2.4 Extra exercises 2

The Bisection Method is a relatively simple algorithm to find the root of a continuous function f defined on an interval $[a, b]$. Basically, we look at the value of $f(x)$ at the midpoint of the interval $[a, b]$ and then decide whether to continue with the left half or the right half of the interval, based on the signs of $f(x)$ at both endpoints and the midpoint of the interval.

Step 1. Read about the "Bisection Method" in Wikipedia. Make sure you understand the basics of the method and how it works. Specify a new class `BisectionRootFinder` analogous to the class `NewtonRootFinder` presented in the lectures. The main properties of a `BisectionRootFinder` are the search interval $[a, b]$ and the function f. Define additional attributes which are necessary for the "Bisection Method". Create an UML class diagram for a `BisectionRootFinder` class with an appropriate constructor.

Step 2. The Wikipedia document contains some pseudo code for the bisection method. Include a `bisect` method in your class diagram based on the corresponding pseudo code example. Draw a Nassi-Shneiderman diagram to clearly see the structure of the method.

Step 3. Based on your Nassi-Shneidermann diagrams, implement the bisection method. Use the `UserFunction` class discussed in the lecture to define the given function f as `UserFunction f`. Remember that the value of $f(x)$ is then `f.valueAt(x)`. Think about problems which might arise when using different functions (e.g. a function has no root).

Step 4. Implement a test program to verify the functionality of your methods. As a test function, compute the numerical value of $\sqrt{2}$ by finding the root of the equation $x^2 - 2$. Use code such as `UserFunction("x*x-2", "x")` to define a function f. Because `UserFunction` throws a so-called exception, add the code `throws Exception` to the header of the `main` method.

Check the results of your bisection methods by comparing your computed value to the value computed by `Math.sqrt()`. Define an attribute `epsilon` implied in the first pseudo code example and check the accuracy of your computed value with respect to the value of `epsilon`. (How accurate is the computation in the second pseudo code example?)

Step 5. The Wikipedia document mentions that the number of iterations n needed to obtain the accuracy ε of a root of $f(x)$ on the interval $[a, b]$ is approximately

$$n \approx \log_2 \frac{b-a}{\varepsilon}.$$

Step 6. Based on the mathematical fact that $4\arctan(1) = \pi$, we can define π as the root of the equation

$$\tan\frac{x}{4} = 1.$$

Implement another test program to calculate the value of π. Compare your computed result to `PI` in the `Math` class.

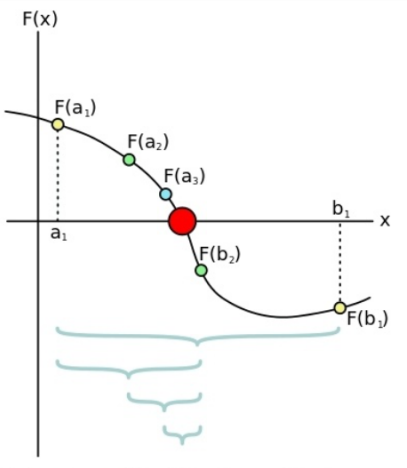
Step 7. An important equation that appears in celestial mechanics for calculating heliocentric polar coordinates is Kepler's Equation,

$$M = E - \varepsilon \cdot \sin E,$$

where the mean anomaly M and the eccentricity ε are given and the unknown eccentric anomaly E is to be calculated. If the eccentricity of Mars is 0.093 and the mean anomaly is 270°, what is the corresponding eccentric anomaly E? (Answer: about 264.7°)

Bisection method (from Wikipedia)

In mathematics, the **bisection method** is a root-finding method that applies to any continuous function for which one knows two values with opposite signs. The method consists of repeatedly bisecting the interval defined by these values and then selecting the subinterval in which the function changes sign, and therefore must contain a root. It is a very simple and robust method, but it is also relatively slow. Because of this, it is often used to obtain a rough approximation to a solution which is then used as a starting point for more rapidly converging methods.[1] The method is also called the **interval halving** method,[2] the **binary search method**,[3] or the **dichotomy method**.[4]



A few steps of the bisection method applied over the starting range $[a_1;b_1]$. The bigger red dot is the root of the function.

