ECON90055 Computational Economics: Exercise 1

Zelin Chen - 797036

March 21, 2017

1. Rounding Error

Numerically we know that $\frac{1}{3} * 3000 = 1000$. But the computer cannot evaluate the true value of $\frac{1}{3}$. Instead, we try four different representations to approximate the value of $\frac{1}{3}$: 0.3, 0.33, 0.3333, and 1/3.

In order to capture the round error, I first create four row vectors, and each element in the same row has identical value from each of 0.3, 0.33, 0.3333, and 1/3. Then I calculate the summation of each row vector and display them under 'format long'.

The following tables display the difference in summation between theoretical value and the practical value from the Matlab computation. From table 1, the practical summations are not exactly equal to theoretical values, as it shows a difference after 10 decimal places.

Approximation	Theoretical Sum	Practical Sum
0.3	900	899.999999999955
0.33	990	990.000000000051
0.3333	999.9	999.900000000021
1/3	1000	1000.000000000044

Table 1: Summation by different approximations

2. More on Rounding Error

We can write out the value of A^{-1}, x_1 and x_2 manually as a function in terms of the value of z:

$$A^{-1} = \begin{Bmatrix} \frac{3+z}{z^2+5z-0.25} & \frac{-2.5}{z^2+5z-0.25} \\ \frac{-2.5}{z^2+5z-0.25} & \frac{2+z}{z^2+5z-0.25} \end{Bmatrix}$$

$$x_1 = \frac{0.45z - 0.025}{z^2 + 5z - 0.25} \qquad x_2 = \frac{0.55z - 0.025}{z^2 + 5z - 0.25}$$

There are two ways to solve for the result. One is to use the *inv*() function to obtain the inverse of A, and then to multiply A^{-1} by column vector b. The other way is to manually reduce the workload until final steps, which is to obtain direct solutioon functions for value of x, as displayed above. Then based on the value of z, computer will calculate the value of x_1 and x_2 straight away without the process of solving for the inverse of A.

Most of the results are difficult to capture its rounding error because of its pattern of decimals are composition of different numbers. However, when $z = 0.05$, it shows something interesting. Each element of A^{-1} shows a rounding error after 12 decimals places. A^{-1} should theoretically equals to:

$$A^{-1} = 10^3 * \begin{Bmatrix} 1.22 & -1 \\ -1 & 0.82 \end{Bmatrix}$$

However, the Matlab result for $inv(A)$ shows the difference:

$$A^{-1} = 10^3 * \begin{Bmatrix} 1.220000000000665 & -1.000000000000546 \\ -1.000000000000546 & 0.820000000000447 \end{Bmatrix}$$

These round errors are further captured in the result for x. x theoretically should equal to:

$$x_1 = -1 \qquad x_2 = 1$$

On the other hand, the $inv(A)b$ provides:

$$x_1 = -1.000000000000602 \qquad x_2 = 1.000000000000508$$

Although we capture the rounding errors in each value of x, the patterns are still considerably small. Therefore, letting computer to do all the jobs will have some, but limited impact on the value of x in this case.

3. Rates of Convergence

To show that the sequence converges sublinearly, we have to show that:

$$\lim_{k\to\infty} \left| \frac{(k+1+1)^{-1} - L}{(k+1)^{-1} - L} \right| = \mu_k$$

where μ_k is the rate of covergence at each k, L is the limit of equation $(k+1)^{-1}$. The property of sublinear convergence states that when $k \to \infty$, $\mu_k \to 1$.

Given equation $(k+1)^{-1}$, we can obtain its limit L by setting $k \to \infty$. It provides $L = 0$. Computationally, it is impossible to let k goes to infinity. Instead, we can show that when k is large enough, μ_k converges to 1.

