```
#!/sw/bin/python2.5


if __name__ == '__main__':
    #import some extension modules
    from math import *
    from Numeric import *
    from tables import *
    #1. Generate the nodes and elements arrays on [0,Lx] x [0,Ly]
    #domain
    Lx = 1.0 #dynamic typing (no declarations), notice no ;'s
    Ly = 1.0

    #generate mesh
    nx = ny = 2**3+1
    hx = Lx/(nx-1.0)
    hy = Ly/(ny-1.0)

    nNodes = nx*ny
    nElements = 2*(nx-1)*(ny-1)
    #nodes
    nodes = zeros((nNodes,3),Float) #multidimensional array
    for i in range(ny):  #loops over lists of integers, notice indentation and no {},
        for j in range(nx):
            nN = i*nx + j
            nodes[nN,0] = j*hx
            nodes[nN,1] = i*hy
    #elements
    elements = zeros((nElements,3),Int)
    for ci in range(ny-1):
        for cj in range(nx-1):
            #subdivide element by placing diagonal from
            #lower left to upper right
            #upper left element, go counterclockwise around nodes
            eN = 2*(ci*(nx-1) + cj)
            elements[eN+1,0] = ci*nx + cj
            elements[eN+1,1] = (ci+1)*nx + cj + 1
            elements[eN+1,2] = (ci+1)*nx+cj
            #lower right element
            elements[eN,0] = ci*nx + cj
            elements[eN,1] = ci*nx + cj + 1
            elements[eN,2] = (ci+1)*nx+cj+1
    #2. Evaluate J,J^{-1} and det(J) for the linear mapping form $T_R$ to $T_e$
    #
    #basis functions and gradients on reference element
    #nodes of reference element (ordered counterclockwise like physical elements)
    xi = array([[0.0,0.0],
                [1.0,0.0],
                [0.0,1.0]])
    def psi0(x): #function definitions
        return 1.0 - x[0] - x[1]
    def psi1(x):
        return x[0]
    def psi2(x):
```

```
        return x[1]
    psi = [psi0,psi1,psi2]
    grad_psi = array([[-1.0,-1.0],
                      [1.0,0.0],
                      [0.0,1.0]])
    #evaluate Jacobians and inverse Jacobians
    J=zeros((nElements,2,2),Float)
    Jinv=zeros((nElements,2,2),Float)
    detJ=zeros((nElements,),Float)
    for eN,elementNodes in enumerate(elements):
        for nN_element,nN_global in enumerate(elementNodes):
            x = nodes[nN_global,0]
            y = nodes[nN_global,1]
            J[eN,0,0] += x*grad_psi[nN_element,0]
            J[eN,0,1] += x*grad_psi[nN_element,1]
            J[eN,1,0] += y*grad_psi[nN_element,0]
            J[eN,1,1] += y*grad_psi[nN_element,1]
        detJ[eN] = J[eN,0,0]*J[eN,1,1] - J[eN,0,1]*J[eN,1,0]
        Jinv[eN,0,0] =  J[eN,1,1]/detJ[eN]
        Jinv[eN,0,1] = -J[eN,0,1]/detJ[eN]
        Jinv[eN,1,0] = -J[eN,1,0]/detJ[eN]
        Jinv[eN,1,1] =  J[eN,0,0]/detJ[eN]
    #3. Evaluate the stiffness matrix
    #
    #(stiffness) matrix
    A = zeros((nNodes,nNodes),Float)
    nodeStar = [set() for i in range(len(nodes))] #high-level set data structure
    grad_x_psi=zeros((3,2),Float)
    for eN,elementNodes in enumerate(elements):
        #build basis function gradients in physical space for this element
        grad_x_psi[:]=0.0
        for i_local in range(3):
            for ii in range(2):
                for jj in range(2):
                    grad_x_psi[i_local,ii] += Jinv[eN,jj,ii]*grad_psi[i_local,jj]
        for i_local,i_global in enumerate(elementNodes):
            for j_local,j_global in enumerate(elementNodes):
                nodeStar[i_global].add(j_global)
                A[i_global,j_global] += 0.5*((grad_x_psi[j_local,0]*
                                              grad_x_psi[i_local,0]
                                              +
                                              grad_x_psi[j_local,1]*
                                              grad_x_psi[i_local,1])
                                             *fabs(detJ[eN]))
    #4. Calculate source  term
    #
    #solution and source
    k_x = 2.0
    k_y = 5.0
    def u(x):
        return sin(k_x * pi * x[0])*sin(k_y * pi * x[1])
    def f(x):
        return pi**2 * (k_x**2 + k_y**2)*sin(k_x * pi * x[0])*sin(k_y * pi * x[1])
    #4. Evaluate the load vector using nodal quadrature rule.
