

Remarks:

- In general \mathcal{S} and \mathcal{S}_m are not linear spaces because of the non-homogeneous boundary condition. However, we can try to homogenize the boundary condition and reduce the original problem to a BVP with homogeneous boundary condition. Then in the weak form of this new BVP, the solution set \mathcal{S} can be chosen as a linear space.
- The dimensions of \mathcal{S}_m and \mathcal{T}_n do not have to be the same. In this situation, the number of equations and unknowns in the above linear system about the coefficients of u_{pg} are not the same, and special numerical methods have to be used to solve this linear system. We therefore usually use \mathcal{S}_m and \mathcal{T}_n such that $m = n$ in a Petrov-Galerkin method.
- We should choose $\phi_1(x), \phi_2(x), \dots, \phi_n(x)$ and $\psi_1(x), \psi_2(x), \dots, \psi_n(x)$ such that they are linearly independent.
- For a symmetric BVP, the matrix in the Galerkin method is symmetric, but the matrix in the PG method is not necessarily symmetric.

A generic framework to solve a BVP by the Galerkin method: Given a BVP such as

$$-(au')' + cu = f, \quad x \in (0, 1) \quad (1)$$

$$a(0)u'(0) = g_0 \quad (2)$$

$$u(1) = g_1 \quad (3)$$

(A). Derive **A Weak Form** for this BVP: find $u \in \mathcal{S}$ such that

$$\int_0^1 av' u' dx + \int_0^1 cvudx = \int_0^1 vfdx - v(0)g_0, \quad \forall v \in \mathcal{T} \quad (5)$$

$$\text{where } \mathcal{T} = \{w \mid w \in H^1(0, 1), w(1) = 0\} \quad (7)$$

$$\mathcal{S} = \{w \mid w \in H^1(0, 1), w(1) = g_1\} \quad (8)$$

(B). Choose/construct $\mathcal{T}_n = \text{span}\{\psi_1(x), \psi_2(x), \dots, \psi_n(x)\} \subseteq \mathcal{T}$, $G(x)$ and use the weak form to set up the Galerkin equation: $(K + M)\vec{u} = \vec{f} + \vec{B}$ which can produce a Galerkin solution

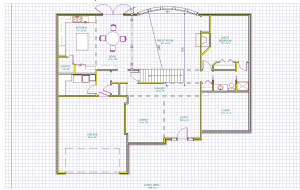
$$u_g(x) = \sum_{j=1}^n u_j \psi_j(x) + G(x) \approx u(x) = ?$$

A few questions: how to choose/create $\psi_i(x)$, $i = 1, 2, \dots, n$? How to find $G(x)$? How to form the matrices K and M ? How to form the vectors \vec{f} and \vec{B} ?

The computations for $u_g(x)$ are essentially determined by our choice/creation of the functions $\psi_i(x)$, $i = 1, 2, \dots, n$.

A VERY sketchy framework for finite element computations

Construction:



Finite Element Methods:

Mesh, finite element functions/space, quadrature rules, etc

Use a weak form and a FE space to set up a Finite Element Scheme/Equation

Solve the FE equation to obtain a finite element solution

Chapters 2, Finite Element Methods for 1D Problems

Examples presented in our previous lectures and exercises indicate the Galerkin method can be used as a computational tool to solve BVPs.

However, a few remaining issues including those listed below need to be resolved before we can adopt the Galerkin method as a general computational method for solving BVPs.

- How to choose the parameter m and n ?
- How to choose/design the functions in test function space?
- How to choose/design the functions in the trial function set?
- It is desirable to design a Galerkin method that has “finite” features suitable for implementations on computers.

Finite element functions can address these issues systematically.

We note that functions in the test function space and the trial function set do not act individually. Instead, they work collectively for a Galerkin method. Similarly, finite element functions are usually developed collectively for the test and trial functions space/set in the following framework:

- Partition the solution domain Ω into finitely many sub-domains. This partition is often called a **mesh** of the solution domain and the sub-domains are called **elements**. For easy implementation, these sub-domains usually are simple in geometry, such as line segments for 1D problems and polygons like triangles and rectangles for 2D problems.
- The basis functions in the test and trial function space/set are defined as piecewise polynomials according to the mesh and the specific continuity determined by the problem or the method for solving this problem.

Remarks:

- Each polynomial can be describe by a finitely number of coefficient, and computations involving polynomials, either algebraic computations or calculus, are usually easy even for computers.
- However, piecewise polynomials are not polynomials. One of the main issues is that they do not have enough global smoothness and this leads to challenges in the theoretical analysis for finite element methods.

2.1, The mesh for a 1D domain: Consider a 1D domain $\Omega = (x_L, x_R) \subset \mathbb{R}$

Three essential components for a mesh for a domain $\Omega = (x_L, x_R)$:

- Choose an integer n and introduce $n + 1$ points/nodes $x_i, i = 1, 2, \dots, n + 1$ in Ω .
- Use all of these nodes to partition Ω into 1D simplexes:

$$K_p = [x_i, x_j], \quad 1 \leq i, j \leq n, \quad x_i < x_j, \quad p = 1, 2, \dots$$

called **elements** of this mesh. These simplexes need to be such that

$$\cup_p K_p = \overline{\Omega} = [x_L, x_R], \quad K_p \cap K_q = \emptyset \quad \text{or} \quad \exists i, 1 \leq i \leq n, \quad K_p \cap K_q = \{x_i\}$$

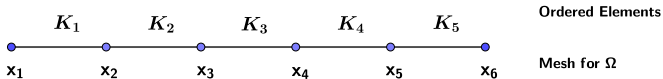
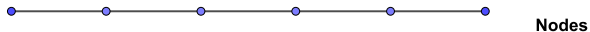
Not allowed: $K_1 = [x_1, x_3], K_2 = [x_2, x_3], K_3 = [x_3, x_4], K_4 = [x_4, x_5], K_5 = [x_5, x_6]$



Also, x_i and x_{i+1} in element $K_p = [x_i, x_j]$ are called the vertices of the element/simplex K_p .

- When $K_p \cap K_q = \{x_i\}$, then x_i is a common vertex of elements K_p and K_q and it is also called an **edge** of the mesh.