A few steps of the bisection method applied over the starting range $[a_1;b_1]$. The bigger red dot is the root of the function.

The method is applicable for numerically solving the equation $f(x) = 0$ for the real variable x, where f is a continuous function defined on an interval $[a, b]$ and where $f(a)$ and $f(b)$ have opposite signs. In this case a and b are said to bracket a root since, by the intermediate value theorem, the continuous function f must have at least one root in the interval (a, b).

At each step the method divides the interval in two parts/halves by computing the midpoint $c = (a+b) / 2$ of the interval and the value of the function $f(c)$ at that point. If c itself is a root then the process has succeeded and stops. Otherwise, there are now only two possibilities: either $f(a)$ and $f(c)$ have opposite signs and bracket a root, or $f(c)$ and $f(b)$ have opposite signs and bracket a root.[5] The method selects the subinterval that is guaranteed to be a bracket as the new interval to be used in the next step. In this way an interval that contains a zero of f is reduced in width by 50% at each step. The process is continued until the interval is sufficiently small.

Explicitly, if $f(c)=0$ then c may be taken as the solution and the process stops. Otherwise, if $f(a)$ and $f(c)$ have opposite signs, then the method sets c as the new value for b, and if $f(b)$ and $f(c)$ have opposite signs then the method sets c as the new a. In both cases, the new $f(a)$ and $f(b)$ have opposite signs, so the method is applicable to this smaller interval.[6]

The input for the method is a continuous function f, an interval $[a, b]$, and the function values $f(a)$ and $f(b)$. The function values are of opposite sign (there is at least one zero crossing within the interval). Each iteration performs these steps:

1. Calculate c, the midpoint of the interval, $c = \frac{a+b}{2}$.
2. Calculate the function value at the midpoint, $f(c)$.
3. If convergence is satisfactory (that is, $c - a$ is sufficiently small, or $|f(c)|$ is sufficiently small), return c and stop iterating.
4. Examine the sign of $f(c)$ and replace either $(a, f(a))$ or $(b, f(b))$ with $(c, f(c))$ so that there is a zero crossing within the new interval.

When implementing the method on a computer, there can be problems with finite precision, so there are often additional convergence tests or limits to the number of iterations. Although f is continuous, finite precision may preclude a function value ever being zero. For example, consider $f(x) = \cos x$; there is no floating-point value approximating $x = \pi/2$ that gives exactly zero. Additionally, the difference between a and b is limited by the floating point precision; i.e., as the difference between a and b decreases, at some point the midpoint of $[a, b]$ will be numerically identical to (within floating point precision of) either a or b.

```
INPUT: Function f,
       endpoint values a, b,
       tolerance TOL,
       maximum iterations NMAX
CONDITIONS: a < b,
            either f(a) < 0 and f(b) > 0 or f(a) > 0 and f(b) < 0
OUTPUT: value which differs from a root of f(x) = 0 by less than TOL

N ← 1
while N ≤ NMAX do // limit iterations to prevent infinite loop
    c ← (a + b)/2 // new midpoint
    if f(c) = 0 or (b - a)/2 < TOL then // solution found
        Output(c)
        Stop
    end if
    N ← N + 1 // increment step counter
    if sign(f(c)) = sign(f(a)) then a ← c else b ← c // new
interval
end while
Output("Method failed.") // max number of steps exceeded
```

Example: Finding the root of a polynomial

Suppose that the bisection method is used to find a root of the polynomial

$$f(x) = x^3 - x - 2\,.$$

First, two numbers a and b have to be found such that $f(a)$ and $f(b)$ have opposite signs. For the above function, $a = 1$ and $b = 2$ satisfy this criterion, as

$$f(1) = (1)^3 - (1) - 2 = -2$$

and

$$f(2) = (2)^3 - (2) - 2 = +4\,.$$

Because the function is continuous, there must be a root within the interval [1, 2].

