

HOMEWORK 5

CEE 361-513: Introduction to Finite Element Methods

Due: Friday Nov. 10 @ Midnight

NB: Students taking CEE 513 must complete all problems. All other students will not be graded for problems marked with *, but are encouraged to attempt them anyhow.

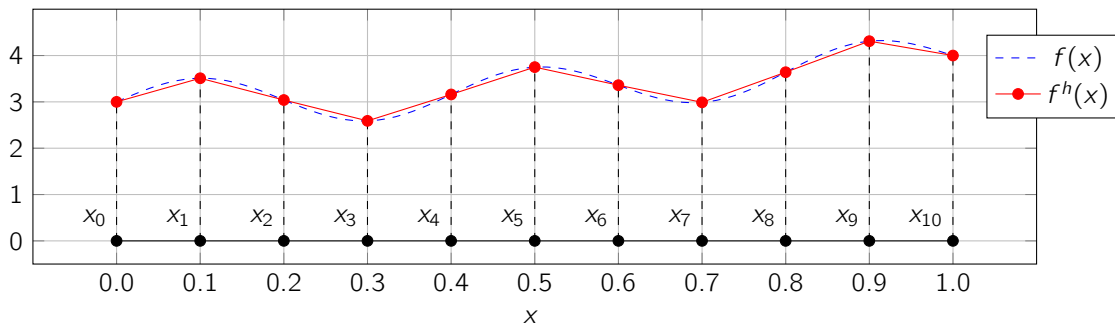
PROBLEM 1: Lagrange polynomials and approximation error

Preamble

A very important family of functions for finite element methods are called the *Lagrange* polynomials. These polynomials are very useful for *interpolating* functions.

As we saw in class, an interpolation $f^h(x)$ of a function $f(x)$ is a function that at certain points (sometimes called nodes or knots), let's denote them $x_i, i = 0 \dots n$, satisfies the condition $f^h(x_i) = f(x_i)$.

An example is the plot below where the function $f(x)$ is interpolated by $f^h(x)$. The function $f^h(x)$ consists of piecewise linear polynomials. Namely, between the nodes x_i and x_{i+1} the function $f^h(x)$ is a linear (in reality, an affine) function.



As we saw in class the function $f^h(x)$ can be written as the linear combination of *basis functions* $\phi_i(x)$ and nodal values $f_i = f(x_i)$ such that

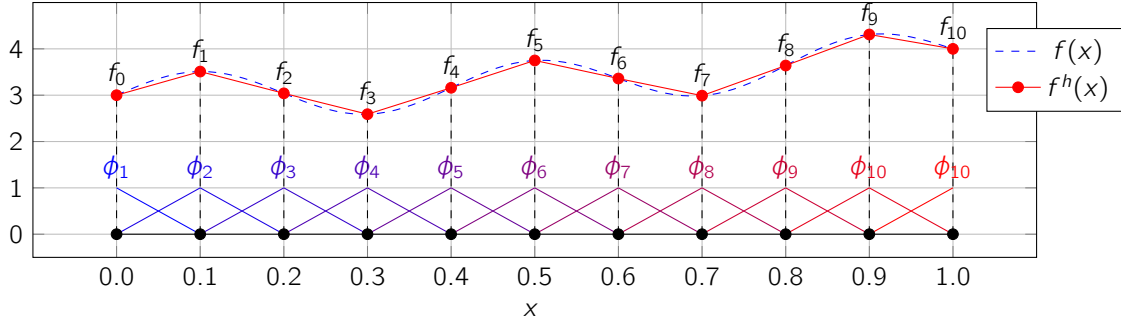
$$f^h(x) = \sum_{i=1}^n f_i \phi_i(x)$$

where the interpolation properties stem from the **really nice property** that $\phi_i(x_j) = \delta_{ij}$; namely, at the i^{th} node all basis functions $\phi_j(x)$ are zero with the only exception of $\phi_{j=i}(x)$. See the plot below for a graphical interpretation.

We remark again that the superscript h here refers to the average element size

$$h \approx 1/(n-1) \sum_{i=1}^{n-1} |x_{i+1} - x_i|.$$

If we *refine the mesh*, namely we increase the number of elements, the average mesh size will decrease and the number of basis functions will increase, hence we will have a better approximation to f (effectively as $h \downarrow$, $|f - f^h| \downarrow$).



Lagrange polynomials basis $\ell_i^p(x)$, $i = 0, \dots, p$ constructed on a set of $p+1$ nodes $(x_i, i = 0, \dots, p)$ represent a set of $p+1$ polynomials of order p (eg. if $p = 1$ we have 2 linear polynomials constructed onto 2 nodes, if $p = 2$ we have 3 quadratic polynomials constructed onto 3 nodes, etc) which satisfy $\ell_i(x_j) = \delta_{ij}$. The plots below show the Lagrange polynomial basis for $p = 1, 2, 3$. As you can see the basis are polynomials of order p and further they satisfy $\ell_i(x_j) = \delta_{ij}$.

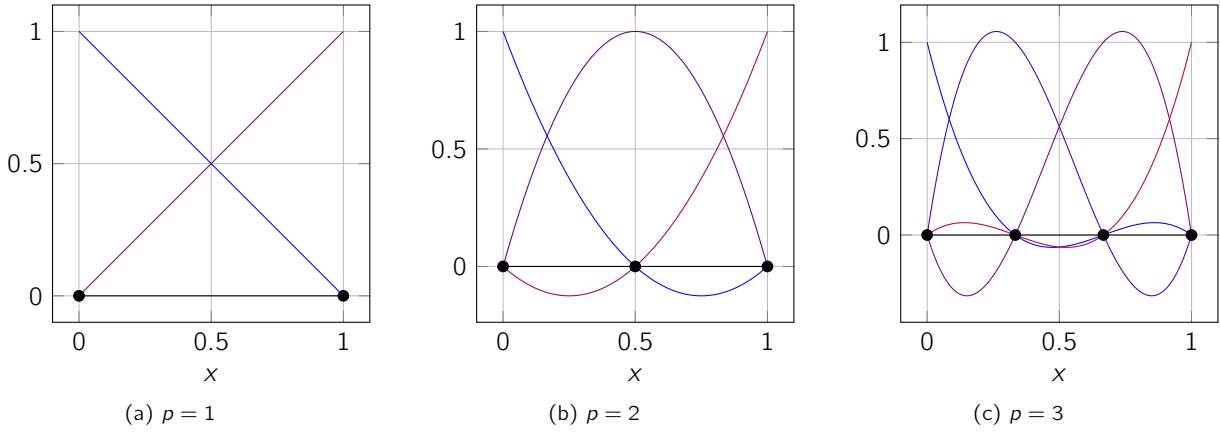


Figure 1: Showcasing the Lagrange polynomial basis

With the above Lagrange polynomial basis one can construct an approximation to any function as

$$f^p(x) = \sum_{i=0}^p f_i \ell_i^p(x).$$

Problem

With the above preamble we move on to the actual problem:

1. Implement a function that, given a set of nodes $x_i, i = 0 \dots p$, returns the *Lagrange* basis function for knot i . The expression for the i^{th} basis function for the p^{th} order polynomials is given by¹

$$\ell_i^p(x) = \prod_{\substack{0 \leq j \leq p \\ i \neq j}} \frac{x - x_j}{x_i - x_j}. \quad (1)$$

Use the following code as a starter code (you will have to fill in minor additions). Plot all the Lagrange basis for $p = 1, 2, 3, 4$ similarly to Figure 1 for the interval $[-1, 1]$ where the nodes are equally spaced as well as for unevenly (you can choose how unevenly) spaced nodes.

¹Note that often we omit the superscript p on ℓ ; namely we write $\ell_i \equiv \ell_i^p$

Lagrange Polynomials Starter Code

# @param[in] nodes the coordinates of the nodes	1
# @param[in] index the index of the basis function	2
# @param[in] order the order of the polynomial basis	3
# @param[in] x the coordinate where to evaluate the basis	4
# @return the value of the index-th basis function at point x	5
def lagrange_basis(nodes, index, order, x):	6
	7
# Check that we have the right number of nodes	8
assert len(nodes) == (order + 1)	9
	10
# Create the initial value of the function	11
ell = 1.	12
	13
# Loop over all nodes	14
for i in range(0,order+1):	15
	16
# If this node is the same as the	17
# support node of the basis function, skip it	18
if i == index:	19
continue	20
	21
# Otherwise perform the multiplication	22
ell *= (x - ???)/(??? - ???) # <- fill here!!!	23
	24
return ell	25

Solution :
The modified code is :

Lagrange Polynomials Starter Code Solution

```
# Otherwise perform the multiplication  
ell *= (x - nodes[i])/(nodes[index] - nodes[i]) # <- fill here!!!
```

1
2

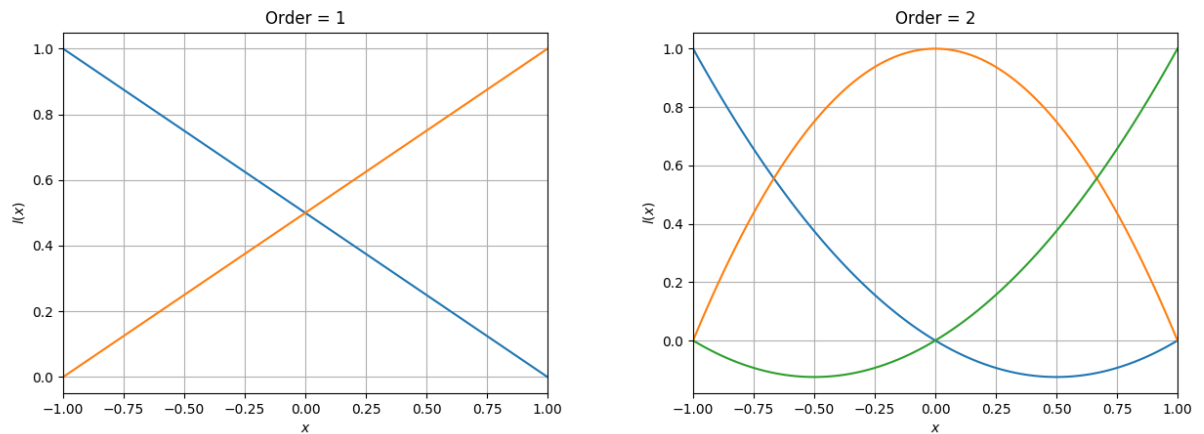


Figure 2: Lagrange polynomials of order 1 and 2 with even spacing

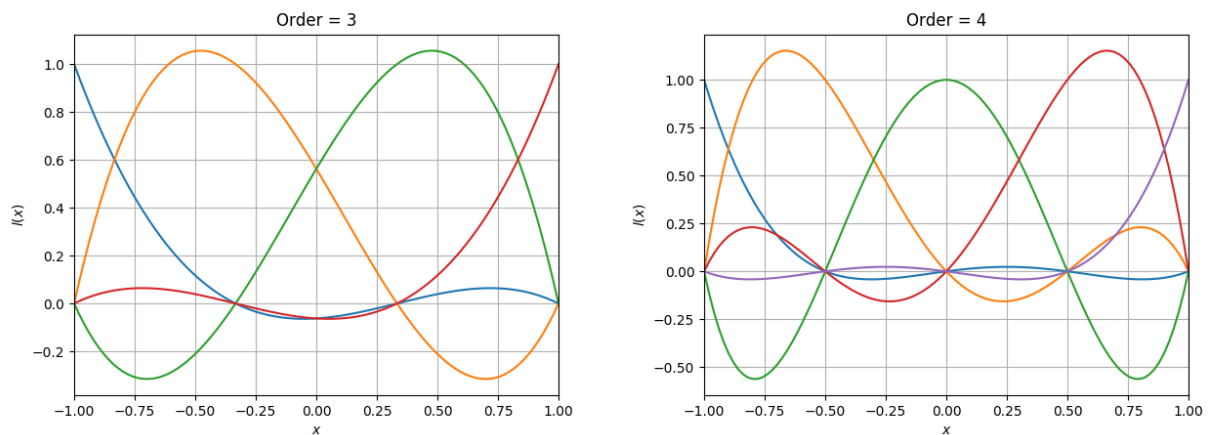


Figure 3: Lagrange polynomials of order 3 and 4 with even spacing

Solution Continued:

Here we have plotted the results for nodes being unevenly spaced.