Due to the 1D geometry of $\Omega = (x_l, x_r)$, a mesh with the above features can be constructed as follows:



Mesh for Ω

A summary:

$$\begin{aligned}\text{Nodes:} \quad & x_l = x_1 < x_2 < x_3 < \cdots < x_n < x_{n+1} = x_r \\ \text{Elements:} \quad & K_p = [x_p, x_{p+1}], \quad p = 1, 2, \cdots, n \\ \text{Mesh size:} \quad & h = \max_{1 \leq p \leq n} |K_p| = \max_{1 \leq p \leq n} |x_{p+1} - x_p|\end{aligned}$$

Three sets to describe a mesh:

- The set of **nodes** $\mathcal{N}_h = \{x_i\}_{i=1}^{n+1}$.
- The set of **elements** $\mathcal{T}_h = \{K_i\}_{i=1}^n$ which are formed with nodes.
- The set of **edges** \mathcal{E}_h which are intersections of elements

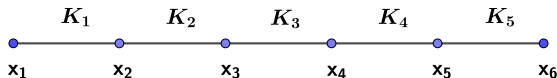
From now on, we often simply call $\mathcal{T}_h = \{K_i\}_{i=1}^n$ a mesh of for a domain Ω .

A mesh \mathcal{T}_h in a computer can be described by two essential matrices/arrays:

- The **nodal coordinate matrix** p . It is of the size $1 \times n$ containing the coordinates of the nodes $x_i, i = 1, 2, \cdots, n$.
- The **connectivity matrix** t for the mesh nodes. We index all the nodes. Put these indices in an integer array t of the size:
(num of vertices forming each element) \times (num. of elements)
such that its p -th column contains the indices of the vertices forming element K_p with a chosen order.

Matrix/Array t has $|\mathcal{T}_h|$ columns it tells us which mesh nodes are used to form which element.

Recall:



We can implement a 1D uniform mesh generator in Matlab as follows:

```
function mesh = mesh_generator_1D(domain, n)
x_l = domain(1); x_r = domain(2);
p = x_l:(x_r - x_l)/n:x_r;

t = zeros(2,n); t(:,1) = [1;2];
for i=2:n
    t(:,i) = t(:,i-1) + 1;
end
mesh = struct('p', p, 't', t);
```

- `domain = [x_l, x_r]` defines the solution domain $\Omega = (x_l, x_r)$.
- `n` is the integer specifying the $n + 1$ uniformly distributed nodes generated in Ω .
- `mesh` is a [structure array](#) with two basic [fields](#) `p` and `t`. The matrix `p` can be extracted by `mesh.p` and `t` can be extracted by `mesh.t`. The less important field is for the set of edge which can be added later or by extending this program.

Example: The 1D mesh generator can be used to generate a mesh as follows:

```
domain = [0, 1]; n = 5;  
mesh = mesh_generator_1D(domain, n);
```

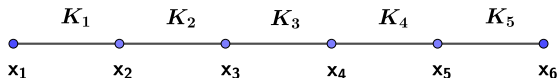
which contains the nodal coordinate matrix/array `mesh.p`:

0	0.2000	0.4000	0.6000	0.8000	1.0000
---	--------	--------	--------	--------	--------

and the connectivity matrix/array `mesh.t`:

1	2	3	4	5
2	3	4	5	6

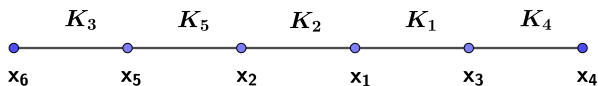
These arrays define a mesh in $\Omega = (0, 1)$ with the following geometry:



These two arrays contain rich information about the mesh. For example, the total number of nodes is `length(mesh.p)`, the total number of elements is `length(mesh.t)`. The coordinate of the 4-th node x_4 in the mesh can be extracted by `mesh.p(4)`. The indices for the vertices/nodes forming the 4-th elements are given by `mesh.t(:, 4)`. We can further find the coordinates of the nodes for the 4-th element by `mesh.p(mesh.t(:, 4))`.

Remark: The nodes and elements have to be indexed, but how to order them is not important.

For example, if we order the nodes and elements in the mesh as follows:



Then, `mehs.p` should be:

```
0.6000 0.4000 0.8000 1.0000 0.2000 0
```

and `mehs.t` should be:

```
1 2 6 3 5
```

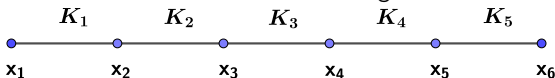
```
3 1 5 4 2
```

Even though these matrices are different from those in the previous example, their usages in finite element computations will be the same.

However, it is desirable for the node and connectivity matrices to have a simpler and more natural order, **IF POSSIBLE**.

Remark: The structure `mesh` can be easily enhance later to include other useful information about a mesh such as the edges, points, etc. For example, recall that an edge in a mesh is a geometric object where two elements meet.

For a 1D mesh , its edges are the nodes, and each edge has two neighbor elements, one is on the left and other is on the right:



Instead of programming edges explicitly, we can introduce an [edge matrix/array](#) e to describe the edges in this mesh together with the nodal coordinate matrix p :

```
1 2 3 4 5 6
0 1 2 3 4 5
1 2 3 4 5 0
```

In this matrix/array, the i -th column contains information for the i -th nodes: the first number is the index of the point for i -th edge, the second and the third numbers are the indices of the elements on the left and right of the i -th edge, respectively. This edge matrix can be used to retrieve information associated with a particular edge.

Here we order the edges points from the left to the right.

We can revise the `mesh_generator_1D.m` to include a new field for the edges as follows:

```
function mesh = mesh_generator_1D(domain, n)
x_l = domain(1); x_r = domain(2);
p = x_l:(x_r - x_l)/n:x_r;

t = zeros(2,n); t(:,1) = [1;2];
for i=2:n
    t(:,i) = t(:,i-1) + 1;
end
e = zeros(3,n+1); e(1,:) = 1:n+1; e(2,:) = 0:n; e(3,:) = [1:n,0];
mesh = struct('p', p, 't', t, 'e', e);
```

Example: For the mesh generated by `mesh = mesh_generator_1D([0,1], 5)`, `mesh.e(:, 4)` produces the following array:

```
4
3
4
```

which states that this edge is formed by the 4-th node x_4 , the element on the left is K_3 and the element on the right is K_4 .

There are two ways to extend a structure with new fields.

- Modify the Matlab function such that it can produce all the new fields together with the original ones.
- When the new fields needs extra inputs, then the above approach requires to change the interface of the Matlab function which might further require change other related codes. To avoid this situation, we can develop a new function for these new fields and extend the structure by Matlab's `setfield` command.

