FEM example in Python

M. M. Sussman

sussmanm@math.pitt.edu

Office Hours: 11:10AM-12:10PM, Thack 622

May 12 – June 19, 2014

Topics

Purpose

- Practice with Python
- Illustrate FEM in 1D in detail
- Coding strategies

Problem description

- ODE $u'' + 2u' + u = f = (x + 2)$
- Neumann boundary conditions
- Why? Because all 3 terms, real solution with exponentials
  - Exact $u = (1 + x)e^{1-x} + x(1 - e^{-x})$
- Done when get correct convergence rate to exact solution
- Other b.c., rhs are available

How to debug and test?

- Never write code without a test plan!
- Test as you go
- Choose exact solutions and test terms one at a time
- Have a `test` function as part of the code.
- When code is “working”
  - Find problems similar to given, but with exact solutions
  - Verify reasonable solution

Strategy

- Leave b.c. and r.h.s as general as possible
- *quadratic* elements
  - Makes nontrivial exact solutions possible
- Model classes on FEniCS classes

Class structure

- **`Mesh`**
- **`Shapefns`**
- **`FiniteElement`**
  - Including integration over this element
- **`FunctionSpace`**
  - Including assembly of integrals

Pseudo code

```
mesh=Mesh( ... )
sfns=Shapefns( ... )
V=FunctionSpace(mesh,sfns)
```

b= $\int f\phi$

A= $-\int \phi_i'\phi_j' + 2\int \phi_i'\phi_j + \int \phi_i\phi_j$

Modify **A** *and* **b** *for boundary conditions*

```
u=la.solve(A,b)
```

Topics

Coding strategy for classes

- Classes contain “hidden” data only
- Information accessed by “accessor” functions only

Details of classes Mesh and Shapefns

- **`Mesh`**
  - **`Mesh(N,a,b)`**
    `N` is number of elements
    `a` is left endpoint
    `b` is right endpoint
  - **`coordinates()`** or **`coordinates(n)`**
    returns all coordinates or $\mathtt{n}^{\text{th}}$ coordinate
  - **`cells()`** or **`cells(n)`**
    returns all nodes numbers of $\mathtt{n}^{\text{th}}$ cell or all node numbers
  - **`size()`**
    returns number of mesh cells
- **`Shapefns`**
  - **`Shapefns()`**
  - **`eval(n,xi)`** returns $\phi_n(\xi)$
  - **`ddx(n,xi)`** returns $\phi_n'(\xi)$
  - **`size()`** returns the number of required nodes

Shapefns

```
class Shapefns(object):
    """
    Define Quadratic Lagrange shape functions
    These will be defined on the (local) interval [0,1], with
        mid point 0.5
    Shapefns()
    eval(n,xi): phi[n](xi)
    ddx(n,xi):  dphi[n](xi)
    size(): number of nodes for these shape functions
    """
```

Shapefns

```
class Shapefns(object):
    """
    Define Quadratic Lagrange shape functions
    These will be defined on the (local) interval [0,1], with
        mid point 0.5
    Shapefns()
    eval(n,xi): phi[n](xi)
    ddx(n,xi):  dphi[n](xi)
    size(): number of nodes for these shape functions
    """

    def __init__(self):
```

Shapefns

```
class Shapefns(object):
    """
    Define Quadratic Lagrange shape functions
    These will be defined on the (local) interval [0,1], with
        mid point 0.5
    Shapefns()
    eval(n,xi): phi[n](xi)
    ddx(n,xi):  dphi[n](xi)
    size(): number of nodes for these shape functions
    """

    def __init__(self):
        """
        an array of functions for phi and deriv phi
        """
        self.__phi=[lambda xi: 2.0 * (xi-0.5) * (xi-1.0) ,\
                    lambda xi: 4.0 * xi * (1.0-xi), \
                    lambda xi: 2.0 * xi * (xi-0.5)]
```

Shapefns

```
class Shapefns(object):
    """
    Define Quadratic Lagrange shape functions
    These will be defined on the (local) interval [0,1], with
        mid point 0.5
    Shapefns()
    eval(n,xi): phi[n](xi)
    ddx(n,xi):  dphi[n](xi)
    size(): number of nodes for these shape functions
    """

    def __init__(self):
        """
        an array of functions for phi and deriv phi
        """
        self.__phi=[lambda xi: 2.0 * (xi-0.5) * (xi-1.0) ,\
                    lambda xi: 4.0 * xi * (1.0-xi), \
                    lambda xi: 2.0 * xi * (xi-0.5)]
        # and dphi (derivative of phi w.r.t. xi)
        # derivative of second factor * first + derivative of first factor * sec
        self.__dphi=[lambda xi: 2.0 * (xi-0.5) + 2.0*(xi - 1.0) ,\
                     lambda xi: -4.0 * xi + 4.0*(1.0 - xi), \
                     lambda xi: 2.0 * xi + 2.0*(xi - 0.5)]
```

Shapefns

```
class Shapefns(object):
    """
    Define Quadratic Lagrange shape functions
    These will be defined on the (local) interval [0,1], with
        mid point 0.5
    Shapefns()
    eval(n,xi): phi[n](xi)
    ddx(n,xi):  dphi[n](xi)
    size(): number of nodes for these shape functions
    """

    def __init__(self):
        """
        an array of functions for phi and deriv phi
        """
        self.__phi=[lambda xi: 2.0 * (xi-0.5) * (xi-1.0) ,\
                    lambda xi: 4.0 * xi * (1.0-xi), \
                    lambda xi: 2.0 * xi * (xi-0.5)]
        # and dphi (derivative of phi w.r.t. xi)
        # derivative of second factor * first + derivative of first factor * sec
        self.__dphi=[lambda xi: 2.0 * (xi-0.5) + 2.0*(xi - 1.0) ,\
                     lambda xi: -4.0 * xi + 4.0*(1.0 - xi), \
                     lambda xi: 2.0 * xi + 2.0*(xi - 0.5)]
        self.__N=3  #number of nodes in quadratic Lagrange polynomial
```

