Implementation of Finite Element Analysis for One Dimensional Bar Elements in Python

Pramod Kumar Yadav

Roll No. 20132015
M.Tech- Machine Design (3rd Sem)
Indian Institute of Technology (BHU) Varanasi

Table of Contents

Overview

For this project I have implemented Finite Element Analysis(FEA) in python for a bar element. Purpose of this project is to automate FEA algorithm or steps so it will become easy to solve problems based on the 1-Dimensional (1-D) bar of any number of elements and any order of approximation function. In this project statnderd python library used are SymPy and NumPy for Mathemetical calculation and MatPlotLib for plotting graphs. Integration method used here is Gauss-Legendre Quadrature method and also implemented it from scratch.

Goals

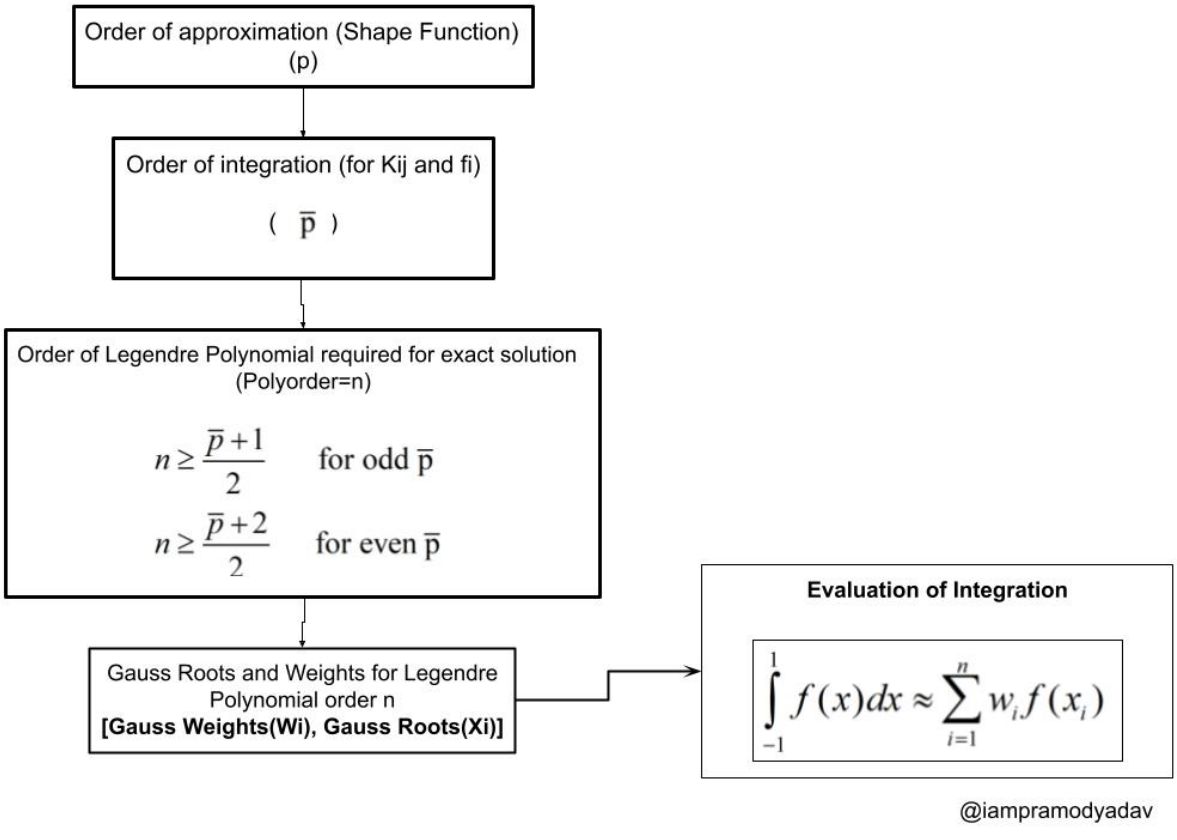
1. Write python code to implement Gauss-Legendre Quadrature for numerical integration
2. Generate Local or Global stiffness matrix and force vector for a given order of approximation (shape function), applied external distributed force and x-Coordinate of nodes

Specification

This project consist of 4 parts-

1. Function (or Subroutine) for generating shape functions of given order and visualization of its plot in natural coordinate system
2. Function to find Gauss roots of Legendre polynomial, weights and plot Legendre polynomial for required order of polynomial (for minimum integration error.)
3. Function for calculation of shape functions at Gauss roots
4. Functions to generate Local and Global, Stiffness matrices and force vector for given problem

---



Detail Code Overview with Example

Shape function and it's derivative

We know that shape function in natural coordinate for P_{th} order of approximation can be written as

$$N_i^k(x(\xi)) = \hat{N}(\xi) = \prod_{\substack{j=1 \\ j\neq i}}^{p+1} \frac{\left(\xi - \xi_j\right)}{\left(\xi_i - \xi_j\right)}, \quad i = 1,2,\ldots,p+1$$

Where physical coordinate x and natural coordinate ξ (here, z in program)Nptel Reference

Function for shape function in Python (Shape)

In [1]:
```
from sympy import *  #importing libraries
import numpy as np
```

In [2]:
```
z=Symbol('z') #for symbolic represenation
x=Symbol('x')
```

In [3]:
```
def Shape(p):
  '''
  ======================================================
  Shape(p)
  p:  order (p) of approximation
  z:  natural coordinate
  This function return Shape funtions values at given p
  ======================================================
  '''
  z=Symbol('z')
  n=[]

  for i in range(0, p+1):
    point=-1
    point=point+2*i/p
    n.append(point)

  shape=[1]*(p+1)

  for i in range(0,p+1):
    for j in range(0,p+1):
      if i!=j:
        shape[i]=shape[i]*((z-n[j])/(n[i]-n[j]))

  return shape
```

In [4]:
```
print(Shape.__doc__)
```

```
  ======================================================
  Shape(p)
  p:  order (p) of approximation
  z:  natural coordinate
  This function return Shape funtions values at given p
```

```
    ====================================================
```

Example: For shape function of 2nd order

In [5]:
```
Shape(2)
```

Out[5]:
```
[-1.0*z*(0.5 - 0.5*z), (1.0 - 1.0*z)*(1.0*z + 1.0), 1.0*z*(0.5*z + 0.5)]
```

Function for derivative of shape function (dShape)

In [6]:
```
def dShape(p):
  '''
  ====================================================
  dShape(p)
  p:  order (p) of approximation

  z:  natural coordinate variable
  This function return derivative of Shape funtions values at given p
  ====================================================
  '''
  z=Symbol('z')
  f=Shape(p)

  return [simplify(diff(f1)) for f1 in f]
```

In [7]:
```
print(dShape.__doc__)
```

```
  ====================================================
  dShape(p)
  p:  order (p) of approximation

  z:  natural coordinate variable
  This function return derivative of Shape funtions values at given p
  ====================================================
```

Example: For derivative of shape function of 2nd order

In [8]:
```
dShape(2)
```

Out[8]:
```
[1.0*z - 0.5, -2.0*z, 1.0*z + 0.5]
```

Function for Visualization of shape function (PlotShape)

In [9]:
```
import seaborn as sns
from matplotlib import style
import matplotlib.pyplot as plt
```

In [10]:
```
def plotShape(p):
  '''
  ====================================================
  plotShape(p)
```

