ECE 150: Fundamentals of Programming

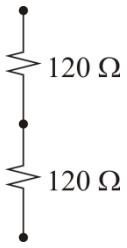
(Sections 001 and 002)

Project 1 Part B

Deadline: 11:59 PM on Friday October 26, 2018

Preamble to Problem 0

Engineers will often develop system that are *mission critical*, meaning that they must work without exception. One problem with the design of such systems is that they usually assume, for example, that all of the circuit elements have exactly the specified voltages, currents, resistances, capacitance, and impedance. For example, with the formula $V = IR$, you can create a circuit called a *voltage divider*, where the voltage at the mid-point is exactly half the voltage across the entire circuit:



Unfortunately, no resister is perfect, so it is necessary to determine what extremes of the voltage are at the mid-point. The voltage at the mid-point is $\frac{R_1}{R_1 + R_2}$ voltage across the circuit, and if the resistances are equal (as shown above), the voltage at the mid-point is half the voltage across. In reality, the best you can do is specify the range that the resistances: for example, the resistances could be anywhere between 108 Ω and 132 Ω, where the relative error is up to 10 %, or if you wanted to pay more, you could purchase components with a relative error of only 5 %. In all cases, however, there will be errors, but mission critical circuits must continue to work even if components are not ideal.

Therefore, it is necessary to determine the possible range of how much the voltage at the mid-point can vary relative to the voltage across. For this, we use interval arithmetic. For example, the resistance may be represented as the interval [108, 132] so the denominator is [108, 132] + [108, 132] = [216, 264], and if the numerator [108, 132] is divided by every possible value in this denominator, the ratio ranges from [0.4091, 0.6111]. The system will still work if the value voltage divider has a relative error of 22%, then it is reasonable to use resistors that have a 10 % relative error.

If, however, this is error is unacceptable, we could go to components that are more expensive, but with a 5 % relative error. In this case, the denominator is now [114, 126] + [114, 126] = [228, 252] and dividing [114, 126] by this interval gives all possible ratios on the range [0.4524, 0.5526].

We will now create a calculator that computes with intervals.

Interval arithmetic calculator

An interval is defined as a pair of values [*a*, *b*] representing all real numbers between these two end-points. Of course, *b* must be greater than or equal to *a*. When you launch the calculator, it will display the following prompt to the user and wait for the user to enter a command:

```
input :>
```

You will be able to enter commands to:

1. perform operations related to storage of intervals and memory,
2. perform interval calculations on the stored intervals,
3. exiting the program.

You will have a loop where you repeatedly request the user to enter a command until the user exits the program. Each command is defined by a string, and this is followed by zero or more operands; the number of operands depends on the command.

The program will have up to two intervals stored in memory. One is the *immediate* interval, the interval on which all operations are performed on, and the other is the *memory-stored* interval. This is similar to a simple calculator that has a value on which you are performing operations together with one memory location referred to by keys such as MC, MS, MR, M+ and M-. We will not have keys, but rather commands.

`exit`

Print the following string and exit the program.

```
Bye bye: Terminating interval calculator program.
```

For example:

```
input :> exit
Bye bye: Terminating interval calculator program.
```

`enter` *a b*

The two arguments are the end-points of an interval to be defined. If $a > b$, do nothing and issue the warning

`Error: invalid interval as` *a* `>` *b*

Otherwise, store this interval as the immediate interval.

In either case, print the immediate interval in the form `[`*a*`, `*b*`]`, where if no immediate interval has yet to be initialized, print `--`

For example:

```
input :> enter 3.2 1.5
Error: invalid interval as 3.2 > 1.5
--
input :> enter 3.2 5.1
[3.2, 5.1]
input :> enter 9.6 4.7
Error: invalid interval as 9.6 > 4.7
[3.2, 5.1]
input :> enter 4.7 9.6
[4.7, 9.6]
```

`negate`

This command takes no operands. If the immediate interval is not initialized, this command does nothing and `--` is printed. If, however, there is an immediate interval, this negates and switches the end-points. This is equivalent to negating each point on the interval $[a, b]$ yielding $[-b, -a]$.

For example:

```
input :> negate
--
input :> enter 3.2 5.1
[3.2, 5.1]
input :> negate
[-5.1, -3.2]
```

`invert`

This command takes no operands. If the immediate interval is not initialized, this command does nothing and `--` is printed. If, however, there is an immediate interval, if the interval contains 0, an error message is printed and the immediate interval is uninitalized, otherwise this inverts and switches the end-points. This is equivalent to inverting each point on the interval $[a, b]$ yielding $[1/b, 1/a]$.

