NGsolve::Give me your element
And let your eyes delight in my ways

Jay Gopalakrishnan
Portland State University
Winter 2015

Download code (works only on version 6.0) for these notes from here.

“Give me your heart / And let your eyes delight in my ways.” –The Bible
Contents

1. Automatic differentiation
2. Shape functions
3. Finite elements and spaces
4. Orientation
5. Bicubic quadrilateral element
Automatic differentiation

Goal: Create variables that know how to (exactly) differentiate themselves.

Idea:

- Differentiation obeys some rules (product rule, quotient rule etc) that we can implement by overloading operators, e.g., overload \(*\) to implement
  \[
  \partial_i (f \ast g) = f(\partial_i g) + g(\partial_i f).
  \]

- Suppose an object representing \(x_i\) (the \(i\)th coordinate) knows its value and the value of its derivatives, at any given point. Then, we can compute both the value and the value of derivatives of \(x_i \ast x_j\) by overloading \(*\) as above.
A minimalist class for differentiation

template<int D> class MyDiffVar{
    // My differentiable variables
    // File: differentiables.hpp

double Value;
double Derivatives[D];

public:

    MyDiffVar () {};

    MyDiffVar (double xi, int i) {
        Value = xi;
        // grad = i-th unit vector
        for (auto & d : Derivatives) d = 0.0;
        Derivatives[i] = 1.0;
    }

    double GetValue () const { return Value; }
    double& SetValue () { return Value; }
    double GetDerivative (int i) const { return Derivatives[i]; }
    double& SetDerivative (int i) { return Derivatives[i]; }
};
**Overload * for the differentiables**

Template implementation of implement $\partial_i (f \ast g) = f(\partial_i g) + g(\partial_i f)$:

```cpp
// implement product rule

template<int D> MyDiffVar<D> MyDiffVar<D> MyDiffVar<D> & f, const MyDiffVar<D> & g) {
    MyDiffVar<D> fg;
    fg.SetValue() = f.GetValue() * g.GetValue();
    for (int i = 0; i < D; i++)
        fg.SetDerivative(i) = f.GetValue() * g.GetDerivative(i) + g.GetValue() * f.GetDerivative(i);
    return fg;
}

Quiz: Open the file and provide operators +, -, and /.
```
Using your class

```cpp
#include "differentiables.hpp"    // File d0.cpp
using namespace std;

int main() {

    MyDiffVar<2> x(0.5, 0), y(2.0, 1);

    cout << "x:" << x << std::endl
         << "y:" << y << std::endl
         << "x*y:" << x*y << std::endl
         << "x*y*y+y:" << x*y*y+y << std::endl;
}
```

Using your simple class, you can now differentiate polynomial expressions built using \( x \) and \( y \) coordinates (or \( x_i, \ i = 1, \ldots, N \), in \( N \)-dimensions).
Exercise!

How would you modify differentiables.hpp so that you can also differentiate expressions like sin(xy)?

Make sure your modified file compiles and runs correctly with this driver:

```cpp
#include "differentiables.hpp"    // File d0x.cpp
using namespace std;

int main() {
    MyDiffVar<2> x(0.5, 0), y(2.0, 1);
    cout << "x:" << x << endl
         << "y:" << y << endl
         << "sin(x*y)/y:" << sin(x*y)/y << endl;
}
```
Netgen’s AutoDiff class

An implementation of these ideas is available in $NGSRC/netgen/libsrc/general/autodiff.hpp.
Here is an example showing how to use it:

```
#include <fem.hpp> // File d1.cpp
using namespace std;

int main() {

    AutoDiff<2> x(0.5, 0), y(2.0, 1); // x and y coords
    AutoDiff<2> b[3] = { x, y, 1-x-y }; // barycentric coords

    cout << "x: " << x << endl
        << "y: " << y << endl
        << "x*y: " << x*y << endl
        << "x*y*y+y: " << x*y*y+y << endl
        << "(b0*b1*b2-1)/y: " << (b[0]*b[1]*b[2] - 1)/y << endl;
}
```

