How to Write a Program Implementing the Main Interval Computation Algorithm: Advice (on the example of Java)

Warning. These are ideas, not exact code. However, if you notice some wrong syntax or a confusing part, please let me know.

Preliminary steps. We need to define a new type `Interval`, with two fields `lower` and `upper` and methods `add`, `subtract`, `multiply`, `inverse`, and `divide` corresponding to operations of interval arithmetic.

As usual, the description of this class should include construction methods. As usual in Java, it is a good idea to design two constructor methods:

- one when we form the element with two bounds, and
- one when we form the interval without knowing the bounds – e.g., an interval $[0, 0]$:

```
public Class Interval{
  public double lower;
  public double upper;

  public Interval(double a, double b){
    lower = a; upper = b;}

  public Interval(){
    lower = 0; upper = 0;}
```

Here is an example of how to implement operations of interval arithmetic – the method `add` should have the following form:

```
public static Interval add(Interval a, Interval b){
  return new Interval(a.lower + b.lower, a.upper + b.upper);}
```

The header for the new method. We want to design a program that, given n intervals x – i.e., an interval array – computes the enclosure for the range – i.e., an interval. So, the header of the following method should have the following form:

```
public static Interval intervalComp(interval[] x){
```

The program should be recursive. The program is recursive:

- it checks for monotonicity with respect to each of the variables x_i, and,
- if the given function is monotonic with respect to the i-th variable, we reduce the original problem to two new ones:
  - the one for which x_i is equal to the lower endpoint $\underline{x}_i$, and
  - the one for which x_i is equal to the upper endpoint $\overline{x}_i$.

How we form the two new problems. In the first of the new problems, we form a new array in which:

- all intervals $[\underline{x}_j, \overline{x}_j]$ with $j \neq i$ remain the same,
- but the i-th original interval is replaced with the *degenerate* interval $[\underline{x}_i, \underline{x}_i]$ (i.e., an interval containing only one value).

In other words, in the new array `x1`:

- all but one interval endpoints are the same as in the original array `x`,
- the only difference is that for the i-th interval, the upper bound is now equal to $\underline{x}_i$ (and not to $\overline{x}_i$ as before):

```
Interval [] x1 = new Interval[x.length];
for(j = 0; j < x.length; j++)
  {x1[j].lower = x[j].lower; x1[j].upper = x[j].upper;}
x1[i].upper = x[i].lower;
```

Similarly, we form the array `x2` corresponding to the second new problem.

Two important issues.

- It is important to avoid checking the monotonicity for the i-th variable when we deal with the new arrays – otherwise, we may get into the infinite loop. So, we need to make sure that we only check monotonicity with respect to non-degenerate intervals, i.e., intervals for which $\overline{x}_i > \underline{x}_i$.
- We also need to make sure that we compute the centered form if the function is not monotonic with respect to each variable.

Resulting (pseudo)code.

```
public static Interval intervalComp(interval[] x){
  boolean monotonic = false; int i = 0;
  while(!monotonic && i < x.length){
    if(x[i].upper > x[i].lower && partial(x,i).lower >= 0)
      {<form arrays x1 and x2>;
       return new Interval(intervalComp(x1).lower, intervalComp(x2).upper);}
    elseif(x[i].upper > x[i].lower && partial(x,i).upper <= 0)
      {<form arrays x1 and x2>;
       return new Interval(intervalComp(x2).lower, intervalComp(x1).upper);}
    else{i++;}}
  double[] tildex = new double[x.length];
  double[] delta = new double[x.length];
  for(int i = 0; i < x.length; i++){
    tildex[i] = (x[i].lower + x[i].upper)/2;
    delta[i] = (x[i].upper - x[i].lower)/2;}
  double tildey = f(tilde x);
  double resultLower = tildey;
  double resultUpper = tildey;
  for(int i = 0; i < x.length; i++)
    {result.lower += multiply(partial(x,i),new Interval(-delta[i], delta[i])).lower;
     result.upper += multiply(partial(x,i),new Interval(-delta[i], delta[i])).upper;}
  return new Interval(resultLower, resultUpper);
}
```

What about bisection. To make your life easier, adding bisection is not a part of the assignment. Feel free to add bisection, for extra credit.