midterm_numanalysis

July 27, 2021

1 Numerical Analysis Midterm Project

In the midterm coursework, we investigate the design and application of approximate schemes for the expression of derivatives of functions on equispaced grids. This will involve polynomial interpolation, transformation of polynomial coefficients and the linear combination of lower-order techniques to achieve better accuracy. In a final application, we use these approximations to solve a Sturm-Liouville-type ordinary differential equation.

1.0.1 Submission (PLEASE READ)

Please fill out the missing parts of this jupyter notebook following the steps below and submit the final notebook via Blackboard. Your submission should have a clear description of your work (using markdown cells = text boxes between the codes (like this one)). Also, augment your code with ample comments (using the #-sign in python). When verifying your code, include the testing in your submission. The submission deadline is Friday, 26. February 17:00 (UK Time).

First, we need to prepare a few routines, related to polynomial interpolation. Each of the steps below should be benchmarked and validated, before moving to the next step.

STEP 1 : Write a python function that computes an interpolating polynomial through a set of points (x_i, y_i) in $\mathbb{R}^2$. The routine should output the coefficients of the interpolating polynomial.

[1]:
```
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt
%matplotlib inline



def multiply(f,g):
    """Takes two lists, represting polynomials, and outputs the list␣
 ↪representing the product of the polynomials."""
    m = len(f)
    n = len(g) # list elements are in order of respective coefficient degree,␣
 ↪starting from lowest.
    mat = np.zeros((m,n))
    for i in range(m):
        for j in range(n):
            mat[i,j] = f[i]*g[j]
            #setup matrix such that i j'th element is the product of of the␣
↪i'th term in f and j'th term in g
    diags = [mat[::-1,:].diagonal(i) for i in range(-m+1,n)]
    return [sum(i.tolist()) for i in diags] # by finding the sum across␣
↪diagonals we can find the coefficients of each term
multiply([2,1],[-2,1])

def LPoly(x,y):
    """Takes in data sets x, y and returns interpolating polynomial as a list.
↪"""
    n = len(x)
    l_list = [0 for i in range(n)] #empty list, this will contain n l_i␣
↪functions
    for i in range(n):
        l = [1] + [0 for i in range(n-1)] # set l = 1
        for j in range(n):  # to work out l_i we workout the product of our␣
↪factors(which can be seen in the formula below)
            if i != j: # don't include i = j or we will divide by 0
                l = multiply(l, [-x[j]/(x[i]- x[j]), 1/(x[i]- x[j])]) #␣
↪multiply by (z - x[j])/(x[i]- x[j])
        l_list[i] = l # input into list
    polyprod = [np.array(l_list[i])*y[i] for i in range(n)]
    coeff = list(sum(polyprod))  # multiplying langrange basis by output and␣
↪adding
    while coeff[-1] == 0:
        coeff.pop()  # gets rid of 0 terms at the end of list
    return coeff
```

[2]:
```
LPoly([0,1,4], [4, 3,12])
# f = 4 - 2x + x^2
# f(0) = 4
# f(1) = 3
# f(4) = 12
```

[2]:
```
[4.0, -2.0, 1.0]
```

[3]:
```
LPoly([0,2,3,5],[1,-3,7,261])
# By plugging x = 0, 2, 3 , 5 into x^4 -3x^3+2x + 1 we get 1, -3, 7, 261
```

[3]:
```
[1.0, 32.0, -31.000000000000007, 7.0]
```

Let our points be x_i, y_i for i = 0 to n-1, so we have n pairs. For any i, $l_i = \prod_{k=0,k\neq i}^{n-1} \frac{x - x_k}{x_i - x_k}$ If P_{n-1}

is our interpolating polynomial, $P_{n-1} = \sum_{i=0}^{n-1} l_i y_i$

With the example data points I use, we can sub in our values of x values into the polynomial and we will get corresponding y-values.

