Using FEniCS for Finite Element Methods

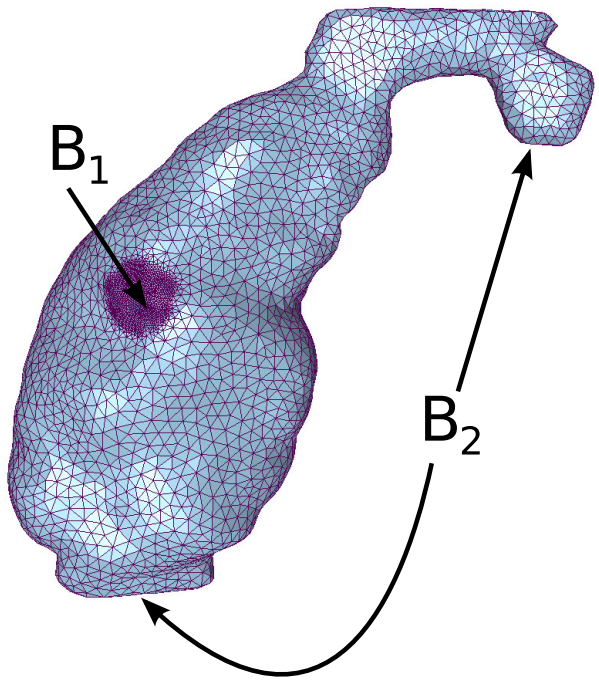
Andrew Gillette

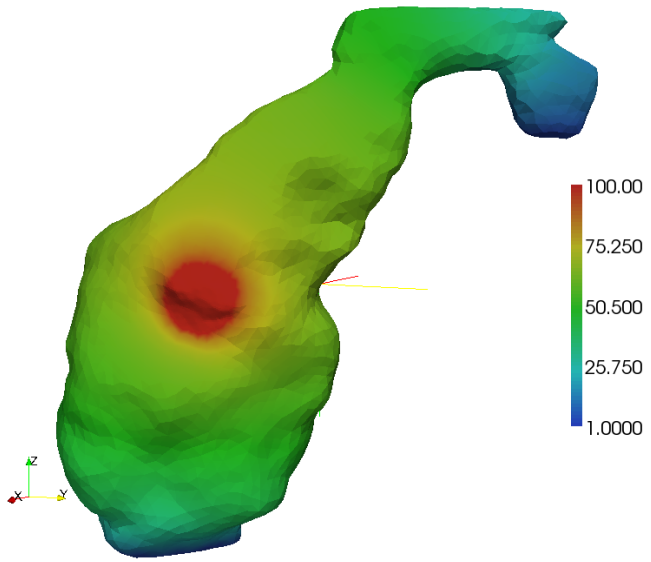
UC San Diego
Department of Mathematics
August 2012

slides adapted from a longer presentation by
Johan Hake, Simula Research Laboratory, Norway

The finite element method (FEM) can be used to discretize and solve PDEs for general geometries

$$\nabla^2 u = 0$$
$$u(x) = 100 \text{ at } B_1$$
$$u(x) = 0 \text{ at } B_2$$





FEniCS and PyDOLFIN are computational tools that bundle a lot of subtle problem setup into a simple interface

$$\frac{\partial c}{\partial t} = D\nabla^2 c$$

www.fenics.org

python™

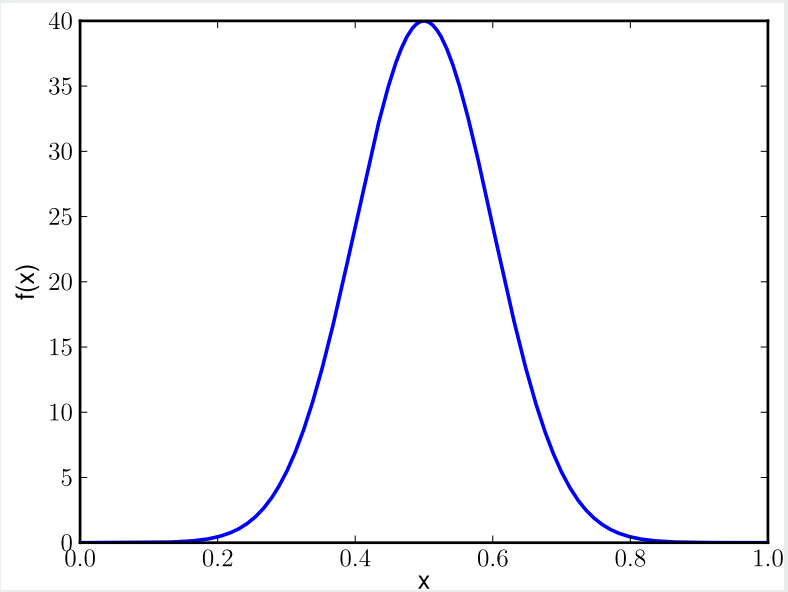
```
>>> from dolfin import *
>>> mesh = Mesh('single-TT.xml')
>>> 
```