```

```
    #
    #righ hand side (load) vector
    b = zeros((nNodes,),Float)
    for eN,elementNodes in enumerate(elements):
        for i_local,i_global in enumerate(elementNodes):
            for j_local,j_global in enumerate(elementNodes):
                b[i_global] += (psi[i_local](xi[j_local])
                                *f(nodes[i_global])*fabs(detJ[eN])/6.0)

    #Set Dirichlet boundary conditions by
    #replacing equation for nodes on boundaries with
    #u = g
    #

    #5. Set Dirichlet conditions on the boundary by replacin rows.
    #
    #For this  problem we have u=0 on all of the boundary
    #y=0,Ly
    for j in range(nx):
        #y=0
        n = j
        for m in nodeStar[n]:
            A[n,m]=0.0
        A[n,n]=1.0
        b[n] = 0.0
        #y=Ly
        n = (ny-1)*nx + j
        for m in nodeStar[n]:
            A[n,m]=0.0
        A[n,n]=1.0
        b[n] = 0.0
    #x=0,Lx
    for i in range(ny):
        #x=0
        n = i*nx
        for m in nodeStar[n]:
            A[n,m]=0.0
        A[n,n]=1.0
        b[n] = 0.0
        #x=Lx
        n = i*nx + nx - 1
        for m in nodeStar[n]:
            A[n,m]=0.0
        A[n,n]=1.0
        b[n] = 0.0

    #6. Solve the system using any method.
    #
    #solve system with Gauss-Seidel
    uh = zeros((nNodes,),Float)
    ua = zeros((nNodes,),Float)
    r = zeros((nNodes,),Float)
    maxIts = 10000
    rNorm0 = 0.0
```

```
    for n in range(nNodes):
        r[n] = b[n]
        for m in nodeStar[n]:
            r[n] -= A[n,m]*uh[m]
        rNorm0 += r[n]*r[n]
    rNorm0 = sqrt(rNorm0)
    for its in range(maxIts):
        rNorm=0.0
        for n in range(len(nodes)):
            r[n] = b[n]
            for m in nodeStar[n]:
                r[n] -= A[n,m]*uh[m]
            rNorm += r[n]*r[n]
            uh[n] += r[n]/A[n,n]
        rNorm = sqrt(rNorm)
        if rNorm < 1.0e-8*rNorm0:
            print "converged ",rNorm,its
            break
    else:
        print "failed to converge in maxIts iterations",rNorm

    #7. Approximate the L_2 norm of the error using the nodal
    #quadrature formula. Make a table of the errors for h,h/2,h/4,h/8
    #
    #calculate error in the L_2 norm
    L2err=0.0
    for eN,nodeList in enumerate(elements):
        for i_global in nodeList:
            L2err += (uh[i_global] - u(nodes[i_global]))**2 * fabs(detJ[eN])/6.0
    print "L2 error",sqrt(L2err)
    #8. Write the approximate solution to a file and plot the result
    #

    #hdf5 file
    h5  = openFile('homework3.h5',mode='w',title="homework3 HDF5")
    elements_h5 = h5.createArray("/",'Elements',elements,'Elements')
    nodes_h5 = h5.createArray("/",'Nodes',nodes,'Nodes')
    solution_h5 = h5.createArray("/",'NumericalSolution',uh,'NumericalSolution')
    for i in range(nNodes):
        ua[i] = u(nodes[i])
    analyticalSolution_h5 = h5.createArray("/",
                                           'AnalyticalSolution',
                                           ua,
                                           'AnalyticalSolution')
    h5.close()
    #xml file
    xml = open('homework3.xmf','w')
    xml.write("""<?xml version="1.0" ?>
<!DOCTYPE Xdmf SYSTEM "Xdmf.dtd" [
<!ENTITY HeavyData "homework3.h5" >
]>
<Xdmf>
<Domain>
""")
```

```
    #format text of xmf file using triple quoted string and substitution
    xmlContents = """<Grid Name="homework3_triangular_mesh">
  <Topology Type="Triangle" NumberOfElements="%i">""" % (nElements,) + """
    <DataStructure Format="HDF" DataType="Int" Dimensions="%i %i">""" % (nElements,
                                                                          3) + """
      &HeavyData;:/Elements
    </DataStructure>
  </Topology>
  <Geometry Type="XYZ">
    <DataStructure Format="HDF" DataType="Float" Dimensions="%i %i">""" % (nNodes,
                                                                            3) + """
      &HeavyData;:/Nodes
    </DataStructure>
  </Geometry>
  <Attribute Name="u" AttributeType="Scalar" Center="Node">
    <DataStructure Format="HDF" DataType="Float" Dimensions="%i %i">""" % (nNodes,
                                                                            1) + """
      &HeavyData;:/NumericalSolution
    </DataStructure>
  </Attribute>
  <Attribute Name="ua" AttributeType="Scalar" Center="Node">
    <DataStructure Format="HDF" DataType="Float" Dimensions="%i %i">""" % (nNodes,
                                                                            1) + """
      &HeavyData;:/AnalyticalSolution
    </DataStructure>
  </Attribute>
</Grid>
</Domain>
</Xdmf>
"""
    xml.write(xmlContents)
    xml.close()
```