STEP 2 : Next, we need a routine that converts a set of coefficients of a polynomial $P(x)$ (i.e., the output from the routine in Step 1) and returns the set of coefficients of the derivative of $P(x)$.

```
[5]: def LprimePoly(c):
         """Derivative of polynomial as a list of coefficients."""
         #Assume first coefficient is degree 0 term
         if len(c) == 1:
             return [0] # if polynomial is degree 0, coefficient list is length 1␣
      ↪and its derivative is 0.
         else:
             return [c[i+1]*(i+1) for i in range(len(c)-1)] # n length array becomes␣
      ↪n-1 length array, a x^(b) becomes a*b x^(b-1)
```

```
[6]: LprimePoly([4,2,1,4,10])
     # Derivative of    4 + 2x + x^2 + 4x^3 + 10x^4    is     2 + 2x + 12x^2 + 40x^3
```

```
[6]: [2, 2, 12, 40]
```

```
[7]: LprimePoly([7])
     #Derivative of 7 is 0
```

```
[7]: [0]
```

STEP 3 : Finally, we implement a routine that evaluates a polynomial, given by its coefficients, at a specific point x_0, returning the value $P(x_0)$.

```
[8]: def EvalPoly(c,x0):
         """Evaluate polynomial c at point x = x0"""
         y0 = sum([x0**i * c[i] for i in range(len(c))])
         return y0
```

```
[9]: EvalPoly([1,2,3], 2)
     # 1 + 2x + 3x^2 evaluated at x=2 returns 17
```

```
[9]: 17
```

```
[10]: EvalPoly([-1,4,4,4,4], -1)
      # -1 + 4x + 4x^2 + 4x^3 + 4x^4 evaluated at x = -1 returns -1
```

```
[10]: -1
```

STEP 4 : Assume we have a set of data points $\{x_i, f_i\}$ either from a known function $f(x)$ or simply as a set of measurements. We wish to approximate the discrete data set by a continuous representation, so that we can take approximate derivatives up to a given order. Instead of fitting one polynomial through all data points, we consider a small set of data points in the neighborhood of a specified point ξ where we wish to compute our derivative. We then fit an interpolating polynomial through this small set of points, differentiate this polynomial and evaluate the derivative at ξ. We take this final value as the approximate derivative of our data set at $x = \xi$. By looping through multiple values of ξ we can compute an approximate derivative of our entire data set.

More specifically, given a set of points $\{x_0, x_1, ..., x_n\}$, an evaluation point ξ, and an integer parameter m which specifies the order of the derivative (e.g., $m = 2$ for the second derivative), we wish to determine the weights w_i such that

$$f^{(m)}(\xi) = \left.\frac{d^m f}{dx^m}\right|_\xi \approx \sum_{i=0}^{n} w_i f_i, \tag{1}$$

i.e., the m-th derivative of a function $f(x)$ at the point ξ, expressed as a weighted sum of the function values $f_i = f(x_i)$. Using the three auxiliary routines above, determine the weight coefficients $\{w_i\}_{i=0}^n$.

For simplicity, let us assume that the points $\{x_i\}$ are equispaced, i.e., they can be described by $h \times \{0, 1, 2, ..., n\}$ with h as the grid spacing. Note that the parameter n can be inferred from the length of the x-values we pass into the routine. Also, the weights we compute should be rendered independent of the parameter h, i.e., we assume $h = 1$ when computing the weights. For different spacings h, we simply have to divide the computed weights by h^m for the m-th derivative.

Complete the function below to compute the weights w for the m-th derivative evaluated at ξ and given by function values at the locations x.

[11]:
```
def Weights(x):
    """Compute Lagrange basis from data set x"""
    n = len(x)
    l_list = [0 for i in range(n)]
    for i in range(n):
        l = [1] + [0 for i in range(n-1)] # set l = 1
        for j in range(n):  # to work out l_i we workout the product of our␣
 ↪factors(which can be seen in the formula below)
            if i != j: # don't include i = j or we will divide by 0
                l = multiply(l, [-x[j]/(x[i]- x[j]), 1/(x[i]- x[j])]) #␣
 ↪multiply by (z - x[j])/(x[i]- x[j])
        l_list[i] = l # input into list
    return l_list

def PolyDeriv(poly, m):
    """Differentiate polynomial m times."""
    if m == 0 :
        return poly
    else:
        return PolyDeriv(LprimePoly(poly), m-1)
```

```
def compFD(x,xi,m=1):
    # input x: array of x-values for the interpolating polynomial
    #       xi: evaluation point for derivative
    #       m: order of derivative (e.g. m=2 for second derivative)
    # output w: array of weight coefficients

    weights = Weights(x)
    return [EvalPoly(PolyDeriv(a,m),xi) for a in weights]
```