To prove that the sequence converges logrithmatically, it is to show that:

$$\lim_{k\to\infty} \left| \frac{(k+1+2)^{-1} - (k+1+1)^{-1}}{(k+1+1)^{-1} - (k+1)^{-1}} \right| = \mu_k$$

and when $k \to \infty$, $\mu_k \to 1$.

From figure 1, we can see clearly that the rate of convergences converges to the unique value of 1 both sublinearly logarithmatically.

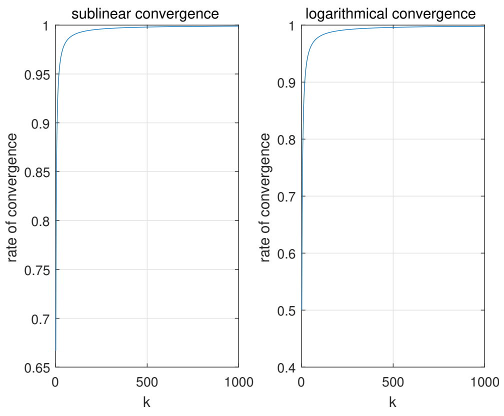

Figure 1: sublinear and logrithmatical rate of convergence

4. Efficient Evaluation of Polynomials

To conduct this experiment, I created one main program and three functions. The main basically enable the creation of a k-order polynomial by the user. The program will first ask user to input the order of the polynomial k. Then the user will be asked to input the coefficient on each x with order from 0 to k, which will be stored in a pre-specified row vector space based on the value of k. The program will also require the value of x and j from the user. Given x and j, the value of all x will be formed and stored in a vector space. With the value that obtained from the user, the program will pass them into each of the function that conduct different calculation methods. The code 'timeit' will help to record the timing of each function to provide results. Finally, based on the time burden of each function, it will show the which method is most time-saving when calculating polynomials.

Method 1: Algorithmically, in each value of x, and for each value of order n from 0 to k, the function will first calculate and store the value of x^n directly by the power operator. Then it multiply x^n by the its coefficient a_n, and store the result in another row vector as the n^{th} fraction of the polynomial. After completing all the fractions given a value of x, the sum of all fractions will be computed and stored based on the value of x.

Method 2: To conduct looping for this method, the function first specifies the value of x^0, which equals to 1. Then it starts the similar loop as method 1: in each value of x, and for each value of order n from 1 to k. It is 1 to k because the value of x^0 has been pre-determinant. In the loop, x^n is calculate by the multiplication of x and x^{n-1}, where the value of x^{n-1} is pre-determinant from previous iteration. This calculation save the time of computing from duplicate calculation of x^{n-1} when solving for x^n, since x^{n-1} has been calculated before. Therefore, every time, it is a call and use, not re-computing the whole thing from the beginning. Then rest of computation is exactly the same as method 1.

Method 3: It writes the polynomial in another format. Similar to Method2, it saves the time of calculating duplicate x^{n-1}. Furthermore, it also saves the time of summation. Instead of creating a loop to compute the fraction of each $a_n x^n$, and then sum all terms as in Method 1 and 2, the loop in Method 3 provides the final result straight away, which saves the time from capturing and writing data into different vectors.

After inputting the value on the commend screen: $k = 3, a_0 = -12, a_1 = -20, a_2 = 1, a_3 = 6$, coefficient of $x = 0.05$, the range of $j = 2$ and 2000, the program produces two results of time burdens as showing below based on the range of j.

Method 1	Method 2	Method 3
0.00000622	0.00000532	0.00000208

Table 2: Time Burden for j = 0,1,2

Method 1	Method 2	Method 3
0.00159610	0.00090587	0.00007611

Table 3: Time Burden for j = 0,1,2,...,2000

From these two tables we can see that, the result is consistent with my numeric analysis on the methods, as Method 1 has the greatest time burden to compute the input polynomial, and Method 3 spends the least time. By comparing two tables, we also observe that large range of j will further demonstrate the simplicity of method 3, since the time consumption difference between Method 3 and Method 1&2 magnifies as j increases.