FEniCS, part III

M. M. Sussman
sussmanm@math.pitt.edu
Office Hours: 11:10AM-12:10PM, Thack 622

May 12 – June 19, 2014

How do I generate complicated meshes?

- **UnitIntervalMesh**, **UnitSquareMesh** and **UnitCubeMesh**
- **RectangleMesh**, **BoxMesh**
- If have a simple figure, can map unit square into it
- Use **MeshEditor**
- Can read **XML** or **OFF** file
- Use a few CGAL functions
- Can use **dolfin-convert**

Mesh mapping

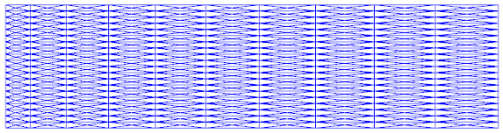
If you have a differentiable function $f : I^2 \to \mathbb{R}^2$, you can map an existing mesh:

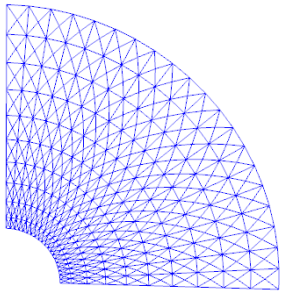
```
from dolfin import *
import numpy

Theta = pi/2
a, b = 1, 5.0
nr = 10 # divisions in r direction
nt = 20 # divisions in theta direction
mesh = RectangleMesh(a, 0, b, 1, nr, nt, "crossed")

# First make a denser mesh towards r=a
x = mesh.coordinates()[:,0]
y = mesh.coordinates()[:,1]
s = 1.3

def denser(x,y):
    return [a + (b-a)*((x-a)/(b-a))**s, y]

x_bar, y_bar = denser(x, y)
xy_bar_coor = numpy.array([x_bar, y_bar]).transpose()
mesh.coordinates()[:] = xy_bar_coor
plot(mesh, title="stretched mesh", interactive=True)
```


Mesh mapping cont'd

```
def cylinder(r, s):
    return [r*numpy.cos(Theta*s),r*numpy.sin(Theta*s)]

x_hat, y_hat = cylinder(x_bar, y_bar)
xy_hat_coor = numpy.array([x_hat, y_hat]).transpose()
mesh.coordinates()[:] = xy_hat_coor
plot(mesh, title="hollow cylinder")
interactive()
```

Modified mesh





MeshEditor

From the book: no one should ever do this kind of thing by hand

```
mesh = Mesh();
editor = MeshEditor();
editor.open(mesh, 2, 2)
editor.init_vertices(4)
editor.init_cells(2)
editor.add_vertex(0, 0.0, 0.0)
editor.add_vertex(1, 1.0, 0.0)
editor.add_vertex(2, 1.0, 1.0)
editor.add_vertex(3, 0.0, 1.0)
editor.add_cell(0, 0, 1, 2)
editor.add_cell(1, 0, 2, 3)
editor.close()
```


XML files

This format is useful for saving and communicating meshes, not for generating them.

```
<?xml version="1.0" encoding="UTF-8"?>
<dolfin smlns:dolfin="http"//fenicsproject.org">
  <mesh celltype="triangle" dim="2">
    <vertices size="9">
      <vertex index="0" x="0" y="0"/>
      <vertex index="1" x="0.5" y="0"/>
      <vertex index="2" x="1" y="0"/>
      <vertex index="3" x="0" y="0.5"/>
      <vertex index="4" x="0.5" y="0.5"/>
      <vertex index="5" x="1" y="0.5"/>
      <vertex index="6" x="0" y="1"/>
      <vertex index="7" x="0.5" y="1"/>
      <vertex index="8" x="1" y="1"/>
    </vertices>
    <cells size="8">
      <triangle index="0" v0="0" v1="1" v2="4"/>
      <triangle index="1" v0="0" v1="3" v2="4"/>
      <triangle index="2" v0="1" v1="2" v2="5"/>
      <triangle index="3" v0="1" v1="4" v2="5"/>
      <triangle index="4" v0="3" v1="4" v2="7"/>
      <triangle index="5" v0="3" v1="6" v2="7"/>
      <triangle index="6" v0="4" v1="5" v2="8"/>
      <triangle index="7" v0="4" v1="7" v2="8"/>
    </cells>
  </mesh>
</dolfin>
```

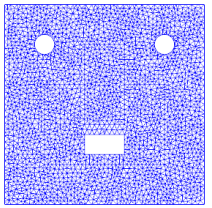

CGAL functions

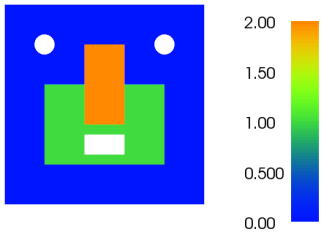
- Computational Geometry Algorithms Library
  - Open-source project
  - Python bindings
  - *Probably* can use more than Dolfin provides.
- Best built-in option
- **`CircleMesh`**
- **`EllipseMesh`**
- **`SphereMesh`**
- **`EllipsoidMesh`**
- **`PolyhedralMeshGenerator`**
- Add and subtract, overlap figures