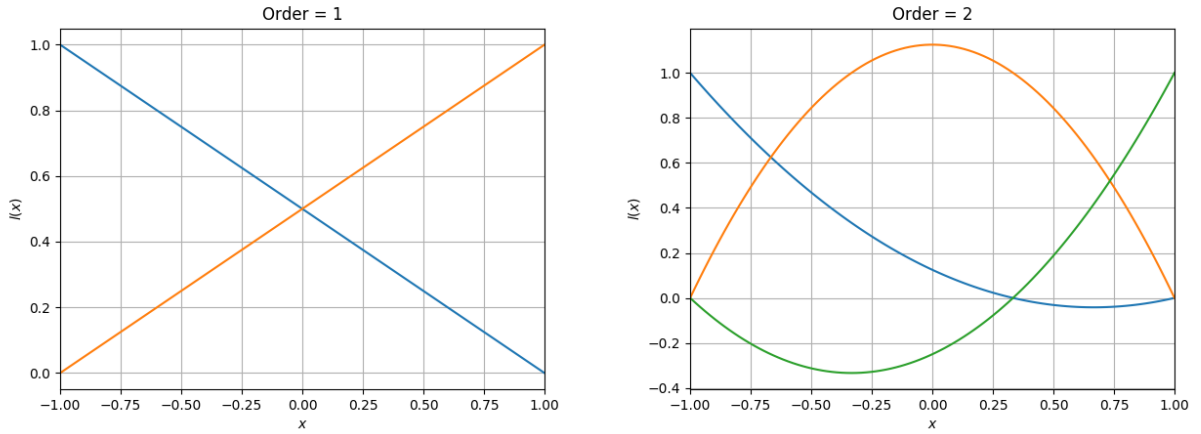


Figure 4: Lagrange polynomials of order 1 and 2 with uneven spacing

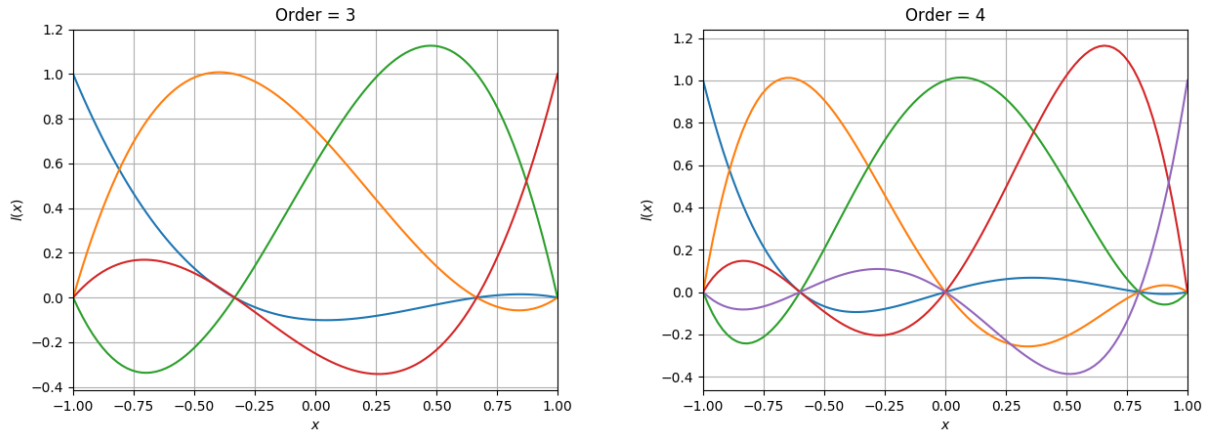


Figure 5: Lagrange polynomials of order 3 and 4 with uneven spacing

- The derivatives of the Lagrange polynomial basis are given by

$$\frac{d\ell_i^p(x)}{dx} = \ell_i^p(x) \sum_{\substack{0 \leq j \leq p \\ i \neq j}} \frac{1}{x - x_j}. \quad (2)$$

Use the following code as a starter code (you will have to fill in minor additions). Plot the derivatives of all the Lagrange basis for $p = 1, 2, 3, 4$ similarly to Figure 1.

Lagrange Polynomials Starter Code

# @param[in] nodes the coordinates of the nodes	1
# @param[in] index the index of the basis function	2
# @param[in] order the order of the polynomial basis	3
# @param[in] x the coordinate where to evaluate the basis	4
# @return the value of the derivative of the	5

#	index-th basis function at point x	6
def	d_lagrange_basis(nodes, index, order, x):	7
		8
	# Check that we have the right number of nodes	9
	assert len(nodes) == (order + 1)	10
		11
	# Create the initial value of the function	12
	d_ell = 0	13
		14
	# Loop over all nodes	15
	for i in range(0,order+1):	16
		17
	# If this node is the same as the	18
	# support node of the basis function, skip it	19
	if i == index:	20
	continue	21
		22
	# Otherwise perform the multiplication	23
	d_ell += 1./(x - ???) # <- fill here!!!	24
		25
	# Multiply by the corresponding Lagrange basis	26
	d_ell *= lagrange_basis(nodes, index, order, x)	27
		28
	return d_ell	29

Solution :

The modified code:

Lagrange Polynomials Starter Code Solution

```
# Otherwise perform the multiplication
d_ell += 1./(x - nodes[i]) # <- fill here!!!
```

