Nonlinear equations:
The Regula Falsi method

MATH2070: Numerical Methods in Scientific Computing I

Location: http://people.sc.fsu.edu/~jburkardt/classes/math2070_2019/nonlinear_regula_falsi/nonlinear_regula_falsi.pdf

---

Sometimes one side far outweighs the other!

The regula falsi method

Bisection only uses the sign of a function when searching for a root. Can we speed up the search using more of the information we see?

1 A Scorecard for Bisection

We used bisection on several different functions. In all cases, our tolerance for the interval size was xtol = 10^{-6}, for the function magnitude was xtol = 10^{-6}, with an iteration limit itmax = 50. For each problem, we had to specify starting points xn and xp where the function was negative and positive. The bisecton function returned a satisfactory solution, and reported the number of steps required.

function	xn	xp	it
cubic()	1	2	**21**
hump()	10	0	**25**
kepler()	10	0	**24**
lambert()	0	2	**23**
trig()	1	0	**20**
wiggle()	8	0	**30**

While bisection gives us very reliable performance, it can be slower and more expensive than other methods. Can we modify the bisection algorithm in a way that keeps its reliability but speeds up the results?

2 Regula Falsi - A better idea?

On each step of the bisection method, we start with values `(xn,f(xn))` and `(xp,f(xp))`, but we only use the values of `xn` and `xp` to make our next estimate for the root:

$$x = \frac{xn + xp}{2}$$

Suppose that the magnitude of $f(xn)$ is much less than that of $f(xp)$. This is a clue that suggests that the root might be closer to `xn` than to `xp`. To quantify this, we compute the linear function that goes through our two data points `(xn,f(xn))` and `(xp,f(xp))`:

$$y(x) = \frac{f(xn)\,(xp - x) + f(xp)\,(x - xn)}{xp - xn}$$

Now if we set $y = 0$, we can solve the above equation to determine where the linear model predicts the root to be:

$$x = \frac{f(xn)\,xp - f(xp)\,xn}{f(xn) - f(xp)}$$

If we replace the bisection formula for the new x by this formula, we have the method known as **regula falsi**, also known as the *rule of false position*.

In regula falsi, we are giving up the guaranteed halving of the size of the interval in exchange for a better guess as to the location of the root. Since the new interval is no longer guaranteed to be exactly half the size of the old one, the α ratio could be anywhere between 0 and 1. We hope that regula falsi tends to give us values of α that are smaller than $\frac{1}{2}$.

Counting the number of iterations, and monitoring the value of α, will give us an idea of whether our regula falsi code is outperforming the bisection code.

3 Pseudocode for Regula Falsi

A pseudocode description of regula falsi is almost identical to our bisection code:

```
1  Input: function f, negative and positive arguments xn and xp, xtol, ftol, itmax
2
3  Begin loop
4
5    Set x to ( f(xn)*xp - f(xp)*xn ) / (f(xn) - f(xp) )
6
7    set old = | xp - xn |
8
9    If f(x) is negative, replace xn by x
10   else replace xp by x
11
12   set new = | xp - xn |
13   set alpha = new / old
14   possibly print alpha
15
16   if new is less than xtol and |f| is less than ftol, success
17   else if it > itmax, failure
18   else repeat
19
20 End loop
21
22 Output: updated values of xm and xp and it
```

Listing 1: regula_falsi_pseudocode.txt

4 A Scorecard for Regula Falsi

For comparison, we repeat our tests using regula falsi (RF), hoping to get successful runs with fewer iterations (it):

function	xn	xp	it (Bi)	it(RF)	Comment
cubic()	1	2	21	**50**	xtol test failed
hump()	10	0	25	**50**	xtol test failed
kepler()	10	0	24	**50**	xtol test failed
lambert()	0	2	23	**41**	
trig()	1	0	20	**50**	xtol test failed
wiggle()	8	0	30	**19**	

These results are very unsatisfactory. Only the `wiggle()` function showed improvement, and four of our tests actually failed. Since the linear approximation idea seems intelligent, it's worth trying to understand the failures and see if we can avoid them.

5 How Regula Falsi Can Fail:

If we print the value of `alpha` during the `humps()` test, here's some of what we see. Remember that `alpha` compares the new interval to the old one, and we are hoping for values much less than 0.5:

```
  F(10) = -5.9773
  F(0) = 5.17647
Step 1: alpha = 0.4641
Step 2: alpha = 0.468366
Step 3: alpha = 0.496983
Step 4: alpha = 0.623868
Step 5: alpha = 0.664884
Step 6: alpha = 0.729801
Step 7: alpha = 0.809779
Step 8: alpha = 0.883887
Step 9: alpha = 0.936835
Step 10: alpha = 0.968109
  ...things get worse and worse...
Step 49: alpha = 1
Step 50: alpha = 1

  After 50 iterations:
  F(1.29955) = -6.21725e-15
  F(1.0934) = 8.48293
```

We see that, rather than having `alpha` much less than 0.5, it quickly grows until it reaches 1. A value of 1 suggests that the interval is not being reduced at all. Actually, there is a tiny reduction, and `alpha` is not 1, but rather 0.999999999, but that's pretty useless to us.

Can we understand what is happening here?