`example9.py` 2D example

```
from dolfin import *

# Define 2D geometry
domain = Rectangle(0., 0., 5., 5.) - \
    Rectangle(2., 1.25, 3., 1.75) - Circle(1, 4, .25) - Circle(4, 4, .25)
domain.set_subdomain(1, Rectangle(1., 1., 4., 3.))
domain.set_subdomain(2, Rectangle(2., 2., 3., 4.))

# Generate and plot mesh
mesh2d = Mesh(domain, 45)
plot(mesh2d, "2D mesh")

# Convert subdomains to mesh function for plotting
mf = MeshFunction("size_t", mesh2d, 2, mesh2d.domains())
plot(mf, "Subdomains")

interactive()
```


Output





Read .OFF files

- .OFF files are written by some programs
- `PolyhedralMeshGenerator` to read OFF files
- Awkward to generate by hand

Real meshes: use `dolfin-convert`

Suffix	File format
`.xml`	DOLFIN XML format
`.ele / .node`	Triangle file format
`.mesh`	Medit format, generated by TetGen with -g
`.msh / .gmsh`	Gmsh* version 2.0
`.grid`	Diffpack tetrahedral grid format
`.inp`	Abaqus tetrahedral grid format
`.e / .exo`	Sandia Exodus II file format
`.ncdf`	ncdump'ed Exodus II
`.vrt / .cell`	Star-CD tetrahedral grid format

*Installed on your VM

Mesh refinement

- **`mesh = refine(mesh)`**: uniform refinement
- **`mesh = refine(mesh, marker_function`**: selective refinement
  - **`marker_function`** is a boolean **`CellFunction`**
  - Depends on good error estimator
- Chapters 28 and 29 include discussion of adaptive mesh refinement
- The goal-oriented adaptive solution in Chapter 29 is implemented in **`AdaptiveLinearVariationalSolver`**.

`AdaptiveLinearVariationalSolver` example

```
# directory: dolfin-demos/documented/auto-adaptive-poisson/python/
# file:          demo_auto-adaptive_poisson.py
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(8, 8)
V = FunctionSpace(mesh, "Lagrange", 1)

# Define boundary condition
u0 = Function(V)
bc = DirichletBC(V, u0, "x[0] < DOLFIN_EPS || x[0] > 1.0 - DOLFIN_EPS")

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Expression("10*exp(-(pow(x[0] - 0.5, 2) + pow(x[1] - 0.5, 2)) / 0.02)",
               degree=1)
g = Expression("sin(5*x[0])", degree=1)
a = inner(grad(u), grad(v))*dx()
L = f*v*dx() + g*v*ds()

# Define function for the solution
u = Function(V)

# Define goal functional (quantity of interest)
M = u*dx()

# Define error tolerance
tol = 1.e-5
```


`AdaptiveLinearVariationalSolver` example cont'd

```
# Solve equation a = L with respect to u and the given boundary
# conditions, such that the estimated error (measured in M) is less
# than tol
problem = LinearVariationalProblem(a, L, u, bc)
solver = AdaptiveLinearVariationalSolver(problem, M)
solver.parameters["error_control"]["dual_variational_solver"]\
          ["linear_solver"] = "cg"
solver.solve(tol)

solver.summary()

# Plot solution(s)
plot(u.root_node(), title="Solution on initial mesh")
plot(u.leaf_node(), title="Solution on final mesh")
interactive()
```


`AdaptiveLinearVariationalSolver` output

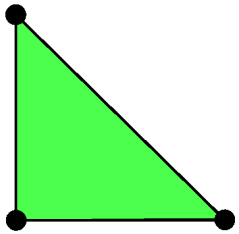
```
Level  |  functional_value  error_estimate  tolerance  num_cells  num_dofs
-------------------------------------------------------------------------
0      |           0.121629        0.001179      1e-05        128        81
1      |           0.125724     0.000881366      1e-05        162        98
2      |           0.125833     0.000852912      1e-05        209       126
3      |           0.126245     0.000456543      1e-05        267       156
4      |           0.125464     0.000308555      1e-05        343       197
5      |           0.124939     0.000247634      1e-05        511       283
6      |             0.1253     0.000216778      1e-05        713       394
7      |           0.125635     0.000143245      1e-05        966       526
8      |           0.125408     9.46871e-05      1e-05       1218       656
9      |           0.125237     7.26173e-05      1e-05       1819       962
10     |           0.125174     5.40451e-05      1e-05       2648      1393
11     |           0.125304     3.39704e-05      1e-05       3694      1928
12     |           0.125285     2.22991e-05      1e-05       4759      2472
13     |           0.125234     1.83246e-05      1e-05       7114      3660
14     |           0.125204     1.38494e-05      1e-05      10308      5280
15     |           0.125243     8.78483e-06      1e-05      14402      7358
```