Example: Assume we have produced the structure `mesh` but generate the array `e` for the edges by another program. Then we can insert `e` into `mesh` by the Matlab command:

```
mesh = setfield(mesh, 'E', E);
```

2.2, 1D finite element spaces:

One key feature of finite element methods is literally “finite”, i.e., all the computations are carried out through finite procedures over all elements in a mesh, element by element repeatedly, and finite element functions over the whole solution domain Ω are rarely programmed or used directly except for mathematical descriptions and analysis of the involved computation procedures.

We now discuss finite element functions for solving differential equations, i.e., we discuss the construction of the trial and test function spaces consisting of finite element functions.

In particular, we consider a finite element function of degree p which is a piecewise polynomial defined on a mesh plus other features required by the weak form used in a finite element method for solving a particular differential equation.

Since the restriction of a standard finite element function of degree p to each element of a mesh is a polynomial of degree p , we will start from the local finite element space defined on each element.

2.2.1, Local p -th degree Lagrange shape functions on each element: Consider a typical element $K = [x_i, x_{i+1}]$ in a mesh \mathcal{T}_h . First, we introduce $p + 1$ equally spaced **local nodes** in K as follows

$$x_i = t_{K,1} < t_{K,2} < \cdots < t_{K,p} < t_{K,p+1} = x_{i+1}$$

Then, for each local node $t_{K,j}$, we introduce a polynomial called a **shape function** as follows

$$L_{K,j}^p(x) = \prod_{k=1, k \neq j}^{p+1} \frac{x - t_{K,k}}{t_{K,j} - t_{K,k}}, \quad j = 1, 2, \dots, p, p+1 \quad (9)$$

- $L_{K,j}^p(x)$, $1 \leq j \leq p+1$ are called Lagrange cardinal functions which are polynomials of degree exactly p determined by the local nodes t_1, t_2, \dots, t_{p+1} in the element K .
-

$$L_{K,j}^p(t_{K,i}) = \delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

- Let $w(x) = \prod_{k=1}^{p+1} (x - t_{K,k})$, then

$$L_{K,j}^p(x) = \frac{w(x)}{(x - t_{K,j})w'(t_{K,j})}, \quad j = 1, 2, \dots, p+1$$

Lagrange shape functions on a reference element: Let $\hat{K} = [-1, 1]$. Then element $K = [x_i, x_{i+1}]$ in a mesh \mathcal{T}_h is homeomorphic to the reference element $\hat{K} = [-1, 1]$ by the following affine mapping:

$$t : K \rightarrow \hat{K}, \quad t(x) = \frac{2}{x_{i+1} - x_i}(x - x_i) - 1, \quad \forall x \in K \quad (10)$$

Let $t_1 = -1 < t_2 < \dots < t_p < t_{p+1} = 1$ be equally spaced nodes in $\hat{K} = [-1, 1]$ and let

$$\hat{L}_j^p(t) = \prod_{k=1, k \neq j}^{p+1} \frac{t - t_k}{t_j - t_k}, \quad j = 1, 2, \dots, p, p+1 \quad (11)$$

Then, we have

$$L_{K,j}^p(x) = \hat{L}_j^p(t(x)) = \hat{L}_j^p\left(\frac{2}{x_{i+1} - x_i}(x - x_i) - 1\right), \quad j = 1, 2, \dots, p, p+1 \quad (12)$$

and

$$\frac{d^r L_{K,j}^p(x)}{dx^r} = \left(\frac{2}{x_{i+1} - x_i}\right)^r \frac{d^r \hat{L}_j^p(t(x))}{dt^r}, \quad j = 1, 2, \dots, p, p+1 \quad (13)$$

Implementation of shape functions on the reference element:

First, we can use a symbolic software such as Mathematica to prepare formulas for the shape functions on the reference element:

```
ClearAll
```

```
(* Ld lists the BI-th p-th degree Lagrange poly and its derivatives *)
```

```
p = 2;
```

```
(*degree*)
```

```
BI = 1; (*base index, BI<=p+1*)
```

```
nodes = Table[-1 + (i - 1)*(2/p), {i, 1, p + 1}];
```

```
L[t_] = Product[(t - nodes[[k]])/(nodes[[BI]] - nodes[[k]]), {k, 1,  
    BI - 1}]*
```

```
    Product[(t - nodes[[k]])/(nodes[[BI]] - nodes[[k]]), {k, BI + 1,  
    p + 1}];
```

```
"The BI-th shape function and its derivatives"
```

```
Ld = Simplify[Expand[Table[D[L[t], {t, i}], {i, 0, p}]]]
```

This will produce the 1st shape function of degree 2 and its derivatives on the reference element:

```
{1/2 (-1+t)t, -(1/2)+t, 1}
```

An implementation for the 1D p -th degree shape function on the reference element:

```
function f = shape_fun_1D_Lagrange_ref(t, shape_index, d_index, degree)

if degree == 0
    if shape_index == 1
        if d_index == 0
            f = ones(size(t));
        elseif d_index > 0
            f = zeros(size(t));
        end
    else
        disp(['shape_index = ', int2str(shape_index)])
        disp('This program is not for this value of shape_index')
        disp('Terminate this program by Ctrl+C')
        pause
    end
end
```

Recall function interface:

```
function f = shape_fun_1D_Lagrange_ref(t, shape_index, d_index, degree)
```

Continued:

```
elseif degree == 1
    if shape_index == 1
        if d_index == 0
            f = (1-t)/2;
        elseif d_index == 1
            f = (-1/2)*ones(size(t));
        elseif d_index > 1
            f = zeros(size(t));
        end
    elseif shape_index == 2
        if d_index == 0
            f = (1+t)/2;
        elseif d_index == 1
            f = (1/2)*ones(size(t));
        elseif d_index > 1
            f = zeros(size(t));
        end
    else
        disp(['shape_index = ', int2str(shape_index)])
        disp('This program is not for this value of shape_index')
        disp('Terminate this program by Ctrl+C')
        pause
    end
end
```

Recall function interface:

```
function f = shape_fun_1D_Lagrange_ref(t, shape_index, d_index, degree)
```

