3 Bisection. Theory and implementation

3.1 Introductory words on root finding

Now it is finally the time to start looking into some mathematical problems. In this course I will start with a discussion of various approaches to find roots of (transcendental) equations of the form

$$f(x) = 0, \quad x \in X \subseteq \mathbf{R}, \tag{3.1}$$

and f will be generally a real-valued function. I write "transcendental" to emphasize that *polynomial* equations

$$a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 = 0, \quad a_j \in \mathbf{R}$$

have their own deeply developed methods to approximate the roots. This is why, for instance, if you asks Mathematica to solve, e.g., $x^6 + 3x^2 - x + 10 = 0$, it will immediately return 6 roots (this is Mathematica code, not Python!):

```
>>>NSolve[x^6+3x^2-x+10==0,x]
#the result it
{{x -> -1.30082 - 0.910873 I}, {x -> -1.30082 + 0.910873 I}, {x ->
   0.0428653 - 1.30481 I}, {x -> 0.0428653 + 1.30481 I}, {x ->
   1.25795 - 0.862663 I}, {x -> 1.25795 + 0.862663 I}}
```

On the other hand, an attempt to solve $\tan x = -2x$ will result in the following:

```
>>>NSolve[Tan[x]==x,x]
#the result is
NSolve::nsmet: This system cannot be solved with the methods available to NSolve.
```

Hence the first warning: there exists no universal algorithm to find roots of (3.1) in general, and usually a computer should be assisted by human experience, guesses, and, sometimes, luck, to achieve the desired goal.

For the following I will need a few mathematical facts. First, *a root* of (3.1) is a (real) constant $\hat{x} \in \mathbf{R}$ such that it turns equation (3.1) into identity. Note that in many problems we have expressions of the form "left hand side is equal to the right hand side," as in, e.g., $\tan x = -2x$, but we can always turn such equations into (3.1) by moving all the terms to one side. Geometrically to solve problem (3.1) means to find the points of intersection of the graph of f with the x-axis, whereas solving $F(x) = G(x)$ geometrically means finding the points of intersections of graphs F and G.

We call function f *continuous* at a point x_0 if for any $\epsilon > 0$ there is constant $\delta > 0$ such that for all x satisfying $|x - x_0| < \delta$ their images satisfy $|f(x) - f(x_0)| < \epsilon$. The function is continuous on (a, b) is it is continuous at each point of this interval. The function is continuous on $[a, b]$ if it is continuous everywhere inside the interval, continuous from the right at a (this means that in the definition above we take x only on the right side of point a), and continuous from the left at b. This is a technical (and not easy) mathematical definition; hence it is also important to have some intuitive understanding of continuous functions. For us, a continuous function on $[a, b]$ will be such function whose graph can be drawn without taking your pencil (or pen) from the paper and does not escape to either minus or

Math 488/688: Numerical Analysis
e-mail: artem.novozhilov@ndsu.edu. Fall 2021

plus infinity. Most of our familiar functions are continuous in their natural domains, an example of a *discontinuous* function is, e.g., $f(x)=1$ if $x\geq 0$ and $f(x)=-1$ if $x<0$ (draw its graph).

For us will be important the following basic theorem (recall that *theorem* in mathematics is an important true statement).

Theorem 3.1 (Intermediate value theorem)**.** *Let f be continuous on $[a,b]$, and assume that $f(a)f(b)<0$. Then there is point $\hat{x}\in(a,b)$ such that*

$$f(\hat{x})=0.$$

Idea of a proof. The proof of this theorem relies on the fact that we can always divide the interval $[a,b]$ into two parts, with the middle being $c=(a+b)/2$, and at the point c function f is either zero, negative, or positive. If it is zero, we are done, if it is negative or positive then we shrink the interval in the way to keep again opposite sings of f at the ends of the new interval (if, say, $f(a)<0, f(b)>0$ initially and $f(c)<0$ then I reassign $a=c, b=b$; if $f(c)>0$ then $a=a, b=c$) and continue this process. In this construction we get a sequence of nested closed intervals whose lengths tend to 0. By the *completeness* property of the real line (this is the deepest point in this sketch of a proof!) there must be only one point $\hat{x}$ that belongs to all the intervals in the limit, and due to the continuity of f, it must be true that $f(\hat{x})=0$. ■

Or intuitively, if f is continuous on $[a,b]$, and the signs of f are opposite at the ends of the interval, then drawing its graph must result in at least one intersection with the x-axis (make a drawing).

