

Lecture 19: Finite Element Methods (2)

Today:

- *Finite element methods for boundary value problems (BVPs)*
 - Implementation, implementation, implementation

Reminder: the FEM equations

For the 1D Poisson problem with zero BCs:

$$\frac{1}{\Delta x} \begin{bmatrix} -2 & 1 & \cdots & 0 & 0 \\ 1 & -2 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & -2 & 1 \\ 0 & 0 & \cdots & 1 & -2 \end{bmatrix} \begin{bmatrix} u_2 \\ u_3 \\ \vdots \\ u_{n-1} \\ u_n \end{bmatrix} = \begin{bmatrix} \int_a^{a+2\Delta x} f(x)b_2(x)dx \\ \int_{a+\Delta x}^{a+3\Delta x} f(x)b_3(x)dx \\ \vdots \\ \int_{a+(n-2)\Delta x}^{a+(n)\Delta x} f(x)b_{n-1}(x)dx \\ \int_{a+(n-1)\Delta x}^{a+(n+1)\Delta x} f(x)b_n(x)dx \end{bmatrix}$$

Punchline: lots of zeros in that matrix; easier to solve than the generic form of (1)

Punchline: Solve matrix system \implies get the u_1, u_2, \dots, u_n that approximate $u(x_1), u(x_2), \dots, u(x_n)$