Elements available

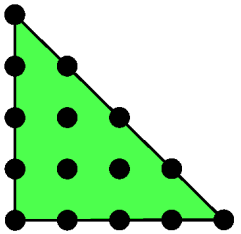
Name	Symbol	Dimension	Degree
Argyris	ARG	2	5
Arnold-Winther	AW	2	
Brezzi-Douglas-Marini	BDM	2,3	1-6
Crouzeix-Raviart	CR	2,3	1
Discontinuous Lagrange	DG	2,3	1-6
Hermite	HER	2,3	
Lagrange	CG	2,3	1-6
Mordal-Tai-Winther	MTW	2	
Morley	MOR	2	
Nédélec 1st kind H(curl)	N1curl	2,3	6
Nédélec 2nd kind H(curl)	N2curl	2,3	6
Raviart-Thomas	RT	2,3	6

Lagrange (CG) elements

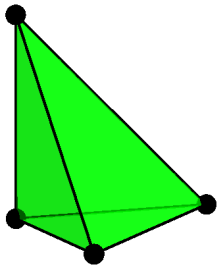
First order on triangles



Fourth order on triangles

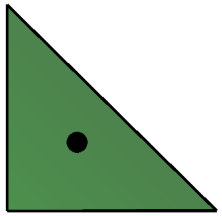


First order on tetrahedra

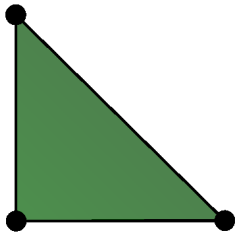


Discontinuous Lagrange (DG) elements

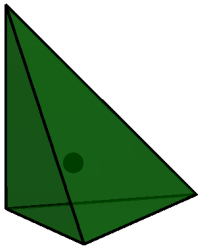
Constant on triangles



Linear on triangles



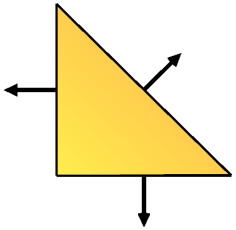
Constant on tetrahedra



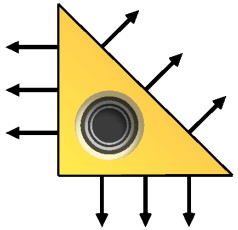
Raviert-Thomas (RT) elements

These elements are $H(\mathrm{div})$-conforming.

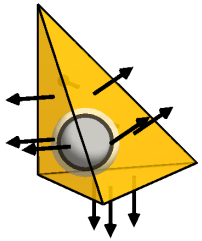
First order on triangles



Third order on triangles



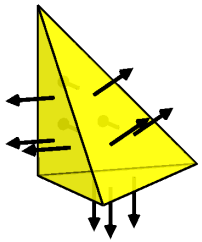
Second order on tetrahedra



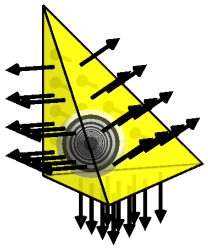
Brezzi-Douglas-Marini (BDM) elements

These elements are $H(\mathrm{div})$-conforming.

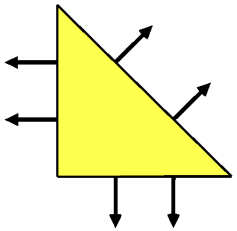
First order on tetrahedra



Third order on tetrahedra



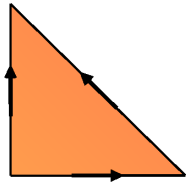
Second order on triangles



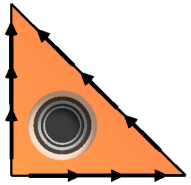
Nédélec First Kind elements

These elements are $H(\mathrm{curl})$-conforming.

First order on triangles



Third order on triangles

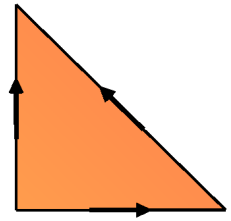


First order on tetrahedra

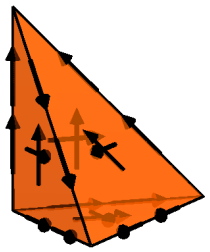
Nédélec Second Kind elements

These elements are $H(\mathrm{curl})$-conforming.

First order on triangles



First order on tetrahedra



A nonlinear problem

Suppose $\Omega = [0, 1]$ is the unit interval, and consider the equation

$$\begin{aligned} -\nabla \cdot (q(u)\nabla u) &= f \\ u &= 0 \text{ for } x = 0 \\ u &= 1 \text{ for } x = 1 \\ q(u) &= (1+u)^m \end{aligned}$$

The exact solution is $u = \left((2^{m+1}-1)x + 1\right)^{1/(m+1)} - 1$

Variational formulation

$$F(u; v) = \int_\Omega q(u)\nabla u \cdot \nabla v \, dx = 0 \quad \forall v \in \hat{V}$$

where

$$V = \{v \in H^1(\Omega) : v = 0 \text{ on } x = 0, v = 1 \text{ on } x = 1\}$$

and

$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } x = 0, v = 0 \text{ on } x = 1\}$$

Picard iteration (successive substitution)

Given an initial guess u^0, recursively define iterates u^{k+1} as solutions of the *linear* variational problem

$$a(u, v) = \int_\Omega q(u^k)\nabla u \cdot \nabla v \, dx$$

with the same boundary conditions as before.

