Lab Assignment 12
The Finite Element Method

ACS II

Spring 2020

Assigned April 2, 2020
Due April 9, 2020

Introduction

The finite element method is one of the most popular methods for numerically solving PDEs. They can be used on complicated geometries, easily handle a variety of boundary conditions, and are well understood theoretically. Although traditionally finite elements are used for PDEs, we will look at applying them to solve an ODE (and systems of ODEs). The reason for this is simplicity. It is much easier to discretize 1D problems than 2D or 3D problems. Once you understand how finite elements can be used to solve simple 1D problems, extending the method to 2D or higher problems is not much more difficult conceptually.

1 Sturm-Liouville

The first problem we will look at is the Sturm-Liouville problem:

$$-\frac{\mathrm{d}}{\mathrm{d}x}\left(p(x)\frac{\mathrm{d}u}{\mathrm{d}x}\right)+q(x)u=f(x), \qquad 0\leq x\leq 1, \tag{1}$$

where $p(x)$, $q(x)$ and $f(x)$ are given functions defined on [0,1]. We assume that $f(x)\in L^2[0,1]$ and $0<p(x)\leq P$ and $0\leq q(x)\leq Q$ for some positive constants P and Q. We also need boundary conditions. For simplicity we will assume homogeneous Dirichlet boundary conditions i.e.:

$$u(0)=0 \qquad u(1)=0.$$

Other boundary conditions, e.g. inhomogeneous Dirichlet, Neumann or Robin are all handled rather naturally by finite elements. The solution $u(x)$ to (1) is called the *classical solution*. If $f,\ p$, or q are not sufficiently smooth such a solution may not exist; however, we may still have what is known as a *weak solution*.

1.1 Weak formulation

To obtain a weak formulation we multiply (1) by a test function $v(x)$ and integrate by parts:

$$\int_0^1 \left(-\frac{\mathrm{d}}{\mathrm{d}x}\left(p(x)\frac{\mathrm{d}u}{\mathrm{d}x}\right) + q(x)u\right) v(x)\ \mathrm{d}x = \int_0^1 f(x)v(x)\ \mathrm{d}x,$$
$$\Rightarrow -p(x)u'(x)v(x)\Big|_0^1 + \int_0^1 p(x)u'(x)v'(x)\ \mathrm{d}x + \int_0^1 q(x)u(x)v(x)\ \mathrm{d}x = \int_0^1 f(x)v(x)\ \mathrm{d}x.$$

A couple things to note:

- In (1) we needed to know the derivative of $p(x)$, after the integration by parts we don't need to know $p'(x)$.
- $u(x)$ went from needing to have two derivatives in (1) to one after the integration by parts. In fact it doesn't even require a derivative everywhere, we only need that the integral of u and u' are both not infinity. In other words u must be *weakly differentiable*. Note that this means that $u(x)$ is allowed to have kinks in it. Mathematically we say that $u \in H^1[0,1]$.
- Since $u(0) = u(1) = 0$, it is 0 on the boundary, u is actually in the subspace $H_0^1[0,1]$.
- $v(x)$ is also required to be weakly differentiable and we will take v to be in the same space as u, i.e $v \in H_0^1[0,1]$.

Let's look at the boundary term $-p(x)u'(x)v(x)\big|_0^1 = p(0)u'(0)v(0) - p(1)u'(1)v(1)$. Since $v \in H_0^1[0,1]$, v is 0 on the boundary, so this term is zero. The *weak formulation* is defined as follows:

$$\begin{aligned}&\text{Seek } u \in H_0^1[0,1] \text{ such that for all } v \in H_0^1[0,1]:\\ &\int_0^1 p(x)u'(x)v'(x)\ \mathrm{d}x + \int_0^1 q(x)u(x)v(x)\ \mathrm{d}x = \int_0^1 f(x)v(x)\ \mathrm{d}x.\end{aligned} \tag{2}$$

We call $u(x)$ a *weak solution*. If $p(x)$, $q(x)$ and $f(x)$ are sufficiently smooth, then the weak solution coincides with the classical solution. We now turn to computing the weak solution.

