Tufts University - Department of Mathematics
Math 253 - Homework 3 Solutions

1. Verify that, for any $k \in \mathbb{Z}$, the function $u_k(x) = \sin\left((k+\frac{1}{2})\pi x\right)$ is a solution of the differential equation $-u''(x) = f(x)$ on $(0,1)$, with boundary conditions $u(0) = 0$ and $u'(1) = 0$. What right-hand side, $f(x)$, is needed for this to hold?

   Answer:
   Merely plug in the solution to the equation:

$$\left(\sin\left(\left((k+\frac{1}{2}\right)\pi x\right)\right)' = \left(k+\frac{1}{2}\right)\pi\cos\left(\left(k+\frac{1}{2}\right)\pi x\right),$$

$$\Rightarrow \left(\sin\left(\left((k+\frac{1}{2}\right)\pi x\right)\right)'' = -\left(k+\frac{1}{2}\right)^2\pi^2\sin\left(\left(k+\frac{1}{2}\right)\pi x\right),$$

$$\Rightarrow -u_k(x)'' = -\left(\sin\left(\left((k+\frac{1}{2}\right)\pi x\right)\right)'' = \left(k+\frac{1}{2}\right)^2\pi^2\sin\left(\left(k+\frac{1}{2}\right)\pi x\right) = \left(k+\frac{1}{2}\right)^2\pi^2 u_k(x).$$

   Also, $u_k(0) = \sin(0) = 0$ and $u_k'(1) = \left(k+\frac{1}{2}\right)\pi\cos\left(\left(k+\frac{1}{2}\right)\pi\right) = 0$.
   $\rightarrow$ So $u_k(x)$ is a solution with right-hand side: $f_k(x) = \left(k+\frac{1}{2}\right)^2\pi^2 u_k(x)$.

2. Implement a code to compute the finite-element solution to the differential equation $-u''(x) = f(x)$ on $(0,1)$, with boundary conditions $u(0) = 0$, $u'(1) = 0$. Your code should take, as input, an array of n points $0 = x_1 < x_2 < \ldots < x_n = 1$, and return, as output, the values of the finite-element solution at these points, $u(x_i)$, $i = 1, \ldots, n$.