For example:

```
input :> invert
--
input :> enter 2.5 5
[2.5, 5]
input :> invert
[0.2, 0.4]
input :> enter -1 1
[-1, 1]
input :> invert
Error: division by zero
--
```

`ms`

This command has no operands. If the immediate interval has been initialized, it is assigned to the memory-stored interval; otherwise, nothing is done. In either case, print the immediate interval in the form `[a, b]`, where if no immediate interval has yet been initialized, print `--`

For example:

```
input :> ms
--
input :> enter 3.2 5.1
[3.2, 5.1]
input :> ms
[3.2, 5.1]
```

At this point, both the immediate and memory-stored intervals are equal.

`mr`

This command has no operands. If no interval has been stored in memory, nothing is done; otherwise, the memory-stored interval is copied to the immediate interval. In either case, print the immediate interval in the form `[a, b]`, where if no immediate interval has yet been initialized, print --

For example:

```
input :> mr
--
input :> enter 3.2 5.1
[3.2, 5.1]
input :> mr
[3.2, 5.1]
input :> ms
[3.2, 5.1]
input :> enter 1.9 12.3
[1.9, 12.3]
input :> mr
[3.2, 5.1]
```

At this point, both the immediate and memory-stored intervals are equal.

`mc`

This command has no operands. It clears the interval from stored memory, so the interval in stored memory is now assumed to be uninitialized. Print the immediate interval in the form `[a, b]`, where if no immediate interval has yet been defined, print --

For example:

```
input :> mc
--
input :> enter 3.2 5.1
[3.2, 5.1]
input :> ms
[3.2, 5.1]
input :> enter 1.9 12.3
[1.9, 12.3]
input :> mr
[3.2, 5.1]
input :> mc
[3.2, 5.1]
input :> mr
[3.2, 5.1]
input :> ms
[3.2, 5.1]
input :> mr
[3.2, 5.1]
```

`m+`

This command takes no operands. If both the immediate and memory-stored intervals are initialized, the immediate interval is added to the one stored in memory. If the immediate interval is not initialized, this command does nothing and `--` is printed, otherwise it prints [a, b].

For example:

```
input :> enter 3.2 5.1
[3.2, 5.1]
input :> ms
[3.2, 5.1]
input :> m+
[3.2, 5.1]
input :> m+
[3.2, 5.1]
input :> mr
[9.6, 15.3]
```

`m-`

This command takes no operands. If both the immediate and memory-stored intervals are initialized, the immediate interval is subtracted (interval subtract) from the one stored in memory. If the immediate interval is not initialized or the memory-stored interval is not initialized, this command does nothing and `--` is printed, otherwise it prints [a, b].

For example:

```
input :> enter 3.2 5.1
[3.2, 5.1]
input :> ms
[3.2, 5.1]
input :> m-
[3.2, 5.1]
input :> m-
[3.2, 5.1]
input :> mr
[-7, -1.3]
```

We will now proceed to describe the arithmetic operations with scalars.

`scalar_add` *c*

This command takes one operand *c*. If the immediate interval is not initialized, this command does nothing and `--` is printed. If, however, there is an immediate interval, this value is added to both end-points. This is equivalent to adding *c* to each point in the interval $[a, b]$ yielding $[a + c, b + c]$.

For example:

```
input :> scalar_add 9.8
--
input :> enter 3.2 5.1
[3.2, 5.1]
input :> scalar_add 9.8
[13, 14.9]
```

`scalar_subtract` *c*

This command takes one operand *c*. If the immediate interval is not initialized, this command does nothing and `--` is printed. If, however, there is an immediate interval, this value is subtracted from both end-points. This is equivalent to subtracting *c* from each point in the interval $[a, b]$ yielding $[a - c, b - c]$.

For example:

```
input :> scalar_subtract 9.8
--
input :> enter 3.2 5.1
[3.2, 5.1]
input :> scalar_subtract 9.8
[-6.6 -4.7]
```

`scalar_multiply` *c*

This command takes one operand *c*. If the immediate interval is not initialized, this command does nothing and `--` is printed. If, however, there is an immediate interval, each end-point is multiplied by this scalar; however, this may reverse the order of the end-points. Multiplication by 0 is equivalent to creating the trivial interval [0, 0]. This is equivalent to multiply *c* by each point in the interval [*a*, *b*] yielding [*ac*, *bc*] if $c > 0$, [0, 0] if $c = 0$ and [*bc*, *ac*] if $c < 0$.