In the first iteration, the end points of the interval which brackets the root are $a_1 = 1$ and $b_1 = 2$, so the midpoint is

$$c_1 = \frac{2+1}{2} = 1.5$$

The function value at the midpoint is $f(c_1) = (1.5)^3 - (1.5) - 2 = -0.125$. Because $f(c_1)$ is negative, $a = 1$ is replaced with $a = 1.5$ for the next iteration to ensure that $f(a)$ and $f(b)$ have opposite signs. As this continues, the interval between a and b will become increasingly smaller, converging on the root of the function. See this happen in the table below.

Iteration	a_n	b_n	c_n	$f(c_n)$
1	1	2	1.5	−0.125
2	1.5	2	1.75	1.6093750
3	1.5	1.75	1.625	0.6660156
4	1.5	1.625	1.5625	0.2521973
5	1.5	1.5625	1.5312500	0.0591125
6	1.5	1.5312500	1.5156250	−0.0340538
7	1.5156250	1.5312500	1.5234375	0.0122504
8	1.5156250	1.5234375	1.5195313	−0.0109712
9	1.5195313	1.5234375	1.5214844	0.0006222
10	1.5195313	1.5214844	1.5205078	−0.0051789
11	1.5205078	1.5214844	1.5209961	−0.0022794
12	1.5209961	1.5214844	1.5212402	−0.0008289
13	1.5212402	1.5214844	1.5213623	−0.0001034
14	1.5213623	1.5214844	1.5214233	0.0002594
15	1.5213623	1.5214233	1.5213928	0.0000780

After 13 iterations, it becomes apparent that there is a convergence to about 1.521: a root for the polynomial.

The method is guaranteed to converge to a root of *f* if *f* is a continuous function on the interval [*a*, *b*] and *f*(*a*) and *f*(*b*) have opposite signs. The absolute error is halved at each step so the method converges linearly. Specifically, if $c_1 = \frac{a+b}{2}$ is the midpoint of the initial interval, and c_n is the midpoint of the interval in the *n*th step, then the difference between c_n and a solution *c* is bounded by[8]

$$|c_n - c| \leq \frac{|b-a|}{2^n}.$$

This formula can be used to determine, in advance, an upper bound on the number of iterations that the bisection method needs to converge to a root to within a certain tolerance. The number *n* of iterations needed to achieve a required tolerance ε (that is, an error guaranteed to be at most ε), is bounded by

$$n \leq n_{1/2} \equiv \left\lceil \log_2 \left(\frac{\epsilon_0}{\epsilon} \right) \right\rceil,$$

where the initial bracket size $\epsilon_0 = |b-a|$ and the required bracket size $\epsilon \leq \epsilon_0$. The main motivation to use the bisection method is that over the set of continuous functions, no other method can guarantee to produce an estimate c_n to the solution c that in the *worst case* has an ϵ absolute error with less than $n_{1/2}$ iterations.[9] This is also true under several common assumptions on function f and the behaviour of the function in the neighbourhood of the root.[9][10]

The bisection method for finding a root

Given a function $f: \mathbb{R} \to \mathbb{R}$, we search for a root of on a given interval $[a,b]$.

