High Order One Dimension

For our last project, we created code using linear basis functions and gaussian quadrature to evaluate a variety of one-dimensional phenomena.

1 Modifications to Higher-Degree Polynomials

Modify your code from Project 2 to operate on more general basis functions. We could choose either Lagrange basis functions or Bernstein polynomials, but for this class we will choose Bernstein polynomials. To complete this task, you will need to:

1. Change the function you use to compute basis functions to work on arbitrary polynomial degree. This means that you will need to have your function that evaluates a basis function to additionally take as input polynomial degree. Use the following formula for Bernstein polynomial bases, where $t \in [0, 1]$:

$$N_A^p(t) = \binom{p}{A} t^A (1-t)^{p-A}$$

2. Change the function you use to compute derivatives of basis functions to work on arbitrary polynomial degree. We talked about how to do this in class. If you forgot, just use the product rule on the above equation.

3. Modify your function that compute the mapping from a parent domain to the current element ($x(\xi)$) and your function that computes the derivative of this map ($x_{,\xi}(\xi)$) to operate on the interval $[0, 1]$ rather than $[-1, 1]$.

4. Modify your quadrature routines from operating on the interval $[-1, 1]$ to the interval $[0, 1]$. In particular, you will need to linearly transform all of your quadrature points to be in the new interval, and all integrals must now be scaled by an additional Jacobian (i.e. change of interval) term.

5. Change your ID and IEN functions to include more basis functions per element and more global basis functions given the same number of elements.

6. Modify your local stiffness matrix and your local force vector computations to output a $(p + 1) \times (p + 1)$ matrix and a $(p + 1) \times (1)$ vector, respectively. Particularly, you may need to change the range of your `for` loops.

7. Change any boundary condition information that hard-coded the use of -1 for a point to be evaluated in the local domain to instead operate on the first value of the new interval to be used (i.e. 0).

8. Update your plotting functions to display approximate solutions of the appropriate degree.

To help in this task, a detailed description of how to complete all of the above tasks is described in the appendix. If you need help completing this project, try using the appendix as support. However, you will learn the material better if you do not follow the appendix step-by-step, but instead try to do these operations on your own... at least to start.

2 Convergence Studies

Solving the following problem:

$$(S)\begin{cases}\text{Let } \Omega = (0,1), \bar{\Omega} = [0,1]. \text{ Given } f : \bar{\Omega} \to \mathbb{R} \text{ and constants } g \text{ and } h, \text{ find } u : \bar{\Omega} \to \mathbb{R}, \text{ such that} \\ u_{,xx} + 20\sin(15x) = 0 \quad \text{on } \Omega \\ u(1) = 2 \\ -u_{,x}(0) = 1\end{cases}$$

The true solution to this problem is given by Equation 1.2.3 of the textbook, and is

$$u(x) = 2 + (1-x) - \frac{4}{45}\sin(15) + \frac{4}{3} + \frac{4}{45}\sin(15x) - \frac{4}{3}x.$$

Similarly, its derivative is

$$u_{,x}(x) = \frac{4}{3}\cos(15x) - \frac{7}{3}$$

Recall from page 190 of the text that

$$||e||_m \leq ch^{p+1-m}||u||_{p+1},$$

where the book uses k to denote polynomial degree, but we use p here. Also recall that for uniform element meshes, $h := \frac{1}{n_{elem}}$. Taking the logarithm of both sides gives

$$e_0 := \log\left[\left(\int_\Omega (u^h - u)^2 d\Omega\right)^{\frac{1}{2}}\right] \leq \log(c) + (p+1)\log(h) + \log(||u||_{p+1}) \tag{1}$$

and

$$e_1 := \log\left[\left(\int_\Omega (u^h - u)^2 + (u^h_{,x} - u_{,x})^2 d\Omega\right)^{\frac{1}{2}}\right] \leq \log(c) + p\log(h) + \log(||u||_{p+1}) \tag{2}$$

The Python code below will compute e_0 (output `total_l2_error`) and e_1 (output `total_h1_error`) for you for a given element size and polynomial degree. Now, for a fixed polynomial degree, p, evaluate the error in the solution when solving with $n_{elem} = 2, 4, 8, 16, 32$, and 64 elements. This means that you will have to use this function six times—one of each number of elements. Check the convergence rates of your finite element solution to see that they match those for both the L^2 and H^1 norms (i.e. Equations 1 and 2).

```
def ErrorValues(fullD,xvals,p,quadrature,exact_u,exact_u_derv):

    l2_error_vec = []
    h1_error_vec = []
    n_elem = len(xvals)-1

    quad_wts = quadrature.quad_wts
    quad_pts = quadrature.quad_pts
    n_quad = quadrature.n_quad

    IEN = CreateIENArray(p+1, n_elem)

    for e in range(0,n_elem):
        l2error = 0
```

```
        h1error = 0
        x0 = xvals[e]
        x1 = xvals[e+1]
        for q in range(0,n_quad):
            uh = 0
            uhx = 0
            ksi_q = quad_pts[q]
            x_q = bf.XMap(x0, x1, p, ksi_q)
            w_q = quad_wts[q]
            x_inv = (bf.XMapDerv(x0, x1, p, ksi_q))**-1
            for a in range(0,p+1):
                A = IEN[a,e]
                uh += fullD[A,0] * bf.NBasis(a, p, ksi_q)
                uhx += fullD[A,0] * bf.NBasisDerv(a, p, ksi_q) * x_inv
            l2error += w_q * (uh - exact_u(x_q))**2 * (x1-x0)/2
            h1error += w_q * ((uh - exact_u(x_q))**2 + (uhx - exact_u_derv(x_q))**2) *
                (x1-x0)/2
        l2_error_vec.append(l2error)
        h1_error_vec.append(h1error)

    total_l2_error = (sum(l2_error_vec))**0.5
    total_h1_error = (sum(h1_error_vec))**0.5

    return [total_l2_error,total_h1_error]
```

