ISSN 0974 - 9373

Vol. 15 No.4 (2011) **Journal of International Academy of Physical Sciences** pp. 455-466

Comparative Study of Complexity of Algorithms for Iterative Solution of Non-Linear Equations

R. N. Jat
Department of Mathematics, University of Rajasthan, Jaipur
E-mail: khurkhuria_rnjat@yahoo.com

D. S. Ruhela
Department of Computer Application, S. M. L. (P.G.) College, Jhunjhunu (Raj.)
E-mail: dsruhela@yahoo.com

(Received December 22, 2010)

Abstract: In this paper algorithms for Bisection, Secant, Rugla-Falsi, Newton Raphson and Adaptive Bisection methods to find the roots of non-linear equation are studied for their complexity. Using RAM (Random Access Machine) model the complexity of all algorithms are calculated. In RAM model we count the primitive operations executed by given algorithm and then find the time in function t(n). The complexity can represent in the form of Big-Oh notation in term of time function. In order to compare the adaptive Bisection method with Bisection method, Secant method, Regula-Falsi method and Newton Raphson method a variety of functions are used with same criteria i.e. 10-6. The time complexity of all algorithms are O(n) where n is number of iteration. The result shown in table shows that for function f(x) =ex-3x the number of iteration in Bisection methods are 20 while in adaptive Bisection method is 5. Since time complexity of both algorithms are O(n) , so Adaptive bisection algorithm will execute faster to compare all other method and it will take less time to run the program.
Keywords: Bisection method, RAM (Random Access Machine), Big-Oh.

1. Introduction

Computational complexity can be defined as a function of the size of input resources required for computation[1]. The running time of an algorithm or data structure operation typically depends on number of factors like input size, hardware and software used[2]. We can study its running time on various inputs and recording the actual time spent in each execution. Such measurement can be taken in an accurate manner by using system calls built in language for which algorithm is written. In general, we are interested in determining the dependency of running time on size of input. In order to determine this, we can perform several experiments on many different test

inputs of various sizes. Very often in engineering and mathematics, non-linear equation are solved using iterative methods. The Bisection Method[3] is among the few iterative methods which guarantee convergence. Here we will present the algorithm for iterative methods and then find the complexity for it.

RAM Machine Model Definition 1.2

An approach of simply counting primitive operations gives rise to a computational model called Random Access Machine (RAM)[2]. In this model we define a set of high level primitive operations that are largely independent from programming language used and can be defined also in the pseudo-code. Primitive operation include following:

1. Assigning the value to variable.
2. Calling a method.
3. Performing an arithmetic operation.
4. Comparing two numbers.
5. Indexing into array.
6. Following an object reference.
7. Returning from a method.

Counting Primitive Operation 1.3

We now show to how count the number of primitive operations executed by an algorithm, used for all methods. We do this analysis by focusing on each step of the algorithm and counting primitive operation that it takes, taking into consideration that some operation are repeated, because they are enclosed in the body of a loop.

Asymptotic Notation[2] 1.4

We have clearly gone into detail for evaluating the running time of simple algorithms. Such an approach would clearly prove cumbersome if we had to perform it for more complicated algorithms. In general, each step in pseudo-code description and each statement in a high level language implementation corresponds to small number of primitive operations that does not depend on input size. Thus we can perform a simplified analysis that estimates the number of primitive operation executed up to a constant factor, by counting the steps of pseudo-code or the statements of high level language executed. Fortunately, there is a notation that allows us to characterize the main factor affecting an algorithm's running time without going into all detail of exactly how many primitive operations are performed for each constant time of instructions.

The "Big-Oh" Notation[2] 1.5

Let f (n) and g(n) be function mapping nonnegative integers to real numbers. We say that f(n) is O(g(n)) if there is a real constant c>0 and an integer constant n0>=1 such that f(n)<=cg(n) for every integer n>=n0. This definition is referred to as "big-Oh" notation. Alternatively, we can also say "f(n) is order g(n)[4].

2. Bisection Method

The Bisection method (BM)[3], is among few iterative method which guarantee convergence which is in linear rate.

Algorithm for Bisection Method 2.1

To find the root of f(x) =0 with a prescribed tolerance say epsilon.
Given that a_k and b_k such that $f(a_k)*f(b_k)<0$. The value c_k is used to store the middle point of the interval.