Continued:

```
elseif degree == 2
    if shape_index == 1
        if d_index == 0
            f = -(1/2)*(1-t).*t;
        elseif d_index == 1
            f = -1/2+t;
        elseif d_index == 2
            f = ones(size(t));
        elseif d_index > 2
            f = zeros(size(t));
        end
    elseif shape_index == 2
        if d_index == 0
            f = 1-t.^2;
        elseif d_index == 1
            f = -2*t;
        elseif d_index == 2
            f = -2*ones(size(t));
        elseif d_index > 2
            f = zeros(size(t));
        end
    end

elseif shape_index == 3
    if d_index == 0
        f = (1/2)*t.*(1+t);
    elseif d_index == 1
        f = 1/2 + t;
    elseif d_index == 2
        f = ones(size(t));
    elseif d_index > 2
        f = zeros(size(t));
    end
else
    disp(['shape_index = ', int2str(shape_index)])
    disp('This program is not for this value of shape_index')
    disp('Terminate this program by Ctrl+C')
    pause
end

elseif degree > 3
    disp(['degree = ', int2str(degree)])
    disp('This program is not for this value of degree')
    disp('Terminate this program by Ctrl+C')
    pause
end

end
```

An implementation of the 1D local shape functions: Since both the finite element functions and their derivatives are used to solve partial differential equations, we should implement the local shape function such that it can generate both the function and its derivative values according to the demand.

For

$$\frac{d^r L_{K,j}^p(x)}{dx^r} = \left(\frac{2}{x_{i+1} - x_i} \right)^r \frac{d^r \hat{L}_j^p(t(x))}{dt^r}, \quad j = 1, 2, \dots, p, p+1 \quad (13)$$

we can implement it with a Matlab function in the following format:

```
function f = shape_fun_1D_Lagrange(x, elem, degree, ...
                                shape_index, d_index)
t = (2/(elem(2)-elem(1)))*(x-elem(1)) - 1;
f = ((2/(elem(2)-elem(1)))^d_index)*shape_fun_1D_Lagrange_ref(t, ...
                        shape_index, d_index, degree);
```

- `x` is a real array where we would like to evaluate the shape function or its derivatives.
- `elem` provides the coordinates for the vertices of the element K . Note `elem` for element K_i in a mesh is `elem = mesh.p(mesh.t(:,i))`.
- `shape_index` specifies which local shape function is evaluated.
- `d_index` specifies which derivative of the shape function is evaluated.

We can use this shape function as follows

```
elem = [0, 1]; degree = 1; shape_index = 1; d_index = 0;  
x = elem(1):0.001:elem(2);  
y1 = shape_fun_1D_Lagrange(x, elem, degree, shape_index, d_index);  
plot(x,y1)
```

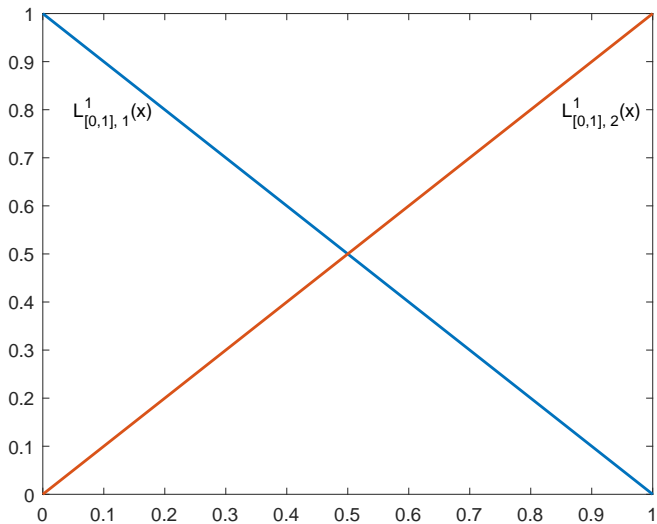
or

```
elem = [0, 1]; degree = 1; d_index = 0;  
x = linspace(elem(1), elem(2), 200);  
y1 = shape_fun_1D_Lagrange(x, elem, degree, 1, d_index);  
y2 = shape_fun_1D_Lagrange(x, elem, degree, 2, d_index);  
plot(x, y1, x, y2)
```

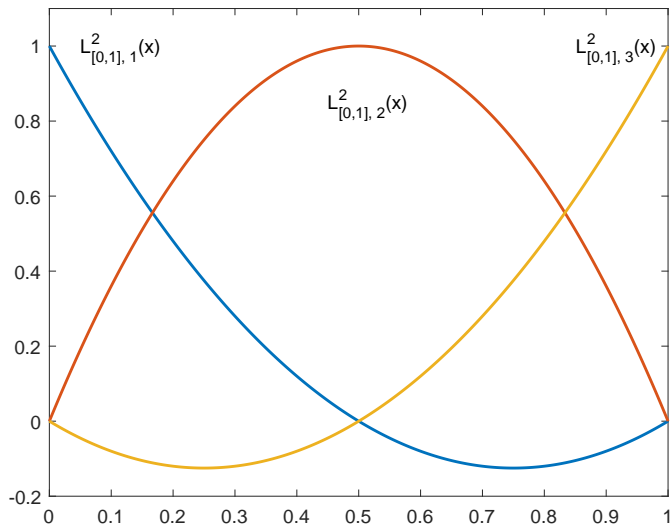
We can also use [shape_fun_1D_Lagrange](#) as functions:

```
elem = [0, 1]; degree = 1; d_index = 0;  
x = linspace(elem(1), elem(2), 200);  
shfun1 = @(x) shape_fun_1D_Lagrange(x, elem, degree, 1, d_index);  
shfun2 = @(x) shape_fun_1D_Lagrange(x, elem, degree, 2, d_index);  
plot(x, shfun1(x), x, shfun2(x))
```

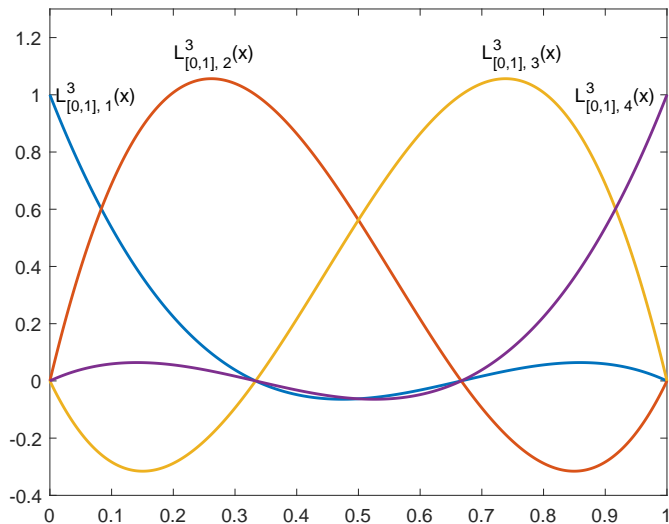
All the 1st degree shape function over $[0, 1]$:



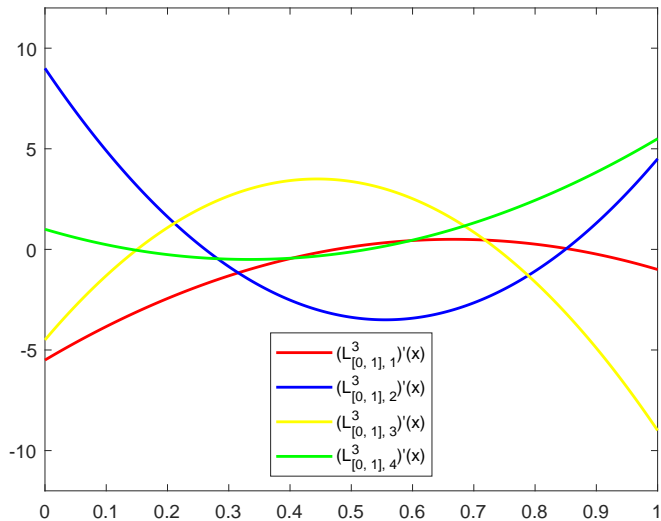
All the 2nd degree shape functions over $[0, 1]$:



All the 3rd degree shape functions over $[0, 1]$:



1st derivative of 3rd degree shape functions over $[0, 1]$:



The local p -th degree FE space on an element: for every element $K \in \mathcal{T}_h$,

$$V_h^p(K) = \text{span}\{L_{K,j}^p(x), \quad j = 1, 2, \dots, p, p+1\} = \Pi_p \quad (14)$$

where Π_p is the space of polynomials of degree p or less.

2.2.2, The global p -th degree finite element spaces:

Finite element functions over the solution domain Ω should be developed with the following considerations:

- Use shape functions locally on elements. This is problem independent.
- Use features for the trial and test functions specified by the weak form of the PDE problems.
- Use features for the finite element functions demanded by the chosen finite element methods.
- Others ...

Here, let us first consider 1D continuous Lagrange type finite element functions for solving the prototype 1D BVP problem. Recall the BVP: find u such that

$$-(au')' + cu = f, \quad x \in \Omega = (0, 1), \quad (1)$$

$$a(0)u'(0) = g_0, \quad (2)$$

$$u(1) = g_1, \quad (3)$$

A Weak Form for this BVP: find $u \in \mathcal{S}$ such that

$$\int_0^1 av'u'dx + \int_0^1 cvudx = \int_0^1 vfdx - v(0)g_0, \quad \forall v \in \mathcal{T}, \quad (5)$$

where

$$\mathcal{T} = \{w \mid w \in H^1(0, 1), w(1) = 0\}. \quad (7)$$

$$\mathcal{S} = \{w \mid w \in H^1(0, 1), w(1) = g_1\}. \quad (8)$$

The Galerkin or Petrov-Galerkin framework suggests to use finite element functions that is in the space $H^1(0, 1)$.

Theorem 2.1

Assume that $u(x)$ is a piecewise polynomial according to a mesh \mathcal{T}_h of $\Omega \subset \mathbb{R}^1$. Then $u \in H^1(\Omega)$ if and only $u \in C^0(\overline{\Omega})$.

C^0 (i.e., continuous) p -th degree Lagrange type FE space:

- Form a mesh $\mathcal{T}_h = \{K_j\}$ of Ω described by nodes $x_i, 1 \leq i \leq n + 1$.
- Form the set of **finite element nodes** for the p -th degree finite element space by collecting **local nodes** from all elements of \mathcal{T}_h , denote this set by $\mathcal{N}_{h,g}$.
- Index finite element nodes. Each of these indices corresponds to a finite element **basis function** defined on the whole Ω ; hence, we call each of these indices a **global degree of freedom**. We put all the global degrees of freedom into an array T_g of size $(p + 1) \times |\mathcal{T}_h|$ such that its j -th column contains indices of the finite element nodes in element K_j ordered according to local nodes on that element. We call T_g the connectivity matrix for the global degrees of freedom.
- For each $\tilde{x}_j \in \mathcal{N}_{h,g}$, we define a piecewise polynomial $\phi_j(x)$ such that:

$$\phi_j \in C^0(\Omega), \quad \phi_j|_K \in V_h^p(K), \quad \forall K \in \mathcal{T}_h \quad (15)$$

$$\phi_j(\tilde{x}_i) = \delta_{ij}, \quad \forall \tilde{x}_i \in \mathcal{N}_{h,g} \quad (\text{Lagrange property}) \quad (16)$$

- The p -th degree C^0 Lagrange type FE space is the following space

$$V_h^p(\Omega) = \text{span}\{\phi_i, \quad \forall \tilde{x}_i \in \mathcal{N}_{h,g}\} \quad (17)$$

Each ϕ_j is called a **global basis function** for the finite element space $V_h^p(\Omega)$. These finite element basis functions are linearly independent according to the property specified in (16), and $\dim(V_h^p(\Omega)) = |\mathcal{N}_{h,g}|$.

Recall the finite element space $V_h^p(\Omega) = \text{span}\{\phi_i, \forall \tilde{x}_i \in \mathcal{N}_{h,g}\}$.

Every finite element function $u_h(x)$ in $V_h^p(\Omega)$ can be written as

$$u_h(x) = \sum_{j=1}^{|\mathcal{N}_{h,g}|} u_j \phi_j(x) \quad (18)$$

which means a finite element function $u_h(x)$ is mathematically determined by the array $\vec{u} = (u_1, u_2, \dots, u_{|\mathcal{N}_{h,g}|})^t$.

For Lagrange type finite element functions, we note

$$u_h(\tilde{x}_i) = \sum_{j=1}^{|\mathcal{N}_{h,g}|} u_j \phi_j(\tilde{x}_i) = u_i, \quad i = 1, 2, \dots, |\mathcal{N}_{h,g}| \quad (19)$$

Hence the coefficient u_i in a finite element function $u_h(x) = \sum_{j=1}^{|\mathcal{N}_{h,g}|} u_j \phi_j(x)$ is its value at the finite element node \tilde{x}_i .

