Finite Elements with Symbolic Computations and Code Generation

Kent-Andre Mardal and Sandve Alnæs

February 28, 2008

Outline

- We will focus on creating/assembling matrices based on FEM
- Manual implementation of the integrand/variational form based on quadrature dominates the available software
- This approach is certainly the most accessible and easy approach
- Can symbolic mathematics combined with code generation compete with this approach ?
- What additional tools are needed ?
- What about efficiency and generality ?

Additional tools

- Language to express finite element methods
  (Python modules in the Fenics project : FIAT, FFC and SyFi)
- Compilers that translate code to Fortran/C/C++ for efficient evaluation
  (Python modules in the Fenics project : FIAT, FFC and SyFi)
- A low level specification of how the generated code look
  (The Fenics project UFC (Unified Form-assembly Code), containing a C header file)

Short Description of SyFi

Purpose

- SyFi is a tool for defining polygons, polynomial spaces, degrees of freedom, and finite elements
- SyFi makes it easy to define weak forms (differentiation and integration of polynomials over polygons)
- SyFi generates efficient C++ code for the computation of matrices

Dependencies

- SyFi relies heavily on GiNaC and Swiginac for the symbolic computations and code generation
- SyFi can generate matrices based on either a Dolfin or a Diffpack mesh (we plan to include other meshes soon)
- SyFi can generate either Epetra or PyCC matrices (we plan to include other matrices soon)

Short Description of GiNaC and Swiginac

GiNaC

- GiNaC is a C++ library for symbolic mathematics
- Authors: C. Bauer, C. Dams A. Frink, V. Kisil, R. Kreckel, A. Sheplyakov, J. Vollinga
- URL: www.ginac.de
- License : GPL

Swiginac

- Swiginac is a Python interface to GiNaC
- Authors: O. Skavhaug and O. Certik
- URL: http://swiginac.berlios.de/
- License: Open

Swiginac Code Example

```
from swiginac import *

# create the symbols x and y
x = symbol('x'); y = symbol('y')

# create a function f = x*x*y*y
f = x*x*y*y

# differentiate f with respect to x
dfdx = diff(f,x)

# integrate f on x=[0,1]
intf = integral(x,0,1,f).eval_integ()

# The Taylor series of order 10 of cos(x) around 0.5
g = cos(x)
print g.series(x == 0.5, 10)
```

SyFi Extends GiNaC/Swiginac

GiNaC/Swiginac supports:

- Polynomials
- Differentiation with respect to one variable
- Integration with respect to one variable

Basic Extensions in SyFi:

- Polynomial spaces (such as Legendre and Bernstein)
- Differentiations with respect to several variables
- Integration over polygons

$\rightarrow$ SyFi extends GiNaC/Swiginac with the ingredients typically needed in finite element methods

Demo: Bernstein polynomials on a Triangle

```
from swiginac import *
from SyFi import *

# create the reference triangle
t = ReferenceTriangle()

# the space of Bernstein polynomials (order 2)
polynom, coeffs, basis  = bernstein(2, t, 'a')

# differentiate the polynom
dpdx = diff(polynom, x)

# integrate the polynom over the triangle t
integral = t.integrate(polynom)

# integrate the dpdx over the edge 1
integral2 = t.line(1).integrate(dpdx)
```

Elements currently implemented in SyFi

Finite Elements

- continuous and discontinuous Lagrangian elements (arbitrary order)
- Nedelec elements (arbitrary order)
- Nedelec H(div) elements (arbitrary order)
- Raviart-Thomas elements (arbitrary order)
- Crouzeix-Raviart elements
- Hermite elements
- Bubble elements
- Arnold-Falk-Winther elasticity element (weak symmetry) (arbitrary order)

Evaluation of Weak Forms in SyFi

We will now demonstrate the computation of various element matrices in SyFi

- The mass matrix on a reference triangle:

$$\boldsymbol{M}_{ij} = \int_T N_i \, N_j \, dx$$

- The stiffness matrix on a mapped tetrahedron:

$$\boldsymbol{A}_{ij} = \int_T (J^{-1} \nabla N_i) \cdot (J^{-1} \nabla N_j) \, dx$$

where J is the Jacobian of the geometry mapping

Demo: Computing the Mass Matrix on the Reference Triangle

```
from swiginac import *
from SyFi import *

t = ReferenceTriangle()
fe = LagrangeFE(t, 3)

for i in range(0, fe.nbf()):
    for j in range(0, fe.nbf()):
        Aij = t.integrate(fe.N(i)*fe.N(i))
```