`example10.py`: Picard iteration

```
from dolfin import *
import numpy as np
import scipy.linalg as la

# Create mesh and define function space
mesh = UnitIntervalMesh(10)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
fuzz = 1E-14
def left_boundary(x, on_boundary):
    return on_boundary and abs(x[0]) < fuzz

def right_boundary(x, on_boundary):
    return on_boundary and abs(x[0]-1) < fuzz

Gamma_0 = DirichletBC(V, Constant(0.0), left_boundary)
Gamma_1 = DirichletBC(V, Constant(1.0), right_boundary)
bcs = [Gamma_0, Gamma_1]

# Choice of nonlinear coefficient
m = 2

def q(u):
    return (1+u)**m
```


`example10.py`: Picard iteration, cont'd

```
# Define variational problem for Picard iteration
u = TrialFunction(V)
v = TestFunction(V)
u_k = interpolate(Constant(0.0), V)  # previous (known) u
a = inner(q(u_k)*nabla_grad(u), nabla_grad(v))*dx
f = Constant(0.0)
L = f*v*dx

# Picard iterations
u = Function(V)      # new unknown function
eps = 1.0            # error measure ||u-u_k||
rho = 0              # convergence rate
tol = 1.0E-6         # tolerance
iter = 0             # iteration counter
maxiter = 25         # max no of iterations allowed
for iter in range(maxiter):
    solve(a == L, u, bcs)
    diff = u.vector().array() - u_k.vector().array()
    oldeps = eps
    eps = la.norm(diff, ord=np.Inf) / la.norm(u.vector().array(), ord=np.Inf)
    rho = eps/oldeps
    print 'iter=%d: norm=%g, rho=%g' % (iter, eps, rho)
    if eps < tol * (1.0-rho):
        break
    u_k.assign(u)    # update for next iteration
```

`assign` ??

- "Assigns" one function to another
- Presumably efficient
- It means `u_k.vector()[:]=u.vector()`

`example10.py`: Picard iteration, cont'd

```
convergence = 'convergence after %d Picard iterations' % iter
if iter >= maxiter-1:
    convergence = 'no ' + convergence
print convergence

# Find max error
u_exact = Expression('pow((pow(2, m+1)-1)*x[0] + 1, 1.0/(m+1)) - 1', m=m)
u_e = interpolate(u_exact, V)
diff = la.norm((u_e.vector().array() - u.vector().array()), ord=np.Inf)
print 'Max error:', diff
```


`example10.py`: results

```
iter=0: norm=1, rho=1
iter=1: norm=0.171129, rho=0.171129
iter=2: norm=0.0149607, rho=0.0874237
iter=3: norm=0.00620223, rho=0.414567
iter=4: norm=0.000799731, rho=0.128943
iter=5: norm=0.000240982, rho=0.301329)
iter=6: norm=3.99044e-05, rho=0.165591
iter=7: norm=8.81351e-06, rho=0.220866
iter=8: norm=1.85317e-06, rho=0.210265
iter=9: norm=3.30119e-07, rho=0.178137
convergence after 9 Picard iterations
```


Newton instead of Picard iteration

- System of nonlinear equations must hold $\forall v \in \hat{V}_h$

$$a(u,v) - L(v) = \int_\Omega q(u)\nabla u \cdot \nabla v \, dx - \int_\Omega fv \, dx = 0$$

- Approximate by a system of nonlinear equations

$$F_i(u) = a(u, \phi_i) - L(\phi_i) = 0$$

- Given an inital guess, u^0, solve the nonlinear system

$$(J\,\delta u^k)_i = (J(u^k - u^{k-1}))_i = a(u^{k-1}, \phi_i) - L(\phi_i)$$

- δu^k satisfies *homogeneous* Dirichlet conditions
- J is Jacobian matrix
- FEniCS needs this expressed in weak form

Newton iteration as weak form

Writing a vector $U = (u_1, u_2, \dots)^T$, Newton iteration is applied to the nonlinear system

$$F_i(\sum_\ell U_\ell \phi_\ell) = \int_\Omega q(\sum_\ell U_\ell \phi_\ell) \sum_\ell U_\ell \nabla\phi_\ell \cdot \nabla\phi_i \, dx - \int_\Omega f\phi_i \, dx = 0$$

Given a starting vector U^0, define subsequent iterates as

$$\begin{aligned} J(U^k)\,\delta U^{k+1} &= -F_i(U^k) \\ U^{k+1} &= U^k + \delta U^{k+1} \end{aligned}$$

