ΜΑΣ 473 - Μεθόδοι Πεπερασμένων Στοιχείων Τελική Εργασία

Παναγιώτα Δαμιανού
University of Cyprus
Mathematics and Statistics Department

December 18, 2021

Contents

List of Figures

Abstract

Consider the following boundary value problem (BVP):

$$\begin{gathered} -(d(x)u'(x))' + c(x)u(x) = f(x),\ x \in I = (a,b) \\ u(a) = u(b) = 0 \end{gathered} \tag{BVP}$$

where $a, b \in \mathbb{R}$ and $c(x), d(x)$ given functions that satisfy the following smoothness assumptions: $d \in C^1(I),\ c, f \in C(I),\ d(x) \geq a > 0, \forall x \in (a,b)$. Using the hp-FEM version of the finite element method we will approximate the solution $u(x)$ of (BVP), as well as make some observations regarding the class and rate of convergence for the error term regarding this approximation.

1 The hp-FEM version

In the hp-FEM version of the finite element method for solving differential equations, our interval/domain $I = (a,b)$ is split into M-sub-intervals that define our mesh grid; $\Delta = \{a = x_1 < x_2 < \cdots < x_M < x_{M+1} = b\}$, and in each sub-interval we use polynomials of degree p_i to approximate the solution to our problem; $u(x)$. We set $\vec{p} = (p_1, \ldots, p_M)^T \in \mathbb{R}^M$ the vector with the degrees of our "sub-polynomials" in each sub-interval.

Of course, we will further analyze through more rigorous analysis at Section 3 the different variants of the hp-FEM version; the h variant and p variant of the finite element method. In the h variant of the method we hold constant the degree of the sub-polynomials we use while changing the number of points in our mesh. On the contrary, in the p variant we keep constant the number of points in our mesh grid while fluctuating the degree of the polynomials used in each sub-interval. As mentioned above, these two variants are a special case of the more general hp-FEM version. All these will become especially apparent, by making different analyses on these methods which will be implemented through code I created in the Python programming language. This code will be provided at the end in a specially dedicated section.

2 Model Problem

We consider the BVP (BVP). We multiply the differential equation of (BVP) with a control function $v(x)$ (called an **extraction function**) and integrate over I. This gives us the following:

$$-\int_I (d(x)u'(x))'v(x)dx + \int_I c(x)u(x)v(x)dx = \int_I f(x)v(x)dx \tag{1}$$

Now we proceed with integration by parts by integrating through the differential the term $(d(x)u'(x))'$, which means:

$$-\int_I (d(x)u'(x))'v(x)dx = [d(x)u'(x)v(x)]_b^a + \int_I d(x)u'(x)v'(x)dx$$

Now consider the linear subspace of $C^1(I)$; $H_0'(I)$, where
$H_0^1(I) = \{g \in C^1(I) : g(a) = g(b) = 0\}$, if our control function $v \in H_0'(I)$ then the first term in the above expression is equal to 0. Thus:

$$-\int_I (d(x)u'(x))'v(x)dx = \int_I d(x)u'(x)v'(x)dx$$

Now substituting back at (1), we have:

$$\int_I (d(x)u'(x)v'(x) + c(x)u(x)v(x))dx = \int_I f(x)v(x)dx \tag{2}$$

This means that through the use of this control function $v \in H_0^1(I)$, we turn our BVP (BVP) to the following weak form:

$$\text{Find a function } u \in H_0^1(I) \text{ such that: } B(u,v) = F(v),\ \forall v \in H_0^1(I) \tag{W}$$

where $B(u,v) = \int_I (d(x)u'(x)v'(x)+c(x)u(x)v(x))dx$ a bilinear form and $F(v) = \int_I f(x)v(x)dx$ a linear functional. This is called the **weak formulation of the differential equation $\mathcal{L}u = f$** - (BVP).

In this case, we will solve the "discrete-variant" of (W) to find the solution of finite elements to our initial problem; u_{FE}. The discrete-variant of (W) is:

Discrete variant of the weak form problem (W)

$$\begin{gathered}\text{Find a function } u_{FE} \in \mathcal{S} \lneq H_0^1(I) \text{ such that:}\\ B(u_{FE}, v) = F(v),\ \forall v \in \mathcal{S}\end{gathered} \tag{W$_M$}$$

where we will define $\mathcal{S}$ in a bit and B, F are defined as in (W).

Before we continue, we first establish some definitions and notations used in this paper.

Let Δ be a mesh of the interval $\bar{I} = [a,b]$ that consists of the following nodal points: $\Delta : a = x_1 < x_2, \cdots < x_M < x_{M+1} = b$ where we use the notation Ω_k for the kth closed sub-interval of this partition; $\Omega_k = [x_k, x_{k+1}]$ for $k = 1, 2, \ldots, M$.

We also define the reference element Ω_{ST} which is the closed interval [-1,1], that is: $\Omega_{ST} = [-1,1]$. We know that for all $k = 1, 2, \dots, M$, Ω_{ST} and Ω_k are connected through the following homeomorphism between these two intervals:

$$Q_k(\xi) = x = \frac{1-\xi}{2} x_k + \frac{1+\xi}{2} x_{k+1},\ \xi \in \Omega_{ST}$$

This is the homeomorphism that takes us from Ω_{ST} to Ω_k, for all $k = 1, \dots, M$. The inverse function that takes us from Ω_k to Ω_{ST} is:

$$\xi = Q_k^{-1}(x) = \frac{2x - x_k - x_{k+1}}{x_{k+1} - x_k},\ x \in \Omega_k$$

We now define the space $E(I) := \{u \in C^1(I) : B(u,u) < +\infty\}$ - called the **energy space**, which for our current problem is exactly the sub-space of $C^1(I)$ defined above; $H_0^1(I)$. The space of all polynomials defined in Ω_{ST} with degree $\leq p$ is denoted by $\Pi_p(\Omega_{ST})$.

We define $\mathcal{S}^{\vec{p}}(\bar{I}, \Delta)$, which is called the **Finite Element space**, and is defined as follows (it is a subspace of the energy space defined above):

$$\mathcal{S}^{\vec{p}}(\bar{I}, \Delta) := \{u \in E(I) : u(Q_k(\xi)) \in \Pi_{p_k}(\Omega_{ST}), \forall k = 1, 2, \dots, M\}$$

This means that this set is a linear space that consists of functions that have a finite bilinear form $B(u,u)$, that under the homeomorphism $Q_k(\xi)$ with respect to the mesh grid of I; Δ, for each $k = 1, 2, \dots M$, they are polynomials of degree $\leq p_k$ (where these degrees are defined by the vector $\vec{p}$ we mentioned above) defined in Ω_{ST}.

Now, N_i, for $i = 1, \dots, p+1$ where $p \in \mathbb{N}_0$ denote the **hierarchical basis functions** that are defined in $\mathbb{R}$ as polynomials, and are given by the following:

$$N_i(x) = \begin{cases} \frac{1}{2}(1-x), & i = 1 \\ \frac{1}{2}(1+x), & i = 2 \\ \phi_{i-1}(x), & i \geq 3 \end{cases}$$

where $\phi_j(x) = \sqrt{\frac{2j-1}{2}} \int_{-1}^{x} L_{j-1}(t)dt$ and $L_j(t)$ is the jth degree Legendre Polynomial; $j \in \mathbb{N}$ and $L_{j-1}(1) = 1,\ \forall j \in \mathbb{N}_0$. The two first functions, N_1, N_2 are called **modal or extremal hierarchical basis functions** while for $\forall i \geq 3$, $N_i(x)$ are called **internal hierarchical basis functions**.

It is not difficult to prove that $\{N_1, \dots, N_{p+1}\}$ constitute a linear independent set for all $p \in \mathbb{N}_0$. This means that as polynomials of degree $\leq p$, as a linear independent set and because $dim(\Pi_p(\Omega_{ST})) = p+1$, then $\Pi_{p_k}(\Omega_{ST}) =$

$Span(\{N_1, \dots, N_{p_k+1}\})$, $\forall k = 1, 2, \dots, M$ and as such, from definition we can prove using simple linear algebra that:

$$dim(\mathcal{S}^{\vec{p}}(\bar{I}, \Delta)) = \left(\sum_{k=1}^{M} (p_k + 1) \right) - M + 1 \tag{3}$$

Now since in our problem (BVP) and (W$_M$) we have Dirichlet boundary conditions, this means that:

$$dim(\mathcal{S}^{\vec{p}}(\bar{I}, \Delta)) = \left(\sum_{k=1}^{M} p_k \right) - 1 \tag{4}$$

This is because, that while it might seem that we need $\sum_{k=1}^{M}(p_k + 1)$ constants to define an element of $\mathcal{S}^{\vec{p}}(\bar{I}, \Delta)$, due to $dim(\Pi_p(\Omega_{ST})) = p_k + 1$, $\forall k = 1, 2, \dots, M$, let us not forget that we require continuity in the inner nodal points of our mesh grid for functions of the space $\mathcal{S}^{\vec{p}}(\bar{I}, \Delta)$, so essentially we lose M-1 degrees of freedom. This gives us (3). But we also lose 2 degrees of freedom due to the constraints induced by the Dirichlet boundary conditions. As such, if we subtract 2 from (3) and carry out the operations, we are left with (4).

So now revisiting (W$_M$) with the notation and definitions we established, we have to solve the following discrete weak form problem:

Revisiting (W$_M$)

$$\begin{aligned} &\text{Find a function } u_{FE} \in \mathcal{S}^{\vec{p}}(\bar{I}, \Delta) \text{ such that:} \\ &B(u_{FE}, v) = F(v), \ \forall v \in \mathcal{S}^{\vec{p}}(\bar{I}, \Delta) \end{aligned} \tag{W$^*_M$}$$

We now use the properties of the Riemann-integral to write our bilinear form B and linear functional F as sums in the following manner:

$$B(u_{FE}, v) = \sum_{k=1}^{M} B^{[k]}(u_{FE}, v)$$

$$F(v) = \sum_{k=1}^{M} F^{[k]}(v)$$

where $\forall k = 1, 2, \dots, M$:

$$\begin{aligned} B^{[k]}(u_{FE}, v) &= \int_{x_k}^{x_{k+1}} (d(x)u'_{FE}(x)v'(x) + c(x)u_{FE}(x)v(x))dx \\ F^{[k]}(v) &= \int_{x_k}^{x_{k+1}} f(x)v(x)dx \end{aligned} \tag{5}$$

The matrix that corresponds to the bilinear forms $B^{[k]}(u_{FE}, v)$ is called the **stiffness matrix** and the vector that corresponds to the linear functionals $F^{[k]}(v)$ is called the **load vector**. We define these in a bit.

