The Birth of an Algorithm

Namir Shammas

Introduction

Solving for the roots of nonlinear functions and polynomials is one of the cornerstones of numerical analysis. Over the centuries, mathematicians have developed new root-seeking algorithms and refined existing ones. The Bisection method remains one of the simplest and slowest root-seeking methods. Numerical Analysis books still discuss the Bisection method mainly for historical reasons. The method's main virtue is that is it guaranteed to obtain the root in proper ranges of values that contain a root.

This article shows you how to start with the Bisection method and create a new version that significantly accelerates the rate of convergence to a root value. The process is iterative. You will see a few design choices—ones that work and one that does not. I will present implementations for the HP Prime graphing calculator.

The Bisection Method

The Bisection method implements a simple idea. Given a root-bracketing range [A, B] and a function f(x) = 0 that is continuous in that range, the method iteratively shrinks that root-bracketing range. The basic idea as follows:

- Pick a point m inside the range [A, B]. The best choice for m is the median of that range.
- Calculate the value of f(m).
- Compare the signs of f(m) and f(A). If the two functions have the same sign, then replace A with m. Otherwise replace B with m. This step reduces the size of the root-bracketing interval by half.
- Repeat the above steps until the absolute difference between the updated values of A and B is equal and/or below a small tolerance value. This tolerance value reflects the accuracy of the calculated root.

We can calculate the number of iterations required to reduce the initial root-bracketing interval into one equal or less than the tolerance value. The estimated number of iterations is approximately log(|B-A|/tolerance)/log(2). This is a unique feature of the Bisection method. The implementations of the Bisection method usually store the values of f(A) and f(B) to avoid recalculating them in subsequent iterations. In the range-reducing step, the stored value of f(A) or f(B) is replaced by the value of f(m). Figure 1 shows a general plot for a Bisection method iteration.

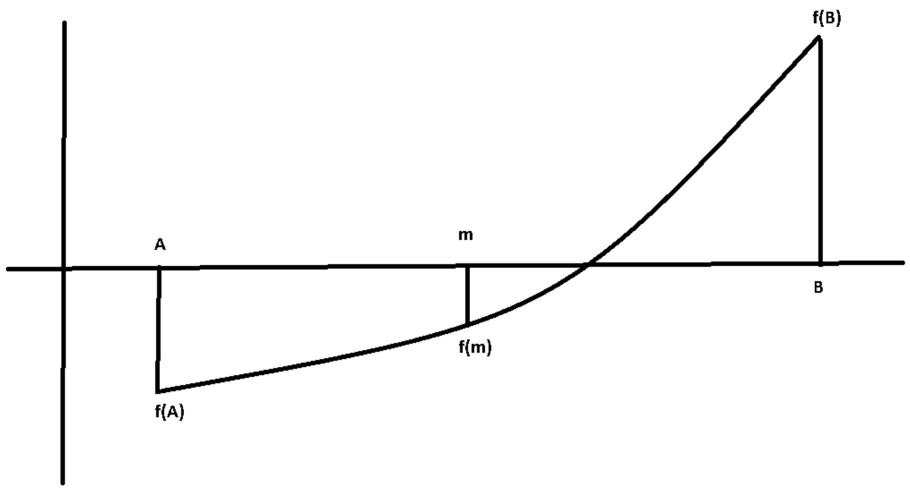


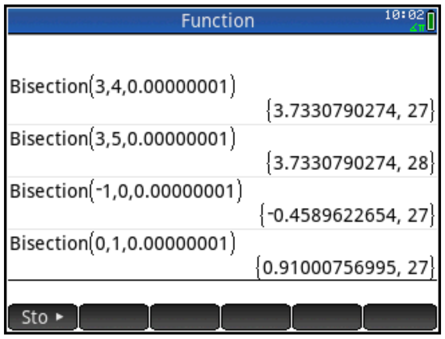
Figure 2 – The Bisection method.

Listing 1 shows the HP Prime source code for functions **MYFX** and **Bisection**. The first Prime function implements the mathematical function f(x) = 0. The second Prime function implements the Bisection method. This function has three argument, namely, a, b, and toler (the tolerance value). The function returns a list containing the root value and the number of iterations. Listing 1 has the target function coded as $e^x – 3x^2$. Throughout this article we will find the roots of this function. If you want to locate the roots of another function, you need to edit the statements inside **MYFX**.

```
EXPORT MYFX(x)
BEGIN
  RETURN e^(x)-3*x²;
END;

EXPORT Bisection(a,b,toler)
BEGIN
  LOCAL Fa, Fb, m, Fm, iter;

  Fa:=MYFX(a);
  Fb:=MYFX(b);
  IF Fa*Fb>0 THEN
    RETURN "A and B have same sign functions";
  END;

  iter:=0;
  REPEAT
    iter:=iter+1;
    m:=(a+b)/2;
    Fm:=MYFX(m);
    IF Fa*Fm>0 THEN
      a:=m;
      Fa:=Fm;
    ELSE
      b:=m;
      Fb:=Fm;
    END;
  UNTIL (ABS(a-b)<toler OR Fm==0);
  RETURN {(a+b)/2,iter};
END;
```

Listing1 – The Bisection function.