We will use AutoDiff and the following classes to program finite elements.
FlatVector, SliceVector, etc.

```cpp
#include <bla.hpp> // File: flatvec.cpp
using namespace std; using namespace ngbla;

int main() {

    double mem[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    FlatVector<double> f1(2, mem); // A vector class that steals memory.
    FlatVector<double> f2(2, mem+3); // memory from elsewhere.
    cout << "f1:\n" << f1 << endl; // This prints 1, 2.
    cout << "f2:\n" << f2 << endl; // This prints 5, 6.

    SliceVector<> s1(4, 2, mem); // Also steals memory.
    cout << "s1:\n" << s1 << endl; // This prints 1, 3, 5, 7.
    SliceVector<> s2(5, 1, mem+4); // What is this?
    // :

    // SliceVector class does not allocate or delete memory.
    // Their constructors just create/copy pointers.
```
class ScalarFiniteElement

template <int D>
class ScalarFiniteElement : public FiniteElement {
    
    virtual void CalcShape(const IntegrationPoint & ip,
                           SliceVector<> shape) const = 0;

    virtual void CalcDShape(const IntegrationPoint & ip,
                             SliceMatrix<> dshape) const = 0;

    // ...
};

- shape and dshape are cheap to pass by value as function arguments even when they contain many elements.

- Any derived finite element class must provide shape functions and their derivatives.
Visualizing finite element “shape functions”

FILE: shapes.pde

geometry = square.in2d
mesh = squareTrg.vol

fespace v -type=h1ho -order=2
gridfunction u -fespace=v

numproc shapetester nptest -gridfunction=u

○ The numproc shapetester is an NGsolve tool to visualize global basis functions (called global shape functions) of an FESpace.

○ Load this PDE file. Click Solve button before doing anything else.

○ Look for a tiny window called Shape Tester that pops up.

○ The number (0, 1, ...) that you input in Shape Tester window determines which basis function will be set in gridfunction u.

○ Got to Visual menu and pick gridfunction u to visualize.
Visualizing finite element “shape functions”

```plaintext
# FILE: shapes.pde

geometry = square.in2d
mesh = squareTrg.vol

fespace v -type=h1ho -order=2
gridfunction u -fespace=v

numproc shapetester nptest -gridfunction=u
```

Global shape functions
Prepare to write your own finite element

Study these files in the folder my_little_ngsolve:

- myElement.hpp, myElement.cpp,
  myHOElement.hpp, myHOElement.cpp

All elements in a mesh are mapped from a fixed “reference element”. Pay particular attention to CalcShape(..) and CalcDshape(..). They give the values and derivatives of all local shape functions on the reference element.

- myFESpace.hpp, myFESpace.cpp,
  myHOFESpace.hpp, myHOFESpace.cpp

Each global degree of freedom (“dof”) gives a global basis function and is associated to a geometrical object of the mesh (like a vertex, edge, or element). Pay particular attention to GetDofNrs(...), which return global dof-numbers connected to an element.

Jay Gopalakrishnan
Your assignment is to code the bicubic finite element $Q_3$ in bicubicelem.cpp. On the reference element, the unit square, this element consists of the space of functions

$$Q_3 = \text{span}\{x^i y^j : 0 \leq i \leq 3, 0 \leq j \leq 3\}.$$ 

Also code a bicubic finite element space (derived from FESpace), for any mesh of quadrilateral elements, in file bicubicspace.cpp.

Then, use your space to approximate the operator $-\Delta + I$ and solve a Neumann boundary value problem. Tabulate errors.