1
2

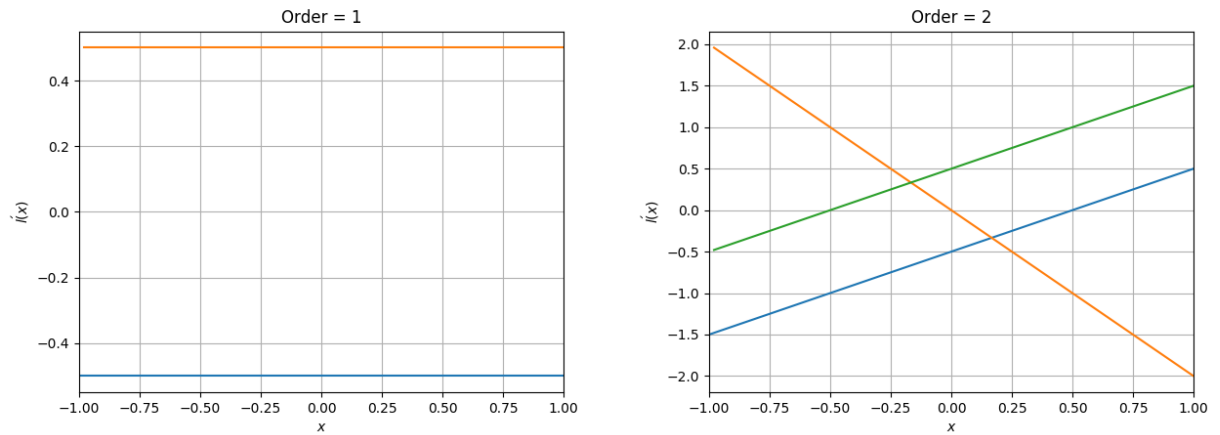


Figure 6: Derivative of Lagrange polynomials of order 1 and 2 with even spacing

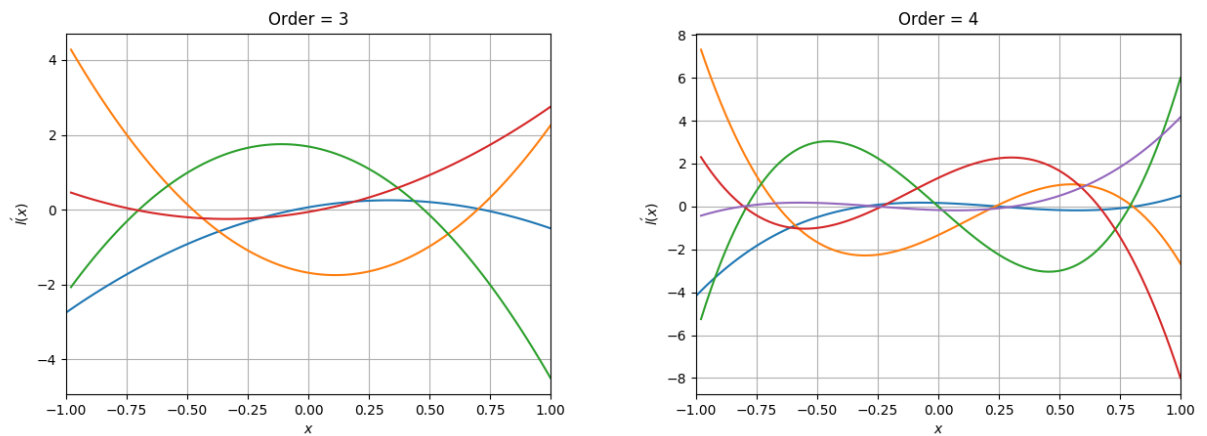


Figure 7: Derivative of Lagrange polynomials of order 3 and 4 with even spacing

3. Suppose now that you are given a function

$$f(x) = \frac{1}{2} \cos(2\pi x) + \exp(x) + x^3$$

and you want to approximate this function f with f^P where

$$f^P(x) = \sum_{i=0}^P f(x_i) \ell_i^P(x). \quad (3)$$

Save the two functions from part 1. and 2. in a file called `lagrange_polynomials.py`. Using the starter

code below implement the function f^p and plot for $p = 1, \dots, 10$ the function $f(x)$ and the function $f^p(x)$ over the interval $[-1, 1]$. Use as starter code the file `lagrange_interpolation.py`.

Lagrange Polynomials Starter Code

# @param[in] f the function to be interpolated	1
# @param[in] nodes the nodes of the interpolation	2
# @param[in] poly_order the polynomial order of the interp	3
# @param[in] x the point where to evaluate the interpolation	4
# @return the value of the interpolation	5
def lagrange_interpolation(f, nodes, poly_order, x):	6
	7
# Initialize the value of the interpolation	8
fp_val = 0	9
	10
# Sum up over all the basis	11
for i in range(0, poly_order+1):	12
	13
fp_val += ??? # <-- fill here	14
	15
return fp_val	16

Solution :

The modified code is :

Lagrange Polynomials Starter Code Solution

```
fp_val += f(nodes[i])*lagrange_basis(nodes, i, poly_order, x)
```