We will first use FEniCS to solve a stationary diffusion problem, the Poisson equation on the unit interval

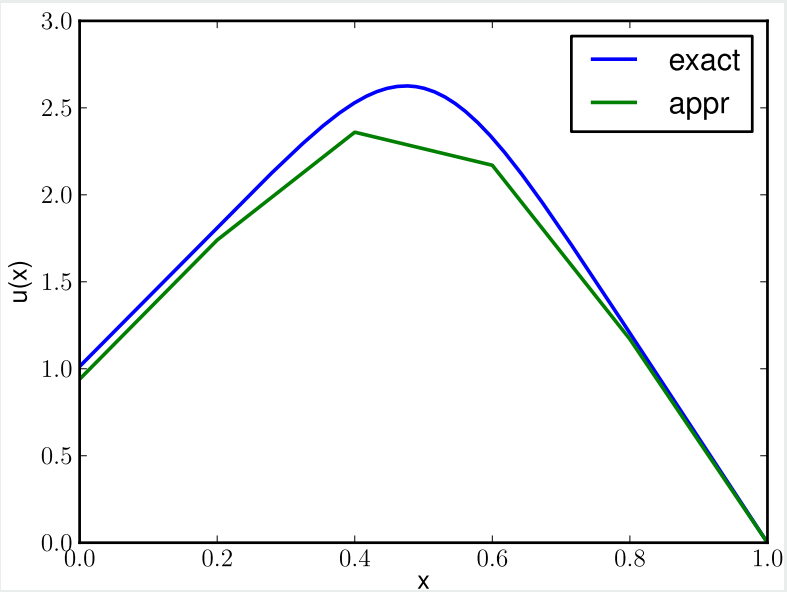
Poisson

$$-u'' = f; \quad u'(0) = 4; \quad u(1) = 0$$

Source, $f(x)$



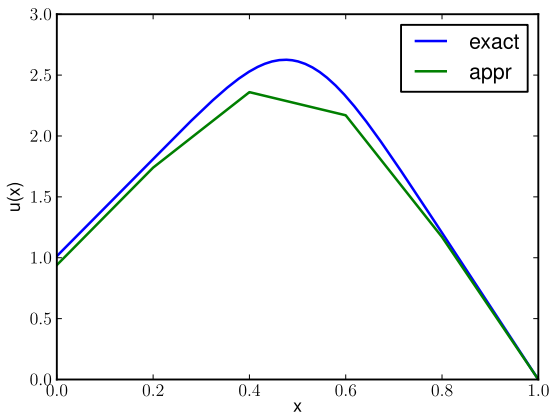
Solution, $u(x)$



The problem can be stated, solved and plotted, using a **PyDOLFIN** script written in 16 lines

```
>>> from dolfin import *
>>>
>>> mesh = UnitInterval(5)
>>> V = FunctionSpace(mesh, "CG", 1)
>>>
>>> def right(x, on_boundary):
...     return x > 1-DOLFIN_EPS
...
>>> g = Constant(0)
>>> bc = DirichletBC(V, g, right)
>>>
>>> f = Expression("A*exp(-pow(x[0]-0.5,2)/(2*pow(sigma,2)))")
>>> f.A, f.sigma = 40, 0.1
>>>
>>> u = TrialFunction(V)
>>> v = TestFunction(V)
>>>
>>> a = inner(grad(v),grad(u))*dx
>>> L = f*v*dx - 4*v*ds
>>>
>>> problem = VariationalProblem(a, L, bc)
>>> u = problem.solve()
>>>
>>> plot(u)
```

