SyFi Tutorial

Kent-Andre Mardal

November 22, 2005

Contents

1 Introduction

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

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

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

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

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

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

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

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

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

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

Our motivation behind this project is threefold. First, we wish to make advanced finite element methods more readily available. We want to do this by implementing a variety of finite elements and functions for computing element matrices. Second, in our experience developing and debugging codes for the finite element method 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 it is reasonable that it is possible to generate efficient code by traversing the expressions as three structures (which GiNaC has support for). Alternatively, we can simply evaluate the basis functions, its derivatives, etc. in quadrature points and generate C++ code for these expressions. GiNaC has basic tools for code generation.

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

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

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

```
void compute_element_matrix(Polygon& T, int order) {
  std::map<std::pair<int,int>, ex> A;      // matrix of expression
  std::pair<int,int> index;                // index in matrix
  LagrangeFE fe;                           // Lagrangian element of any order
  fe.set(order);                           // set the order
  fe.set(domain);                          // set the polygon
  fe.compute_basis_functions();            // compute the basis functions
  for (int i=1; i<= fe.nbf() ; i++) {
    index.first = i;
    for (int j=1; j<= fe.nbf() ; j++) {
```

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

```
      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 Lagrangian elements and the weak form for the Poisson problem have actually been implemented (at least in a nice clean way, some other methods can be found in the sandbox, but these are not yet finished.).

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 rely 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.

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., Ciarlet [**?**].

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, $\{v_i\}_{i=1}^n$, we form the linear system of equations,

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

and solve it.

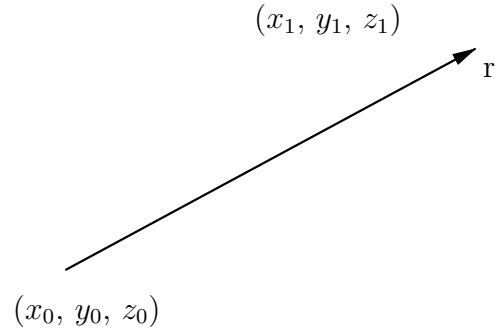
Example 3.1 (Linear Lagrangian element on the reference triangle) *Let T be the unit triangle with vertices $(0,0)$, $(1,0)$, and $(0,1)$. The polynomial space V consists of linear polynomials, i.e., polynomials on the form $a+bx+cy$. The degrees of freedom for this element are simply the pointvalues at the vertices, x_i, $L_i(v_j) = v_j(\mathbf{x}_i)$. The degrees of freedom and (1) determined a, b, and c for each basis function. For instance v_1, which is on the form $a_1 + b_1 x + c_1 y$, is determined by,*

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

or written out as a system of linear equations,

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

Figure 1: A line.



Hence,

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

The functions v_2 and v_3 can be determined similarly.

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

3.1.1 Polygons

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

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

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

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,

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

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);
}
```

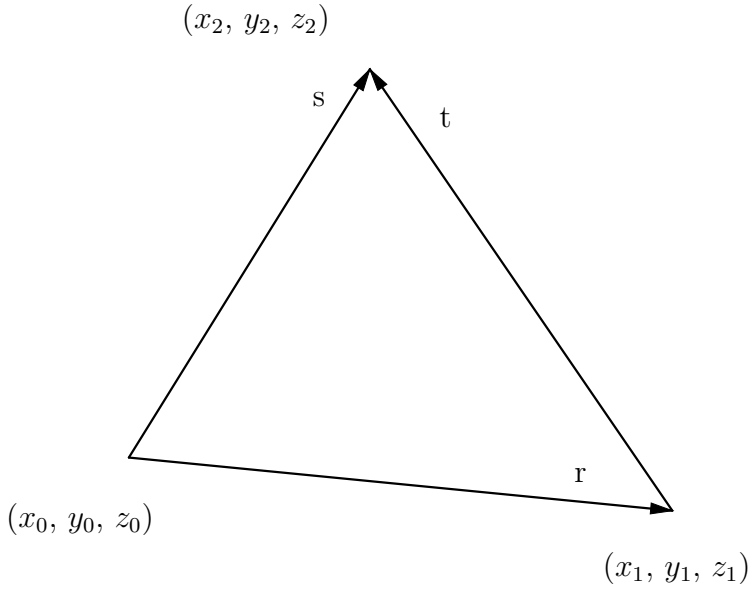
Most of the functions in this class are self-explanatory. However, the function `repr` deserves special attention. The function `repr` returns the mathematical definition of a line. To be precise, this function returns a list of expressions (`lst`), where the items are the items in (2)-(5) (see also 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 this is that our test examples also serve as regression tests for the package.