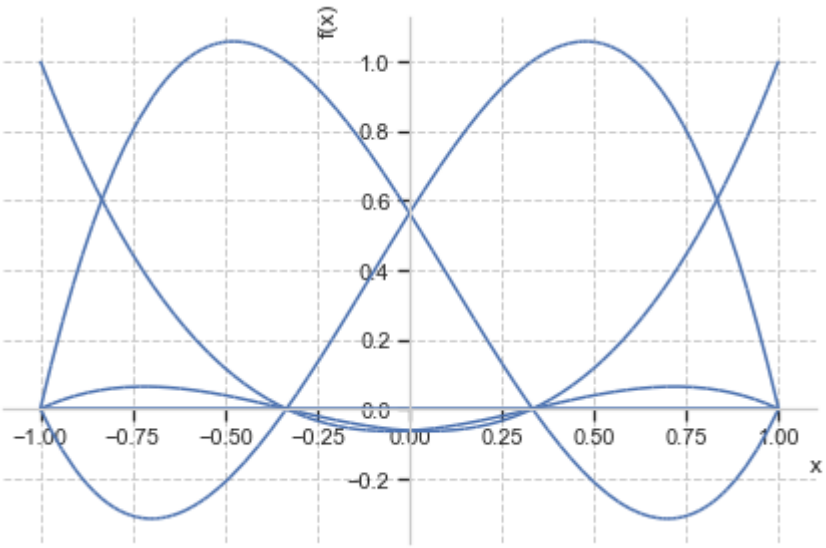
```
    p: order (p) of approximation
    z:value of natural coordinate
    This function return plot of Shape funtions (Natural Coordinate)
    ======================================================
    '''
    sns.set()
    sns.set_style("whitegrid", {'grid.linestyle': '--'})

    z=Symbol('z')
    f=Shape(p)
    p1=plot(0,(z,-1,1),show=False)
    for i in f:
      p2=plot(i,(z,-1,1),show=False)
      p1.append(p2[0])

    return p1.show()
```

In [11]:
```
print(plotShape.__doc__)
```

```
    ======================================================
    plotShape(p)

    p: order (p) of approximation
    z:value of natural coordinate
    This function return plot of Shape funtions (Natural Coordinate)
    ======================================================
```

Example: Plot shape function of 3rd order

In [12]:
```
plotShape(3)
```



Gauss-Legendre Quadrature implementation in Python

According to Gauss-Legendre Quadrature rule in any numerical integration scheme, the integral is expressed as a sum over certain points of the domain of integration called as **sampling points** . A typical term in the sum consists of the value of the function at that point multiplied by a suitable number called as the **weight** . Thus,

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

Here, w_i are the weights. In case of Gauss quadrature rule, the sampling points $i = 1, 2, 3....n$ are called as the Gauss points and $x_i, i = 1, 2, 3....n$ are called as Gauss points coordinates (Roots of Legendre polynomial.)

Function for Legendre Polynomial in symbolic form (Legendre)

To write program for Legendre polynomial we will Bonnet's recursion formula for Legendre polynomial, which can be written as:

- $$\boxed{(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)}$$
- $$P_0(x) = 1$$
- $$P_1(x) = x$$

In [13]:
```
from sympy import *
import numpy as np
```

In [14]:
```
z=Symbol('z') #for symbolic represenation
x=Symbol('x')
```

In [15]:
```
def Legendre(n):
  """
  ======================================================
  n: Order of polynomial
  x: Variable
  This function print Legendre polynomial of order n (Symbolic)
  ======================================================
  """

  x=symbols('x')
  if (n==0):
    return x*0+1.0
  elif (n==1):
    return x
  else:
    return ((2.0*n-1.0)*x*Legendre(n-1)-(n-1)*Legendre(n-2))/n
```

In [16]:
```
print(Legendre.__doc__)
```

```
  ======================================================
  n: Order of polynomial
  x: Variable
  This function print Legendre polynomial of order n (Symbolic)
  ======================================================
```

Example: Find Legendre Polynomial for 2nd and 3rd order

In [17]:
```
Legendre(2) #2nd Order
```

Out[17]:

$$1.5x^2 - 0.5$$

In [18]:
```
Legendre(3) #3rd order
```

Out[18]:
$$1.66666666666667x\left(1.5x^2 - 0.5\right) - \frac{2x}{3}$$

In [19]:
```
simplify(Legendre(3)) #simplified form
```

Out[19]:
$$x\left(2.5x^2 - 1.5\right)$$

Function for Derivative of Legendre Polynomial in symbolic form (dLegendre)

Here also Bonnet's recursion formula is used. it can be writen as (Source.)

$$\boxed{\frac{x^2-1}{n}\frac{d}{dx}P_n(x) = xP_n(x) - P_{n-1}(x)}$$

or

$$\boxed{\frac{d}{dx}P_n(x) = \frac{n}{x^2-1}(xP_n(x) - P_{n-1}(x))}$$

In [20]:
```
def dLegendre(n):
  """
  ====================================================
  n: Order of polynomial
  x: Variable
  This function print Derivative of Legendre polynomial of order n  (Symbolic)
  ====================================================
  """
  x=symbols('x')
  if (n==0):
    return x*0
  elif (n==1):
    return x*0+1.0
  else:
    return (n/(x**2-1.0))*(x*Legendre(n)-Legendre(n-1))
```

In [21]:
```
print(dLegendre.__doc__)
```

```
  ====================================================
  n: Order of polynomial
  x: Variable
  This function print Derivative of Legendre polynomial of order n  (Symbolic)
  ====================================================
```

Example: Find derivative of Legendre Polynomial for 2nd and 3rd order

In [22]:
```
dLegendre(2)
```

Out[22]:

$$\frac{2\left(x\left(1.5x^2-0.5\right)-x\right)}{x^2-1.0}$$

In [23]:
```
simplify(dLegendre(2)) #2nd order simpified form
```

Out[23]: $3.0x$

In [24]:
```
dLegendre(3)
```

Out[24]: $$\frac{3\left(-1.5x^2+x\left(1.66666666666667x\left(1.5x^2-0.5\right)-\frac{2x}{3}\right)+0.5\right)}{x^2-1.0}$$

In [25]:
```
simplify(dLegendre(3)) #3rd order simpified form
```

Out[25]: $7.5x^2-1.5$

Function to find roots of Legendre Polynomial or Gauss points (LegendreRoots)

Here to find legendre roots Newton–Raphson method is used, which can be written as

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

The first guess x_0 for the i-th root of a n order polynomial P_n can be given by

$$x_0 = \cos\left(\frac{\pi(i-1/4)}{n+1/2}\right)$$

In [26]:
```
def LegendreRoots(polyorder,tolerance=1e-20):
  """
  ==========================================================
  LegendreRoots(polyorder,tolerance=1e-20)

  polyorder: Order of polynomial(>2)
  tolerance: tolerance of error (default: 1e-20)
  Output: [roots, err]
  This function return Roots of Legendre polynomial
  ==========================================================
  """
  if polyorder<2:
    err=1

  else:
    roots=[]

    for i in range(1,int((polyorder)/2) +1):
      x1=cos(pi*(i-0.25)/(polyorder+0.5))
      error=10*tolerance
      iters=0

      while (error>tolerance) and (iters<1000):
        dx=-Legendre(polyorder)/dLegendre(polyorder)
        dx=N(dx.subs(x,x1))
        x1=N(x1+dx)
```

```
        iters=iters+1
        error=abs(dx)

      #print(roots)
      roots.append(x1)
      #print(roots)

    roots=np.array(roots)
    if polyorder%2==0:
      roots=np.concatenate( (-1.0*roots, roots[::-1]) )

    else:
      roots=np.concatenate( (-1.0*roots, [0.0], roots[::-1]) )

    err=0

  return [roots, err]
```

In [27]:

```
print(LegendreRoots.__doc__)
```

```
  ====================================================
  LegendreRoots(polyorder,tolerance=1e-20)

  polyorder: Order of polynomial(>2)
  tolerance: tolerance of error (default: 1e-20)
  Output: [roots, err]
  This function return Roots of Legendre polynomial
  ====================================================
```

