FENICSTOOLS
SMART ADD-ONS FOR FENICS

M. Mortensen, M. Kuchta
mikaem@math.uio.no/mirok@math.uio.no

fenicstools is a collection of extensions to FEniCS library with particular focus on efficient visualization and postprocessing of results of large scale computations.

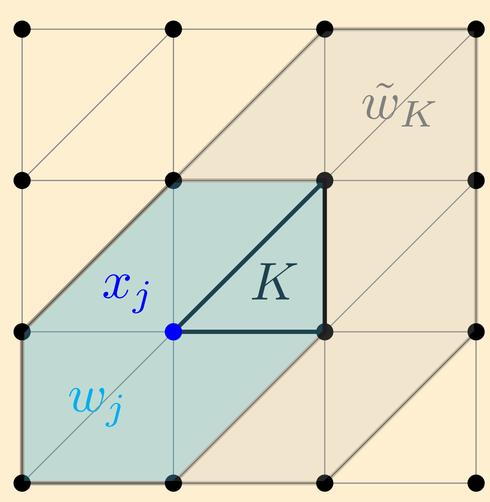
Clément interpolation

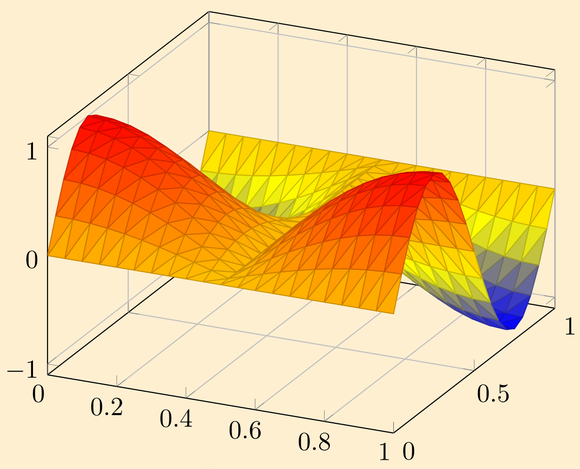
Evaluating quantities derived from primary unknowns, e.g. strain from displacement, is a frequent part of computational loops. Such quantities often lack the H^2 regularity required for nodal interpolation and therefore can be computed in FEniCS only by L^2 projection. An alternative method is the Clément interpolation - a numerical technique for constructing interpolants of H^1 functions based on local regularization.

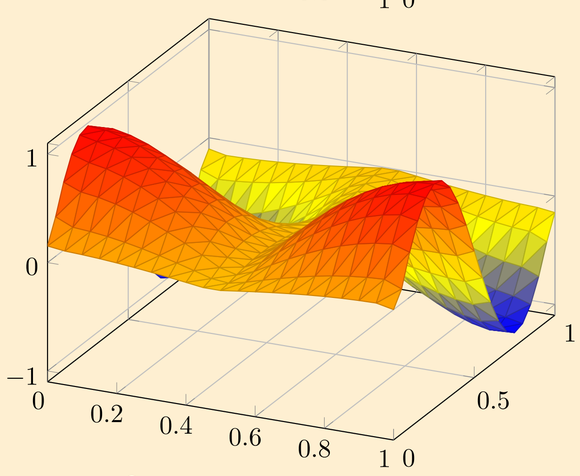
▷ **Figure**: fenicstools implements the lowest order Clément interpolation operator resulting in a CG_1 approximation of interpolated function f. The degree of freedom at x_j is computed as v minimizing $\|f - v\|^2_{0,w_j}$ over constant fields on patch w_j. The interpolation error is controlled on the union $\tilde{w}_K$. Therefore no power h is lost in the error estimates:

$$\|u - I_h u\|_{m,K} \leq C h_K^{1-m} \|u\|_{1,\tilde{w}_K}$$

for $u \in H^1$, $m = 0, 1$ and K and element of triangulation.

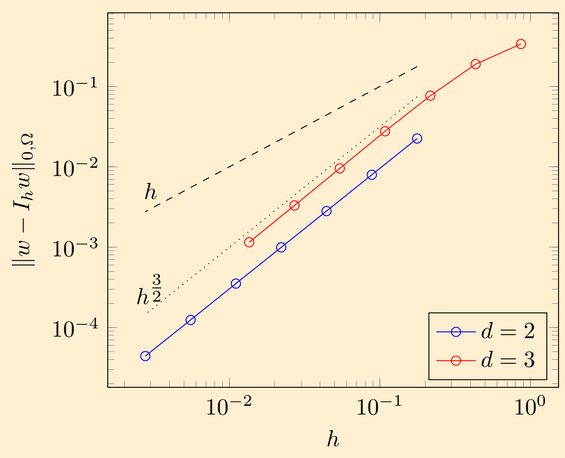






◁ **Figure**: The local regularization/averaging procedure results in smearing of gradients. However, the largest errors are localized near the boundaries where the interpolant fails to preserve the boundary values.