Notice that function **Bisection** checks the signs of the initial values of f(A) and f(B). If they are the same, the function returns an error message. Figure 2 show sample use of the **Bisection** function. The



```
Bisection(3,4,0.00000001)
                              {3.7330790274, 27}
Bisection(3,5,0.00000001)
                              {3.7330790274, 28}
Bisection(-1,0,0.00000001)
                             {-0.4589622654, 27}
Bisection(0,1,0.00000001)
                             {0.91000756995, 27}
```

Figure 2 – Sample use of the Bisection Function.

calculations show how to obtain roots in the intervals [3, 4], [3, 5], [–1, 0], and [0, 1]. The first two intervals lead to the same root. The number of iterations is rather high due to the slow convergence.

The Bisection Plus Algorithm Take 1

Now that you have seen and worked with the Bisection method, let's look into improving it. While selecting the median of the root-bracketing interval is optimum in certain ways, let's see if we can do better. The first approach is to calculate a random value *around* the median. This approach requires defining a range around the median from which we get a random value. If use a broad range around the median we may drift close towards the end of the interval [A, B]. Statistically, this wide range will give us poorer performance on the average. By contrast, a rather narrow interval around the median might help us tweak the midpoint selection. I choose to make that interval as |(B–A)|/4. This means that the interval will range from |(B–A)|/8 below the median to |(B–A)|/8 above it. Mathematically this translates into:

m = (A+B)/2 + |(B–A)|/4 * (Random number between –0.5 and 0.5)

Figure 3 shows a general plot for the Bisection Plus (take 1) method. The red region depicts the range used to calculate the random value of m. The figure also marks the midpoint of the range [A, B].

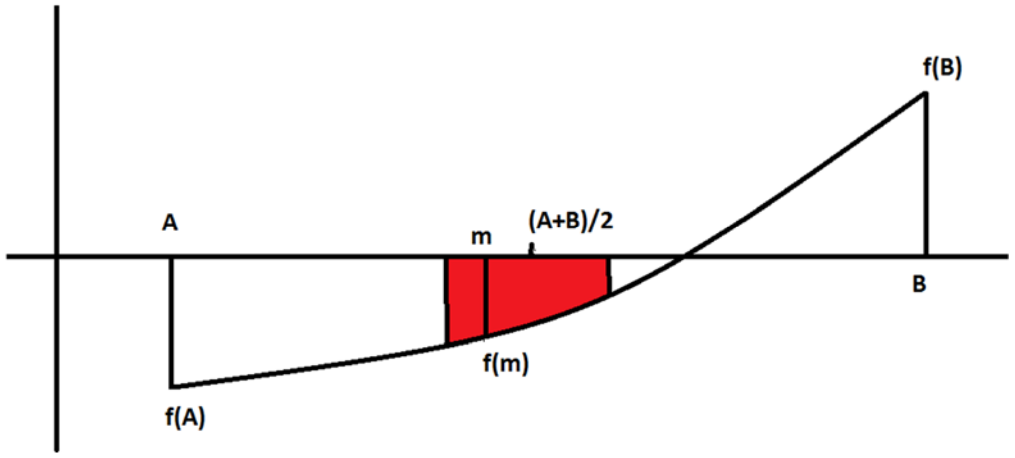


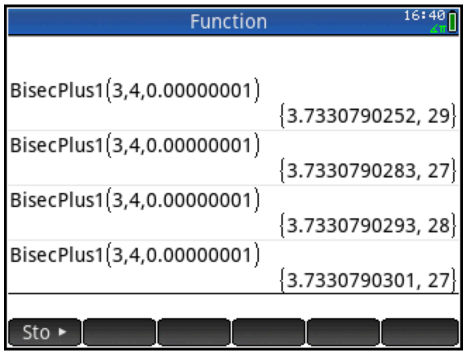
Figure 3 – The Bisection Plus (take 1) method.