1. Begin
2. Read a_k,b_k // Input the values of a_k and b_k
3. Read Epsilon //Input the tolerance
4. Repeat step 5 to 6 while(($|a_k-b_k|/a_k$)<epsilon) and ($f(c_k)\neq 0$))
5. $c_k=(a_k+b_k)/2$
6. If($f(a_k)*f(c_k)<0$) then
   Set $b_k=c_k$
   Else
   Set $a_k=c_k$
   End if
7. Write "The approximate root is",c_k
8. End

Counting Primitive Operation2.2

1. Reading the value of variable a_k,b_k contributes two unit of count.
2. Reading the value of epsilons contributes one unit of count.
3. Before entering the body of loop condition $(a_k-b_k)/a_k$<epsilon is verified. This action corresponds to three (one for subtraction, one for division and one for comparison) primitive operation and performed n times.
4. At the beginning of loop c_k is calculated. This action corresponds to executing three primitive operations.
5. In the body of the loop, condition $f(a_k)*f(b_k)<0$ is verified. This action corresponds to executing four operations.

6. As per result of verification one assignment statement will execute requires one operation.
7. The body of loop is executed n times. Hence, at each iteration of loop, nine primitive operations is performed 9n times.
8. Printing the value of C_k requires one operation.

To summarize, the number of primitive operations t (n) executed by algorithm is at least.

$$t(n) = 2+1+(3+3+4+1)n+1=11n+4.$$

The "Big-Oh" Notation 2.3

By the big-Oh definition, we need to find a real constant c>0 and an integer constant n0>=1 such that 11n+4 <=cn for every integer n>=n0. It is easily see that a possible choice is c=15 and n0=1. The big- Oh notation allows us to say that a function of n is "less than or equal to" another function (by the equality "<=" in the definition), up to constant factor eventually (by the statement "n>=n0" in the definition).

The number of primitive operations executed by algorithm for Bisection method is at most 11n+4. We may therefore apply the big-Oh definition with c=15 and n0=1 and conclude the running time of algorithm in O(n).

=O(n). So the complexity of algorithm used for Bisection method is O(n).

In fact, any polynomial $a_k n^k+a_{k-1}n^{k-1}\ldots\ldots+a_0$ *will always be* $O(n^k)$

3. Regula-Falsi method

A way to avoid such pathology is to ensure that the root is bracketed between the two starting values and remains between the successive pairs. When this is done , the method is known as linear interpolation or method of false position. This technique is similar to Bisection[3] except the next iterate is taken at the intersection of a line between the pairs of (x, f(x)) values and the x-axis is rather than at the midpoint of x-values.

Algorithm for Regular –False method 3.1

1. Read x0 , x1 and epsilon
2. Repeat
3. Set x2=x1-f(x1)*(x0-x1)/(f(x0)-f(x1))
4. If f(x2) is opposite sign to f(xo) then
5. Set x1=x2
6. Else

7. Set x0=x2
8. Endif
9. Until absolute(f(x2)<epsilon
10. Print x2
11. End

Counting primitive operations 3.3

1. Reading the value of variable x0,x1 and epsilon contributes three unit of count.
2. As per result of verification one x2 is calculated using 8 operations
3. The condition f(x0)*f(x2)<0 contributes 3 operation.
4. One operation for assignment statement after the condition.
5. Two operations are required for f(x2)<epsilon.
6. The body of loop is executed n times. Hence, at each iteration of loop, [8+3+1+2] 14 primitive operations is performed 14n times.
7. Printing the value of x2 requires one operation.

To summarize, the number of primitive operations t (n) executed by algorithm is at least.

$$t(n) = 2+1+14n+1=14\ n+4.$$

The "Big-Oh" Notation 3.2

By the big-Oh definition, we need to find a real constant c>0 and an integer constant n0>=1 such that 14n+4 <=cn for every integer n>=n0. It is easily see that a possible choice is c=18 and n0=1. The big- Oh notation allows us to say that a function of n is "less than or equal to" another function (by the equality "<=" in the definition), up to constant factor eventually (by the statement "n>=n0" in the definition).

The number of primitive operations executed by algorithm for Regula-False method is at most 14n+4. We may therefore apply the big-Oh definition with c=18 and n0=1 and conclude the running time of algorithm in O(n).=O(n). So the complexity of algorithm used for Regula-Falsi method is O(n).

4. The Secant method

The Secant method begins by finding two points on curve of f(x), hopefully near to the root we seek. If f(x) were truly linear, the straight line would intersect at the x-axis at root. But f(x) will never be exactly linear because we would never use a root finding method on a linear function. That

means the intersection of the line with x-axis is not at x=r but that it should be close to it. From the obvious similar triangles we can write

(x1-x2)/f(x1)=(x0-x1)/f(x0)-f(x1),

and from this solve from x2

$$x2 = x1 - f(x1) * \frac{x0 - x1}{(f(x0) - f(x1)'}$$

Because f(x) is not exactly linear x2 is not equal to r but it should closer than either of the two points we begin with. If we repeat this we have:

$$x(n+1) = xn - f(xn) * \frac{x(n-1) - xn}{f(x(n-1)) - f(xn)},$$

Because each newly computed value should be nearer to the root, we can do it easily after second iterate has been computed, by always using the last two computed points. But after the first point there aren't "two last computed points". So we make sure to start with x_1 closer to the root than x_0 by testing $f(x_0)$ and $f(x_1)$ and swapping if first functional value is smaller.