1

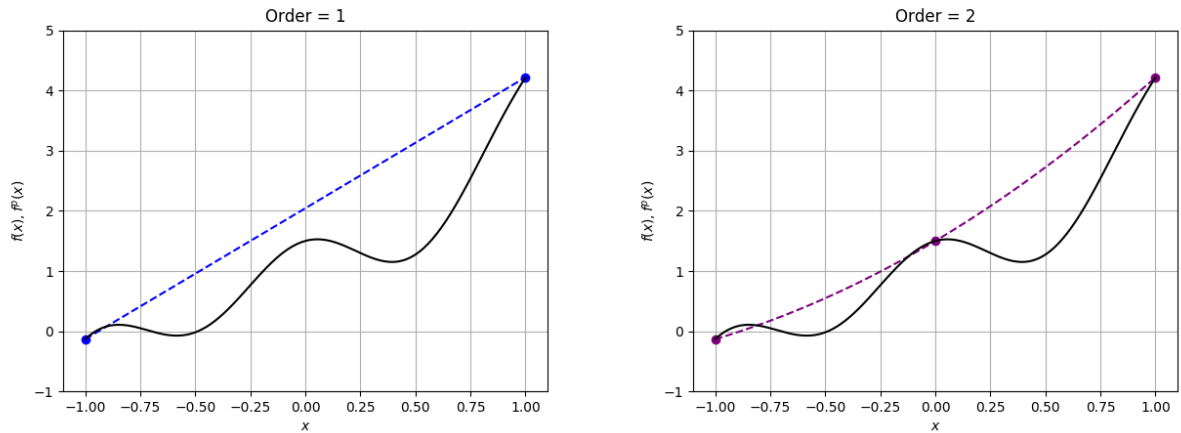


Figure 8: Function and Lagrange interpolation with polynomial orders 1 and 2

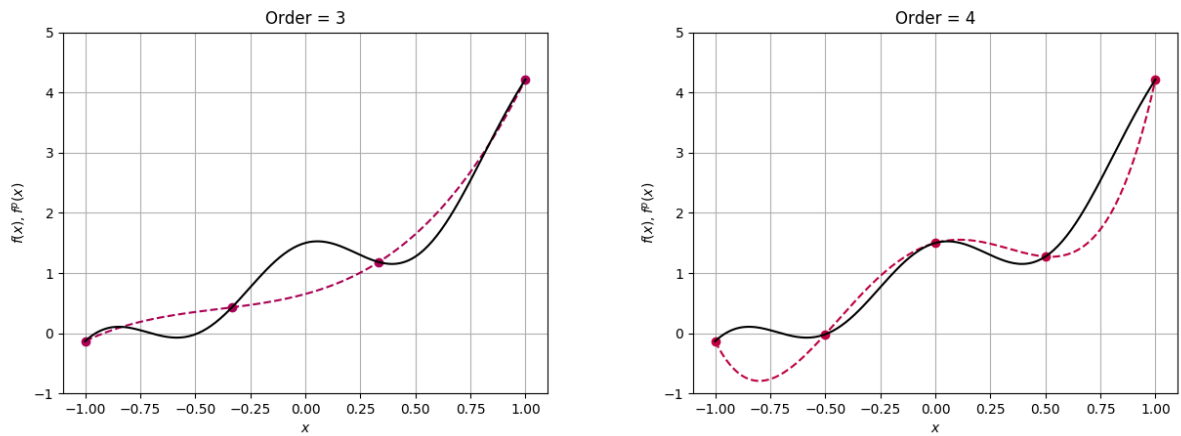


Figure 9: Function and Lagrange interpolation with polynomial orders 3 and 4

Solution Continued:

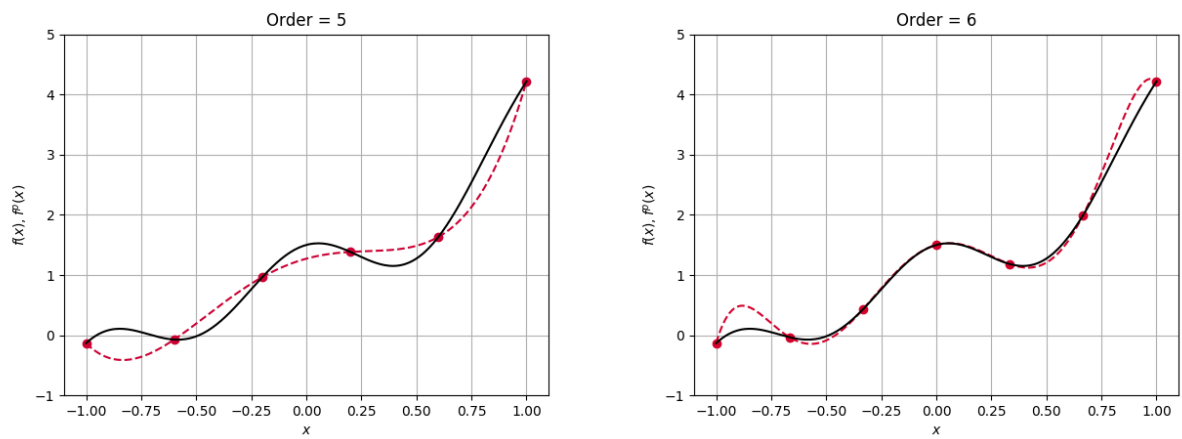


Figure 10: Function and Lagrange interpolation with polynomial orders 5 and 6

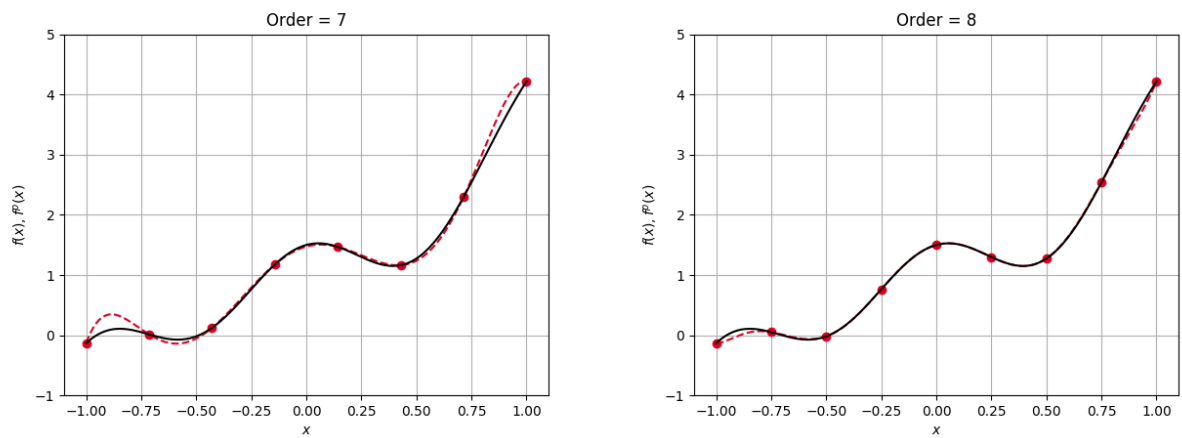


Figure 11: Function and Lagrange interpolation with polynomial orders 7 and 8

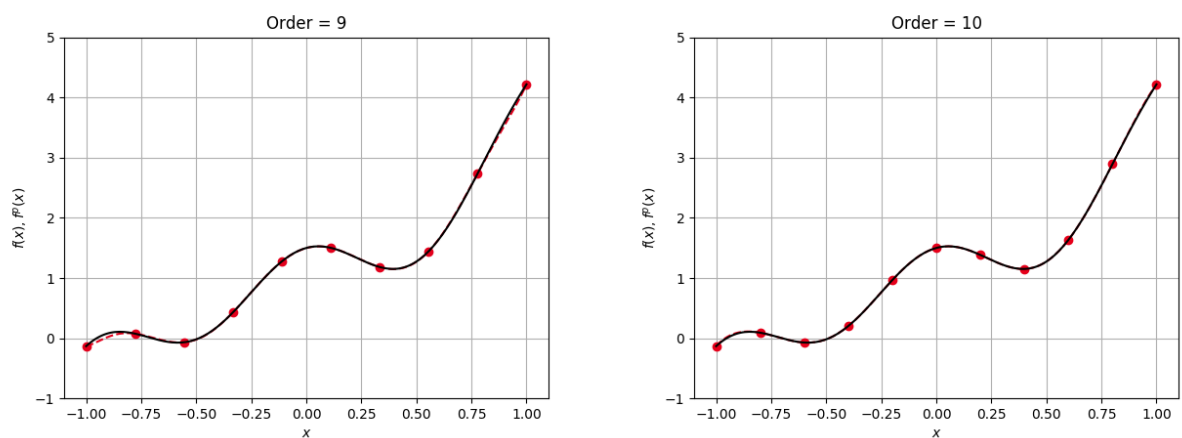


Figure 12: Function and Lagrange interpolation with polynomial orders 9 and 10

4. We would like to measure the error between $f^p(x)$ and $f(x)$ for a few values of p . To this extent we need to define a way to measure the error. To do so we introduce the L^2 -norm of a function $g(x)$ over an interval $[a, b]$ defined as

$$\|g\|_{L^2([a,b])} = \left[\int_a^b g^2 dx \right]^{1/2}.$$

For our one dimensional example, if we define the error as $e(x) = |f^p(x) - f(x)|$, we have that the $L^2([-1, 1])$ norm of the error is defined as

$$\|e\|_{L^2([-1,1])} = \left[\int_{-1}^1 e^2 dx \right]^{1/2}.$$

The above integral can be very hard to perform as $f^p(x)$ is a rather complicated expression and $f(x)$ could also be very challenging to integrate analytically.

