APPM 4600 Lab 8
Playing with splines

1 Overview

Splines are piecewise approximations. In class, we derived linear and cubic splines. In this lab, you will start developing your code for creating these approximations. You will build it in small pieces strengthening your coding skills.

2 Before lab

Write a subroutine that constructs and evaluates a line that goes through the points $(x_0, f(x_0))$ and $(x_1, f(x_1))$ at a point α.

3 Lab Day: Building splines

During lab, you will build both a linear and cubic spline codes.

3.1 Constructing piecewise linear approximations

In this section, you will take your pre-lab codes and build a linear spline code. Below is a Python code that is missing the two subroutines that you made previously. The calls to your previously created subroutines in the subroutine named `eval_lin_spline`. You can download the template as a .py file from the lab assignment.

```
import matplotlib.pyplot as plt
import numpy as np
import math
from numpy.linalg import inv


def driver():

    f = lambda x: np.exp(x)
    a = 0
    b = 1

    ''' create points you want to evaluate at'''
    Neval = 100
    xeval =  np.linspace(a,b,Neval)

    ''' number of intervals'''
    Nint = 10

    '''evaluate the linear spline'''
    yeval = eval_lin_spline(xeval,Neval,a,b,f,Nint)
```

```
    ''' evaluate f at the evaluation points'''
    fex = f(xeval)

    plt.figure()
    plt.plot(xeval,fex,'ro-')
    plt.plot(xeval,yeval,'bs-')
    plt.legend()
    plt.show

    err = abs(yeval-fex)
    plt.figure()
    plt.plot(xeval,err,'ro-')
    plt.show

def  eval_lin_spline(xeval,Neval,a,b,f,Nint):

    '''create the intervals for piecewise approximations'''
    xint = np.linspace(a,b,Nint+1)

    '''create vector to store the evaluation of the linear splines'''
    yeval = np.zeros(Neval)


    for jint in range(Nint):
        '''find indices of xeval in interval (xint(jint),xint(jint+1))'''
        '''let ind denote the indices in the intervals'''

        atmp = xint[j]
        btmp= xint[j+1]

#   find indices of values of xeval in the interval
        ind= np.where((xeval >= atmp) & (xeval <= btmp))
        xloc = xeval[ind]
        n = len(xloc)

        '''temporarily store your info for creating a line in the interval of
         interest'''
        fa = f(atmp)
        fb = f(btmp)

        yloc = np.zeros(len(xloc))
        for kk in range(n):
        #use your line evaluator to evaluate the spline at each location
            yloc[kk] = #Call your line evaluator with points (atmp,fa) and (btmp,fb)

#       Copy yloc into the final vector
        yeval[ind] = yloc
```

```
    return yeval

driver()
```

3.2 Exercise

Consider the function

$$f(x) = \frac{1}{1 + (10x)^2}$$

on the interval $[-1, 1]$. Perform the same experiments as in Lab 7 but with your linear spline evaluator. How does this perform? Is it better or worse than global interpolation with uniform nodes?

3.3 Constructing cubic splines

Download the `demo_cubicspline.py` code. Like the linear spline code, portions of the code need to be filled to complete the code. In this lab, you will create the natural spline. In your homework, you will modify this code to create the clamped spline.

1. In the subroutine `create_natural_spline` there are several pieces missing. First you must create the matrix needed in order to identify the $\{M_i\}_{i=0}^{n-1}$ coefficients.

2. Next, use the linear algebra package in Numpy (`inv` loaded at the top of the Python code) to solve for these coefficients.
   Make sure to validate that your coefficients have been evaluated correctly.

3. Evaluate the coefficients $\{C_i\}_{i=0}^{n-1}$ and $\{D_i\}_{i=0}^{n-1}$ using the now computed $\{M_i\}_{i=0}^{n-1}$ values.

4. The subroutine `eval_local_spline` evaluates a spline defined on a subinterval. It takes as input the locations to evaluate the polynomial, the endpoints of the interval and the spline coefficients. The subroutine is missing the cubic polynomial evaluator. Add this so that your cubic spline code is complete.

3.4 Exercise

Consider the function

$$f(x) = \frac{1}{1 + (10x)^2}$$

on the interval $[-1, 1]$. Perform the same experiments as in Lab 7 but with your cubic spline evaluator. How does this perform compared to all the other methods.

4 Deliverables

Report your solutions to the questions in the exercise section on Canvas, including some plots. Push your codes to Git as usual.

5 The incomplete cubic spline code

```
import matplotlib.pyplot as plt
import numpy as np
import math
from numpy.linalg import inv
from numpy.linalg import norm


def driver():

    f = lambda x: np.exp(x)
    a = 0
    b = 1

    ''' number of intervals'''
    Nint = 3
    xint = np.linspace(a,b,Nint+1)
    yint = f(xint)

    ''' create points you want to evaluate at'''
    Neval = 100
    xeval =  np.linspace(xint[0],xint[Nint],Neval+1)

#   Create the coefficients for the natural spline
    (M,C,D) = create_natural_spline(yint,xint,Nint)

#  evaluate the cubic spline
    yeval = eval_cubic_spline(xeval,Neval,xint,Nint,M,C,D)


    ''' evaluate f at the evaluation points'''
    fex = f(xeval)

    nerr = norm(fex-yeval)
    print('nerr = ', nerr)

    plt.figure()
    plt.plot(xeval,fex,'ro-',label='exact function')
    plt.plot(xeval,yeval,'bs--',label='natural spline')
    plt.legend
    plt.show()

    err = abs(yeval-fex)
    plt.figure()
    plt.semilogy(xeval,err,'ro--',label='absolute error')
    plt.legend()
```

```
    plt.show()

def create_natural_spline(yint,xint,N):

#    create the right  hand side for the linear system
    b = np.zeros(N+1)
#  vector values
    h = np.zeros(N+1)
    h[0] = xint[i]-xint[i-1]
    for i in range(1,N):
       h[i] = xint[i+1] - xint[i]
       b[i] = (yint[i+1]-yint[i])/h[i] - (yint[i]-yint[i-1])/h[i-1]

#  create the matrix A so you can solve for the M values
    A = np.zeros((N+1,N+1))

#  Invert A
    Ainv =

# solver for M
    M  =

#  Create the linear coefficients
    C = np.zeros(N)
    D = np.zeros(N)
    for j in range(N):
       C[j] = # find the C coefficients
       D[j] = # find the D coefficients
    return(M,C,D)

def eval_local_spline(xeval,xi,xip,Mi,Mip,C,D):
# Evaluates the local spline as defined in class
# xip = x_{i+1}; xi = x_i
# Mip = M_{i+1}; Mi = M_i

    hi = xip-xi

    yeval =
    return yeval


def  eval_cubic_spline(xeval,Neval,xint,Nint,M,C,D):

    yeval = np.zeros(Neval+1)

    for j in range(Nint):
        '''find indices of xeval in interval (xint(jint),xint(jint+1))'''
```

```
        '''let ind denote the indices in the intervals'''
        atmp = xint[j]
        btmp= xint[j+1]

#   find indices of values of xeval in the interval
        ind= np.where((xeval >= atmp) & (xeval <= btmp))
        xloc = xeval[ind]

# evaluate the spline
        yloc = eval_local_spline(xloc,atmp,btmp,M[j],M[j+1],C[j],D[j])
#   copy into yeval
        yeval[ind] = yloc

    return(yeval)

driver()
```