The Jacobian matrix is

$$J_{ij} = \frac{\partial F_i}{\partial U_j} =$$

$$\int_\Omega q|_{\sum_\ell U_\ell\phi_\ell} \nabla\phi_j \cdot \nabla\phi_i \, dx + \int_\Omega \left.\frac{dq}{du}\right|_{\sum_\ell U_\ell\phi_\ell} \phi_j \, (\sum_\ell U_\ell \nabla\phi_\ell) \cdot \nabla\phi_i \, dx$$

Newton iteration cont'd

Writing $q^k = q(\sum_\ell U_\ell^k \phi_\ell)$ and $u^k = \sum_\ell U_\ell^k \phi_\ell$, The Newton update system becomes

$$\begin{pmatrix} & \vdots & \\ \dots & \int_\Omega q^k \nabla\phi_j \cdot \nabla\phi_i \, dx + \int_\Omega \left(\frac{dq}{du}\right)^k \phi_j \nabla u^k \cdot \nabla\phi_i \, dx & \dots \\ & \vdots & \end{pmatrix} \begin{pmatrix} \vdots \\ \delta U_j \\ \vdots \end{pmatrix}$$

$$= \begin{pmatrix} \vdots \\ \int_\Omega q^k \nabla u^k \cdot \nabla\phi_i \, dx - \int_\Omega f\phi_i \, dx \\ \vdots \end{pmatrix}$$

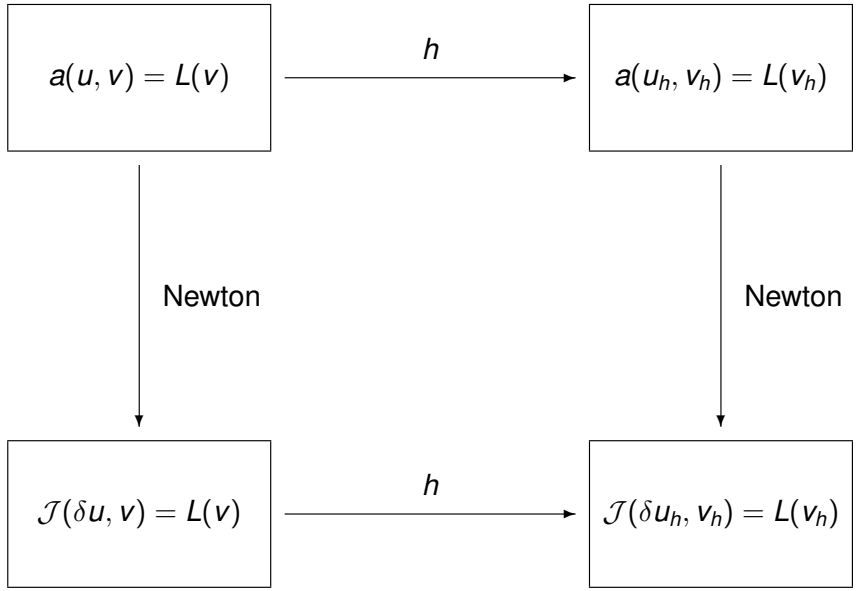
This is the system that would arise if you wanted to solve

$$\begin{aligned} \mathcal{J}(\delta u, v) &= \int_\Omega q(u^k)\nabla\delta u \cdot \nabla v \, dx + \int_\Omega \left(\frac{dq}{du}\right)^k \delta u \nabla u^k \cdot \nabla v \, dx \\ L(v) &= \int_\Omega q^k \nabla u^k \cdot \nabla v \, dx - \int_\Omega f\phi_i \, dx \end{aligned}$$

Exercise 17 (8 points)

Show by direct calculation that the same bilinear operator $\mathcal{J}$ arises from computing the Jacobian of the original nonlinear form $a(u, v) = \int_\Omega q(u)\nabla u \cdot \nabla v$ directly. (8 points) (See FEniCS book, Section 1.2.4.)

The diagram commutes!



example11.py: Newton solution

```
from dolfin import *
import numpy as np
import scipy.linalg as la

# Create mesh and define function space
mesh = UnitIntervalMesh(20)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions for initial guess
fuzz = 1E-14
def left_boundary(x, on_boundary):
    return on_boundary and abs(x[0]) < fuzz

def right_boundary(x, on_boundary):
    return on_boundary and abs(x[0]-1) < fuzz

Gamma_0 = DirichletBC(V, Constant(0.0), left_boundary)
Gamma_1 = DirichletBC(V, Constant(1.0), right_boundary)
bcs = [Gamma_0, Gamma_1]

# Define variational problem for initial guess (q(u)=1, i.e., m=0)
u = TrialFunction(V)
v = TestFunction(V)
a = inner(nabla_grad(u), nabla_grad(v))*dx
f = Constant(0.0)
L = f*v*dx
A, b = assemble_system(a, L, bcs)
u_k = Function(V)
solve(A, u_k.vector(), b, 'lu')
```


