MAT 4310 - Root Finding and Interpolation Capstone

Jefferson Ong

4/7/2020

Question 1:

There are several suggested production level codes' that can be programmed for general purpose root finding. All assume that there is an interval that brackets the root to start. The code does either linear or quadratic inverse interpolation" to produce a new estimate of the root. If the convergence is slow, then the method reverts to bisection for a few steps to cut down on the size of the interval.

Your job: Consider the two programs

- *Brent developed in the 1970s: quadratic interpolation + secant (linear interpolation) + bisection)*
- *Le developed in 2012: simplification of Brent by reducing the number of tests to see which interpolation to use, and staying with quadratic interpolation longer.*

Test the two algorithms using the root finding problems below. Include answers to the following questions: Does the code find a (correct) solution to the desired accuracy ? If not, why not? How many function evaluations were needed? How many bisection, linear and quadratic steps were taken? Hint: Graphing or solving in Maple may help answer these questions.

(a) $f(x) = \cos(x) - x^3$ on [-4,4] accurate to 1.0E-10.

(b) $f(x) = (x-1)(x-2)(x-3)(x-4)(x-5)(x-6)(x-7)$ on [-.5,7.5] accurate to 1.0E-10.

(c) $f(x) = (1-x)e^{-1/(x-1)^2}$ on [0,3] accurate to 1.0E-10.

Le claims that his simplification of Brent is better because it eliminates a lot of the complicated conditional testing and still converges either about the same or faster than Brent's original algorithm. Does your experimentation support this claim?

```
import math

def f(x):
  return (x - 1)*(x - 2)*(x - 3)*(x - 4)*(x- 5)*(x - 6)*(x - 7.0000000001)#function for which root is d

def brent(a,b,relerr,abserr):
# Calculates the root of an externally provided function f(x)
# in the interval [a,b]
# with specified relative and absolute error tolerances.
    max_func_evals = 500 #the user should change this value if more evaluations are desired

    #calculate machine epsilon and make sure tolerance is not too small
    epsilon = 1.0
    while 1 + epsilon > 1.0:
         epsilon = epsilon / 2.0
    epsilon = 2.0 * epsilon
    if ((relerr < 10.0*epsilon) or (abserr < 0.0)):
```

```
        print("Error in specifying the tolerances.")

    count = 0
    width = abs(a-b)
    fb = f(b)
    fa = f(a)
    num_func_evals = 2

    if fb == 0.0:  # see if the root is already found
        print(" The root is", b)
        Print("and the function value at the root is", 0.0)
        return
    if fa * fb >= 0.0:  # make sure the root is between the endpoints/
        print("[b,c] does not bracket the root.")
        return
    if abs(fa) < abs(fb):  # keep the value closest to the axis
        temp = a
        a = b
        b = temp
        temp = fa
        fa = fb
        fb = temp
    initial_residual = fa
    c = a
    fc = fa
    mflag = True
    lin = 0
    quad = 0
    while num_func_evals < max_func_evals:
        if fa != fc and fb != fc:  #quadratic interpolation
            s = (a*fb*fc)/((fa-fb)*(fa-fc)) + (b*fa*fc)/((fb-fa)*(fb-fc)) + (c*fa*fb)/((fc-fa)*(fc-fb))
            quad +=1
        else: # linear (secant) interpolation
            s = b - fb * (b-a) / (fb-fa)
            lin +=1
        bisect = False

        #see if we need to switch to bisection
        if s < (3.0*a + b)/4.0 or s > b:
            bisect = True
        elif mflag and abs(s-b) >= abs((b-c)/2.0):
            bisect = True
        elif not mflag and abs(s-b) >= abs((c-d)/2.0):
            bisect = True
        elif mflag and abs(b-c) < abserr:
            bisect = True
        elif not mflag and abs(c-d)< abserr:
            bisect = True

        # do bisection if we should
        if bisect:
            s = a + (b-a)/2.0
            mflag = True
```

```
        else:
            mflag = False

        #set up for next iteration
        fs = f(s)
        num_func_evals += 1
        d = c
        c = b
        fc = fb
        if fa*fs < 0.0:
            b = s
            fb = fs
        else:
            a = s
            fa = fs
        if abs(fa) < abs(fb):
            temp = a
            a = b
            b = temp
            temp = fa
            fa = fb
            fb = temp

        # check to see if we found a root or a pole or if we did the max function evaluations
        tol = max(abserr, abs(b)*relerr)
        if abs((b-a)/2.0) <= tol:
            if abs(fb) > abs(100.0*initial_residual):
                print("There is a pole at", b)
                return
            else:
                print(" The root is", b, "and the function value at the root is", fb)
                print(" The number of function calls:", num_func_evals)
                print("linear steps:", lin, "quadratic steps:", quad)
                return
        if (num_func_evals >= max_func_evals):
            print("Too much work. The number of function calls:", num_func_evals)
            print("There is a root between:", a, b)
            return
        if abs(fb) ==0.0:
            print(" The root is", b)
            print("and the function value at the root is", fb)
            print(" The number of function calls:", num_func_evals)
            print("linear steps:", lin, "quadratic steps:", quad)
            return


a = -.5
b = 7.5
relerr = 1*10**-10
abserr = 1*10**-10
brent(a, b, relerr, abserr)
```

```
##  The root is 7.0000000001
## and the function value at the root is 0.0
```

```
##  The number of function calls: 3
## linear steps: 1 quadratic steps: 0
```

```
import math

def f(x):
  return (x - 1)*(x - 2)*(x - 3)*(x - 4)*(x- 5)*(x - 6)*(x - 7.0000000001)#function for which root is d

def LE_brent(a,b,relerr, abserr):
# Calculates the root of an externally provided function f(x)
# in the interval [a,b]
# with specified relative and absolute error tolerances.
    max_func_evals = 500 #the user should change this value if more evaluations are desired

    #calculate machine epsilon and make sure tolerance is not too small
    epsilon = 1.0
    while 1 + epsilon > 1.0:
         epsilon = epsilon / 2.0
    epsilon = 2.0 * epsilon
    if ((relerr < 10.0*epsilon) or (abserr < 0.0)):
        print("Error in specifying the tolerances.")
        return

    fb = f(b)
    fa = f(a)
    num_func_evals = 2
    initial_residual = fb
    quad = 0
    lin = 0

    while num_func_evals < max_func_evals:
        # use midpoint as third point
        c = a + (b-a)/2.0
        fc = f(c)
        if fa != fc and fb != fc:  #quadratic interpolation
            s = (a*fb*fc)/((fa-fb)*(fa-fc)) + (b*fa*fc)/((fb-fa)*(fb-fc)) + (c*fa*fb)/((fc-fa)*(fc-fb))
            quad +=1
        else:   # linear (secant) interpolation
            s = b - fb * (b-a) / (fb-fa)
            lin +=1
        fs = f(s)
        num_func_evals += 2

        #arrange points to keep root between a and b
        if c > s:
            temp = c
            c = s
            s = temp
            temp = fc
            fc = fs
            fs = temp
        if fc * fs < 0.0:
            a = c
            fa = fc
```

```
            b = s
            fb = fs
        elif fs * fb < 0.0:
            a = s
            fa = fs
        else:
            b = c
            fb = fc

        tol = max(abserr, abs(b)*relerr)


        if abs((b-a)/2.0) <= tol:
            if abs(fb) > abs(100.0*initial_residual):
                print("There is a pole at", b)
                return
            else:
                print("The root is", b)
                print("and the function value at the root is", fb)
                print("The number of function evaluations:", num_func_evals)
                print("linear steps:", lin, "quadratic steps:", quad)
                return
        if (num_func_evals >= max_func_evals):
            print("Too much work. The number of function calls was", num_func_evals)
            print("There is a root between:", a, b)
            return
        if abs(fb) ==0.0:
            print(" The root is", b)
            print("and the function value at the root is", fb)
            print(" The number of function evaluations:", num_func_evals)
            print("linear steps:", lin, "quadratic steps:", quad)
a = -.5
b = 7.5
relerr = 1*10**-10
abserr = 1*10**-10
LE_brent(a, b, relerr, abserr)
```

```
## The root is 1.0
## and the function value at the root is 0.0
## The number of function evaluations: 20
## linear steps: 0 quadratic steps: 9
```