Example: Consider the C^0 linear and quadratic basis functions defined on a uniform mesh \mathcal{T}_h of the domain $\Omega = (2, 3)$ with 5 nodes.

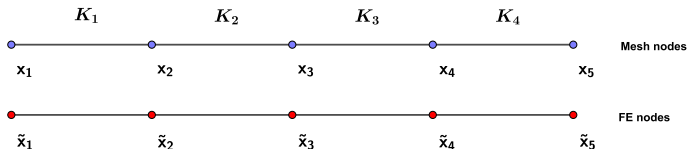
Mesh: $x_1 = 2, x_2 = 2.25, x_3 = 2.5, x_4 = 2.75, x_5 = 3$

$$\mathcal{N}_h = \{x_1, x_2, x_3, x_4, x_5\}, \quad \mathcal{T}_h = \{[x_1, x_2], [x_2, x_3], [x_3, x_4], [x_4, x_5]\}$$

Linear finite element basis functions: Note that there are two local nodes for linear polynomials on each element, we can simply let

$$\mathcal{N}_{h,g} = \{\tilde{x}_1, \tilde{x}_2, \tilde{x}_3, \tilde{x}_4, \tilde{x}_5\} = \{x_1, x_2, x_3, x_4, x_5\} = \mathcal{N}_h$$

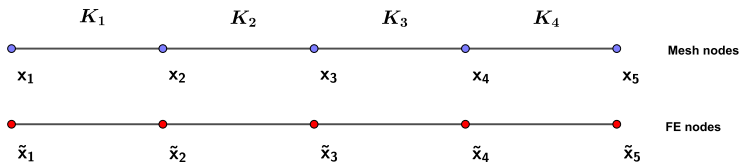
and there are 5 linear basis functions.



The relationship between linear finite element basis functions and elements in the mesh can be described by the connectivity matrix for the global degrees of freedom:

$$T_g = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

Let us discuss how a finite element basis function is described by the local shape functions on elements of a mesh. Recall the mesh:



For $\phi_1(x)$, the basis function associated with \tilde{x}_1 : By (15) and (16),

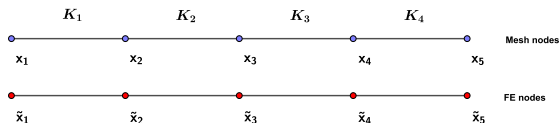
$$\phi_1(x) \in C^0([0, 1]), \phi_1|_K \in \Pi_1, \forall K \in \mathcal{T}_h, \phi_1(\tilde{x}_1) = 1, \phi_1(\tilde{x}_j) = 0, j \neq 1$$

On the 1st element $K_1 = [x_1, x_2]$, $\phi_1(x)$ is a linear polynomial with $\phi_1(x_1) = 1, \phi_1(x_2) = 0$; hence, $\phi_1(x) = L_{K_1,1}^1(x)$ for $x \in K_1$. Applying similar arguments we can see that $\phi_1(x) = 0$ for $x \in K_j, j \neq 1$. Thus,

$$\phi_1(x) = \begin{cases} L_{K_1,1}^1(x), & x \in K_1 = [x_1, x_2] \\ 0, & x \in K_i, i = 2, 3, 4 \end{cases}$$

and it is easy to verify that $\phi_1(x)$ defined by the formula above is a continuous (C^0) piecewise linear polynomial such that $\phi_1(\tilde{x}_j) = \delta_{1j}, 1 \leq j \leq 5$.

Recall the mesh and the connectivity array for the degrees of freedom:



$$T_g = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

In fact, the formula for the global finite element basis function $\phi_1(x)$ can be derived from the mesh and T_g as follows: Since the index of the basis function $\phi_1(x)$ appears in the 1st column only, the piecewise polynomial $\phi_1(x)$ should be a zero polynomial on all elements except for the 1st element.

Also, since the index of the basis function $\phi_1(x)$ appears in the **1st column and 1st row**, we know that the piecewise polynomial $\phi_1(x)$ is the polynomial (shape function) $L_{K_1,1}^1(x)$ on 1st element. Hence,

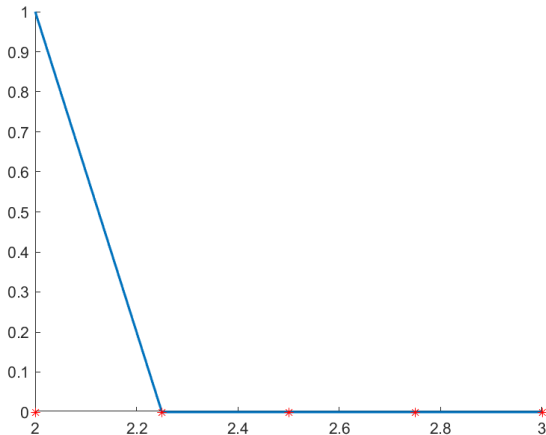
$$\phi_1(x) = \begin{cases} L_{K_1,1}^1(x), & x \in K_1 = [x_1, x_2] \\ 0, & x \in K_i, i = 2, 3, 4 \end{cases}$$

The finite element basis function $\phi_1(x)$ mathematically defined by (15) and (16) is actually implemented by:

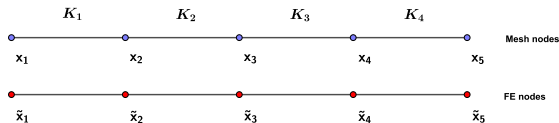
- (1) a mesh.
- (2) finite element nodes and T_g
- (3) the shape functions on elements

The formula and plot for basis function $\phi_1(x)$:

$$\phi_1(x) = \begin{cases} L_{K_1,1}^1(x), & x \in K_1 = [x_1, x_2] \\ 0, & x \in K_i, i = 2, 3, 4 \end{cases}$$



Recall the mesh and the connectivity array for the degrees of freedom:



$$T_g = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

For $\phi_3(x)$, the basis function associated with \tilde{x}_3 :

Since the index of the basis function $\phi_3(x)$ appears in the 2nd and 3rd columns only, the piecewise polynomial $\phi_3(x)$ should be a zero polynomial on all elements except for the 2nd and 3rd elements.

