Lecture 14: Boundary Value Problems

Today:

- ***Finite difference methods*** applied to ***boundary value problems (BVPs)***

Where are we up to now?

After function approximation, we…

(A) developed ***finite difference methods*** (***one-step*** and ***multi-step***) methods for solving ***initial value problems***

(B) learned how to predict the accuracy of a given method (i.e., how its global error scales) by combining the topics of ***local truncation error*** and ***absolute stability.*** The FD solution converges to the true solution at order p if

(i) the method has a truncation error that scales as $O(\Delta t^p)$

(ii) the method is absolutely stable at $\Delta t = 0$

(C) used the concept of absolute stability to give us a way of determining a Δt that would lead to a solution that does not blow up (***note***: this does not guarantee anything about accuracy until itself!)

This week. Finite difference methods for BVPs.

- Still based on ***local polynomial interpolation,*** but the different nature of the ODE changes the numerical approximation technique

Reminder: what is a BVP?

Before talking about solving BVPs, let's remind ourselves what a BVP is.

Philosophy behind BVPs

Whereas IVPs describe the dynamical response of a system to stimuli/initial conditions, BVPs describe the steady state (or equilibrium) response of a system to forcing.

The canonical BVP we will consider is

$$\alpha \frac{d^2u}{dx^2} + \beta \frac{du}{dx} + \gamma u = f, \quad x \in [a, b]$$

for real constants α and β, γ, boundary points a and b ($a < b$)

Key difference from IVPs: instead of an initial condition, BVPs require ***boundary conditions on both ends*** a, b**.** There are many options, but we will focus on ***Dirichlet conditions*** for now (prescribing the value of the solution at a, b**)**

$$u(a) = u_a, \; u(b) = u_b$$

These are given/prescribed

A simple example BVP: the 1D Poisson problem

We will focus in many of our lectures on developing our numerical method against the simple case when $\alpha = 1$, $\beta = 0$, $\gamma = 0$:

$$\frac{d^2u}{dx^2} = f, \quad x \in [a, b] \qquad (*)$$

(with appropriate ***boundary conditions, BCs,*** such as Dirichlet conditions)

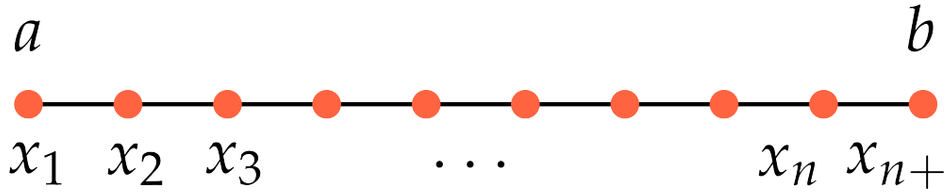
This has a known analytical solution (just integrate both sides twice!)

$\Longrightarrow$ useful test bed for developing and checking our numerical techniques!

Finite difference (FD) methods for BVPs: ***Discretizing*** the domain

Create a uniformly distributed set of points:

$$x_j = a + \frac{(b-a)(j-1)}{n}, \quad j = 1, \ldots, n+1$$



Brain teaser: should we be scared about using uniformly spaced points?

No! We are using ***local*** interpolation, which means we will be dealing with ***low-degree polynomials***

If we were using global interpolation, and requiring high-degree polynomials, this uniform distribution would be problematic

FD methods for BVPs:
Approximating the unknown solution with polynomial interpolation

We will approximate $u(x)$ ***locally*** using a p^{th} order polynomial.

That is, represent $u(x)$ as a degree-p polynomial at each discrete point, $x_j, \;\; j = 1, \ldots, n+1$

For each x_j, we therefore need $p+1$ points to define the interpolant.

We will pick p points in addition to x_j, ***centered*** around x_j

$$\{x_{j-p/2}, \ldots, x_j, \ldots, x_{j+p/2}\}$$

Leads to a class of FD methods called ***centered FD methods***

Centered interpolation points are very common, but not the only option. E.g., could look only to the right of point p and get a ***one side FD method***

Note: requires p is even!

How do we represent $u(x)$ as a degree p polynomial at these $p+1$ points?

Same approach as in function approximation!

Represent as an expansion in the ***Lagrange basis*** in terms of ***unknown coefficients*** that we will solve for!