Let now $h_k = x_{k+1} - x_k$ be the length of the k-th sub-interval of our mesh grid. Like mentioned above, we will make the transition from the sub-interval $\Omega_k = [x_k, x_{k+1}]$ to the interval $\Omega_{ST} = [-1, 1]$ using the homeomorphism defined above. That is $\forall k = 1, 2, \ldots, M$:

$$\begin{aligned} Q_k(\xi) = x &= \frac{1-\xi}{2}x_k + \frac{1+\xi}{2}x_{k+1}, \ \xi \in \Omega_{ST} \\ &\Rightarrow \boxed{\frac{dx}{d\xi} = Q'_k(\xi) = \frac{x_{k+1} - x_k}{2} = \frac{h_k}{2}}, \ \xi \in \Omega_{ST} \end{aligned} \tag{6}$$

and:

$$\boxed{\frac{d\xi}{dx} = (Q_k^{-1})'(x) = \frac{1}{Q'_k(Q_k^{-1}(x))} = \frac{2}{h_k}}, \ x \in \Omega_k \tag{7}$$

With these in mind, we see that (5) becomes:

$$\begin{aligned} B^{[k]}(u_{FE}, v) \overset{\because \text{Chain Rule}}{\underset{\because \text{Change of Vars.}}{=\!=\!=\!=}} \int_{-1}^{1} \bigg(& d(Q_k(\xi))\frac{d}{d\xi}\frac{d\xi}{dx}(u_{FE}(Q_k(\xi))\frac{d}{d\xi}\frac{d\xi}{dx}(v(Q_k(\xi)) \\ & + c(Q_k(\xi))u_{FE}(Q_k(\xi))v(Q_k(\xi))\bigg)Q'_k(\xi)d\xi \end{aligned}$$

So due to (6) and (7) we have:

$$\boxed{B^{[k]}(\widetilde{u_{FE}}, \widetilde{v}) = \frac{2}{h_k}\int_{-1}^{1} \widetilde{d}(\xi)\widetilde{u_{FE}}'(\xi)\widetilde{v}'(\xi)d\xi + \frac{h_k}{2}\int_{-1}^{1} \widetilde{c}(\xi)\widetilde{u_{FE}}(\xi)\widetilde{v}(\xi)d\xi}$$

where we define above: $\widetilde{d}(\xi) := d(Q_k(\xi))$, $\widetilde{u_{FE}}(\xi) := u_{FE}(Q_k(\xi))$, $\widetilde{v}(\xi) := v(Q_k(\xi))$ and $\widetilde{c}(\xi) := c(Q_k(\xi))$.

And for the k-th component of the linear functional:

$$F^{[k]}(v) \overset{\because\text{Chain Rule}}{\underset{\because\text{Change of Vars.}}{=\!=\!=\!=\!=}} \int_{-1}^{1} f(Q_k(\xi))v(Q_k(\xi))Q_k'(\xi)d\xi$$

Now using (6) we gain similarly:

$$\boxed{F^{[k]}(\widetilde{v}) = \frac{h_k}{2}\int_{-1}^{1} \widetilde{f}(\xi)\widetilde{v}(\xi)d\xi}$$

where $\widetilde{v}$ defined as above and $\widetilde{f}(\xi) := f(Q_k(\xi))$.

With this change of variables we are now "residing" within $\Omega_{ST} = [-1,1]$, which means that due to (W_M^*) and the definition of $\mathcal{S}^{\vec{p}}(\bar{I},\Delta)$, using the fact we observed above that $\Pi_{p_k}(\Omega_{ST}) = Span(\{N_1,\ldots,N_{p_k+1}\}),\ \forall k = 1,2,\ldots,M$, then we can write $\widetilde{u_{FE}}(\xi),\ \widetilde{v}(\xi)$ as the following linear combinations for each $k = 1,2,\ldots,M$:

$$\widetilde{u_{FE}}(\xi) = \sum_{i=1}^{p_k+1} \alpha_i^{[k]} N_i(\xi) \text{ and } \widetilde{v}(\xi) = \sum_{j=1}^{p_k+1} \beta_j^{[k]} N_j(\xi) \tag{8}$$

Substituting now these back into $B^{[k]}(\widetilde{u_{FE}},\widetilde{v})$, we have the following result:

$$\begin{aligned} B^{[k]}(\widetilde{u_{FE}},\widetilde{v}) &\overset{\because(8)}{=\!=} \frac{2}{h_k}\int_{-1}^{1} \widetilde{d}(\xi)\sum_{i=1}^{p_k+1}\alpha_i^{[k]}N_i'(\xi)\sum_{j=1}^{p_k+1}\beta_j^{[k]}N_j'(\xi)d\xi \\ &+ \frac{h_k}{2}\int_{-1}^{1}\widetilde{c}(\xi)\sum_{i=1}^{p_k+1}\alpha_i^{[k]}N_i(\xi)\sum_{j=1}^{p_k+1}\beta_j^{[k]}N_j(\xi)d\xi \\ \overset{\because\text{ Finite}}{\underset{\text{Sums}}{\Longleftrightarrow}} B^{[k]}(\widetilde{u_{FE}},\widetilde{v}) &= \sum_{i=1}^{p_k+1}\sum_{j=1}^{p_k+1}\alpha_i^{[k]}\beta_j^{[k]}\left\{\frac{2}{h_k}\int_{-1}^{1}\widetilde{d}(\xi)N_i'(\xi)N_j'(\xi)d\xi\right. \\ &\left.+\frac{h_k}{2}\int_{-1}^{1}\widetilde{c}(\xi)N_i(\xi)N_j(\xi)d\xi\right\} \end{aligned}$$

We now define the following, $\forall k = 1,2,\ldots,M$ and $\forall i,j = 1,2,\ldots,p_k+1$:

$$[K_k]_{ij} = \frac{2}{h_k}\int_{-1}^{1}\widetilde{d}(\xi)N_i'(\xi)N_j'(\xi)d\xi \text{ and } [G_k]_{ij} = \frac{h_k}{2}\int_{-1}^{1}\widetilde{c}(\xi)N_i(\xi)N_j(\xi)d\xi$$

Which means that we have:

$$\boxed{[B^{[k]}(\widetilde{u_{FE}},\widetilde{v})]_{ij} = \sum_{i=1}^{p_k+1}\sum_{j=1}^{p_k+1} \alpha_i^{[k]}\beta_j^{[k]}([K_k]_{ij} + [G_k]_{ij})} \tag{9}$$

This defines for each k the aforementioned **stiffness matrix**.

Here we define $[K_k]$ and $[G_k]$ in $\mathbb{R}^{(p_k+1)\times(p_k+1)}$, which are respectively called the **elemental stiffness matrix** and **elemental mass matrix**. Their elements are calculated through numerical integration (due to their definition above), except in the case that c and d are constant functions (then so are $\widetilde{c}$ and $\widetilde{d}$).

In the case that c and d are constants, while noticing due to the symmetry of the usual $\mathcal{L}_2$ inner product that $[K_k]$, $[G_k]$ are symmetric, their elements are (due to the orthogonality of the Legendre polynomials that define the hierarchical basis functions):

$$[K_k]_{ij} = \frac{2d}{h_k} \times \begin{cases} \frac{1}{2}, & i=j=1 \text{ and } i=j=2 \\ -\frac{1}{2}, & i=1,\ j=2\ (i=2,\ j=1) \\ 1, & i=j\geq 3 \\ 0, & \text{elsewhere} \end{cases}$$

$$[G_k]_{ij} = \frac{ch_k}{2} \times \begin{cases} \frac{2}{3}, & i=j=1 \text{ and } i=j=2 \\ \frac{1}{3}, & i=1,\ j=2\ (i=2,\ j=1) \\ -\frac{1}{\sqrt{6}}, & i=1,\ j=3\ (i=3,\ j=1) \\ \frac{1}{3\sqrt{10}}, & i=1,\ j=4\ (i=4,\ j=1) \\ -\frac{1}{\sqrt{6}}, & i=2,\ j=3\ (i=3,\ j=2) \\ -\frac{1}{3\sqrt{10}}, & i=2,\ j=4\ (i=4,\ j=2) \\ \frac{2}{(2i-1)(2i-5)}, & i=j\geq 3 \\ -\frac{1}{(2i-1)\sqrt{(2i-3)(2i+1)}}, & j=i+2,\ i\geq 3 \end{cases}$$

Likewise for $F^{[k]}(\widetilde{v})$, using (8) we gain:

$$[F^{[k]}(\widetilde{v})]_j = \frac{h_k}{2}\int_{-1}^{1}\widetilde{f}(\xi)\sum_{j=1}^{p_k+1}\beta_j^{[k]}N_j(\xi)d\xi = \sum_{j=1}^{p_k+1}\beta_j^{[k]}\frac{h_k}{2}\int_{-1}^{1}\widetilde{f}(\xi)N_j(\xi)d\xi$$
$$\Leftrightarrow \boxed{[F^{[k]}(\widetilde{v})]_j = \sum_{j=1}^{p_k+1}\beta_j^{[k]}[\vec{b_k}]_j} \tag{10}$$

where this defines for each k the aforementioned **load vector**. That is $\forall k = 1, 2, \dots, M$ and $j = 1, 2, \dots, p_k + 1$:

$$[\vec{b_k}]_j = \frac{h_k}{2} \int_{-1}^{1} \widetilde{f}(\xi) N_j(\xi) d\xi$$

Like with the elements of the elemental stiffness and mass matrices, the elements of the load vector are calculated through numerical integration using specialized software.

Now lets get back to (W_M^*). Because we want: $B(u_{FE}, v) = F(v)$, $\forall v \in \mathcal{S}^{\vec{p}}(\bar{I}, \Delta) \Leftrightarrow \sum_{k=1}^{M} B^{[k]}(u_{FE}, v) = \sum_{k=1}^{M} F^{[k]}(v)$, $\forall v \in \mathcal{S}^{\vec{p}}(\bar{I}, \Delta)$, we require that each corresponding term with respect to k in these two sums are equal. This leaves us, with the following linear equations $\forall k = 1, 2, \dots, M$:

$$B^{[k]}(\widetilde{u_{FE}}, \widetilde{v}) = F^{[k]}(\widetilde{v})$$

Where using (9) and (10) we get - by using the definition of matrix multiplication and the euclidean inner product, as well as the properties of said product:

$$\boxed{< \vec{\beta}^{[k]}, [K_k]\vec{\alpha}^{[k]} > + < \vec{\beta}^{[k]}, [G_k]\vec{\alpha}^{[k]} > = < \vec{\beta}^{[k]}, [\vec{F}^{[k]}] >} \tag{11}$$

Where use the following notation, based on (9) and (10):

$$\vec{\beta}^{[k]} = (\beta_1^{[k]}, \dots, \beta_{p_k+1}^{[k]})^T$$
$$\vec{\alpha}^{[k]} = (\alpha_1^{[k]}, \dots, \alpha_{p_k+1}^{[k]})^T$$

$[K_k], [G_k]$ are the elemental stiffness and mass matrices respectively

$[\vec{F}^{[k]}]$ as defined in (10)

Now since for our problem - (W_M^*), we want to find the function u_{FE} that satisfies our condition for **every** function $v \in \mathcal{S}^{\vec{p}}(\bar{I}, \Delta)$, then essentially we want (11) to hold for **every** $\vec{\beta}^{[k]} \in \mathbb{R}^{p_k+1}$. As such, (11) becomes the following linear system in $\mathbb{R}^{p_k+1}$, for every $k = 1, 2, \dots, M$:

$$\boxed{([K_k] + [G_k])\vec{\alpha}^{[k]} = [\vec{F}^{[k]}]} \tag{$\star$}$$

The next step is the construction of the **global linear system** that corresponds to all the elements of the problem. Its solution will provide us with the desired approximation of u at (BVP); u_{FE}.

This procedure is called **assembly**. Initially, we have to construct the so called **pointer matrix** $\mathscr{P}$ which is a matrix in $\mathbb{N}_0^{M\times(p_{max}+1)}$ where $p_{max} := \|\vec{p}\|_\infty$ and $\vec{p} = (p_1, \dots, p_M)^T$ and M is the number of sub-intervals in our mesh grid. This matrix relates the local and global basis functions in the following way: If the i, j element of the point matrix is equal to l, that is $\mathscr{P}_{ij} = l \in \mathbb{N}_0$ then this means that in the ith position, the jth basis function of this element corresponds to lth basis function in the global system, defined below.