Examples Find Gauss roots or Legendre Roots for 3rd,4th and 7th order legendre polynomial

In [28]:

```
[Xis, err]=LegendreRoots(3)
print("Gauss points for 3rd order:\n",Xis)
```

```
Gauss points for 3rd order:
 [-0.774596669241483 0.0 0.774596669241483]
```

In [29]:

```
[Xis, err]=LegendreRoots(4)
print("Gauss points for 4th order:\n",Xis)
```

```
Gauss points for 4th order:
 [-0.861136311594053 -0.339981043584856 0.339981043584856 0.861136311594053]
```

In [30]:

```
[Xis, err]=LegendreRoots(7)
print("Gauss points for 7th order:\n",Xis)
```

```
Gauss points for 7th order:
 [-0.949107912342759 -0.741531185599394 -0.405845151377397 0.0
 0.405845151377397 0.741531185599394 0.949107912342759]
```

Function to find Weight Coefficient (GaussWeights)

To find weight coefficient the $i-th$ Gauss node, x_i, is the $i-th$ root of P_n and the weights are given by the formula (Abramowitz & Stegun 1972)

$$w_i = \frac{2}{\left(1-x_i^2\right)\left[P_n'(x_i)\right]^2}.$$

In [31]:
```
def GaussWeights(polyorder):
  """
  ======================================================
  GaussWeights(polyorder)

  polyorder: Order of polynomial
  Output: [W, xis, err]
  This function return Weights(W) and Roots(xis) of Legendre polynomial
  ======================================================
  """
  W=[]
  [xis,err]=LegendreRoots(polyorder)
  xis=list(xis)
  if err==0:
    for x1 in xis:
      w=2.0/( (1.0-x**2)*(dLegendre(polyorder)**2) )
      w=w.subs(x,x1)
      W.append(w)
      err=0
  else:
    err=1 # could not determine roots - so no weights
  return [W, xis, err]
```

In [32]:
```
print(GaussWeights.__doc__)
```

```
  ======================================================
  GaussWeights(polyorder)

  polyorder: Order of polynomial
  Output: [W, xis, err]
  This function return Weights(W) and Roots(xis) of Legendre polynomial
  ======================================================
```

Example: Find weights for 3rd,4th and 7th order legendre polynomial

In [33]:
```
[W, xis, err]=GaussWeights(3)
print("Gauss weights for 3rd order:\n",W)
```

```
Gauss weights for 3rd order:
 [0.555555555555556, 0.888888888888889, 0.555555555555556]
```

In [34]:
```
[W, xis, err]=GaussWeights(4)
print("Gauss weights for 4th order:\n",W)
```

```
Gauss weights for 4th order:
 [0.347854845137454, 0.652145154862546, 0.652145154862546, 0.347854845137454]
```

In [35]:
```
[W, xis, err]=GaussWeights(7)
print("Gauss weights for 7th order:\n",W)
```

```
Gauss weights for 7th order:
 [0.129484966168870, 0.279705391489277, 0.381830050505119, 0.417959183673469, 0.3818300505
05119, 0.279705391489277, 0.129484966168870]
```

Function to Plot Legendre Polynomial (PlotLegendre)

Function can be written as,

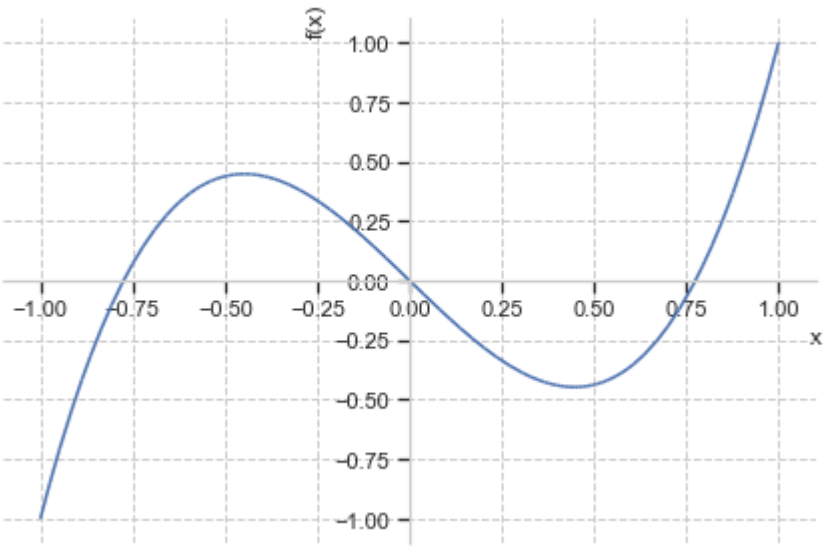
In [36]:
```
def PlotLegendre(polyorder):
  """
  ======================================================
  PlotLegendre(polyorder)
  polyorder: Order of polynomial
  This function plot legendre plynomial
  ======================================================
  """
  sns.set()
  sns.set_style("whitegrid", {'grid.linestyle': '--'})
  plot(Legendre(polyorder),(x,-1,1))
```

In [37]:
```
print(PlotLegendre.__doc__)
```

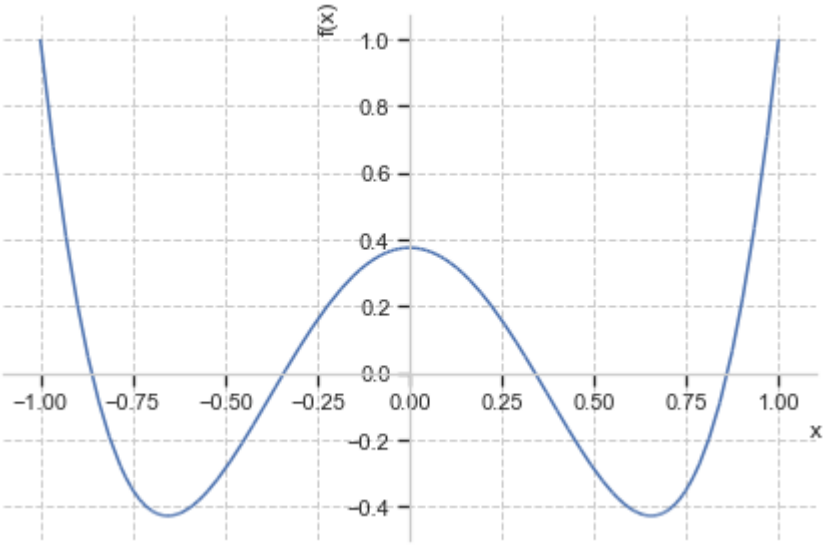
```
  ======================================================
  PlotLegendre(polyorder)
  polyorder: Order of polynomial
  This function plot legendre plynomial
  ======================================================
```

Example:plot 3rd,4th order legendre polynomial

In [38]:
```
PlotLegendre(3)
```



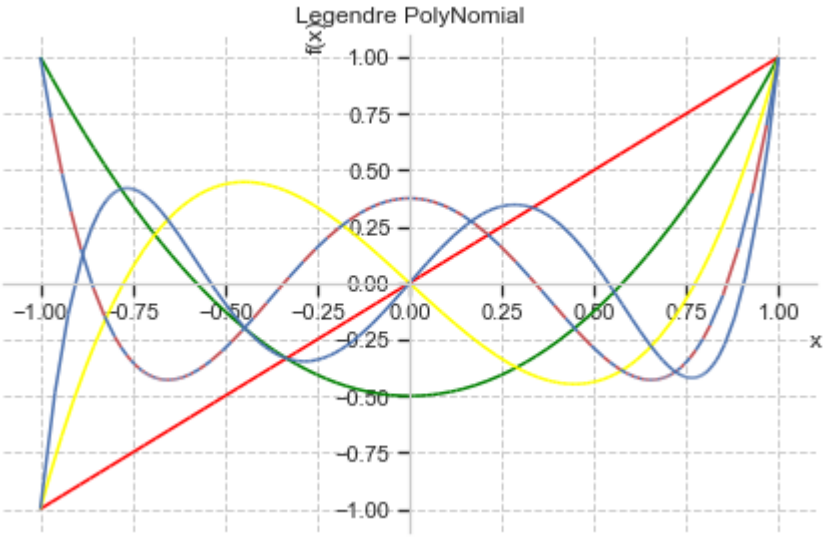
In [39]:
```
PlotLegendre(4)
```



Plot of legendre polynomial from 1 to 5th order

In [40]:

```
p=plot(Legendre(1),Legendre(2),Legendre(3),Legendre(4),Legendre(5),(x,-1,1),
       title='Legendre PolyNomial',show=False);
p[0].line_color = 'red'
p[1].line_color = 'green'
p[2].line_color = 'yellow'
p[3].line_color = ['b','r']

p.show()
```