Listing 2 shows the HP Prime implementation of the above method as function **BisecPlus1**.

```
EXPORT BisecPlus1(a,b,toler)
BEGIN
  LOCAL Fa, Fb, m, Fm, iter, r;

  Fa:=MYFX(a);
  Fb:=MYFX(b);
  IF Fa*Fb>0 THEN
    RETURN "A and B have same sign functions";
  END;

  iter:=0;
  REPEAT
    iter:=iter+1;
    m:=(a+b)/2;
    r:=ABS(b-a)/4;
    m:=m+r*(RANDOM()-0.5);
    Fm:=MYFX(m);
    IF Fa*Fm>0 THEN
      a:=m;
      Fa:=Fm;
    ELSE
      b:=m;
      Fb:=Fm;
    END;
  UNTIL (ABS(a-b)<toler OR Fm==0);
  RETURN {(a+b)/2,iter};
END;
```

Listing2 – The BisecPlus1 function.

Listing 2 shows the code for function **BisecPlus1**. This function has the same parameters and returns the same type of values as function **Bisection**. Keep in mind that the number of iterations will vary when you run the function several times, since the code uses random numbers. Figure 4 shows sample runs for the function **BisecPlus1**. The figure shows several runs for the root in the interval [3, 4]. Notice that the number of iterations ranges from 27 to 29. Nevertheless the results are disappointing since the change in the algorithm did not significantly reduce the number of iterations.



```
BisecPlus1(3,4,0.00000001)
                         {3.7330790252, 29}
BisecPlus1(3,4,0.00000001)
                         {3.7330790283, 27}
BisecPlus1(3,4,0.00000001)
                         {3.7330790293, 28}
BisecPlus1(3,4,0.00000001)
                         {3.7330790301, 27}
```

Figure 4 – Sample use of the BisecPlus1 Function.

The Bisection Plus Algorithm Take 2

The first attempt to improve the Bisection method basically fell flat on its face. Back to the proverbial drawing board!

I was inspired for the next approach by Ostrowski's root-seeking method[3]. Unlike typical root-seeking algorithms, Ostrowski designed his algorithm to calculate not one, but two refinements for the root in each iteration! Thus the new approach observes the following tasks:

- Calculate the midpoint m and its function value f(m) just like with the Bisection method.
- Use the point at the median and either range endpoints to calculate a straight line. Find the X intersect of that line, call it m2. An alternative approach is to calculate m2 as the X intersect of the line going through the points at A and B. This alternate approach does well for some cases, but not as good as the approach using the point at (m, f(m)).
- Determine if m and m2 have functions of opposite signs. If this condition is true, then the method replaces the root-bracketing range [A, B] with the narrower range of [m, m2].
- If m and m2 have functions of the same sign, replace A or B with m2, just like in the Bisection method.

- The iterations should also check if the value of m2 is very close to the one in the previous iteration. If this is the case, the method can return a root value.

The design of the step for calculating m2 went through several refinements. The choices available for calculating m2 were:

1. Always calculate m2 using points (A, f(A)) and (m, f(m)).
2. Systematically alternate the calculation of m2 between using points (A, f(A)) and (m, f(m)) and points (B, f(B)) and (m, f(m)).
3. Select A or B, such that the sign of the associated function value is opposite of the sign of f(m). Use that end point and (m, f(m)) to calculate m2.

The third choice proved to be the best, because it made sure that m2 falls in the interval of [A, B]. In addition, the calculated straight line is using the point at m, which is usually closer to the root than the points at A and B. Figure 5 shows a general plot for the new Bisection (take 2) method.

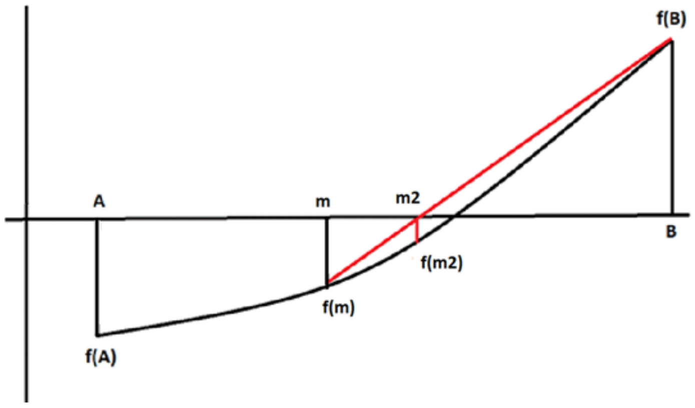


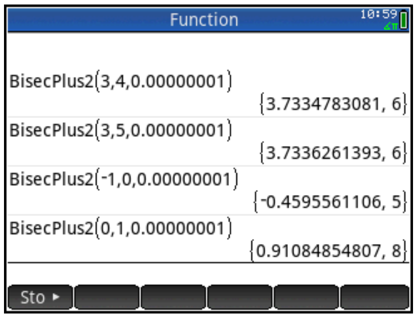
Figure 5 – The Bisection Plus (take 2) method.