1.2 Discrete weak formulation

We transformed (1) into (2), now we must solve it. The first step to do this is to replace $u(x)$ in (2) with a Galerkin approximation $u^h(x)$ in a finite dimensional subspace $V^h \subset H_0^1[0,1]$. We will require u^h to satisfy (2), but only for all $v^h \in V^h$. This is called the *discrete weak formulation*:

$$\begin{aligned} &\text{Seek } u^h \in V^h \text{ such that for all } v^h \in V^h: \\ &\int_0^1 p(x)(u^h)'(x)(v^h)'(x)\ \mathrm{d}x + \int_0^1 q(x)u^h(x)v^h(x)\ \mathrm{d}x = \int_0^1 f(x)v^h(x)\ \mathrm{d}x. \end{aligned} \tag{3}$$

We'd like to convert the discrete weak problem to a linear system of equations that we can solve. To do this we must specify exactly what V^h is and choose a specific basis for it. We will take V^h to be the space of all continuous linear piecewise polynomials defined on a partition of [0,1] that satisfy the homogeneous Dirichlet boundary conditions. In particular we will consider partitioning [0,1] into equally spaced intervals with spacing h. See Figure 1 for an example of such a function.

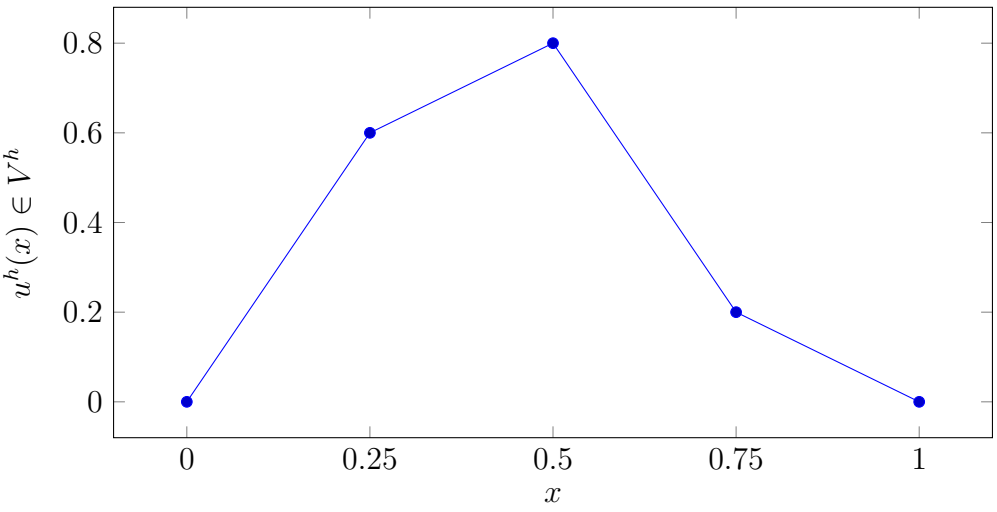

Figure 1: Example of a continuous piecewise linear polynomial in V^h on 4 equally spaced intervals. Our solution $u^h(x)$ will take such a form.

The next step is to choose a basis for V^h. There are many options, however we will

choose the so called "hat" functions. For $1 \leq i \leq N$, these are defined as:

$$\phi_i(x) = \begin{cases} \dfrac{x - x_{i-1}}{h} & \text{for } x_{i-1} \leq x \leq x_i, \\ \dfrac{x_{i+1} - x}{h} & \text{for } x_i \leq x \leq x_{i+1}, \\ 0 & \text{elsewhere.} \end{cases} \tag{4}$$

These basis functions are plotted for $h = 1/4$ in Figure 2.

