y_0)$ and $\mathbf{x}_1 = (x_1, y_1)$ can be represented as a point in $\mathbb{R}^4$, e.g., (x_0, y_0, x_1, y_1) if $\mathbf{x}_0 < \mathbf{x}_1$ and (x_1, y_1, x_0, y_0) otherwise. Another simpler approach is to represent an edge by its midpoint.

4 Some Examples of Finite Elements

Earlier in Section 3.4, we described the usage of a general finite element. In this section we will show how various finite elements are constructed/implemented in SyFi.

4.1 Finite Elements in H^1

4.1.1 The Lagrangian Element

We will describe the construction of a Lagrangian element on a 2D triangle. The actual implementation of the element in both 1D, 2D and 3D can be found in the class `LagrangeFE`.

As we saw in Section 3.3, the polynomial space $\mathbb{P}_n$ in 2D can be written on the form

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

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

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

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.,

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

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. When this is done, it is simply a matter of solving the linear system

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

for each basis function N_k.

Software Component: The Lagrangian Element The Lagrangian element is implemented as a subclass of `StandardFE`. The class definition is:

```
class LagrangeFE : public StandardFE {
  public:
  LagrangeFE() {}
  virtual ~LagrangeFE() {}

  virtual void set(int order);
  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);
};
```

The Construction of the Lagrangian Element The Lagrangian element of arbitrary order in 1D, 2D, and 3D, is implemented in `LagrangeFE.cpp`. The following code is taken from `fe_ex3.cpp`.

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

// the polynomial spaces on the form:
// first item:     a0 + a1*x + a2*y + a3*x^2 + a4*x*y ...     the polynom
// 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`.

4.1.2 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 Crouzeix-Raviart class definition is similar to class defined for the Lagrangian element:

```
class CrouzeixRaviart : public StandardFE {
  public:
  CrouzeixRaviart();
  virtual ~CrouzeixRaviart() {}

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

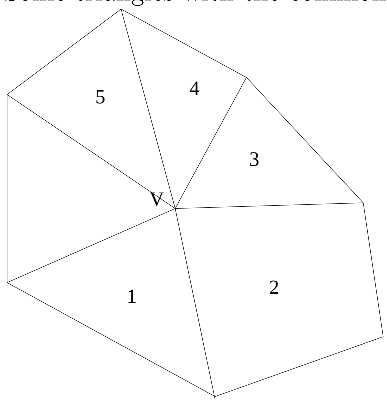
The Construction of the Crouzeix-Raviart Element The following code, which is from the file `CrouzeixRaviart.cpp`, shows how this element can be defined in 2D. The definition of the element in 3D can also be found in this file.

```
    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);
```

Figure 6: Some triangles with the common vertex V.




```
fe.set(p);
fe.compute_basis_functions();
for (int i=0; i< fe.nbf(); i++) {
  cout <<"fe.N("<<i<<")="<<fe.N(i)<<endl;
}
```

See also the Python implementation of this element in Section 7.

4.2 Finite Elements in L^2

4.2.1 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.

4.2.2 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 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 6 we see that the triangles $1, \ldots, 5$ all share the common vertex V. For continuous Lagrangian elements, this means that there will be only 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 depictured in Figure 6, 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=0; 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=0; 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.

4.3 Finite Elements in $H(div)$

4.3.1 The Raviart-Thomas 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

$$\mathbb{P}_n^d + \mathbf{x}\mathbb{P}_n. \tag{35}$$

And the degrees of freedom are,

$$\int_{e_i} \mathbf{v} \cdot \mathbf{n}\, p_k \, ds, \qquad \forall p_k \in \mathbb{P}_k(e_i), \tag{36}$$

$$\int_T \mathbf{v} \cdot \mathbf{p}_{k-1} \, dx, \qquad \forall p_{k-1} \in \mathbb{P}_{k-1}^d(T), \tag{37}$$

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 4.1.2. 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 (35), and the polynomial spaces on faces or edges of the polygon, as in (36). Both concepts will be dealt with below.

Software Component: The Raviart-Thomas Element Notice that for the previously defined Lagrangian and Crouizex-Raviart elements, the basis functions were scalar functions. The basis functions of the Raviart-Thomas elements are vector functions, but still, thanks to the general `ex` class, the Raviart-Thomas element class can be defined in the same way as earlier. The class definition is:

```
class RaviartThomas : public StandardFE {
  public:
  RaviartThomas() {}
  virtual ~RaviartThomas() {}

  virtual void set(int order);
  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);
};
```

The Construction of the Raviart-Thomas Element First, we described how to make the polynomial space (35). The polynomial spaces, $\mathbb{P}_n(T)$ and $\mathbb{P}^d_n(T)$ on a polygonal domain, can be made by the functions `bernstein` and `bernsteinv`, respectively. However, we can not just add the spaces $\mathbb{P}^d_n(T)$ 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 $\mathbb{P}^d_1(T)$, 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 $\mathbb{P}^d_n(T)$ 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 $\mathbb{P}^d_n(T)$ 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 (36)-(37). 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,

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

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`.

4.4 Finite Elements in $H(curl)$

4.4.1 The Nedelec Element

In electromagnetic applications, the family of Nedelec elements are very common. As was also the case with the Raviart-Thomas elements, $\mathbb{P}^n$ is not the most convenient space to define the basis functions. Instead, we will use

$$\mathbb{P}^d_{n-1} + \hat{\mathbb{H}}^k, \tag{38}$$

where

$$\hat{\mathbb{H}}^k = \{\mathbf{h} \in \mathbb{H}^d_k : \mathbf{h} \cdot \mathbf{x} = 0\}$$

and $\mathbb{H}$ is the space of homogenous polynomials described in Section 3.3.3. The degrees of freedom that defines the Nedelec elements are (in 2D),

$$\int_e \mathbf{t} \cdot \mathbf{u} p \, dx, \quad \forall p \in \mathbb{P}_{k-1}(e), \tag{39}$$

$$\int_T \mathbf{u} \cdot \mathbf{p} \, dx, \quad \forall \mathbf{p} \in \mathbb{P}^n_{k-2}(T). \tag{40}$$

Software Component: The Nedelec Element The Nedelec element class definition is similar to the previous element definitions.

```
class Nedelec : public StandardFE {
  public:
  Nedelec() {}
  virtual ~Nedelec() {}

  virtual void set(int order);
  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);
};
```

The Construction of the Nedelec Element The Nedelec element of arbitrary order in both 2D and 3D is implemented in `Nedelec.cpp`. Here we will for simplicity describe how the element is implemented in 2D.

We first consider the polynomial space (38),

```
// create r
GiNaC::ex R_k = homogenous_polv(2,k+1, 2, "a");
GiNaC::ex R_k_x = R_k.op(0).op(0);
GiNaC::ex R_k_y = R_k.op(0).op(1);


// Equations that make sure that r*x = 0
GiNaC::ex rx = (R_k_x*x + R_k_y*y).expand();
ex_ex_map pol_map = pol2basisandcoeff(rx);
ex_ex_it iter;
for (iter = pol_map.begin(); iter != pol_map.end(); iter++) {
  if ((*iter).second != 0 ) {
    equations.append((*iter).second == 0 );
    removed_dofs++;
  }
}
```

The degree of freedom associated with the edges (39) are implemented as,

```
GiNaC::ex dofi;
// dofs related to edges
for (int i=1; i<= 3; i++) {
  Line line = triangle.line(i);
  GiNaC::lst tangent_vec = tangent(triangle, i);
  GiNaC::ex bernstein_pol = bernstein(order, line, istr("a",i));
  GiNaC::ex basis_space = bernstein_pol.op(2);
  GiNaC::ex pspace_t = inner(pspace, tangent_vec);