Testing Final Result

Plot and find Gauss roots and weights for polynomial order 2 to 6

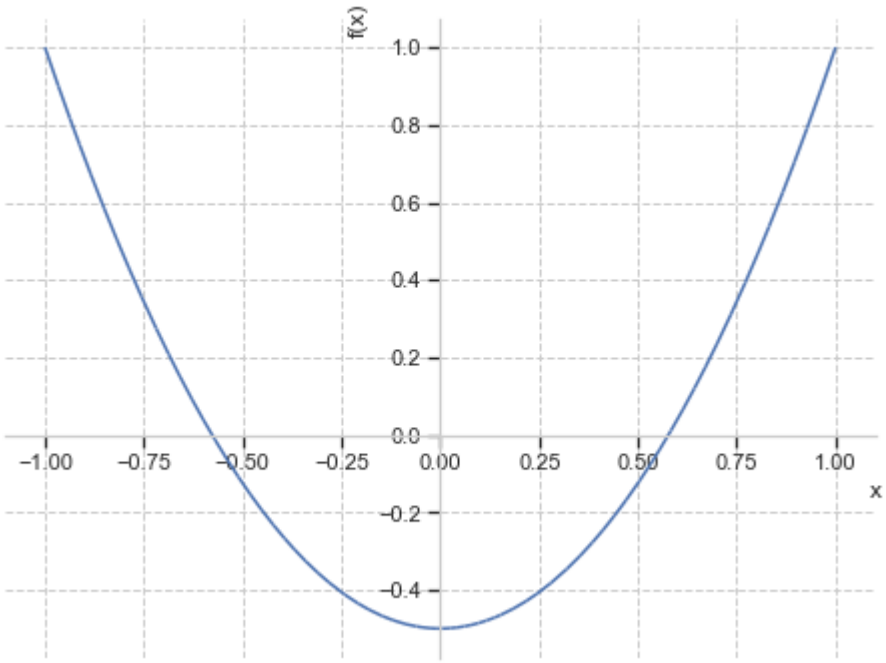
In [41]:

```
from matplotlib import style
import matplotlib.pyplot as plt
```

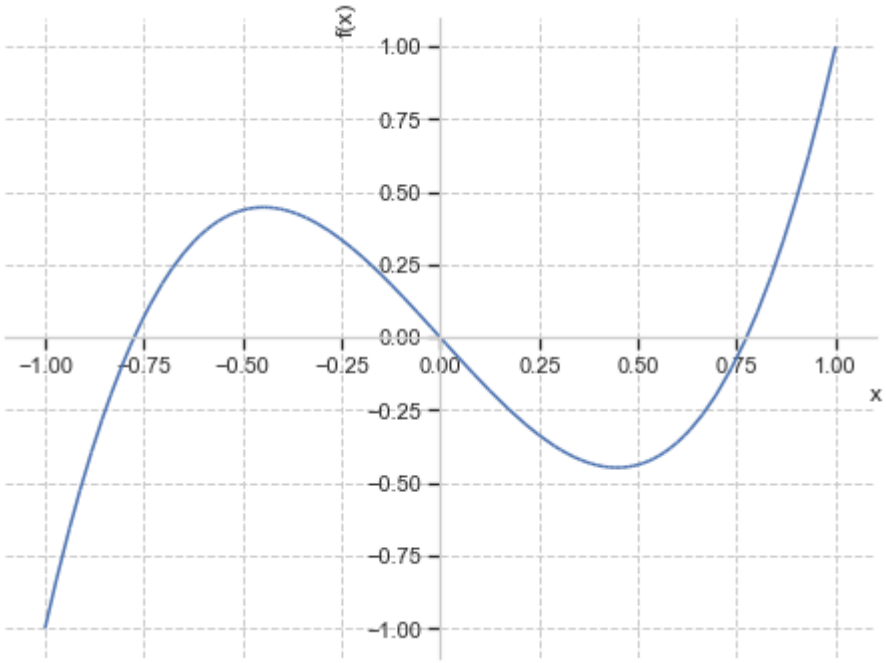
In [42]:

```
for Polyorder in range(2,7):
  plt.rcParams['figure.figsize'] = 6.4, 4.8
  plot(Legendre(Polyorder),(x,-1,1))

  [W, xis, err]=GaussWeights(Polyorder)
```

```
    print("order:", Polyorder)
    print("Weights:", W)
    print ("Roots    : ",xis)

    print('='*90)
    print('\n\n')
    print('='*90)
```



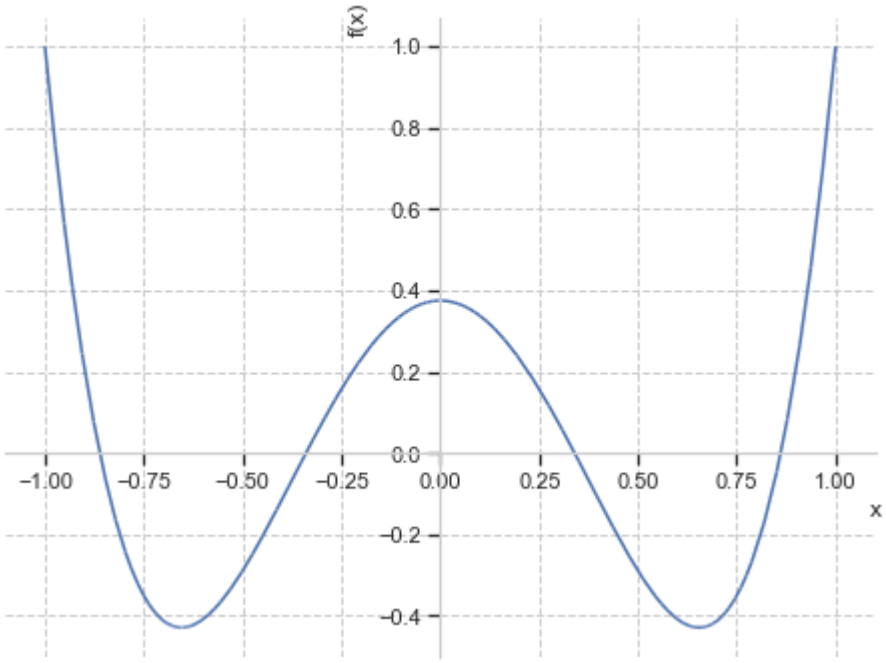

```
order: 2
Weights: [1.00000000000000, 1.00000000000000]
Roots    :  [-0.577350269189626, 0.577350269189626]
==========================================================================================
======


==========================================================================================
======
```




```
order: 3
Weights: [0.555555555555556, 0.888888888888889, 0.555555555555556]
Roots    :  [-0.774596669241483, 0.0, 0.774596669241483]
==========================================================================================
```