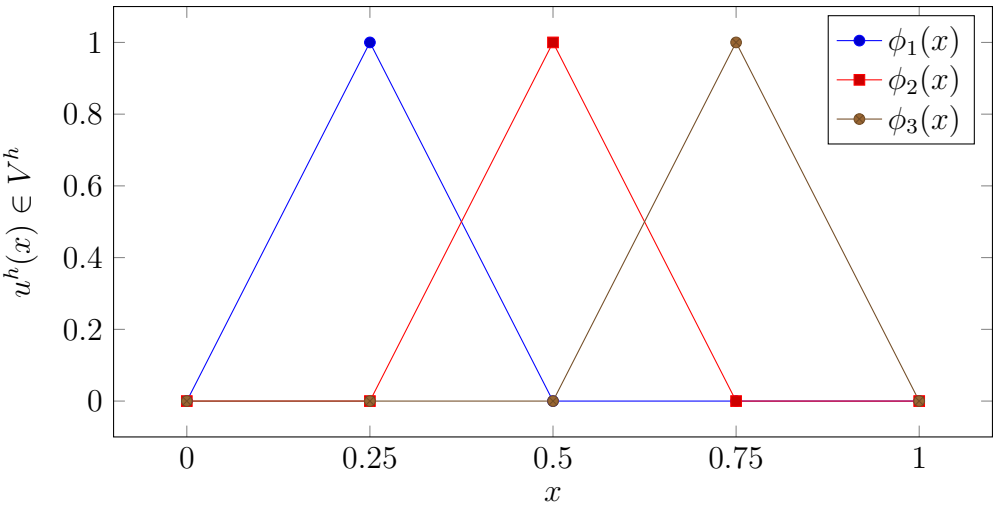

Figure 2: "Hat" basis functions defined over 4 intervals on [0,1].

Since $\{\phi_i\}_{i=1}^N$ form a basis for V^h, any function in V^h can be written as a linear combination of these basis functions. In particular we have,

$$u^h(x) = \sum_{j=1}^{N} c_j \phi_j(x), \tag{5}$$

where $\{c_j\}_{j=1}^N$ are unknown coefficients. Our task is now to find these coefficients.

Since (3) holds for any $v^h \in V^h$ it must hold for all the basis functions. Plugging (5) into (3) and enforcing it only for the basis functions gives the system:

$$\int_0^1 p(x) \sum_{j=1}^{N} c_j \phi_j'(x) \phi_i'(x) \, \mathrm{d}x + \int_0^1 q(x) \sum_{j=1}^{N} c_j \phi_j(x) \phi_i(x) \, \mathrm{d}x$$
$$= \int_0^1 f(x) \phi_i(x) \, \mathrm{d}x, \qquad \text{for } i = 1, \cdots, N.$$

This can rewritten as the linear system,

$$(\mathcal{M} + \mathcal{S})\mathbf{c} = \mathbf{f}, \tag{6}$$

where

$$\begin{aligned} \mathcal{M}_{ij} &= \int_0^1 q(x)\phi_j(x)\phi_i(x)\ \mathrm{d}x, \qquad \mathcal{S}_{ij} = \int_0^1 p(x)\phi_j'(x)\phi_i'(x)\ \mathrm{d}x, \\ \mathbf{c} &= [c_1, \cdots, c_N]^T, \qquad \mathbf{f}_i = \int_0^1 f(x)\phi_i(x)\ \mathrm{d}x. \end{aligned} \tag{7}$$

The matrix $\mathcal{M}$ is often called the *mass matrix*, while $\mathcal{S}$ is often called the *stiffness matrix*.

So that's it. Solve (6), then you can use $\mathbf{c}$ to evaluate $u^h(x)$ at any point in [0,1]. In fact c_i is the value of u^h at node x_i (try to convince yourself of this). In contrast to finite difference methods that only construct discrete approximations to the solution, finite element methods create a function that can be evaluated everywhere. What's more, we can evaluate $u'(x)$ at any point inside the domain (except at the nodes). The challenge now turns to the assembly of the matrices and vectors.