But… we will use a different polynomial for each x_j point, so we will need to change our notation for the Lagrange basis functions a bit…

FD methods for BVPs:
Defining the Lagrange basis functions centered at x_j

The (j) means this Lagrange function is centered at x_j

$\Longrightarrow$ we call $L_k^{(j)}$ the k^{th} Lagrange function centered at x_j

$$L_k^{(j)}(x) = \prod_{\substack{m=j-p/2 \\ (m\neq k)}}^{j+p/2} \frac{x - x_m}{x_k - x_m} \qquad k = j - p/2, \ldots, j, \ldots, j + p/2.$$

Same general structure as the old formula, but with some differences…

Bounds on the product indices ***do not*** go from 1 to $n + 1$, and ***only*** involve the points used to locally interpolate u at x_j

Same comment for the k indices

These formulae are always scary to work with at first. Let's take an example:

If we wanted to interpolate $u(x)$ locally as a 2^{nd} order polynomial about a point x_5, we would write

$$u(x) = c_4 L_4^{(5)}(x) + c_5 L_5^{(5)}(x) + c_6 L_6^{(5)}(x)$$

for some unknown constants c_4, c_5, c_6 .

And we can write out the Lagrange basis functions as

$$L_4^{(5)}(x) = \frac{(x - x_5)(x - x_6)}{(x_4 - x_5)(x_4 - x_6)} \quad L_5^{(5)}(x) = \frac{(x - x_4)(x - x_6)}{(x_5 - x_4)(x_5 - x_6)} \quad L_6^{(5)}(x) = \frac{(x - x_4)(x - x_5)}{(x_6 - x_4)(x_6 - x_5)}$$

FD methods for BVPs:
Using the Lagrange basis functions to approximate the BVP solution

We use the Lagrange basis functions to represent the BVP solution $u(x)$ locally about each x_j in terms of unknown coefficients that we will solve for:

$$(**) \quad u(x) \approx \sum_{i=j-p/2}^{j+p/2} u_i L_i^{(j)}(x) \qquad \text{(for } j \text{ ranging the number of interpolation points)}$$

Note that the unknown coefficients are u_i, the FD approximation for the solution $u(x_i)$

Why?

Consider evaluating $u(x)$ at one of the interpolation points:

$$u(x_k) \approx \sum_{i=j-p/2}^{j+p/2} c_i L_i^{(j)}(x_k)$$

Remember, the Lagrange basis functions satisfy

$$L_i^{(j)}(x_k) = \begin{cases} 1 & i = k \\ 0 & \text{else} \end{cases}$$

$$= c_k$$

Punchline: the coefficients in the expansion (**) are the approximate values of u at the various interpolation points!

FD methods for BVPs:
Solving for the unknown coefficients $u_1, \ldots, u_{n+1}$ via the FD equations

Plug (**) into the BVP (*) to get

$$\sum_{i=j-p/2}^{j+p/2} u_i \frac{d^2 L_i^{(j)}}{dx^2}\bigg|_{x=x_j} = f(x_j)$$

This gives an equation for each x_j which we can use to solve for the various coefficients $u_1, \ldots, u_{n+1}$

This result is for a general p. It is very common to pick $p = 2$, so we will consider that case from here on out…

For $p = 2$:

(**) reduces to $u(x) \approx \sum_{i=j-1}^{j+1} u_i L_i^{(j)}(x), \quad j = 2, \ldots, n$ (***)

And the Lagrange basis functions are more manageable. For example,

$$L_{j-1}^{(j)}(x) = \frac{(x - x_j)(x - x_{j+1})}{(x_{j-1} - x_j)(x_{j-1} - x_{j+1})}$$
$$= \frac{1}{2\Delta x^2}(x - x_j)(x - x_{j+1})$$

FD methods for BVPs:
Solving for the unknown coefficients $u_1, \ldots, u_{n+1}$ via the FD equations (cont)

Plugging this polynomial-based approximation (***) for $u(x)$ into (*):

$$u_{j-1}\frac{d^2}{dx^2}\left(\frac{1}{2\Delta x^2}(x-x_j)(x-x_{j+1})\right) + u_j\frac{d^2}{dx^2}\left(-\frac{1}{\Delta x^2}(x-x_{j-1})(x-x_{j+1})\right)$$
$$+u_{j+1}\frac{d^2}{dx^2}\left(\frac{1}{2\Delta x^2}(x-x_{j-1})(x-x_j)\right) = f(x_j), \quad j = 2, \ldots, n$$

One can compute the various derivatives of the polynomials to get

$$(****) \qquad \frac{1}{\Delta x^2}\left(u_{j-1} - 2u_j + u_{j+1}\right) = f(x_j), \quad j = 2, \ldots, n$$

We're almost there! We have $n-1$ equations in terms of the $n+1$ unknowns $u_1, \ldots, u_{n+1}$
But notice that (****) doesn't hold for $j = 1$ or $j = n+1$ (e.g., $u_{j-1} = u_0$ when $j = 1$, which doesn't exist)