Shapefns, cont'd

```
def eval(self,n,xi):
    """
    the function phi[n](xi), for any xi
    """
    return self.__phi[n](xi)

def ddx(self,n,xi):
    """
    the function dphi[n](xi), for any xi
    """
    return self.__dphi[n](xi)

def size(self):
    """
    the  number of points
    """
    return self.__N
```

Shapefns, cont'd

```
def eval(self,n,xi):
    """
    the function phi[n](xi), for any xi
    """
    return self.__phi[n](xi)

def ddx(self,n,xi):
    """
    the function dphi[n](xi), for any xi
    """
    return self.__dphi[n](xi)

def size(self):
    """
    the  number of points
    """
    return self.__N
```

What about the `Mesh` Class?

Homework, Exercise 11, 10 points

Test it!

- Place code at the end of the file to test it!
- Can preceed it with `if __name__ == '__main__':`
- Test early, test often, don't throw away the testing code!

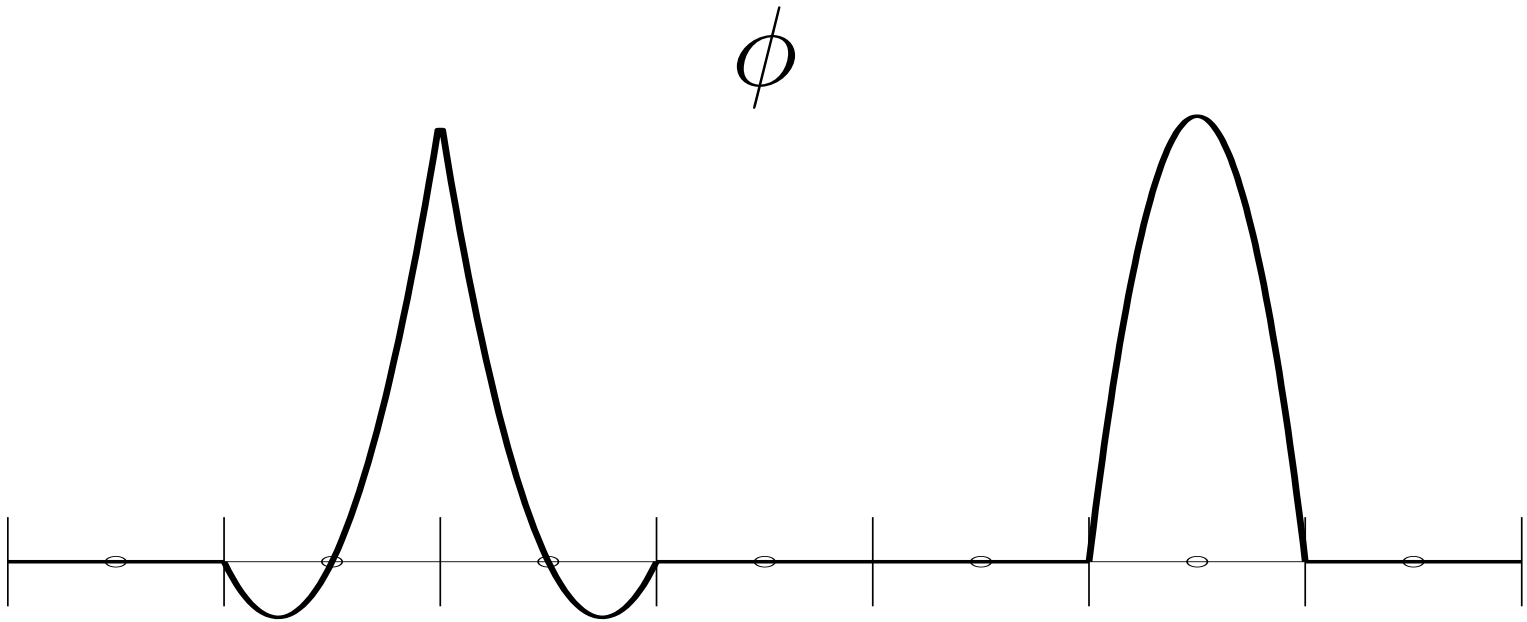
Test code

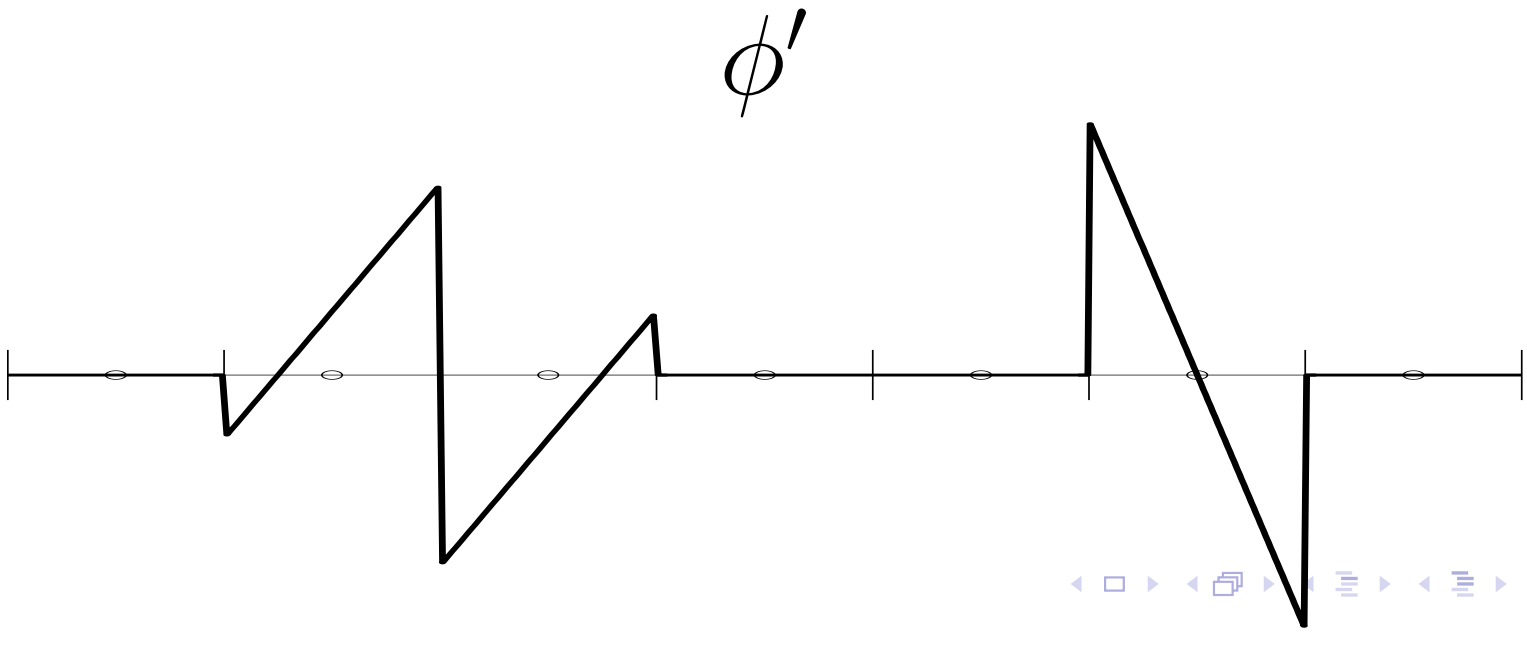
```
N=5
rightpt = 5.0
print "\n\nFEM1D.py Test case, dx=",rightpt/N

mesh = Mesh(N,0.0,rightpt)
coords = mesh.coordinates()
print "mesh.coordinates()=",coords

sfns = Shapefns()
print "sfns.size()-3=", sfns.size()-3
xi = np.linspace(0,1,100)
import matplotlib.pyplot as plt
if True:
    for n in range(3):
        plt.plot(xi,sfns.eval(n,xi))
        plt.show()
        plt.plot(xi,sfns.ddx(n,xi))
        plt.show()
```

Shape function plots





FiniteElement class

- Describes a *single* finite element
- A **`FunctionSpace`** will contain many of these
- Constructed from
  1. **`Mesh`**
  2. shape functions
  3. this element's number (agrees with one of the **`Mesh`** intervals
  4. 3 numbers to use for DOFs.
- Methods:

  `endpts()`: element end points
  `dofpts()`: array of DOF locations
  `dofnos()`: DOF numbers
  `numDofs()`: length of **`dofnos()`**
  `eval(n,x)`: evaluate $\phi_n(x)$ (not ξ)
  `ddxl(n,x)`: evaluate $\phi_n'(x)$ (not ξ)
  `integral(f1,f2,derivative)`: $\int_e f_1(x) f_2(x) \phi_n(x)\, dx$

  - **`f1`** and **`f2`** are optional
  - put ϕ' in integral if **`derivative=True`**