   Answer:
   See the following code, which documents the procedure discussed in class.

```
function[u,A,b] = FiniteElement1DPoisson(x, f, DR)
% The following will implement a finite-element discretization and solution
% to the 1D poisson problem -u''(x) = f(x) with boundary conditions u(0)=0
% and u'(1)=0. The input is a simple array of points
% $0=x_1<x_2<x_3...<x_n=1 and right side function f(x) as a function handle
% DR Indicates whether the right node is Dirichlet (DR=1 => u(1) = 0) or if
% it is Neumann (DR=0 => u'(1) = 0).
% The output is an array of values u(x_i) for i=1,2,...,n.

% Need to build the stiffness matrix and rhs (as calculated in class)
% We have a basis function for every node (n nodes) on our graph and the
% stiffness matrix is an nxn matrix: A(i,j) = <phi_j', phi_i'>
% The right-hand side vector is the inner product of our right-hand side
% function and the basis functions: b(i) = <f(x),phi_i>
n=length(x);
A = zeros(n);
b = zeros(n,1);

% To build the matrix we will loop over each element (interval) of the
% grid.  However, we know that this inner product is nonzero only
% for a certain combination of basis functions.  Thus, we build a local
% stiffness matrix on each element, which gives the contributions for each
% basis function in that element (2 per interval).

% We know that the left point is known so the equation should be u(x_1) = 0
% This means the first row of A should just be an identity -> [1 0 0 ... 0]
% Consquently, it shouldn't contribute to any other component of the
% solution, so the column 1 should also be all 0's under the 1.
A(1,1) = 1;
% If the right-node is Dirichlet also apply this to the last row u(x_n) = 0
if(DR)
    A(n,n) = 1;
end

% Loop over all the elements (intervals)
for elm=1:n-1
    leftnode = elm; % Left endpoint of the element (x_{i-1})
    rightnode = elm+1; % Right endpoint of the element (x_i)
    hi = x(rightnode)-x(leftnode); % Element length (x_i - x_{i-1})

    % Build the local stiffness matrix from class
    % | 1/h_i    -1/h_i |
    % | -1/h_i    1/h_i  |    on element [x_{i-1} - x_i|
    ALoc = (1/hi).*ones(2);
    ALoc(1,2) = -ALoc(1,2);
    ALoc(2,1) = -ALoc(2,1);

    % Now put this in the Global matrix in the appropriate spot.
    % The matrix is ordered by nodes, thus with the exception of the first
    % element and the last element, there will be two rows and two columns
    % filled in for each element (for each node in the element).
    % If the left node is the boundary, though it shouldn't contribute to
    % the matrix since it's value is known.  The only contribution would be
    % from the right-node to rightnode entry ALoc(2,2);
    % Similarly, if we have Dirichlet boundary on the right the only
    % contribution for the last element should come from ALoc(1,1) (left
    % node).
    if(leftnode==1 && (DR==1 && rightnode==n))
        % Do nothing.  You have 1 element and two nodes and both values are
        % known...
    elseif(leftnode==1)
        A(rightnode,rightnode) = A(rightnode,rightnode) + ALoc(2,2);
    elseif (DR==1 && rightnode==n)
        A(leftnode,leftnode) = A(leftnode,leftnode) + ALoc(1,1);
    else
        A(leftnode,leftnode) = A(leftnode,leftnode) + ALoc(1,1);
        A(leftnode,rightnode) = A(leftnode,rightnode) + ALoc(1,2);
        A(rightnode,leftnode) = A(rightnode,leftnode) + ALoc(2,1);
        A(rightnode,rightnode) = A(rightnode,rightnode) + ALoc(2,2);
    end

    % Next build right-hand side vector b_i = <f,phi_i>
    % For a given element each node gets a basis function
    % from the left node and the right node (unless left node is left
    % boundary or right node is right boundary and it's Dirichlet).
    % Here we use Matlab's quad function to integrate a
    % function handle with the basis function on the given interval.
    if(leftnode==1 && (DR==1 && rightnode==n))
        % Do nothing.  You have 1 element and two nodes and both values are
        % known...
    elseif(leftnode==1)
        phi_right = @(xx) (xx-x(leftnode))/hi;
        fxright = @(xx) f(xx).*phi_right(xx);
        b(rightnode) = b(rightnode) + quad(fxright, x(leftnode),...
            x(rightnode));
    elseif(DR==1 && rightnode==n)
        phi_left = @(xx) 1 - (xx-x(leftnode))/hi;
        fxleft = @(xx) f(xx).*phi_left(xx);
        b(leftnode) = b(leftnode) + quad(fxleft, x(leftnode),...
            x(rightnode));
    else
        phi_right = @(xx) (xx-x(leftnode))/hi;
        phi_left = @(xx) 1-phi_right(xx);
        fxright = @(xx) f(xx).*phi_right(xx);
        fxleft = @(xx) f(xx).*phi_left(xx);
        b(leftnode) = b(leftnode) + quad(fxleft, x(leftnode),...
            x(rightnode));
        b(rightnode) = b(rightnode) + quad(fxright, x(leftnode),...
            x(rightnode));
    end
end

%Now we get our coefficients vector by solving the matrix equation
u = A\b;

end
```

(a) For a "smooth" f, verify that the error in your finite-element solution satisfies the expected bound, depending on h and $\|f\|$. Remember that these bounds are in terms of norms of the functions and not vector norms of your discrete solution. (How are the vector norms and function norms related?) In particular, show that the error decreases by an appropriate factor when h is repeatedly halved.

Answer:
To test the code here, we use the right-hand side from problem one, which should give smooth solution, $u_k(x) = \sin\big((k+\frac{1}{2})\pi x\big)$. In order to compute the $L^2([0,1])$ norm, we will consider two formulations. One was discussed in class, such that for a solution vector, v, defined on the nodes of the interval, $||v||_{L^2([0,1])} \approx ||v||_h = \left(h \sum\limits_{i=1}^{n} v(x_i)^2\right)$. However, since this is only an approximation, we will also simultaneously compute the "actual" L^2 norm. To do this, we know that $v(x) = \sum\limits_{i=1}^{n} v(x_i)\varphi_i(x)$. Thus, we compute $||v||_{L^2([0,1])} = ||v|| =$

$\left(\int_0^1 \left(\sum\limits_{i=1}^{n} v(x_i)\varphi_i(x)\right)^2\right)^{1/2}$. This can be rewritten over each element in a much easier way, since there are only two basis functions defined on each element:

$$||v|| = \left(\sum_{i=1}^{n-1} \int_{x_{i-1}}^{x_i} \left(v(x_{i-1})\varphi_{i-1}(x) + v(x_i)\varphi_i(x)\right)^2\, dx\right)^{1/2}.$$

