SyFi - An Element Matrix Factory, with Emphasis on the Incompressible Navier-Stokes Equations

Kent-Andre Mardal

Simula Research Laboratory, P.O. Box 134, NO-1325 Lysaker, Norway,
`kent-and@simula.no`,
WWW home page: `http://www.simula.no/portal_memberdata/kent-and`

Abstract. SyFi is an open source C++ library for defining and using variational forms and finite elements based on symbolic representations of polygonal domains, degrees of freedom and polynomial spaces. Once the finite elements and variational forms are defined, they are used to generate efficient C/C++ code.

1 Introduction

SyFi [11], which stands for Symbolic Finite Elements, is a C++ library for finite element computations. It relies on the symbolic mathematics library GiNaC [8] and the Python interface to GiNaC called Swiginac [13]. All the projects GiNaC, Swiginac and SyFi are open source projects. Similar to GiNaC, SyFi has a Python interface generated by using SWIG [12]. This paper is only a short overview of the SyFi project in the context of finite element methods for the incompressible Navier-Stokes equations. A more comprehensive description of the project can be found on its webpage http://syfi.sf.net, which contains a tutorial, a complete reference and the source code. We will show various code snippets in this paper, the complete code examples can be found in the subdirectory `para06` in the SyFi source code tree.

There are quite a few other projects that are similar in various respects to SyFi. Within the FEniCS [3] project there are two Python projects: FIAT [5] and FFC [4]. FIAT is a Python module for defining finite elements while FFC generates C++ code based on a high–level Python description of variational forms. The DSEL project [2] is a project which employs high–level C++ programming techniques such as expression templates and meta-programming for defining variational forms, performing automatic differentiation, interpolation and more. Sundance [10] is a C++ library with a powerful symbolic engine which supports automatic generation of discrete system for a given variational form. Analysa [1], GetDP [7], and FreeFem++ [6] define domain-specific languages for finite element computations. The main difference between SyFi and the other projects is that it uses a high level symbolic framework in Python to generate efficient C/C++ code.

A key point in the design of SyFi is, as already mentioned, that we want to employ symbolic mathematics and code generation in place of the numerics. The powerful symbolic engine GiNaC and the combination of the high–level languages C++ and Python have so far proven to be a solid platform. Consider for instance, the computation of, e.g., the mass element matrix:

$$A_{ij} = \int_T N_i N_j \, dx, \tag{1}$$

where T is a polygonal domain and N_i and N_j are finite element basis functions that are standard polynomials. For instance, in the case of a linear element defined on the reference triangle the basis functions are,

$$N_0 = 1 - y - x, \tag{2}$$
$$N_1 = x, \tag{3}$$
$$N_2 = y \tag{4}$$

Using SyFi, the computations of the integrals in (1) with, e.g., the basis functions (2)-(4) are carried out symbolically prior to the code generation. Needless to say, it is possible to generate more efficient code in this way than the traditional way which involves a loop over quadrature points and numerical evaluation of the finite element basis functions. As will be explained later, other advantages of this approach include an easy way of defining finite elements, and straightforward computation of the Jacobian in the case of nonlinear PDEs.

2 Using Finite Elements and Evaluating Variational Forms

One main goal with SyFi has been that it should be a tool with *strong support for differentiation and integration of polynomials on polygonal domains*, which are basic ingredients both when defining finite elements and using finite elements to define variational forms. Many finite elements have been implemented in SyFi. Of particular importance for the simulation of incompressible fluids are the continuous and discontinuous Lagrangian elements of arbitrary order and the Crouzeix-Raviart element. However, also the $H(\mathrm{div})$-Raviart-Thomas elements and the $H(\mathrm{curl})$- Nedelec elements of arbitrary order have been implemented. We will come back to the construction of finite elements in Section 4. In this section we concentrate on the usage of already implemented elements.