We use the ***boundary conditions (BCs)*** for the final two unknowns

$$u_1 = u_a, u_{n+1} = u_b$$

FD methods for BVPs:
Assembling the system of equations to solve for the unknowns

We now have $n+1$ equations for the $n+1$ unknowns! We can assemble the equations in matrix form as

This just says $u_1 = u_a$

$$\frac{1}{\Delta x^2}\begin{bmatrix} \Delta x^2 & & & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & -2 & 1 & \\ & & & & 1 & -2 & 1 \\ & & & & & & \Delta x^2 \end{bmatrix}\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{n-1} \\ u_n \\ u_{n+1} \end{bmatrix} = \begin{bmatrix} u_a \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_{n-1}) \\ f(x_n) \\ u_b \end{bmatrix}$$

This just says $u_{n+1} = u_b$

This is an $(n+1)\times(n+1)$ dimensional system

This is perfectly correct, but a bit unnecessary.
Can remove the first and last rows and columns and rearrange to get

$$\frac{1}{\Delta x^2}\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix}\begin{bmatrix} u_2 \\ u_3 \\ \vdots \\ u_{n-1} \\ u_n \end{bmatrix} = \begin{bmatrix} f(x_2) - \frac{u_a}{\Delta x^2} \\ f(x_3) \\ \vdots \\ f(x_{n-1}) \\ f(x_n) - \frac{u_b}{\Delta x^2} \end{bmatrix}$$

This is an $(n-1)\times(n-1)$ dimensional system

What does it look like to code this up?

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.colors import ListedColormap, LinearSegmentedColormap
from numpy import linalg as LA

#--Functions have to be defined up front

#Exact solution
def uex( x, L ):
    om = 1
    return x*np.cos(om*2*np.pi*L*x)

def f( x, L ):
    om = 1
    return - 4*L*om*np.pi*np.sin(2*np.pi*L*om*x) - \
    4*L**2*om**2*x*np.pi**2*np.cos(2*np.pi*L*om*x);

#Use Latex font
plt.rcParams['text.usetex'] = "True"

#enabling math bold font
plt.rcParams['text.latex.preamble'] = r'\usepackage{amsmath,amssymb,bm}'

#Interval properties
a = 2
b = 4
L = (b-a)

#fine grid for pretty plotting of soln
xx = np.linspace(a,b,1000)

#BCs
alpha = uex( a, L )
beta = uex( b, L )
```

```python

    if j < 3:
        ax[j, 0].plot( xj[range(1,n-1)], uj, 'o', label=r'$\hat{\bm{u}}$' )
        ax[j, 0].plot( xx, uex(xx, L), label='$u(x)$' )
        ax[j, 0].set_yticks( np.arange(-5, 10, 5) )
        ax[j, 0].set_yticklabels( np.arange(-5, 10, 5) )
        ax[j, 0].grid( linewidth=0.5 )
        ax[j, 0].set_ylim([-5, 5])
        ax[j, 0].set_xlim([2, 4])
        ax[j, 0].set_ylabel( '$n = %i$'%n )

        if j == 0:
            ax[j,0].legend(loc = 'lower left')
        elif j == 2:
            ax[j, 0].set_xlabel( '$x$' )
            ax[j, 0].set_xticks( np.arange(2, 4.5, 0.5) )
            ax[j, 0].set_xticklabels( np.arange(2, 4.5, 0.5) )


plt.tight_layout()
# plt.show()
plt.savefig('FD_1D_PoissDense_soln.png', dpi = 400)

```

```python
#Consider different n values to see how solution changes as we change n
nv = np.array([10, 20, 40, 80, 160, 320])

fig, ax = plt.subplots(len(nv[range(3)]), 1, sharex=True, squeeze=False)

err = np.zeros([len(nv),1])

for j in range(len(nv)):

    n = nv[j]

    #Build interp points
    xj = ( a + (b-a)*np.arange(n)/(n-1) )

    #grid spacing
    dx = xj[1]-xj[0]

    #Build A
    #Diag terms
    ind = range(n-2)
    A = np.diag(-2*np.ones(n-2)) + np.diag(np.ones(n-3),k=1) + np.diag(np.ones(n-3),k=-1)

    A = 1/(dx**2) * A

    #Build RHS
    fv = f( xj[range(1,n-1)], L )
    fv[0] = fv[0] - alpha/(dx**2)
    fv[n-3] = fv[n-3] - beta/(dx**2)

    uj = LA.solve( A, fv )


    errtmp = np.sqrt(dx) * LA.norm((uj - uex(xj[range(1,n-1)], L) ) )
    err[j] = errtmp

```