1.3 Assembly

Before we talk about the assembly algorithm we should make a few remarks about the matrices $\mathcal{M}$ and $\mathcal{S}$. The first thing to notice is that they are symmetric, i.e $\mathcal{M}_{ij} = \mathcal{M}_{ji}$ and $\mathcal{S}_{ij} = \mathcal{S}_{ji}$.

We can say more about their structure. Note how in Figure 2 $\phi_1(x)$ and $\phi_3(x)$ do have any nonzero overlap. This is not a coincidence. The "hat" basis functions are a nice choice of basis because they have what is known as *compact support*. In other words they are each nonzero over only a portion of the domain. In particular if we call each interval on [0,1] an element, then each basis function is nonzero over at most two elements.

What does this mean for the matrix structure? Since ϕ_1 and ϕ_3 have no nonzero overlap, we can say the following:

$$\int_0^1 \phi_1(x)\phi_3(x)\ \mathrm{d}x = \int_0^1 \phi_1'(x)\phi_3'(x)\ \mathrm{d}x = 0 \Rightarrow \mathcal{M}_{13} = \mathcal{M}_{31} = \mathcal{S}_{13} = \mathcal{S}_{31} = 0.$$

Again, this is not a coincidence. Since each basis function is nonzero over at most two elements, it can only have nonzero overlap with itself and two other basis functions. Thus $\phi_i(x)\phi_j(x)$ is nonzero only if $|i - j| \leq 1$. We can use this to simplify the mass and stiffness

matrices:

$$\mathcal{M}_{ij} = \begin{cases} \int_0^1 q(x)\phi_i(x)\phi_j(x)\ \mathrm{d}x & \text{if } |i-j| \leq 1, \\ 0 & \text{otherwise,} \end{cases} \qquad \mathcal{S}_{ij} = \begin{cases} \int_0^1 p(x)\phi_i'(x)\phi_j'(x)\ \mathrm{d}x & \text{if } |i-j| \leq 1, \\ 0 & \text{otherwise.} \end{cases}$$

In other words $\mathcal{M}$ and $\mathcal{S}$ are both tridiagonal and so (6) can be solved using the Thomas algorithm for example. In general, depending on our choice of V^h and basis, this will not be the case. It is typical of finite element methods however to have very sparse linear systems to solve since basis functions are almost always chosen to have compact support.

We will exploit the compact support of the basis functions to efficiently assemble $\mathcal{M}$, $\mathcal{S}$ and $\mathbf{f}$. The algorithm described below may seen needless complicated at first, however it scales very well to higher dimensions or different basis functions. This is how almost all finite element software assembles the linear system.

The first thing to realize is that the integrals in (7) can be split up over the elements. For example,

$$\int_0^1 q(x)\phi_i(x)\phi_j(x)\ \mathrm{d}x = \sum_{k=0}^{N} \left(\int_{x_k}^{x^{k+1}} q(x)\phi_i(x)\phi_j(x)\ \mathrm{d}x \right). \tag{8}$$

The interval $[x_k, x_{k+1}]$ is an element, we'll denote it as I_k. Over I_k there are only two nonzero basis functions, ϕ_k and ϕ_{k+1} (on boundary elements there's only a single nonzero basis function). For our assembly we will take advantage of this fact by looping over the elements and adding contributions to the matrices and right hand side from basis functions that are nonzero over that element.

Next we need to compute the integrals. If $q(x)$ is simple enough there may be an analytic way to do this. For general $q(x)$ a closed form integral may not exist. We can always use numerical quadrature however. Given L quadrature points $\{x_\ell\}_{\ell=1}^L$ on the interval $[x_k, x_{k+1}]$ and corresponding weights $\{w_\ell\}_{\ell=1}^L$, we can approximate the integrals in (8) by,

$$\int_{x_k}^{x^{k+1}} q(x)\phi_i(x)\phi_j(x)\ \mathrm{d}x \approx \sum_{\ell=1}^{L} q(x_\ell)\phi_i(x_\ell)\phi_j(x_\ell)w_\ell.$$

Using these observations the Algorithm 1 is proposed to assemble $\mathcal{M}$, $\mathcal{S}$ and $\mathbf{f}$. Note the flexibility of this algorithm. We have not specified the shape of the basis functions or the shape of the elements. This same algorithm could be applied to 2D or 3D problems, or problems with more complicated basis functions.