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

Figure 2: Triangle



represented 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");
```

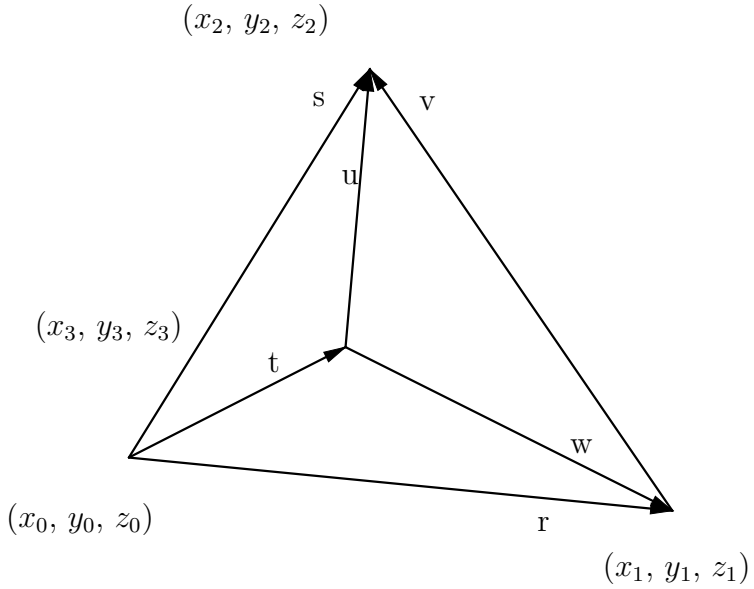
Tetrahedron A tetrahedron is defined by four points $\mathbf{x}_0$, $\mathbf{x}_1$, $\mathbf{x}_2$, and $\mathbf{x}_3$. Associated with a tetrahedron are four triangles and six lines. The convention used so far is that

- 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 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. So far we have used the perhaps visually nicest representation, 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[2]. However, we have chosen this basis in our initial version, because of its, as mentioned, nice representation. We will add the Bernstein basis in the next release.

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

$$v = a_0 + a_1 x + \ldots a_n x^n = \sum_{i=0}^{n} a_i x^i \tag{18}$$

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

$$v = \sum_{i,j=0}^{i+j<=n} a_{ij} x^i y^j \tag{19}$$

[2]In that case, one should use e.g., the Bernstein polynomials or the Legendre polynomials. These polynomials have not yet been implemented.

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

$$v = \sum_{i,j=0}^{i,j<=n} a_{ij} x^i y^j \tag{20}$$

The corresponding polynomial spaces in 3D are completely analogous.

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

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

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

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

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

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

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

ex p = pol(order,nsd, "a");
cout <<"polynomial 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 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 only has two interesting components, the basis functions $\{N_i\}$ and the corresponding degrees of freedom $\{L_i\}$. The basis function (and its derivatives) is used to compute the element matrices and element vectors, while the degrees of freedom are used to define the mapping between the element matrices and element vectors to the global matrix and global vector. As we see in the following, the observation that only these two components are needed leads 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);
  virtual ex N(int i);
  virtual ex dof(int i);
  virtual int nbf();
};
```

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

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

The computation of the element matrix in 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=1; i<= fe.nbf() ; i++) {
    index.first = i;
    for (int j=1; j<= fe.nbf() ; j++) {
      index.second = j;
      ex nabla = inner(grad(fe.N(i)), grad(fe.N(j)));
      ex Aij = T.integrate(nabla);
      A[index] = Aij;
    }
  }
```

The Construction of a Finite Element Finally, we will describe the construction of a Lagrangian element on a 2D triangle (the implementation of more general Lagrangian elements can be found in `LagrangeFE`). As we saw in Section 3.2, the polynomial space $\mathbb{P}^n$ in 2D is on the form

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

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

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

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

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

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

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

for each basis function v_k. This can be done as follows, (see `fe3_ex.cpp`),

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

// the polynomial spaces on the form:
//      a0 + a1*x + a2*y + a3*x^2 + a4*x*y ...
polynom = pol(order, 2, "b");
// the variables a0,a1,a2 ..
variables = coeffs(polynom);

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 N_j and all points xi.
// Each basis function N_j is determined by a set of linear equations:
//   N_j(x_i) = 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 x_i
    ex point = points.op(i-1);
    // The equation N_j(x_i) = dirac(i,j)
    ex eq = polynom == dirac(i,j);
    // The equation is appended 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 N_j
  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 `ex`es.

3.4 Degrees of Freedom

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

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

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

$$(e, i) \rightarrow L_i^e \rightarrow L_j \rightarrow j. \tag{21}$$

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

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

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

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

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

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

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

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

The basic usage of `Dof` is (from `dof_ex.cpp`)

```
Dof dof;

Triangle t1(lst(0.0,0.0), lst(1.0,0.0), lst(0.0,1.0));
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=1; i<= fe.nbf() ; i++) {
  cout <<"fe.dof("<<i<<")= "<<fe.dof(i)<<endl;
  // insert local dof in global set of dofs
  dof.insert_dof(1,i, fe.dof(i));
}

// Create a finite element and corresponding
// degrees of freedom on the second triangle
fe.set(t2);
fe.compute_basis_functions();
for (int i=1; i<= fe.nbf() ; i++) {
  cout <<"fe.dof("<<i<<")= "<<fe.dof(i)<<endl;
  // insert local dof in global set of dofs
  dof.insert_dof(2,i, fe.dof(i));
}

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

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

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

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

Our next example concerns degrees of freedom which are line integrals over the edges of triangles. Let T be a triangle with the edges e_i. The degree of freedom 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(0.0,0.0), lst(1.0,0.0), lst(0.0,1.0));
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
}
```

References

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