ECE26400 Programming Exercise #3

In this exercise, you will implement two different ways to perform numerical integration of a function that is unknown to you. This exercise is related to PE04 and PA02.

The main learning goals are:

1. How to perform numerical integration of a function.
2. How to use `argc` and `argv` correctly in `main`.

1 Getting started

You should unzip on eceprog.ecn.purdue.edu the zip file `pe03_files.zip` using the following command:

```
unzip pe03_files.zip
```

The zip file `pe03_files.zip` contains a folder, named `PE03`, and five files within the folder:

1. `answer03.c`: This is the file that you have to modify. It has the descriptions of two numerical integration methods in it, and you must implement the numerical integration methods in these functions. **You will submit the modified file.**
2. `answer03.h`: This is a "header" file and it declares the functions you will be writing for this exercise.
3. `pe03.c`: You should use this file to write the main function that would call the appropriate numerical integration function. **You will also submit the modified file.**
4. `aux03.h`: This is an include file that declares the function to be integrated.
5. `aux03.o`: This is the object code for the function to be integrated.

To get started, read this document in its entirety. You will be writing code in the `answer03.c` file. You will also write code in the `pe03.c` file to call the correct functions implemented in `answer03.c`.

2 Numerical integration

We assume that you understand what it means to integrate a function over a bounded interval. Given a function $f(x)$, the integration of the function over a bounded interval defined by a lower limit a and a upper limit b is represented as

$$\int_a^b f(x)dx.$$

We use $[a,b]$ to represent the bounded interval defined by the lower limit a and the upper limit b. Note that the terms lower and upper refer to their relative positions next to the symbol representing the integration operation. It does not imply that the lower limit is no larger than the upper limit.

For example, if $f(x) = x$,

$$\int xdx = x^2/2 + C,$$

where C is a constant. We can integrate over a bounded interval defined by a lower limit 2 and an upper limit 10:

$$\int_2^{10} x dx = 10^2/2 - 2^2/2 = 48.$$

We can also integrate over a bounded interval defined by a lower limit 10 and an upper limit 2,

$$\int_{10}^{2} x dx = 2^2/2 - 10^2/2 = -48.$$

In the preceding example, we know the analytical form of the integral; therefore, we can calculate the integral over the bounded interval $[2,10]$ or the bounded interval $[10,2]$ precisely.

In reality, we may be dealing with a function that does not have an analytical form. For example, we do not have a function, but only samples $(x, f(x))$ obtained at different values of x. In the engineering world, we encounter that frequently when we use sensors to measure certain aspects of the environment. In that case, x may be the time and $f(x)$ is the sensed datum. Even when the function has an analytical form, the integral of the function (called integrand) may be too complicated or impossible to calculate. There are also cases where it is impossible to write down the integrand in analytical form.

In these cases, we can use numerical integration to approximate the integrand. There are many different numerical integration methods. We will focus on two methods in this exercise: the mid-point rule and the trapezoidal rule.

2.1 Mid-point rule

Consider the approximation of

$$\int_a^b f(x) dx.$$

The mid-point rule approximates the integration by using the area of a rectangle. Let $m = (a+b)/2$, we find $f(m)$ and use it as the height of the rectangle. The width of the rectangle is defined to be $(b-a)$. (Note that $(b-a)$ may be negative if $b < a$.) The integration is approximated as

$$\int_a^b f(x) dx \approx (b-a) \times f((a+b)/2) = (b-a) \times f(m).$$

Of course, this may not be accurate. The accuracy may be improved if we divide the bounded interval uniformly into many contiguous bounded intervals. Let n be the number of intervals. The bounded interval is divided into the following contiguous bounded intervals:

$$[a, a+(b-a)/n], [a+(b-a)/n, a+2(b-a)/n], ..., [a+(n-1)(b-a)/n, b].$$

We can re-write the integration as

$$\int_a^b f(x) dx = \int_a^{a+(b-a)/n} f(x) dx + \int_{a+(b-a)/n}^{a+2(b-a)/n} f(x) dx + ... + \int_{a+(n-1)(b-a)/n}^{b} f(x) dx.$$

Now, we can apply the mid-point rule to each of the intervals. The sum of all approximations of the intervals is an approximation to $\int_a^b f(x) dx$.