example11.py: Newton solution cont'd

```
# Note that all Dirichlet conditions must be zero for
# the correction function in a Newton-type method
Gamma_0_du = DirichletBC(V, Constant(0.0), left_boundary)
Gamma_1_du = DirichletBC(V, Constant(0.0), right_boundary)
bcs_du = [Gamma_0_du, Gamma_1_du]

# Choice of nonlinear coefficient
m = 2

def q(u):
    return (1+u)**m

def Dq(u):
    return m*(1+u)**(m-1)

# Define variational problem for the matrix and vector
# in a Newton iteration
du = TrialFunction(V) # u = u_k + omega*du
J = inner(q(u_k)*nabla_grad(du), nabla_grad(v))*dx + \
    inner(Dq(u_k)*du*nabla_grad(u_k), nabla_grad(v))*dx
L = -inner(q(u_k)*nabla_grad(u_k), nabla_grad(v))*dx
```


example11.py: Newton solution cont'd

```
# Newton iteration at the algebraic level
du = Function(V)
u  = Function(V)  # u = u_k + omega*du
omega = 1.0       # relaxation parameter
err = 1.0
tol = 1.0E-5
iter = 0
maxiter = 25
# u_k has correct nonhomogeneous boundary conditions
u.assign(u_k)
for iter in range(maxiter):
    print iter, "iteration",
    AJ, b = assemble_system(J, L, bcs_du)
    solve(AJ, du.vector(), b)
    u.vector()[:] += omega*du.vector()
    # or, better for parallel computing
    #u.assign(u_k)  # u = u_k
    #u.vector().axpy(omega, du.vector())
    olderr = err
    err = la.norm(du.vector().array(), ord=np.Inf) / \
          la.norm(u.vector().array(), ord=np.Inf)
    rho = err/olderr
    print "Norm=%g, rho=%g"% (err, rho)
    if err < tol * (1.0 - rho):
        break
    u_k.assign(u)
```


example11.py: Newton solution cont'd

```
convergence = 'convergence after %d Newton iterations' % iter
if iter >= maxiter:
    convergence = 'no ' + convergence

# Find max error
u_exact = Expression('pow((pow(2, m+1)-1)*x[0] + 1, 1.0/(m+1)) - 1', m=m)
u_e = interpolate(u_exact, V)
diff = la.norm((u_e.vector().array() - u.vector().array()), ord=np.Inf)
print 'Max error:', diff
```

Automatic Jacobian computation is available!

- Deriving Jacobians automatically!
- Replace the following code in **example11.py**:

```
J = inner(q(u_k)*nabla_grad(du), nabla_grad(v))*dx + \
    inner(Dq(u_k)*du*nabla_grad(u_k), nabla_grad(v))*dx

L = -inner(q(u_k)*nabla_grad(u_k), nabla_grad(v))*dx
```

- With the following code (**example12.py**)

```
L = inner(q(u_k)*nabla_grad(u_k), nabla_grad(v))*dx
J = derivative(L, u_k, du)
L = -L
```


More on the derivative

```
dolfin.derivative = derivative(form, u, du=None)
```

- Compute derivative of **form** with respect to **u**
- Resulting form has second variable (**du**)
- **v** is in same space as **u**
- "A tuple of Coefficients may be provided in place of a single Coefficient, in which case the new Argument argument is based on a MixedElement created from this tuple."

Some results

- Manual Jacobian (**example11.py**)

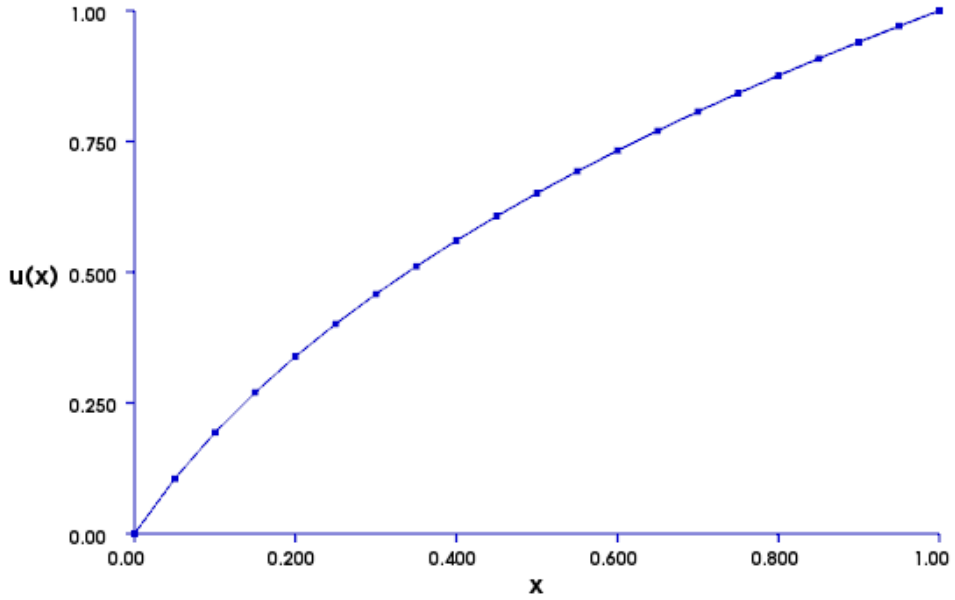
```
0 iteration Norm=0.181001, rho=0.181001
1 iteration Norm=0.0197372, rho=0.109044
2 iteration Norm=0.000269528, rho=0.0136559
3 iteration Norm=4.98281e-08, rho=0.000184872
Max error: 1.72084568817e-15
```

