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 $f : I^2 \rightarrow \mathbb{R}^2$, you can map an existing mesh:

```python
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

```python
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

```python
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
<?xml version="1.0" encoding="UTF-8"?>
<dolfin xmlns: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
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()
Mesh refinement

- `mesh = refine(mesh)`: uniform refinement
- `mesh = refine(mesh, marker_function)`: selective refinement
  - `marker_function` is a boolean `CellFunction`
  - Depends on a 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

```python
# directory: dolfin-demos/documentated/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()
M = u*dx()

# Define error tolerance
tol = 1.e-5

# Define goal functional (quantity of interest)
problem = LinearVariationalProblem(a, L, u, bc)
solver = AdaptiveLinearVariationalSolver(problem, M)
solver.parameters["error_control"]."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

<table>
<thead>
<tr>
<th>Level</th>
<th>functional_value</th>
<th>error_estimate</th>
<th>tolerance</th>
<th>num_cells</th>
<th>num_dofs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.121629</td>
<td>0.001179</td>
<td>1e-05</td>
<td>128</td>
<td>81</td>
</tr>
<tr>
<td>1</td>
<td>0.125724</td>
<td>0.000881366</td>
<td>1e-05</td>
<td>162</td>
<td>98</td>
</tr>
<tr>
<td>2</td>
<td>0.125833</td>
<td>0.000852912</td>
<td>1e-05</td>
<td>209</td>
<td>126</td>
</tr>
<tr>
<td>3</td>
<td>0.126245</td>
<td>0.000456543</td>
<td>1e-05</td>
<td>267</td>
<td>156</td>
</tr>
<tr>
<td>4</td>
<td>0.125446</td>
<td>0.0003838855</td>
<td>1e-05</td>
<td>343</td>
<td>197</td>
</tr>
<tr>
<td>5</td>
<td>0.124959</td>
<td>0.000274634</td>
<td>1e-05</td>
<td>511</td>
<td>283</td>
</tr>
<tr>
<td>6</td>
<td>0.125351</td>
<td>0.00016778</td>
<td>1e-05</td>
<td>713</td>
<td>394</td>
</tr>
<tr>
<td>7</td>
<td>0.126350</td>
<td>0.000143245</td>
<td>1e-05</td>
<td>966</td>
<td>526</td>
</tr>
<tr>
<td>8</td>
<td>0.125408</td>
<td>9.46871e-05</td>
<td>1e-05</td>
<td>1218</td>
<td>656</td>
</tr>
<tr>
<td>9</td>
<td>0.125237</td>
<td>7.26173e-05</td>
<td>1e-05</td>
<td>1819</td>
<td>962</td>
</tr>
<tr>
<td>10</td>
<td>0.125174</td>
<td>5.40451e-05</td>
<td>1e-05</td>
<td>2648</td>
<td>1393</td>
</tr>
<tr>
<td>11</td>
<td>0.125304</td>
<td>3.39074e-05</td>
<td>1e-05</td>
<td>3694</td>
<td>1928</td>
</tr>
<tr>
<td>12</td>
<td>0.125285</td>
<td>2.22991e-05</td>
<td>1e-05</td>
<td>4759</td>
<td>2472</td>
</tr>
<tr>
<td>13</td>
<td>0.125234</td>
<td>1.83246e-05</td>
<td>1e-05</td>
<td>7114</td>
<td>3660</td>
</tr>
<tr>
<td>14</td>
<td>0.125204</td>
<td>1.38494e-05</td>
<td>1e-05</td>
<td>10308</td>
<td>5280</td>
</tr>
<tr>
<td>15</td>
<td>0.125243</td>
<td>8.78493e-06</td>
<td>1e-05</td>
<td>14402</td>
<td>7358</td>
</tr>
</tbody>
</table>
```
### Elements available

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

### Lagrange (CG) elements

- **First order on triangles**
- **Fourth order on triangles**

### Discontinuous Lagrange (DG) elements

- **Constant on triangles**
- **Linear on triangles**

### Raviert-Thomas (RT) elements

- **First order on tetrahedra**
- **Third order on triangles**
- **Second order on tetrahedra**

These elements are $H(\text{div})$-conforming.
Brezzi-Douglas-Marini (BDM) elements

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

First order on tetrahedra

Third order on tetrahedra

Second order on triangles

Nédélec First Kind elements

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

First order on triangles

Third order on triangles

First order on tetrahedra

Nédélec Second Kind elements

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

First order on triangles

First order on tetrahedra

A nonlinear problem

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

$-\nabla \cdot (q(u) \nabla u) = f$

$u = 0$ for $x = 0$

$u = 1$ for $x = 1$

$q(u) = (1 + u)^m$

The exact solution is $u = \left( (2m+1)^{(m+1)} - 1 \right)$
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
left_boundary(x, on_boundary):
    return on_boundary and abs(x[0]) < fuzz
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

# 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 = '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=1.88317e-06, rho=0.410265
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 initial guess, $u^0$, solve the nonlinear system

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

▶ $\delta u^k$ satisfies homogeneous Dirichlet conditions
▶ $J$ is Jacobian matrix