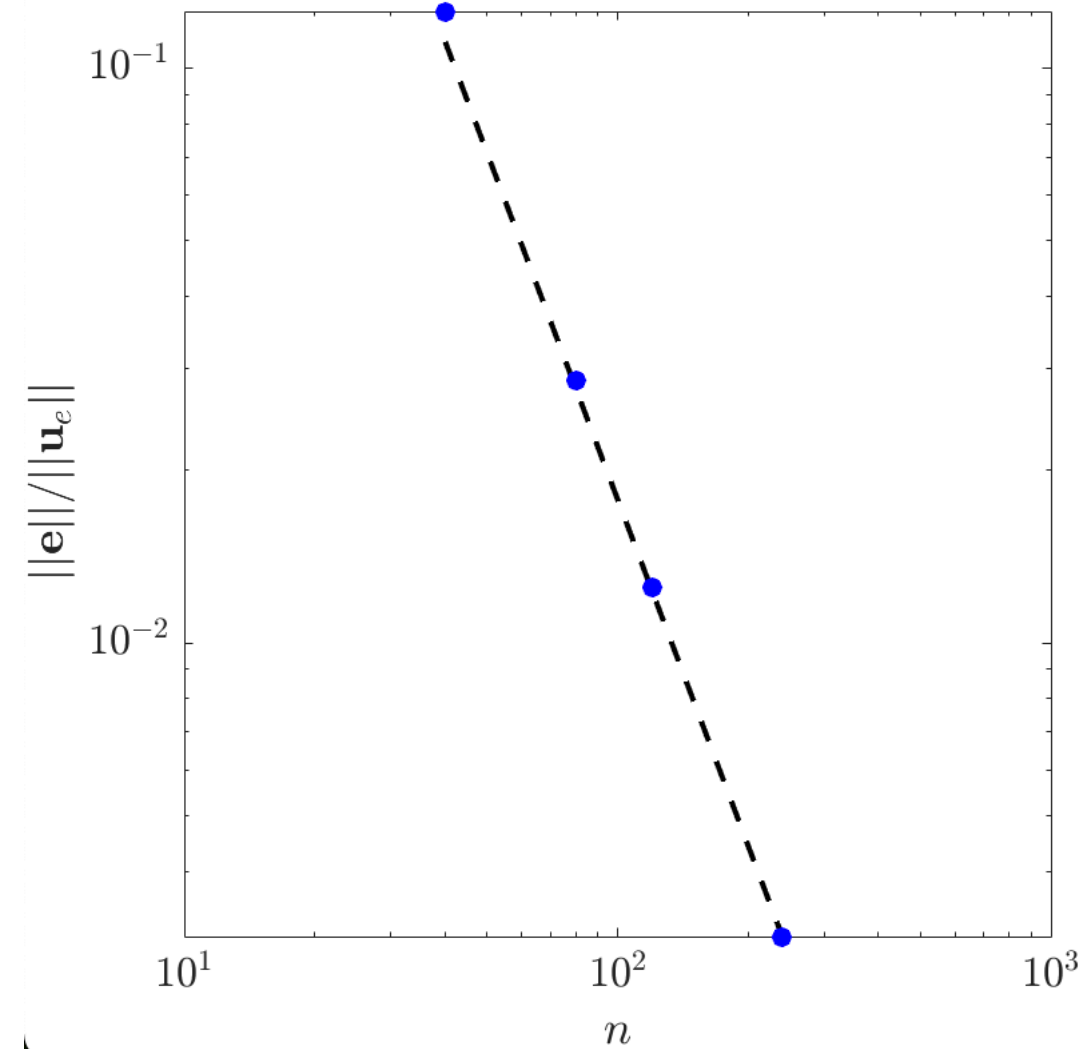
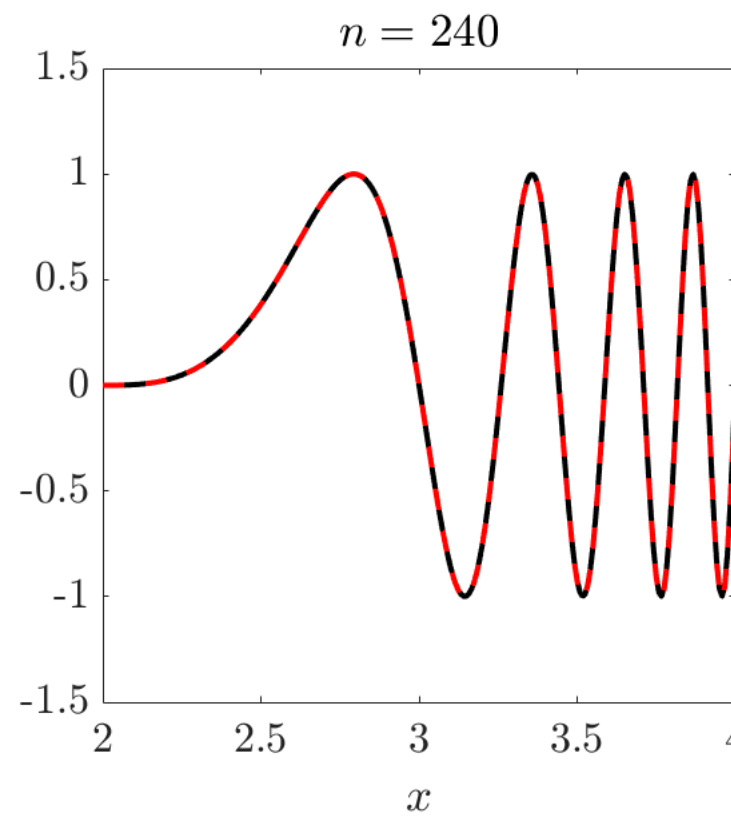
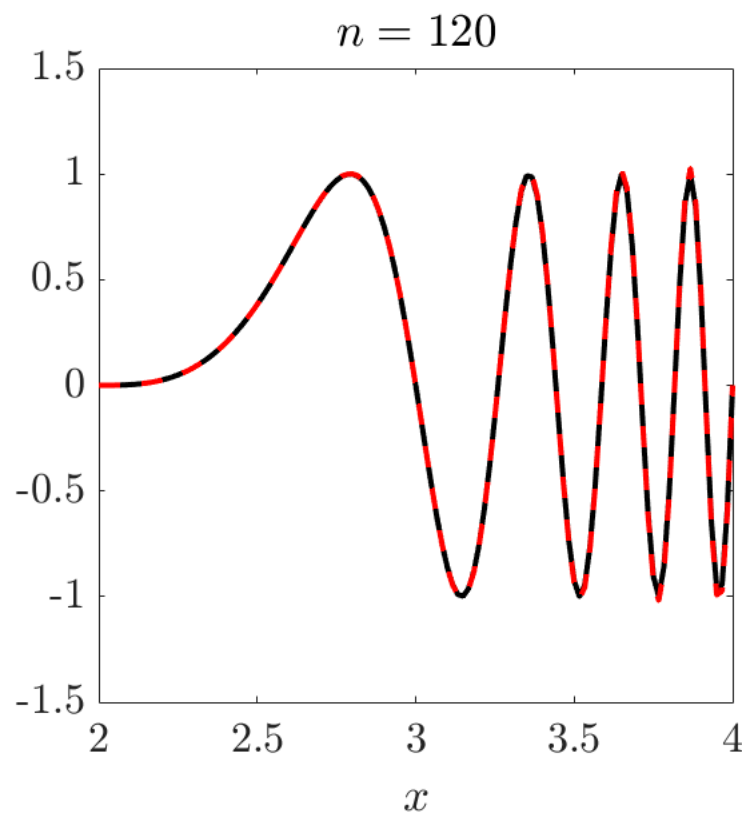
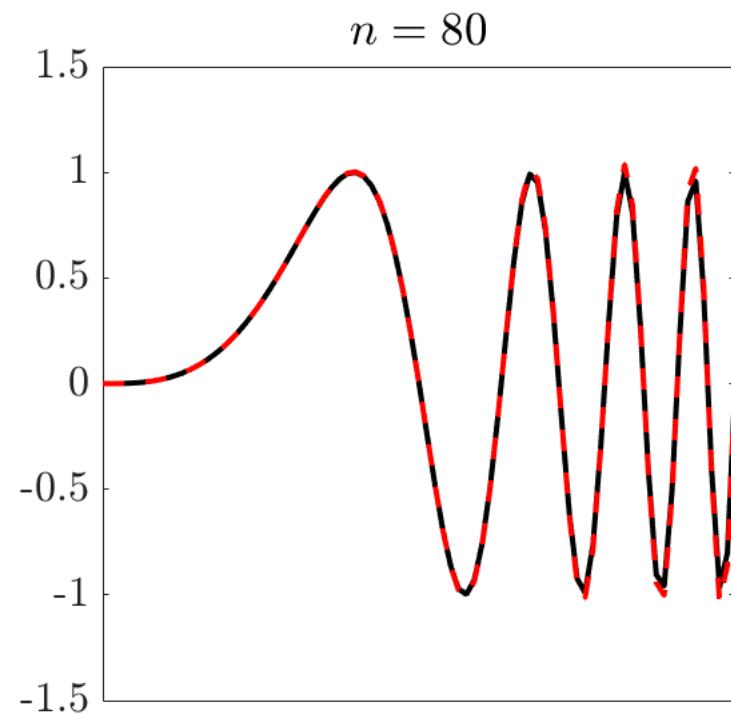
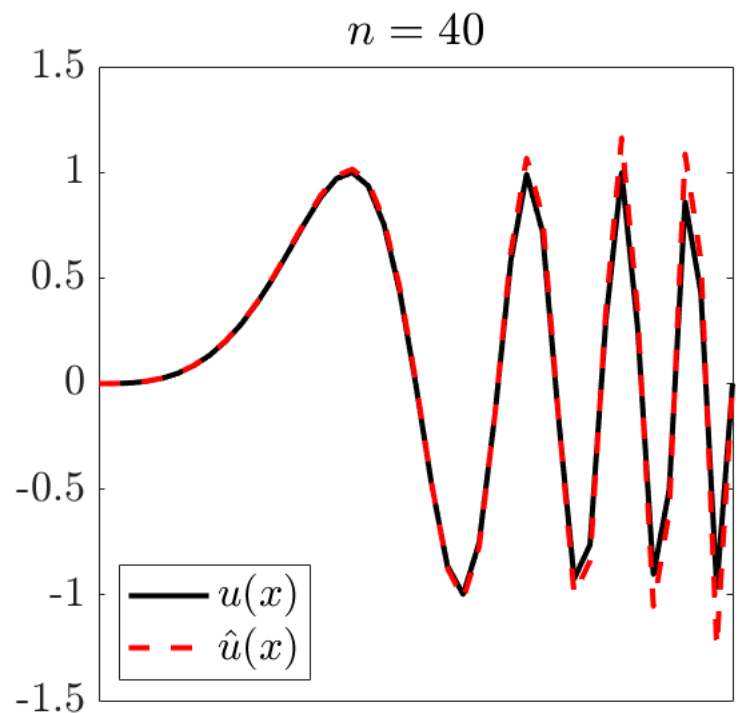
This time: some coding examples and concept discussions!