  GiNaC::ex basis;
  for (int j=0; j< basis_space.nops(); j++) {
    counter++;
    basis = basis_space.op(j);
    GiNaC::ex integrand = pspace_t*basis;
    dofi =  line.integrate(integrand);
    dofs.insert(dofs.end(), dofi);
    GiNaC::ex eq = dofi == GiNaC::numeric(0);
    equations.append(eq);
  }
}
```

The degree of freedom associated with whole triangle (40) are implemented as,

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

5 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 [8] and Girault and Raviart[10].

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. The elements themselves and their implementation were discussed in the previous section.

5.1 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.

5.2 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 4.1.2, while the P_0 element was described in Section 4.2.1.

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();
```

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

The velocity element is the Raviart-Thomas element described in Section 4.3.1. The pressure element is discontinuous polynomials of degree n. The $\mathbb{P}_0$ element is described in Section 4.2.1, while the discontinuous $\mathbb{P}_n$ element is described in Section 4.2.2.

The can be made as such (see also `raviartthomas_ex2`):

```
int order = 3;

ReferenceTriangle triangle("t");
RaviartThomas vfe;
vfe.set(triangle);
vfe.set(order);
vfe.compute_basis_functions();

DiscontinuousLagrangeFE pfe;
pfe.set(triangle);
pfe.set(order);
pfe.compute_basis_functions();

for (int i=0; i< vfe.nbf(); i++)
  cout <<"vfe.N("<<i<<")="<<vfe.N(i)<<endl;
```

```
for (int i=0; i< pfe.nbf(); i++)
  cout <<"pfe.N("<<i<<")="<<pfe.N(i)<<endl;
```

6 Computing Element Matrices

Our next task is to compute element matrices. As earlier, everything will be done symbolically. 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, $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n$, 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 LaTeX format, for easy reading.
- In Section 8 we generate C++ code from the exactly computed element matrices.

6.1 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,

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

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 and Scott [7], Ciarlet [9], or Langtangen [11],

$$A_{ij} = a(N_i, N_j) = \int_T \nabla N_j \cdot \nabla N_i \, dx. \tag{41}$$

The computation of (41) 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=0; 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=0; i< fe.nbf(); i++) {
    index.first = dof.glob_dof(fe.dof(i));       // fetch the global dof for Ni
    for (int j=0; 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(lst(0,0), lst(1,0), lst(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(lst(x0,y0), lst(x1,y1), lst(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 LaTeX 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).

$$
\begin{aligned}
A[1,1] = & \frac{1}{2}\frac{x_0^2|(-x_0+x_1)(y_2-y_0)-(-x_0+x_2)(y_1-y_0)|}{(-y_1x_2-x_0y_2+y_0x_2+y_2x_1+x_0y_1-y_0x_1)^2} \\
& -\frac{y_1y_0|(-x_0+x_1)(y_2-y_0)-(-x_0+x_2)(y_1-y_0)|}{(-y_1x_2-x_0y_2+y_0x_2+y_2x_1+x_0y_1-y_0x_1)^2} \\
& +\frac{1}{2}\frac{y_0^2|(-x_0+x_1)(y_2-y_0)-(-x_0+x_2)(y_1-y_0)|}{(-y_1x_2-x_0y_2+y_0x_2+y_2x_1+x_0y_1-y_0x_1)^2} \\
& -\frac{x_0|(-x_0+x_1)(y_2-y_0)-(-x_0+x_2)(y_1-y_0)|x_1}{(-y_1x_2-x_0y_2+y_0x_2+y_2x_1+x_0y_1-y_0x_1)^2} \\
& +\frac{1}{2}\frac{|(-x_0+x_1)(y_2-y_0)-(-x_0+x_2)(y_1-y_0)|x_1^2}{(-y_1x_2-x_0y_2+y_0x_2+y_2x_1+x_0y_1-y_0x_1)^2} \\
& +\frac{1}{2}\frac{y_1^2|(-x_0+x_1)(y_2-y_0)-(-x_0+x_2)(y_1-y_0)|}{(-y_1x_2-x_0y_2+y_0x_2+y_2x_1+x_0y_1-y_0x_1)^2}
\end{aligned}
$$

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.