We construct the commonly used Taylor–Hood element $\mathbb{P}_2^2 - \mathbb{P}_1$ as follows (see also `div.py`),

```
from swiginac import *
from SyFi import *

polygon = ReferenceTriangle()

v_element = VectorLagrangeFE(polygon,2)
```

```
v_element.set_size(2)
v_element.compute_basis_functions()

p_element = LagrangeFE(polygon,1)
p_element.compute_basis_functions()
```

The polygonal domain here is a reference triangle, but it may be a line, a triangle, a square, a tetrahedron or a box. Furthermore, these geometries are not limited to typical reference geometries. For instance, we may construct the elements on a global triangle defined by the points (x_0, y_0), (x_1, y_1), and (x_2, y_2) where $x_0, \ldots, y_2$ might be both numbers and/or symbols. The following code shows the Taylor–Hood element on a triangle defined in terms of the symbols $x_0, \ldots, y_2$, (see also `div_global.py`),

```
x0 = symbol("x0"); y0 = symbol("y0")
x1 = symbol("x1"); y1 = symbol("y1")
x2 = symbol("x2"); y2 = symbol("y2")

p0 = [x0,y0]; p1 = [x1,y1]; p2 = [x2,y2]

polygon = Triangle(p0,p1,p2)
v_element = VectorLagrangeFE(polygon,2)
v_element.set_size(2)
v_element.compute_basis_functions()

p_element = LagrangeFE(polygon,1)
p_element.compute_basis_functions()
```

The computed basis functions are standard polynomials also in this case, although they depend on $x_0, \ldots, y_2$. These polynomials can be added, multiplied, differentiated, integrated etc. in the standard way (within a symbolic framework). Consider for example, the computation of the divergence constraint,

$$B_{ij} = \int_T \operatorname{div} \boldsymbol{N}_i \, L_j \, dx,$$

where $\boldsymbol{N}_i$ and L_j are the basis functions for the velocity and pressure elements, respectively, and T is a polygonal domain. This matrix can be computed as follows (see also `div_global.py`):

```
... construct the element

for i in range(0,v_element.nbf()):
    for j in range(0,p_element.nbf()):
        integrand = div(v_element.N(i))*p_element.N(j)
        Bij =  polygon.integrate(integrand)
```

Another example that demonstrates the power of this approach, in which we utilize a symbolic mathematics engine, is the computation of the Jacobian of the nonlinear convection-diffusion equations that typically appear in incompressible flow simulations. Let

$$\boldsymbol{F}_i = \int_T (\boldsymbol{u} \cdot \nabla \boldsymbol{u}) \cdot \boldsymbol{N}_i + \nabla \boldsymbol{u} : \nabla \boldsymbol{N}_i \, dx,$$

where $\boldsymbol{u} = \sum_k u_k \boldsymbol{N}_k$. Then,

$$\boldsymbol{J}_{ij} = \frac{\partial \boldsymbol{F}_i}{\partial u_j} = \frac{\partial}{\partial u_j} \int_T (\boldsymbol{u} \cdot \nabla \boldsymbol{u}) \cdot \boldsymbol{N}_i + \nabla \boldsymbol{u} : \nabla \boldsymbol{N}_i \, dx, \tag{5}$$

The computation of such Jacobian matrices and the implementation of corresponding simulations software are usually tedious and error-prone. It seems that one main reason for this difficulty is the gap between the computations done by hand and the corresponding numerical algorithm to be implemented. After all, the computation of (5) only involves straightforward operations. SyFi aims at closing this gap. We will now show the code for computing (5) with SyFi. The complete source code is in `conv-diffusion.py`. First, we compute the finite elements as shown in the previous example. Secondly, we compute the $\boldsymbol{F}_i$ and differentiate to get the Jacobian:

```
u, ujs = sum("u", fe)

for i in range(0,fe.nbf()):

  # compute diffusion term
  fi_diffusion = inner(grad(u), grad(fe.N(i)))

  # compute convection term
  uxgradu = (u.transpose()*grad(u)).evalm()
  fi_convection = inner(uxgradu, fe.N(i), True)

  # add together diffusion and convection
  fi = fi_diffusion + fi_convection

  # compute the integral
  Fi = polygon.integrate(fi)

  for j in range(0,fe.nbf()):
    # differentiate to get the Jacobian
    uj = ujs.op(j)
    Jij = diff(Fi, uj)
    #print out the Jacobian
    print "J[%d,%d]=%s;\n"%(i,j,Jij)
```