[12]:
```
compFD([5, 6, 9, 11],5,1)
```

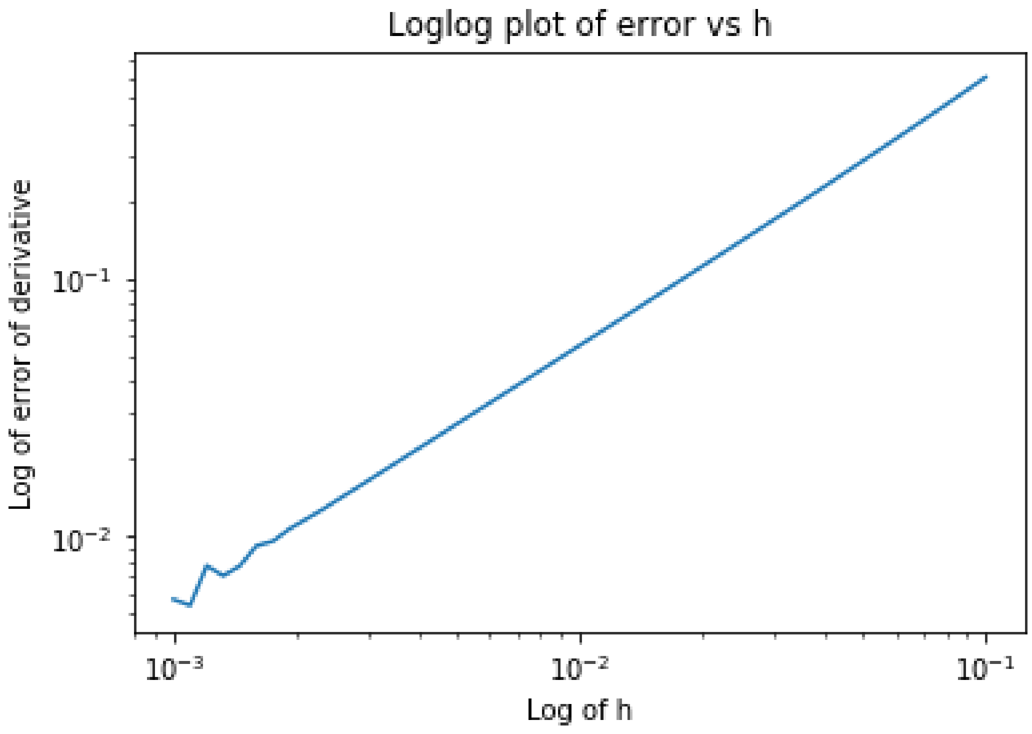
[12]:
```
[-1.4166666666666679, 1.6000000000000014, -0.25, 0.06666666666666643]
```

For some $\epsilon \in [x_0, x_n]$, $w_i = l_i(\epsilon)$, where l_i are the functions from step 1 that we use to work out our interpolating polynomial

STEP 5 : Design differentiation weights w_i for a first derivative using five equispaced grid points $\{x_0, x_1, x_2, x_3, x_4\}$ with $\xi = x_2$. Test the accuracy by using the formula on the function $\exp(x)$ for $x = 1$. Compute and plot the error (absolute difference between the approximate and exact derivative) as a function of the grid spacing $h \in [10^{-3}, 10^{-1}]$. The error will follow a power-law $e \sim h^p$. Determine the exponent p.