Anyway, it is not the subtleties of the proof above that should worry us now (you should take a rigorous analysis class to appreciate them), but the algorithmic nature of the proof. In words, the intermediate value theorem guarantees that if the conditions are true then 1) we must have a root in the given interval; and, moreover, 2) we can build a sequence of shrinking intervals each of which must have a common root. This sounds like something that can be efficiently turned into a computer program, which I discuss next.

3.2 Bisection algorithm

Never start writing your code before you have a clear understanding of the exact steps of the algorithm you want to realize, the initial data for your program, and the condition that would guarantee that your program runs in a finite time. Another aspect is to be prepared to carefully test your program even before you produced any code. Your test should be something for which you already know the exact answer. For the bisection algorithm I will take the equation

$$f(x)=x^2-2=0,$$

for which f is continuous, $f(1)=-1<0$, $f(2)=2>0$, and therefore the interval $[1,2]$ must have a root, and of course I know this root, which is $\hat{x}=\sqrt{2}$, and which can be approximated (by, e.g., again using Mathematica) as

$$\sqrt{2}\approx 1.41421356237309504880$$

to have 20 correct digits after the decimal point. So in a perfect world, my program should return an answer, which should be close to the approximation above, and, ideally, I should also be able to somehow control the number of correct digits.

(A side remark: I hope it is clear that computers cannot in general operate with numbers like $\sqrt{2}$, only with their rational approximations. I will discuss these issues later in the course, for now just do not expect to see $\sqrt{2}$ as an output of my (or your) program. A deeper truth is that quite a few modern mathematical software, such as Mathematica, Maple, Matlab, etc are capable performing operations with exact irrational numbers, but this is a different story.)

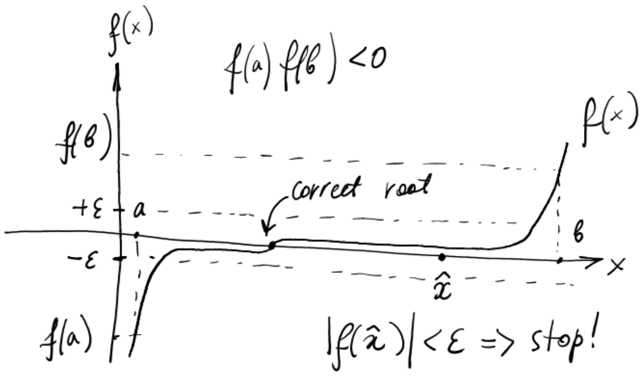
So, I chose my test function (usually, it should be more than one, of course, you should always try to come up with some "difficult" examples to test your code). The initial data must be function itself and an interval $[a, b]$ such that $f(a)f(b) < 0$. For my test function this is the interval $[1, 2]$.

The most not obvious point here is to choose a condition for the program to stop. Certainly, if I am lucky enough to find an $\hat{x}$ such that $f(\hat{x}) = 0$ I have to stop and return $\hat{x}$ (which, incidentally, could be not the exact root), but in most cases this situation will never happen, my intervals will be halved and halved without finding the exactly this point that terns f into zero.

One definite possibility is to quit the computation when the absolute value of f evaluated at some point $\hat{x}$ is below some chosen threshold:

$$|f(\hat{x})| < \epsilon \implies \hat{x} \text{ is our approximation of the (unknown) root.} \tag{3.2}$$

While it is one of the natural conditions for many other computations, for finding the roots of scalar equations it is not the best one. There are two reasons for this: even if $|f(\hat{x})|$ is very small, one can be still quite far from the root (see the figure). The second reason is that the knowledge of ϵ does not tell us much about the number of correct digits out computation has produced.



Another possibility is to observe the subsequent approximations of the quantity we are after, call them $c_1, c_2, \ldots, c_n, \ldots$, the stop condition can be

$$|c_{n+1} - c_n| < \epsilon, \tag{3.3}$$

or, better for some situations,

$$\left| \frac{c_{n+1} - c_n}{c_{n+1}} \right| < \epsilon, \tag{3.4}$$

where now the *relative* error is computed.

All these conditions are used in various computations (as we will see later), but for the current problem there is definitely a better one. Since in the bisection method that I am about to implement I know for sure that my root is seating within a given interval, whose length l will be

$$l = \frac{|b - a|}{2^n} \tag{3.5}$$

at the n-th step, then I know for sure that my root is approximately at the center of this n-th interval with the error that does not exceed $\frac{l}{2}$. Therefore, if I know my acceptable level of the error, say ϵ, I can conservatively guarantee it by taking

$$n = \left\lfloor \log_2 \frac{|b-a|}{\epsilon} \right\rfloor + 1,$$

where $\lfloor\cdot\rfloor$ is the floor function (it takes the integer part for positive numbers).