Algorithm for Secant method4.1

1. Read x0, x1 // that are near to the root to determine a root of f(x)=0
2. If |f(x0)|<|f(x1)| then swap x0, x1 //interchange x0 with x1
3. Repeat
4. Set x2=x1-f(x1)*(x0-x1)/(f(x0)-f(x1)
5. Set x0=x1
6. Set x1=x2
7. Until |f(x2)|< tolerance e
8. Print "root is" x2
9. End

Counting primitive operations 4.2

1. Reading the value of variable x1,x2,e,contributes three unit of count.
2. Step 2 requires four operations.
3. In the body of loop x2 is calculated which requires nine operations.
4. To compute x1 and x0 contributes two operations.

5. To check the condition |f(x2)|<e contributes two operations, and hence the body of loop will execute n times, so (9+2+2) are executed n times..
6. Printing the value of x2 requires one operation.

To summarize, the number of primitive operations t (n) executed by algorithm is at least.

$t(n) = 3+2+13n+1=13n+6$

The "Big-Oh" Notation 4.3

By the big-Oh definition, we need to find a real constant c>0 and an integer constant n0>=1 such that 13n+6 <=cn for every integer n>=n0. It is easily seen that a possible choice is c=19 and n0=1. The big- Oh notation allows us to say that a function of n is "less than or equal to" another function (by the equality "<=" in the definition), up to constant factor eventually (by the statement "n>=n0" in the definition).

The number of primitive operations executed by algorithm for Secant method is at most 13n+6. We may therefore apply the big-Oh definition with c=19 and n0=1 and conclude that the running time of algorithm in O(n).=O(n). So the complexity of algorithm used for Secant method is O(n).

5. Newton Rapson method

This method is based on a linear approximation of the function but does so, using a tangent to the curve. Starting from a single initial value x0, that is not too far from a root, we move along the tangent to its intersection with x-axis, and take that the next approximation. This is continued until either the successive x-values are sufficiently close or the value of the function is sufficiently near zero.

The general terms

$$x(n+1) = xn - \frac{f(xn)}{f'(xn)},$$

For n=0,1,2,3……

Algorithm5.1

To determine a root of f(x)=0, given x_0 reasonably close to the root.

1. Read x0,e
2. If($f(x_0)$<>0) and f'(x_0)<>0 then

3. Repeat
4. Set $x=x_0-f(x_0)/f'(x_0)$
5. Set $x_1=x_0$
6. Until(Absolute(x_1-x_0)<e
7. Print x1
8. Endif

Counting primitive operations5.2

1. Reading the value of variable x0,e contributes two unit of count.
2. To compute f(x0) and f"'(x0) and compare with 0 contributes four operation.
3. To compute x0 contributes one count.
4. To compute x1 contributes five operations to count.
5. To check the condition Absolute(x1-x0)<e contributes three operations. The loop is executed n times, so (1+ 3+5) operations executed 9n times.
6. To print x1 contributes one operation.

To summarize, the number of primitive operations t (n) executed by algorithm is at least.

$$t(n) = 2+4+(1+5+3)n+1=9n+7.$$

The "Big-Oh" Notation5.3

By the big-Oh definition, we need to find a real constant c>0 and an integer constant n0>=1 such that 9n+7 <=cn for every integer n>=n0. It is easily see that a possible choice is c=16 and n0=1. The big- Oh notation allows us to say that a function of n is "less than or equal to" another function (by the equality "<=" in the definition), up to constant factor eventually (by the statement "n>=n0" in the definition).

The number of primitive operations executed by algorithm for Secant method is at most 9n+7. We may therefore apply the big-Oh definition with c=16 and n0=1 and conclude the running time of algorithm in O(n).=O(n). *So the* complexity of algorithm used for Newton Raphson method is O(n).

6. Adaptive Weighted Bisection Method (AWBM)

Let f be continuous and twice differentiable over the interval[a,b] and f(a)*f(b)< 0 such that there exist a number r ϵ [a,b] where f(r)=0 and f''(r) $\neq$ 0 if we define the sequence { C_k; k=0,1,2,……} as[2]