Plot $\log h$ compared to $\log e_i$, with e_0 being the error in L^2 and e_1 being the error H^1 (i.e. the energy norm). What is the slope of these plots for $p = 1, 2, 3, 4, 5$, particularly between refinements with 32 and 64 elements? Does this coincide with the

3 Evaluations on Previous Project Problems

Solve one of the problems again from Project 1 or 2. Comment on accuracy of your method as you continue to use higher-order polynomials. Is there any point at which refinement in h (element size) or p (polynomial degree) is no longer necessary given your problem conditions?

A How to Modify Code for Higher-Order Polynomials

1. Modify Basis Functions: Before, your code looked like this:

```
# Basis_Function.py
def NBasis(a,x0,x1,x):
    denom = x1-x0
    if a == 0:
        numer = x1-x
    elif a == 1:
        numer = x-x0
    return numer/denom
```

Make p an input before a in the function. Also, we will only be operating on the interval $[0, 1]$, so $x0$ and $x1$ are extraneous: if you would like to include them, consider a google search on how to do so. Import the module math to this Python file. Then for,

$$N_a^p(t) = \binom{p}{a} t^a (1-t)^{p-a}$$

- $\binom{p}{a}$ can be defined using `math.comb(p,a)`
- t^a and $(1-t)^{p-a}$ involves just using powers in Python
- final result is the product of the above terms.

2. Modify the derivative of the Basis Functions: Before, your code looked like this:

```
# Basis_Function.py
def NBasisDerv(a,x0,x1,x):
    denom = x1-x0
    if a == 0:
        numer = -1
    elif a == 1:
        numer = 1
    return numer/denom
```

As before, include p as an input and remove $x0$ and $x1$ unless you wanted to Google how to do transformations for this to work on an arbitrary interval. Next,

$$\begin{aligned}\frac{d}{dt}N_a^p(t) &= \frac{d}{dt}\left(\binom{p}{a} t^a (1-t)^{p-a}\right) \\ &= \binom{p}{a}\left(a \cdot t^{a-1}(1-t)^{p-a} - (p-a)\cdot t^a (1-t)^{p-a-1}\right)\end{aligned}$$

Do similar operations as for the basis function code, but on this new equation. Note that if $a = 0$ the first term in the sum should go away, while if $a = p$, the last term in the sum should. If you are not careful, you may get an error in these cases.

3. Modify Transformation Code: Before, the functions performing $x(\xi)$ and $x_{,\xi}(\xi)$ should have looked something like this:

```
def XMap(x0,x1,ksi):
    x = 0
```

```
    xvals = [x0,x1]
    for a in range(0,2):
        x += NBasis(a,-1,1,ksi) * xvals[a]
    return x

def XMapDerv(x0,x1,ksi):
    x_derv = 0
    xvals = [x0,x1]
    for a in range(0,2):
        x_derv += NBasisDerv(a,-1,1,ksi) * xvals[a]
    return x_derv
```

Now, for `xvals`, we need to split up the interval $[x0, x1]$ into $p+1$ evenly-distributed values between $[x0, x1]$ (e.g. using `numpy`'s `linspace`). Modify the inputs for these functions to accept p as an input. Next, two functions and two x points are no longer in our loop, but $p+1$. Modify the range of the `for` loops accordingly. The actual operations in the loops will stay the same, but the inputs for `NBasis` and `NBasisDerv` have now changed.