Implementation: python

```
34 #preamble stuff
35
36 #Interval properties
37 a = 2
38 b = 4
39 L = (b-a)
40
41 #BCs
42 alpha = uex( a, a, b )
43 beta = uex( b, a, b )
44
45 #Consider different n values to see how solution changes as we change n
46 nv = np.array([40, 80, 120, 240])
47
48 #inititalize
49 err = np.zeros([len(nv),1])
50 fig, ax = plt.subplots(len(nv[range(3)]), 1, sharex=True, squeeze=False)
51
52 for j in range(len(nv)):
53     n = nv[j]
54
55     #Build grid
56     dx = float((b-a)/n)
57     xj = np.arange(a,b+dx/2.0,dx)
58
59     #Build A
60     #Diag terms
61     G = np.diag(-2.0*np.ones(n-1)) + np.diag(np.ones(n-2),k=1) + np.diag(np.ones(n-2),k=-1)
62     G *= 1.0/(dx)
63
64     #solve for coeffs
```

```
65     bv = np.zeros(n-1)
66
67     #matrix is diagonal so can solve for each coeff independently
68     for jj in range(n-1):
69         #rhs
70         #could replace trapz with a better quadrature rule!
71         # print(jj)
72         #define x from j-1 to j
73         x_lft = np.arange(a+jj*dx, a+(jj+1)*dx+dx/2.0, dx)
74         #define phi_j from j-1 to j (upward sloping part of phi_j)
75         phi_j_lft = 1/dx * ( x_lft-a-jj*dx )
76         #inner product contribution from j-1 to j
77         bv[jj] = np.trapz( f( x_lft ,a,b ) * phi_j_lft, x=x_lft )
78
79         #define x from j to j+1 (downward sloping part of phi_j)
80         x_rgt = np.arange(a+(jj+1)*dx, a+(jj+2)*dx+dx/2.0, dx)
81         #phi_j from j to j+1
82         phi_j_rgt = -1/dx * ( x_rgt-a-(jj+2)*dx );
83         #inner product contribution from j-1 to j
84         bv[jj] += np.trapz( f( x_rgt,a,b ) * phi_j_rgt, x=x_rgt );
85
86     # solve for coeffs
87     u = LA.solve( G, bv )
88     u = np.concatenate(([alpha], u , [beta] ))
89
90     #get error
91     err[j] = LA.norm(u - uex(xj, a,b))/ LA.norm(uex(xj,a,b))
92
93     #Then do some plotting...
94
```