Program for Adaptive Bisection Method for x^3+2

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#define f(x) (x*x*x+2)
#define f1(x) (3*x*x)
#define f11(x) (6*x)
#include<conio.h>
void main()
{
inti,k;
float r,a0,b0,c0,e,s,wk,wkopt;
clrscr();
printf("\nEnter first point of interval");
scanf("%f",&a0);
printf("Enter the second point of interval");
scanf("%f",&b0);
printf("Enter the prescribed tolrance");
scanf("%f",&e);
s=f(a0)*f(b0);
if(s>0)
 {
printf("Starting values are unsuitable");
getch();
exit(1);
 }
for(k=0;;++k)
  {
    if(((f1(b0)<0)&&(f11(b0)<0))||((f1(b0)>0) && (f11(b0)>0)))
      {
wk=-1*(f(b0)/(a0-b0))*f1(b0);
if((wk>0) && (wk<1)) wkopt=wk;
else
wkopt=1/2.0;
        c0=b0+wkopt*(a0-b0);
       }
else
      {
wk=-1*f(a0)/((b0-a0)*f1(a0));
if((wk>0) && (wk<1))
wkopt=wk;
```

```
else
wkopt=1.0/2.0;
      c0=a0+wkopt*(b0-a0);
   }
if(fabs(f(c0))<.000001)
    {
     r=c0;
printf("Root is %f",r);
printf("No of iteration is %d",k+1);
getch();
exit(1);
    }
If(f(a0)*f(c0)<0)
     b0=c0;
else
    a0=c0;
  }
printf("No of iteration=%d",k);
getch();
 }
```

Counting Primitive Operation 6.2

1. Reading the value of variable a0,b0 contributes two unit of count
2. Reading the value of epsilons contributes one unit of count.
3. At the beginning of loop sign of f' (b0) and f''(b0) compared. This action corresponds to executing one primitive operation.
4. In the loop wk is calculated using assignment it takes one operation.
5. Wk is compared with 1 and 0 then chooses wkopt requires three operations.
6. Ck is compared with e needs one operation and assigning ck to r requires one operation, so two more operation is required.
7. At end f(a0) and f(c0) is calculated and compared with 0 need one operation and then two more assignment are required for a0 and b0, total five operation is required.
8. The loop executed n+1 times. Hence each iteration of loop, 12 primitive operations is performed.

To summarize, the number of primitive operations t(n) executed by algorithm is at least. 3+12(n+1)t(n)=12n+15.

Function	Initial Intervals	No of Iteration n				
		Bisection O(n)	Newton Rapson Method O(n)	Secant Method	False Method	Adaptive Bisection O(n)
$F(x)=e^x-3x$	[1,2]	20	8	9	19	**5**
$F(x)=x^3-x-1$	[1,2]	20	6	7	17	**5**
$f(x)=\cos(x)-xe^x$	[0,1]	17	6	7	13	**4**
$f(x)=x^3+2$	[-2,2]	22	5	8	19	**5**
F(x)=3x-cos(x)-1	[0,1]	23	4	5	5	**3**

Table: 1

The "Big-Oh" Notation 6.3

By the big-Oh definition, we need to find a real constant c>0 and an integer constant n0>=1 such that 12n+15 <=cn for every integer n>=n0. It is easily see that a possible choice is c=27 and n0=1. The big Oh notation allows us to say that a function of n is "less than or equal to" another function (by the equality "<=" in the definition), up to constant factor infinitely (by the statement "n>=n0" in the definition).

The number of primitive operations executed by algorithm for adaptive Bisection method is at most 12n+15. We may therefore apply the big Oh definition with c=27 and n0=1 and conclude the running time of algorithm is O (n).

So the complexity of algorithm used for Adaptive Bisection method is O(n).

7. Result and Conclusion

In order to compare the adaptive Bisection method with Bisection method, Secant method, Regula False method and Newton Rapson method a variety of functions are used with same criteria i.e. 10^{-6}. The time complexity of all algorithms are O (n) where n is number of iteration. The table shows that for function $f(x) =e^{x-}3x$ the number of iteration in Bisection methods are 20 while in adaptive Bisection method is 5. Since time complexity of both algorithms are O (n), so Adaptive bisection algorithm will execute faster

compared to all other method and it will take less time. Similarly the same result shows for function $f(x) = x^3-x-1$.

References

1. H. Thomas Corner, *Introduction to Algorithms*, PHI, India.
2. T. Michael Goodrich and Roberto Tamassia, *Algorithm Design*, 2008.
3. H. John Mathews, Numerical Methods for Mathematics, Science and Engineering, *2n ed, PHI, New Jersey*, (1992) 54-63.
4. M. Z. Dauhoo and Soobhug, An Adaptive Weighted Bisection Method for Finding Roots of Non-Linesr Equations, *International Journal of Computer Mathematics*, (2003) 897-906.
5. F. Curtis, Gerald and O. Patrick Wheately, Applied Numerical Analysis, *Person Education*, (2009) 33-43.
6. In-won Lee and Gill-Ho Jung A generalized Newton Rapson Method using curvature, *Journal of Communication in Numerical Methods for Engineering*, **11** (1995) 757-763.