Thus we scan all the elements of this matrix and calculate from them the **global matrix** that defines the **global linear system**, in the following way:

Assume that $l = \mathscr{P}_{ki}$ for $k = 1, 2, \dots, M$ and $i = 1, 2, \dots, p_{max} + 1$ and $m = l = \mathscr{P}_{ki}$ for $k = 1, 2, \dots, M$ and $j = 1, 2, \dots, p_{max}+1$. If $l, m \neq 0$, then using cumulative syntax inspired by programming languages we define the following two matrices:

$$[K]_{lm} = [K]_{lm} + [K_k]_{ij}$$
$$[G]_{lm} = [G]_{lm} + [G_k]_{ij}$$

Then the **global matrix** is exactly equal to $K + G$.

If $l = \mathscr{P}_{ki}$ and $l \neq 0$ then we also define the vector corresponding to the linear functional in a similar way $[\vec{F}]_l = [\vec{F}]_l + [\vec{F}^{[k]}]_i$, where $[\vec{F}]$ is knows as the **global load vector.**

Now in spirit of $(\star)$, we have the linear system:

$$([K] + [G])\vec{\alpha} = [\vec{F}] \tag{$\star\star$}$$

where $\vec{\alpha}$ is the vector that contains the unknown coefficients of the solution u_{FE} in the order that is determined by the pointer matrix.

Now that we have found $\vec{\alpha}$ we can finally calculate u_{FE}.
For every $y \in \bar{I} = [a, b]$, we can calculate $u_{FE}(y)$. First we will find in which Ω_k this y point is in, and then we will set $\xi = Q_k^{-1}(y)$. Then:

$$\boxed{u_{FE}(y) = \sum_{i=1}^{p_k+1} \gamma_i N_i(\xi)}$$

where $\gamma_i = [\vec{\alpha}]_{\mathscr{P}_{ki}} = [\vec{\alpha}]_l,\ \forall i = 1, \dots, p_k + 1.$

3 Description of the FEM versions

As we briefly mentioned before there are 3 variants of the FEM. The h version, the p version and the hp version. In this paper will study all three of them.

h-version: For the h version we will use a uniform mesh and a radical-s mesh with the degree of the polynomial in each interval remaining constant, while the mesh is changing. The uniform mesh divides the interval $[a, b]$ in M equal sub-intervals. The width of each interval is the same and equal to $h = \frac{b-a}{M}$. The radical-s mesh with radical exponent $s > 0$ divides $[a, b]$ as follows:

$$x_1 = a,\ x_i = a + (b-a)\left(\frac{i-1}{M}\right)^s,\ i = 2, 3, \ldots, M+1$$

,

p-version: For the p version the mesh remains constant, and the degree of the polynomials is fluctuating. We will use both uniform and geometric-q mesh.The geometric mesh with geometric ratio $q \in (0, 1)$ divides $[a, b]$ as follows:

$$x_1 = a,\ x_i = a + (b-a)q^{M-i+1},\ i = 2, 3, \ldots, M+1$$

hp-version: For the hp version both the mesh and the degrees of the polynomials are fluctuating. For the hp version we will use only the geometric mesh. We note that the number of intervals and the degree of the polynomial basis is the same in every step.

4 Numerical Results

Using the FEM variants mentioned above we will do some numerical calculations and observations. We consider solutions in the form $u(x) = x^\lambda - x$, for different values of λ. The domain will be the interval $[0, 1]$ and the functions $d(x), c(x)$ will be equal to one. Now that we know $u(x)$ we can evaluate $f(x)$. The $u(x)$ satisfies the differential equation of (BVP). So we define,

$$f(x) = -\lambda(\lambda-1)x^{\lambda-2} + x^\lambda - x$$

The variable λ determines the possibility of a singularity. If $\lambda \geq 2$ then u is smooth enough and our analysis so far is valid. If $\lambda \leq \frac{1}{2}$ then $u \notin H_0^1([0, 1])$ and the FEM can't be used in the form that we have seen so far. If $\lambda \in (\frac{1}{2}, 2)$ the rate of convergence can be adversely altered.

From (W) we get the weak form problem:

$$\text{Find a function } u \in H_0^1(I) \text{ such that: } B(u, v) = F(v),\ \forall v \in H_0^1(I) \qquad \text{(W)}$$

where $B(u,v) = \int_I (u'(x)v'(x) + u(x)v(x))dx$ a bi-linear form and $F(v) = \int_I f(x)v(x)dx$ a linear functional, with $f(x) = -\lambda(\lambda-1)x^{\lambda-2} + x^{\lambda} - x$ and $I = [0,1]$.

The **energy norm** is defined as follows:

$$||u||_E := [B(u,u)]^{\frac{1}{2}}, \forall u \in H_0^1(I)$$

Let $u = u_{ex}$. Then from (W) and letting $v = u = u_{ex}$,

$$||u_{ex}||_E = [B(u_{ex}, u_{ex})]^{\frac{1}{2}} = [F(u_{ex})]^{\frac{1}{2}}$$

$$\Rightarrow ||u_{ex}||^2 = [F(u_{ex})] = \int_0^1 (-\lambda(\lambda-1)x^{\lambda-2} + x^{\lambda} - x)(x^{\lambda} - x)dx = \frac{6\lambda^4 + e\lambda^3 - 28\lambda^2 + 11\lambda + 4}{12\lambda^3 + 24\lambda^2 - 3\lambda - 6}$$

Because the bi-linear form is symmetrical,

$$||u_{FE} - u_{ex}||_E = \sqrt{||u_{FE}||_E^2 - ||u_{ex}||_E^2}$$

So, the percentage of the relative error is given as follows:

$$relE = 100 \times \frac{||u_{FE} - u_{ex}||_E}{||u_{ex}||_E} = 100 \times \frac{\sqrt{||u_{FE}||_E^2 - ||u_{ex}||_E^2}}{||u_{ex}||_E}$$

Let $\lambda = 7.1$ We graph the percentage of the relative error against the degrees of freedom in loglog or semilog axes (will specify in each case the axes used) for the:

(i) h version with uniform mesh for $M = 2^N$ intervals, $N = 1, \ldots, 5$ and the degree of the polynomial being $p = 1$ (constant). (loglog)
(ii) h version with uniform mesh for $M = 2^N$ intervals, $N = 1, \ldots, 5$ and the degree of the polynomial being $p = 2$ (constant). (loglog)
(iii) p version with uniform mesh for 1 (constant) interval and the degrees of the polynomials being $p = 1, \ldots, 6$. (loglog)
(iv) p version with uniform mesh for 4 (constant) intervals and the degrees of the polynomials being $p = 1, \ldots, 6$. (loglog)

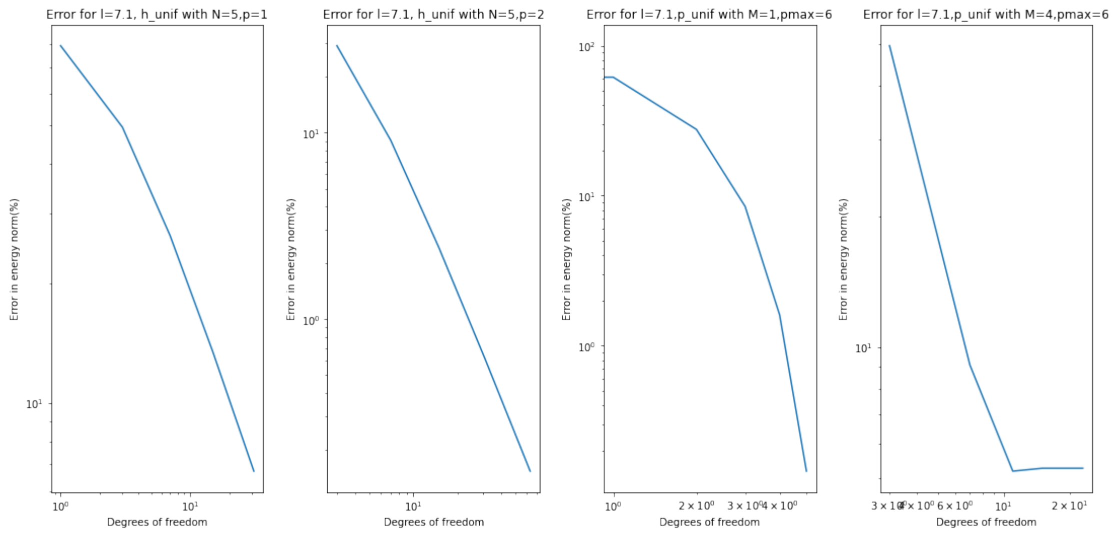

Figure 1: Percentage error in energy norm against DOF for h-FEM and p-FEM with uniform mesh.

We observe that for the h-FEM the convergence rate is algebraic. The slope in (i) is $-0.94907109(\approx -1)$ and in (ii) is $-1.94994539(\approx -2)$. This result was expected, as from theory we know that for the error $E \sim DOF^{-p}$, where DOF is the degrees of freedom. Now, we logarithm to get $log(E) \sim -plog(DOF)$ which represents a line with slope $-p$ and intercept 0, if we use loglog axes.
For the p version, the slope in (iii) is -1.15702719 and in (iv) is -1.24566530. We know that for high λ the p-FEM has almost exponential convergence rate, as u is analytic in $[0,1]$. Specifically, $E \sim C(s)DOF^{-s}$, $\forall s \in [0,p]$. We logarithm to get, $log(E) \sim log(C(s)) - slog(DOF))$, which means that if we use loglog axes it is a line with slope -s. So we stand correct that s is indeed in $[0,p]$.

Let now $\lambda = 2.1$. We once again graph the percentage of the relative error against the degrees of freedom in loglog or semilog axes (will specify in each case the axes used) for the:

(i) h version with uniform mesh for $M = 2^N$ intervals, $N = 1,\ldots,5$ and the degree of the polynomial being $p = 1$ (constant). (loglog)
(ii) h version with uniform mesh for $M = 2^N$ intervals, $N = 1,\ldots,5$ and the degree of the polynomial being $p = 2$ (constant). (loglog)
(iii) p version with uniform mesh for 1 (constant) interval and the degrees of the polynomials being $p = 1,\ldots,6$. (loglog)
(iv) h version with radical-s mesh, for $s = 0.15$, $M = 2^N$ intervals, $N = 1,\ldots,5$ and the degree of the polynomial being $p = 2$ (constant). (loglog)

(v) p version with geometrical-q mesh for $q = 0.15$, 4 (constant) intervals and the degrees of the polynomials being $p = 1, \ldots, 6$. (loglog)
(vi) hp version with geometrical-q mesh for $q = 0.15$, $M = 2^N$ intervals ,$N = 1, \ldots, 5$ and the degrees of the polynomials being $p = M$. (semilogy)

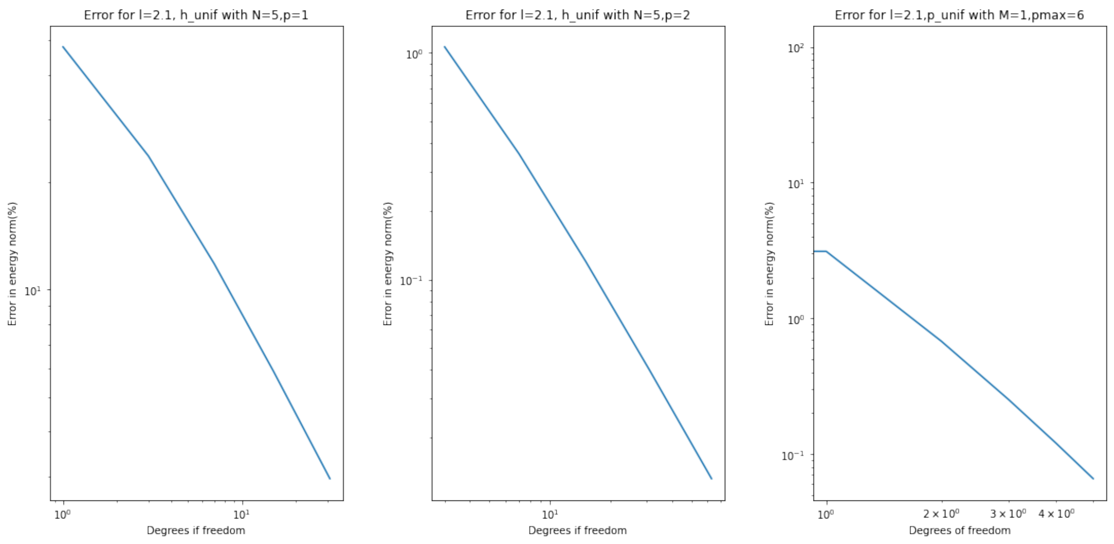

Figure 2: Percentage error in energy norm against DOF for h-FEM and p-FEM with uniform mesh.