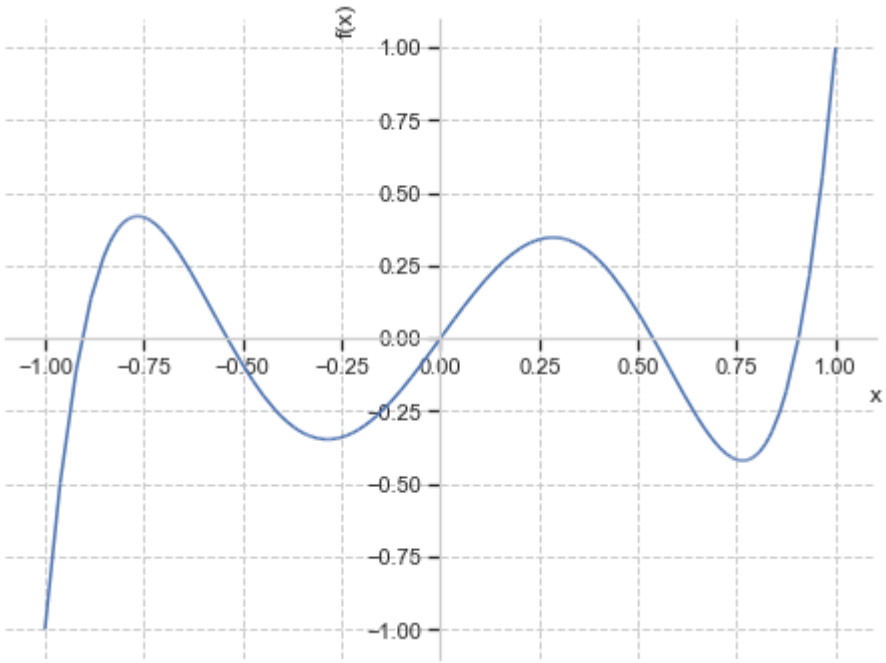
======

==========
======



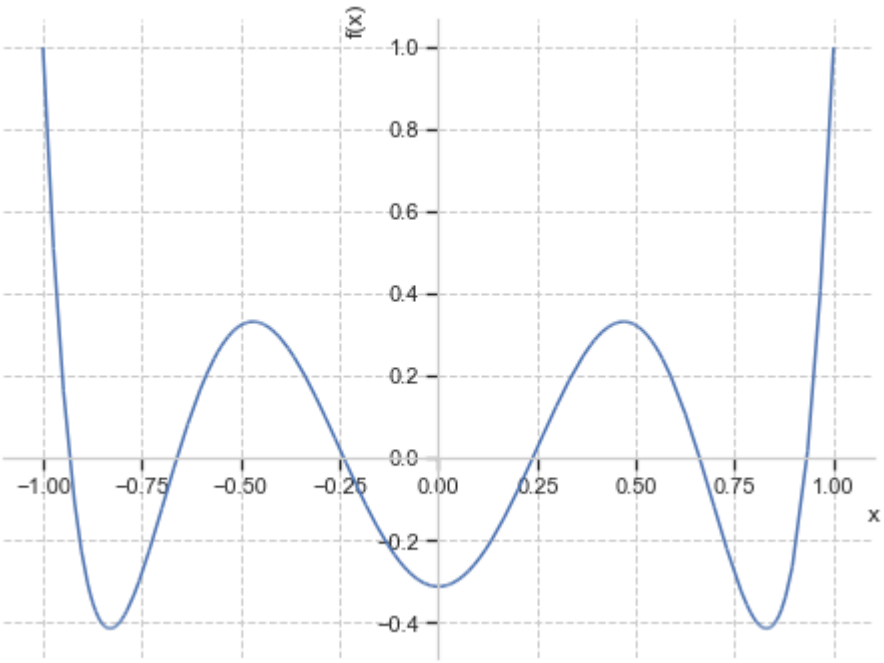
order: 4
Weights: [0.347854845137454, 0.652145154862546, 0.652145154862546, 0.347854845137454]
Roots   : [-0.861136311594053, -0.339981043584856, 0.339981043584856, 0.861136311594053]
==========
======

==========
======



order: 5
Weights: [0.236926885056189, 0.478628670499366, 0.568888888888889, 0.478628670499366, 0.23
6926885056189]
Roots   : [-0.906179845938664, -0.538469310105683, 0.0, 0.538469310105683, 0.90617984593
8664]
==========
======

```
================================================================================================
======
```




```
order: 6
Weights: [0.171324492379171, 0.360761573048138, 0.467913934572691, 0.467913934572691, 0.36
0761573048138, 0.171324492379171]
Roots    :  [-0.932469514203152, -0.661209386466265, -0.238619186083197, 0.23861918608319
7, 0.661209386466265, 0.932469514203152]
================================================================================================
======
```

```
================================================================================================
======
```

Function for Values of Shape Function and derivative at Gauss Points (ShapeLegRoot)

This function calculates the values of shape functions and their derivatives for given order of approximation p Note: Here I assumed,

- Area is maximum of 2nd order i.e. $Area = a_0 + a_1 x + a_2 x^2$
- Distributed Force also is maximum of 2nd order i.e. $f = f_0 + f_1 x + f_2 x^2$

  We can increase it by just one line code change.

In [43]:

```
def ShapeLegRoot(p):
  """
  ======================================================
  ShapeLegRoot(p)

  p : order of approximation (integer)

  Output: [n, W, xis, ShapeValueXi, dShapeValueXi]
  Here,
  n-    Order of gauss legendre polynomial
  W-    Legendre Weights
```

```
  xis-  Gauss point coordinates (real)
  ShapeValueXi-   Value of Shape function at  at all the  Xi (real)(Legendre Roots)
  ShapeValueXi-   Value of derivative of Shape function at Gauss points Xi(Legendre Roots)
  ======================================================
  """

  p=int(p)

  pbar=max(2*p, p+2)
  if pbar%2!=0:                                    #if odd
    n=(pbar+1)/2
  else:
    n=(pbar+2)/2

  n=int(n)

  #print("Order of gauss legendre polynomial:",n)

  [W, xis, err]=GaussWeights(n)           #order: Polyorder(n), Weights:  W, Roots: xis

  ShapeValueXi=np.zeros((n, p+1))
  dShapeValueXi=np.zeros((n, p+1))

  #print('ndofel(p+1):',p+1)
  #print('Shape of Shape value matrix:',ShapeValueXi.Shape)

  s=Shape(p)
  ds=dShape(p)

  for i in range(0,n):
    for j in range(0,p+1):

      ShapeValueXi[i][j]=s[j].subs(z,xis[i])          #w.subs(x,x1)
      dShapeValueXi[i][j]=ds[j].subs(z,xis[i])

  return [n, W, xis, ShapeValueXi, dShapeValueXi]
```

In [44]:

```
print(ShapeLegRoot.__doc__)
```

```
======================================================
ShapeLegRoot(p)

p : order of approximation (integer)

Output: [n, W, xis, ShapeValueXi, dShapeValueXi]
Here,
n-    Order of gauss legendre polynomial
W-    Legendre Weights
xis-  Gauss point coordinates (real)
ShapeValueXi-   Value of Shape function at  at all the  Xi (real)(Legendre Roots)
ShapeValueXi-   Value of derivative of Shape function at Gauss points Xi(Legendre Roots)
======================================================
```

Example: find shape function and derivative of shape function at legendre roots (Gauss Points) for 1st and 2nd order approximation

In [45]:

```
[n, W,xis,ShapeValueXi,dShapeValueXi]=ShapeLegRoot(1)
print('Shape Function values at Legendre roots:\n',ShapeValueXi)
print('Derivative of Shape Function values at Legendre roots\n:',dShapeValueXi)
```

```
Shape Function values at Legendre roots:
 [[0.78867513 0.21132487]
 [0.21132487 0.78867513]]
Derivative of Shape Function values at Legendre roots
: [[-0.5  0.5]
 [-0.5  0.5]]
```