For example:

```
input :> scalar_multiply 2.0
--
input :> enter 3.2 5.1
[3.2, 5.1]
input :> scalar_multiply 2.0
[6.2, 10.2]
input :> scalar_multiply -1
[-10.2, -6.2]
```

`scalar_divide` *c*

This command takes one operand *c*. If the immediate interval is not initialized, this command does nothing and `--` is printed. If, however, there is an immediate interval, each end-point is divided by this scalar; however, this may reverse the order of the end-points. Division by 0 results in an error message and uninitializes the immediate interval.

This is equivalent to dividing each point in the interval [*a*, *b*] by *c* yielding [*a*/*c*, *b*/*c*] if $c > 0$, an error if $c = 0$ and [*b*/*c*, *a*/*c*] if $c < 0$.

For example:

```
input :> scalar_divide 2.0
--
input :> enter 3.2 5.1
[3.2, 5.1]
input :> scalar_divide 2.0
[1.6, 2.55]
input :> scalar_divide 0
Error: division by zero
--
input :> scalar_divide -1
[-2.55,-1.6]
```

`scalar_divided_by` *c*

This command takes one operand *c*. If the immediate interval is not initialized, this command does nothing and `--` is printed. If, however, there is an immediate interval, this scalar is divided by each end-point; however, this may reverse the order of the end-points. If the immediate interval contains 0, this results in an error message and uninitializes the immediate interval.

This is equivalent to dividing c by each point in the interval [*a*, *b*] yielding:

- [*c*/*b*, *c*/*a*] if *c* > 0,
- [*c*/*a*, *c*/*b*] if *c* < 0, and
- "Error: Division by zero." if $a \leq 0 \leq b$. This is if 0 is within the interval [a,b].

For example:

```
input :> scalar_divided_by 2.0
--
input :> enter 2 4
[2, 4]
input :> scalar_divided_by 5
[1.25, 2.5]
input :> scalar_divided_by -1
[-0.8, -0.4]
input :> scalar_divided_by 0
[-0,-0]
input :> enter -1 1
[-1, 1]
input :> scalar_divided_by 2.5
Error: division by zero.
```

We will now proceed to describe the arithmetic operations with intervals.

`interval_add` *`c d`*

This command takes two operands *c*, *d* which defines the interval [*c*, *d*]. If this is not a valid interval (*c* > *d*), an error message is printed and then the immediate interval is printed (either `--` or [*a*, *b*], as appropriate). If the immediate interval is not initialized, this command does nothing and `--` is printed. If, however, there is an immediate interval, this results in [*a* + *c*, *b* + *d*]. This is equivalent to adding each point in the immediate interval [a, b] to each point in the interval [*c*, *d*] yielding [*a* + *c*, *b* + *d*].

For example:

```
input :> interval_add 9.8 11.5
--
input :> interval_add 9.8 1.5
Error: invalid interval as 9.8 > 1.5
--
input :> enter 3.2 5.1
[3.2, 5.1]
input :> interval_add 4.7 6.8
[7.9, 11.9]
input :> interval_add 4.2 2.9
Error: invalid interval as 4.2 > 2.9
[7.9, 11.9]
```

`interval_subtract` *`c d`*

This command takes two operands *c*, *d* which defines the interval [*c*, *d*]. If this is not a valid interval (*c* > *d*), an error message is printed and then the immediate interval is printed (either `--` or [*a*, *b*], as appropriate). If the immediate interval is not initialized, this command does nothing and `--` is printed. If, however, there is an immediate interval, this results in [*a* – *d*, *b* – *c*]. This is equivalent to subtracting each point in the immediate interval [a, b] by each point in the interval [*c*, *d*] yielding [*a* – *d*, *b* – *c*].

For example:

```
input :> enter 3.2 5.1
[3.2, 5.1]
input :> interval_subtract 1.7 3.5
[-0.3 3.4]
```

`interval_multiply` *`c d`*

This command takes two operands *c*, *d* which defines the interval [*c*, *d*]. If this is not a valid interval (*c* > *d*), an error message is printed as above and then the immediate interval is printed (either `--` or [*`a, b`*], as appropriate). If the immediate interval is not initialized, this command does nothing and `--` is printed. If, however, there is an immediate interval, this results in a new interval that contains [min(*ac*, *ad*, *bc*, *bd*), max(*ac*, *ad*, *bc*, *bd*)]. This is equivalent to multiplying each point in the immediate interval [a, b] by each point in the interval [*c*, *d*].