6.2 A Poisson Problem on Mixed Form

The Poisson problem can also be written on mixed form,

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

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

$$
\begin{aligned}
a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) &= \mathbf{G}(\mathbf{v}), \quad \forall \mathbf{v} \in \mathbf{V}_0, && (42) \\
b(\mathbf{u}, q) &= F(q), \quad \forall q \in Q, && (43)
\end{aligned}
$$

where

$$
\begin{aligned}
a(\mathbf{u}, \mathbf{v}) &= \int_\Omega \mathbf{u} \cdot \mathbf{v}\, dx, && (44) \\
b(\mathbf{u}, q) &= \int_\Omega \nabla \cdot \mathbf{u}\, q\, dx, && (45) \\
F(q) &= \int_\Omega f\, q\, dx, && (46) \\
\mathbf{G}(\mathbf{v}) &= \int_{\Omega_E} h\,\mathbf{n} \cdot \mathbf{v}\, ds && (47)
\end{aligned}
$$

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

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

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

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 mxpoisson_ex.cpp.

6.3 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

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

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

$$Q = \begin{cases} L^2_0 \text{ if } \partial\Omega_E = \partial\Omega, \\ L^2 \text{ else.} \end{cases}$$

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=0; i< v_fe.nbf(); i++) {
    dof.insert_dof(1,i,v_fe.dof(i));
  }
  for (int i=0; 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=0; i< v_fe.nbf(); i++) {
    index.first = dof.glob_dof(v_fe.dof(i));     // fetch the global dof for v_i
    for (int j=0; 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=0; 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;
    }
  }
}
```

6.4 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,

$$(\mathbf{u}\cdot\nabla)\mathbf{u} - \Delta\mathbf{u} = \mathbf{f}, \quad \text{in } \Omega, \tag{48}$$

$$\mathbf{u} = \mathbf{g}, \quad \text{on } \partial\Omega. \tag{49}$$

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=0; 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=0; k< fe.nbf(); k++)  {
    UU +=ujs.op(k)*fe.N(k);   // U += u_k N_k
  }

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

  for (int i=0; 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=0; j< fe.nbf() ; j++) {
      index.second = dof.glob_dof(fe.dof(j));  // fetch global dof
      symbol uj = ex_to<symbol>(ujs.op(j));  // 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 $A[1,1]$,

$$\begin{aligned} A[1,1] &= \frac{1}{2} + \frac{2}{105}u_3 + \frac{2}{105}u_7 + \frac{1}{21}u_2\frac{13}{420}u_1 - \frac{1}{280}u_{11} && (50)\\ &- \frac{1}{21}u_6 - \frac{1}{280}u_5\frac{1}{140}u_{10} + \frac{1}{210}u_9 - \frac{1}{140}u_4. && (51)\end{aligned}$$

We have used GiNaC to generate the LaTeXcode, as described on Page 39.

7 Python Support

SyFi now comes with Python support. The SyFi Python module 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 forth 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(0,fe.nbf()):
  for j in range(0,fe.nbf()):
    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 Crouizex-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.
    """
    polspace = bernstein(1,triangle,"a")
    N = polspace.eval()[0]
    variables = polspace.eval()[1]
```

```
    for i in range(0,3):
      line = triangle.line(i+1)
      dofi = line.integrate(toex(N))
      self.dofs.append(dofi)

    for i in range(0,3):
      equations = []
      for j in range(0,3):
        equations.append(self.dofs[j].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(0,fe.nbf()):
  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()
```

8 Code Generation

In this section we will describe some matrix factories created for the PyCC project [3], which have been made by using SyFi, GiNaC and Swiginac. At present, we have written ca. 1500 lines of Python code using SyFi, Swiginac etc., which have generated roughly 60 000 lines of C++ code for the computation (of various variants) of the mass matrix, the stiffness matrix, the convection matrix and the divergence matrix using Lagrangian elements of order 1-5 in 2D and 1-3 in 3D. Furthermore, the generated C++ code is efficient, since everything except the geometry mapping can be computed exactly. Notice also that although only Lagrangian elements have been used so far, most of the Python code that generated the C++ code is completely element independent. In addition to the generated C++ code we have also written about 1500 lines of code which loops over the cells of a Dolfin mesh [1] such that global matrices are made.