  - returns a vector, one value for each n

FiniteElement constructor

```
def __init__(self,mesh,sfns,eltno,dofnos):
    """
    mesh is the mesh it is built on
    sfns is the Shapefuns member
    eltno is this element's number
    endnos is a pair of ints giving the numbers of the endpoints
        in the mesh
    dofnos is an array of ints giving the numbers of the dofs
    """
```

FiniteElement constructor

```
def __init__(self,mesh,sfns,eltno,dofnos):
    """
    mesh is the mesh it is built on
    sfns is the Shapefuns member
    eltno is this element's number
    endnos is a pair of ints giving the numbers of the endpoints
        in the mesh
    dofnos is an array of ints giving the numbers of the dofs
    """
    # this element no. is same as mesh element no.
    assert(0 <= eltno < mesh.size())
    self.__eltno = eltno
    endnos = mesh.cells(eltno)
    assert(len(endnos) == 2)
```

FiniteElement constructor

```
def __init__(self,mesh,sfns,eltno,dofnos):
    """
    mesh is the mesh it is built on
    sfns is the Shapefuns member
    eltno is this element's number
    endnos is a pair of ints giving the numbers of the endpoints
        in the mesh
    dofnos is an array of ints giving the numbers of the dofs
    """
    # this element no. is same as mesh element no.
    assert(0 <= eltno < mesh.size())
    self.__eltno = eltno
    endnos = mesh.cells(eltno)
    assert(len(endnos) == 2)
    self.__endpts = np.array(mesh.coordinates(endnos))
```

FiniteElement constructor

```
def __init__(self,mesh,sfns,eltno,dofnos):
    """
    mesh is the mesh it is built on
    sfns is the Shapefuns member
    eltno is this element's number
    endnos is a pair of ints giving the numbers of the endpoints
        in the mesh
    dofnos is an array of ints giving the numbers of the dofs
    """
    # this element no. is same as mesh element no.
    assert(0 <= eltno < mesh.size())
    self.__eltno = eltno
    endnos = mesh.cells(eltno)
    assert(len(endnos) == 2)
    self.__endpts = np.array(mesh.coordinates(endnos))
    self.__numDofs = sfns.size()
```

