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 advanced finite elements based on symbolic representations of polygonal domains, degrees of freedom and polynomial spaces. Once the finite elements and weak forms are defined, they are used to generate efficient C/C++ code.

1   Introduction

SyFi [3] is an open source C++ library for defining finite elements and weak forms as symbolic expressions. These expressions are then used to generate efficient C/C++ code. It relies on the open source C++ library GiNaC [2], which is a library for symbolic computations with a strong support for polynomials. SyFi has Python bindings created by using SWIG [4].

In this short paper we will demonstrate its use by two examples. These examples are explained further in the SyFi tutorial, which can be found on the homepage [3].

The first example shows how a finite element, namely the Crouzeix-Raviart element [1] is defined in Python using SyFi. The degrees of freedom for the Crouzeix-Raviart element are the integral of the function over an edge,

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

where $v = a_0 + a_1 x + a_2 y$. These degrees of freedom are then used to form a linear system of equations,

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

which determine the basis functions of the Crouzeix-Raviart element. The following Python code demonstrates how this can be done with SyFi,

```
class CrouzeixRaviart(StandardFE):
  def __init__(self):
    StandardFE.__init__(self)
```

```
def compute_basis_functions(self):

  # create a polynomial space, which is a list with the following items
  # 1. the polynom   : a0 + a1*x + a2*y
  # 2. the variables : a0, a1, a2
  # 3. the basis      : 1, x, y
  polspace = pol(1,2,"a") # create a polynomial space

  N = polspace.eval()[0]              # fetch the polynomial
  variables = polspace.eval()[1]      # fetch the variables

  for i in range(1,4):                # run over all sides (1,2,3)
    line = triangle.line(i)           # fetch the side/line
    dofi = line.integrate(toex(N))    # the dof as a line integral
    self.dofs.append(dofi)            # append to the list of dofs

  #create the linear system of equations
  for i in range(1,4):
    equations = []
    for j in range(1,4):
      equations.append(self.dofs[j-1].eval() == dirac(i,j) )
    sub = lsolve(equations, variables)  # solve the system
    Ni = N.subs(sub)                     # substitute a0, a1, a2
    self.Ns.append(toex(Ni));            # append to the basisfunc list
```

As can be seen below, it is straightforward to used this Python element to generate C code for the basis functions, their derivatives and the corresponding degrees of freedom:

```
triangle = ReferenceTriangle()
fe = CrouzeixRaviart()
fe.set(triangle)
fe.compute_basis_functions()
for i in range(1,fe.nbf()+1):
  print "N(%d)  = "%i,fe.N(i).eval().printc()       # print N_i in C
  print "dN(%d) = "%i,grad(fe.N(i)).eval().printc() # print grad in C
  print "dof(%d)= "%i,fe.dof(i).eval().printc()      # print dof_i in C
```

The second example shows the (symbolic) computation of the Jacobian of a nonlinear convection-diffusion equation,

$$(\mathbf{u} \cdot \nabla)\mathbf{u} - \Delta\mathbf{u} = \mathbf{f}$$

The equation on weak form is given by:

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

By letting $\mathbf{u} = \hat{\mathbf{u}} = \sum_j u_j \mathbf{N}_j$ and $\mathbf{v} = \mathbf{N}_i$, where $\mathbf{N}_i$ and $\mathbf{N}_j$ are finite element basis functions and u_j the corresponding degrees of freedom, we can compute the Jacobian to be used in a finite element method,

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

The following code shows how this can be done with SyFi. We emphasize that all of the computations are done symbolically, simply by differentiating functions of polynomials. Therefore, the code below works for any finite element.

```
void compute_nlconvdiff_element_matrix(
        FE& fe,
        matrix& A)
{
  Polygon& domain = fe.getPolygon();

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

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

  for (int i=1; i<= fe.nbf() ; i++) {

    // 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 UgradU = (U.transpose()*gradU).evalm();        // compute U*grad(U)
    ex Fi_convection = inner(UgradU, fe.N(i), true); // compute U*grad(U)*Ni

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

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

At present continuous and discontinuous Lagrangian elements, of arbitrary order in both 2D and 3D have been implemented in SyFi. Hence, Taylor-Hood elements and $\mathbb{P}_n^d - \mathbb{P}_{n-2}$ elements are readily available. The Crouzeix-Raviart element has also been implemented in 2D and 3D. In addition element matrices for Stokes problems as well as the above described nonlinear convection-diffusion equation have been implemented. Currently, we are coupling the SyFi framework with the Trilinos package [5] to solve the linear system of equations obtained by SyFi.

References

1. M. Crouzeix and P.A. Raviart, Conforming and non–conforming finite element methods for solving the stationary Stokes equations, RAIRO Anal. Numér. 7 (1973), pp. 33–76.
2. GiNaC - is not a CAS, http://www.ginac.de

4

3. SyFi - Symbolic Finite Elements, http://syfi.sf.net
4. SWIG - Simplified Wrapper and Interface Generator, http://www.swig.org
5. Trilinos, http://software.sandia.gov/trilinos/