In [46]:

```
[n, W,xis,ShapeValueXi,dShapeValueXi]=ShapeLegRoot(3)
print('Shape Function values at Legendre roots:\n',ShapeValueXi)
print('Derivative of Shape Function values at Legendre roots\n:',dShapeValueXi)
```

```
Shape Function values at Legendre roots:
 [[ 0.66000567  0.52093769 -0.2301879   0.04924455]
 [ 0.00337374  1.00488585 -0.00992135  0.00166176]
 [ 0.00166176 -0.00992135  1.00488585  0.00337374]
 [ 0.04924455 -0.2301879   0.52093769  0.66000567]]
Derivative of Shape Function values at Legendre roots
: [[-2.15765367  3.03540432 -1.09784762  0.22009697]
 [-0.51503192 -0.71986158  1.48481893 -0.24992543]
 [ 0.24992543 -1.48481893  0.71986158  0.51503192]
 [-0.22009697  1.09784762 -3.03540432  2.15765367]]
```

Element Calculation

Here I will implement function to calculate Stiffness matrix and Force vector for different cases:

- Here linear mapping function used between the physical coordinate (x) and the natural coordinate ξ (here, z) :
  - $$x = x_1^k \left[\frac{1}{2}(1-\xi)\right] + x_{p+1}^k \left[\frac{1}{2}(1+\xi)\right]$$
    - Hence, x_1^k (xL) and x_{p+1}^k (xR) are the coordinates of the end nodes of the element k and p denotes the order of approximation.
- Expressions for Element Stiffness Matrix and Force Vector
  - $$\boxed{k_{ij}^k = \int_{-1}^{1} \hat{E}(\xi)\hat{A}(\xi)\frac{d\hat{N}_i(\xi)}{d\xi}\frac{d\hat{N}_j(\xi)}{d\xi}\frac{2}{h_k}d\xi \quad \text{for } i,j = 1,2,\ldots,p+1}$$
  - $$\boxed{f_i^k = \int_{-1}^{1} \hat{f}\,(\xi)\hat{N}_i(\xi)\frac{h_k}{2}d\xi \quad i = 1,2,\ldots,p+1}$$

Function for Global (All elements of same size) K_{ij} & f_i if 1-D Bar (ElemKfi)

In [47]:

```
def ElemKfi(k,p,xL,xR,E,a0,a1=0,a2=0,f0=0,f1=0,f2=0):
  """
  ======================================================
  ElemKfi(k,p,xL,xR,E,a0,a1=0,a2=0,f0=0,f1=0,f2=0)

  k :   element number (integer)
  p :   order of approximation (integer)
  xL :  x -coordinate of the left end of the element (real)
```

```
  xR :  x -coordinate of the right end of the element (real)
  E :   Young's modules ( E ) of the bar material which is taken as constant (real)
  a0,a1,a2 : coefficients in (A=a0 + a1*x + a2*(x**2)) for the area of cross-section ( A )
  f0,f1,f2 : coefficients in (f = f0 + f1*x + f2*x*x) for the distributed force ( f ) acti

  Output: [Ke, fe]
  Ke : element stiffness matrix (real)
  fe : element force vector (real)
  ======================================================
  """

  Ke=np.zeros((p+1,p+1))  # size of element matrix
  h=xR - xL               # h : element length (real)

  [n, W,xis,ShapeValueXi,dShapeValueXi]=ShapeLegRoot(p)
  fe=np.zeros(p+1)

  for m in range(0,n):
    xi=xis[m]
    x = xL*(1 - z)/2 + xR*(1 + z)/2

    EA = E*(a0 + a1*x+a2*(x**2))
    EA = EA.subs(z,xi)

    f = f0 + f1*x + f2*x*x
    f=f.subs(z,xi)

    for i in range(0, p+1):
      fe[i] = fe[i] + f*ShapeValueXi[m][i]*(h/2)*W[m]

      for j in range(0,p+1):
        Ke[i][j] = Ke[i][j]+EA*dShapeValueXi[m][i]*dShapeValueXi[m][j]*(2/h)*W[m]

  return [Ke, fe]
```

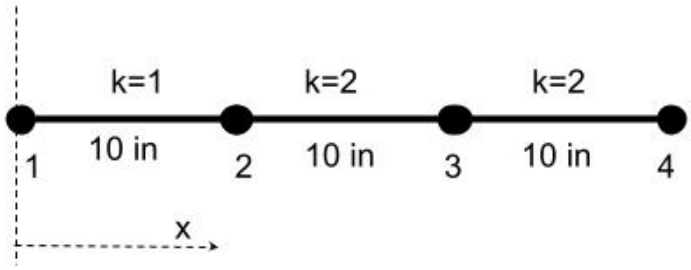
In [48]:

```
print(ElemKfi.__doc__)
```

```
  ======================================================
  ElemKfi(k,p,xL,xR,E,a0,a1=0,a2=0,f0=0,f1=0,f2=0)

  k :   element number (integer)
  p :   order of approximation (integer)
  xL :  x -coordinate of the left end of the element (real)
  xR :  x -coordinate of the right end of the element (real)
  E :   Young's modules ( E ) of the bar material which is taken as constant (real)
  a0,a1,a2 : coefficients in (A=a0 + a1*x + a2*(x**2)) for the area of cross-section ( A )
of the bar (real)
  f0,f1,f2 : coefficients in (f = f0 + f1*x + f2*x*x) for the distributed force ( f ) acti
ng on the bar (real)

  Output: [Ke, fe]
  Ke : element stiffness matrix (real)
  fe : element force vector (real)
  ======================================================
```

Example:1

Find Local stiffness matrix for each element and also find force vector due to distributed force.



Area=6-0.1x
Distributed force (f)=0.0216(6-0.1x)

In [49]:
```
[ke,fe]=ElemKfi(k=1,p=1,xL=0,xR=10,a0=6,a1=-0.1,a2=0,E=30,f0=0.1296,f1=-0.00216)
print("Local Stiffness Matrix(1): \n\n", ke,"\n\n")
print("Local Force Vector (due to distributed force):\n\n ", fe)
```

```
Local Stiffness Matrix(1):

 [[ 16.5 -16.5]
 [-16.5  16.5]]


Local Force Vector (due to distributed force):

  [0.612 0.576]
```

In [50]:
```
[ke,fe]=ElemKfi(k=2,p=1,xL=10,xR=20,a0=6,a1=-0.1,E=30,f0=0.1296,f1=-0.00216)
print("Local Stiffness Matrix(2): \n\n", ke,"\n\n")
print("Lobal Force Vector (due to distributed force):\n\n ", fe)
```

```
Local Stiffness Matrix(2):

 [[ 13.5 -13.5]
 [-13.5  13.5]]


Lobal Force Vector (due to distributed force):

  [0.504 0.468]
```