To this extent we are going to adopt *Gaussian quadrature* (we will further talk about this in class). Gaussian quadrature is a way to approximate an integral by a weighted sum of function values at special points. The weights and the special sampling points are called quadrature weights w_i and quadrature points \hat{x}_i ; together $\{(w_i, \hat{x}_i)\}_{i=1}^n$ form a quadrature rule. More specifically we will perform the following approximation

$$\int_a^b g(x) dx \approx \sum_{i=1}^n f(\hat{x}_i) w_i.$$

where n is the number of quadrature points. Of course, both the quadrature weights and quadrature points depend on the domain of integration. Some quadrature rules are exact for certain order of polynomials. For now we will not get hung up on the accuracy of various quadrature rules as we will talk about it in class but I encourage you to read on the subject.

Luckily for us some of the more famous quadratures rules are already implemented in Python. We will be using Gauss-Legendre quadrature of order n that integrates exactly polynomials of order $p = 2 * n - 1$. For example, if we want to integrate a polynomial of order p we should choose a quadrature rule of order $n = \lceil (p + 1)/2 \rceil$.

In our case we are integrating a function $f(x)$ that is rather complicated (i.e. is not just a simple polynomial) thus we will use a very large quadrature rule to ensure the quadrature error does not pollutes our interpolation error $\|e\|_{L^2([-1,1])}$. To perform the integration you simply have to add the following lines in the right places in the code of part 3. Using the following code and the previous code plot on a log-log plot (in matplotlib `loglog(x, y)`) the error vs the number of basis functions (i.e. $p + 1$).

Computing the L^2 norm of the error

# Get the gauss quadrature rule	1
quadrature_order = max(poly_order + 1, 10)	2
gauss_points, gauss_weights = \	3
np.polynomial.legendre.leggauss(quadrature_order)	4
	5
# Compute the L2 norm of the error	6
e = f(gauss_points) - fp(gauss_points)	7
l2_err = np.sqrt(sum(gauss_weights*(pow(e ,2))))	8

Solution :

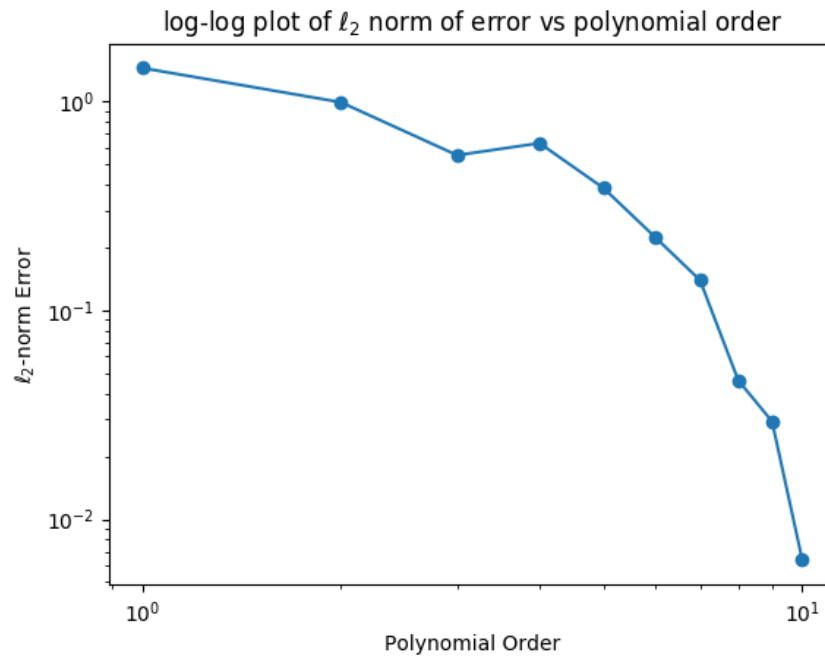


Figure 13: Log-Log plot of error vs order of polynomial

5. Look up what Weierstrass Approximation Theorem states and explain why the choice of (Lagrange) polynomials is a good choice to approximate functions.

Solution :

Weierstrass Approximation Theorem : For a continuous real-valued function f defined on the real-interval $[a, b]$, a polynomial p could be found such that for all $x \in [a, b]$ we have $|f(x) - p(x)| < \epsilon$ for any arbitrary $\epsilon > 0$.

This justifies our choice of Lagrange polynomials for approximation continuous functions.

PROBLEM 2: Constructing Element Arrays

Preamble

Consider the simple problem of solving for $u : [-1, 1] \rightarrow \mathbb{R}$ such that

$$\frac{d^2 u}{dx^2} = f \quad \forall x \in (-1, 1)$$

and

$$u(-1) = g_l, \quad u(1) = g_r.$$

The weak form becomes: find $u \in \mathcal{S}$, with \mathcal{S} being the set of trial functions, such that

$$a(u, v) = F(v) \quad \forall v \in \mathcal{V}$$

where \mathcal{V} is the set of test functions and

$$a(u, v) = \int_{-1}^1 \frac{du}{dx} \frac{dv}{dx} dx, \quad F(v) = - \int_{-1}^1 f v dx.$$

We then approximate the space of trial and test functions with piecewise polynomials as to arrive at the Galerkin approximation as we have done in class.

In this part of the homework we are going through an implementation of the finite element solution for the above problem. Note that a lot of the infrastructure (the different parts of the code) are reusable. Effectively the code consists of a few parts.