4. Change Quadrature Points: Currently, your quadrature points are in the interval between -1 and 1. Use the following function to translate the points only (not the weights) to be between 0 and 1.

$$\xi_g \mapsto \frac{1}{2}(\xi_g + 1)$$

If you use the quadrature functionality that was provided in class, this could be done in the following manner. Regardless of the method that you use, please comment your quadrature method to describe what you do.

```
class GaussQuadrature:

    def __init__(self,n_quad,start = -1,end = 1):
        self.n_quad = n_quad
        self.jacobian = 1
        [self.quad_pts,self.quad_wts] = ComputeQuadraturePtsWts(self.n_quad)
        self.start = start
        self.end = end
        if start != -1 or end != 1:
            self.__TransformToInterval__(start,end)

    def __TransformToInterval__(self,start,end):
        new_pts = []
        for pt in self.quad_pts:
            new_pts.append((end-start)/2 * pt + (start+end)/2)
        self.quad_pts = new_pts
        self.jacobian = (end-start)/2
```

5. Change ID and IEN functions: Before, we had $n_{elem} + 1$ basis functions, where n_{elem} is the number of elements, where the first element introduces two new basis functions and all subsequent elements only introduce one. Now, however, we have $p+1$ new basis functions introduced on the first element and p new basis functions on each subsequent element. Consequently,

there will be $p \cdot n_{elem} + 1$ basis functions in total for the refined space. The length of ID needs to change accordingly. Otherwise, information about the boundary conditions will remain the same.

For IEN, we now need $p+1$ entries per element. Before, our mapping was $IEN(a,e) = a + e$ for basis function $a \in \{0,1\}, e \in \{0, n_{elem} - 1\}$. Now, (e.g. if you draw a degree 2 basis over a few elements, you will find that) $IEN(a,e) = a + p * e$ for $p \geq 1$. Change IEN accordingly. Particularly, if you used the following code,

```
def CreateIENArray(n_basis_function_on_elem,n_elem):
    IEN = np.zeros((n_basis_function_on_elem,n_elem)).astype('int')
    for e in range(0,n_elem):
        for a in range(0,n_basis_function_on_elem):
            Q = a+e
            IEN[a,e]=Q
    return IEN
```

you would change $Q = a + p * e$ and set $p = n_{basis\_function\_on\_element} - 1$ before defining the IEN array.

6. Change local assembly operations:

   - Local Stiffness: Assume that your local stiffness matrix code (without contributions from Robin boundary conditions) looks something like this:

```
def LocalStiffness(e,xvals,n_elem_funcs,bc_left,bc_right,quadrature):
    x0 = xvals[e]
    x1 = xvals[e+1]

    quad_wts = quadrature.quad_wts
    quad_pts = quadrature.quad_pts
    n_quad = quadrature.n_quad

    ke = np.zeros((n_elem_funcs,n_elem_funcs))

    for g in range(0,n_quad):
        w_g = quad_wts[g]
        ksi_g = quad_pts[g]
        x_g = bf.XMap(x0,x1,ksi_g)
        x_derv = bf.XMapDerv(x0,x1,ksi_g)
        x_derv_inv = 1.0/x_derv

        for a in range(0,n_elem_funcs):
            Na_x = bf.NBasisDerv(a,-1,1,ksi_g)
            for b in range(a,n_elem_funcs):
                Nb_x = bf.NBasisDerv(b,-1,1,ksi_g)
                ke[a,b] += w_g * Na_x * Nb_x * x_derv_inv

    # enforce symmetry
    for a in range(0,n_elem_funcs):
        for b in range(a,n_elem_funcs):
            ke[b,a] = ke[a,b]
```

     To modify for higher-degree polynomials, we must

(a) Ensure that we are sampling enough quadrature points to account for the higher polynomial degree

(b) Use our new definitions of `NBasisDerv`

(c) Evaluations happen on the interval from 0 to 1, not from -1 to 1

(d) Summations due to quadrature should be multiplied by 0.5, as indicated in the wikipedia page. If you used the above-indicated quadrature functionality, you should multiply summations due to quadrature by `quadrature.jacobian`.

Similarly, your boundary condition evaluations should operate on parametric elements between 0 and 1, not -1 and 1. If you modified your quadrature routines as suggested above, you may want to use the following functionality in your quadrature class: `quadrature.start` for the left-side evaluations of your interval and `quadrature.end` for the right-side evaluations of your interval.

Note: If you are careful, you will notice that using ***`XMap`*** *and* ***`XMapDerv`*** *as places to use* ***`linspace`*** *operations for sampling the geometry of the interval is expensive and redundant. If you want things to run more quickly, you should move these* ***`linspace`*** *operations out to the element stiffness and force function routines.*

- Local Force: Modifications here will be very similar to those done in the local stiffness method.

7. Change Boundary Condition Points of Evaluation: This should have been accomplished in the previous operation (e.g using `quadrature.start` and `quadrature.end`).

8. Change Method by which Solutions are Plotted: Previously, you probably simply plotted your output vector, D. However, if you try doing this now, you will see that you get very erratic behavior. Instead, recall that

$$u^h(x) = \sum_{P=0}^{n_{basis}} d_P N_P(x).$$

However, on each element, we have

$$u^h\big(x^e(\xi)\big) = \sum_{a=0}^{p+1} d_P N_a(\xi).$$

where P can be found using the IEN array.

Loop through all of the elements and perform the above operation a few times per element at different (increasing) values of ξ. Store both $x^e(\xi)$ and $u^h\big(x^e(\xi)\big)$. Plot the results.