[14]:
```
hlist = 10**np.linspace(-3,-1,50) #contains powers of 10, where power is␣
 ↪inbetween -3 and -1

def xvals(h):
    return np.array([1+ h*(i) for i in range(5)]) # this function returns the x␣
 ↪values of the grid points when h is given

error = []
for i in hlist:
    deriv_weights = compFD(xvals(i),1+2*i,1)
    deriv = np.dot(deriv_weights, np.exp(xvals(i)))
    error.append(abs(deriv - np.exp(1)))
plt.xlabel("Log of h")
plt.ylabel("Log of error of derivative")
plt.title("Loglog plot of error vs h")
plt.loglog(hlist, error)
```

[14]:
```
[<matplotlib.lines.Line2D at 0x27ad575c0c8>]
```



[15]:
```
slope, intercept = np.polyfit(np.log(hlist), np.log(np.array(error)), 1)
print (f"The slope is equal to {slope} which is roughly equal to 1.")
```

```
The slope is equal to 1.0143128244748667 which is roughly equal to 1.
```

Note that if e is roughly proportional to h^p for some p, then log(e) is proportional to logh. We do a log plot of the error against h then the gradient of the graph will represent p. Our graph shows some strange behaviour near h = 0.001, but other than that the gradient seems to be constant and close to 1 which is what we desire.

STEP 6 : More accuracy can be gained by moving to compact finite-difference schemes.

Compact schemes express a linear combination of derivatives as a linear combination of function values. For example, for a first derivative we have

$$\sum_{j=0}^{m} W_j f'_j = \sum_{j=0}^{n} w_j f_j \tag{2}$$

which uses a different set of grid points to express the derivative values and the function values. These two sets of grid points do not necessarily have to coincide or even intersect. Mathematically, the above approximation is equivalent to expressing the derivative by an interpolation by a rational polynomial, rather than a standard polynomial (as in the finite-difference case above).

The procedure to determine the weights goes as follows. We introduce fictitious points to the grid points on the right-hand (function) side and determine the finite-difference weights for each of the grid points on the left-hand (derivative) side. We then compute a linear combination of these finite-difference weights such that the weights for the fictitious points are zero. This procedure then gives us the compact-difference weights $\{W_j\}$ and $\{w_j\}$.

Let us demonstrate this technique by considering the compact-difference formula for the second derivative that uses the three points $\{x_0, x_1, x_2\}$ for both the function and its derivative, i.e.,

$$W_0 f_0'' + W_1 f_1'' + W_2 f_2'' = w_0 f_0 + w_1 f_1 + w_2 f_2. \tag{3}$$

We add two fictitious points to the right (function) grid points: x_3 and x_4, and compute the finite-difference weights for each of the grid points on the left (derivative) side. We get (for a grid spacing of $h = 1$)

$$\begin{aligned} f_0'' &= \tfrac{35}{12} f_0 - \tfrac{26}{3} f_1 + \tfrac{19}{2} f_2 - \tfrac{14}{3} f_3 + \tfrac{11}{12} f_4, \\ f_1'' &= \tfrac{11}{12} f_0 - \tfrac{5}{3} f_1 + \tfrac{1}{2} f_2 + \tfrac{1}{3} f_3 - \tfrac{1}{12} f_4, \\ f_2'' &= -\tfrac{1}{12} f_0 + \tfrac{4}{3} f_1 - \tfrac{5}{2} f_2 + \tfrac{4}{3} f_3 - \tfrac{1}{12} f_4. \end{aligned} \tag{4}$$

With these results, we compute a linear combination of the three expressions, such that the coefficients for the f_3- and f_4-terms are zero. This leads to the compact-difference scheme

$$f_0'' + 10 f_1'' + f_2'' = 12 f_0 - 24 f_1 + 12 f_2 \tag{5}$$

which is the desired approximation using the points $\{x_0, x_1, x_2\}$ for both the derivatives and the function values.

Using your finite-difference function from step 4, write a python function that computes the weights for a compact-difference scheme based on fictitious points for arbitrary grid points for the function values (xRHS) and derivative values (xLHS) and for a given derivative order m.