For example:

```
input :> interval_multiply 2 1
Error: invalid interval as 2 > 1
--
input :> interval_multiply 1 2
--
input :> enter 2 4
[2, 4]
input :> interval_multiply 3 5
[6, 20]
input :> interval_multiply -1 1
[-20, 20]
input :> interval_multiply -3 -2
[-60, 60]
```

`interval_divide` *`c d`*

This command takes two operands c, d which defines the interval $[c, d]$. If this is not a valid interval ($c > d$), an error message is printed as above and then the immediate interval is printed (either `--` or `[a, b]`, as appropriate). If the immediate interval is not initialized, this command does nothing and `--` is printed. If the interval $[c, d]$ contains 0, a division-by-zero messages is printed and the immediate interval is uninitialized. Otherwise, the result is set to $[\min(a/c, a/d, b/c, b/d), \max(a/c, a/d, b/c, b/d)]$. This is equivalent to dividing each point in the immediate interval [a, b] by each point in the interval $[c, d]$.

For example:

```
input :> interval_divide 1 2
--
input :> enter 2 4
[2, 4]
input :> interval_divide 5 10
[0.2, 0.8]
input :> interval_divide -1 1
Error: division by zero
--
input :> enter -3 -2
[-3, -2]
input :> interval_divide -10 -5
[0.2, 0.6]
```

`intersect` *c d*

This command takes two operands *c* and *d*. If the interval [*c*, *d*] is not valid, then print an error message and then the immediate interval (either `--` or [*a*, *b*], as appropriate). If the immediate interval is not initialized, this command does nothing and `--` is printed. If, however, there is an immediate interval, if there is any intersection between the two, the immediate interval becomes that intersection. If the two do not intersect, uninitialize the immediate interval.

For example:

```
input :> intersect 2 1
Error: invalid interval as 2 > 1
--
input :> intersect 1, 2
--
input :> enter 3, 5
[3, 5]
input :> intersect 4 6
[4, 5]
input :> intersect 1 10
[4, 5]
input :> intersect 1 4.5
[4, 4.5]
input :> intersect 5 6
--
```

`integers`

This command prints all the integers in order that are in the immediate interval. It does not change the immediate interval.

For example:

```
input :> integers
--
input :> enter 2.99 4.99
[2.99, 4.99]
input :> integers
3, 4
[2.99, 4.99]
input :> scalar_multiply 5
[14.95, 24.95]
input :> integers
15, 16, 17, 18, 19, 20, 21, 22, 23, 24
[14.95, 24.95]
input :> scalar_divide 10
[1.495, 2.495]
input :> integers
2
```

`cartesian_integers` *`c d`*

This command takes two operands *c* and *d*. If the interval [*c*, *d*] is not valid, then print an error message and then the immediate interval (either `--` or [*a*, *b*], as appropriate). If the immediate interval is not initialized, this command prints `--` and finishes. Otherwise, this command prints all the integers in the Cartesian product [*a*, *b*] × [*c*, *d*] with the indices changing in the second interval first. It concludes by printing the immediate integer again.

For example:

```
input :> cartesian_integers 4.5 7.3
--
input :> enter 2.99 4.99
[2.99, 4.99]
input :> cartesian_integers 4.5 6.3
(3,5), (3,6), (4,5), (4,6)
[2.99, 4.99]
input :> cartesian_integers 6.1 6.3
[2.99, 4.99]
```

To implement this project, you could do everything in `int main()`, however, more reasonable would be to have a function called `interval_calculator()` so that your main function is quite trivial:

```
int main() {
    interval_calculator();
    return 0;
}
```

Your calculator function will have local variables for storing the immediate and memory-stored intervals.

Design question: How will you determine whether or not the immediate or memory-stored intervals are initialized?

You will then have an infinite loop which:

1. Will print to the console the prompt `"input :> "`
2. Will wait for the user to enter a variable of type `std::string`.
    a. If the command entered was `"exit"`, return from the function.
    b. Otherwise, based on the command entered, you will wait for the user to enter the requisite number of operands, all of which will always be of type `double`. If the command is not recognized, print the error message `"Error: illegal command"`.
    c. Once you have all operands, you will then perform the appropriate action on either the immediate or memory-stored intervals.
3. Return to Step 1.

Design question: There are many operations that are required to be performed over and over again. How will you minimize your work?

Marmoset Submission Instructions

Your code file name must be "`Intervals.cpp`".

Do **not** include the preprocessor directive "#ifndef MARMOSET_TESTING" for this submission.