$$
\begin{aligned}
-u'' &= f \\
u'(0) &= 4 \\
u(1) &= 0
\end{aligned}
$$

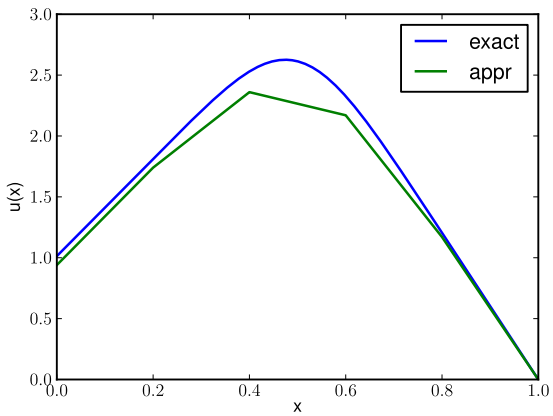


The problem can be stated, solved and plotted, using a **PyDOLFIN** script written in 16 lines

```
>>> from dolfin import *
>>>
>>> mesh = UnitInterval(5)
>>> V = FunctionSpace(mesh, "CG", 1)
>>>
>>> def right(x, on_boundary):
...     return x > 1-DOLFIN_EPS
...
>>> g = Constant(0)
>>> bc = DirichletBC(V, g, right)
>>>
>>> f = Expression("A*exp(-pow(x[0]-0.5,2)/(2*pow(sigma,2)))")
>>> f.A, f.sigma = 40, 0.1
>>>
>>> u = TrialFunction(V)
>>> v = TestFunction(V)
>>>
>>> a = inner(grad(v),grad(u))*dx
>>> L = f*v*dx - 4*v*ds
>>>
>>> problem = VariationalProblem(a, L, bc)
>>> u = problem.solve()
>>>
>>> plot(u)
```

Domain
Solution space

$$
\begin{aligned}
-u'' &= f \\
u'(0) &= 4 \\
u(1) &= 0
\end{aligned}
$$

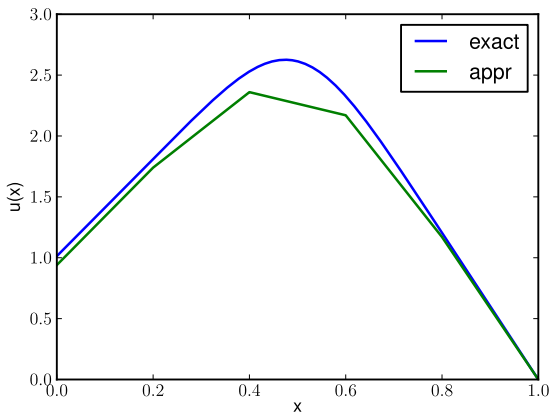


The problem can be stated, solved and plotted, using a **PyDOLFIN** script written in 16 lines

```
>>> from dolfin import *
>>>
>>> mesh = UnitInterval(5)
>>> V = FunctionSpace(mesh, "CG", 1)
>>>
>>> def right(x, on_boundary):
...     return x > 1-DOLFIN_EPS
...
>>> g = Constant(0)
>>> bc = DirichletBC(V, g, right)
>>>
>>> f = Expression("A*exp(-pow(x[0]-0.5,2)/(2*pow(sigma,2)))")
>>> f.A, f.sigma = 40, 0.1
>>>
>>> u = TrialFunction(V)
>>> v = TestFunction(V)
>>>
>>> a = inner(grad(v),grad(u))*dx
>>> L = f*v*dx - 4*v*ds
>>>
>>> problem = VariationalProblem(a, L, bc)
>>> u = problem.solve()
>>>
>>> plot(u)
```

Boundary condition

$$
\begin{aligned}
-u'' &= f \\
u'(0) &= 4 \\
u(1) &= 0
\end{aligned}
$$

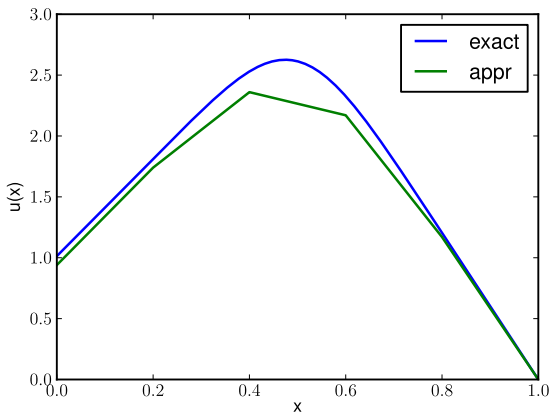


The problem can be stated, solved and plotted, using a **PyDOLFIN** script written in 16 lines

```
>>> from dolfin import *
>>>
>>> mesh = UnitInterval(5)
>>> V = FunctionSpace(mesh, "CG", 1)
>>>
>>> def right(x, on_boundary):
...     return x > 1-DOLFIN_EPS
...
>>> g = Constant(0)
>>> bc = DirichletBC(V, g, right)
>>>
>>> f = Expression("A*exp(-pow(x[0]-0.5,2)/(2*pow(sigma,2)))")
>>> f.A, f.sigma = 40, 0.1
>>>
>>> u = TrialFunction(V)
>>> v = TestFunction(V)
>>>
>>> a = inner(grad(v),grad(u))*dx
>>> L = f*v*dx - 4*v*ds
>>>
>>> problem = VariationalProblem(a, L, bc)
>>> u = problem.solve()
>>>
>>> plot(u)
```

