**ISC 5935 - Computational Tools for Finite Elements**
Homework # 4 - Solution

---

1. The table I got.

```
N     L2 ratios     H1 ratios
 4    3.913042      1.950864
 8    3.978432      1.987806
16    3.994618      1.996957
32    3.998655      1.999240
64    3.999664      1.999810
```

This table shows the factor by which the error decreases as we double the number of elements. Doubling the number of elements also halves the $H^1$ error and reduces the $L^2$ error by a factor 4. Let $h = 1/n$. We speculate that the $L^2$ error decays like $O(h^2)$, while the $H^1$ error decays like $O(h)$.

Here's the code snippet for computing the error:

```
def compute_error(ua,ue,uep,x,xg,wg):
  """
  Compute the integral of a function, using a composite Gauss rule over
  all elements.
  Inputs:
    ua       - double, array of finite element coefficients
    ue       - function, exact solution
    uep      - function, derivative of the exact solution
    x        - double, list of finite element nodes.
    xg       - double, Gauss nodes on the interval [0,1]
    wg       - double, Gauss weights

  Outputs:
    error_l2 - double, L2 error
    error_h1 - double, H1 error
  """
  # Initialize the the L2 and H10 errors
  l2_error = 0.0
  h1_error = 0.0
```

```
n = len(x)-1

for e in range(n):

  l = e
  r = e + 1

  xl = x[l]
  xr = x[r]

  # Consider quadrature point Q: (0, 1, 2 ) in element E.
  for q in range ( len(xg) ):
    #  Map XG and WG from [0,1] to
    #      XQ and QQ in [XL,XR].
    xq = xl + xg[q] * ( xr - xl )
    wq = wg[q] * ( xr - xl )

    #  Evaluate at XQ the basis functions and derivatives for XL and XR.
    phil = ( xr - xq  ) / ( xr - xl )
    philp = - 1.0 / ( xr - xl )

    phir = ( xq - xl ) / ( xr - xl )
    phirp = 1.0 / ( xr - xl )


    # Evaluate at XQ the finite element approximation
    ua_loc  = ua[l]*phil + ua[r]*phir
    uap_loc = ua[l]*philp + ua[r]*phirp

    #  Evaluate the exact function and its derivative at XQ
    ue_loc  = exact_fn(xq)
    uep_loc = exact_fn_der(xq)

    # Compute the
    l2_error = l2_error + wq * (ua_loc - ue_loc)**2.0
    h1_error = h1_error + wq * (uap_loc - uep_loc)**2.0


# Take square roots
```

```
 l2_error = np.sqrt(l2_error)
 h1_error = np.sqrt(h1_error)

 return l2_error, h1_error
```

2. Output:

```
The spatial average of u:
n   u_ave
2   0.028846
4   0.035648
8   0.037325
```

Here is the code that computes the average:

```
def compute_average(ua,x,xg,wg):
 """
 Compute the spatial average of a finite element function ua

 Inputs:
   ua - double, finite element coefficients of the function
   x  - double, finite element nodes
   xg - double, Gaussian quadrature nodes on [0,1]
   wg - double, Gaussian quadrature weights on [0,1]

 Outputs:
   u_ave - double, spatial average of ua
 """
 # Initialize the average
 ua_ave = 0.0
 # number of elements
 n = len(x)-1

 for e in range(n):

   l = e
   r = e + 1
   xl = x[l]
   xr = x[r]
```

```
    # Consider quadrature point Q: (0, 1, 2 ) in element E.
    for q in range ( len(xg) ):
      #  Map XG and WG from [0,1] to
      #      XQ and QQ in [XL,XR].
      xq = xl + xg[q] * ( xr - xl )
      wq = wg[q] * ( xr - xl )

      #  Evaluate at XQ the basis functions for XL and XR.
      phil = ( xr - xq  ) / ( xr - xl )
      phir = ( xq - xl ) / ( xr - xl )
      # Evaluate at XQ the finite element approximation
      ua_loc  = ua[l]*phil + ua[r]*phir

      # Update the average
      ua_ave += wq * ua_loc

  return ua_ave
```

3. To check whether the function is flat at the right endpoint, we compare the computed finite element coefficients corresponding to the rightmost two x-values.