Data: number of intervals N, functions $p(x)$, $q(x)$ and $f(x)$
Result: $\mathcal{M}$, $\mathcal{S}$, $\mathbf{f}$
initialize :
$\mathcal{M} = \text{zeros}(N-1, N-1)$;
$\mathcal{S} = \text{zeros}(N-1, N-1)$;
$\mathbf{f} = \text{zeros}(N-1, 1)$;
for *each element* I_k **do**
  get nonzero basis functions on I_k;
  get $\{x_\ell\}_{\ell=1}^L$ and $\{w_\ell\}_{\ell=1}^L$ on I_k;
  for $\ell = 1, \ldots, L$ **do**
    for *each nonzero basis function on* I_k **do**
      $i \leftarrow$ index of basis function;
      $f_i = f_i + f(x_\ell)\phi_i(x_\ell)w_\ell$;
      for *each nonzero basis function on* I_k **do**
        $j \leftarrow$ index of basis function;
        $\mathcal{M}_{ij} = \mathcal{M}_{ij} + q(x_\ell)\phi_i(x_\ell)\phi_j(x_\ell)w_\ell$;
        $\mathcal{S}_{ij} = \mathcal{S}_{ij} + p(x_\ell)\phi_i'(x_\ell)\phi_j'(x_\ell)w_\ell$;
      end
    end
  end
end

Algorithm 1: Assembly algorithm for the matrices and right hand side.

What quadrature rule should we use? For piecewise linear basis functions it turns out that the midpoint rule is good enough. In that case each interval has only a single quadrature point. On interval I_k the point x_q and w_q are given by

$$x_q = \frac{x_k + x_{k-1}}{2}, \qquad w_q = x_k - x_{k-1}.$$

2 Deliverable

Your task is to write a finite element solver for (1). For this lab you are free to use any language of your choice, including Python or Matlab. To keep the program you write flexible, we will write the code in parts and verify each part as you go along.

2.1 Geometry routine

The first routine you will need is a routine to discretize your domain and set up various look-up tables that we will use when assembling the matrices. In particular we will need to know which basis functions are nonzero over each element as well as which nodes map to which unknown.

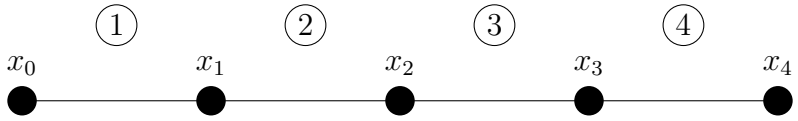

Figure 3: Partitioning the domain into four equal elements.

Consider the discretization shown in Figure 3. Here we have four elements. Remember that on each element there are at most two basis functions that are nonzero. Which basis functions are these? Take element 2 for example.

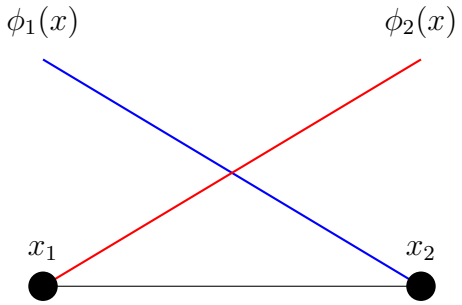

Figure 4: Local nodes and basis functions on element 2.

On element 2, the only nodes that matter are nodes 1 and 2. The local basis functions ϕ_1 and ϕ_2 are both nonzero on element 2. Note that this simple numbering scheme is only because we are using linear basis functions with homogeneous boundary conditions (in addition we are indexing from $i = 0, \ldots, N$, if you are using Matlab and indexing from $i = 1, \ldots, N + 1$ your unknown numbers and node numbers will not line up). In general basis function i may or may not be centered at node i. If we use higher order elements then each element may have more than two local nodes. For example quadratic elements have an additional node in the center of the element.