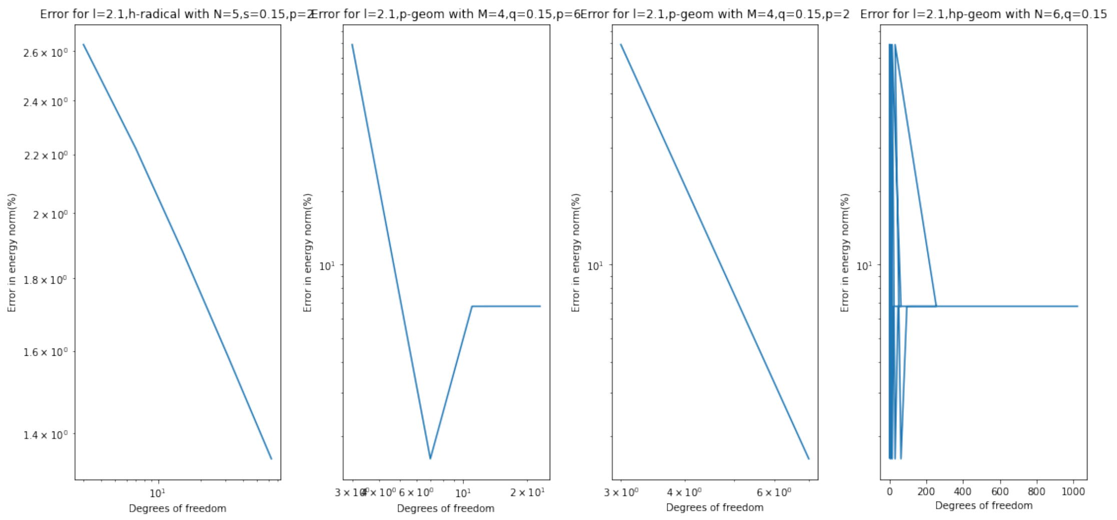

Figure 3: Percentage error in energy norm against DOF for h-FEM with radical mesh, p-FEM with geometrical mesh and hp-FEM with geometrical mesh.

For the h version again we have algebraic convergence rate, because the error is of order $\mathcal{O}(DOF^{-p})$, which is also justified by the graphs $(i),(ii),(iii)$. On the other hand, for the p version we won't have exponential rate of convergence as the theory suggests, but algebraic as λ is low. From the graph (iv) we get a line with slope -2.68721159. For (v) we almost have a straight line with slope -4.59146629. We see that if we change p to be equal to 2 then we see that the anomaly that was apparent before is nullified, thus we get a line with slope -3.54044444. Finally for the hp version we know from our theory that $E \sim e^{-\beta N^{\gamma}}$. Now, we logarithm both sides. So, $log(E) \sim -\beta N^{\gamma}$, which shows that hp has an exponential convergence rate. From the graph we are not able to confirm this and instead we are left a hodgepodge. Due to the fact that all the other graphs were correct and produced predictable results and were based on code that was also used to produce this graph this leaves us to "blame" other extraneous factors, like problem inputs, or errors regarding numerical precision.

Finally, we want to compare the absolute relevant error, $\frac{|u_{ex}(x)-u_{fe}(x)|}{|u_{ex}(x)|}$, in the interval $I=[0,1]$. We will take four cases for $\lambda=2.1$ and graph the absolute error and also the u_{fe} solution and u_{ex} solution.

1. h version with uniform mesh with 10 intervals and p=1.
2. h version with uniform mesh with 1 interval and p=6.
3. p version with geometric-q mesh with 10 intervals, q=0.15 and p=2.
3. p version with geometric-q mesh with 10 intervals, q=0.15 and p=4.

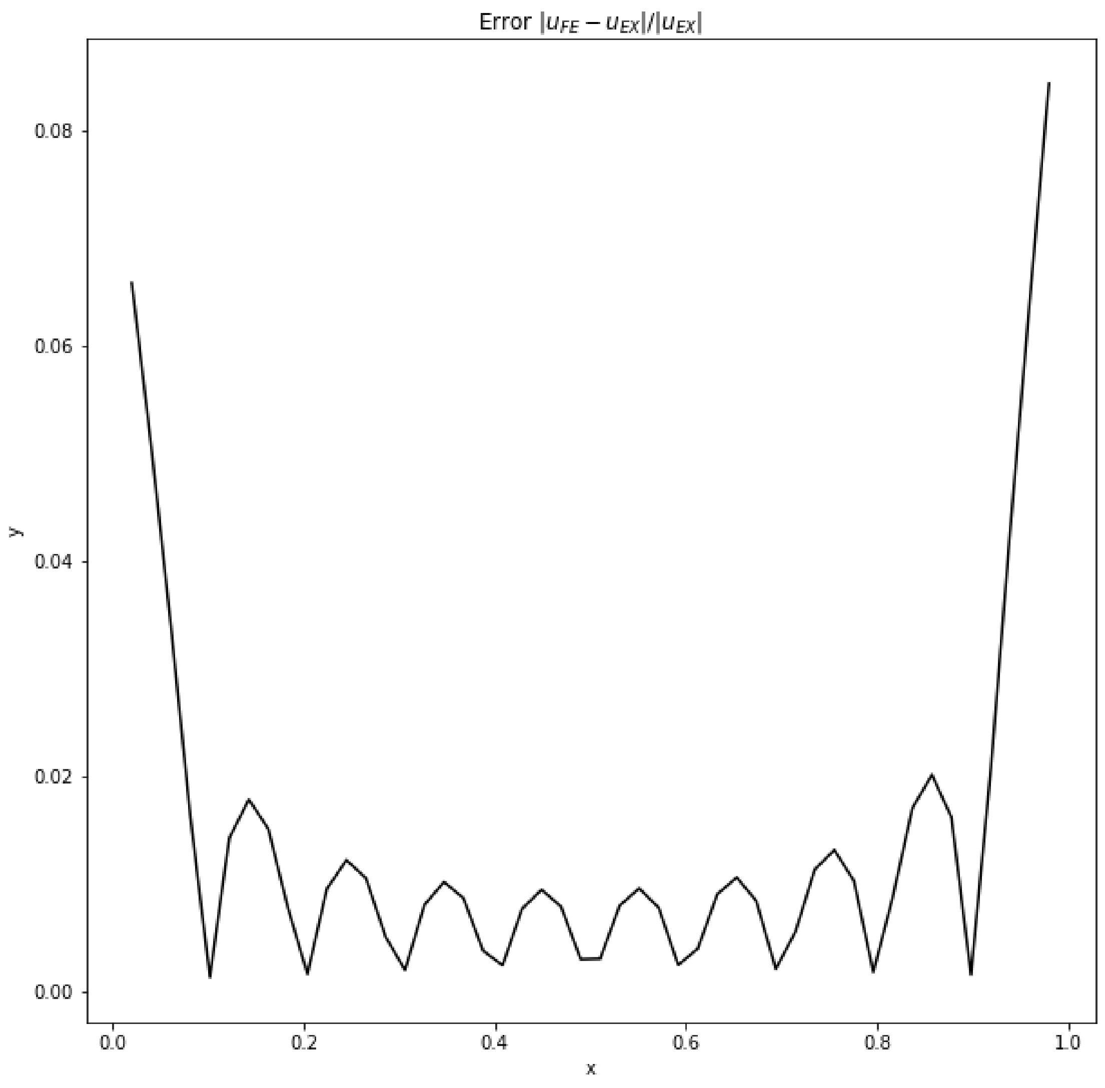

Figure 4: Absolute error of the h-FEM with uniform mesh, 10 intervals and p=1.

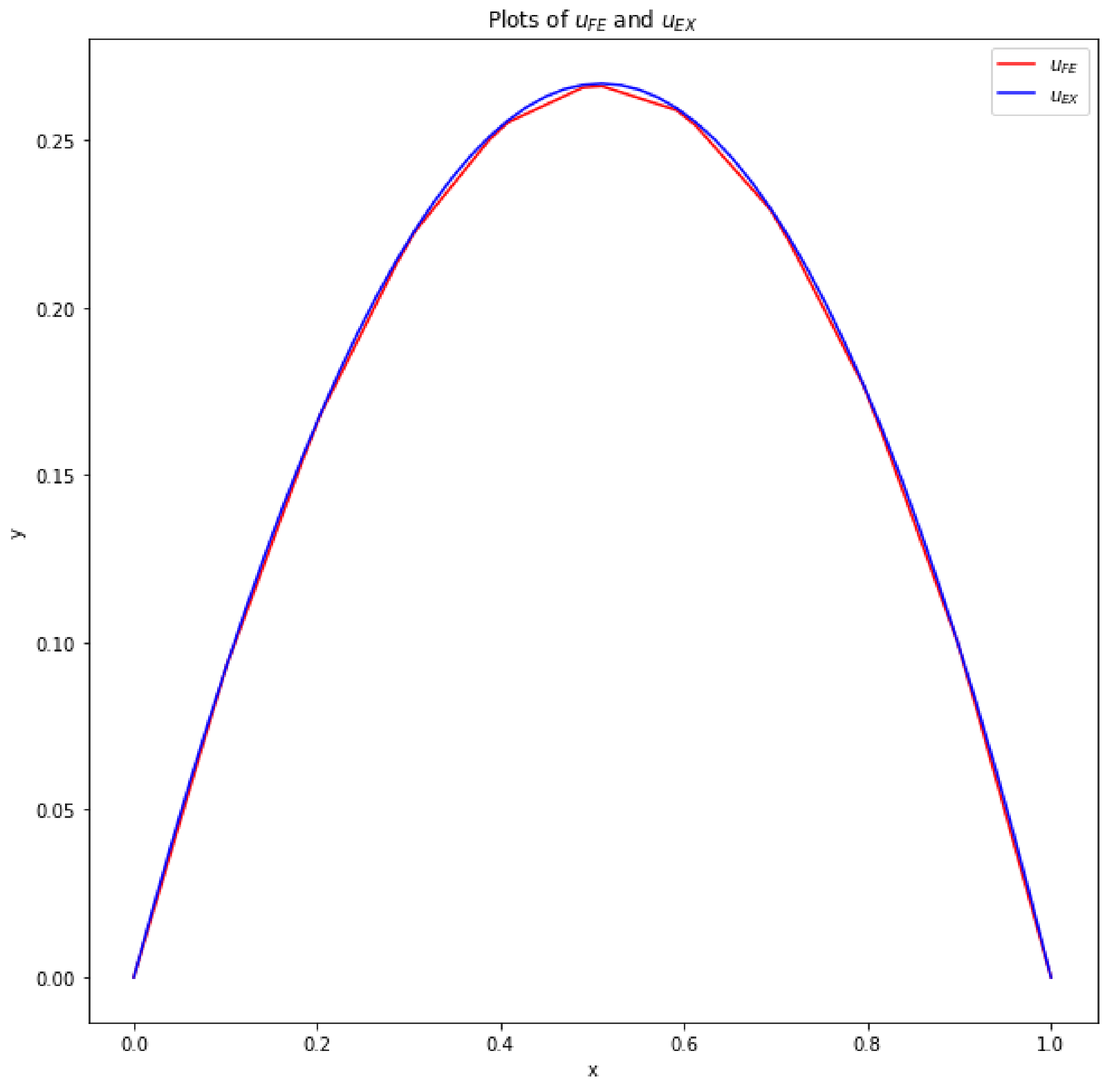

Figure 5: Finite element solution and exact solution of the h-FEM with uniform mesh, 10 intervals and p=1.