▽ **Figure**:fenicstools supports Clément interpolation of all* valid UFL expressions. How about evaluating $w = \nabla\cdot(u \otimes \nabla v)$ for $u \in [\mathrm{CG}_1]^2$, $v \in \mathrm{CG}_1$ or $w = \sin(\det \nabla u)$ where u is a scalar field in $\mathbb{R}^3$?



Unlike L^2 projection, which requires solution of large linear system, Clément interpolant is constructed from local linear systems of size 1 assembled over patches surrounding mesh vertices. In fenicstools the mapping is realized more efficiently using a precomputed averaging operator A such that $Ab(u) = I_h u$. Due to this choice the setup cost of the interpolant is higher (4x) than that of L^2 projector. However, the subsequent (repetitive) evaluation comes at a cost of a matrix-vector product. ▷ **Table**: MPI.MAX-ed timings (in seconds) of action of Clément interpolator and L^2 projector.

CPUs	degrees of freedom		
	2101250	8396802	14612418
1	(1.0, 5.2)	(4.1, 21.0)	(7.5, 37.2)
2	(0.7, 3.5)	(2.7, 14.0)	(4.0, 26.4)
4	(0.7, 2.2)	(1.5, 10.8)	(2.0, 19.1)
8	(0.3, 1.6)	(0.7, 8.5)	(1.8, 13.9)
16	(0.2, 1.1)	(0.7, 7.5)	(1.7, 13.4)

Development cycle

fenicstools is developed using Travis Continuous Integration and Anaconda Cloud. With Travis CI all tests are automatically executed in an Ubuntu environment on travis-ci.org whenever there is a new commit to origin/master, or if someone creates a pull request. A tailored Anaconda FEniCS version is used in a Mini-Conda environment for fast integration.

```
conda config --add channels mikaem/label/travis
conda config --add channels mikaem
conda install fenics=1.7.0 pyvtk h5py=2.6.0
```

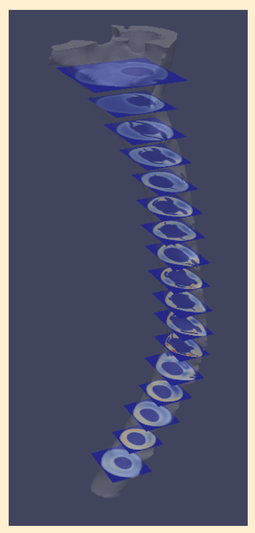
Probes/Structured grids

A turbulent flow simulation often requires setting a probe at a certain location inside the flow, where regular samplings are made over time. This can be done efficiently with *Probes/StatisticsProbes* classes.

```
from dolfin import *
from numpy import array
from fenicstools import *
mesh = UnitSquareMesh(10, 10)
V = FunctionSpace(mesh, "CG", 1)
x = array([[0.1, 0.1], [0.5, 0.5]])
probes = Probes(x.flatten(), V)
u = project(Expression("x[0]*x[1]"), V)
probes(u)    # evaluate probes once
print probes.array() # [ 0.01  0.25]
```

△ **Code**: Probe two locations defined by the array.

▷ **Figure**: The *StructuredGrid* class allows you to set probes in a $2d$ slice through a $3d$ (or $2d$) geometry - or it can be used to set probes in a $3d$ box for some interesting part of the simulated geometry. The slice may be stored as VTK files and viewed, e.g., in Paraview.



Lagrangian particles

Lagrangian particle tracking has been implemented to enable tracking of passive scalars throughout a flow domain. Particles are created with the *LagrangianParticles* class.

```
lp = LagrangianParticles(V)
particle_positions = RandomCircle(...)
lp.add_particles(particle_positions)
lp.step(u, dt=dt)  # Time integration
```

△ **Code**: Create an instance of the LagrangianParticles class and advect particles inside fluid advector loop.

The particles are advected according to

$$\frac{d\boldsymbol{x}_p}{dt} = \boldsymbol{u}(\boldsymbol{x}, t)$$

where $\boldsymbol{u}$ is the velocity vector of the fluid and $\boldsymbol{x}_p$ is the location of the particle.

▽ **Figure:** The *LagrangianParticles* class has been used to study drug delivery and mixing in a cerebrospinal fluid flow. Here particles have been used since the drug has such low diffusivity that regular FEM schemes are unstable.