And the results...



Some concept questions

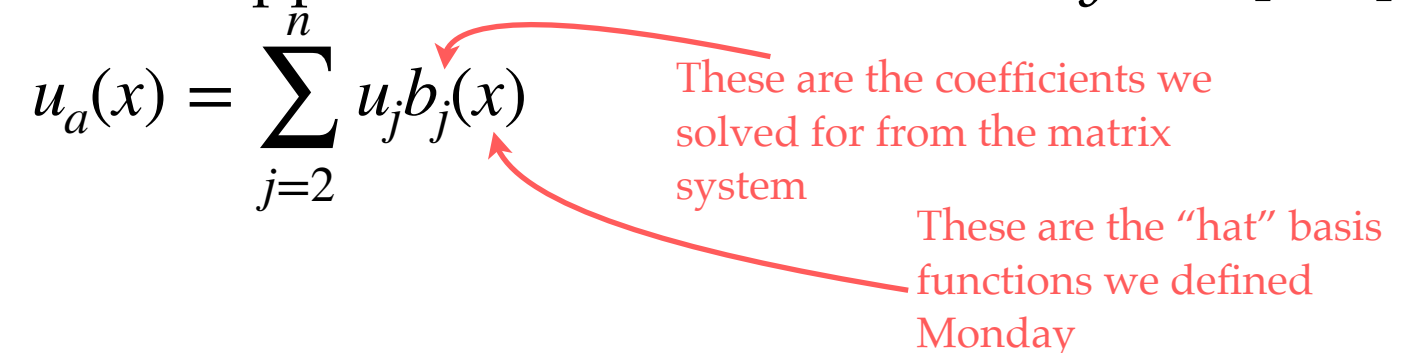
The $n = 40$ plot was jagged because there weren't enough x points to represent the solution with sufficient resolution. Could we have made smoother plots keeping n fixed?

Yes! We have a representation for our approximation function for *any* $x \in [a, b]$

$$u_a(x) = \sum_{j=2}^n u_j b_j(x)$$

These are the coefficients we solved for from the matrix system

These are the "hat" basis functions we defined Monday



We simply plotted u_a evaluated at the *nodes* $x_j, j = 1, \dots, n + 1$

Would this approach make the approximate solution more accurate?

No! The error in the solution scales as $O(\Delta x^2)$, where Δx is the spacing used for the coefficients. We can make the plot smoother by introducing more plotting points, but the error between the true and approximate solution will not change.

How would you change the FEM solution for a different BVP?

Use the right energy inner product and build the matrix system using that energy IP!