FiniteElement constructor

```
def __init__(self,mesh,sfns,eltno,dofnos):
    """
    mesh is the mesh it is built on
    sfns is the Shapefuns member
    eltno is this element's number
    endnos is a pair of ints giving the numbers of the endpoints
        in the mesh
    dofnos is an array of ints giving the numbers of the dofs
    """
    # this element no. is same as mesh element no.
    assert(0 <= eltno < mesh.size())
    self.__eltno = eltno
    endnos = mesh.cells(eltno)
    assert(len(endnos) == 2)
    self.__endpts = np.array(mesh.coordinates(endnos))
    self.__numDofs = sfns.size()
    assert(sfns.size() == len(dofnos))
    self.__dofnos=dofnos
```

FiniteElement constructor

```
def __init__(self,mesh,sfns,eltno,dofnos):
    """
    mesh is the mesh it is built on
    sfns is the Shapefuns member
    eltno is this element's number
    endnos is a pair of ints giving the numbers of the endpoints
        in the mesh
    dofnos is an array of ints giving the numbers of the dofs
    """
    # this element no. is same as mesh element no.
    assert(0 <= eltno < mesh.size())
    self.__eltno = eltno
    endnos = mesh.cells(eltno)
    assert(len(endnos) == 2)
    self.__endpts = np.array(mesh.coordinates(endnos))
    self.__numDofs = sfns.size()
    assert(sfns.size() == len(dofnos))
    self.__dofnos=dofnos
    self.__dofpts=np.linspace(self.__endpts[0],self.__endpts[1],self.__numDofs)
```

FiniteElement constructor

```
def __init__(self,mesh,sfns,eltno,dofnos):
    """
    mesh is the mesh it is built on
    sfns is the Shapefuns member
    eltno is this element's number
    endnos is a pair of ints giving the numbers of the endpoints
        in the mesh
    dofnos is an array of ints giving the numbers of the dofs
    """
    # this element no. is same as mesh element no.
    assert(0 <= eltno < mesh.size())
    self.__eltno = eltno
    endnos = mesh.cells(eltno)
    assert(len(endnos) == 2)
    self.__endpts = np.array(mesh.coordinates(endnos))
    self.__numDofs = sfns.size()
    assert(sfns.size() == len(dofnos))
    self.__dofnos=dofnos
    self.__dofpts=np.linspace(self.__endpts[0],self.__endpts[1],self.__numDofs)
    self.__sfns=sfns
```

FiniteElement constructor

```
def __init__(self,mesh,sfns,eltno,dofnos):
    """
    mesh is the mesh it is built on
    sfns is the Shapefuns member
    eltno is this element's number
    endnos is a pair of ints giving the numbers of the endpoints
        in the mesh
    dofnos is an array of ints giving the numbers of the dofs
    """
    # this element no. is same as mesh element no.
    assert(0 <= eltno < mesh.size())
    self.__eltno = eltno
    endnos = mesh.cells(eltno)
    assert(len(endnos) == 2)
    self.__endpts = np.array(mesh.coordinates(endnos))
    self.__numDofs = sfns.size()
    assert(sfns.size() == len(dofnos))
    self.__dofnos=dofnos
    self.__dofpts=np.linspace(self.__endpts[0],self.__endpts[1],self.__numDofs)
    self.__sfns=sfns
    # Gauss points and weights: 3-pts are high enough for this
    self.__gausspts = np.array(\
      (.11270166537925831148207346002,.5,.88729833462074168851792653997 8))
    self.__gausswts = np.array((5.0/18.0,8.0/18.0,5.0/18.0))
```

FiniteElement constructor

```
def __init__(self,mesh,sfns,eltno,dofnos):
    """
    mesh is the mesh it is built on
    sfns is the Shapefuns member
    eltno is this element's number
    endnos is a pair of ints giving the numbers of the endpoints
        in the mesh
    dofnos is an array of ints giving the numbers of the dofs
    """
    # this element no. is same as mesh element no.
    assert(0 <= eltno < mesh.size())
    self.__eltno = eltno
    endnos = mesh.cells(eltno)
    assert(len(endnos) == 2)
    self.__endpts = np.array(mesh.coordinates(endnos))
    self.__numDofs = sfns.size()
    assert(sfns.size() == len(dofnos))
    self.__dofnos=dofnos
    self.__dofpts=np.linspace(self.__endpts[0],self.__endpts[1],self.__numDofs)
    self.__sfns=sfns
    # Gauss points and weights: 3-pts are high enough for this
    self.__gausspts = np.array(\
      (.11270166537925831148207346002,.5,.88729833462074168851792653997))
    self.__gausswts = np.array((5.0/18.0,8.0/18.0,5.0/18.0))
    # for efficiency, generate an array of shape functions evaluated
    # at the Gauss points
    self.__gaussvals = np.empty([self.__numDofs,self.__gausspts.size])
    for n in range(self.__numDofs):
        self.__gaussvals[n,:]=sfns.eval(n,self.__gausspts[:])
```

FiniteElement accessor methods

```
def endpts(self):
    """ access endpoints """
    return self.__endpts

def dofpts(self):
    """ access dofpoints """
    return self.__dofpts

def dofnos(self):
    """ access dof point numbers """
    return self.__dofnos

def numDofs(self):
    """ access numDofs """
    return self.__numDofs
```

FiniteElement ϕ evaluation methods

```
def eval(self,n,x):
    """
    evaluate the n-th shape function on this element
    at the spatial coordinate x
    """
    # map x to xi
    xx=np.array(x)
    xi=(xx-self.__endpts[0])/(self.__endpts[1]-self.__endpts[0])
    # evaluate
    return self.__sfns.eval(n,xi)*(xi >= 0.)*(xi <= 1.)

def ddx(self,n,x):
    """
    evaluate the n-th shape function on this element
    at the spatial coordinate x
    """
    # map x to xi
    xi=(x-self.__endpts[0])/(self.__endpts[1]-self.__endpts[0])
    # evaluate
    return self.__sfns.ddx(n,xi)*(xi>=0.)*(xi <= 1.0)
```

Gauß integration on the reference element

- $Q = \sum_i f(\xi_i) w_i$
- Integration points (“Gauß points”) tabulated
- Weights w_i tabulated
- Integral on the true element requires affine transformation, $x = L\xi + x_0$
- Assume functions f are given at DOF points.