(a)

Brent:

```
(' The root is', 0.8654740331016133, 'and the function value at the root is', 3.4416913763379853e-15)
(' The number of function calls:', 35)
('linear steps:', 31, 'quadratic steps:', 2)
```

Le:

```
('The root is', 0.8654740331016145)
('and the function value at the root is', -2.220446049250313e-16)
('The number of function evaluations:', 22)
('linear steps:', 0, 'quadratic steps:', 10)
```

Maple:

```
('The root is', 0.8654740331)
```

- Graph is a quadratic where the x = 0 and y = 1. As we can see, the Brent and Le algorithms are able to find the root of the function to the desired accuracy. We can see that it took the Brent algorithm 35 function evaluations while the Le was able to do it in 22 evaluations. They differ only at around 14 digits. It took Brent 31 linear steps while none for Le, for the quadratic steps, 2 for Brent but 10 for Le.

(b) Brent:

```
(' The root is', 7.0)
('and the function value at the root is', 0.0)
(' The number of function calls:', 3)
('linear steps:', 1, 'quadratic steps:', 0)
```

Le:

```
('The root is', 1.0)
('and the function value at the root is', 0.0)
('The number of function evaluations:', 20)
('linear steps:', 0, 'quadratic steps:', 9)
```

Maple:

```
('The root is', 1,2,3,4,5,6,7)
```

- Graph touches the x axis at x = 1, 2, 3, 4, 5, 6, 7. This is obvious from looking at the function itself. We can see that Brent and Le both landed only a root but stopped after finding one. They are correct in that sense. It took Brent quickly to reach the 7 while only 3 function evaluations, 1 linear step, and taking no quadratic. On the other hand, it took Le several(20) evaluations, no linear, and 9 quadratic steps to reach 1. In this case, Le is not more efficient.