- i. A function for the calculation of the *local* element arrays (stiffness and source terms)
- ii. A local to global map for the degrees of freedom
- iii. A routine to perform the assembly as we did for matrix structural analysis

The above are implemented in the attached code named `finite_element_example.py` in three functions:

- i. `element_stiffness`
- ii. `local_to_global_map`
- iii. `assemble_global`

Effectively the ultimate goal of any finite element code is to assemble the “stiffness matrix” K and a source vector F such that the values of the degrees of freedoms (the u_i in $u^h = \sum_i u_i \phi_i$) can be found by solving $[K]\{U\} = \{F\}$. To do so we perform the following steps:

- i. We loop over all elements in the mesh
- ii. For each element we compute the local element arrays. To do so we
 - (a) Loop over all of the quadrature points inside the element
 - (b) At each quadrature point (ξ_g, w_g) we evaluate the integrands and add them to the local element arrays

$$k_{ij}^e = \frac{d\phi_i}{dx}(\xi_g) \frac{d\phi_j}{dx}(\xi_g) \left(\frac{d\hat{x}}{d\xi}(\xi_g) \right)^{-1} w_g \approx \int \frac{d\phi_i}{dx} \frac{d\phi_j}{dx} \left(\frac{d\hat{x}}{d\xi} \right)^{-1} d\xi$$

and

$$f_i^e = \phi_i(\xi_g) f(\hat{x}(\xi_g)) \frac{d\hat{x}}{d\xi}(\xi_g) w_g \approx \int \phi_i(\xi) f(\hat{x}(\xi)) \frac{d\hat{x}}{d\xi} d\xi$$

- iii. We assemble k^e and f^e in the global system matrix

Problem

With the above preamble let's move forward to the actual problem.

We would like to manufacture boundary conditions g_l , g_r and source terms f such that the solution of the above boundary problem is exactly

$$u^e(x) = \frac{1}{2} \cos(2\pi x) + \exp(x) + x^3.$$

To do so we simply set

$$g_l = u^e(-1), \quad g_r = u^e(1), \quad f(x) = \frac{d^2 u^e}{dx^2}.$$

The above is already implemented for you in the following lines using `sympy`

```
x = sp.symbols('x')
ue = 1./2*sp.cos( 2.*sp.pi*x ) + sp.exp(x) + x**3
f = sp.lambdify(x, sp.diff(ue, x, 2))
ue = sp.lambdify(x, ue)
```

1
2
3
4

1. Run the code provided in `finite_element_example.py` and plot the solution for 2^4 elements.

Solution :

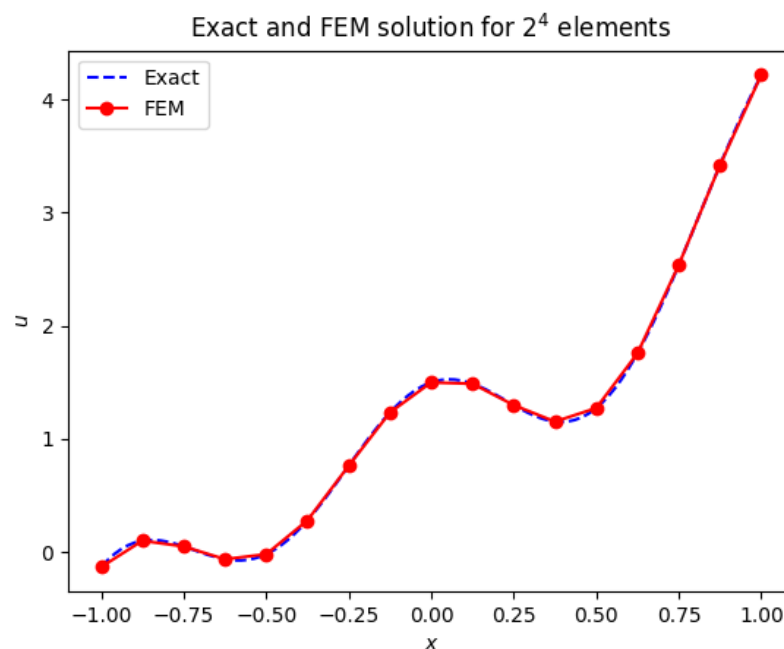


Figure 14: Plot of exact and FEM solution

2. Compute the L^2 norm of the error using the provided function. Namely you will have to write somewhere after the computation of the solution

```
l2_err = compute_l2_norm_error(coordinates_array, connectivity_array, \
                                poly_order, u, ue )
```

1
2

Solution :

The ℓ_2 -norm error for 2^4 elements is 0.0289

3. Run the above code for $2^1, 2^4, \dots, 2^{10}$ elements and for each solution compute the error. Plot the computed errors as a function of the mesh size $h = 2/2^1, \dots, 2/2^{10}$ on a log-log plot (use matplotlib loglog plot). Here you will have to write a for loop that wraps the relevant parts of the code.

Solution :

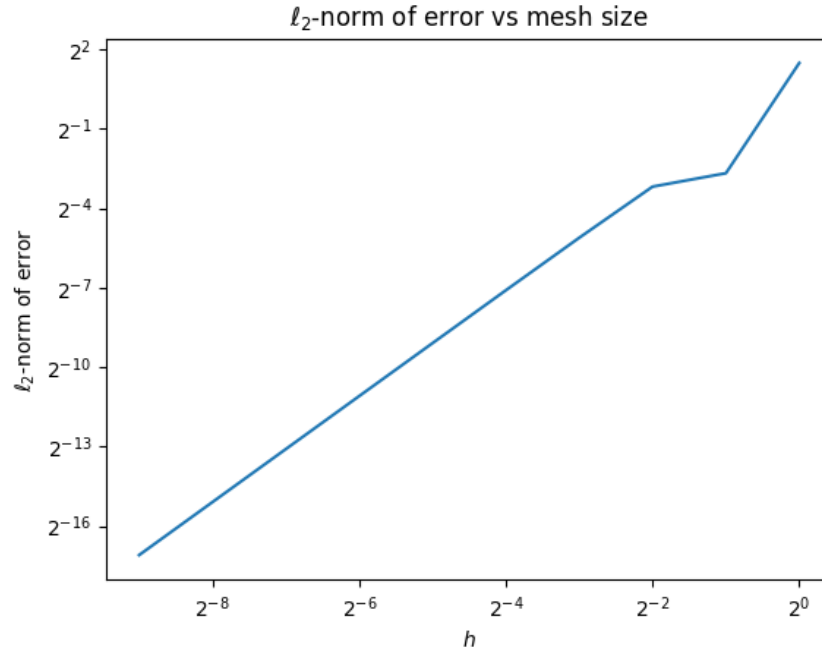


Figure 15: ℓ_2 norm of error vs mesh size

4. What is the rate of convergence of the solution? Namely, we see that for a mesh size h the L^2 norm of the error is

$$\|e^h\|_{L^2} = \|u^h - u^e\| \approx \alpha h^k$$

where α is a constant independent of h . Our goal is to determine k .

