MATH 934 – 1D FINITE ELEMENT METHODS

INSTRUCTOR: DR. ADAM LARIOS

Consider the follow Poisson problem in 1D with certain boundary conditions:

$$\text{Find } u \text{ such that } \begin{cases} -u'' = f \quad \text{in } (0,1), \\ u(0) = 0, \\ u'(1) = 0. \end{cases}$$

The function f is assumed to be a square-integrate function, i.e., $\|f\|_{L^2} := \left(\int_0^1 |f(x)|^2\,dx\right)^{1/2} < \infty$. We want to solve this problem using finite element methods.

1. Brief background on function spaces

To set the stage, let us define some spaces. We use the following notation:

":=" means "defined by" or "equal to by definition".

Now for our spaces:

$$\begin{aligned} L^1_{\text{loc}} &:= \left\{ u : \int_K |u(x)|\,dx < \infty \text{ for every closed set } K \subset (0,1) \right\}, \\ L^2 = L^2(0,1) &:= \left\{ u \in L^1_{\text{loc}} : \|u\|_{L^2} < \infty \right\}, \\ H^1 = H^1(0,1) &:= \left\{ u \in L^2 : u' \in L^2 \right\}, \\ V &:= \left\{ u \in H^1 : u(0) = 0 \right\}. \end{aligned}$$

The first space is called the space of *locally integrable functions*[1]. It just tells us that doing this like integrating functions makes sense.

Before we go on, let us mention an unsettling issue arises with all of these spaces: We want to be able to say, for example, that $\|u - v\|_{L^2} = 0$ means that $u = v$. However, this is clearly not true, since we can assign different values to both u and v at one point (or many points) without changing their integrals. To get around this, we just consider two functions u and v to be *equivalent in the sense of* L^2 if $\|u - v\|_{L^2} = 0$ (we usually just say u equals v and we write $u = v$, with the understanding that this only means equivalence in the L^2 sense). This means point-wise values don't really make sense anymore: writing $u(0)$ or $u(1)$ doesn't really make sense if we can give u a different value at $x = 0$ or $x = 1$. In practice, this doesn't bother us too much because our weak forms are defined in the sense of integrals, but beware: it can cause problems when trying to assign boundary conditions! More on this later.

L^2 is one of the most widely-used spaces in analysis. It is the space of all square-integrable functions. A major reason that it is so useful mathematically is that it has a natural inner-product given by

$$(u, v) := \int_0^1 u(x)\overline{v(x)}\,dx,$$

where $\overline{v(x)}$ is the complex-conjugate of $v(x)$. This is useful in defining angles and in particular, orthogonality: If $(u, v) = 0$, we say that u and v are *orthogonal* in the L^2-sense. Note that $\|u\|_{L^2} = \sqrt{(u, u)}$. A major reason that L^2 is so useful from an scientific/engineering perspective is that if our functions are say, the possible velocities of an object, then functions in L^2 are those functions with *finite kinetic energy*. Indeed,

[1] If you have taken measure theory, we can more rigorously define $L^1_{\text{loc}} = L^1_{\text{loc}}(\Omega)$ as the set of all *measurable* functions with finite L^1 norm over every compact subset of Ω, with integrals taken in the Lebesgue sense).

sometime L^2 is call "the energy space." Since all known physical objects have finite kinetic energy, it is very useful to consider velocities which are in L^2!

H^1 is the space of L^2 functions whose derivative (taken in the *weak* sense) are also L^2 functions. It turns out that in 1D, if $u \in H^1$, then u is (equivalent to) a continuous function, and its derivative is continuous (i.e., equivalent to a continuous function). This is *not* true in higher dimensions.

The last space V makes sense because of the previous paragraph. Since $u \in H^1(0,1)$, it is continuous, so it makes sense to plug in point-wise values (we plug them into the continuous version of u), and writing $u(0) = 0$ is no longer ambiguous.

2. The weak form and the discrete form

In class, we defined the weak form of problem above; namely:

$$\text{Find } u \in V \text{ such that } (u', v') = (f, v) \quad \text{for all } v \in V.$$

One can use the Lax-Millgram Theorem (as in your previous homework) to show that this problem has a solution, and that this solution is unique.

Given a finite-dimensional subset $S \subset V$, we defined the following discrete problem:

$$\text{Find } u_S \in S \text{ such that } (u_S', v') = (f, v) \quad \text{for all } v \in S.$$

We saw in class that this problem has a unique solution for any finite dimensional subspace $S \subset V$. This solution of the discrete problem *hopefully* is an approximation of the solution to the weak problem in some sense, say $\|u - u_S\|$ gets smaller as the dimension of S increases; however, this might not be the case! Our finite-dimensional space S had to satisfy an *approximation assumption* (see the notes) for this to work.