Having said all this, I now realize that I must have another piece of initial data, namely, the *tolerance* (the acceptable level of the error), and I will stop my computation if the length of the interval at some step is below this tolerance. To be completely on the safe side, I also would like to set a limit on the number of bisections my program performs (it could happen that, due to the specific internal computer number realization the length of my interval will never be smaller than given initially tolerance, no matter how many divisions I will make; I know it sounds strange, but it is still possible). Let me take, again, very conservatively, this number to be equal 50 such that my initial interval will be reduced by the factor

$$2^{-50} \approx 10^{-15}.$$

Now everything is ready to write the code.

3.3 Bisection implementation

Here is my function. Below I will leave a few comments on the syntax I did not discuss in the previous sections.

```python
def bisection(f, a, b, epsilon = 10e-8):
    '''gets function f, the boundaries of the interval
    where a root should be, and the tolerance epsilon.
    Returns the root with the accuracy plus minus epsilon.
    '''

    max_iterations = 50 #the maximal number of iterations

    fa, fb = f(a), f(b)

    if fa * fb >= 0: #checking the conditions for the bisection
        raise ValueError("The sings of f at the boundaries should be opposite.")

    if a > b:
        a, b = b, a #make sure that a is the left boundary and b is the right one

    for i in range(max_iterations):

        dx = b - a
        mid = a + dx/2
        fmid = f(mid)

        if dx < epsilon or fmid == 0:
            return mid
        if fa * fmid < 0:
            b, fb = mid, fmid
        else:
            a, fa = mid, fmid

    raise ValueError('Too many bisections.')
```

Here are some extended comments on the above code:

- Note that one of the formal function parameters, `epsilon`, in the definition of the function already has a value: `epsilon=10e-8` (notation `10e-8` is a convenient way to write $10^{-8} = 0.00000001$). This means that this parameter has a default value, and if I am satisfied with this value, I do not need to provide it when calling this function.
- Within the code I check two possible situations when my function fails to find a root with the desired tolerance. Initially I check that the user actually submitted a and b such that $f(a)f(b) < 0$ (any one can make a typing mistake, so I do not want my function work on a problem that possibly has no solution), I also wait for all 50 iterations to make sure that required tolerance is reached. If not, I would like to make the user to be aware of it. In both of these situations I technically have an error: my function does not do what it is supposed to do. Therefore I decided to return an error. In Python I can accomplish this by using the key word `raise` followed by the function `ValueError` with the message I want the user to see.

To compare my code with another possible realization, I copied another bisection function (from a good textbook).

```
import numpy as np #for functions sign and abs

def bisection1(f, a, b, tol):

    if np.sign(f(a))==np.sign(f(b)):
        raise Exception("The scalars a and b do not bound a root")

    m = (a + b)/2

    if np.abs(f(m)) < tol:
        return m

    elif np.sign(f(a)) == np.sign(f(m)):
        return bisection1(f, m, b, tol)

    elif np.sign(f(b)) == np.sign(f(m)):
        return bisection1(f, a, m, tol)
```

This function, although produces usually a correct answer, is somewhat suboptimal from my point of view. Here are the issues I see with the code in `bisection1`:

- The function uses the so-called *recursion* whereas it calls for itself inside the body of the function. This is a legitimate approach, but here it makes the function to work until the desired level of tolerance is reached without fixing the maximal allowable number of calls. What will happen if I will set a very low tolerance? Or the required level simply cannot be reached by the nasty nature of the function?

- There are multiple instances when the same function is evaluated at the same value. For instance $f(m)$ can be computed 3 times just within the same body, these are unnecessary computations. In general, e.g., $f(a)$ can be computed multiple time during the code performance, which is pricey if the computation of f is not straightforward.
- Using condition (3.2) to stop the computation does not tell us much about the number of correct digits we get.
- There is an unnecessary comparison at the last line. If the program gets there the condition `np.sign(f(b)) == np.sign(f(m))` always will be true.

Exercise 1. Can you also criticize my code? Can it be improved?

3.4 Testing

Let me test the code now.

```
def fun1(x):
    return x*x-2

bisection(fun1, 1, 2, 10e-10)
#1.4142135619185865

bisection1(fun1, 1, 2, 10e-10)
#1.4142135623842478
```

By changing the tolerance one can see that in most cases the number of correct digits is at least `m-2` if the tolerance is `10e-m` (I hope you can figure out a rule of thumb). Using the second function actually gives us often more correct digits, but we cannot be sure how many without careful analysis of function f, which has to be done on the case by case basis.