* A root of f is a number

$$x \in [a,b] \text{ such that } f(x) = 0$$

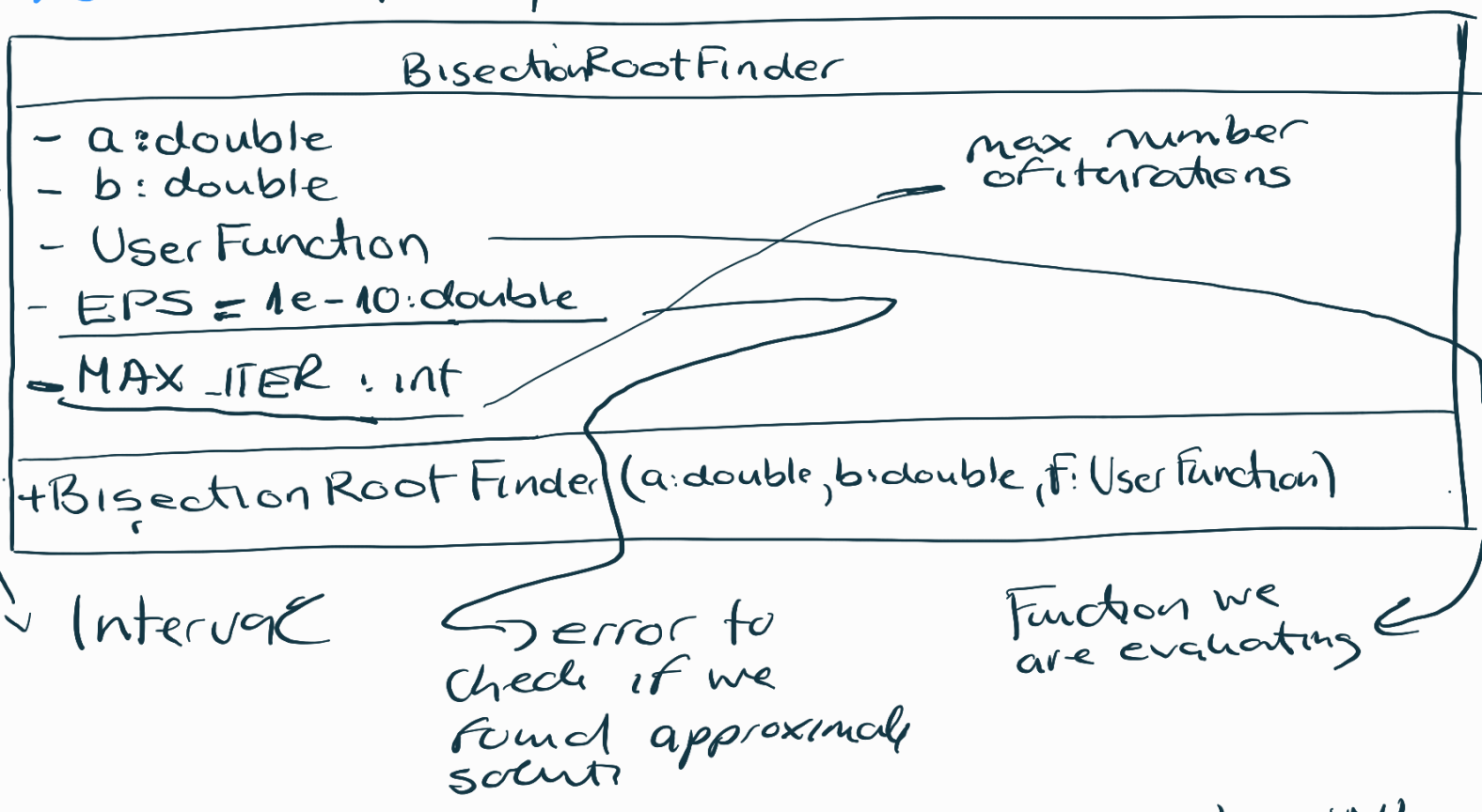
Using numerical methods (as opposed to analytical approaches), we can only search for numerical approximations of roots