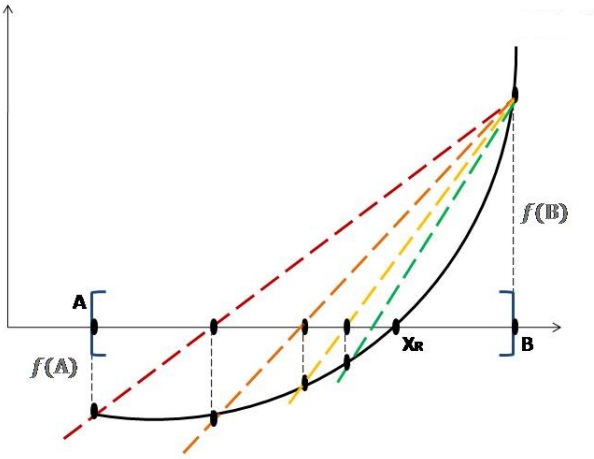
6 A lopsided prediction

We're trying to do two things: *find the root,* **and** *bracket the root.*

Regula falsi does a much better job of estimating where the root is. However, it doesn't always pay attention to the task of shrinking both ends of the interval of uncertainty.

If the function $f(x)$ is convex over the interval $[a, b]$, then every secant prediction will always fall to the same side of the root. This means that only one side of the interval of uncertainty will shrink, while the other will

tend to stay fixed. Once one side is very small, the values of `alpha` will tend to 1 as the secant prediction never lands on the "far" side of the root.



A convex function can defeat regula falsi!

7 The Regula Falsi method - version #2

One way to diagnose the problem we encountered is to monitor the value of `alpha`. If our regula falsi step gave us a value greater than 0.5, we can insist that the following step be a bisection step. That way, we guarantee that the regula falsi code will, at worst, take twice as many steps as a bisection code.

This modification is easy, since we already have `alpha` available. The portion of the code that decides which method to use looks like this:

```
1     bisect = ( 0.50 < alpha );
2
3     xold = x;
4     if ( bisect )
5       x = ( xn + xp ) / 2.0;
6     else
7       x = ( xn * f(xp) - xp * f(xn) ) / ( f(xp) - f(xn) );
8     end
```

Listing 2: extract from regula_falsi2.m

8 A Scorecard for Regula Falsi2

We now can test out the regula falsi #2 code (RF2):

function	xn	xp	it (Bi)	it(RF)	it (RF2)	Comment
cubic()	1	2	21	50	**18**	No longer failing!
hump()	10	0	25	50	**20**	No longer failing!
kepler()	10	0	24	50	**26**	No longer failing!
lambert()	0	2	23	41	**18**	better than Bi
trig()	1	0	20	50	**36**	No longer failing!
wiggle()	8	0	30	19	**27**	slightly better than Bi

By adding the bisection option, we have fixed the failures we saw before, but now the good results for the other functions have significantly deteriorated. Perhaps we could try to use bisection less often, with a test like `bisect = ( 0.70 < alpha );`?

9 The Regula Falsi method - version #3

There's an alternative way to deal with our problem. If the function seems to be too convex for regula falsi to work well, we could try to adjust the function. Presumably, the function value on the "neglected" side of the interval is rather large. If we notice that we are not reducing that side of the interval, we could simply rescale that function value, dividing it by 2, and repeatedly, if necessary, until an estimated root is predicted on that side.

To do this, imagine in the diagram above that we halve the size of `f(B)`. In that case, the intersection of the secant line with the x-axis will move towards B. If we halve it often enough, eventually the intersection will be between the root and `B`, and so we will be able to shrink the interval.

The details are a little involved. The interesting action happens when we update one endpoint. At that time, we need to check whether we have updated that endpoint at least twice in a row, in which case we are going to decrease the function value at the opposite end. To see the whole story, here is the full MATLAB code:

```
function [ xn, xp, it ] = regula_falsi2 ( f, xn, xp, xtol, ftol, itmax )

  it = 0;
  fn = f ( xn );
  fp = f ( xp );
  x = xn;
  fx = f ( xn );

  while ( true )

    it = it + 1;
    xold = x;
    fxold = fx;
    x = ( xn * fp - xp * fn ) / ( fp - fn );
    fx = f ( x );
    old = abs ( xp - xn );

    if ( fx < 0.0 )
      xn = x;
      fn = fx;
      if ( fxold < 0.0 )
        fp = fp / 2.0;
      end
    else
      xp = x;
      fp = fx;
      if ( fxold > 0.0 )
        fn = fn / 2.0;
      end
    end

    new = abs ( xp - xn );
    alpha = new / old;
    fprintf ( 1, 'Step %d: alpha = %g\n', it, alpha );

    if (( new <= xtol ) \&\& ( abs ( f ( x ) ) <= ftol ) )
      return
    end
    if ( itmax <= it )
```

```
40          return
41        end
42
43      end
44
45      return
46  end
```

Listing 3: regula_falsi2.m

10 A Scorecard for Regula Falsi3

We now can test out the regula falsi #3 code (RF3).

function	xn	xp	it (Bi)	it(RF)	it(RF2)	it (RF3)
cubic()	1	2	21	50	18	**6**
hump()	10	0	25	50	20	**9**
kepler()	10	0	24	50	26	**8**
lambert()	0	2	23	41	18	**9**
trig()	1	0	20	50	36	**6**
wiggle()	8	0	30	19	27	**15**

With some careful programming, we seem to have recovered the reliability of the bisection method, and the speed of regula falsi, while eliminating the difficulty caused by convexity.

11 No Assignment for this lab!