The python function should take as input the grid point location for the left- and right-hand side, as well as the order of the derivative. As output we want the corresponding weights for the grid points on the derivative (left) and function (right) side of the above expression.

[17]:
```python
def compCD(xLHS,xRHS,m):
    # input xLHS: array of x-values on the left-hand side (derivative side)
    #       xRHS: array of x-values on the right-hand side (function side)
    #       m: order of derivative
    # output: wLHS: array of weights for the left-hand side (derivative side)
    #         wRHS: array of weights for the right-hand side (function side)

    i = len(xLHS) # num of eqns (derivatives)
    j = len(xRHS) # num of variables (function points)

    new_xRHS = np.roll(xRHS, i-1)
    #will eliminate i-1 variables so we shift by (i-1) to the right to make␣
 ↪elimination easier

    A = np.array([compFD(new_xRHS,xlhs,m) for xlhs in xLHS])
    # Create our matrix representing a system of equations
    Q, R =  np.linalg.qr(A) # q-r decomposition of our matrix
    wLHS = Q.T[-1]  # Take last row of tranpose of Q (because this moves to the␣
↪LHS of the equation)
    wRHS = R[-1,i-1:] #Take last row of R and remove first i elements


    u1 = abs(min(wLHS))  # this section of code gets rid of some fractions/
↪decimals if possible
    u2 = abs(min(wRHS))
    u = min(u1, u2)
    if u != 0:
        u1 = min(wLHS)
        wLHS = wLHS/u
        wRHS = wRHS/u

    return wLHS, wRHS
```

Validate your program on the example given above. Then, compute the compact-difference scheme (i) for a second derivative with xLHS = (-1,0,1), xRHS = (-2,-1,0,1,2) and (ii) for a first derivative with xLHS(-3/2,-1/2,1/2,3/2), xRHS = (-2,-1,0,1,2). In both cases, assume a grid spacing of $h = 1$.

[65]:
```
a, b = compCD([1, 2, 3], range(1, 6), 2)
print (a, b)
# When assuming h = 1, we able to get the equation found in the example.

c, d = compCD([-1,0,1],[-2,-1,0,1,2],2)
print (c,d)

e, f = compCD([-3/2,-1/2,1/2,3/2],[-2,-1,0,1,2],1)
print (e/min(abs(e)),f/min(abs(e)))

Y, Z = np.array([1,2])
print (Y)
```

```
[ 1. 10.  1.] [ 12. -24.  12.]
[ 6.5 -1.   0.5] [  6. -12.   6.]
[-26.   5.  -4.   1.] [ 24. -24.]
1
```

STEP 7 : Use the schemes from the finite-difference routine compFD(x,xi,m) for $x = \{-1, 0, 1\}$ and $\xi = 0$ to solve the ordinary differential equation

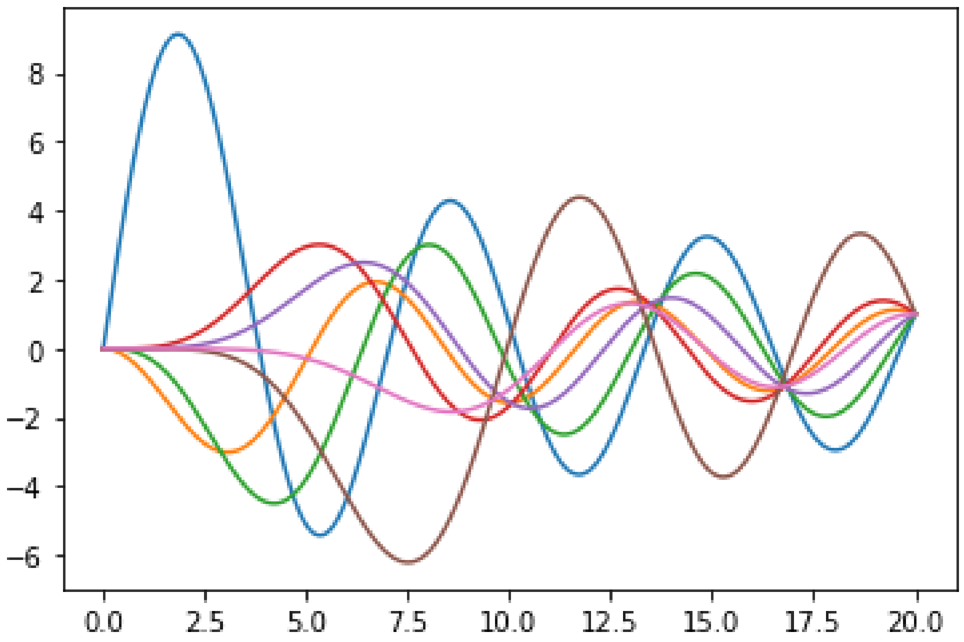
$$x^2y'' + xy' + (x^2 - \nu^2)y = 0 \qquad y(0) = 0,\ y(20) = 1, \quad x \in [0, 20] \tag{6}$$