(c) Brent:

```
(' The root is', 0.9831404360069544)
('and the function value at the root is', 0.0)
(' The number of function calls:', 11)
('linear steps:', 7, 'quadratic steps:', 2)
```

Le:

```
(' The root is', 1.0098393223851945)
('and the function value at the root is', -0.0)
(' The number of function calls:', 12)
('linear steps:', 0, 'quadratic steps:', 5)
```

Maple:

- Maple did not evaluate to any particular root. The graph shows this, when the y = 0 when x is between 0.5 and 1.5, meaning that all of these values are possible roots. We can see that the Brent and Le algorithm are within this range. It took the Brent 11 evaluations, 7 linear, and 2 quadratic while taking the Le 12 evaluation, no linear, and 5 quadratic. It's difficult to say for certain that the Le is more efficient since quadratic steps are more complicated that linear. However from a pure calculation standpoint, it took the Brent (7 + 2) * 11 = 99 steps, while Le took 5(12) = 60 steps. A bit more efficient.

Conclusion: We can see that Le isn't always better, it really depends on the function and the range we are given. The first function is a "standard" problem with one root and the Le algorithm performed much better. However in the second function, Brent performed much better than Le, but the function had 7 roots. The

last function doesn't show us too much more, where the Le performed slightly better but not by much. So we don't have enough evidence to support the claim.

Question 2:

Modify the function from part b in problem 1 by changing $(x - 7)$ to $(x - 7.0000000001)$, i.e. This is a very small perturbation of one root of this seventh degree polynomial. Run Brent and Le again. Do you find the same roots with the same amount of work on the part of the program? Describe what happens.

1b) Brent:

```
(' The root is', 7.0000000001)
('and the function value at the root is', 0.0)
(' The number of function calls:', 3)
('linear steps:', 1, 'quadratic steps:', 0)
```

Le:

```
('The root is', 1.0)
('and the function value at the root is', 0.0)
('The number of function evaluations:', 20)
('linear steps:', 0, 'quadratic steps:', 9)
```

- We found that the small pertubation of one root does not show any effect on both algorithms. The Brent algorithm's root just changes to 7.0000000001 from the previous 7. While the Le algorithm has no change. No increase or decrease on function evaluation, linear, and quadratic steps.

Question 3:

Suppose we want to write code to find the (positive) square root and cube root of a number, a. We can find these two numbers by finding the roots of:

- $f(x) = x^2 - a$
- $f(x) = x^3 - a$.

We can create a designer code using Newton's Method. For example, for $f(x) = x^2 - a$, the Newton iteration is

$$x_{new} = x - \frac{f(x)}{f'(x)} = x - \frac{x^2 - a}{2x}$$

For $f(x) = x^3 - a$, the Newton iteration is

$$x_{new} = x - \frac{f(x)}{f'(x)} = x - \frac{x^3 - a}{3x^2}$$

Use this idea to write an efficient code to print out the square root and cube root of a correct to 0.00001, using $a/2$ as the initial x-value for the square root and $a/3$ as the initial x-value for the cube root. Simplify the Newton step so that there are no subtractions to avoid possible cancellation error. Test your code on a variety of numbers between 0 and 1000. Use the following function definition statement:

```
def roots(a):
```

The code should print the following statement (assuming 5 was input as a):

"The square and cube roots of 5 are: 2.23606798 and 1.70997595"

- We can see that we can reform the Newton iteration to isolate 'a'.

$$(((xnew - x) * -nx^{n-1}) - x^n) * -1 = a$$

(a)ˆ(1/2) = square. So for example, we want the square root of 5. We will take the initial value of x = 5/2 = 2.5, We then feed this into to loop. We square this, so (2.5)ˆ2 = 6.25, add the original number, 5. So 6.25 + 5 = 11.25, then divide by 2x, 11.25/(2* 2.5) = 2.25 as our xnew value. Compare to the real sqrt(5) = 2.3606.

(a)ˆ(1/3) = cubic

```
def roots(a):
  sx = a/2
  cx = a/3
  max_its = 100
  for its in range(1,max_its+1):
    sx = (sx**2 + a)/(2*sx)
    cx = (2*cx**3 + a)/(3*cx**2)
  if (abs(sx**2 - a) < .00001 and abs(cx**3-a) < .00001):
    print("The square and cube roots of 5 are", sx,"and", cx)

roots(8)
```

```
## The square and cube roots of 5 are 2.82842712474619 and 2.0
```

https://repl.it/@ongjk/TubbyWrithingRuntimeerror

$$x_{new} = x - \frac{f(x)}{f'(x)} = x - \frac{x^2 - a}{2x} = x * \frac{2x}{2x} - \frac{x^2 - a}{2x} = \frac{2x^2}{2x} - \frac{x^2 - a}{2x} = \frac{x^2 + a}{2x}$$

So this is the formula for the square root Newton's Method. The cube root on the other hand is this,

$$x_{new} = x - \frac{f(x)}{f'(x)} = x - \frac{x^3 - a}{3x^2} = x * \frac{3x^2}{3x^2} - \frac{x^3 - a}{3x^2} = \frac{3x^3}{3x^2} - \frac{x^3 - a}{3x^2} = \frac{2x^3 + a}{3x^2}$$