We have create two matrix factories. These are implemented in `MatrixFactory` and `MatrixFactory_highorder`. There are three differences between these two factories. The first difference is that `MatrixFactory` employs the numbering of degrees of freedom in the Dolfin mesh. Therefore, this `MatrixFactory` is limited to linear Lagrangian elements. On the other hand, `MatrixFactory_highorder` uses `DofT`, described in Section 3.5, which works for general elements. The second difference is that in `MatrixFactory` the integration is performed on a global element with global basis functions, e.g., for the stiffness matrix,

$$A_{ij} = \int_T \nabla N_i \nabla N_j dx. \tag{52}$$

In MatrixFactory_highorder, the integration is performed on the reference element with a geometry tensor G (the Jacobian of the geometry mapping) and $D = \det(G)$,

$$A_{ij} = \int_{\hat{T}} (G^{-T} \nabla \hat{N}_i) \cdot (G^{-T} \nabla \hat{N}_j) D \, dx. \tag{53}$$

which is the typical way to do it in finite element codes. At present, we favor (53) to (52) simply because it produces much smaller expressions and therefore faster code. However, the large expressions in (52) typically involve subexpressions repeated many times. Hence, it should be possible to postprocess these expressions to create smaller expressions and faster code. However, we have not done this yet. Finally, the third difference is that MatrixFactory_highorder works for the FastMatSparse matrix in PyCC, the Epetra matrix in Trilinos [6] and for STL maps of type map<pair<int,int>,double>.

8.1 Basic Tools

We will illustrate the code generation by considering what was done for the mass matrix in MatrixFactory_highorder.

The entries of a mass matrix are:

$$M_{kl} = \int_T N_k N_l \, dx = \int_{\hat{T}} \hat{N}_i \hat{N}_j D \, dx, ,$$

where T is the global polygon, N_k and N_l are the k'th and l'th global basis functions, respectively, $\hat{T}$ is the reference polygon, $\hat{N}_i$ and $\hat{N}_j$ are the i'th and j'th basis functions on the reference polygon corresponding to k and l, respectively, and D is the determinant of the Jacobian of the geometry mapping. The following code shows how this can be done (see also code_gen.py):

```
def create_A_string_mass(fe):
  A_str = "  double A[%d][%d];\n"%(fe.nbf(), fe.nbf())
  domain = fe.getPolygon()

  # loop over all N(i)
  for i in range(0,fe.nbf()):

    # loop over all N(j)
    for j in range(0,fe.nbf()):

      # compute the integrand N(i)*N(j)
      integrand = fe.N(i).eval()*fe.N(j).eval()

      # integrate over the domain
      Aij = domain.integrate(toex(integrand))

      # generate C string and append the string to the rest
      A_str += "  A[%d][%d]=(%s)*D;\n"%(i,j,Aij.eval().evalf().printc())
```

The following output is produced, when using linear element on a 2D triangle (see also matrix_factory_mass_2D.cc, which also contains code for higher order Lagrangian elements),

```
  double A[3][3];
  A[0][0]=(8.3333333333333329e-02)*D;
  A[0][1]=(4.1666666666666664e-02)*D;
```

```
  A[0][2]=(4.1666666666666664e-02)*D;
  A[1][0]=(4.1666666666666664e-02)*D;
  A[1][1]=(8.3333333333333329e-02)*D;
  A[1][2]=(4.1666666666666664e-02)*D;
  A[2][0]=(4.1666666666666664e-02)*D;
  A[2][1]=(4.1666666666666664e-02)*D;
  A[2][2]=(8.3333333333333329e-02)*D;
```

Hence, this is the mass element matrix on the reference element multiplied with D. In addition to computing the element matrix we also need to computed the global degrees of freedom and generate a C function. We will not go into details on this, but recommend the reader to have a look in `code_gen.py`.