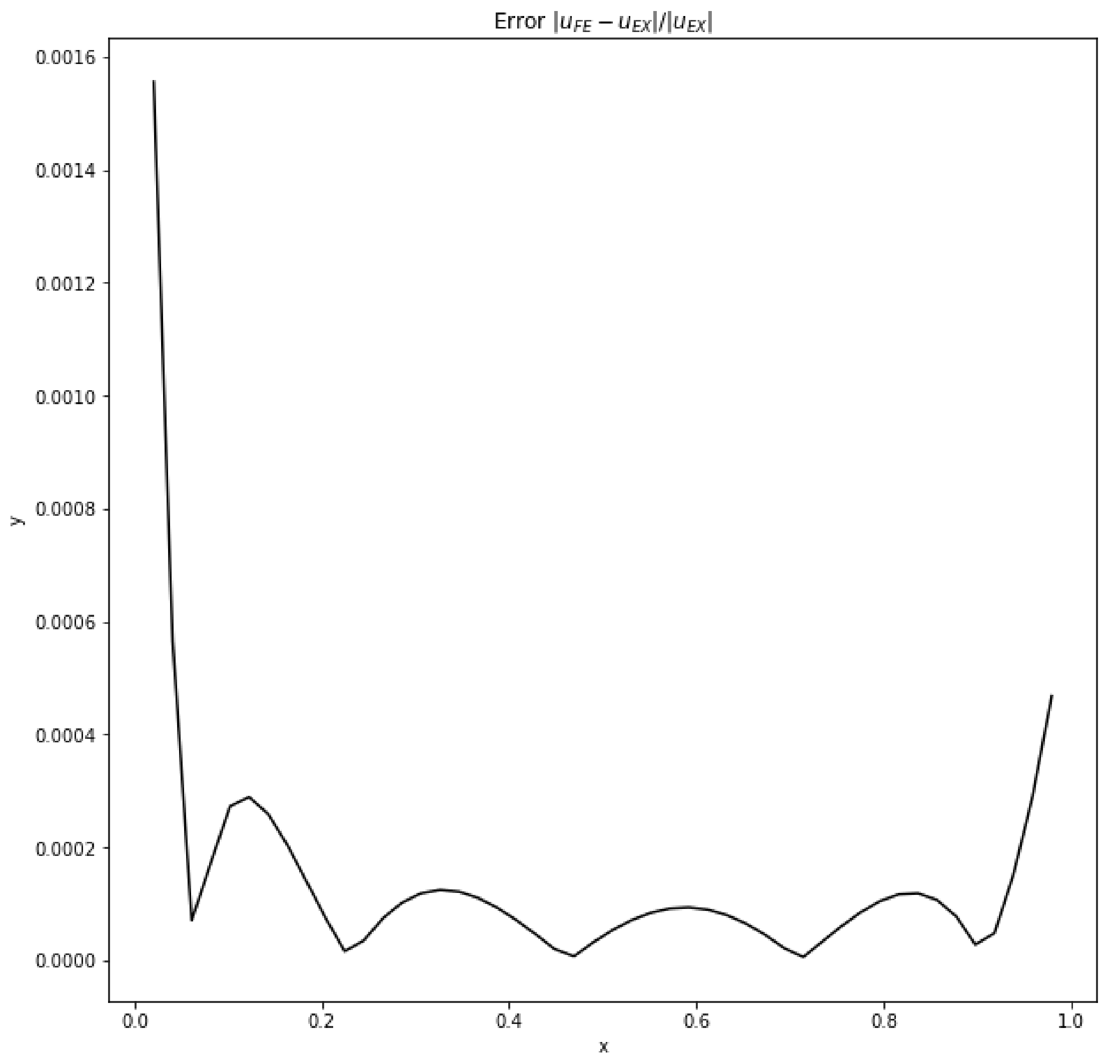

Figure 6: Absolute error of the h-FEM with uniform mesh, 1 intervals and p=6.

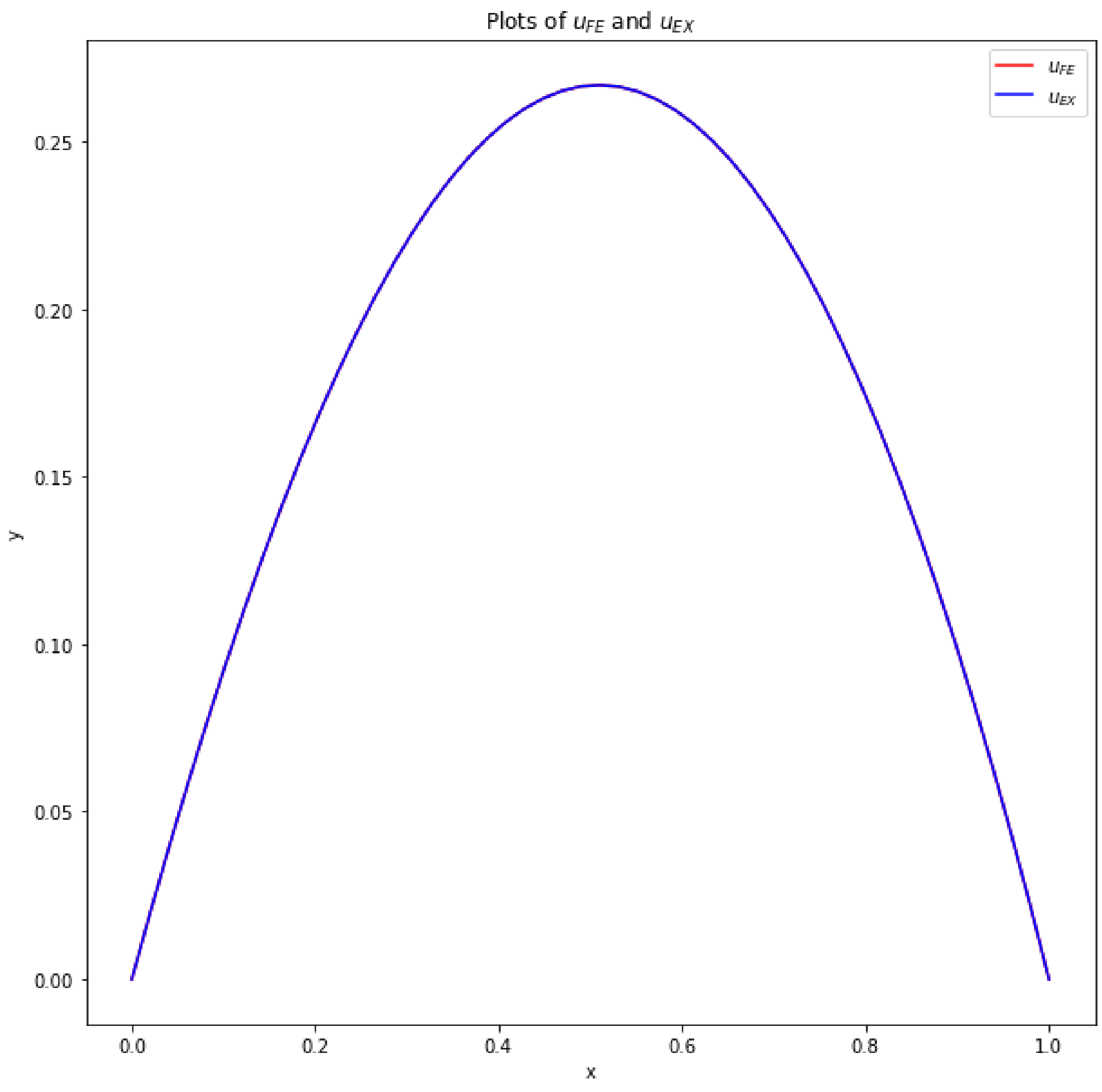

Figure 7: Finite element solution and exact solution of the h-FEM with uniform mesh, 1 intervals and p=6.

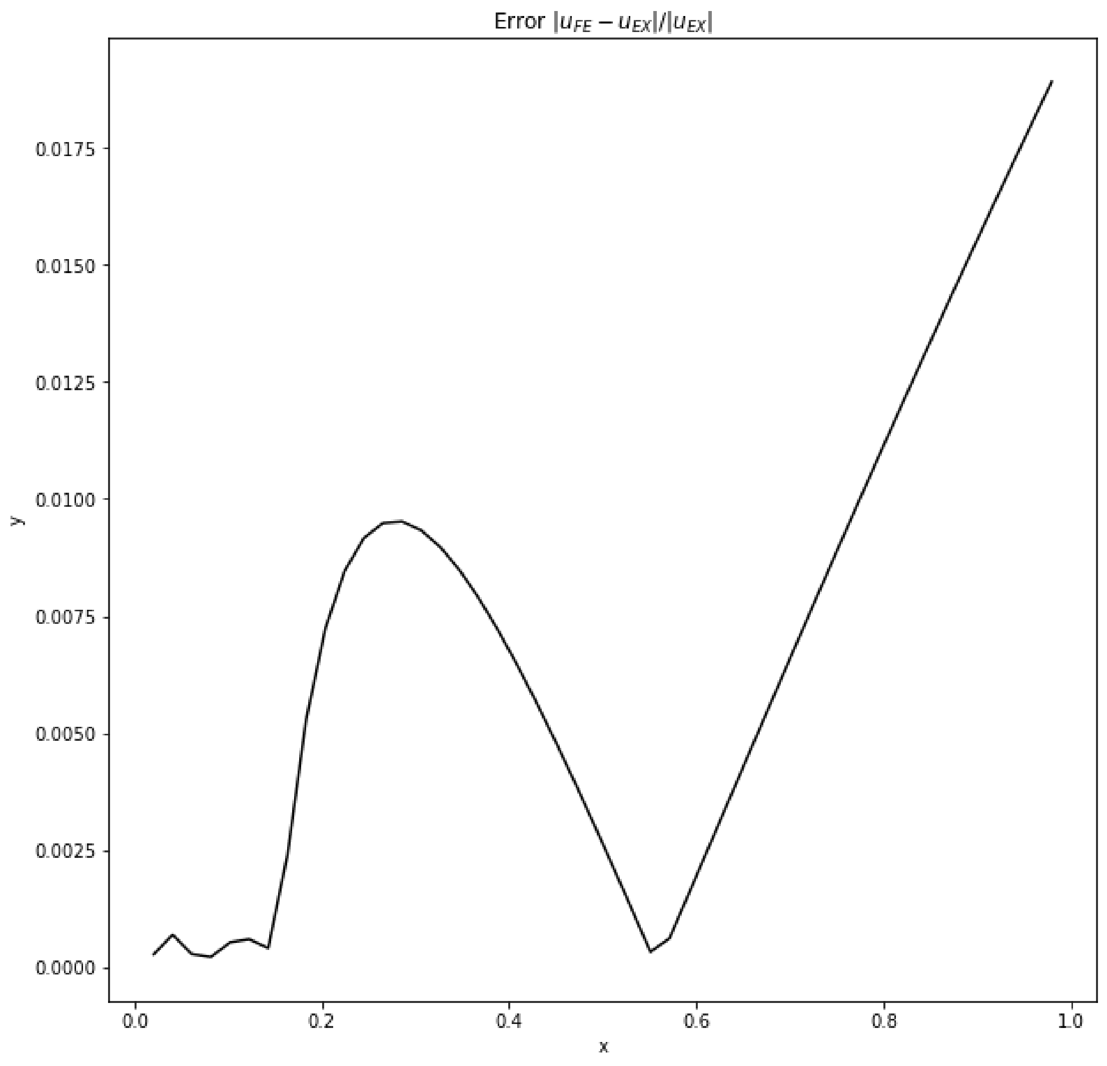

Figure 8: Absolute error of the p-FEM with geometrical mesh, 10 intervals, q=0.15 and p=2.

