Bisection Method

April 20, 2022

1 The Bisection Method.

An algorithms for solving non-linear equations, applied to any continuous function we know two values with opposite signs.

To find a solution of the equation

$$f(x) = 0$$

We find the midpoint c of the interval $[a, b]$. Suppose that $f(c) \neq 0$ (otherwise we have found a solution). Either $f(a) \times f(c)$ is negative, so that the interval $[a, c]$ must contain a solution of the equation, or $f(c) \times f(b)$ is negative, so that the interval $[c, b]$ contains a solution of the equation. Therefore, we can construct an interval containing solution, and whose length is half the length of the interval $[a, b]$. By iterating this construction, we obtain a sequence of intervals with the expected properties.

1.0.1 Implementation

We define `bisection_method` function

[1]:
```
def bisection_method(f, start_point, end_point, number_of_iterations = 10):

    # if f(a) × f(b) is positive, there are no roots

    if (f(start_point) * f(end_point) > 0) :
        return None


    for i in range(number_of_iterations):
        mid_point = (start_point + end_point) / 2

        if (f(mid_point) == 0) :
            return mid_point

        elif (f(start_point) * f(mid_point) < 0) :
            end_point = mid_point

        elif (f(start_point) * f(mid_point) > 0) :
            start_point = mid_point
```

```
    return mid_point
```

1.1 Test with an example

Use the bisection method to find a solution to the equation

$$x^2 + ln(x) = 0$$

using 4 iterations of the bisection method over the interval $[0.5, 1]$

Find maximum error bound ε, error measured to exact solution and plot $f(x)$

[2]:
```
# Define f(x)

f(x) = x^2 + ln(x)
```

[3]:
```
our_solution = bisection_method(f, 0.5, 1, 4)
our_solution
```

[3]: 0.656250000000000

Exact value

[4]:
```
exact_solution = solve(f(x) == 0, x)
exact_solution
```

[4]: [x == -sqrt(-log(x)), x == sqrt(-log(x))]

As seen, `sage` can't find a numerical solution for the equation using `solve` function, here is the explination and alternative method from `sage` documentation

1.1.1 Solving Equations Numerically

Often times, `solve` will not be able to find an exact solution to the equation or equations specified. When it fails, you can use `find_root` to find a numerical solution. For example, solve does not return anything interesting for the following equation:

```
sage: theta = var('theta')
sage: solve(cos(theta)==sin(theta), theta)
[sin(theta) == cos(theta)]
```

On the other hand, we can use find_root to find a solution to the above equation in the range $0 < \varphi < \pi/2$:

```
sage: phi = var('phi')
sage: find_root(cos(phi)==sin(phi),0,pi/2)
0.785398163397448...
```

[5]:
```
exact_solution = find_root(f, 0.5, 1)
exact_solution
```

[5]: 0.6529186404192052

Find error and maximum error bound ε

[6]:
```
error = abs(our_solution - exact_solution)
error
```

[6]: 0.00333135958079478

[7]:
```
maximum_error_bound = (1 - 0.5) / 2^4
maximum_error_bound
```

[7]: 0.0312500000000000

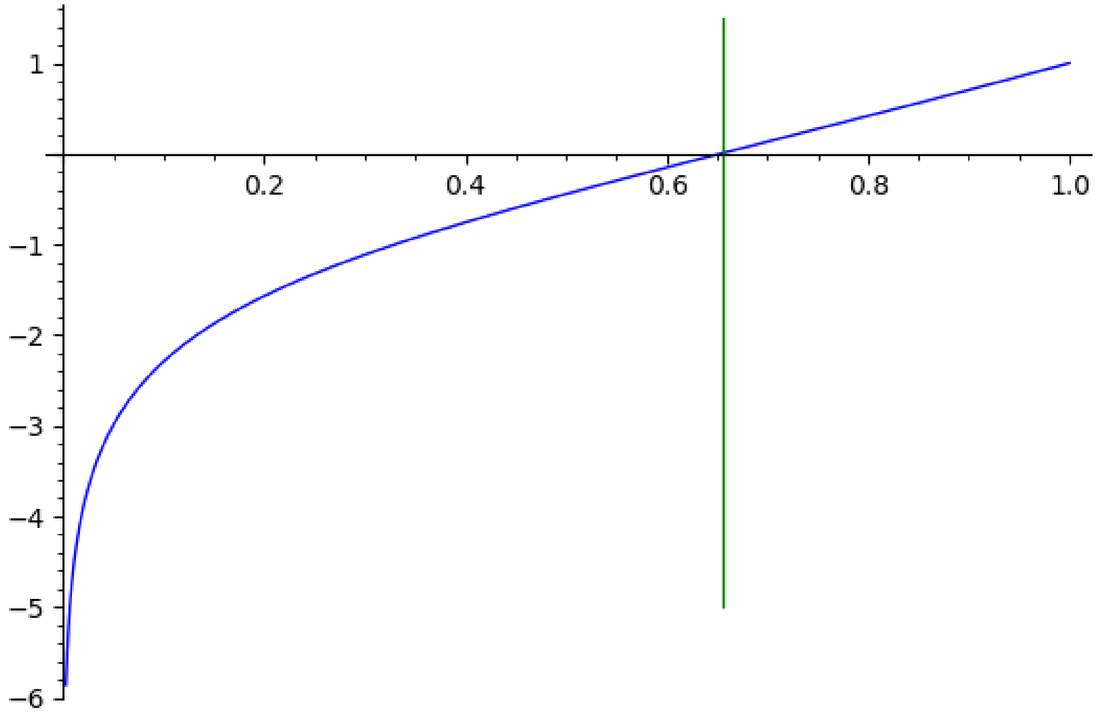
As seen, error is much less than maximum error bound

plot the function and the solution

[8]:
```
p1 =  plot (f, x)

p2 = line([(our_solution, -5), (our_solution, 1.5)], color = "green")

my_plot = p1 + p2

my_plot
```

```
verbose 0 (3835: plot.py, generate_plot_points) WARNING: When plotting, failed
to evaluate function at 100 points.
verbose 0 (3835: plot.py, generate_plot_points) Last error message: 'can't
convert complex to float'
```

[8]:



1
0.2
0.4
0.6
0.8
1.0
−1
−2
−3
−4
−5
−6