The output when `conv-diffusion.py` is executed is:

```
J[0,0]=1+1/24*u2-1/12*u1-1/24*u5-1/6*u0-1/24*u4-1/24*u3;
J[0,1]=-1/12*u0+1/12*u4;
J[0,2]=-1/2+1/12*u2+1/24*u0+1/24*u4;
J[0,3]=-1/24*u0+1/24*u4;
J[0,4]=-1/2+1/24*u2+1/12*u1+1/24*u5-1/24*u0+1/24*u3;
...
```

We can now extend the above code such that it also can include power-law viscosity models, i.e.,

$$\boldsymbol{F}_i^p = \int_T (\boldsymbol{u} \cdot \nabla \boldsymbol{u}) \cdot \boldsymbol{N}_i + \mu(\boldsymbol{u}) \nabla \boldsymbol{u} : \nabla \boldsymbol{N}_i \, dx,$$

where $\mu = \mu_0 \|\nabla \boldsymbol{u}\|^n$. The Jacobian matrix is then

$$\boldsymbol{J}_{ij}^p = \frac{\partial \boldsymbol{F}_i^p}{\partial u_j}$$

The only thing we need to change then in the above script is the diffusion term (see also `conv-diffusion-power-law.py`):

```
# nonlinear power-law diffusion term
mu = inner(grad(u), grad(u))
fi_diffusion = mu0*pow(mu,n)*inner(grad(u), grad(fe.N(i)))
```

In addition, we also need to declare n and μ_0 to be either symbols or numbers.

3 Code Generation for Quadrature Based FEM systems

SyFi can also be used to generate C++ code for other FEM systems. We will here consider code generation for finite element basis functions in a format specified by the user. Other code generation examples can be found in the SyFi tutorial and source code, where code for creating both PyCC and Epetra matrices for various problems are generated. Furthermore, notice that one can print the expressions out in either of the formats: ASCII, C, LaTeX, and Python.

The following code demonstrates how C code for the basis functions is generated (see also `code_gen_simple.py`):

```
polygon = ReferenceTriangle()
fe = LagrangeFE(polygon,2)
fe.compute_basis_functions()

N_string = ""
for i in range(0,fe.nbf()):
    N_string += "    N[%d]=%s;\n"% (i, fe.N(i).printc())

c_code = """
void basis2D(double N[%d], double x, double y) {
%s
} """ % (fe.nbf(), N_string)

print c_code
```

Notice that C code for the expressions are generated with the function `printc`. The output when `code_gen_simple.py` is runned is:

```
void basis2D(double N[6], double x, double y) {
    N[0]=pow(-y-x+1.0,2.0)-(-y-x+1.0)*y-(-y-x+1.0)*x;
    N[1]=4.0*(-y-x+1.0)*x;
    N[2]=-y*x+(x*x)-(-y-x+1.0)*x;
    N[3]=4.0*(-y-x+1.0)*y;
    N[4]=4.0*y*x;
    N[5]=-y*x+(y*y)-(-y-x+1.0)*y;
}
```

Finally, notice that to change the above code to produce code for, e.g., 5th order elements all you need to do is change the degree of the element i.e.,

```
polygon = ReferenceTriangle()
fe = LagrangeFE(polygon,5)
fe.compute_basis_functions()

...
```

4 Defining a Finite Element in SyFi

Defining a finite element may of course be more technical than using it, in particular for advanced elements. Furthermore, the implementation shown below involves more of GiNaC and SyFi than the earlier examples, so the reader should have access to both the SyFi and GiNaC tutorial. We will describe the implementation of an element recently added to SyFi. The element was introduced in [9]. The special feature of this element is that it works well for both Darcy and Stokes types of flow.