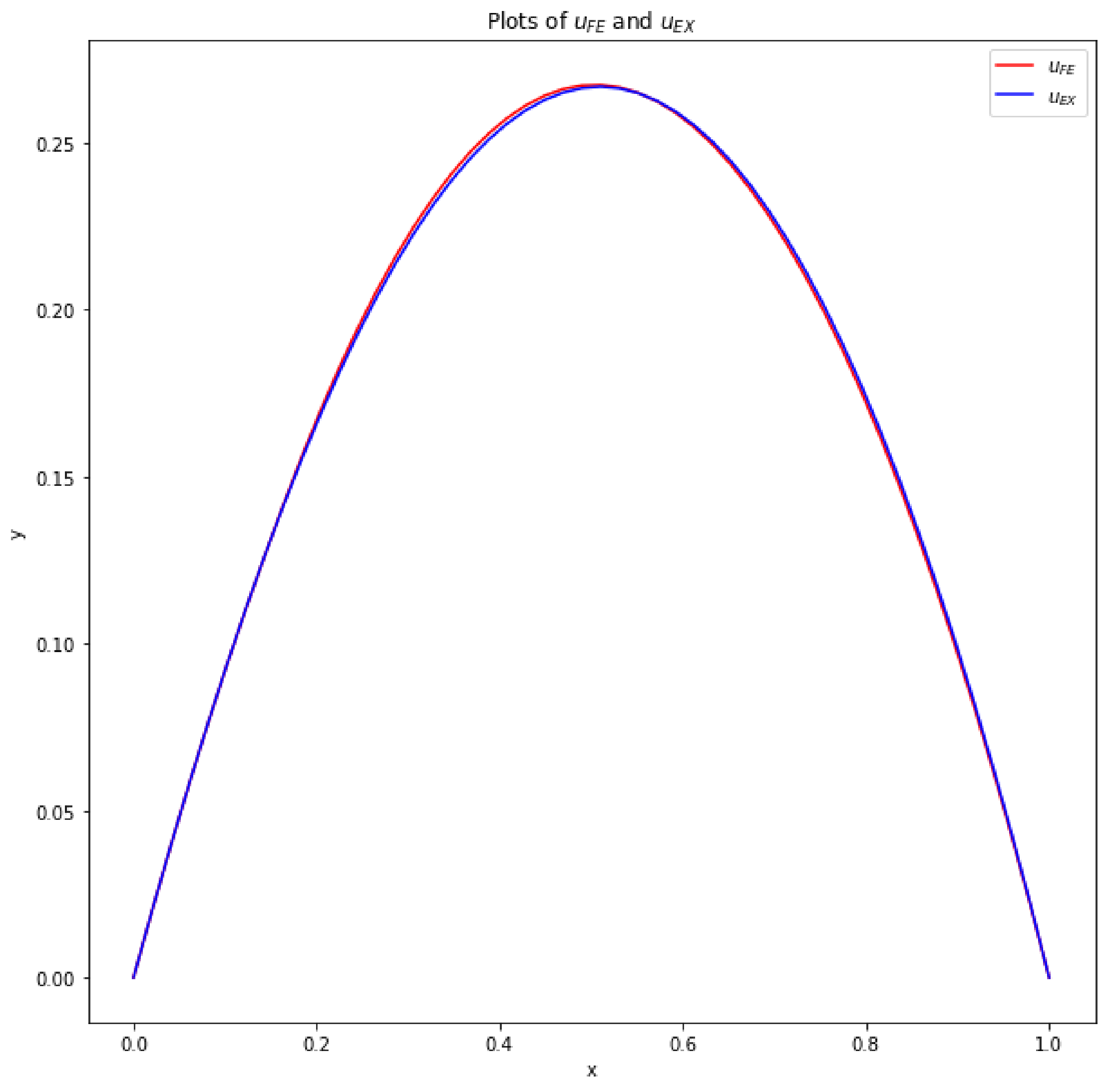

Figure 9: Finite element solution and exact solution of the p-FEM with geometrical mesh, 10 intervals, q=0.15 and p=2.

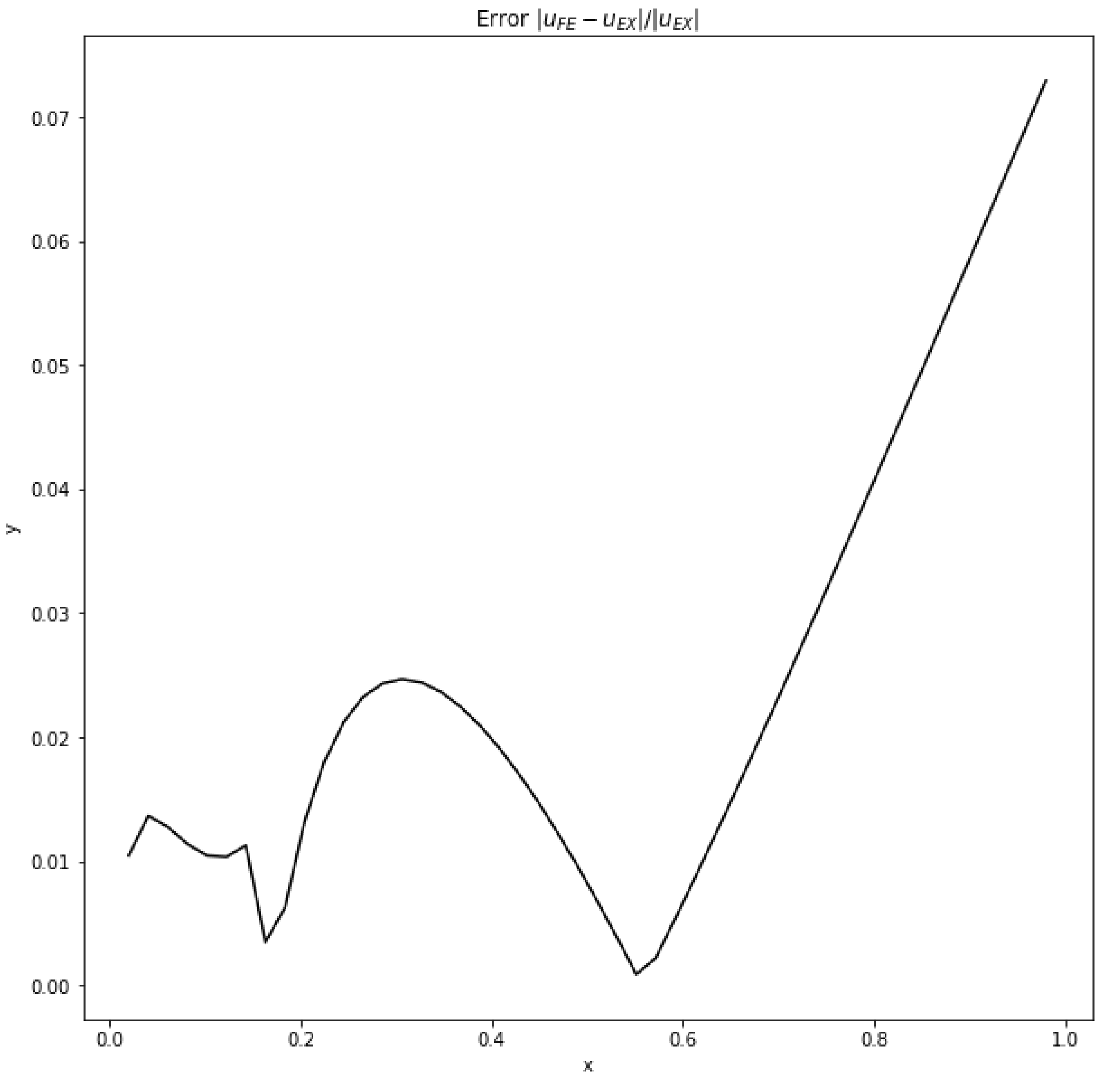

Figure 10: Absolute error of the p-FEM with geometrical mesh, 10 intervals, q=0.15 and p=4.

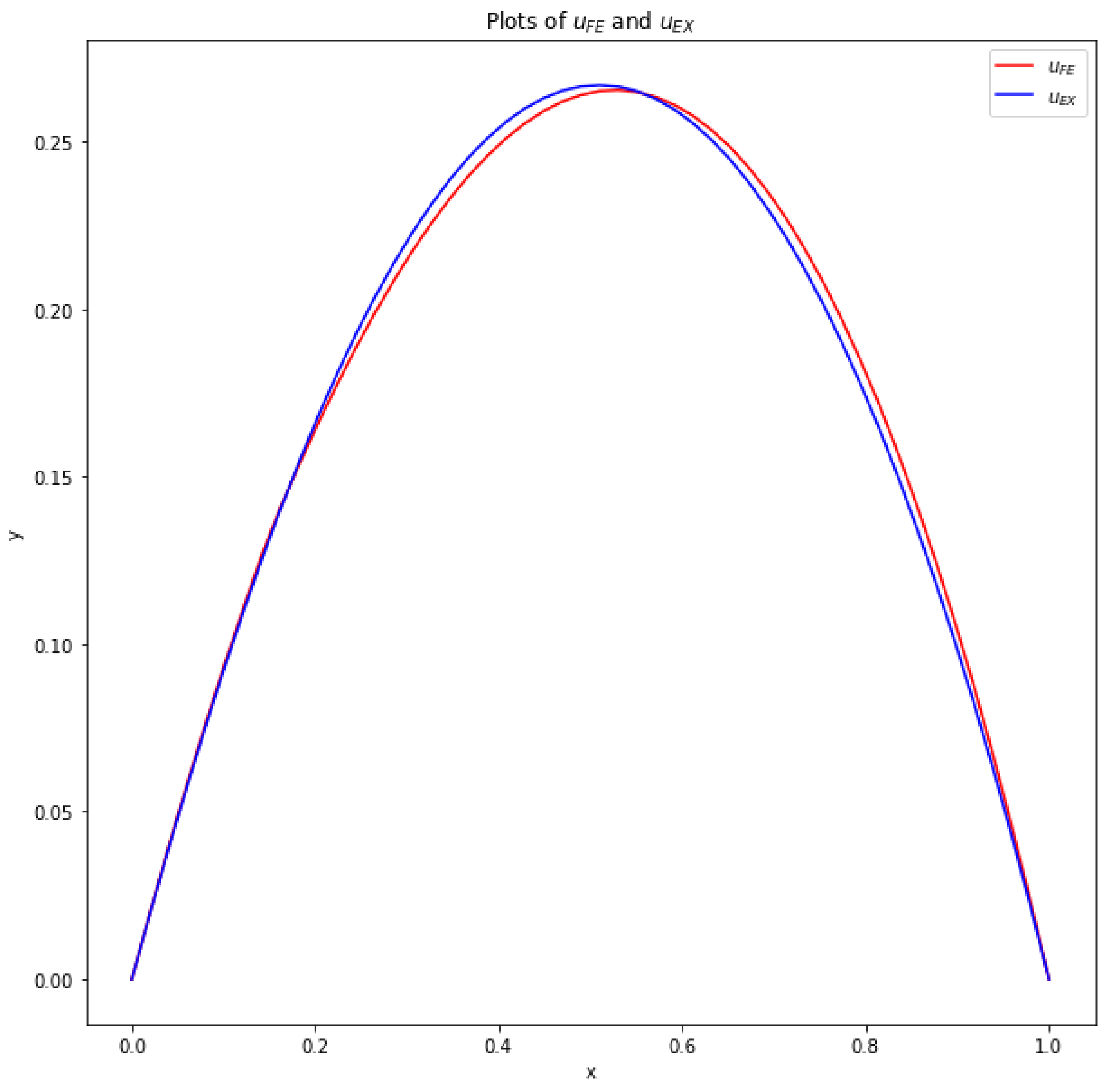

Figure 11: Finite element solution and exact solution of the p-FEM with geometrical mesh, 10 intervals, q=0.15 and p=4.