UFC - Unified Form-assembly Code

UFC

- UFC - Unified Form-assembly Code
- Low level specification of class declaration and function signatures for finite elements, element tensors etc.
- Joint work with Logg, Alnæs, Skavhaug and Langtangen

Demo: Generated Code for the Mass Matrix

```
 virtual void tabulate_tensor(double* A,
          const double * const * w,
          const cell& c) const {

  // compute D

  A[10*0 + 0]=(5.6547619047619046e-03)*D;
  A[10*0 + 1]=(1.3392857142857143e-03)*D;
  A[10*0 + 3]=(8.1845238095238097e-04)*D;
  A[10*0 + 4]=(1.3392857142857143e-03)*D;
  ...
  A[10*9 + 7]=(1.3392857142857143e-03)*D;
  A[10*9 + 8]=(1.3392857142857143e-03)*D;
  A[10*9 + 9]=(5.6547619047619046e-03)*D;
}
```

Demo: Computing the Stiffness Matrix on a Mapped Tethrahedron

```
from swiginac import *
from SyFi import *

t = ReferenceTriangle()
fe = LagrangeFE(t, 3)

J = symbolic_matrix(3,3, ``J'')

for i in range(0, fe.nbf()):
   for j in range(0, fe.nbf()):
      integrand = inner(grad(J, fe.N(i)),
                        grad(J, fe.N(j)))
      Aij = t.integrate(integrand)
```

Demo: Generated Code for the Stiffness Matrix

```
A[10*0+0]=(8.4999999999999998e-01*Jinv01*Jinv00
          +4.2499999999999999e-01*(Jinv10*Jinv10)
          +8.4999999999999998e-01*Jinv11*Jinv10
          +4.2499999999999999e-01*(Jinv01*Jinv01)
          +4.2499999999999999e-01*(Jinv11*Jinv11)
          +4.2499999999999999e-01*(Jinv00*Jinv00))*D;

A[10*0+1]=(-6.3749999999999996e-01*Jinv01*Jinv00
          -6.7500000000000004e-01*(Jinv10*Jinv10)
          -6.3749999999999996e-01*Jinv11*Jinv10
          +3.7499999999999999e-02*(Jinv01*Jinv01)
          +3.7499999999999999e-02*(Jinv11*Jinv11)
          -6.7500000000000004e-01*(Jinv00*Jinv00))*D;

...
```

Efficiency

The generated code is very efficient

- The generated code for the mass and stiffness matrices is very efficient
- The reason is that the spatial variables x, y and z are integrated away
- What remains is polynomials in terms of J^{-1}
- The code is almost as efficient as similar code generated by FFC
- Adding the element matrix to the global matrix now dominates the time used to assemble the matrix

The Computation of the Jacobian matrix

We will now demonstrate the computation the Jacobian matrix for a nonlinear PDE

- $\boldsymbol{u} = \sum_j u_j \boldsymbol{N}_j$
- Nonlinear convection diffusion equation

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

- The Jacobian matrix

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

SyFi Code Example : Convection-Diffusion Matrix

```
.. initialize element

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))
  fi_convection = inner(uxgradu, fe.N(i), True)

  fi = fi_diffusion + fi_convection
  Fi = polygon.integrate(fi)

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

Efficiency

Efficiency and Generality

- Again the generated code is efficient!
- There is now need to compute the Jacobian by hand, it is done automatically
- The approach is very general!
- Each entry in the matrix is now a polynomial which is linear in $\{u_i\}$ and quadratic in J^{-1}

Complicated nonlinear problems

- We have tested this approach for more complicated nonlinear problems like hyper-elastic equation described by the Fung law
- The equations are so large that they want fit this slide
- The matrix expressions are several Megabytes!
- The expression is a polynomial in many variables
- The polynomial is expressed in a bad way

Conclusion

- Symbolic mathematics and code generation is an alternative to the traditional quadrature based approach
- This approach requires a symbolic engine like e.g. GiNaC
- The generated code is often very efficient
- The symbolic engine eliminates a lot of the cumbersome task like computing the Jacobian in case of a non-linear problem
- Verification via MMS (the method of manufactured solutions) comes essentially for free
- Efficiency rely on an efficient representation of the (typically polynomial) expression
- More info: www.fenics.org