Data

```
exit:
        mov     x0, #0
        mov     w8, #93
        svc     #0



.data

a:
        .double -1.0
b:
        .double 1.0

coeff:
        .double 0.2, 3.1, -0.3, 1.9, 0.2

tol:
        .double 0.0000000001

N:
        .dword 4

print_root: .ascii "Root: %lf\nf(root): %lf\n\0"


.end
```

The data section is structured like it is in the project writeup on canvas. The tolerance is very low just so the root is more accurate.

```
.text
.global _start
.extern printf

_start:
        adr     x0, a
        ldr     d1, [x0]        //get address and load a


        adr     x0, b
        ldr     d2, [x0]        //get address and load b

        adr     x0, tol
        ldr     d3, [x0]        //get address and load tolerance

        adr     x0, N
        ldr     x1, [x0]        //NMax
        mov     x2, #1          //N

        fsub    d10, d2, d1     //b - a
        b       bisect


```

We start by loading up all the values we are going to need to do this a, b, tolerance and n. Nmax is never actually used, since thats a relic of the v1 implementation of the pseudo-code off of the wikipedia page linked here: https://en.wikipedia.org/wiki/Bisection_method. The actual method repeats infinitely until it finds a root or reaches the set tolerance. The d10 = b-a is also a remnant of an old version of the code that didnt work properly.

Here is the driver code for bisect:

```
bisect:
        // c = (a+b)/2
        fadd    d4, d1, d2      //a+b
        fmov    d5, #2.0
        fdiv    d4, d4, d5      //d4(c) = (a+b)/2



        // calculate f(c) and check if f(c) = 0

        fmov    d0, d4          //move d4 = c to d0 to be x
        bl      f_x             //calculate f(c)
        fcmp    d0, 0.0         //if f(c) = 0, its a root
        b.eq    print           //branch to print since we are done


        //Check if (b-a)/2 < tolerance

        fsub    d9, d2, d1
        fdiv    d9, d9, d5      //d9 now equals (b-a)/2
        fcmp    d9, d3          // check if (b-a)/2 < tolerance
        b.lt    print           //if so, we are done




        // N += 1 and check if sign(f(c)) == sign(f(a)): If so a = c, otherwise b = c

        add     x2, x2, 1       //increment n
        fmov    d10, d0         //move f(c) to d10
        fmov    d0, d1          //move a to d0
        bl      f_x             //calculate f(a)

        signcheck:                      //now to check if the signs are equal
                scvtf   d12, xzr
                fmul    d11, d0, d10
                fcmp    d11, d12
                b.lt    negativeSign
                fmov    d1, d4  //else positive, move c into a
                b       bisect

                negativeSign:
                        fmov    d2, d4 //if negative, move c into b
                        b       bisect
```

Most of the little details are in the comments. The bisection method begins by calculating c = (a+)/2 we then branch to the f(x) method (see below) to calculate f(c). If f(c) = 0, we have found a root and we are done and we can print our result. Otherwise, check if (b-a)/2 dipped below the tolerance, in which case we are also done, as we either found a root, or are reaching 0 in our calculation and we can print. This is dependent on the declared tolerance in the data section.

The incrementation of n is also a relic from the old implementation off of the wikipedia.

Moving onwards, we now have to check if the signs of f(c) and f(a) are equal. This is the signcheck branch on line 57. If the signs are equal, we set a = c. If the signs are different we set b = c to make a new interval.

F(x)

```
70 f_x:
71        scvtf   d8, xzr         //d8 will be result
72        adr     x0, N           //get address of degree
73        ldr     x5, [x0]        //move it into x5 to use as a loop counter (and also to calculate offset)
74        adr     x0, coeff       //get the address of coefficients (we will need it later)
75
76        f_x_Outer:
77                cmp     x5, #0          //check if we are done operating on the polynomial
78                b.lt    f_x_exit        //if so go to the exit
79                mov     x7, x5          //This will be a counter for the inner loop
80                lsl     x6, x5, 3       //shifting the counter x5 left by 3 so that we can use x6 as an offset to get our coefficient
81                ldr     d6, [x0, x6]    //x0 contains the address of our coefficients, put it into d6
82                fmov    d7, d0          //move d0 (the x for f(x)) into d7
83                b.eq    addlastCoeff    //If we already did that, go to the end since we finished our calculations in the inner loop
84
85                f_x_Inner:
86                        cmp     x7, 1           //check if our inner loop is done
87                        b.le    InnerExit       //if so, exit
88                        fmul    d7, d7, d0      //Power operation x*x
89                        sub     x7, x7, 1       //subtract from x8, which is our current power
90                        b       f_x_Inner       //go again to multiply until our power reaches 1
91
92                InnerExit:
93                        fmul    d7, d6, d7      //Multiply our x^n power by the current coefficient in d6
94                        fadd    d8, d8, d7      //add to the final result d8
95                        sub     x5, x5, 1       //subtract from the main loop counter
96                        b       f_x_Outer       //go to main loop and do it again until we reach the final coefficient
97
98        addlastCoeff:
99                fadd    d8, d8, d6      //add the coefficient without an x
100               b       f_x_exit        //go to exit
101
102       f_x_exit:
103               fmov    d0, d8  //return the result in d0
104               br      x30     //go back
```

This is the driver code for the F(x) calculation. d0 is our “input” register so to speak. We begin by storing 0 into d8, which will be our result tracker. We also load N (x5) to use as a counter and to calculate the offset when we need to. Naturally we also get the address of our coefficients. We start off by checking if our counter(which is really representing the current degree) has dipped below zero. If so, we are done. Otherwise, we set a new counter for the inner loop, shift x5 by 3 to get the offset. Once we load up our required coefficient, we move to the inner loop, which handles the power calculation. The new counter we set earlier keeps track of how many more times we have to multiply the x by itself to get our result. Once we are done with this, we branch to InnerExit, which handles the final multiplication of x^n power times the current coefficient. We repeat this exact process of going back and forth between the outer and inner loop until we reach 0 on the counter, meaning we have reached the front of the coefficient array. We branch to a separate label that will handle the addition of the final coefficient. We finish by returning d8 to d0.

Printing and exit

```
print:
        fmov    d0, d4          //move c into d0
        bl      f_x             //f(c)
        fmov    d1, d0          //move result to d1
        fmov    d0, d4          //move c to d0
        adr     x0, print_root
        bl      printf
        b       exit





exit:
        mov     x0, #0
        mov     w8, #93
        svc     #0

```

Once we have found our root c, which is in d4, we move it to d0, which serves as the input for our f(x) function. We get f(c) and move that to d1. We then restore c back into d4 and print in the format:
Root: c.data
F(root): f(c)

After this we are done, and we exit