104283 - Introduction to Numerical Analysis
Spring 2023
Python Assignment 2

Name: Haoyi Yang
Student ID: 999009798

April 22, 2023

Polynomial Interpolation

1. Question

Given a set of distinct numbers $x_0, x_1, ..., x_{n+1}$ and the values $f(x_0), f(x_1), ..., f(x_{n+1})$ we want to compute the interpolating Lagrange polynomial P_n of degree n (page 108 Theorem 3.2).
Implement the following function in Python: **interpolate**(**points**)
The function gets a sequence of points which represent the set of $x_i, f(x_i)$ pairs.
The function computes and returns the corresponding Lagrange polynomial.
The polynomial must be defined using symbolic variables.
Test your function to find the unique interpolating polynomial defined by the points:

x	0	1	3	4	5
$f(x)$	-1	-0.5	1	2	3

Display the resulting polynomial in the report.

2. Code and Explanation

After learning Python Appendix 2, we know that we can import Sympy to use variable x and to simplify the formula. Since we need to input $x, f(x)$ and only one thing is allowed to be the function input, so we use 2-D array here. Then according to the Theorem 3.2 in page 108:

If $x_0, x_1, \ldots, x_n$ are $n+1$ distinct numbers and f is a function whose values are given at these numbers, then a unique polynomial $P(x)$ of degree at most n exists with

$$f(x_k) = P(x_k), \quad \text{for each } k = 0, 1, \ldots, n.$$

This polynomial is given by

$$P(x) = f(x_0)L_{n,0}(x) + \cdots + f(x_n)L_{n,n}(x) = \sum_{k=0}^{n} f(x_k)L_{n,k}(x), \tag{3.1}$$

where, for each $k = 0, 1, \ldots, n$,

$$L_{n,k}(x) = \frac{(x-x_0)(x-x_1)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_n)}{(x_k-x_0)(x_k-x_1)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)} \tag{3.2}$$

$$= \prod_{\substack{i=0 \\ i\neq k}}^{n} \frac{(x-x_i)}{(x_k-x_i)}.$$

■

We will write $L_{n,k}(x)$ simply as $L_k(x)$ when there is no confusion as to its degree.

It's easy to implement the whole Python function code:

```python
import sympy as sp
from sympy.abc import x
def interpolate(points):
    P = 0
    for i in range(len(points)):
        L = 1
        for m in range(len(points)):
            if m != i:
                L = L * ((x - points[m][0]) / (points[i][0] - points[m][0]))
                # The formula (3.1)
        P = sp.simplify((P + (points[i][1]) * L)) # The formula (3.2)
    return (f'P(x) = {P}')
```

Then we add the following code to print result:

```python
print(interpolate([[0,-1],[1,-0.5],[3,1],[4,2],[5,3]]))
```

And we get the output:
P(x) = -0.00416666666666667*x**4 + 0.0333333333333333*x**3 + 0.00416666666666714*x**2 + 0.466666666666666*x - 1.0
That is, the resulting polynomial is:
$P(x) = -0.00416666666666667x^4 + 0.0333333333333333x^3 + 0.00416666666666714x^2 + 0.466666666666666x - 1$

Newton's Method

1. Question

Implement the following function in Python:
newton(**p0**, **f**, **tol**, **max_iter**)
The function gets the following parameters as input:

- Parameter p0 is the initial guess.
- Some function f (defined symbolically).
- Convergence tolerance - tol.
- Maximum number of iterations - max_iter.

The function searches for a root of $f(x)$ using Newton's method (page 83 eq. (2.13)). The function will output the solution for which $f(x) = 0$ (if found) and the number of iterations used.
Use this function to find a real root of the polynomial found in the previous section. Add your results to the report.

2. Code and Explanation

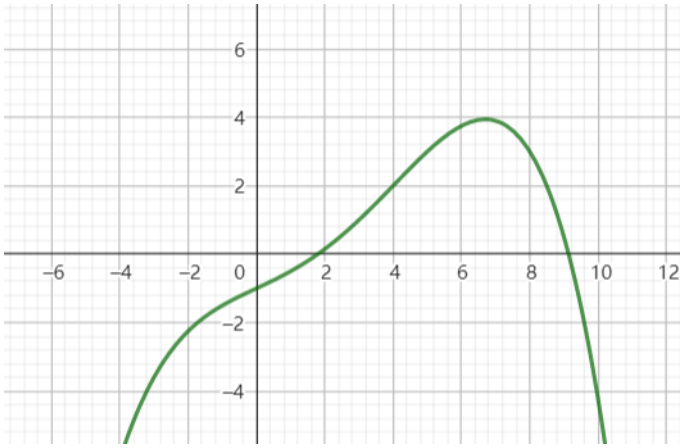
According to the basic idea of Newton's Method and the equation in 2.13, we can implement the following Python function code:

```
from sympy import diff
from sympy.abc import x
def newton(p0,f,tol,max_iter):
    df = diff(f,x)# This is f'(x).
    ddf = diff(diff(f,x))# This is f''(x).
    p = p0
    p_temp = 0
    num_iter = 1
    while num_iter <= max_iter: # Here to test if it's smaller than the max iterations.
        if abs(p - p_temp) < tol:
            return (f'The root is {p}.The number of iterations is {num_iter}.')
        else:
            p_temp = p
            p = p - ((f.subs(x,p)*df.subs(x,p))/((df.subs(x,p)**2)-
            \f.subs(x,p)*ddf.subs(x,p)))
            # p is the equation in 2.13
            # p is too long so I add "\" move it to a new line.
            # You can see the original version in the code I attached.
            num_iter += 1
    return(f'The method failed after {max_iter} iterations.')
```

I think this code is easy to understand and I guess the only thing may cause confusing is: what is *p_temp*? Since we need to compare whether the difference

between the current p and the previous p is less than tol, so I set a p_temp here to store the previous p. And in Python, we need to set an initial value when define a variable, since the root of the $P(x)$ in the section 1 must not be 0, so we set the initial value of p_temp equals to 0.

Then in order to decide the appropriate initial guess p_0, I use GeoGebra to draw a rough graph of $P(x)$:



It can be seen that the root of $P(x)$ is between $(0, 2)$ and $(8, 10)$, so I decide to choose five appropriate initial guesses p_0, they are: $-1, 3, 5, 7, 11$.

Then I define the function what we find in section 1 and set the maximum number of iterations to 100. We add the following codes to the function code above:

```
f = -0.00416666666666667*x**4 + 0.0333333333333333*x**3\
+ 0.00416666666666714*x**2 + 0.466666666666666*x - 1.0
# f is too long so I add "\" move it to a new line.
# You can see the original version in the code I attached.
print(newton(-1,f,10**(-5),100))
print(newton(3,f,10**(-5),100))
print(newton(5,f,10**(-5),100))
print(newton(7,f,10**(-5),100))
print(newton(11,f,10**(-5),100))
```

After running this code, we get the following results:
The root is 1.79412313572600.The number of iterations is 6.
The root is 1.79412313572878.The number of iterations is 5.
The root is 1.79412313572879.The number of iterations is 6.
The root is 9.13648185960766.The number of iterations is 9.
The root is 9.13648185960766.The number of iterations is 7.

From these results, we can find that although we choose different p_0, they all converges to 2 solutions with tolerance 10^{-5} after iterations. So we can say the two real roots are found and the code is correct.