The following code computes the error between our approximation and the true solution using this norm:

```
function [L2err] = ComputeL2Error(u,x,utrue);
% This routine computes the L2 Error of a linear FEM approximation to the
% 1D Poisson Equation, -u'' = f.  u is the approximate solution defined at
% the nodes x.  utrue is a function handle that evaluates the true solution
% anywhere.

% L2err = sqrt(sum_(intervals) quad_(interval) (utrue - u)^2 dx)
% Since u = sum_(nodes) u(x_i) phi_i(x), we can loop over each element to
% build the function uFEM(x) on each element and integrate using Matlab's
% quad function.
L2err = 0;

n = length(x);
nelm = n-1;

for(i=1:nelm)
    leftnode = i; % Left endpoint of the element (x_{i-1})
    rightnode = i+1; % Right endpoint of the element (x_i)
    hi = x(rightnode)-x(leftnode); % Element length (x_i - x_{i-1})
    phi_right = @(xx) (xx-x(leftnode))/hi; %Right basis function on element
    phi_left = @(xx) 1-phi_right(xx); % Left basis function on element

    % Compute approximation by linear combination of basis functions.
    uFEM = @(xx) u(leftnode).*phi_left(xx) + u(rightnode).*phi_right(xx);
    % Compute the quantity (uFem - utrue)^2
    uErrsq = @(xx) (uFEM(xx) - utrue(xx)).^2;
    % Compute quad and add to other intervals.
    L2err = L2err + quad(uErrsq,x(leftnode),x(rightnode));
end

% Final answer is square root of quads.
L2err = sqrt(L2err);
```

The following table gives the error of the numerical solution from the true solution (interpolated on the grid) using the two norms: $e_h = ||u_{FEM} - u_{TRUE}||_h$ and $E_h = ||u_{FEM} - u_{TRUE}||$, the ratio of error reduction as we refine the mesh from size H to size h, $\frac{e_H}{e_h}$ and $\frac{E_H}{E_h}$, the expected error bound as computed in class, $b_h = \frac{h^2}{2}||f||$, and the expected error reduction from the error bounds (i.e., $\frac{H^2}{h^2}$). For the following tests, we assume a uniform grid spacing, $h_i = h = 1/(n-1)$.

n	e_h	$\frac{e_H}{e_h}$	E_h	$\frac{E_H}{E_h}$	b_h	$\frac{b_H}{b_h}$
2	2.892e-09	-	1.509e-01	-	8.724e-01	-
3	2.369e-09	1.220e+00	3.928e-02	3.841e+00	2.181e-01	4.000e+00
5	3.300e-11	7.180e+01	9.921e-03	3.960e+00	5.452e-02	4.000e+00
9	4.848e-13	6.806e+01	2.487e-03	3.990e+00	1.363e-02	4.000e+00
17	1.256e-14	3.861e+01	6.220e-04	3.997e+00	3.408e-03	4.000e+00
33	7.191e-15	1.746e+00	1.555e-04	3.999e+00	8.519e-04	4.000e+00
65	2.780e-14	2.587e-01	3.888e-05	4.000e+00	2.130e-04	4.000e+00
129	3.520e-14	7.898e-01	9.721e-06	4.000e+00	5.324e-05	4.000e+00
257	1.056e-13	3.332e-01	2.430e-06	4.000e+00	1.331e-05	4.000e+00
513	1.208e-13	8.745e-01	6.076e-07	4.000e+00	3.328e-06	4.000e+00

Table 1: $u(x) = \sin\left((\frac{1}{2})\pi x\right)$

Tables 1 and 2 both show that the error gets significantly reduced as we refine the grid. Also, the error is always well below the expected error bound using either norm. As the grid space gets halved at each test in the above simulation we expect the error to go down by a factor of 4. We see this in the ratios of the expected error bounds and using the "actual" L^2 norm. However, errors using the $||\cdot||_h$ norm reduce much faster until they reach a very small number. At this point, we are close to machine precision and round-off error may occur. As it turns out, in 1D, linear finite-elements approximates the solution exactly at the grid points. Thus, when we compute the error at these nodes, they will always be very small and the $||\cdot||_h$ norm will seem better than it should. In 2D this is less pronounced.

n	e_h	$\frac{e_H}{e_h}$	E_h	$\frac{E_H}{E_h}$	b_h	$\frac{b_H}{b_h}$
2	8.134e-09	-	9.119e-01	-	3.847e+02	-
3	3.099e-09	2.625e+00	8.384e-01	1.088e+00	9.618e+01	4.000e+00
5	4.284e-09	7.233e-01	8.538e-01	9.820e-01	2.404e+01	4.000e+00
9	1.615e-08	2.652e-01	7.468e-01	1.143e+00	6.011e+00	4.000e+00
17	1.578e-09	1.024e+01	2.498e-01	2.989e+00	1.503e+00	4.000e+00
33	1.145e-09	1.378e+00	6.702e-02	3.728e+00	3.757e-01	4.000e+00
65	9.583e-10	1.195e+00	1.705e-02	3.931e+00	9.392e-02	4.000e+00
129	2.356e-12	4.068e+02	4.281e-03	3.983e+00	2.348e-02	4.000e+00
257	1.089e-13	2.163e+01	1.071e-03	3.996e+00	5.870e-03	4.000e+00
513	2.051e-13	5.309e-01	2.679e-04	3.999e+00	1.468e-03	4.000e+00

Table 2: $u(x) = \sin\left((10 + \frac{1}{2})\pi x\right)$

The following plots for the two solutions, show that we are approximating the solutions well.