So, can we find a space S which satisfies the approximation assumption? Our strategy was to break up the interval $[0,1]$ via points $0 = x_0 < x_1 < x_2 \cdots < x_N = 1$, and to consider continuous functions which were linear on each subinterval $I_i := [x_i, x_{i+1}]$. This space we called $\mathbb{P}_1$. Letting $S := \{u \in \mathbb{P}_1 : u(0) = 0\}$, we then showed that $S \subset V$, and that S satisfies the approximation assumption. This space also has a nice basis, the "hat" basis $\{\varphi_i\}_{i=1}^n$:

$$\varphi_i(x) := \begin{cases} 1 \text{ if } x = x_i, \\ 0 \text{ if } x = x_j, j \neq i, \\ \text{ linear on each } I_k \text{ otherwise.} \end{cases}$$

We then defined the (global) stiffness matrix $A = (a_{ij})_{i,j=1}^n$ by:

$$a_{ij} := \int_0^1 \varphi_i'(x)\varphi_j'(x)\,dx,$$

and the force-vector $\vec{F} = (F_i)_{i=1}^n$ defined by

$$F_i = \int_0^1 f(x)\varphi_i(x)\,dx.$$

If we express our unknown solution u_S by $u_S(x) = \sum_{i=1}^n U_i\varphi_i(x)$, and write $\vec{U} = (U_i)_{i=1}^n$, then we showed that solving the discrete problem with our particular choice of S was equivalent to solving the linear matrix problem

$$A\vec{U} = \vec{F}.$$

3. Tasks

Task 1: Go to `fenicsproject.org` and install FEniCS. We will use this in a future project.
Task 2: Write a MATLAB code to solve the above 1D Poison equation using the methods described above. Follow the steps below.

3.1. **The stiffness matrix.** Note that computing the stiffness matrix can be done by hand. However, in higher dimensions, it is more common to do this in a systematic way by breaking up the integral across each interval or triangle or quadrilateral, or hexagon, or tetrahedron, or whatever shapes you are working with. This saves an enormous cost, since it means you only have to compute the non-zero parts of the stiffness matrix, but it introduces the need for assembly of the matrix.

(1) Compute the local stiffness matrix by hand (we actually already did this in class, so you can just use the result).
(2) Start a new Matlab program that inputs a number of grid points N, and right-hand side function $f(x) = 6x + e^{-x}$. (Quick calculus exercise: find the exact solution for this f.) Generate a random mesh, and the corresponding h-values like this:

```
x = unique([sort(rand(1,N-1)),1]);
h = diff(x);
```

You don't have to include $x = 0$, since you already know that $u(0) = 0$ from the boundary conditions.
(3) Write a loop over the intervals which first computes the local stiffness matrix, and then adds it to the global stiffness matrix. Since your NxN matrix A is sparse, and has $3N - 2$ non-zero entries, you should use a sparse matrix for A. You need to initialize this matrix outside the loop though! You can initialize or *allocate* the memory like this:

```
A = spalloc(N,N,3*N-2);
```

After this, you would start your loop to assemble the global stiffness matrix by first computing the local stiffness matrix. The inner part of this loop might look something like this:

```
A_local = % a 2x2 matrix built out of the correct h-values
A(i:(i+1),i:(i+1)) = A(i:(i+1),i:(i+1)) + A_local;
```

Does this have the right values for the boundary conditions? Do you need to modify this matrix slightly after the loop completes, or is it OK as it is? You should always ask yourself this question after matrix assembly.
(4) Next we have to compute $F_i = \int_0^1 f(x)\varphi_i(x)\,dx$. You could do this by hand using calculus, but that sounds very annoying, and not the computationally-minded way to go, since f should be able to be a very nasty function (e.g., maybe it comes purely from data, or from the evolving solution of another PDE). Instead, we can use *numerical quadrature*. The simplest quadrature is Riemann summation, but should you use the left-hand rule, the right-hand rule, or the trapezoidal rule? You can increase the accuracy of the integration by using, e.g., Simpson's rule, but does this increase the accuracy in your solution? We can investigate this when we look at error later.

Moreover, computation of F_i should again be done by assembly. Inside the same loop you use to assemble your stiffness matrix, you can also assemble your load vector $\vec{F}$. The assembly might look something like this:

```
F_local = % a 2x1 vector with built of out local load vectors
F(i:(i+1)) = F(i:(i+1)) + F_local;
```

Mathematically,

$$\texttt{F\_local(i)} = \begin{bmatrix} \int_{x_i}^{x_{i+1}} f(x)\varphi_i(x)\,dx \\ \\ \int_{x_i}^{x_{i+1}} f(x)\varphi_{i+1}(x)\,dx \end{bmatrix},$$

but these integrals should be evaluated by quadrature.
(5) Solve the linear system using MATLAB's "backslash" operation:

```
U(2:N) = A\F;
U(1) = 0;       % add on the missing boundary condition.
plot([0,x],U,'-o');
```

Note that plotting the solution is straight-forward here, since Matlab already does a linear interpolation, and your data is already arranged left-to-right. Plotting the solution can be significantly more complicated when using, e.g., $\mathbb{P}_2$ elements, or when working in higher-dimensions.

Your solution should be exact at the grid points, as we saw in class.

(6) Compute the L^2 error in your solution as the resolution increases. To make it a fair test, don't just compute the L^2-norm based on the grid points; use linear interpolation, with say, 1000 uniformly-spaced points. The function `interp1` may be useful in doing this. Try something like this:

```
x_interp = unique(sort([x,linspace(0,1,1000)]));
u_interp = interp1([0,x],U,x_interp);
```

You can then compare this by comparing with the exact solution (evaluated at) by computing the L^2 norm of their difference. If your error seems too large, you may which to look at a plot of the exact solution minus the approximate solution. Take a careful look at the endpoints. How does the error depend on the maximum grid spacing? Does it matter which quadrature you use for computing F_i?

Have fun, and happy coding!