The complete function for the computation of the element matrix, in the case of linear Lagrangian elements, and the insertion of the element matrix in the global matrix can be found in `matrix_factory_mass_2D.cc` is:

```
void matrix_factory_mass_2D_order1 (map<pair<int,int>,double>& matrix,
         DofT<ptv,ptv_is_less>& dof,
         int element, double pp0[2], double pp1[2], double pp2[2]){

  // geometry related stuff

  double x0 = pp0[0]; double y0 = pp0[1];
  double x1 = pp1[0]; double y1 = pp1[1];
  double x2 = pp2[0]; double y2 = pp2[1];

  double G00 = x1 - x0; double G01 = x2 - x0;
  double G10 = y1 - y0; double G11 = y2 - y0;

  double D = fabs(G00*G11-G01*G10);

  // inserting local dofs in the global dof handler (dof)

  int iidof[3];
  double dof1[2];
  dof1[0]=x0;  dof1[1]=y0;
  ptv pdof1(2,dof1);
  iidof[0] = dof.insert_dof(element,1,pdof1);

  double dof2[2];
  dof2[0]=G01+x0;  dof2[1]=y0+G11;
  ptv pdof2(2,dof2);
  iidof[1] = dof.insert_dof(element,2,pdof2);

  double dof3[2];
  dof3[0]=G00+x0;  dof3[1]=G10+y0;
  ptv pdof3(2,dof3);
  iidof[2] = dof.insert_dof(element,3,pdof3);


  // compute the element matrix

  double A[3][3];
  A[0][0]=(8.3333333333333329e-02)*D;
  A[0][1]=(4.1666666666666664e-02)*D;
  A[0][2]=(4.1666666666666664e-02)*D;
  A[1][0]=(4.1666666666666664e-02)*D;
  A[1][1]=(8.3333333333333329e-02)*D;
  A[1][2]=(4.1666666666666664e-02)*D;
  A[2][0]=(4.1666666666666664e-02)*D;
  A[2][1]=(4.1666666666666664e-02)*D;
  A[2][2]=(8.3333333333333329e-02)*D;


  // insert element matrix into global matrix
```

```
  int nbf = 3;
  pair<int,int> index;
  for (int i=0; i< nbf; i++) {
    index.first = iidof[i];
    for (int j=0; j< nbf; j++) {
      index.second = iidof[j];
      matrix[index] += A[i][j];
    }
  }
}
```

Finally, we show how the above function is used in PyCC to compute the mass matrix on a Dolfin mesh (see also `MatrixFactory_highorder.cpp`)

```
void MapMatrixFactory:: computeMassMatrix(){

  int e = -1;

  if (mesh->numSpaceDim() == 2) {

    double p0[2];
    double p1[2];
    double p2[2];

    for (CellIterator cell(*mesh); !cell.end(); ++cell) {

      e++;

      // Obtain vertices from Dolfin mesh
      Vertex& v0 = (*cell).vertex(0);
      Vertex& v1 = (*cell).vertex(1);
      Vertex& v2 = (*cell).vertex(2);


      // Create double arrays with the data from the vertices
      p0[0] = v0.coord().x;  p0[1] = v0.coord().y;
      p1[0] = v1.coord().x;  p1[1] = v1.coord().y;
      p2[0] = v2.coord().x;  p2[1] = v2.coord().y;


      switch(order1) {
        case 1 :
          matrix_factory_mass_2D_order1(*matrix,*idof,e,p0,p1,p2);
          break;
        case 2 :
          matrix_factory_mass_2D_order2(*matrix,*idof,e,p0,p1,p2);
          break;
        case 3 :
          matrix_factory_mass_2D_order3(*matrix,*idof,e,p0,p1,p2);
          break;
        case 4 :
          matrix_factory_mass_2D_order4(*matrix,*idof,e,p0,p1,p2);
          break;
        case 5 :
          matrix_factory_mass_2D_order5(*matrix,*idof,e,p0,p1,p2);
          break;
      }
    }
  }
}
```