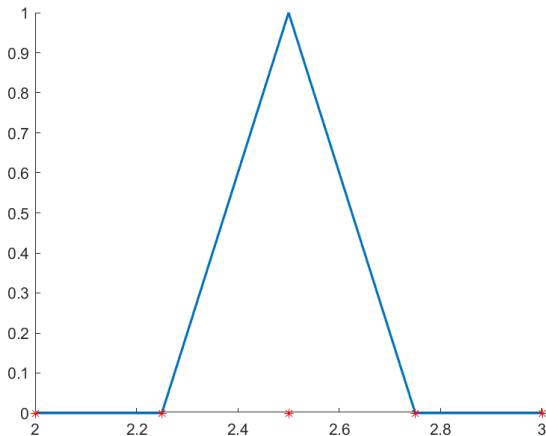
Also, since the index of the basis function $\phi_3(x)$ appears in the **2nd column and 2nd row**, we know that the piecewise polynomial $\phi_3(x)$ is the polynomial (shape function) $L_{K_2,2}^1(x)$ on 2nd element. Similarly, $\phi_3(x)$ is the polynomial (shape function) $L_{K_3,1}^1(x)$ on 3rd element. Hence

$$\phi_3(x) = \begin{cases} L_{K_2,2}^1(x), & x \in K_2 = [x_2, x_3] \\ L_{K_3,1}^1(x), & x \in K_3 = [x_3, x_4] \\ 0, & x \in K_i, i \neq 2, 3 \end{cases}$$

Again, the finite element basis function $\phi_3(x)$ mathematically defined by (15) and (16) is implemented through the mesh, T_g , and the shape functions on elements.

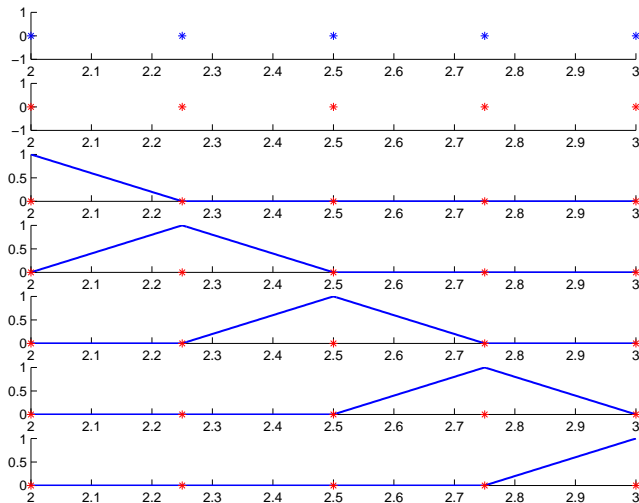
The formula and plot for basis function $\phi_3(x)$:

$$\phi_3(x) = \begin{cases} L_{K_2,2}^1(x), & x \in K_2 = [x_2, x_3] \\ L_{K_3,1}^1(x), & x \in K_3 = [x_3, x_4] \\ 0, & x \in K_i, i \neq 2, 3 \end{cases}$$



The linear finite element basis functions at other nodes can be described by the local shape function similarly. Putting them together, we have the following plot for all the 5 linear finite element basis functions defined on the mesh:

$$\phi_1(x), \phi_2(x), \phi_3(x), \phi_4(x), \phi_5(x)$$



Quadratic finite element basis functions: Recall that the mesh is characterized by

$$x_1 = 2, x_2 = 2.25, x_3 = 2.5, x_4 = 2.75, x_5 = 3,$$

$$\mathcal{N}_h = \{x_1, x_2, x_3, x_4, x_5\}, \quad \mathcal{T}_h = \{[x_1, x_2], [x_2, x_3], [x_3, x_4], [x_4, x_5]\}$$

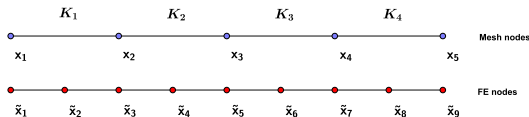
Since there are 3 local nodes for quadratic polynomials on each element, we have 9 nodes for quadratic finite element functions:

$$\mathcal{N}_{h,g} = \{\tilde{x}_1, \tilde{x}_2, \tilde{x}_3, \tilde{x}_4, \tilde{x}_5, \tilde{x}_6, \tilde{x}_7, \tilde{x}_8, \tilde{x}_9\} \supseteq \mathcal{N}_h$$

Hence, there are 9 quadratic finite element basis functions. Without loss of generality, let

$$\tilde{x}_1 = x_1, \tilde{x}_3 = x_2, \tilde{x}_5 = x_3, \tilde{x}_7 = x_4, \tilde{x}_9 = x_5$$

$$\tilde{x}_2 = (\tilde{x}_1 + \tilde{x}_3)/2, \tilde{x}_4 = (\tilde{x}_3 + \tilde{x}_5)/2, \tilde{x}_6 = (\tilde{x}_5 + \tilde{x}_7)/2, \tilde{x}_8 = (\tilde{x}_7 + \tilde{x}_9)/2$$



The connectivity matrix for the degrees of freedom:

$$T_g = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \\ 3 & 5 & 7 & 9 \end{bmatrix}$$

Remark: Again, how to order the finite element node \tilde{x}_i s does not matter much, but we need to order them to form the matrix T_g .

Recall

$$T_g = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \\ 3 & 5 & 7 & 9 \end{bmatrix}$$

For $\phi_1(x)$, the quadratic basis function associated with \tilde{x}_1 : Since the index of $\phi_1(x)$ appears in the first column only, this piecewise polynomial should be the zero polynomial on all elements except for the 1st element. Since the index of $\phi_1(x)$ appears in the first column and 1st row, piecewise polynomial $\phi_1(x)$ should be the polynomial (shape function) $L_{K1,1}^2(x)$. Hence,

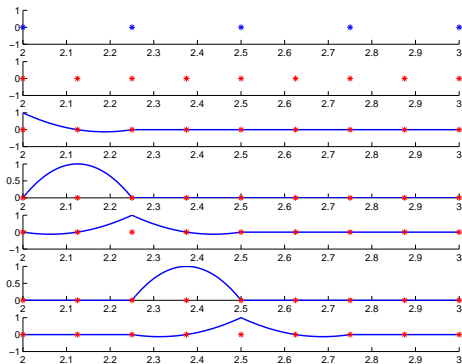
$$\phi_1(x) = \begin{cases} L_{K1,1}^2(x), & x \in K_1 \\ 0, & x \in K_j, j \neq 1 \end{cases}$$