Source term

$$
\begin{aligned}
-u'' &= f \\
u'(0) &= 4 \\
u(1) &= 0
\end{aligned}
$$

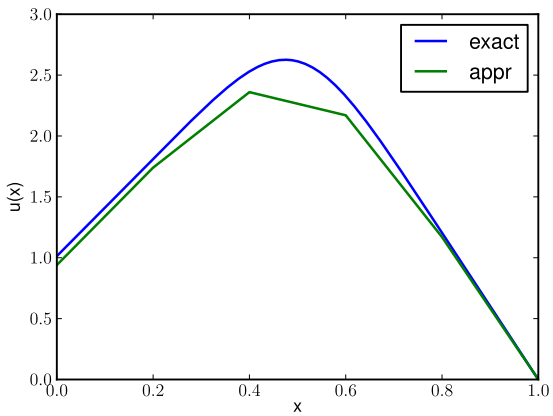


The problem can be stated, solved and plotted, using a **PyDOLFIN** script written in 16 lines

```
>>> from dolfin import *
>>>
>>> mesh = UnitInterval(5)
>>> V = FunctionSpace(mesh, "CG", 1)
>>>
>>> def right(x, on_boundary):
...     return x > 1-DOLFIN_EPS
...
>>> g = Constant(0)
>>> bc = DirichletBC(V, g, right)
>>>
>>> f = Expression("A*exp(-pow(x[0]-0.5,2)/(2*pow(sigma,2)))")
>>> f.A, f.sigma = 40, 0.1
>>>
>>> u = TrialFunction(V)
>>> v = TestFunction(V)
>>>
>>> a = inner(grad(v),grad(u))*dx
>>> L = f*v*dx - 4*v*ds
>>>
>>> problem = VariationalProblem(a, L, bc)
>>> u = problem.solve()
>>>
>>> plot(u)
```

Variational formulation

$$
\begin{aligned}
-u'' &= f \\
u'(0) &= 4 \\
u(1) &= 0
\end{aligned}
$$

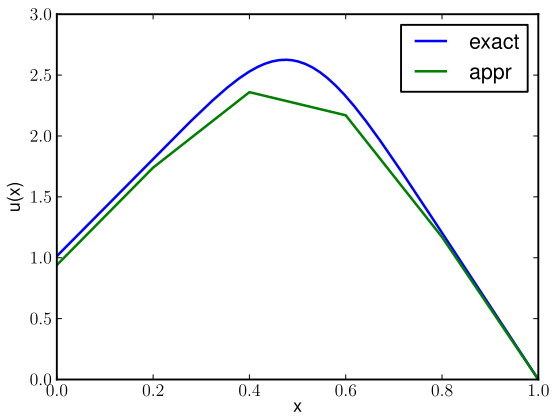


The problem can be stated, solved and plotted, using a **PyDOLFIN** script written in 16 lines

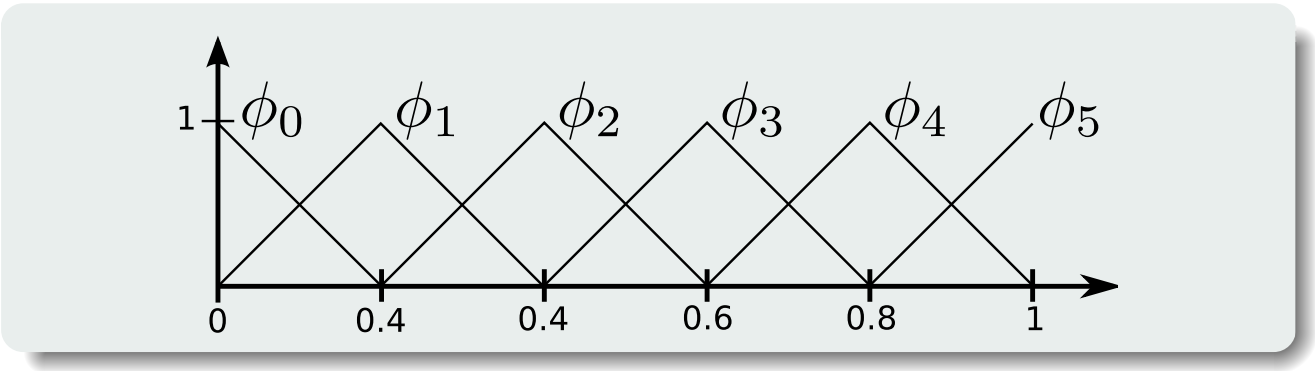
```
>>> from dolfin import *
>>>
>>> mesh = UnitInterval(5)
>>> V = FunctionSpace(mesh, "CG", 1)
>>>
>>> def right(x, on_boundary):
...     return x > 1-DOLFIN_EPS
...
>>> g = Constant(0)
>>> bc = DirichletBC(V, g, right)
>>>
>>> f = Expression("A*exp(-pow(x[0]-0.5,2)/(2*pow(sigma,2)))")
>>> f.A, f.sigma = 40, 0.1
>>>
>>> u = TrialFunction(V)
>>> v = TestFunction(V)
>>>
>>> a = inner(grad(v),grad(u))*dx
>>> L = f*v*dx - 4*v*ds
>>>
>>> problem = VariationalProblem(a, L, bc)
>>> u = problem.solve()
>>>
>>> plot(u)
```