The definition of the element is as follows,

$$\boldsymbol{V}(T) = \{\boldsymbol{v} \in \mathbb{P}_3^2 \ : \ \operatorname{div} \boldsymbol{v} \in \mathbb{P}_0, \ (\boldsymbol{v} \cdot \boldsymbol{n}_e)|_e \in \mathbb{P}_1 \ \forall e \in E(T)\},$$

where T is a given triangle, $E(T)$ is the edges of T, $\boldsymbol{n}_e$ is the normal vector on edge e, and $\mathbb{P}_k$ is the space of polynomials of degree k and $\mathbb{P}_k^d$ the corresponding vector space. The degrees of freedom are,

$$\begin{aligned} &\int_e (\boldsymbol{v} \cdot \boldsymbol{n}) \tau^k \, d\tau, \quad k = 0, 1, && \forall e \in E(T), \\ &\int_e (\boldsymbol{v} \cdot \boldsymbol{t}) \, d\tau, && \forall e \in E(T). \end{aligned}$$

The definition of the element is more complicated than most of the common elements. Still, we will show that it can be implemented in SyFi in about 100 lines of codes. We will compute this element in four steps:

1. Constructing the polynomial space $\boldsymbol{V}(T)$.
2. Spesifying the constraints.
3. Spesifying the degrees of freedom.
4. Solving the resulting linear system of equations.

Considering the first step, SyFi implements the Bernstein polynomials (in barycentric coordinates) with the functions `bernstein` and `bernsteinv`, for scalar and vector polynomials, respectively. The `bernstein` functions returns a list (`lst`) with the items:

- The polynomial $(a_0 x + a_1 y + a_2(1 - x - y) + \ldots)$.
- The variables $(a_0, a_1, a_2, \ldots)$.
- The polynomial basis $(x, y, 1 - x - y, \ldots)$.

In the following we construct $\mathbb{P}_3^2$:

```
Triangle triangle
ex V_space = bernsteinv(2, 3, triangle, "a");
ex V_polynomial = V_space.op(0);
ex V_variables = V_space.op(1);
```

Here V_space is the above mentioned list, V_polynomial contains the polynomial, and V_variables contains the variables.

In the second step we first specify the constraint div $\boldsymbol{v} \in \mathbb{P}_0$:

```
lst equations;
ex divV = div(V);
ex_ex_map b2c = pol2basisandcoeff(divV);
ex_ex_it iter;
// div constraints:
for (iter = b2c.begin(); iter != b2c.end(); iter++) {
    ex basis = (*iter).first;
    ex coeff= (*iter).second;
    if ( coeff != 0 && ( basis.degree(x) > 0
                        || basis.degree(y)  > 0 )  ) {
        equations.append( coeff == 0 );
    }
}
```

Here, the divergence is computed with the div function. The divergence of a function in $\mathbb{P}_3^2$ is in $\mathbb{P}_2$. Hence, it is on the form $b_0 + b_1 x + b_2 y + b_3 xy + b_4 x^2 + b_5 y^2$. In the above code we find the coefficients b_i, as expressions involving the above mentioned variables a_i and the corresponding polynomial basis, with the function pol2basisandcoeff. Then we ensure that the only coefficient which is not zero is b_0.

The next constraints $(\boldsymbol{v} \cdot \boldsymbol{n}_e)|_e \in \mathbb{P}_1$ are implemented in much of the same way as the divergence constraint. We create a loop over each edge e of the triangle and multiply $\boldsymbol{v}$ with the normal $\boldsymbol{n}_e$. Then we substitute the expression for the edge, i.e., in mathematical notation $|_e$, into $\boldsymbol{v} \cdot \boldsymbol{n}$. After substituting the expression for these lines to get $(\boldsymbol{v} \cdot \boldsymbol{n}_e)|_e$, we check that the remaining polynomial is in $\mathbb{P}_1$ in the same way as we did above.

```
// constraints on edges:
for (int i=1; i<= 3; i++) {
    Line line = triangle.line(i);
    symbol s("s");
    lst normal_vec = normal(triangle, i);
    ex Vn = inner(V, normal_vec);
    Vn = Vn.subs(line.repr(s).op(0)).subs(line.repr(s).op(1));
    b2c = pol2basisandcoeff(Vn,s);
    for (iter = b2c.begin(); iter != b2c.end(); iter++) {
        ex basis = (*iter).first;
        ex coeff= (*iter).second;
        if ( coeff != 0 &&  basis.degree(s) > 1 )
        {
            equations.append( coeff == 0 );
        }
    }
}
```

