UNITED STATES MILITARY ACADEMY

PROGRAMMING ASSIGNMENT 2

MA386: NUMERICAL ANALYSIS

SECTION G2

COL JOSEPH LINDQUIST

By

CADET EZRA HARRIS '22, CO E2

WEST POINT, NEW YORK

23 SEPTEMBER 2021

EH MY DOCUMENTATION IDENTIFIES ALL SOURCES USED AND ASSISTANCE RECEIVED IN COMPLETING THIS ASSIGNMENT.

_____ I DID NOT USE ANY SOURCES OR ASSISTANCE REQUIRING DOCUMENTATION IN COMPLETING THIS ASSIGNMENT.

SIGNATURE: EzraHarris

Tech Lab 2

Ezra Harris

September 23, 2021

Abstract

Horner's method allows for more efficient polynomial evaluation by creating a nested polynomial that reduces floating point operations. By testing two different polynomials we see a time savings of two-thirds. Additionally the reduction in floating point operations yields a more accurate result.

1 Introduction

In this assignment we seek to improve the evaluation of higher order polynomials by using Horner's method. Using Horner's method we can reduce the number of floating point operations that must be conducted thus improving the speed and accuracy of the results of the evaluations. Additionally Horner's method calculates the derivative of a function through the process of nesting. This makes root finding methods, such as Newton's method, more efficient. This is important because computing power is limited and when modeling very large systems efficiency must be gained where possible.

2 Methods

We evaluated Horner's method by comparing the computation time between a brute force evaluation of a polynomial and the evaluation of a nested polynomial produced by Horner's method. The brute force evaluation of the polynomial was done using a for loop. We passed the variable x^n a parameter and then evaluated it with every coefficient c_n and then added the result to subsequent evaluation for an nth degree polynomial.

Horner's method creates a nested polynomial that reduces the number of operations required to compute the value. It does this by repeatedly factoring the polynomial. To do this we define a function such that:

$$b_k = a_k + b_{k+1}x_0, \ for \ k = n-1, n-2, ..., 1, 0 \tag{1}$$

Where $b_0 = P(x_0)$. We implement this form to calculate the function $P(x)$ at x_0. The equation recursively substitutes b_n into the inner most nested polynomial (which should have a degree of 1). This creates an equation $Q(x)$ where:

$$Q(x) = b_n x^{n-1} + b_{n-1}x^{n-2} + ... + b_2 x + b_1 \tag{2}$$

This allows us to reconstruct the function $P(x)$ as:

$$P(x) = (x - x0)Q(x) + b_0 \tag{3}$$

I will not prove why $P(x)$ equals equation 3 as that is not the subject of this tech lab. It is sufficient to acknowledge the practical applications it offers for polynomial division. It does however allow an easy method to calculate the derivative of the function P(x) at a point x_0. Using the product rule we see that:

$$P'(x) = Q(x) + (x - x_0)Q'(x) \tag{4}$$

Immediately we can evaluate:

$$P'(x_0) = Q(x_0) \tag{5}$$

3 Analysis and Results

We begin our analysis by coding the brute force polynomial evaluator using the method outlined in the methods section. We evaluated two polynomials. A fourth order polynomial:

$$f(x) = -16x^4 - x^3 + 4x^2 - 3x + 1 \tag{6}$$

and a 1000th order polynomial. For the 4th order polynomial we calculated the value of $f(.5) = -.625$ in a time of 3.5999983083456755e-06s. For the 1000th degree polynomial we found that $P(.65) = 14.562128103863039$ in a time of 0.00022850000823382288s.

Next we coded Horner's method. We first established the function Horner and passed it the variables n, c, and x_0. n is the degree of the polynomial, c is an array that we defined as coef and passed it the variables generated by a random number generator with a seed[48], and x_0 is the number at which we will evaluate the function $P(x)$. We then defined the variable y and z and set them equal to the value of c at the nth degree of the polynomial that we were evaluating. In order to determine equation 1 we established a for loop that calculated b_n and set it equal to y. At the iteration n=0 the for loop returns b_0 which is equal to $P(x_0)$. We then constructed the function $Q(x)$ and set it equal to z. By equation 4 and 5 we know that $P'(x_0) = Q(x_0)$.

Our calculation of the 4th order polynomial yielded a value of $f(.5) = -.625$ in a time of 2.6999914553016424e-06s. Our calculation of the 1000th order polynomial yielded a value of $P(.65) = 14.562128103863042$ in a time of 0.00015650001296307892s. The value of the derivative was 40.45410374394076.

We see that the calculation of the values are not quite the same as the size of the polynomial increases and as the decimal values extend beyond 10^{-15}. This is caused by a reduction in round off error due to Horner's method having less operations. Additionally Horner's method took about 2/3's the time as the brute force polynomial evaluation. This is again caused by the reduction in operations required to be conducted. Finally and most beneficially however Horner's method computed the derivative as a by product of the reduction of the polynomial.

4 Conclusion

Horner's method is a powerful tool for analyzing polynomials because it increases the speed and efficiency of the evaluation. For large or very complex systems this savings in computing power can greatly improve the model. Additionally, the by product of calculating the derivative is very useful for other root finding methods such as Newton's method because of the savings in computations.

5 Appendix A

```
# -*- coding: utf-8 -*-
"""
Created on Thu Sep  9 07:57:18 2021

@author: Ezra
"""

def brute(n,c,x0):
    """

    Parameters
    ----------
    n : integer
        This is the degree of the polynomial that we are seeking
        to evaluate.
    c : float
        This is the coefficients of the polynomial.
    x0 : float
        This is the point which we are evaluating the polynomial.

    Returns
    -------
    p : float
        This is the value of the function evaluated at x0.

    """


    p=0
    for i in range(n+1):
        p = p + c[i]*x0**i
    return p


n0=.5
import random
random.seed(48)
#coef=[random.randint(-50,50) for i in range(1001)]
#print(coef)

coef=[1,-3,4,-1,-16]
n = len(coef)-1

bruteY=brute(n,coef,0.5)
print(bruteY)

import time
t1 = time.perf_counter()
```

```
bruteY = brute(n,coef,0.5)
tBrute  = time.perf_counter()-t1
print(tBrute)

def horner(n,c,x0):
    """


    Parameters
    ----------
    n : integer
        degree of the polynomial.
    c : float
        coefficients of the polynomial.
    x0 : float
        where we are evaluating the polynomial.

    Returns
    -------
    y : float
        the value of the function evaluated at x0.
    z : float
        the value of the derivative evaluated at x0.

    """

    y=c[n]
    z=c[n]
    for i in range(n-1,0,-1):
        y= x0*y+c[i]
        z=x0*z+y
    y=x0*y+c[0] #this computes the b0
    return y,z


t0 = time.perf_counter()
hornerY, hornerZ=horner(n,coef,0.5)
tHorner = time.perf_counter()-t0

print("Value of the gigantic polynomial", hornerY, "The direvative of the gigantic polynomial:", horner
print("the time it took to compute the grigantic polynomial using Horner's method:", tHorner)
```

6 References

Horner's Method. Wikipedia The Free Encyclopedia. Read through this article to get a better understanding of the method. However, the information that I gleaned form the article was also found in the book upon further review. https://en.wikipedia.org/wiki/HornerAccessed 22 Sep 2021.