Solve and plot

$$
\begin{aligned}
-u'' &= f \\
u'(0) &= 4 \\
u(1) &= 0
\end{aligned}
$$



How did this become a discrete problem?



A **FunctionSpace** in PyDOLFIN takes a mesh and a *finite element* as arguments.

```
>>> mesh = UnitInterval(5)
>>> V = FunctionSpace(mesh, "Lagrange", 1)
```

The first order *Lagrange* is the finite element described by the piecewise linear *nodal basis* functions.

The PDE can be re-written using the *discrete weak* formulation

Strong formulation

$$-u'' = f$$

- Should be true for every point (*strong*) in space

Discrete weak formulation

$$\begin{aligned} u(x) &= \sum_{j=0}^{5} u_j \phi_j(x) \\ -\int_0^1 u'' \phi_i dx &= \int_0^1 f \phi_i dx, \; i = 0, ..., 5 \end{aligned}$$

- By weighting the equation with ϕ_i and taking the integral over the whole domain, we solve an approximation of u (*weak*)

Integrate the left side 'by parts' and simplify

$$-\int_0^1 u''\phi_i dx \quad = \quad \int_0^1 u'\phi_i' dx + u'(0)\phi_i(0) - u'(1)\phi_i(1)$$

- This includes the derivatives at the boundaries!
- Recall that our *boundary conditions* implies that $\phi_i(1) = 0$ and $u'(0) = 4$, which gives us:

$$\int_0^1 u'\phi_i' dx = \int_0^1 f\phi_i dx - 4\phi_i(1), \; i = 0, ..., 5$$

- We use v instead of ϕ_i and write the **variational problem**:

Find $u \in V$ such that

$$a(u, v) = L(v), \quad \forall v \in V, \quad \text{where}$$

$$a(u, v) = \int_\Omega u'v' dx \quad \text{and} \quad L(v) = \int_\Omega f\, v\, dx - \int_{\partial\Omega} 4v\, ds$$

FEniCS can be used to describe variational forms, and **PyDOLFIN** can be used to solve **VariationalProblems**

Mathematical notation:

Find $u \in V$ such that

$$a(u, v) = L(v), \quad \forall v \in V$$

where:

$$a(u, v) = \int_\Omega u' v' dx$$

$$L(v) = \int_\Omega f\, v\, dx - \int_{\partial\Omega} 4v\, ds$$

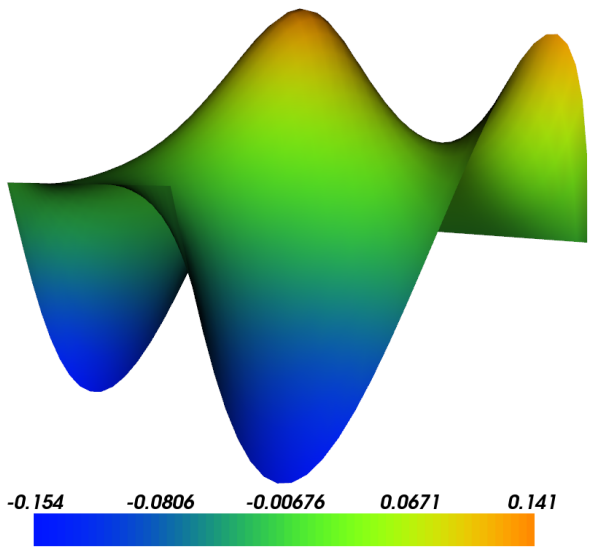
PyDOLFIN notation:

```
>>> u = TrialFunction(V)
>>> v = TestFunction(V)
>>>
>>> a = inner(grad(v),grad(u))*dx
>>> L = f*v*dx - 4*v*ds
>>>
>>> problem = VariationalProblem(a, L, bc)
>>> u = problem.solve()
```