Listing 3 shows the HP Prime implementation of the above method as function **BisecPlus2**.

```
EXPORT BisecPlus2(a,b,toler)
BEGIN
  LOCAL Fa, Fb, m, Fm, iter, lastm2;
  LOCAL m2, Fm2, slope, intercept;

  Fa:=MYFX(a);
  Fb:=MYFX(b);
  IF Fa*Fb>0 THEN
    RETURN "A and B have same sign functions";
  END;

  lastm2:=a;
  iter:=0;
  REPEAT
    iter:=iter+1;
    m:=(a+b)/2;
    Fm:=MYFX(m);
    IF Fa*Fm>0 THEN
      slope:=(Fb-Fm)/(b-m);
      intercept:=Fb-slope*b;
    ELSE
      slope:=(Fa-Fm)/(a-m);
      intercept:=Fa-slope*a;
    End;
    m2:=-intercept/slope;
    Fm2:=MYFX(m2);
    IF Fm*Fm2<0 THEN
      a:=m;
      Fa:=Fm;
      b:=m2;
      Fb:=Fm2;

    ELSE
      IF Fa*Fm2>0 THEN
        a:=m2;
        Fa:=Fm2;
      ELSE
        b:=m2;
        Fb:=Fm2;
      END;
    END;
    IF Fa*Fm>0 THEN
      a:=m;
      Fa:=Fm;
    ELSE
      b:=m;
      Fb:=Fm;
    END;
    IF ABS(lastm2-m2)<toler THEN
      RETURN {(a+b)/2,iter};
    END;
    lastm2:=m2;
  UNTIL (ABS(a-b)<toler OR Fm==0);
  RETURN {(a+b)/2,iter};
END;
```

Listing3 – The BisecPlus3 function.

Listing 3 shows the code for function **BisecPlus2**. This function has the same parameters and returns the same type of values as function **Bisection**. Figure 6 shows several runs for the root in the intervals [3, 4], [3, 5], [–1, 0] and [0, 1]. The results are very encouraging and show that the additional linear interpolation step contributes significantly to zooming in on the root. The price to pay for this acceleration is having two function calls per iteration, as opposed to a single one in the Bisection method.



```
BisecPlus2(3,4,0.00000001)
                                 {3.7334783081, 6}
BisecPlus2(3,5,0.00000001)
                                 {3.7336261393, 6}
BisecPlus2(-1,0,0.00000001)
                                {-0.4595561106, 5}
BisecPlus2(0,1,0.00000001)
                                {0.91084854807, 8}
```

Figure 6 – Sample use of the BisecPlus2 Function.

Nevertheless, the total number of function calls in this version of the Bisection Plus is still less than that of the Bisection method.

How Good is the Bisection Plus Method?

I have compared the results of the Bisection Plus method with Newton's method (the most popular method) and the results are very encouraging. In most cases, the Bisection Plus gives results that are not far behind those of Newton's method. In some cases, the Bisection Plus is able to match or even outdo Newton's method.

Observations and Conclusions

This article showed you the process of crafting a new algorithm from an old one. While the first attempt failed, the second one produced encouraging results. The Bisection Plus method uses midpoint selection followed by a linear interpolation to zoom in on the root. The new method is not susceptible to low tangent values near the root as is the case with Newton's method.

References

1. William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Numerical Recipes: The Art of Scientific Computing, 3rd edition, Cambridge University Press; 3rd edition, September 10, 2007.
2. Richard L. Burden, J. Douglas Faires, Numerical Analysis, Cengage Learning, 9th edition, August 9, 2010.
3. Namir Shammas, *Ostrowski's Method for Finding Roots*, HP Solve, July 2012, issue #28.

About the Author

Namir Shammas is a native of Baghdad, Iraq. He resides in Richmond, Virginia, USA. Namir graduated with a degree in Chemical Engineering from the University of Baghdad. He also received a master degree in Chemical Engineering from the University of Michigan, Ann Arbor. He worked for a few years in the field of water treatment before focusing for 17 years on writing programming books and articles. Later he worked in corporate technical documentation. He is a big fan of HP calculators and collects many vintage models. His hobbies also include traveling, music, movies (especially French movies), chemistry, cosmology, Jungian psychology, mythology, statistics, and math. As a former PPC and CHHU member, Namir enjoys attending the HHC conferences. Email me at: nshammas@aol.com