FiniteElement integration

```
def integral(self, f1=None, f2=None, derivative=False):
    """
    Integrate either phi[i](xi)*f1(xi)*f2(xi) or dphi[i]*f1*f2
    over this element, depending on if derivative is False or True
    Returns a vector of 3 results, one for
    phi[0], one for phi[1], and one for phi[2].
    f1 and f2 are assumed to have been mapped to this element
      as arrays
    if derivative is True, phi is replaced with dphi
    """
    L = self.__endpts[1]-self.__endpts[0]  # length of element
    t = self.__gausswts.copy()
    gp = self.__gausspts
```

FiniteElement integration

```
def integral(self, f1=None, f2=None, derivative=False):
    """
    Integrate either phi[i](xi)*f1(xi)*f2(xi) or dphi[i]*f1*f2
    over this element, depending on if derivative is False or True
    Returns a vector of 3 results, one for
    phi[0], one for phi[1], and one for phi[2].
    f1 and f2 are assumed to have been mapped to this element
      as arrays
    if derivative is True, phi is replaced with dphi
    """
    L = self.__endpts[1]-self.__endpts[0]  # length of element
    t = self.__gausswts.copy()
    gp = self.__gausspts
```

FiniteElement integration

```
def integral(self, f1=None, f2=None, derivative=False):
    """
    Integrate either phi[i](xi)*f1(xi)*f2(xi) or dphi[i]*f1*f2
    over this element, depending on if derivative is False or True
    Returns a vector of 3 results, one for
    phi[0], one for phi[1], and one for phi[2].
    f1 and f2 are assumed to have been mapped to this element
      as arrays
    if derivative is True, phi is replaced with dphi
    """
    L = self.__endpts[1]-self.__endpts[0]  # length of element
    t = self.__gausswts.copy()
    gp = self.__gausspts

    if f1 != None:
      assert(len(f1) == self.__numDofs)
      fvals = np.zeros([self.__gausspts.size])
      for n in range(self.__numDofs):
          fvals += f1[n]*self.__gaussvals[n,:]
      t*=fvals
```

FiniteElement integration

```
def integral(self, f1=None, f2=None, derivative=False):
    """
    Integrate either phi[i](xi)*f1(xi)*f2(xi) or dphi[i]*f1*f2
    over this element, depending on if derivative is False or True
    Returns a vector of 3 results, one for
    phi[0], one for phi[1], and one for phi[2].
    f1 and f2 are assumed to have been mapped to this element
      as arrays
    if derivative is True, phi is replaced with dphi
    """
    L = self.__endpts[1]-self.__endpts[0]  # length of element
    t = self.__gausswts.copy()
    gp = self.__gausspts

    if f1 != None:
      assert(len(f1) == self.__numDofs)
      fvals = np.zeros([self.__gausspts.size])
      for n in range(self.__numDofs):
          fvals += f1[n]*self.__gaussvals[n,:]
      t*=fvals

    if f2 != None:
      assert(len(f2) == self.__numDofs)
      fvals = np.zeros([self.__gausspts.size])
      for n in range(self.__numDofs):
          fvals += f2[n]*self.__gaussvals[n,:]
      t *= fvals
```

FiniteElement integration cont'd

```
if derivative:
    # really: t *= L*(1/L)
    q = np.dot(np.array([self.__sfns.ddx(0,gp), \
                         self.__sfns.ddx(1,gp), \
                         self.__sfns.ddx(2,gp)]),t)
else:
    t *= L  # correct for affine map x->xi
    q = np.dot(np.array([self.__sfns.eval(0,gp), \
                         self.__sfns.eval(1,gp), \
                         self.__sfns.eval(2,gp)]),t)

return q
```

Test it!

Test early! Test often! Don't discard your test code!
Compare *vs.* MATLAB symbolic toolbox

```
elt = FiniteElement(mesh,sfns,0,[0,1,2])

if N == 5 and np.abs(coords[-1]-5.0) < 1.e-10:
    # test some integrals
    print "elt integral() err=",max(abs(elt.integral()-[1./6,2./3,1./6]))
    print "integral(deriv) err=",\
          max(abs(elt.integral(derivative=True)-[-1,0,1]))
```

Test it!

Test early! Test often! Don't discard your test code!
Compare *vs.* MATLAB symbolic toolbox

```
elt = FiniteElement(mesh,sfns,0,[0,1,2])

if N == 5 and np.abs(coords[-1]-5.0) < 1.e-10:
    # test some integrals
    print "elt integral() err=",max(abs(elt.integral()-[1./6,2./3,1./6]))
    print "integral(deriv) err=",\
          max(abs(elt.integral(derivative=True)-[-1,0,1]))
    # pick the function f(x)=x, find its expansion coefs
    ex = np.empty([sfns.size()])
    ex[0] = elt.endpts()[0]
    ex[2] = elt.endpts()[1]
    ex[1] = .5*(ex[0]+ex[2])
    ex2 = ex**2
    print "integral(x) err=",max(abs(elt.integral(ex)-[0,1./3,1./6]))
```

Test it!