$$x \in [a,b] \text{ such that } |f(x)| < \varepsilon$$

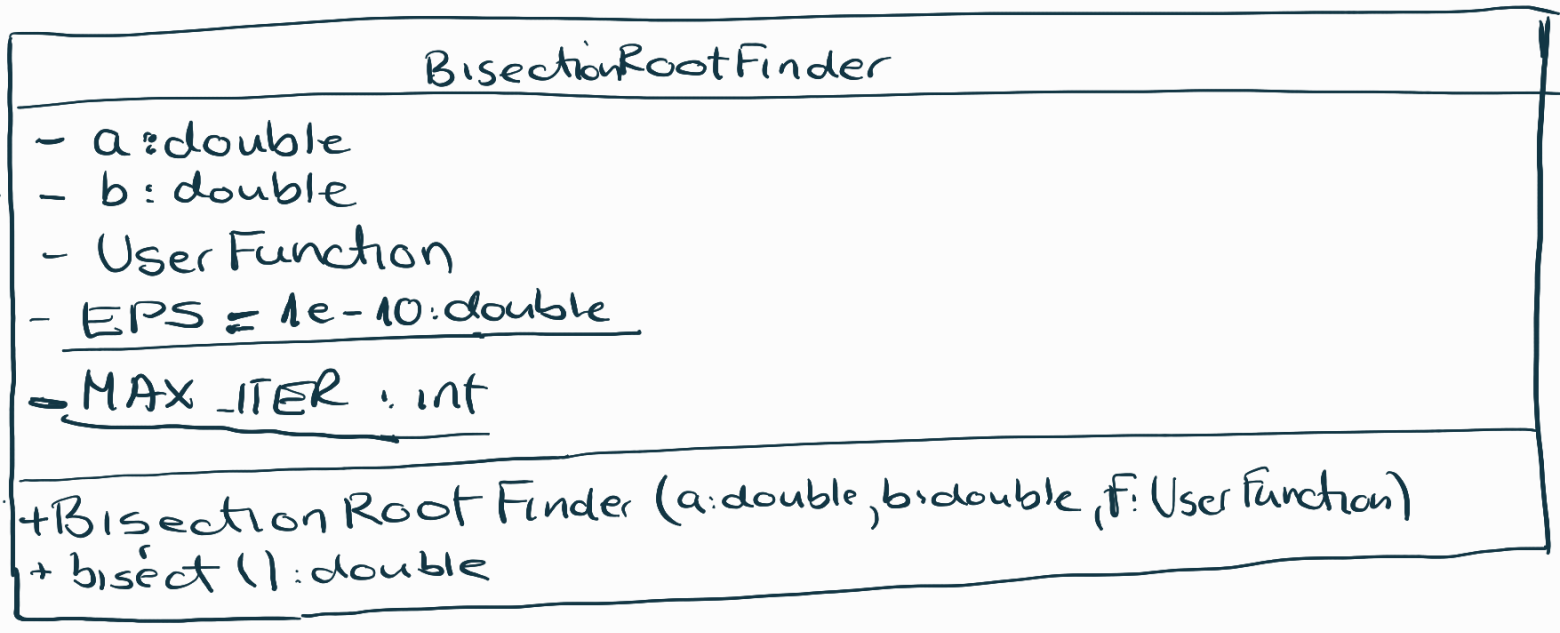
where $0 < \varepsilon << 1$

Inputs: User function we are evaluating, a, b - interval, number of iterations, ε - error

