Name: Harris Spahic
Date: 12/5/2021
Pledge: “I pledge my honor, I have abided by the Stevens Honor System.”

At the start of my function I define global variables I might need.
X1 = N or the degree of the polynomial
X2 = address of array of coefficients
D1 = A or the left x value of our interval
D2 = B or the right x value of our interval
D3 = allowed error bound (squared for later use to get rid of sign)
D14 = 2.0 since fmul requires constants to be in a register
D15 = 0.0 same as D14

bisection.s

```
.text
.global _start
.extern printf

_start:
    ldr x1, =N
    ldr x1, [x1]        // X1 --> Degree of polynomial
    ldr x2, =coeff      // X2 --> &coeff array

    adr x0, const1
    LDUR d0, [x0]
    fmov d1, d0         // D1 --> a, the left x val

    adr x0, const2
    LDUR d0, [x0]
    fmov d2, d0         // D2 --> b, the right x val

    adr x0, err
    LDUR d0, [x0]
    fmov d3, d0         // D3 --> err
    fmul d3, d3, d3

    adr x0, temp1
    LDUR d0, [x0]
    fmov d14, d0        // D14 --> 2.0

    fsub d15, d15, d15  // D15 -> obligitory 0 values
```

Then I branch to the main procedure **bisection**. Bisection uses a helper procedure called **“calc”**

```
    fsub d15, d15, d15  // D15 -> obligitory 0 values

    bl bisection
    bl print_result
```

Calc runs as follows. On input it assumes:
X1 = N
X2 = address of the array of coefficients
D4 = value of x to be calculated

And returns to **D0**.
It then puts all 3 of these variables + the link register onto the stack to prevent damage. We reset **D0** our return value to 0, and store the value of x into **D10** so we can keep increasing its power by multiplying **D4**.

Then we set **X9** to point to the end of the coeff-array to be checked as an exit condition in our main loop, and set a counter **X11** to 0 so that if we are at the first element in the array, we don't increase the power of x.

If **X11** = 0, we just add the constant coeff to **D0** otherwise, we multiply our coefficient by the current power of x held in **D10,** add the result to **D0** and increase the power of x by one. We then move to the next element in the array, and increase our counter by one. We run this loop until **X2** reaches the address after its last value **X9**.

We finally pop the stack, rewrite all our input (and branch) variables back to our previous values, and branch to our previous memory instructions.

```
// Assume X1 = N, X2 = &coeff, D4 = value to input into f(x) || Returns result at D0
calc:
    sub sp, sp, #32
    stur lr, [sp, #24]
    stur x1, [sp, #16]
    stur x2, [sp, #8]
    stur d4, [sp, #0]

    fmov d0, d15          // Reset result d0 -> 0
    fmov d10, d4          // Store power
    mov x9, x2            // Set x9 --> &coeff
    lsl x10, x1, #3
    add x9, x9, x10       // Set x9 --> &coeff + 8 * N -> &coeff[-1]
    mov x11, #0           // Set counter = 0

calc_loop:
    cmp x2, x9            // check if &coeff[i] < &coeff[0]
    b.gt calc_end         // if true, end loop
    ldr d9, [x2]          // else load coeff[i] into d9
    cmp x11, #0
    b.ne calc_cont
    fadd d0, d9, d0       // if adding constant (0th run), only add constant
    b calc_loop_end

calc_cont:
    fmul d9, d9, d10      // else coeff * val
    fadd d0, d9, d0
    fmul d10, d10, d4     // square value

calc_loop_end:
    add x11, x11, #1
    add x2, x2, #8
    b calc_loop

calc_end:
    ldur d4, [sp, #0]
    ldur x2, [sp, #8]
    ldur x1, [sp, #16]
    ldur lr, [sp, #24]
    add sp, sp, #32
    br lr
```

Now **bisection** takes in inputs
X1 = N
X2 = address of coefficient array
D1 = A = const1
D2 = B = const2
D3 = error * error (not just err thats a mistake)

Now once again we make a stack to save the link register, just in case. Since this is our main procedure, its not necessary to stur all our input variables. We won't care about their storage after **bisection** finishes running anyways.

We then check if N < 1, because if it is then our polynomial is just a constant and thus has no zero. (technically could have infinite zeros but not helpful) In which case we print an error message that will be shown later.

Then we calculate **D19** = **f(A), D20** = **f(B)**. And separately calculate **D5 = C**, our mid value and **D21 = f(C).**

Our next step is to check if **f(C)** is less than our error bound. Since we don't know the sign of **f(C)** and it's a pain to find, we square it and compare that to our squared error bound, since then both will always be positive. If **f(C)** is less than the bound, end the bisection call.

Else, multiply **f(A)** with **f(C)**, if the result is positive. We know they both have the *same sign*, and thus b must remain its old value & A becomes C. Otherwise, A remains the same & B becomes C.

We then loop back to the calculation part of bisection, and repeat until we narrow our **f(C)** to a value below our error bound.

Once we find such a value, we put our **C** in **D1** and our **f(C)** in **D2**, restore the original stack pointer and return to the **_start** branch whose line is after **bl bisection**. This takes us to **bl print_result** which will print our result.

```
// Assume x1 = N, X2 = &ceoff, D1 = A, D2 = B, D3 = err
bisection:
    sub sp, sp, #8
    stur lr, [sp, #0]
    cmp x1, #1
    b.lt print_error

loop_bisection:
    fmov d4, d1
    bl calc              // Calculate f(A)
    fmov d19, d0         // d19 = f(A)

    fmov d4, d2
    bl calc              // Calculate f(B)
    fmov d20, d0         // d20 = f(B)

    fadd d5, d2, d1      // D5 = B + A

    fdiv d5, d5, d14     // C = D5 / 2
    fmov d4, d5
    bl calc
    fmov d21, d0         // d21 = f(C)

    fmul d22, d21, d21   // d22 = f(c)^2
    fcmp d22, d3
    b.lt end_bisection

    fmul d23, d21, d19   // d23 = f(c) * f(a) Check if f(C) & f(B) have the same sign
    fcmp d23, d15        // If they don't then go to dif_signs
    b.lt dif_signs
    fmov d1, d5          // Else interval becomes [A, C] where A is pos & C is negative
    b loop_bisection

dif_signs:
    fmov d2, d5          //If signs are different --> C must be negative & B positve
    b loop_bisection
```


```
dif_signs:
    fmov d2, d5          //If signs are different --> C must be negative & B positve
    b loop_bisection
```


```
end_bisection:
    fmov d1, d5
    fmov d2, d21
    ldur lr, [sp, #0]
    add sp, sp, #8
    br lr
```

Print loads in our "solution" data string, and prints **C** & **f(C)** just stored in **D1 & D2**. Then it ends the print procedure with a branch to **print_end** and a system call to end our program.

We also have a **print_error** occasionally called in the beginning of **bisection** for when **N** < 1.

```
print_result:
    ldr x0, =solution
    bl printf

    b print_end

print_error:
    ldr x0, =error
    bl printf

print_end:
    mov x0, #0
    mov x8, #93
    svc #0
```

Finally we have our data path, which holds the variables, our interval & error bound, some temporary constants and the strings we'd like to outprint in **print_result**.

```
.data
    N:
        .dword 3
    coeff:
        .double 5.3, 0.0, 2.9, -3.1
    const1:
        .double 1.0
    const2:
        .double 2.0
    err:
        .double 0.01
    temp1:
        .double 2.0

    solution:
        .ascii "f(c) = %lf | Polynomial has solution at x = %lf \n\0"
    error:
        .ascii "Polynomial has no solution in this interval\n\0"

.end
```