Similarly,

$$\phi_2(x) = \begin{cases} L_{K1,2}^2(x), & x \in K_1 \\ 0, & x \in K_j, j \neq 1 \end{cases}$$

but

$$\phi_3(x) = \begin{cases} L_{K1,3}^2(x), & x \in K_1 \\ L_{K2,1}^2(x), & x \in K_2 \\ 0, & x \in K_j, j \neq 1, 2 \end{cases}$$



Implementation of global finite element basis functions:

Recall the mathematical definition of the nodal FE basis functions

$$\phi_j \in C^0(\Omega), \quad \phi_j|_K \in V_h^p(K), \quad \forall K \in \mathcal{T}_h \quad (15)$$

$$\phi_j(\tilde{x}_i) = \delta_{ij}, \quad \forall \tilde{x}_i \in \mathcal{N}_{h,g} \quad (\text{Lagrange property}) \quad (16)$$

These p -th degree C^0 finite element basis functions are determined/programmed with

```
function mesh = mesh_generator_1D(domain, n)
function f = shape_fun_1D_Lagrange(x, elem, degree, ...
                                   shape_index, d_index)
```

and the connectivity array T_g for the global degrees of freedom. We emphasize that finite element functions are rarely used globally. Computations involving finite element functions are generally carried out element by element without direct usage of the global finite element basis functions, we even do not have to program the global basis functions.

To facilitate the implementation of this idea, we need to develop a program for creating the following arrays:

For $\mathcal{N}_{h,g}$: a matrix containing coordinates for all finite element nodes of the finite element space.

T_g : the connectivity matrix for the global degrees of freedom. This matrix is of the size (number of FE nodes in each element) \times (number of elements)

The k -th column of T_g contains the indices of the finite element nodes inside k -th element.

We can implement these matrices with a Matlab function as follows:

```
fem = fem_generator_Lagrange_1D(mesh, degree)
```

`fem` is a structure for describing finite element basis functions.

`fem.p` provides coordinates for points in $\mathcal{N}_{h,g}$.

`fem.t` is the connectivity matrix T_g for the global degrees of freedom.

`fem.degree` is the degree of polynomials used for the finite element space.

Here, we assume that the data structure `mesh` has been prepared by

```
function mesh = mesh_generator_1D(domain, n)
```

```

function fem = fem_generator_Lagrange_1D(mesh, degree)
n_elem = size(mesh.t,2);
t = zeros(degree+1,n_elem); t(:,1) = (1:degree+1)';
for k=2:n_elem
    t(:,k) = t(:,k-1)+degree;
end

p = zeros(1, (degree+1) + (n_elem - 1)*degree);
k = 1;
elem = mesh.p(mesh.t(:,k));
h = (elem(2) - elem(1))/degree;
p(1:degree+1) = elem(1):h:elem(2);
pnt_count = degree + 1;
for k=2:n_elem
    elem = mesh.p(mesh.t(:,k));
    h = (elem(2) - elem(1))/degree;
    p(pnt_count+1:pnt_count+degree) = elem(1)+h:h:elem(2);
    pnt_count = pnt_count+degree;
end

fem = struct('p', p, 't', t, 'degree', degree);

```

Example: For the Lagrange type C^0 FE of degree 3 defined on a mesh with 4 element on $\Omega = (2, 3)$, we have following arrays/structures:

```
domain = [2, 3]; n = 4; mesh = mesh_generator_1D(domain, n);  
degree = 3; fem = fem_generator_Lagrange_1D(mesh, degree);
```

Then typing `mesh.p`, `mesh.t` in Matlab will show the following arrays:

```
2.0000    2.2500    2.5000    2.7500    3.0000
```

```
1      2      3      4  
2      3      4      5
```

But typing `fem.p`, and `fem.t` will show

```
Columns 1 through 11
```

```
2.0000    2.0833    2.1667    2.2500    2.3333    2.4167    2.5000    2.5833
```

```
Columns 9 through 13
```

```
2.6667    2.7500    2.8333    2.9167    3.0000
```

```
1      4      7      10  
2      5      8      11  
3      6      9      12  
4      7     10     13
```

A summary:

In mathematical description of finite element methods, we need the p -th degree C^0 finite element space:

$$V_h^p(\Omega) = \text{span}\{\phi_i, \forall \tilde{x}_i \in \mathcal{N}_{h,g}\} \quad (17)$$

with

$$\phi_j \in C^0(\Omega), \quad \phi_j|_K \in V_h^p(K), \quad \forall K \in \mathcal{T}_h \quad (15)$$

$$\phi_j(\tilde{x}_i) = \delta_{ij}, \quad \forall \tilde{x}_i \in \mathcal{N}_{h,g} \quad (\text{Lagrange property}) \quad (16)$$

In computing/programming, the p -th degree C^0 finite element basis functions $\phi_i, \forall \tilde{x}_i \in \mathcal{N}_{h,g}$ are determined with

```
(1): function mesh = mesh_generator_1D(domain, n)
(2): function fem = fem_generator_Lagrange_1D(mesh, degree)
(3): function f = shape_fun_1D_Lagrange(x, elem, degree, ...
                                     shape_index, d_index)
```

In other words, we have implemented the p -th degree C^0 finite element space defined by (17).

We proceed to discussing about how to use functions in the p -th degree C^0 finite element space.

Remarks:

- In finite element computations, the two structures **mesh** and **fem** and the program for **shape functions** on each element are sufficient in general for describing finite element functions defined on the mesh.
- The word “finite” in finite element method is very special. Finite element methods are computational methods, programming of finite element methods is guided by the “finite” spirit. For example, the finite element functions just introduced have several finite flavors. A finite element function on the whole domain is actually described by polynomials locally on elements of a mesh. Here, consider “globally on the whole domain” as everywhere and unlimited, but “locally on an element” hints limited and finite. Also, a polynomial is determined by a finite number of coefficients. As another example, for our prototype 1D BVP, seeking a weak solution is to find a function from the space $H^1(\Omega)$ which is of infinite dimension, but computing an approximation to the weak solution by the Petrove-Galerkin method or the Galerkin method is to find a function from a finite dimensional space. Finiteness is a necessary feature of finite element computations to be carried out on computers which are ultimately finite devices.