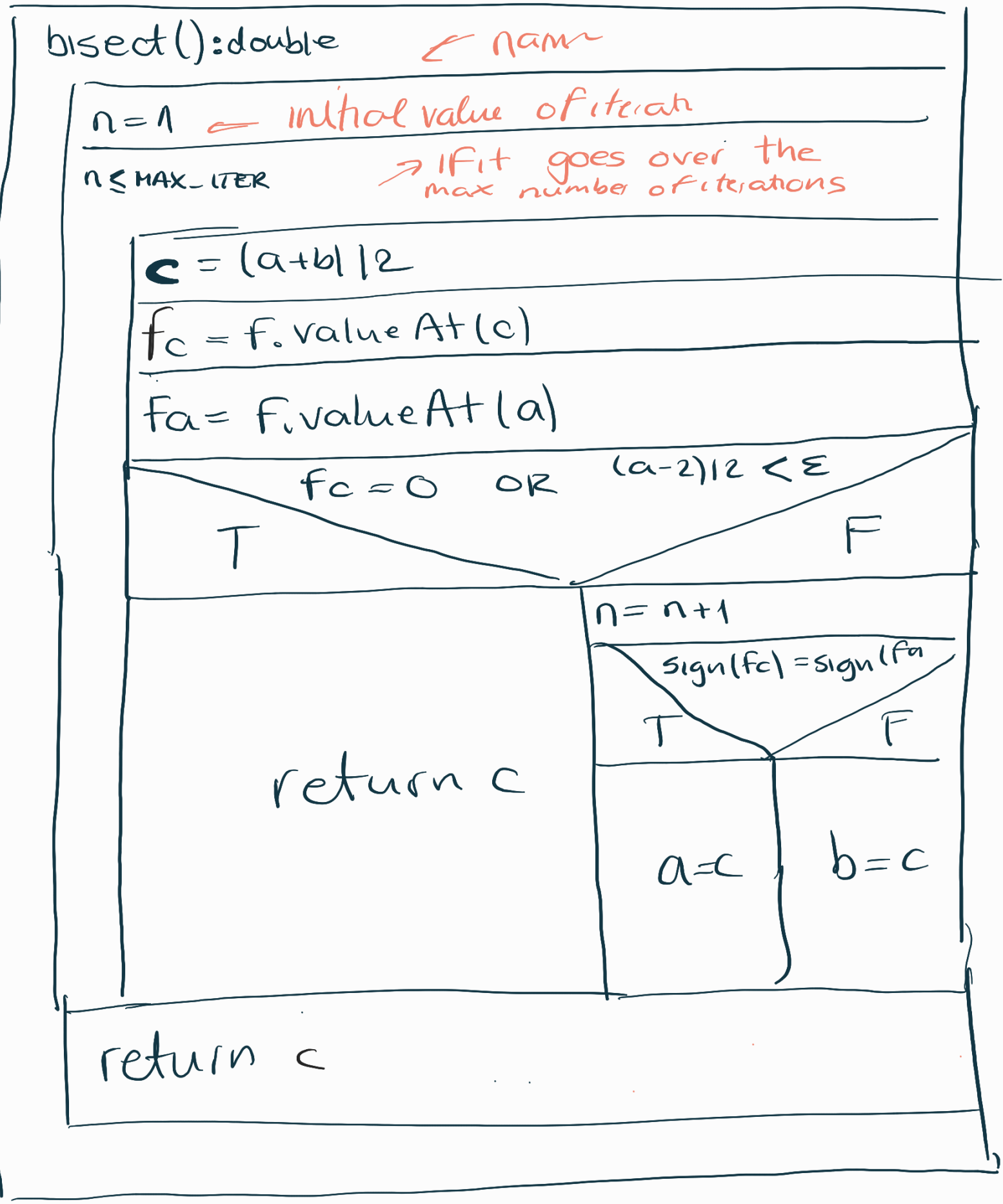
Step 1 - UML diagram with a constructor, attributes needed



Step 2 - add a bisect method to UML diagram, draw a Nassi-Shneiderman diagram for the method based on the pseudo-code



UML


bisect():double
name
n=1
initial value of iteration
n ≤ MAX_ITER
If it goes over the max number of iterations
c = (a+b)/2
fc = f.valueAt(c)
fa = f.valueAt(a)
fc = 0 OR (a-2)/2 < ε
T
F
n = n+1
sign(fc) = sign(fa
T
F
return c
a=c
b=c
return c