5 Python code that was used

```python
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
import sympy
import matplotlib.pyplot as plt
from scipy import integrate
from scipy.special import legendre
from numpy.polynomial.legendre import Legendre
import matplotlib.ticker as mtick
import itertools


# Legendre polynomial
def leg(n, x):
    return Legendre(np.concatenate((np.zeros(n), np.array([1]))))(x)

def hbasis(i,x):
#  Evaluates the function Ni at x
    if i==0:
        Ni=0.5 *(1-x)
    elif i==1:
        Ni=0.5 *(1+x)
    else:
        Ni=(np.sqrt(1/(4*(i+1)-6)))*(leg(i,x)-leg(i-2,x))
    return Ni


def stifness_matrix(p):
#evaluates the elemental stifness matrix of size (p+1)x(p+1)
    K=np.zeros((p+1,p+1))
    K[0,0]=K[1,1]=0.5
    K[0,1]=K[1,0]=-0.5
    if p>=1:
        for i in range(2,p+1):
            K[i,i]=1
    return K

def stifness_matrix(p):
#evaluates the elemental stifness matrix of size (p+1)x(p+1)
    K=np.zeros((p+1,p+1))
    K[0,0]=K[1,1]=0.5
    K[0,1]=K[1,0]=-0.5
    if p>=1:
        for i in range(2,p+1):
            K[i,i]=1
    return K

def mass_matrix(p):
# Evaluates the elemental mass matrix of size (p+1)x(p+1)
    G=np.zeros((p+1,p+1))
    G[0,0]=G[1,1]=2/3
    G[0,1]=G[1,0]=1/3
    if p>=2:
        G[0,2]=G[1,2]=G[2,0]=G[2,1]=-1/np.sqrt(6)
        for i in range(2,p+1):
            G[i,i]=2/((2*(i+1)-1)*((2*(i+1)-5)))
    if p>=3:
        G[0,3]=G[3,0]=1/3*np.sqrt(10)
        G[1,3]=G[3,1]=-1/3*np.sqrt(10)
        for i in range(2,p+1):
            if i+2<p+1:
                G[i,i+2]=G[i+2,i]=(-1)/(((2*(i+1)-1)*np.sqrt(((2*(i+1)-3)* \
                ((2*(i+1)+1))))))
    return G

def load_vector(x_k,x_kk,p_k,f):
#Evaluates the elemental load vector
    vals=[]
    for i in range(p_k+1):
        g = lambda t : f((1-t)*x_k/2+(1+t)*x_kk/2)*hbasis(i,t)
        vals.append(integrate.quad(g,-1,1)[0])
    b=np.array(vals)
    return b

def el_stiff(x_k,x_kk,p_k):
    h_k=x_kk-x_k
    Kk=(2/h_k)*stifness_matrix(p_k)
    return Kk

def el_mass(x_k,x_kk,p_k):
    h_k=x_kk-x_k
    Gk=(h_k/2)*mass_matrix(p_k)
    return Gk

def pointer(M,p):
# [P] = pointer(M,p)
#
# Calculates the pointer matrix P, such that
# P(i,j)=k means that over the ith element
# the jth local basis function corresponds
# to the kth global basis function.
#
# M is the number of elements
# p is the degree vector of size M
# P is M by (max(p)+1)
#
    pmax=max(p)
    P=np.zeros((M,pmax+1))
    P=P.astype(int)
    for i in range(M):
        P[i,0] = i
        P[i,1] = i+1
    P[M-1,1]=0
    for i in range(M):
        for j in range(2,1+p[i]):
            P[i,j] = M
            M=M+1
    return P

def global_matrix(x,p):
    #Evaluates the elemental matrices, stiffnes(Kk) and mass(Gk) and then
    #the global matrix (G+K)
    P=pointer(len(x)-1,p)
    K=np.zeros((sum(p)-1,sum(p)-1))
    G=np.zeros((sum(p)-1,sum(p)-1))
```

```python
    for k in range(len(x)-1):
        Kk=np.zeros((p[k]+1,p[k]+1))
        Gk=np.zeros((p[k]+1,p[k]+1))
        Kk=el_stiff(x[k],x[k+1],p[k])
        Gk=el_mass(x[k],x[k+1],p[k])
        for i in range(p[k]+1):
            l=P[k,i]
            for j in range(p[k]+1):
                m=P[k,j]
                if l!=0 and m!=0:
                    K[l-1,m-1]=K[l-1,m-1]+Kk[i,j]
                    G[l-1,m-1]=G[l-1,m-1]+Gk[i,j]
    GL=K+G
    return GL

def el_load(x_k,x_kk,p_k,f):
#evaluates the elemental load vector
    h_k=x_kk-x_k
    Fk=(h_k/2)*load_vector(x_k,x_kk,p_k,f)
    return Fk

def global_load_vector(x,p,f):
#Evaluates the global load vector
    P=pointer(len(x)-1,p)
    F=np.zeros((sum(p)-1))
    for k in range(len(x)-1):
        Fk=el_load(x[k],x[k+1],p[k],f)
        for i in range(p[k]+1):
            l=P[k,i]
            if l!=0:
                F[l-1]=F[l-1]+Fk[i]
    return F

def fem_solution(x,p,f):
# Evaluates the finite element method solution and the coifficients that
#generate the solution
    A=global_matrix(x,p)
    b=global_load_vector(x,p,f)
    a=np.linalg.solve(A,b)
    return a
```

```

def energy_norm(x,p,f,n):
# Evaluates the energy norm of the finite element method solution and it finds
# the percentage of the error
    a=fem_solution(x,p,f)
    DOF=len(a)
    enorm = np.dot(a, np.array(global_load_vector(x,p,f)))
    y = lambda z : (z**n-z)*f(z)
    enorm_uex=integrate.quad(y,0,1)[0]
    relE=100*np.sqrt(abs(enorm-enorm_uex)/abs(enorm_uex))
    return enorm,relE,DOF

def pairwise(iterable):
    a, b = itertools.tee(iterable)
    next(b, None)
    return zip(a,b)

def solid1d(y,x,M,p,f):
#Evaluates the finite element method solution for every y that belongs to
#[a,b] interval
    x_intervals = list(pairwise(x))
    k = [0]
    for point in y[1:]:
        for i, interval in enumerate(x_intervals[k[-1]:]):
            if point>=interval[0] and point<interval[1]:
                k.append(k[-1]+i)
                break
    k.append(len(x_intervals)-1)
    ksi=[]
    for k_val, y_i in zip(k,y):
        ksi.append((2*y_i-x_intervals[k_val][0]-x_intervals[k_val][1])/ \
                   (x_intervals[k_val][1]-x_intervals[k_val][0]))
    P=pointer(M,p)
    ufe=[]
    c=fem_solution(x,p,f)
    for k_val, ksi_i in zip(k,ksi):
        proxy=[]
        for i in range(1,p[k_val]+2):
            l=P[k_val,i-1]
            if l!=0:
                proxy.append(c[l-1]*hbasis(i-1,ksi_i))
            else:
                proxy.append(0)
        ufe.append(sum(proxy))
    return ufe


def h_unif(M,pmax,a,b):
#It creates a list of the uniform mesh and a list with the degrees of the
#polynomials which are constant.
    x=[]
    p=[]
    h=(b-a)/M
    for i in range(M+1):
        z=a+i*h
        x.append(z)
    for j in range(M):
        p.append(pmax)
    return x,p


def h_radical(M,pmax,s,a,b):
#It creates a list of the root-s mesh and a list with the degrees of the
#polynomials which are constant.
    p=[]
    x=[]
    for i in range(M+1):
        z=a+(b-a)*((i/M)**s)
        x.append(z)
    for j in range(M):
        p.append(pmax)
    return x,p

def p_unif(M,pmax,a,b):
#It creates a list of the uniform mesh which is constant and a list with
#the degrees of the polynomials that goes 1:pmax.
    p=[]
    x=[]
    h=(b-a)/M
    for i in range(M+1):
        z=a+i*h
        x.append(z)
    for j in range(1,pmax+1):
        p.append(j)
    return x,p

def p_geom(M,pmax,q,a,b):
#It creates a list of the geometric-q mesh which is constant and a list with
# the degrees of the polynomials that goes 1:pmax.
    p=[]
    x=[a]
    for i in range(2,M+2):
        z=a+(b-a)*(q**(M-i+1))
        x.append(z)
    for j in range(1,pmax+1):
        p.append(j)
    return x,p

def hp_geom(M,q,a,b):
#It creates a list of the geometric-q mesh and a list with the degrees of the
# polynomials that goes 1:M, M:=number of elements.
    x=[a]
    p=[]
    pmax=M
    for i in range(2,M+2):
        z=a+(b-a)*(q**(M-i+1))
        x.append(z)
    for j in range(1,pmax+1):
        p.append(j)
    return x,p



# In[2]:


#f(x)=x**n-x
#n=7.1,a=0,b=1
#Graphs the error against DOF  in logaritmic axes
#First it uses h fem with uniform mesh for polynomial degrees 1 and then 2
```

```
a=0
b=1
n=7.1
N=5
f = lambda x : (x**n)-x-n*(n-1)*(x**(n-2))
fig, axes = plt.subplots(1,4, figsize=(15,7))
fig.tight_layout()
fig.subplots_adjust(wspace=0.3)
for j in range(1,3):
    pmax=j
    errors=[]
    DOF=[]
    for i in range(1,N+1):
        M=2**i
        x=h_unif(M,pmax,a,b)[0]
        p=h_unif(M,pmax,a,b)[1]
        errors.append(energy_norm(x,p,f,n)[1])
        DOF.append(energy_norm(x,p,f,n)[2])
    axes[j-1].loglog(DOF,errors)
    axes[j-1].set_xlabel("Degrees of freedom")
    axes[j-1].set_ylabel("Error in energy norm(%)")
    axes[j-1].set_title(f"Error for l=7.1, h_unif with N=5,p={j}")
    print(f"the slope is:",(np.log(errors[-1])-np.log(errors[-2]))/ \
          (np.log(DOF[-1])-np.log(DOF[-2])))

#Second it uses p fem with uniform mesh with 1 element and then two elements
#for polynomial degrees 1,....,6
P_uni_list=[1,4]
for j,Mp in enumerate(P_uni_list):
    pmax=6
    errors=[]
    DOF=[]
    x=p_unif(Mp,pmax,a,b)[0]
    p=p_unif(Mp,pmax,a,b)[1]
    for i in p:
        p_list=[]
        for k in range(Mp):
            p_list.append(i)
        errors.append(energy_norm(x,p_list,f,n)[1])
        DOF.append(energy_norm(x,p_list,f,n)[2])
    axes[j+2].loglog(DOF,errors)
    axes[j+2].set_xlabel("Degrees of freedom")
    axes[j+2].set_ylabel("Error in energy norm(%)")
    axes[j+2].set_title(f"Error for l=7.1,p_unif with M={Mp},pmax=6")
    print(f"the slope is:",(np.log(errors[-4])-np.log(errors[-5]))/ \
          (np.log(DOF[-4])-np.log(DOF[-5])))


# In[3]:


#f(x)=x**n-x
#n=2.1,a=0,b=1
#Graphs the error against DOF  in logaritmic axes
#First it uses h fem with uniform mesh for polynomial degrees 1 and then 2
a=0
b=1
n=2.1
N=5
f = lambda x : (x**n)-x-n*(n-1)*(x**(n-2))
fig, axes2 = plt.subplots(1,3, figsize=(15,7))
fig.tight_layout()
fig.subplots_adjust(wspace=0.3)
for j in range(1,3):
    pmax=j
    errors=[]
    DOF=[]
    for i in range(1,N+1):
        M=2**i
        x=h_unif(M,pmax,a,b)[0]
        p=h_unif(M,pmax,a,b)[1]
        errors.append(energy_norm(x,p,f,n)[1])
        DOF.append(energy_norm(x,p,f,n)[2])
    axes2[j-1].loglog(DOF,errors)
    axes2[j-1].set_xlabel("Degrees if freedom")
    axes2[j-1].set_ylabel("Error in energy norm(%)")
    axes2[j-1].set_title(f"Error for l=2.1, h_unif with N=5,p={j}")
    print(f"the slope is:",(np.log(errors[-1])-np.log(errors[-2]))/ \
          (np.log(DOF[-1])-np.log(DOF[-2])))
#Second it uses p fem with uniform mesh with 1 element and then two elements
```

```python
#for polynomial degrees 1,....,6
M=1
pmax=6
errors=[]
DOF=[]
x=p_unif(M,pmax,a,b)[0]
p=p_unif(M,pmax,a,b)[1]
for i in p:
    p_list=[]
    for k in range(M):
        p_list.append(i)
    errors.append(energy_norm(x,p_list,f,n)[1])
    DOF.append(energy_norm(x,p_list,f,n)[2])
axes2[2].loglog(DOF,errors)
axes2[2].set_xlabel("Degrees of freedom")
axes2[2].set_ylabel("Error in energy norm(%)")
axes2[2].set_title(f"Error for l=2.1,p_unif with M={M},pmax=6")
print(f"the slope is:",(np.log(errors[-1])-np.log(errors[-2]))/ \
      (np.log(DOF[-1])-np.log(DOF[-2])))



# In[4]:


#f(x)=x**n-x
#n=2.1,a=0,b=1
#Graphs the error against DOF  in logaritmic axes
a=0
b=1
n=2.1
f = lambda x : (x**n)-x-n*(n-1)*(x**(n-2))
fig, axes3 = plt.subplots(1,4, figsize=(15,7))
fig.tight_layout()
fig.subplots_adjust(wspace=0.3)
j=0

#It uses h radical fem for N=5, s=0.15 for polynomial degrees 2
N=5
s=0.15
```

```
pmax=2
errors=[]
DOF=[]
for i in range(1,N+1):
    M=2**i
    x=h_radical(M,pmax,s,a,b)[0]
    p=h_radical(M,pmax,s,a,b)[1]
    errors.append(energy_norm(x,p,f,n)[1])
    DOF.append(energy_norm(x,p,f,n)[2])
axes3[j].loglog(DOF,errors)
axes3[j].set_xlabel("Degrees of freedom")
axes3[j].set_ylabel("Error in energy norm(%)")
axes3[j].set_title(f"Error for l=2.1,h-radical with N=5,s=0.15,p=2")
print(f"the slope is:",(np.log(errors[-1])-np.log(errors[-2]))/ \
      (np.log(DOF[-1])-np.log(DOF[-2])))
#It uses p fem with geometric mesh for M=4, q=0.15 for polynomial degrees
#p=1,....,6
M=4
q=0.15
pmax=6
errors=[]
DOF=[]
x=p_geom(M,pmax,q,a,b)[0]
p=p_geom(M,pmax,q,a,b)[1]
for i in p:
    p_list=[]
    for k in range(M):
        p_list.append(i)
    errors.append(energy_norm(x,p_list,f,n)[1])
    DOF.append(energy_norm(x,p_list,f,n)[2])
axes3[j+1].loglog(DOF,errors)
axes3[j+1].set_xlabel("Degrees of freedom")
axes3[j+1].set_ylabel("Error in energy norm(%)")
axes3[j+1].set_title(f"Error for l=2.1,p-geom with M=4,q=0.15,p=6")
print(f"the slope is:",(np.log(errors[-5])-np.log(errors[-6]))/ \
      (np.log(DOF[-5])-np.log(DOF[-6])))
#It uses p fem with geometric mesh for M=4, q=0.15 for polynomial degrees 1,2
M=4
q=0.15
pmax=2
```

```
errors=[]
DOF=[]
x=p_geom(M,pmax,q,a,b)[0]
p=p_geom(M,pmax,q,a,b)[1]
for i in p:
    p_list=[]
    for k in range(M):
        p_list.append(i)
    errors.append(energy_norm(x,p_list,f,n)[1])
    DOF.append(energy_norm(x,p_list,f,n)[2])
axes3[j+2].loglog(DOF,errors)
axes3[j+2].set_xlabel("Degrees of freedom")
axes3[j+2].set_ylabel("Error in energy norm(%)")
axes3[j+2].set_title(f"Error for l=2.1,p-geom with M=4,q=0.15,p=2")
print(f"the slope is:",(np.log(errors[-1])-np.log(errors[-2]))/ \
      (np.log(DOF[-1])-np.log(DOF[-2])))

#It uses hp fem with geometric mesh for N=5, q=0.15 for polynomial degrees p=M
N=5
q=0.15
errors=[]
DOF=[]
for i in range(1,N+1):
    M=2**i
    x=hp_geom(M,q,a,b)[0]
    p=hp_geom(M,q,a,b)[1]
    for z in p:
        plist=[]
        for k in range(M):
            plist.append(z)
        errors.append(energy_norm(x,plist,f,n)[1])
        DOF.append(energy_norm(x,plist,f,n)[2])
axes3[j+3].semilogy(DOF,errors)
axes3[j+3].set_xlabel("Degrees of freedom")
axes3[j+3].set_ylabel("Error in energy norm(%)")
axes3[j+3].set_title(f"Error for l=2.1,hp-geom with N=6,q=0.15")
print(f"the slope is:",(np.log(errors[-4])-np.log(errors[-5]))/ \
      (np.log(DOF[-4])-np.log(DOF[-5])))
```

```
# In[5]:


import warnings
warnings.filterwarnings('ignore')

#n=2.1
#a=0,b=1
#Graphs the absolute error between the finite element solution(uFE)and the
#exact solution (uEX)

a = 0
b = 1
n = 2.1
M = 10
q = 0.15
f = lambda x : (x**n)-x-n*(n-1)*(x**(n-2))
y = np.linspace(0,1)
u_Ex_fun = lambda x : (x-x**n)

#First;y I use the h fem with uniform mesh for M=10,p=1
x=h_unif(M,1,a,b)[0]
p=h_unif(M,1,a,b)[1]
u_Fe = -np.array(solid1d(y,x,M,p,f))
u_Ex_values = np.array([u_Ex_fun(val) for val in y])
fig , axes4 =plt.subplots(figsize=(10,10))
axes4.plot(y,abs(u_Ex_values-u_Fe)/abs(u_Ex_values),color='black')
axes4.set_xlabel("x")
axes4.set_ylabel("y")
axes4.set_title(r'Error ${|u_{FE}-u_{EX}|/|u_{EX}|}$')
#Graphs the uFE against uEX
fig , axes5 =plt.subplots(figsize=(10,10))
axes5.plot(y,u_Fe, color="red")
axes5.plot(y,u_Ex_values, color="blue")
axes5.set_xlabel("x")
axes5.set_ylabel("y")
axes5.set_title(r'Plots of $u_{FE}$ and $u_{EX}$')
axes5.legend([r'$u_{FE}$','$u_{EX}$'])

#Secondly I use the h fem with uniform mesh for M=1,p=6
```

```
x=h_unif(1,6,a,b)[0]
p=h_unif(1,6,a,b)[1]
u_Fe = -np.array(solid1d(y,x,1,p,f))
u_Ex_values = np.array([u_Ex_fun(val) for val in y])
fig , axes6 =plt.subplots(figsize=(10,10))
axes6.plot(y,abs(u_Ex_values-u_Fe)/abs(u_Ex_values),color='black')
axes6.set_xlabel("x")
axes6.set_ylabel("y")
axes6.set_title(r'Error ${|u_{FE}-u_{EX}|/|u_{EX}|}$')
#Graphs the uFE against uEX
fig , axes7 =plt.subplots(figsize=(10,10))
axes7.plot(y,u_Fe, color="red")
axes7.plot(y,u_Ex_values, color="blue")
axes7.set_xlabel("x")
axes7.set_ylabel("y")
axes7.set_title(r'Plots of $u_{FE}$ and $u_{EX}$')
axes7.legend([r'$u_{FE}$','$u_{EX}$'])

#Thirdly I use the p fem with geometric mesh for M=10,p=2 for all elements
x=p_geom(M,2,q,a,b)[0]
p=[]
for r in range(M):
    p.append(2)
u_Fe = -np.array(solid1d(y,x,M,p,f))
u_Ex_values = np.array([u_Ex_fun(val) for val in y])
fig , axes8 =plt.subplots(figsize=(10,10))
axes8.plot(y,abs(u_Ex_values-u_Fe)/abs(u_Ex_values),color='black')
axes8.set_xlabel("x")
axes8.set_ylabel("y")
axes8.set_title(r'Error ${|u_{FE}-u_{EX}|/|u_{EX}|}$')
#Graphs the uFE against uEX
fig , axes9 =plt.subplots(figsize=(10,10))
axes9.plot(y,u_Fe, color="red")
axes9.plot(y,u_Ex_values, color="blue")
axes9.set_xlabel("x")
axes9.set_ylabel("y")
axes9.set_title(r'Plots of $u_{FE}$ and $u_{EX}$')
axes9.legend([r'$u_{FE}$','$u_{EX}$'])

#Fourthly I use the p fem with geometric mesh for M=10,p=4 for all elements
```

```
x=p_geom(M,4,q,a,b)[0]
p=[]
for r in range(M):
    p.append(4)
u_Fe = -np.array(solid1d(y,x,M,p,f))
u_Ex_values = np.array([u_Ex_fun(val) for val in y])
fig , axes10 =plt.subplots(figsize=(10,10))
axes10.plot(y,abs(u_Ex_values-u_Fe)/abs(u_Ex_values),color='black')
axes10.set_xlabel("x")
axes10.set_ylabel("y")
axes10.set_title(r'Error ${|u_{FE}-u_{EX}|/|u_{EX}|}$')
#Graphs the uFE against uEX
fig , axes11 =plt.subplots(figsize=(10,10))
axes11.plot(y,u_Fe, color="red")
axes11.plot(y,u_Ex_values, color="blue")
axes11.set_xlabel("x")
axes11.set_ylabel("y")
axes11.set_title(r'Plots of $u_{FE}$ and $u_{EX}$')
axes11.legend([r'$u_{FE}$','$u_{EX}$'])
```

Figure 12: Python Code used. Figure 1 corresponds to In[1], Figure 2 corresponds to In[2], Figure 3 corresponds to In[3], Figure 4-11 corresponds to In[4].