In the third step we specify the degrees of freedom. First, we specify the equations coming from $\int_e (\boldsymbol{v} \cdot \boldsymbol{n}) \tau^k, k = 0, 1$ on all edges. To do this we need to create a loop over all edges, and on each edge we create the space of linear Bernstein polynomials in barycentric coordinates on e, i.e., $\mathbb{P}_1(e)$. Then we create a loop over the basis functions τ^k in $\mathbb{P}_1(e)$ and compute the integral $\int_e (\boldsymbol{v} \cdot \boldsymbol{n}) \tau^k \, d\tau$.

```
// dofs related to the normal on the edges
for (int i=1; i<= 3; i++) {
    Line line = triangle.line(i);
    lst normal_vec = normal(triangle, i);
    ex P1_space = bernstein(1, line, istr("a",i));
    ex P1 = P1_space.op(2);
    ex Vn = inner(V, normal_vec);

    ex basis;
    for (int j=0; j< P1.nops(); j++) {
        basis = P1.op(j);
        ex integrand = Vn*basis;
        ex dofi =  line.integrate(integrand);
        dofs.insert(dofs.end(), lst(line.vertex(0),
                                    line.vertex(1), j));
        ex eq = dofi == numeric(0);
        equations.append(eq);

    }
}
```

Finally, the degrees of freedom $\int_e (\boldsymbol{v} \cdot \boldsymbol{t}) d\tau$, can be implemented in basically the same fashion as the previously described degrees of freedom To summarize, we have now specified 20 equations which is precisely the number of unknowns in $\mathbb{P}_3^2$. Hence, the space $\boldsymbol{V}(T)$ is uniquely defined, what remains is simply to solve a linear system with 20 equations and 20 unknowns. The complete source code is in `Robust.cpp`.

References

1. B. Bagheri, L. R. Scott, Analysa software package, http://people.cs.uchicago.edu/r̃idg/al/aa.html
2. C. Prud'homme, DSEL software package, http://www.hpc2n.umu.se/para06/papers/paper_147.pdf
3. T. Dupont, J. Hoffman, J. Jansson, C. Johnson R. C. Kirby, M. Knepley, M. Larson, A. Logg, R. Scott, G. N. Wells, FEniCS software package, http://www.fenics.org
4. A. Logg, FFC software package, http://www.fenics.org/ffc/
5. R. C. Kirby, FIAT software package, http://www.fenics.org/fiat/
6. O. Pironneau, F. Hecht, A. L. Hyaric, FreeFEM software package, http://www.freefem.org/ff++/index.htm
7. P. Dular, C. Geuzaine, GetDP software package, http://www.geuz.org/getdp/
8. C. Bauer, C. Dams, A. Frink, V. V. Kisil, R. Kreckel, A. Sheplyakov, J. Vollinga, GiNaC - is not a CAS, http://www.ginac.de

9. K.A. Mardal, X.-C. Tai and R. Winther, A robust finite element method for Darcy–Stokes flow, SIAM J. Numer. Anal. 40 (2002), pp. 1605–1631.
10. K. Long, Sundance software package, http://software.sandia.gov/sundance/
11. K.-A. Mardal, SyFi - Symbolic Finite Elements, http://syfi.sf.net
12. D. Beazley et. al., SWIG - Simplified Wrapper and Interface Generator, http://www.swig.org
13. O. Skavhaug, O. Certik, Swiginac - Python interface to GiNaC http://swiginac.berlios.de/
14. M. Heroux et. al., Trilinos, http://software.sandia.gov/trilinos/