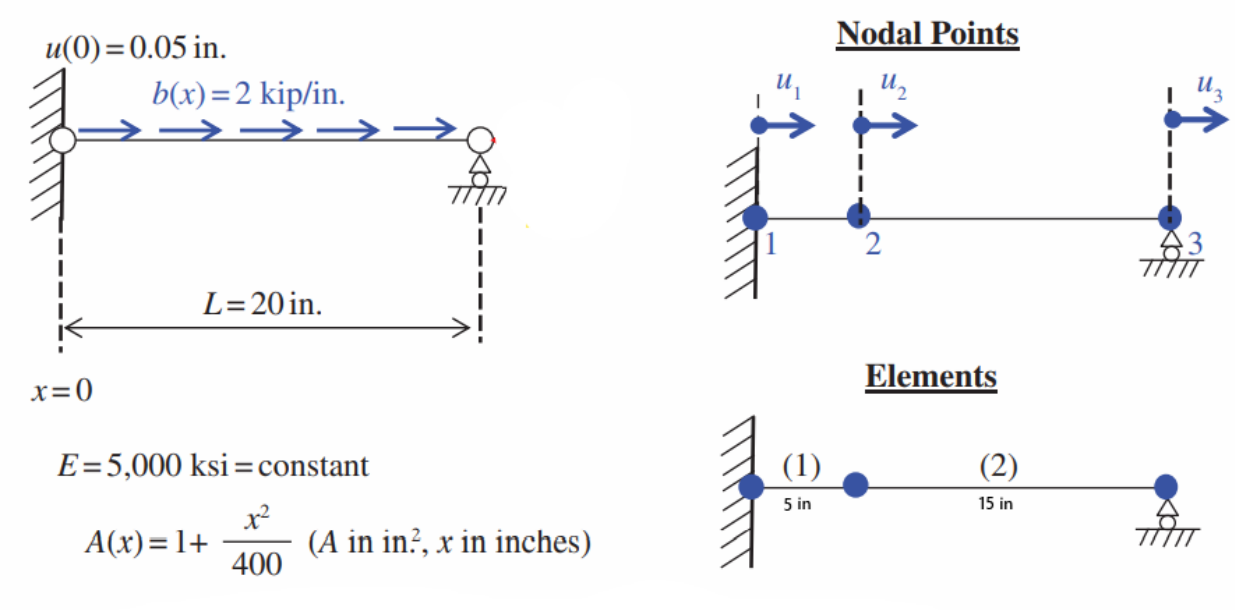
In [51]:
```
[ke,fe]=ElemKfi(k=3,p=1,xL=20,xR=30,a0=6,a1=-0.1,E=30,f0=0.1296,f1=-0.00216)
print("Local Stiffness Matrix(3): \n\n", ke,"\n\n")
print("Lobal Force Vector (due to distributed force):\n\n ", fe)
```

```
Local Stiffness Matrix(3):

 [[ 10.5 -10.5]
 [-10.5  10.5]]


Lobal Force Vector (due to distributed force):

  [0.396 0.36 ]
```

Example:2

Find Local stiffness matrix for each element and also find force vector due to distributed force.

- book: Fundamentals of Finite Element Analysis Linear Finite Element Analysis-Ioannis Koutromanos
- example-33.2
- page-87/729



Stiffeness and Force

In [52]:
```
[ke,fe]=ElemKfi(k=1,p=1,xL=0,xR=5,E=5000,a0=1,a1=0,a2=1/400,f0=2) #for element-1
print("Local Stiffness Matrix(1): \n\n", ke,"\n\n")
print("Lobal Force Vector (due to distributed force):\n\n ", fe)
```

```
Local Stiffness Matrix(1):

 [[ 1020.83333333 -1020.83333333]
 [-1020.83333333  1020.83333333]]


Lobal Force Vector (due to distributed force):

  [5. 5.]
```

In [53]:
```
[ke,fe]=ElemKfi(k=1,p=1,xL=5,xR=20,E=5000,a0=1,a1=0,a2=1/400,f0=2) #for element-2
print("Local Stiffness Matrix(2): \n\n", ke,"\n\n")
print("Lobal Force Vector (due to distributed force):\n\n ", fe)
```

```
Local Stiffness Matrix(2):

 [[ 479.16666667 -479.16666667]
 [-479.16666667  479.16666667]]


Lobal Force Vector (due to distributed force):

  [15. 15.]
```

Function for Global (All elements of same size) K_{ij} & f_i if 1-D Bar (GlobalKfi)

In [54]:

```
def GlobalKfi(k,p,xL,xR,E,a0,a1=0,a2=0,f0=0,f1=0,f2=0):

  """
  =====================================================
    Note: For elements of equal length

    GlobalKfi(k,p,xL,xR,E,a0,a1=0,a2=0,f0=0,f1=0,f2=0)

    k :   number of elements of same size (integer)
    p :   order of approximation (integer)
    xL :  x -coordinate of the left end of the element (real)
    xR :  x -coordinate of the right end of the element (real)
    E :   Young's modules ( E ) of the bar material which is taken as constant (real)
    a0,a1,a2 : coefficients in (A=a0 + a1*x + a2*(x**2)) for the area of cross-section ( A
    f0,f1,f2 : coefficients in (f = f0 + f1*x + f2*x*x) for the distributed force ( f ) ac

    Output: [Kg, fg]
    Kg : Global stiffness matrix (real)
    fg : Global force vector (real)
  =====================================================
  """

  Kg=np.zeros((k*p+1,k*p+1))   #size of element matrix
  h=xR - xL

  [n, W,xis,ShapeValueXi,dShapeValueXi]=ShapeLegRoot(p)
  fg=np.zeros(k*p+1)

  for e in range(0,k*p,p):
    #xL=N[e]
    #xR=N[e+1]
    for m in range(0,n):
      xi=xis[m]
      x = (xL+h*int(e/p))*(1 - z)/2 + (xR+h*int(e/p))*(1 + z)/2

      EA = E*(a0 + a1*x+a2*(x**2))
      EA = EA.subs(z,xi)

      f = f0 + f1*x + f2*x*x
      f=f.subs(z,xi)

      for i in range(0, p+1):
        fg[i+e] = fg[i+e] + f*ShapeValueXi[m][i]*(h/2)*W[m]

        for j in range(0,p+1):
          Kg[i+e][j+e] = Kg[i+e][j+e]+EA*dShapeValueXi[m][i]*dShapeValueXi[m][j]*(2/h)*W[r

  return [Kg, fg]
```

In [55]:

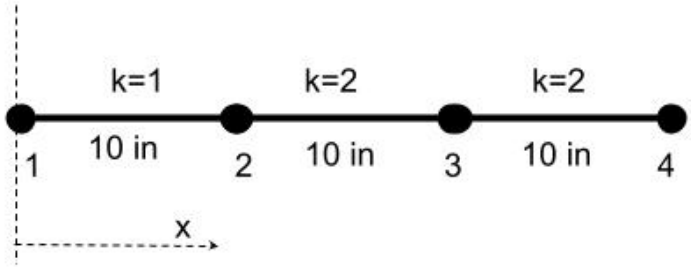
```
print(GlobalKfi.__doc__)
```

```
 =====================================================
   Note: For elements of equal length

   GlobalKfi(k,p,xL,xR,E,a0,a1=0,a2=0,f0=0,f1=0,f2=0)

   k :   number of elements of same size (integer)
   p :   order of approximation (integer)
   xL :  x -coordinate of the left end of the element (real)
   xR :  x -coordinate of the right end of the element (real)
   E :   Young's modules ( E ) of the bar material which is taken as constant (real)
```

```
    a0,a1,a2 : coefficients in (A=a0 + a1*x + a2*(x**2)) for the area of cross-section ( A
) of the bar (real)
    f0,f1,f2 : coefficients in (f = f0 + f1*x + f2*x*x) for the distributed force ( f ) ac
ting on the bar (real)

    Output: [Kg, fg]
    Kg : Global stiffness matrix (real)
    fg : Global force vector (real)
  ==================================================
```

Example:1

Find Global stiffness matrix for each element and also find force vector due to distributed force.
Take shape function as 1st order of approximation(p=1)



Area=6-0.1x
Distributed force (f)=0.0216(6-0.1x)

Solution: Local Stiffeness Matrices (alreday evaluated above)