To do so choose a fixed h (for example $h = 2/2^7$) then look at

$$\frac{\|e^h\|_{L^2}}{\|e^{h/2}\|_{L^2}} \approx \frac{\alpha h^k}{\alpha (h/2)^k} = 2^k \Rightarrow k = \log_2 \frac{\|e^h\|_{L^2}}{\|e^{h/2}\|_{L^2}}.$$

Solution :

The rate of convergence for piecewise linear basis functions is 2.

5. So far we have used piecewise linear basis functions to approximate the solution. Since in the previous problem we also implemented higher order polynomials it would be nice to use them. To do so simply change the parameter `poly_order`. Effectively the basis functions will become piecewise p order polynomials over each element, as shown for $p = 2$ in the figure below. For $p = 2, 3$ repeat the above steps. Namely, on a single log-log plot, plot the L^2 norm of the error vs the mesh size for $p = 1, 2, 3$ and compute the rates of convergence.

Solution :

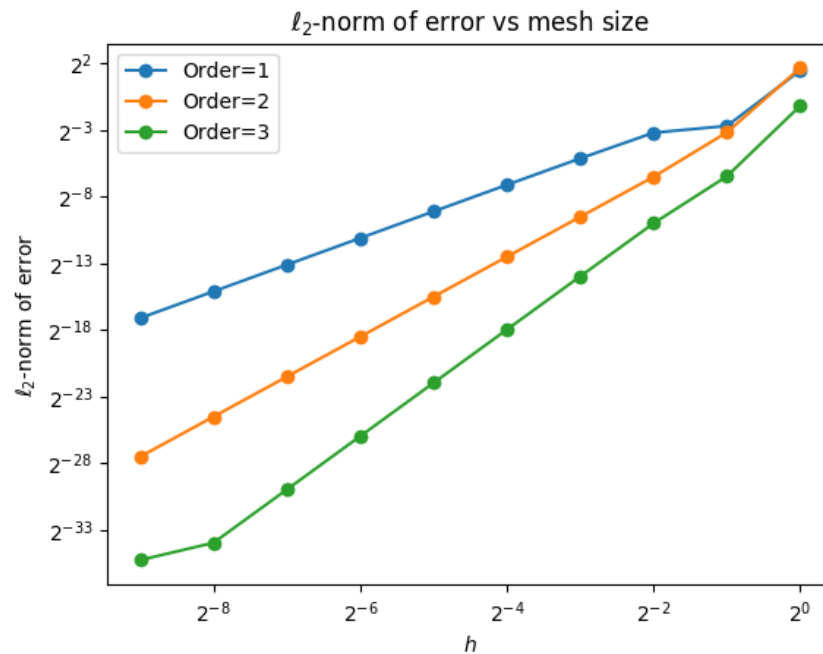


Figure 16: ℓ_2 norm of error vs mesh size

The rate of convergences are:

Order of polynomial	Rate of convergence
1	2
2	3
3	4

Table 1: Order of polynomial and rate of convergence

6. If similarly to the mid-term exam we wanted to solve instead

$$\frac{d^2 u}{dx^2} + ku = f \quad \forall x \in (0, 1)$$

keeping the boundary conditions the same. How would you implement this? Namely

(a) State which function you would modify

Solution :

The function that would be modified is the element_stiffness.

(b) Write the pseudo-code to illustrate how you would implement it (you don't have to actually implement it).

Solution :

Will be posted after HW 6

7. * Suppose now that you were interested in measuring the error in your derivatives. To do so you can use

another measure of error which is the H^1 norm of the error defined as

$$\|e\|_{H^1([-1,1])} = \left[\int_{-1}^1 e^2 dx + \int_{-1}^1 \left(\frac{de}{dx} \right)^2 dx \right]^{1/2}.$$

Implement a function to compute the above, using `compute_l2_norm_error` as a starting point. On a single log-log plot for $p = 1, 2, 3$ show the error vs the mesh size and compute (and report) the convergence rates of the H^1 norm of the error.

Solution :

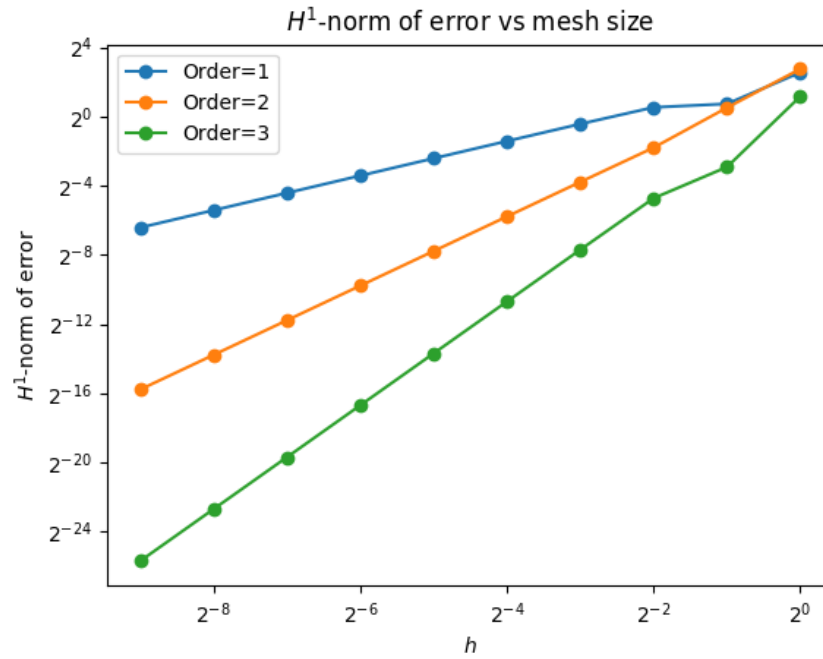


Figure 17: H^1 norm of error vs mesh size

Order of polynomial	Rate of convergence
1	1
2	2
3	3

Table 2: Order of polynomial and rate of convergence