Test early! Test often! Don't discard your test code!
Compare *vs.* MATLAB symbolic toolbox

```
elt = FiniteElement(mesh,sfns,0,[0,1,2])

if N == 5 and np.abs(coords[-1]-5.0) < 1.e-10:
    # test some integrals
    print "elt integral() err=",max(abs(elt.integral()-[1./6,2./3,1./6]))
    print "integral(deriv) err=",\
          max(abs(elt.integral(derivative=True)-[-1,0,1]))
    # pick the function f(x)=x, find its expansion coefs
    ex = np.empty([sfns.size()])
    ex[0] = elt.endpts()[0]
    ex[2] = elt.endpts()[1]
    ex[1] = .5*(ex[0]+ex[2])
    ex2 = ex**2
    print "integral(x) err=",max(abs(elt.integral(ex)-[0,1./3,1./6]))
    print "integral(x**2) err=",max(abs(elt.integral(ex2)-[-1./60,1./5,3./20]))
    print "integral(x**2) err=",max(abs(elt.integral(ex,ex)-[-1./60,1./5,3./20])
```

Test it!

Test early! Test often! Don't discard your test code!
Compare *vs.* MATLAB symbolic toolbox

```
elt = FiniteElement(mesh,sfns,0,[0,1,2])

if N == 5 and np.abs(coords[-1]-5.0) < 1.e-10:
    # test some integrals
    print "elt integral() err=",max(abs(elt.integral()-[1./6,2./3,1./6]))
    print "integral(deriv) err=",\
          max(abs(elt.integral(derivative=True)-[-1,0,1]))
    # pick the function f(x)=x, find its expansion coefs
    ex = np.empty([sfns.size()])
    ex[0] = elt.endpts()[0]
    ex[2] = elt.endpts()[1]
    ex[1] = .5*(ex[0]+ex[2])
    ex2 = ex**2
    print "integral(x) err=",max(abs(elt.integral(ex)-[0,1./3,1./6]))
    print "integral(x**2) err=",max(abs(elt.integral(ex2)-[-1./60,1./5,3./20]))
    print "integral(x**2) err=",max(abs(elt.integral(ex,ex)-[-1./60,1./5,3./20])
    print "integral(x,phi') err=",\
          max(abs(elt.integral(ex,derivative=True)-[-1./6,-2./3,5./6]))
    print "integral(x**2,phi') err=",\
          max(abs(elt.integral(ex2,derivative=True)-[0,-2./3,2./3]))
```

Test results

```
elt integral() err= 1.11022302463e-16
integral(deriv) err= 0.0
integral(x) err= 5.55111512313e-17
integral(x**2) err= 2.77555756156e-17
integral(x**2) err= 2.77555756156e-17
integral(x,phi') err= 1.11022302463e-16
integral(x**2,phi') err= 2.22044604925e-16
```

FunctionSpace class

- A **`FunctionSpace`** has a list of elements
- Elements are numbered as in mesh
- Constructed from **`mesh`** and **`Shapefns`**
- Methods:

  `size()`: number of elements

  `ndofs()`: number of all dofs

  `dofpts()`: coordinates of dof[n] or all dofs

  `int_phi_phi(c,derivative)`: $\int c(x)\phi_i(x)\phi_j(x)\,dx$
  - c is optional
  - **`derivative`** is a pair of booleans determining whether to use ϕ or ϕ'
  - returns 2D array of values

  `int_phi(f,derivative)`: $\int f(x)\phi_i(x)\,dx$
  - f is optional
  - **`derivative`** is a boolean determining ϕ or ϕ'
  - returns vector of values

FunctionSpace constructor

```
def __init__(self,mesh,sfns):
    """
    mesh is the mesh
    sfns is the Shapefuns
    """
```

FunctionSpace constructor

```
def __init__(self,mesh,sfns):
    """
    mesh is the mesh
    sfns is the Shapefuns
    """
    self.__size=mesh.size()
    # number the elements in same way as mesh
    self.__elts = list([])
    self.__dofpts = list([])
    self.__nDOFs=0
```

FunctionSpace constructor

```
def __init__(self,mesh,sfns):
    """
    mesh is the mesh
    sfns is the Shapefuns
    """
    self.__size=mesh.size()
    # number the elements in same way as mesh
    self.__elts = list([])
    self.__dofpts = list([])
    self.__nDOFs=0
    for n in range(self.__size):
        # ASSUMING only boundary points are number 0 and (self.__size)
        if n == 0:
            self.__nDOFs += 3
            dofs = [2*n, 2*n+1, 2*n+2]
            newdofs = range(3)
        else:
            self.__nDOFs += 2
            dofs=[2*n, 2*n+1, 2*n+2]
            newdofs = range(1,3)
        fe = FiniteElement(mesh,sfns,n,dofs)
        self.__elts.append(fe)
        for i in newdofs:
            self.__dofpts.append(fe.dofpts()[i])
```

FunctionSpace constructor

```
def __init__(self,mesh,sfns):
    """
    mesh is the mesh
    sfns is the Shapefuns
    """
    self.__size=mesh.size()
    # number the elements in same way as mesh
    self.__elts = list([])
    self.__dofpts = list([])
    self.__nDOFs=0
    for n in range(self.__size):
        # ASSUMING only boundary points are number 0 and (self.__size)
        if n == 0:
            self.__nDOFs += 3
            dofs = [2*n, 2*n+1, 2*n+2]
            newdofs = range(3)
        else:
            self.__nDOFs += 2
            dofs=[2*n, 2*n+1, 2*n+2]
            newdofs = range(1,3)
        fe = FiniteElement(mesh,sfns,n,dofs)
        self.__elts.append(fe)
        for i in newdofs:
            self.__dofpts.append(fe.dofpts()[i])
```

FunctionSpace constructor

```
def __init__(self,mesh,sfns):
    """
    mesh is the mesh
    sfns is the Shapefuns
    """
    self.__size=mesh.size()
    # number the elements in same way as mesh
    self.__elts = list([])
    self.__dofpts = list([])
    self.__nDOFs=0
    for n in range(self.__size):
        # ASSUMING only boundary points are number 0 and (self.__size)
        if n == 0:
            self.__nDOFs += 3
            dofs = [2*n, 2*n+1, 2*n+2]
            newdofs = range(3)
        else:
            self.__nDOFs += 2
            dofs=[2*n, 2*n+1, 2*n+2]
            newdofs = range(1,3)
        fe = FiniteElement(mesh,sfns,n,dofs)
        self.__elts.append(fe)
        for i in newdofs:
            self.__dofpts.append(fe.dofpts()[i])
    self.__dofpts = np.array(self.__dofpts)
```