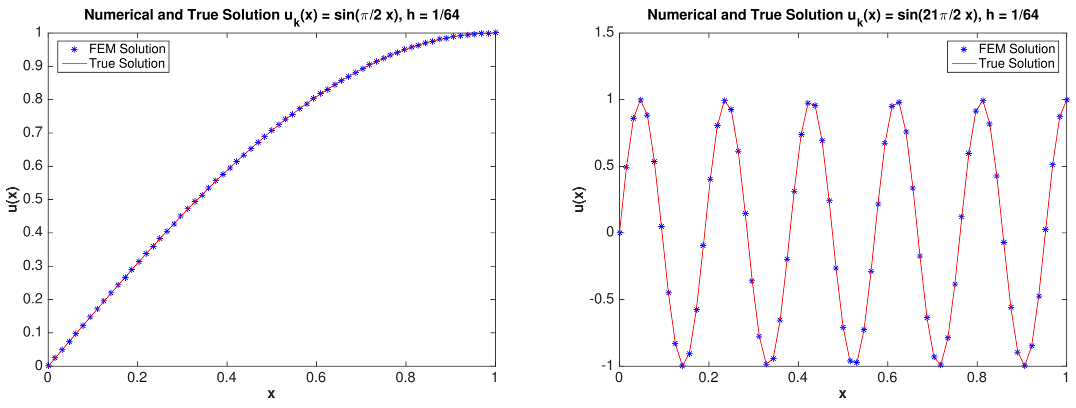

Figure 1: Left: $u_0(x)$, $n = 64$. Right: $u_{10}(x)$, $n = 64$.

(b) Modify your code to allow for Dirichlet BCs at both ends. This will make the following tests easier. Next, consider solutions, $u(x)$, of the differential equation that have different amounts of smoothness ($u \in \mathcal{C}^\ell$ for different ℓ). How does the dependence of the error in the finite-element solution change with smoothness of $u(x)$ (really what you are changing is the smoothness of f)?

Answer:
The code above describes how to modify for a Dirichlet BC on the right end point as well. The previous question illustrates what happens if we have a C^∞ function. The theorem in class, says we get the expected bounds as long as $u \in C^2$. So here we test out a function that is only in C^1:

$$u(x) = \begin{cases} \frac{1}{8}x & x \le \frac{1}{2} \\ -\frac{1}{2}x^2 + \frac{5}{8}x - \frac{1}{8} & x > \frac{1}{2}. \end{cases}$$

You can verify that this satisfies the boundary conditions and is continuous. Then,

$$u'(x) = \begin{cases} \frac{1}{8} & x \le \frac{1}{2} \\ -x + \frac{5}{8} & x > \frac{1}{2}, \end{cases}$$

which is also continuous, but

$$u''(x) = \begin{cases} 0 & x \le \frac{1}{2} \\ -1 & x > \frac{1}{2}, \end{cases}$$

is not continuous. Thus,

$$f(x) = -u''(x) = \begin{cases} 0 & x \le \frac{1}{2} \\ 1 & x > \frac{1}{2}, \end{cases}$$

Applying our code to this right-hand side we obtain the following results:

n	e_h	$\frac{e_H}{e_h}$	E_h	$\frac{E_H}{E_h}$	b_h	$\frac{b_H}{b_h}$
2	0.000e+00	-	4.707e-02	-	3.536e-01	-
3	1.823e-06	0.000e+00	1.614e-02	2.917e+00	8.839e-02	4.000e+00
5	1.579e-06	1.155e+00	4.035e-03	3.999e+00	2.210e-02	4.000e+00
9	1.512e-06	1.044e+00	1.009e-03	3.998e+00	5.524e-03	4.000e+00
17	1.494e-06	1.012e+00	2.530e-04	3.990e+00	1.381e-03	4.000e+00
33	1.490e-06	1.003e+00	6.388e-05	3.960e+00	3.453e-04	4.000e+00
65	1.489e-06	1.001e+00	1.664e-05	3.840e+00	8.632e-05	4.000e+00
129	1.489e-06	1.000e+00	4.929e-06	3.375e+00	2.158e-05	4.000e+00
257	1.489e-06	1.000e+00	2.197e-06	2.244e+00	5.395e-06	4.000e+00
513	1.489e-06	1.000e+00	1.639e-06	1.340e+00	1.349e-06	4.000e+00

Notice that our solution error isn't being reduced any further and eventually we are not below the expected error bound. There is no reason to believe it should, since the theorem does not apply to C^1 functions.