Global Stiffeness Matrix

1. Equal Elements Length (Methode-1)

In [56]:
```
[Kij, fi]=GlobalKfi(k=3,p=1,xL=0,xR=10,a0=6,a1=-0.1,a2=0,E=30,f0=0.1296,f1=-0.00216)
print("Global Stiffness Matrix: \n\n", Kij,"\n\n")
print("Global Force Vector (due to distributed force):\n\n ", fi)
```

```
Global Stiffness Matrix:

 [[ 16.5 -16.5   0.    0. ]
 [-16.5  30.  -13.5   0. ]
 [  0.  -13.5  24.  -10.5]
 [  0.    0.  -10.5  10.5]]


Global Force Vector (due to distributed force):

  [0.612 1.08  0.864 0.36 ]
```

Function for Global (Elements of different Size or same size) K_{ij} & f_i if 1-D Bar (GlobalKfiU)

- NodesList=Nodes X- Cordinate as list
- k=Number of element
- here,K+1=length(NodesList)

In [57]:

```
def GlobalKfiU(k,p,E,NodesList,a0,a1=0,a2=0,f0=0,f1=0,f2=0):

  """
  =====================================================
Note: For elements of unequal (or equal) length

GlobalKfiU(k,p,E,NodesList,a0,a1=0,a2=0,f0=0,f1=0,f2=0)

k :   number of elements (integer)
p :   order of approximation (integer)
NodesList:  [x0,x1,x2....xk] (X coordinates of nodes)
E :   Young's modules ( E ) of the bar material which is taken as constant (real)
a0,a1,a2 : coefficients in (A=a0 + a1*x + a2*(x**2)) for the area of cross-section ( A ) c
f0,f1,f2 : coefficients in (f = f0 + f1*x + f2*x*x) for the distributed force ( f ) acting

Output: [Kg, fg]
Kg : Globel stiffness matrix (real)
fg : Globel force vector (real)
  =====================================================
  """

  Kg=np.zeros((k*p+1,k*p+1))                          #size of element matrix

  [n, W,xis,ShapeValueXi,dShapeValueXi]=ShapeLegRoot(p)
  fg=np.zeros(k*p+1)

  for e in range(0,k*p,p):
    xL=NodesList[int(e/p)]
    xR=NodesList[int(e/p)+1]
    h=xR - xL

    for m in range(0,n):
      xi=xis[m]
      x = (xL)*(1 - z)/2 + (xR)*(1 + z)/2

      EA = E*(a0 + a1*x+a2*(x**2))
      EA = EA.subs(z,xi)

      f = f0 + f1*x + f2*x*x
      f=f.subs(z,xi)

      for i in range(0, p+1):
        fg[i+e] = fg[i+e] + f*ShapeValueXi[m][i]*(h/2)*W[m]

        for j in range(0,p+1):
          Kg[i+e][j+e] = Kg[i+e][j+e]+EA*dShapeValueXi[m][i]*dShapeValueXi[m][j]*(2/h)*W[

  return [Kg, fg]
```

In [58]:

```
print(GlobalKfiU.__doc__)
```

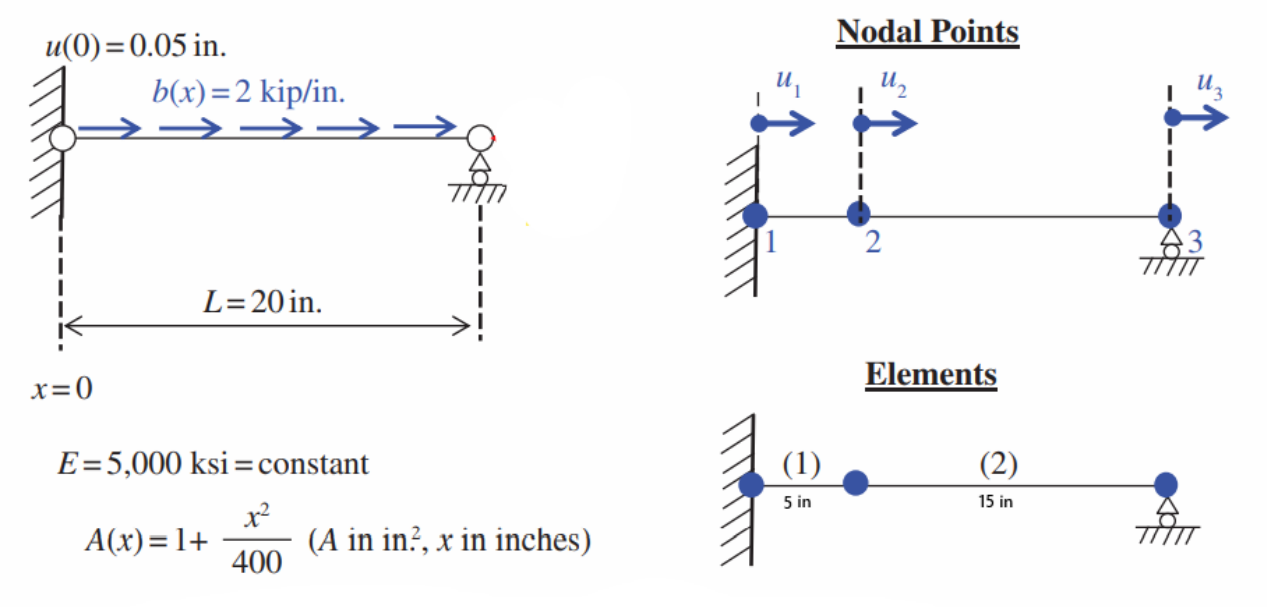
```
  =====================================================
Note: For elements of unequal (or equal) length

GlobalKfiU(k,p,E,NodesList,a0,a1=0,a2=0,f0=0,f1=0,f2=0)

k :   number of elements (integer)
p :   order of approximation (integer)
```

```
NodesList:  [x0,x1,x2....xk] (X coordinates of nodes)
E :   Young's modules ( E ) of the bar material which is taken as constant (real)
a0,a1,a2 : coefficients in (A=a0 + a1*x + a2*(x**2)) for the area of cross-section ( A ) o
f the bar (real)
f0,f1,f2 : coefficients in (f = f0 + f1*x + f2*x*x) for the distributed force ( f ) acting
on the bar (real)

Output: [Kg, fg]
Kg : Globel stiffness matrix (real)
fg : Globel force vector (real)
  ====================================================
```

Example:2

Find Global stiffness matrix for each element and also find force vector due to distributed force.

(take p=1)



Solution

In [59]:
```
[Kij,fi]=GlobalKfiU(k=2,p=1,NodesList=[0,5,20],E=5000,a0=1,a1=0,a2=1/400,f0=2)
print("Global Stiffness Matrix: \n\n", Kij,"\n\n")
print("Global Force Vector (due to distributed force):\n\n ", fi)
```

```
Global Stiffness Matrix:

 [[ 1020.83333333 -1020.83333333     0.        ]
 [-1020.83333333  1500.          -479.16666667]
 [    0.          -479.16666667   479.16666667]]


Global Force Vector (due to distributed force):

  [ 5. 20. 15.]
```

Example:1 (Method-2)

In [60]:
```
[Kij,fi]=GlobalKfiU(k=3,p=1,NodesList=[0,10,20,30],a0=6,a1=-0.1,a2=0,E=30,f0=0.1296,f1=-0
print("Global Stiffness Matrix: \n\n", Kij,"\n\n")
print("Global Force Vector (due to distributed force):\n\n ", fi)
```

```
Global Stiffness Matrix:
```

```
[[ 16.5 -16.5   0.    0. ]
[-16.5  30.  -13.5   0. ]
[  0.  -13.5  24.  -10.5]
[  0.    0.  -10.5  10.5]]


Global Force Vector (due to distributed force):

  [0.612 1.08  0.864 0.36 ]
```

Future scope

- Find unknown reaction force and displacement vector for any bar problem.
- Code can be optimized for low latency.