FunctionSpace `int_phi_phi`

- **`derivative`** is now a *pair* of booleans
- **`A`** is a large matrix **`nDOFs X nDOFs`**
- Elements have to be assembled into **`A`** in the right places
- Trick: treat first ϕ like f when call elemental integration

int_phi_phi code

```
def int_phi_phi(self, c=None, derivative=[False,False]):
    """
    assemble $\int c(x)\phi(x)\phi(x) dx$ or with $d\phi/dx$
    """
    A = np.zeros([self.__nDOFs,self.__nDOFs])
    # loop over elements
    for elt in self.__elts:
        d0=elt.dofnos()



        N = elt.numDofs()


        for j in range(N):



                phi = elt.eval(j,elt.dofpts())
            A[d0,d0[j]] += elt.integral(phi,cc,derivative=derivative[0])
    return A
```

int_phi_phi code

```
def int_phi_phi(self, c=None, derivative=[False,False]):
    """
    assemble $\int c(x)\phi(x)\phi(x) dx$ or with $d\phi/dx$
    """
    A = np.zeros([self.__nDOFs,self.__nDOFs])
    # loop over elements
    for elt in self.__elts:
        d0=elt.dofnos()
        if c != None:
            cc = c[d0]
        else:
            cc = None
        N = elt.numDofs()


        for j in range(N):



                phi = elt.eval(j,elt.dofpts())
            A[d0,d0[j]] += elt.integral(phi,cc,derivative=derivative[0])
    return A
```

int_phi_phi code

```
def int_phi_phi(self, c=None, derivative=[False,False]):
    """
    assemble $\int c(x)\phi(x)\phi(x) dx$ or with $d\phi/dx$
    """
    A = np.zeros([self.__nDOFs,self.__nDOFs])
    # loop over elements
    for elt in self.__elts:
        d0=elt.dofnos()
        if c != None:
            cc = c[d0]
        else:
            cc = None
        N = elt.numDofs()
        endpts=elt.endpts()
        L = endpts[1]-endpts[0] # length of elt
        for j in range(N):
           if derivative[1]:
               # chain rule: d(xi)/d(x)=1/L
               phi = elt.ddx(j,elt.dofpts())/L
           else:
               phi = elt.eval(j,elt.dofpts())
           A[d0,d0[j]] += elt.integral(phi,cc,derivative=derivative[0])
    return A
```

Exercise 12 (5 points)

Complete the code for **`FunctionSpace.int_phi`** Be sure to test your code!

Test it!

Test early! Test often! Don't discard your test code!

```
V=FunctionSpace(mesh,sfns)
print "V.Ndofs()-correct=",V.Ndofs()-(2*N+1)
print "V.size()-correct=",V.size()-N

x = V.dofpts()
f = x.copy()
print "error in integral x over [",x[0],",",x[-1],"]=",\
    np.sum(V.int_phi(f))-x[-1]**2/2.
f = 0.0*x+1
print "error in integral 1 over [",x[0],",",x[-1],"]=",\
    np.sum(V.int_phi(f))-x[-1]
f = x.copy()**2
print "error in integral x**2 over [",x[0],",",x[-1],"]=",\
    np.sum(V.int_phi(f))-x[-1]**3/3.
f = x.copy()**3
print "error in integral x**3 over [",x[0],",",x[-1],"]=",\
    np.sum(V.int_phi(f))-x[-1]**4/4.
f = x.copy()**4
print "error in integral x**4 over [",x[0],",",x[-1],"]=",\
    np.sum(V.int_phi(f))-x[-1]**5/5.," should be nonzero."

print "norm(V.dofpts()-correct)=",\
       la.norm(V.dofpts()-np.linspace(0,coords[-1],2*N+1))
```

Note that $\sum_i \phi_i(x) = 1$.

Test results

```
V.Ndofs()-correct= 0
V.size()-correct= 0
error in integral x over [ 0.0 , 5.0 ]= 0.0
error in integral 1 over [ 0.0 , 5.0 ]= 8.881784197e-16
error in integral x**2 over [ 0.0 , 5.0 ]= 0.0
error in integral x**3 over [ 0.0 , 5.0 ]= 2.84217094304e-14
error in integral x**4 over [ 0.0 , 5.0 ]= 0.0416666666667  should be nonzero.
norm(V.dofpts()-correct)= 0.0
```

Test FunctionSpace matrix integration

$$A_{ij} = \int \phi_i \phi_j$$

```
A=V.int_phi_phi()
if N == 5 and np.abs(coords[-1]-5.0) < 1.e-10:
    print "error A00=",A[0,0]-2./15.
    print "error A01=",A[0,1]-1./15.
    print "error A02=",A[0,2]+1./30.
    print "error A11=",A[1,1]-8./15.
    print "error A12=",A[1,2]-1./15.
    print
    print "error A22=",A[2,2]-4./15.
    print "error A23=",A[2,3]-1./15.
    print "error A24=",A[2,4]+1./30.
    print "error A33=",A[3,3]-8./15.
    print "error A34=",A[3,4]-1./15.
    print

# trivial check with coefficient
c = np.ones([Ndofs])
A1 = V.int_phi_phi(c)
print "Norm difference matrices=",la.norm(A-A1)
```