for $\nu = 1, 2, 3, 4, 5, 6, 7$.

The unknown function $y(x)$ will be represented on an equispaced grid with n points (n as a user-specified parameter). Use the finite-difference weights to express the second and first derivatives in terms of the unknown function values at the grid points. Recast the ordinary differential equation into a linear system and solve for the solution $y(x)$ on the grid points.

Visualize your results for $\nu = 1, 2, ..., 7$.

[89]:
```
from scipy.linalg import null_space

def BVP_ODE(n,nu):
    # input n: number of (equispaced) grid points on the interval [0,20]
    #       nu: value for parameter nu
    # output x: array of grid values x
    #        y: array of function values y on the grid x

    grid = np.linspace(0,20,n)
    h = 20/n #spacing inbetween grid points
    w1 = np.array(compFD([-1,0,1],0,1))/h  # df/dx |x=x_i weights in terms of␣
↪function values at, before and after x_i
    w2 = np.array(compFD([-1,0,1],0,2))/h**2 # df/dx |x=x_i weights in terms of␣
↪function values at, before and after x_i

    D1 = np.diag(grid[1:-1]) #D1 is the diagonal matrix where its entries are␣
↪the grid values, removing the first and last
    D2 = np.diag(grid**2 - nu**2)[1:-1] #D2 is the diagonal matrix where its␣
↪entries are the grid values, removing the first and last

    A = np.zeros((n-2,n)) # A and B are matrices n-2 by n matrices that hold␣
↪the 3 weights from compFD along the
    B = np.zeros((n-2,n)) # 3 middle diagonals of the matrix
    for i in range(n-2):
        A[i,i] = w2[0]
        A[i,i+1] = w2[1]
        A[i,i+2] = w2[2]
        B[i,i] = w1[0]
        B[i,i+1] = w1[1]
        B[i,i+2] = w1[2]

    D = (D1**2 @ A) + (D1 @ B) + D2     #Input matrices into differential␣
↪equation
    V = np.vstack(([1] + [0 for _ in range(n - 1)], D, [0 for _ in range(n -␣
↪1)] + [1])) #Apply boundary conditions
    b = np.zeros((n,1))
    b[-1] = 1     # y(0) = 0, y(20) = 1
    y = np.linalg.solve(V,b) # solve system

    return grid, y

#================== UNCOMMENT FOR VISUALIZATION =================
import time
start_time = time.time()

plt.figure(1)
n  = 1000
for k in range(1,8):
    x,y = BVP_ODE(n,k)
    plt.plot(x,y)
plt.show()

print("Time taken to run code:", (time.time() - start_time))
```



```
Time taken to run code: 1.0980656147003174
```

We use the matrices: D1 is the diagonal matrix where its entries are the grid values, D2 is the diagonal matrix with entries $d_{ii} = {x_i}^2 - \nu$, A and B are matrices n-2 by n matrices that hold the 3 weights from compFD along the 3 middle diagonals of the matrix and F is a column vector of function values for the solution that we want to find. If we let $U = D1^2 A + D * B + D2$, DF = 0, this is a system of n-2 equations with n variables. We add the initial conditions to make it an n by n system.

```
[ ]:
```