- Automatic Jacobian (**example12.py**)

```
0 iteration Norm=0.181001, rho=0.181001
1 iteration Norm=0.0197372, rho=0.109044
2 iteration Norm=0.000269528, rho=0.0136559
3 iteration Norm=4.98281e-08, rho=0.000184872
Max error: 1.72084568817e-15
```

- Same convergence histories
- Quadratic convergence

What does the solution look like?



Nonlinear solve can be automated!

1. Define weak form as usual
2. Turn it into a vector function using **`action`**
3. Compute Jacobian using **`derivative`**
4. Define the problem using **`NonlinearVariationalProblem`**
5. Create a solver using **`NonlinearVariationalSolver`**
6. Set solver parameters if desired
7. **`solver.solve()`**

`example13.py`: automated nonlinear solve

```
# Define variational problem
v   = TestFunction(V)
u   = TrialFunction(V)
F   = inner(q(u)*nabla_grad(u), nabla_grad(v))*dx
u_  = Function(V)
# Make functional into a vector function
F   = action(F, u_ )

# Automatic differentiation
J = derivative(F, u_ )

# set initial guess
# u_  is zero by default
uinit = interpolate(Expression("2.*x[0]*x[0]"),V)
u_.assign(uinit)

# Compute solution
problem = NonlinearVariationalProblem(F, u_ , bcs, J)
solver  = NonlinearVariationalSolver(problem)

solver.solve()
```


Controlling the nonlinear solution

- Very sensitive to initial guess
- Often hard to converge
- Linear solve inside Newton loop
- What linear method to use?
- What preconditioner to use?

`example13.py`: Parameters

```
solver  = NonlinearVariationalSolver(problem)

prm = solver.parameters
info(prm, True)
prm["nonlinear_solver"]="newton"   # default.  could be "snes"
prm["newton_solver"]["absolute_tolerance"]   = 1E-8
prm["newton_solver"]["relative_tolerance"]   = 1E-7
prm["newton_solver"]["maximum_iterations"]   =  25
prm["newton_solver"]["relaxation_parameter"] = 1.0
prm["newton_solver"]["linear_solver"] = "gmres"
prm["newton_solver"]["krylov_solver"]["absolute_tolerance"] = 1E-9
prm["newton_solver"]["krylov_solver"]["relative_tolerance"] = 1E-7
prm["newton_solver"]["krylov_solver"]["maximum_iterations"] = 1000
prm["newton_solver"]["krylov_solver"]["monitor_convergence"] = True
prm["newton_solver"]["krylov_solver"]["nonzero_initial_guess"] = False
prm["newton_solver"]["krylov_solver"]["gmres"]["restart"] = 40
prm["newton_solver"]["preconditioner"] = "jacobi" # default is "ilu"
prm["newton_solver"]["krylov_solver"]["preconditioner"]["structure"]\
    = "same_nonzero_pattern"
prm["newton_solver"]["krylov_solver"]["preconditioner"]["ilu"]["fill_level"] =0

set_log_level(PROGRESS)
```

Go deeper: use `PETSc`

- `PETSc` offers enormous control via command line
- `petsc4py` can be used
- May have to use a compiled language for full control

Recommendations

- Start: automatic differentiation, no initial guess, and automated nonlinear solve
- Fails? pick better initial guess
- Diverging? `prm["newton_solver"]["relaxation_parameter"]` smaller
- Still Fails? `set_log_level(PROGRESS)` or `DEBUG`
- Newton iteration fails? SNES.
- Linear sub-solve fails?
  - Out of memory in LU? Use Krylov solver (GMRES)
  - Begin sub-solve from previous solution instead of zero
  - Construct better preconditioner matrix
  - Larger relative and/or absolute tolerance
  - Different linear solver
- Simplify: same problems in 1D?