Trying

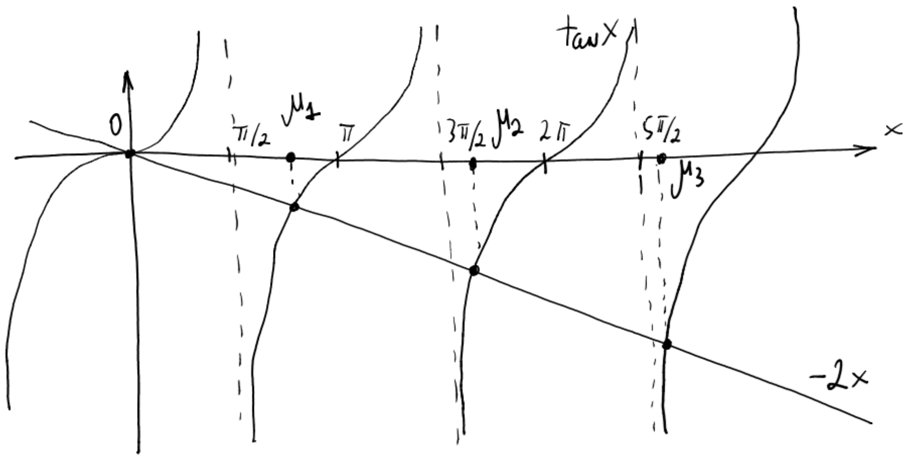
```
bisection(fun1, 1, 2, 10e-20)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-20-9437a2d716fd> in <module>
----> 1 bisection(fun1, 1, 2, 10e-20)

<ipython-input-1-d160ecb01d86> in bisection(f, a, b, epsilon)
     27             a, fa = mid, fmid
     28 
---> 29     raise ValueError('Too many bisections.')
     30 

ValueError: Too many bisections.
```

immediately gives the error “Too many bisections.” If you try it with `bisection1`, you will wait for a few seconds and will get a huge system message which boils down to “ maximum recursion depth exceeded.”

Let me go back to the equation

$$\tan x = -2x,$$

and formulate the problem of finding a few first positive solutions. How do I know they exists? Because of the graphs of the functions. In general, if you are required to solve some scalar equation, it is always advisable to make a plot first (you can see the instructions how to do it in Python in the textbook, I encourage you to practice plotting simple functions in Python). Here I have the following picture.



This indicates that I have infinitely many positive roots (I denote them μ_j), such that $\mu_1 \in (\pi/2, \pi)$, $\mu_2 \in (3\pi/2, 2\pi)$ etc. Note however that I cannot use, e.g., $\pi/2$ as the left boundary of my interval since the function at this point is not defined, the program will not run properly. I can, however, always take a close point $\pi/2 + \delta$ for some small positive δ.

```
import math
def mf(x):
    return math.tan(x)+2*x

bisection(mf,math.pi/2+0.001, math.pi,10e-10)
#1.836597203321693
```

The correct answer for μ_1 (I hope it is correct, it was found with Mathematica using computations with accuracy 30 digits after the decimal point) is 1.8365972031521257227, which again shows that I can be sure about 9 digits after the decimal point.

Now let me try to work with a discontinuous function:

$$f(x) = \begin{cases} 1, & x \geq 0, \\ -1, & x < 0. \end{cases}$$

```
def sign(x):
    if x < 0:
        return -1
    else:
        return 1

bisection(sign,-1,1)
#-2.9802322387695312e-08

bisection1(sign,-1,1,10e-3)
#RecursionError: maximum recursion depth exceeded while calling a Python object
```

This shows yet another advantage of the chosen stop condition for the computations. My function actually identified correctly the discontinuity point of the function (which sometimes can be as important as finding actual roots). The other routine again reached the maximum level of recursions and stopped.

Clearly, it is also possible to have functions, which are discontinuous and unbounded (think of, e.g., $1/x$ on the interval $[-1,1]$). In this case my function will also return an answer very close to zero. We can, however, plug this answer back into function to realize that now we are dealing with an unlimited growth. In a nutshell, one should always tinker with the problem to be sure that the found result makes sense.

3.5 Conclusions

In this section I discussed the general problem of root finding and one specific method: bisection. The clear advantage of this method is that, if properly implemented and applied to a right object, it is doomed to succeed. Why would we need something else? The truth to be told: this method is ridiculously slow, especially if one needs to solve thousands of problems with high degree of accuracy. In the next section I will introduce a few other methods, which you will analyze using Python. As a piece of advice, however, I would like to finish this section with the following sentiment: If in your actual activity you need to solve just one specific equation $f(x) = 0$ and you surely know that the root you are looking for is within certain a and b then use bisection, its elementary character and slow convergence notwithstanding. It will always lead you in the right direction, slowly yet unavoidably.