   Here is the code:

```
#! /usr/bin/env python
"""
Created on Fri Oct 10 13:40:45 2014

Homework 4, Question 3: We change the code to incorporate Neumann boundary
                        conditions. In particular,

                        1) we delete the enforcement of Dirichlet boundary
                           conditions on the left (the last node), and
                        2) we change the exact function and right hand side

                        To check whether the solution is flat, we compare the
                        last two finite element coefficients.
@author: hans-werner
"""
```

```
#
def fem1d_model ( ):
#
## FEM1D_MODEL solves a 1D "model" boundary value problem using finite elements.
#
#  Location:
#
#    http://people.sc.fsu.edu/~jburkardt/py_src/fem1d/fem1d_model.py
#
#  Discussion:
#
#    The PDE is defined for 0 < x < 1:
#      -u'' + u = x^3 + 3x^2 - 15x - 6
#    with boundary conditions
#      u(0) = 0,
#      u'(1) = 0.
#
#    The exact solution is:
#      exact(x) = x^3 + 3x^2 - 9x
#
#
#  Licensing:
#
#    This code is distributed under the GNU LGPL license.
#
#  Modified:
#
#    23 September 2014
#
#  Author:
#
#    John Burkardt
#
#  Local parameters:
#
#    Local, integer N, the number of elements.
#
  import numpy as np
  import scipy.linalg as la
```

```
#
#  The mesh will use N+1 points between A and B.
#  These will be indexed X[0] through X[N].
#
  a = 0.0
  b = 1.0
  n = 50
  x = np.linspace ( a, b, n + 1 )

#
#  Set a 3 point quadrature rule on the reference interval [0,1].
#
  ng = 3

  xg = np.array ( ( \
    0.112701665379258311482073460022, \
    0.5, \
    0.887298334620741688517926539978 ) )

  wg = np.array ( ( \
    5.0 / 18.0, \
    8.0 / 18.0, \
    5.0 / 18.0 ) )
#
#  Compute the system matrix A and right hand side RHS.
#
  A = np.zeros ( ( n + 1, n + 1 ) )
  rhs = np.zeros ( n + 1 )
#
#  Look at element E: (0, 1, 2, ..., N-1).
#
  for e in range ( 0, n ):

    l = e
    r = e + 1

    xl = x[l]
    xr = x[r]
#
```

```
#  Consider quadrature point Q: (0, 1, 2 ) in element E.
#
    for q in range ( 0, ng ):
#
#  Map XG and WG from [0,1] to
#      XQ and QQ in [XL,XR].
#
      xq = xl + xg[q] * ( xr - xl )
      wq = wg[q] * ( xr - xl )
#
#  Evaluate at XQ the basis functions and derivatives for XL and XR.
#
      phil = ( xr - xq  ) / ( xr - xl )
      philp = - 1.0 / ( xr - xl )

      phir = ( xq - xl ) / ( xr - xl )
      phirp = 1.0 / ( xr - xl )
#
#  Compute the following contributions:
#
#    L,L  L,R  L,Fx
#    R,L  R,R  R,Fx
#
      A[l][l] = A[l][l] + wq * ( philp * philp + phil * phil )
      A[l][r] = A[l][r] + wq * ( philp * phirp + phil * phir )
      rhs[l]  = rhs[l]  + wq *                   phil * rhs_fn ( xq )

      A[r][l] = A[r][l] + wq * ( phirp * philp + phir * phil )
      A[r][r] = A[r][r] + wq * ( phirp * phirp + phir * phir )
      rhs[r]  = rhs[r]  + wq *                   phir * rhs_fn ( xq )
#
#  Modify the linear system to enforce the left boundary condition.
#
  A[0,0] = 1.0
  A[0,1:n+1] = 0.0
  rhs[0] = 0.0
#
#  Solve the linear system.
#
  u = la.solve ( A, rhs )
```

```
  print ""
  print "Is the function flat at the right endpoint?"
  print "u(%f) = %f" % (x[-2],u[-2])
  print "u(%f) = %f" % (x[-1],u[-1])

#
#  That is the end of the main program.
#  Now we list some helper functions.
#
def exact_fn ( x ):
#
## EXACT_FN evaluates the exact solution.
#
  value = x**3 + 3*x**2 - 9*x
  return value

def rhs_fn ( x ):
#
## RHS_FN evaluates the right hand side.
#
  value = x**3 + 3*x**2-15*x-6
  return value
#
#  If this script is called directly, then run it as a program.
#
if ( __name__ == '__main__' ):
  fem1d_model ( )
```

and here is the output

```
Is the function flat at the right endpoint?
u(0.980000) = -4.997726
u(1.000000) = -5.000118
```

The function values indeed look close.