Transient simulations

Transient simulations follow a similar outline

1. Set up the mesh, function spaces, *etc.*
2. Set up initial condition
3. Assemble constant matrices and vectors
4. Enter timestepping loop
   - 4.1 Assemble changing matrices and vectors
   - 4.2 Solve for the new time step values

Method of lines: implicit Euler timestepping

- Transient heat equation

$$\frac{\partial u}{\partial t} = \nabla \cdot \nabla u + f$$

- Discretize in t, continuous in space

$$\frac{u^{k+1} - u^k}{\Delta t} = \nabla \cdot \nabla u^{k+1} + f^{k+1}$$

- Re-write

$$u^{k+1} - \Delta t \nabla \cdot \nabla u^{k+1} = u^k + f^{k+1}$$

- Weak form

$$(u^{k+1}, v) + \Delta t (\nabla u^{k+1}, \nabla v) = (u^k, v) + (f^{k+1}, v)$$

`example14.py`: Transient heat equation

```
from dolfin import *
import numpy

# Create mesh and define function space
mesh = UnitSquareMesh(20,10)
V = FunctionSpace(mesh, "Lagrange", 2)

# Define boundary conditions
alpha = 3; beta = 1.2
u0 = Expression("1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t",
                alpha=alpha, beta=beta, t=0)

class Boundary(SubDomain):  # define the Dirichlet boundary
    def inside(self, x, on_boundary):
        return on_boundary

boundary = Boundary()
bc = DirichletBC(V, u0, boundary)

# Initial condition
u_k = interpolate(u0, V)
dt = 0.3      # time step

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(beta - 2 - 2*alpha)
a = u*v*dx + dt*inner(nabla_grad(u), nabla_grad(v))*dx
L = (u_k + dt*f)*v*dx
```


`example14.py`: cont'd

```
A = assemble(a)   # assemble only once, before the time stepping
b = None          # trick: first time through loop below, assemble creates b

# Timestep loop
u = Function(V)   # the unknown at a new time level
T = 1.9           # total simulation time
t = dt
while t <= T:
    print "time =", t ,
    b = assemble(L, tensor=b)
    u0.t = t
    bc.apply(A, b)
    solve(A, u.vector(), b)   # trick

    # Verify
    u_e = interpolate(u0, V)
    diff = numpy.abs(u_e.vector().array() - u.vector().array())
    print "Max error: %-10.3e" %  diff.max()

    t += dt
    u_k.assign(u)
```


Exercise 18 (15 points)

1. **`example14.py`** implements the backward or implicit Euler method. In this exercise, you are to modify that program to implement the Crank-Nicolson method.

$$\frac{u^{k+1} - u^k}{\Delta t} = \Delta\left(\frac{u^{k+1} + u^k}{2}\right) + \frac{f^{k+1} + f^k}{2}$$

2. This method is *second* order in time, so it is exact for quadratic functions of t. Modify the exact solution so it includes a t^2 term and demonstrate numerically that the error on each timestep is zero or roundoff.

Efficiency

- You should assemble the constant matrices once
- Right side vectors need to be assembled each step
- Much of the work can be compressed into matrix-vector products

Saving matrices

- Weak form

$$(u^{k+1}, v) + \Delta t(\nabla u^{k+1}, \nabla v) = (u^k, v) + (f^{k+1}, v)$$

- Assembles to

$$(M + \Delta t K)U^{k+1} = MU^k + MF^k$$

$$M_{ij} = (\phi_i, \phi_j) \qquad K_{ij} = (\nabla\phi_i, \nabla\phi_j) \qquad u = \sum_i U_i\phi_i \qquad f \approx \sum_i F_i\phi_i$$

- It is usually cheaper in 3D to save M than to assemble the right side each timestep.
- In 2D problems, it depends on the details.

Exercise 19 (5 points)

Modify **`example14.py`** to construct and store the matrix M and replace the assembly of the right side vector **`b`** with multiplication my M. Be sure that your program still results in roundoff-sized errors.

Automated timestepping

- ODE integrators are available
- PETSc has them
- Perhaps FEniCS will take advantage in the future
- You could call PETSc functions directly
- **`petsc4py`** or C++

`example15.py`: Use SLEPc for eigenpairs

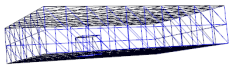
```
# Define basis and bilinear form (Laplace matrix)
u = TrialFunction(V)
v = TestFunction(V)
a = dot(grad(u), grad(v))*dx

# Assemble stiffness form
A = PETScMatrix()
assemble(a, tensor=A)

# Create eigensolver
eigensolver = SLEPcEigenSolver(A)

# Compute all eigenvalues of A x = \lambda x
print "Computing eigenvalues. This can take a minute."
eigensolver.solve()

# Extract largest (first) eigenpair
r, c, rx, cx = eigensolver.get_eigenpair(0)
```

SLEPc parameters

- `prm = eigensolver.parameters`
  `info(prm, True)`
- `pydoc dolfin.SLEPcEigenSolver`
- `"spectrum": "largest magnitude", "smallest magnitude", "largest real", "smallest real", "largest imaginary", "smallest imaginary" "target real", "target imaginary"`
- `"solver"`
- `"tolerance"` (default 1.e-15)
- `"maximum_iterations"` (positive integer)
- `"problem_type": "hermitian", "non_hermitian", "gen_hermitian", "gen_non_hermitian", "pos_gen_non_hermitian"`
- Generalized problem: $Ax = \lambda Bx$
- `"spectral_transform": "shift-and-invert"`
- `"spectral_shift"` (real number)

`example15.py` output

Mesh



Eigenvectors 1 and 200