The approach applies to 2D and 3D

$$-\nabla^2 u = f;\ \frac{\partial u}{\partial n}(:,[0,1]) = g;\ u([0,1],:) = 0$$

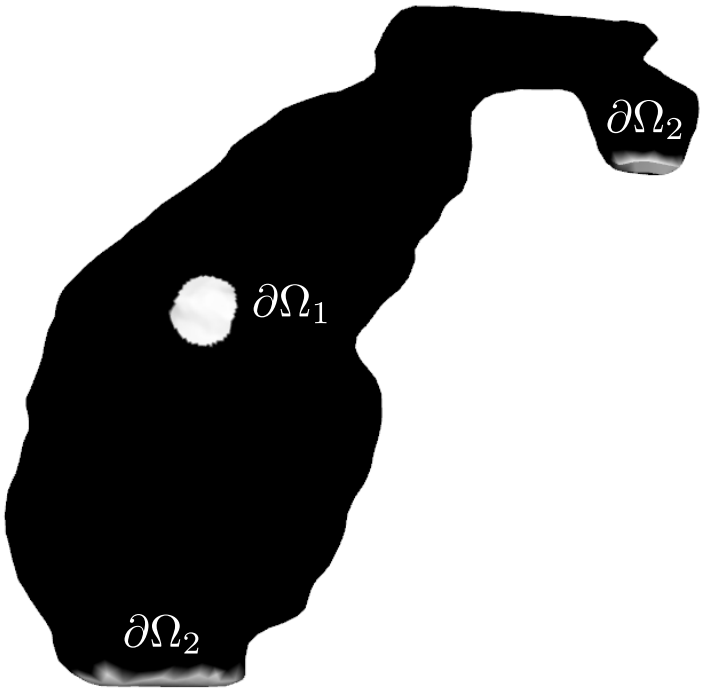
```
>>> from dolfin import *
>>>
>>> mesh = UnitSquare(32, 32)
>>> V = FunctionSpace(mesh, "CG", 1)
>>>
>>> def boundary(x):
...     return x[0] < DOLFIN_EPS or x[0] > 1.0 - DOLFIN_EPS
...
>>> u0 = Constant(0.0)
>>> bc = DirichletBC(V, u0, boundary)
>>>
>>> v = TestFunction(V)
>>> u = TrialFunction(V)
>>> f = Expression("10*exp(-(pow(x[0] - 0.5, 2) + pow(x[1] - 0.5, 2)) / 0.02)")
>>> g = Expression("-sin(5*x[0])")
>>> a = inner(grad(v), grad(u))*dx
>>> L = v*f*dx + v*g*ds
>>>
>>> problem = VariationalProblem(a, L, bc)
>>> u = problem.solve()
>>> plot(u, interactive=True)
```



Time dependent system, like the diffusion equation, can be solved using the same framework

PDE

$$
\begin{aligned}
\dot{u} &= D\nabla^2 u && \text{in } \Omega \\
D\frac{\partial u}{\partial n} &= J(t) && \text{on } \partial\Omega_1 \\
u &= 1 && \text{on } \partial\Omega_2 \\
u(0,:) &= 1 \\
J(t) &= \begin{cases} 100 : t \leq J_{stop} \\ 0 \quad : t > J_{stop} \end{cases}
\end{aligned}
$$



Diffusion equation solved using **PyDOLFIN** (Domain declarations)

```
>>> from dolfin import *
>>>
>>> mesh = Mesh("single-TT.xml.gz")
>>> subdomains = MeshFunction("uint", mesh, 2)
>>> V = FunctionSpace(mesh, "CG", 1)
>>>
>>> tstop = 3.0; J_stop = 2.0; dt = 0.02; u0 = 1; J0 = 100; D = 100
>>>
>>> class Outflow(SubDomain):
...     def inside(self, x, on_boundary):
...         return (on_boundary and -110 < x[0] and x[0] < -50 and \
...                 70 < x[1] and x[1] < 130 and \
...                 22 < x[2] and x[2] < 82) or (5 < x[0] and x[0] < 75 and \
...                 -105 < x[1] and x[1] < -35 and \
...                 -210 < x[2] and x[2] < -140)
...
>>> class Inflow(SubDomain):
...     def inside(self, x, on_boundary):
...         return on_boundary and ((x[0]-65)**2+(x[1]+30)**2+x[2]**2 < 16**2)
...
>>> outflow = Outflow()
>>> inflow = Inflow()
>>>
>>> inflow.mark(subdomains, 2)
>>>
>>> out_values = Constant(1)
>>> bc = DirichletBC(V, out_values, outflow)
```