More test FunctionSpace matrix integration

$$B_{ij} = \int c\phi_i\phi_j$$

```
c = (1.0+x)
B = V.int_phi_phi(c)
if N == 5 and np.abs(coords[-1]-5.0) < 1.e-10:
    print "error B00=",B[0,0]-3./20.
    print "error B01=",B[0,1]-1./15.
    print "error B02=",B[0,2]+1./20.
    print "error B11=",B[1,1]-12./15.
    print "error B12=",B[1,2]-2./15.
    print
    print "error B22=",B[2,2]-8./15.
    print "error B23=",B[2,3]-2./15.
    print "error B24=",B[2,4]+1./12.
    print "error B33=",B[3,3]-4./3.
    print "error B34=",B[3,4]-3./15.
```

Test FunctionSpace Laplace matrix integration

$$C_{ij} = \int \phi'_i \phi'_j$$

```
C = V.int_phi_phi(derivative=[True,True])
if N == 5 and np.abs(coords[-1]-5.0) < 1.e-10:
    print "\n Laplace Matrix"
    print "error C00*3=",C[0,0]-7./3.
    print "error C01*3=",C[0,1]+8./3.
    print "error C02*3=",C[0,2]-1./3.
    print "error C11*3=",C[1,1]-16./3.
    print "error C12*3=",C[1,2]+8./3.
    print
    print "error C22*3=",C[2,2]-14./3.
    print "error C23*3=",C[2,3]+8./3.
    print "error C24*3=",C[2,4]-1./3.
    print "error C33*3=",C[3,3]-16./3.
    print "error C34*3=",C[3,4]+8./3.
    print
```

Test FunctionSpace Laplace matrix multiply

$$C_{ij} = \int \phi_i' \phi_j'$$

```
soln2 = np.ones([Ndofs])
b2 = np.dot(C,soln2)
print "const soln Laplace, norm check=",la.norm(b2)

soln = x
b0 = np.dot(C,soln)
rhs0 = V.int_phi(np.zeros([Ndofs]))
# natural b.c. not satisfied, don't check them
rhs0[0] = -b0[0]
rhs0[-1] = -b0[-1]
print "soln=x Laplace, norm check=",la.norm(rhs0+b0)

soln = x**2
b1 = np.dot(C,soln)
rhs1 = V.int_phi(2.0*np.ones([Ndofs]))
# natural b.c. not satisfied on right, don't check it
rhs1[-1] = -b1[-1]
print "soln=x**2 Laplace, norm check=",la.norm(rhs1+b1)
```

Test FunctionSpace “derivative” matrix multiply

$$D_{ij} = \int \phi_i \phi'_j$$

```
D = V.int_phi_phi(derivative=[False,True])
soln = np.ones([V.Ndofs()])
b2 = np.dot(D,soln)
print "norm check (rhs d/dx+Neumann, const soln)=",la.norm(b2)

D[0,0] = 1.0
D[0,1:] = 0.0
D[-1,-1] = 1.0
D[-1,0:-1] = 0.0
soln = x
b3 = np.dot(D,soln)
rhs3 = V.int_phi(np.ones([Ndofs]))
rhs3[0] = soln[0]
rhs3[-1] = soln[-1]
print "norm check (d/dx+Dirichlet soln=x)=",la.norm(rhs3-b3)
```

Topics

Verification

- Solve an easy problem like yours!
  - “Verification”
  - Is your mesh fine enough?
  - Is the program capable of solving your problem?
- Exact solutions are excellent choices

$$u'' + 2u' + u = f \qquad x \in [0, 1]$$

1. $u = 1, f = 1$, Dirichlet b.c.
2. $u = x, f = 1 + x$, Dirichlet b.c.
3. $u = x^2, f = 2 + 2x + x^2$, Dirichlet b.c.

Verification function

```
def verification(title,N,rhsfn,exactfn):
    """
    generic verification runs for PDE u"+2*u'+u=rhs on [0,1]
    Dirichlet b.c.
    title= descriptive title
    N=number of elements
    rhsfn=function for rhs as function of x
    exactfn=function for exact solution as function of x
    MMS 4/16/14
    """
    # N elements in [0,1]
    mesh = Mesh(N,0.0,1.0)

    # shape functions, function space
    sfns = Shapefns()
    V = FunctionSpace(mesh,sfns)

    # rhs and exact
    x = V.dofpts()
    exact = exactfn(x)

    rhs = rhsfn(x)
    b = V.int_phi(rhs)
```

Verification function cont'd

```
# assemble stiffness matrix: u"+2*u'+u
# integration by parts on first term introduces minus sign
A=  -V.int_phi_phi(derivative=[True,True]) \
  +2*V.int_phi_phi(derivative=[False,True]) \
    +V.int_phi_phi()

# insert boundary conditions
# left bndry u=1 (from exact solution)
A[0,0] = 1.0
A[0,1:] = 0.0
b[0] = exact[0]
# right bndry u=1 (from exact solution)
A[-1,-1] = 1.0
A[-1,0:-1] = 0.0
b[-1] = exact[-1]

# solve
u = la.solve(A,b)

# check
print title," relative error=",la.norm(u-exact)/la.norm(exact)
```

Run verification

```
verification("Case 1", 5, rhsfn=lambda(x):0.0*x+1.0, \
    exactfn=lambda(x):0.0*x+1.0)
verification("Case 2", 5, rhsfn=lambda(x):x+2.0, \
    exactfn=lambda(x):x)
verification("Case 3", 5, rhsfn=lambda(x):x**2+4*x+2.0, \
    exactfn=lambda(x):x**2)
```

Solve the given problem

Homework, Exercise 13, 10 points

Convergence estimate

N	Relative Error	Error Ratio
5	5.09313786541e-06	15.85
10	3.21248851405e-07	15.91
20	2.01935531431e-08	15.94
40	1.26705299734e-09	15.86
80	7.98849978221e-11	7.18
160	1.11333740702e-11	

- Appears to be $O(h^4)$
- Only expect $O(h^3)$
- Superconvergent at the nodes because mesh is uniform?