Elizabeth Bruski
I pledge my honor that I have abided by the Stevens Honor System.

I started this project by coding the compute function in cpp, using variable names to help me understand where values needed to be and how they would be manipulated later, as I would code in assembly.

```cpp
double compute(double x, double eq[]){
    double x_x0 = x;
    double sum_x1 = eq[0]; // sum starts at first value
    cout << "first var\n" << eq[0] << endl;
    double thismult_x2 = 1;
    int counter_x3 = 0;
    int offset_x4 = 0;
    double coeff_x5 = eq[offset_x4];
    double xpower_x6 = 1;
    //cout << " coeff should be 5.3, it is " << coeff_x5 << endl;
    //offset_x4 = 1;
    if (counter_x3 == 0){
        cout << "no power \n" << endl;
        goto noPower;
    }
```

For example I labeled all the variables with fake register names at the end of their names so I would remember the correlations between values and what they represent. I also only used if and goto statements so I could keep the logic and flow between assembly and this as similar as possible. This also made debugging substantially easier than normal because I could debug issues that came up with my logic in cpp, with much easier-tracked variables and print statements.

```
compute: // finds the value of a function at a particular x, given x in x0
/*
d0: x, should be given
d1: sum, most important for return
d2: the value of x^whatever power * coeff
x3: counter
x4: offset for current coeff
d5: coeff value
d6: x^power of the counter
d7: coefficients
```

In my compute function in assembly you can see that I keep these registers the same.
D2 is the coefficient multiplied by x^counter, where the counter starts at 0 for the first coefficient and increments by one as I move through the equation, reflecting how many times x should be multiplied by itself. With the way my code is structured I simply multiply x by itself again each time I increment by a coefficient. As you can see below I reset and increment my variables accordingly before looping to find the sum of the next equation chunk. I made sure to use D registers where I have double types of data and the according commands, like FMUL and FADD, as well as MOVi for the direct values.

```
FMUL D6, D6, D0 // x^power * x (basically increases power)
LDR D2, one // reset D2 = 1.
ADD X3, X3, 1 // counter++
ADD X4, X4, 8 // offset + 8
BL nextnum // compute the next number
```

From my bisect function I determine whether the interval is within the tolerance by squaring them both and determining if the square of the interval is less than or equal to the square of the tolerance. If not then I determine the next interval through the below functions, which branch back to bisect again. If it is then I branch to print the strings and values.

```
midpointUpper: // reset b to be c
    FMOV D20, D23 // b = c
    FADD D23, D19, D20 // a + b in D23
    FDIV D23, D23, D22 // c
    B bisection // bisect again

midpointLower: // reset a to c
    FMOV D19, D23 // a = c
    FADD D23, D19, D20 // a + b in D23
    FDIV D23, D23, D22 // c
    B bisection // bisect again
```

The print function moves the proper values to x0, as well as the strings for printing.

```
goprint:  // print function
    ADR X0, funcVal // loads the address of the function call
    FMOV D0, D1 // stick the result in D0
    BL printf // print
    ADR X0, root // the root
    FMOV D0, D23 // move the root to D0
    BL printf // print
    MOV X0, 0 //ecit
    MOV X8, 93
    SVC 0
```