Diffusion equation solved using **PyDOLFIN** (Domain declarations)

```
>>> from dolfin import *
>>>
>>> mesh = Mesh("single-TT.xml.gz")
>>> subdomains = MeshFunction("uint", mesh, 2)
>>> V = FunctionSpace(mesh, "CG", 1)
>>>
>>> tstop = 3.0; J_stop = 2.0; dt = 0.02; u0 = 1; J0 = 100; D = 100
>>>
>>> class Outflow(SubDomain):
...     def inside(self, x, on_boundary):
...         return (on_boundary and -110 < x[0] and x[0] < -50 and \
...                 70 < x[1] and x[1] < 130 and \
...                 22 < x[2] and x[2] < 82) or (5 < x[0] and x[0] < 75 and \
...                 -105 < x[1] and x[1] < -35 and \
...                 -210 < x[2] and x[2] < -140)
...
>>> class Inflow(SubDomain):
...     def inside(self, x, on_boundary):
...         return on_boundary and ((x[0]-65)**2+(x[1]+30)**2+x[2]**2 < 16**2)
...
>>> outflow = Outflow()
>>> inflow = Inflow()
>>>
>>> inflow.mark(subdomains, 2)
>>>
>>> out_values = Constant(1)
>>> bc = DirichletBC(V, out_values, outflow)
```

Domain,
Subdomains &
Solution space

Diffusion equation solved using **PyDOLFIN** (Domain declarations)

```
>>> from dolfin import *
>>>
>>> mesh = Mesh("single-TT.xml.gz")
>>> subdomains = MeshFunction("uint", mesh, 2)
>>> V = FunctionSpace(mesh, "CG", 1)
>>>
>>> tstop = 3.0; J_stop = 2.0; dt = 0.02; u0 = 1; J0 = 100; D = 100
>>>
>>> class Outflow(SubDomain):
...     def inside(self, x, on_boundary):
...         return (on_boundary and -110 < x[0] and x[0] < -50 and \
...                 70 < x[1] and x[1] < 130 and \
...                 22 < x[2] and x[2] < 82) or (5 < x[0] and x[0] < 75 and \
...                 -105 < x[1] and x[1] < -35 and \
...                 -210 < x[2] and x[2] < -140)
...
>>> class Inflow(SubDomain):
...     def inside(self, x, on_boundary):
...         return on_boundary and ((x[0]-65)**2+(x[1]+30)**2+x[2]**2 < 16**2)
...
>>> outflow = Outflow()
>>> inflow = Inflow()
>>>
>>> inflow.mark(subdomains, 2)
>>>
>>> out_values = Constant(1)
>>> bc = DirichletBC(V, out_values, outflow)
```

Model parameters

Diffusion equation solved using **PyDOLFIN** (Domain declarations)

```
>>> from dolfin import *
>>>
>>> mesh = Mesh("single-TT.xml.gz")
>>> subdomains = MeshFunction("uint", mesh, 2)
>>> V = FunctionSpace(mesh, "CG", 1)
>>>
>>> tstop = 3.0; J_stop = 2.0; dt = 0.02; u0 = 1; J0 = 100; D = 100
>>>
>>> class Outflow(SubDomain):
...     def inside(self, x, on_boundary):
...         return (on_boundary and -110 < x[0] and x[0] < -50 and \
...                 70 < x[1] and x[1] < 130 and \
...                 22 < x[2] and x[2] < 82) or (5 < x[0] and x[0] < 75 and \
...                 -105 < x[1] and x[1] < -35 and \
...                 -210 < x[2] and x[2] < -140)
...
>>> class Inflow(SubDomain):
...     def inside(self, x, on_boundary):
...         return on_boundary and ((x[0]-65)**2+(x[1]+30)**2+x[2]**2 < 16**2)
...
>>> outflow = Outflow()
>>> inflow = Inflow()
>>>
>>> inflow.mark(subdomains, 2)
>>>
>>> out_values = Constant(1)
>>> bc = DirichletBC(V, out_values, outflow)
```

Define boundaries

Diffusion equation solved using **PyDOLFIN** (linear algebra initialization)

```
>>> u = TrialFunction(V)
>>> v = TestFunction(V)
>>>
>>> K = assemble(inner(grad(u),grad(v))*dx)
>>> M = assemble(u*v*dx)
>>> source = assemble(v*ds(2), exterior_facet_domains=subdomains)
>>>
>>> u_n = Function(V)
>>>
>>> A = K.copy()
>>> b = Vector(A.size(1))
>>> b[:] = 0.0
>>> x = u_n.vector()
>>> x[:] = u0
```