▶ FEniCS needs this expressed in weak form
Newton iteration as weak form

Writing a vector \( U = (u_1, u_2, \ldots)^T \), 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

\[
J(U^k) \delta U^{k+1} = -F_i(U^k)
\]

\[
U^{k+1} = U^k + \delta U^{k+1}
\]

The Jacobian matrix is

\[
J_{ij} = \frac{\partial F_i}{\partial U_j} = \int_{\Omega} q(\sum \ell U_{\ell} \phi_{\ell}) \delta \phi_j \cdot \nabla \phi_i \, dx + \int_{\Omega} \frac{dq}{du} \phi_j (\sum \ell U_{\ell} \nabla \phi_{\ell}) \cdot \nabla \phi_i \, dx
\]

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.)

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

\[
\begin{bmatrix}
\cdots \\
\int_{\Omega} q^k \nabla \phi_i \cdot \nabla \phi_j \, dx + \int_{\Omega} \left( \frac{dq}{du} \right)^k \phi_j \nabla u^k \cdot \nabla \phi_i \, dx \\
\cdots \\
\end{bmatrix}
\begin{bmatrix}
\delta U_j \\
\cdots \\
\end{bmatrix}
\]

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

\[
\mathcal{J}(\delta u, v) = \int_{\Omega} q(u^k) \nabla \delta u \cdot \nabla v \, dx + \int_{\Omega} \left( \frac{dq}{du} \right)^k \delta u \nabla u^k \cdot \nabla v \, dx
\]

\[
L(v) = \int_{\Omega} q^k \nabla u^k \cdot \nabla v \, dx - \int_{\Omega} f_{\phi_i} \, dx
\]

The diagram commutes!
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')

# 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)

def qjac(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
A, b = assemble_system(J, L, bcs_du)
solve(A, du.vector(), b)
u.vector()[:]+=omega*du.vector()  # or, better for parallel computing
u.assign(u_k)
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)

# Newton iteration at the algebraic level
u = Function(V)
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)

    # or, better for parallel computing
    #u.assign(u_k)
    if iter >= maxiter:
        convergence = 'convergence after %d Newton iterations' % iter
        if iter >= maxiter:
            convergence = 'no convergence'
        break

    u_e = interpolate(u_exact, V)
diff = la.norm((u_e.vector().array() - u.vector().array()), ord=np.Inf)
    print 'Max error:%s, diff %g', convergence, diff

    # Find max error
    u_exact = Expression('pow(pow(2, m+1)-1)*x[0] + 1, 1.0/(m+1)) - 1.0', 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

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

    # Find max error
    u_e = interpolate(u_exact, V)
diff = la.norm((u_e.vector().array() - u.vector().array()), ord=np.Inf)
    print 'Max error:', diff

    # or, better for parallel computing
    #u.assign(u_k)

    if iter >= maxiter:
        convergence = 'convergence after %d Newton iterations' % iter
        break

    u.assign(u_k)
Automatic Jacobian computation is available!

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

\[
J = \text{inner}(q(u_k) \cdot \text{nabla\_grad}(du), \text{nabla\_grad}(v)) \cdot dx + \text{inner}(Dq(u_k) \cdot du \cdot \text{nabla\_grad}(u_k), \text{nabla\_grad}(v)) \cdot dx
\]

\[
L = -\text{inner}(q(u_k) \cdot \text{nabla\_grad}(u_k), \text{nabla\_grad}(v)) \cdot dx
\]

With the following code (example12.py)

\[
J = \text{derivative}(L, u_k, du)
\]

\[
L = -L
\]

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?

\[
\text{u}(x)
\]

\[
0.00 \quad 0.20 \quad 0.40 \quad 0.60 \quad 0.80 \quad 1.00
\]

\[
0.00 \quad 0.250 \quad 0.500 \quad 0.750 \quad 1.00
\]
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()`

```python
# 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?

```python
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

```python
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()
bcs = [DirichletBC(V, u0, boundary)]

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

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

- Assembles to
  \[(M + \Delta tK)U^{k+1} = MU^k + MF^k\]

- \[M_{ij} = (\phi_i, \phi_j)\]
  \[K_{ij} = (\nabla \phi_i, \nabla \phi_j)\]
  \[u = \sum_i U_i \phi_i\]
  \[f \approx \sum_i F_i \phi_i\]

- 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.

Automated timestepping

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

Exercise 19 (5 points)

Modify \texttt{example14.py} to construct and store the matrix \(M\) and replace the assembly of the right side vector \(b\) with multiplication \(M\). Be sure that your program still results in roundoff-sized errors.

\texttt{example15.py: Use SLEPc for eigenpairs}

```python
# Define basis and bilinear form (Laplace matrix)
u = TrialFunction(V)
v = ... # 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)
```

# Define basis and bilinear form (Laplace matrix)
v = TrialFunction(V)
v = ...
```
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)