Notice that this code works for Lagrangian elements of order 1-5 in 2D.

8.2 Debugging

Debugging finite element codes is often extremely hard, at least that is the authors' experience. This has been one of the reasons why we have chosen to employ a symbolic math engine behind the curtain in the first place.

One of the advantages of SyFi is that one obtain explicit symbolic expressions for all the basis functions (and its derivatives). Another good thing is that one can create global finite elements, that is finite elements that are not defined on reference geometries, and perform integration and differentiation on their geometries. For instance, when we created the divergence matrix factory we initially had a mysterious bug which took us several hours to find. To locate the bug, we computed the divergence element matrix on a global element with the vertices $\mathbf{x}_0 = (0.2, 0.2)$, $\mathbf{x}_1 = (0.4, 0.2)$, and $\mathbf{x}_2 = (0.1, 0.3)$, and compared it with the divergence element matrix on the reference element with the corresponding geometry tensor. To do this, we wrote the following code (see also `main_syfi.cpp`):

```
// create global triangle
lst p0(0.2, 0.2);
lst p1(0.4, 0.2);
lst p2(0.1, 0.3);
Triangle triangle(p0,p1,p2);

// create vector element for v on the global triangle
VectorLagrangeFE v_fe;
v_fe.set_size(2);
v_fe.set(vorder);
v_fe.set(triangle);
v_fe.compute_basis_functions();

// create scalar element for p on the global triangle
LagrangeFE p_fe;
p_fe.set(1);
p_fe.set(triangle);
p_fe.compute_basis_functions();


// compute global element matrix
map<pair<int,int>, ex> A;
pair<int,int> index;
for (int i=0; i< p_fe.nbf(); i++) {
  index.first = i;
  for (int j=0; j< v_fe.nbf(); j++) {
    index.second= j;
    ex divV= p_fe.N(i)*div(v_fe.N(j));
    ex Aij = triangle.integrate(divV);
    A[index] = Aij;
  }
}
```

The element matrix created by this code was then printed out and compared with the element matrix computed by the matrix factory on the same polygon (see `dolfin_main.cpp`). By comparing each entry of the two matrices we quickly found the (uninteresting) bug. Hence, in out experience it is extremely valuable to have the concrete basis functions etc. on global element, and being able to work with them both with a pen and a paper and the computer, to reveal what is going on.

References

[1] Dolfin, 2006. http://www.fenics.org/dolfin.

[2] GiNaC, 2006. http://www.ginac.de.

[3] PyCC, 2006. http://home.simula.no/ skavhaug/heart_simulations.html.

[4] SWIG, 2006. http://www.swig.org/.

[5] Swiginac, 2006. http://swiginac.berlios.de/.

[6] Trilinos, 2006. http://software.sandia.gov/trilinos.

[7] S. C. Brenner and L. R. Scott. *The mathematical theory of finite element methods*. Springer Verlag, 1994.

[8] F. Brezzi and M. Fortin. *Mixed and hybrid finite element methods*. Springer Verlag, 1991.

[9] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*. SIAM, 2002.

[10] V. Girault and P.-A. Raviart. *Finite element methods for Navier–Stokes equations*. Springer Verlag, 1986.

[11] H. P. Langtangen. *Computational Partial Differential Equations - Numerical Methods and Diffpack Programming*. Textbooks in Computational Science and Engineering. Springer, 2nd edition, 2003.