Diffusion equation solved using **PyDOLFIN** (linear algebra initialization)

```
>>> u = TrialFunction(V)
>>> v = TestFunction(V)
>>>
>>> K = assemble(inner(grad(u),grad(v))*dx)
>>> M = assemble(u*v*dx)
>>> source = assemble(v*ds(2), exterior_facet_domains=subdomains)
>>>
>>> u_n = Function(V)
>>>
>>> A = K.copy()
>>> b = Vector(A.size(1))
>>> b[:] = 0.0
>>> x = u_n.vector()
>>> x[:] = u0
```

Basis functions

Diffusion equation solved using **PyDOLFIN** (linear algebra initialization)

```
>>> u = TrialFunction(V)
>>> v = TestFunction(V)
>>>
>>> K = assemble(inner(grad(u),grad(v))*dx)
>>> M = assemble(u*v*dx)
>>> source = assemble(v*ds(2), exterior_facet_domains=subdomains)
>>>
>>> u_n = Function(V)
>>>
>>> A = K.copy()
>>> b = Vector(A.size(1))
>>> b[:] = 0.0
>>> x = u_n.vector()
>>> x[:] = u0
```

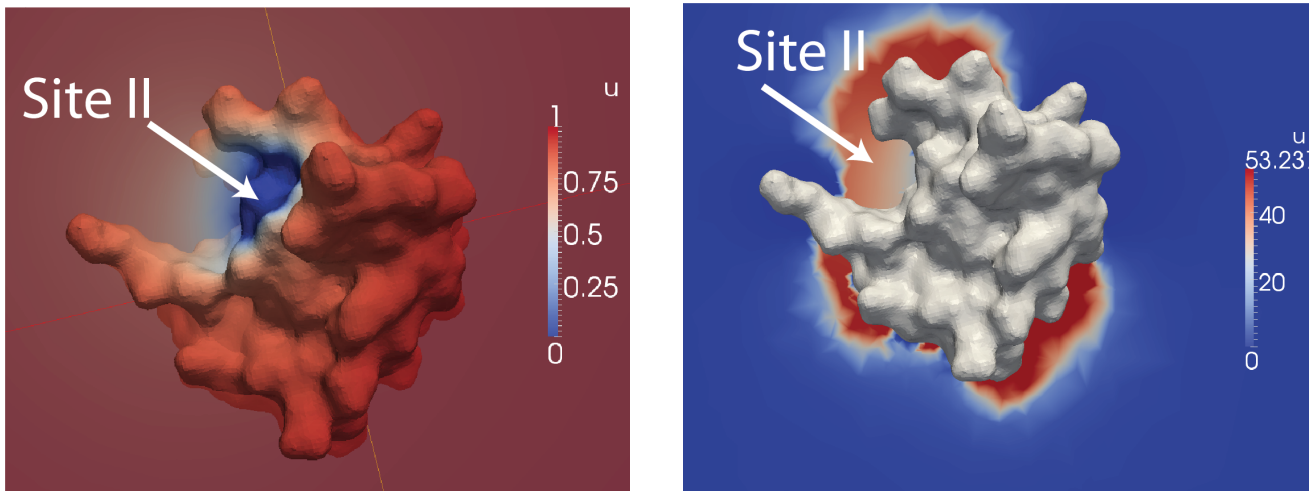
Assemble tensors

Diffusion equation solved using **PyDOLFIN** (linear algebra initialization)

```
>>> u = TrialFunction(V)
>>> v = TestFunction(V)
>>>
>>> K = assemble(inner(grad(u),grad(v))*dx)
>>> M = assemble(u*v*dx)
>>> source = assemble(v*ds(2), exterior_facet_domains=subdomains)
>>>
>>> u_n = Function(V)
>>>
>>> A = K.copy()
>>> b = Vector(A.size(1))
>>> b[:] = 0.0
>>> x = u_n.vector()
>>> x[:] = u0
```

Initialise linear algebra

FEniCS is used in real, modern science!



Steady-state Ca^{2+} distributions about Troponin C, showing the absence and presence of an electrostatic potential.

Kekenes-Huskey, Gillette, Hake, McCammon
Finite Element Estimation of Protein-Ligand Association Rates with Post-Encounter Effects: Applications to Calcium binding in Troponin C and SERCA,
Submitted 2012.

Questions?

- Slides from this presentation are available at my website:

  http://ccom.ucsd.edu/∼agillette

- Thanks to Johan Hake for preparing many of these slides!