Step 3

```
public double bisection ( ) {
    int n=1;
    double a = this.a;
    double b = this.b;
    double c = (a + b)/2;;
    double fc;
    double fa;
    while (n<=10) {
    c = (a + b)/2;
    fc = f.valueAt(c);
    fa = f.valueAt(a);

    if (fc==0 || Math.abs((a-b))/2<EPS) {
        return c;
    } else {
        if (  Math.signum(fc) == Math.signum(fa)) {
            a=c;
        } else {    b=c; }
    }}
    return c;
```

Step 4

```
public static void main (String [] args) throws Exception {

    UserFunction f = new UserFunction("x*x-2", "x");
    double a=-2;
    double b=2;
    double result;

    BisectionRootFinder br = new BisectionRootFinder (a,b,f);
    result = br.bisection();
    System.out.print("The root is:" + result+ "\n");
    double k=Math.sqrt(2);
    System.out.print("The root with Math.sqrt function is:" + k + "\n");

    double difference = Math.abs(Math.abs(result) - Math.abs(k));
    double accuracy = EPS - difference;
    System.out.println("Difference between the value of functions at two calculated roots is :" + difference);
    System.out.println("The value of epsilon is :" + EPS);
    System.out.println("Difference between the epsilon values is :" + accuracy + "\n");
    double accuracyPercent = accuracy/EPS*100;
    System.out.println("Accuracy of the computed solution with respect to EPS is" + accuracyPercent + "%");
}
```

Step 5

BisectionRootFinder

- a : double
- b : double
- UserFunction
- EPS = 1e-10 : double
- MAX_ITER : int

+BisectionRootFinder (a:double, b:double, F: UserFunction)
+ bisect () : double
+ bisect2 () : int

```
public double bisection2 ( ) {
    int n=1;
    int n_final= (int) Math.ceil(Math.log((b-a)/EPS)/Math.log(2));
    double a = this.a;
    double b = this.b;
    double c = (a + b)/2;;
    double fc;
    double fa;
    while (n<=n_final) {
    c = (a + b)/2;
    fc = f.valueAt(c);
    fa = f.valueAt(a);

    if (fc==0 || (b-a)/2<EPS) {
        return c;
    } else {
        if (  Math.signum(fc) == Math.signum(fa)) {
            a=c;
        } else {    b=c; }
        n++;
    }}
    return 0;
}
```

bisect2 (): double ← name

n = 1 ← initial value of iteration

int n_final = $\log_2((b-a)/\varepsilon)$

n ≤ n_final

c = (a+b)/2

fc = f.valueAt(c)

fa = f.valueAt(a)

fc = 0 OR (a-2)/2 < ε

T

F

n = log2((b-a)/EPS)

return c

n = n+1

sign(fc) = sign(fa

T

F

a = c

b = c

return c

Step 7

```
        double c=0;
        double d = Math.PI;
        UserFunction f1 = new UserFunction ("tan(x/4)-1 ", "x");
        BisectionRootFinder br1 = new BisectionRootFinder (c,d,f1);
        double res3 = br1.bisection2();
        System.out.println("Calculated PI is " + res3);
        System.out.println("Value of PI by Math.PI is " + Math.PI);
    }

}
```

Problems | Javadoc | Declaration | Console | Call Hierarchy

```
<terminated> BisectionRootFinder [Java Application] C:\Users\User\Desktop\Programming\Eclipse\jre1.
Difference between the epsilon values is :5.2944973458102144E-11

Accuracy of the computed solution with respect to EPS is52.94497345810214%
Calculated PI is 3.1415926534983605
Value of PI by Math.PI is 3.14159265389793
```

Step 8

```
    double l=0;
    double m=3*Math.PI/2;
    UserFunction f3 = new UserFunction("E - 0.093*sin(E)-(270.0/180.0)*PI", "E");
    BisectionRootFinder br2 = new BisectionRootFinder (l,m,f3);
    double res4 = br2.bisection2();
    System.out.println("Calculated E is " + res4/Math.PI*180);
}
```