The ensuing slides give you hints to complete this homework and suggest separating the work into smaller separate tasks.
Bicubic shape functions on unit square

**Task 1:** In *bicubicelem.cpp*, provide shape functions.

E.g., here is a valid basis set of shape functions (you may use others):

**Vertex basis**

\[
\begin{align*}
\phi_0 &= (1 - x)(1 - y) \\
\phi_1 &= (1 - x)y \\
\phi_2 &= x(1 - y) \\
\phi_3 &= xy
\end{align*}
\]

**Edge basis**

\[
\begin{align*}
\phi_4 &= (\phi_0 \phi_1) \phi_0 \\
\phi_5 &= (\phi_0 \phi_1) \phi_1 \\
\phi_6 &= (\phi_1 \phi_3) \phi_3 \\
\phi_7 &= (\phi_1 \phi_3) \phi_1
\end{align*}
\]

...
Bicubic shape functions on unit square

Task 1: In bicubic elem.cpp, provide shape functions.

- Your basis expressions should go into the CalcShape member function.
- For the CalcDShape member function, you can use AutoDiff variables and the same expressions you need in CalcShape.
- Consider simplifying your code so that you only type the basis expressions once.
**Task 2:** In `bicubicspace.cpp`, write your finite element space.

Remember to keep track of matching local and global orientation (go back and revise your `bicubicelem.cpp` if necessary).

```cpp
/* What is the local orientation? Is the ordering of vertices and edges within the reference element */

/* v2 e1 v3 v3 e1 v2 */
/* o-------o     o-------o */
/* e2 |     | e3 e2 |     | e3 */
/* o-------o     o-------o */
/* v0 e0 v1 , v0 e0 v1 , else? */

What is the global orientation? NGsolve’s mesh edges are directed/oriented. If edge shape functions from adjacent elements are not given in that orientation, then you may lose continuity!
Task 3: Compile the code you wrote and make a shared library

```
make libmyquad.so
```

and check your basis functions on the three given quadrilateral mesh files.

```c
# FILE : bicubicshapes.pde

geometry = square.in2d
#mesh = squareQuad1.vol.gz
#mesh = squareQuad2.vol.gz
mesh = squareQuad3.vol.gz

shared = libmyquad

define fespace v -type=myquadspace

define gridfunction u -fespace=v

numproc shapetester nptest -gridfunction=u
```
Task 4: Using your finite element space, solve this boundary value problem:

\[-\Delta u + u = f \quad \text{on} \ \Omega\]
\[\partial u/\partial n = 0 \quad \text{on} \ \partial \Omega.\]

Hints:

- Do you know the variational formulation for this problem?
- You want to write a PDE file that mixes your finite element space with the NGSolve integrators.
- E.g., the NGSolve integrator \texttt{laplace}, can work with any finite element which provides \texttt{CalcDShape}, by dynamic polymorphism.
Solve a PDE

Task 4: Using your finite element space, solve this boundary value problem:

\[-\Delta u + u = f \quad \text{on } \Omega\]
\[\partial u/\partial n = 0 \quad \text{on } \partial \Omega.\]

This task includes these steps:

1. Set \( f \) so that your exact solution is \( u = \sin(\pi x)^2 \sin(\pi y)^2 \).

2. Compute the \( L^2(\Omega) \) error (code this either in your own C++ numproc – like we did before – or find facilities to directly do it in the pde file).

3. Solve on \( \text{mesh} = \text{squareQuad3.vol.gz} \) by loading your pde file and pressing the Solve button. Compute the \( L^2(\Omega) \)-error. Note it down.

4. Pressing the Solve button again to solve and compute the \( L^2 \)-error on a uniformly refined mesh. Note the \( L^2 \)-error. Repeat (until you can’t).

5. What is the rate of convergence of \( L^2 \)-error with meshsize?
**Student Team Project:** Learn about the “DPG method” and download an implementation of it in GitHub. Your job is to extend it to quadrilateral elements. You will need to code a new finite element space that will serve as the “test” space for the DPG method. Details will be progressively made clear as you proceed with the project.