In any event using linear basis functions, for the geometry in Figure 3 we can construct a table that gives the local nodes for each element:

element	local node 1	local node 2
1	0	1
2	1	2
3	2	3
4	3	4

For each node, we must also associate an unknown. Again, for this particular case this is trivial. Unknown i is located at node i (assuming you're indexing from $i = 0, \ldots, N$).

Write a function `geometry` that takes as an input the number of elements and returns:

- the number of unknowns
- the x coordinate of each node
- an array which associates each local node of an element with its global node; this array should be dimensioned by the number of elements and the number of local nodes per element (2 for linear elements)
- an array that gives the unknown number at each node. If there is no unknown there set the unknown number to zero.
- an array that gives that gives the quadrature points in each element; this array should be dimensioned by the number of elements and the number of quadrature points per element
- an array that gives the weights for each quadrature point in each element; this array should be dimensioned by the number of elements and the number of quadrature points per element

Test your code on the geometry in Figure 3, i.e. with $N = 4$.

2.2 Basis functions

You will also need code to evaluate the basis functions. Write a function `pw_linear_basis_function` that takes as inputs:

- the point x to evaluate the basis function
- the local node number where the basis function is centered (i.e where it is one)
- an array of the x coordinates of the local nodes

and returns:

- the value of the basis function at the given point
- the value of its derivative at the same point

Test your code by evaluating $\phi_2(x)$ in the setup depicted in Figure 2 at the points $x = 0.125,\ 0.375,\ 0.5,\ 0.675,\ 0.8$.

2.3 Matrix assembly

Using these two routines, write a function `system_assembly` that assembles $\mathcal{M}$, $\mathcal{S}$ and $\mathbf{f}$ according to Algorithm 1. This function should take as inputs

- the number of elements N
- functions handles for $p(x)$, $q(x)$ and $f(x)$

and returns $\mathcal{M}$, $\mathcal{S}$, $\mathbf{f}$ and the nodes in your discretization.

Test your code with $p(x) = 1$, $q(x) = 0$ and $N = 16$. In this case your stiffness matrix match the matrix you would get using finite difference.

2.4 Test cases

After you assemble the matrices and right hand side, all that is left is to solve the linear system

$$(\mathcal{M} + \mathcal{S})\mathbf{c} = \mathbf{f}.$$

1. Use your code to solve the boundary value problem (1) with $p(x) = 1$, $q(x) = 0$ and $f(x) = \pi^2 \sin(\pi x)$. The exact solution is $u(x) = \sin(\pi x)$. Plot the exact solution and the coefficients $\mathbf{c}$ at the appropriate nodes for $N = 8,\ 16,\ 32$.

2. Use your code to solve the boundary value problem (1) with $p(x) = 1$, $q(x) = \sin(\pi x)$ and $f(x) = -2\pi \cos(\pi x) + \pi^2 x \sin(\pi x) + x \sin^2(\pi x)$. The exact solution is $u(x) = x \sin(\pi x)$. Plot the exact solution and the coefficients $\mathbf{c}$ at the appropriate nodes for $N = 8,\ 16,\ 32$.

Submission and Grading

To get credit for this assignment, you must submit the following information to Canvas by 11:59pm, April 9, 2020

- your source code files
- Your report as a single file in pdf format, including results from your work and relevant discussion of your observations, results, and conclusions.

This information must be received by 11:59pm, April 9, 2020. Upload the required documents to Canvas. As stated in the course syllabus, late assignment submissions will be subject to a 10% point penalty per 24 hours past the due date at time of submission, to a maximum reduction of 50%, according to the formula:

*[final score] = [raw score] - min(0.5, 0.1 * [# of days past due]) * [maximum score]*