2.2 Trapezoidal rule

Consider the approximation of

$$\int_a^b f(x)dx.$$

The trapezoidal rule approximates the integration by using the area of a trapezoid. The heights of the two parallel sides of the trapezoid are $f(a)$ and $f(b)$. The width of the trapezoid is defined to be $(b-a)$. The integration is approximated as

$$\int_a^b f(x)dx \approx (b-a) \times (f(a)+f(b))/2.$$

Of course, this may not be accurate. The accuracy may be improved if we divide the bounded interval uniformly into many contiguous (bounded) intervals, and apply the trapezoidal rule to each of these intervals. The sum of all approximations of the intervals is an approximation to $\int_a^b f(x)dx$.

3 Functions you have to write

You have to write two functions in `answer03.c` and the `main` function in `pe03.c`.

3.1 Numerical integration using mid-point rule

The function implementing the numerical integration method based on the mid-point rule is declared in `answer03.h` as

```
double mid_point_numerical_integration(double lower_limit, double upper_limit,
                                       int n_intervals);
```

The parameters `lower_limit` and `upper_limit` correspond to the lower and upper limits of the bounded interval $[a,b]$ of

$$\int_a^b f(x)dx.$$

In other words, $a =$ `lower_limit` and $b =$ `upper_limit`.

The parameter `n_intervals` corresponds to the number of intervals we divide the bounded interval $[a,b]$. You may assume that `n_intervals` ≥ 1. The caller function has to pass in an `int` greater or equal to 1.

For this exercise, $f(x)$ is called `function_to_be_integrated(double x)`, which is declared in `aux03.h`, and defined in `aux03.c`. However, you are not provided `aux03.c`. Instead, you are given the object code `aux03.o`. You have to include `aux03.h` in your `answer03.c` file and call `function_to_be_integrated` in `mid_point_numerical_integration`.

You are required to implement in `answer03.c` the numerical integration method based on the mid-point rule, with the bounded interval [`lower_limit`,`upper_limit`] divided into `n_intervals` contiguous bounded intervals.

The sum of the approximations for all intervals should be returned (as a `double`).

3.2 Numerical integration using trapezoidal rule

The function implementing the numerical integration method based on the trapezoidal rule is declared in `answer03.h` as

```
double trapezoidal_numerical_integration(double lower_limit, double upper_limit,
                                         int n_intervals);
```

You are required to implement in `answer03.c` the numerical integration method based on the trapezoidal rule, with the bounded interval [`lower_limit`, `upper_limit`] being divided into `n_intervals` contiguous bounded intervals.

The sum of the approximations for all intervals should be returned (as a `double`).

3.3 `main` function

The executable of this exercise expects 4 arguments. If the executable is not supplied with exactly 4 arguments, the `main` function should return `EXIT_FAILURE`.

The first argument specifies which of the two integration functions you are supposed to run. If the first argument is `"-m"`, you should use the mid-point-rule-based method to perform the numerical integration. If the first argument is `"-t"`, you should use the trapezoidal-rule-based method to perform the numerical integration. If the first argument does not match `"-m"` or `"-t"`, the `main` function should return `EXIT_FAILURE`.

The second argument provides the lower limit (`double`) of the integral. You should use `strtod` (from stdlib.h) to convert the second argument into a `double`.

The third argument provides the upper limit (`double`) of the integral. You should use `strtod` to convert the third argument into a `double`.

If any of these two limits are of the wrong format or out of range (the function `strtod` is kind of similar to `strtol`), the `main` function should return `EXIT_FAILURE`. Moreover, if any of the two limits is `inf` (infinity) or `-inf` (negative infinity), the `main` function should return `EXIT_FAILURE`. You can use the function `isinf` from `math.h` (use the command "`man isinf`" to learn more about the function or related functions) to check whether a value is negative infinity or positive infinity. Furthermore, if any of the two limits is `nan` (not a number) or `-nan` (negative and not a number), the `main` function should return `EXIT_FAILURE`. You can use the function `isnan` from `math.h` (use the command "`man isnan`" to learn more about the function or related functions) to check whether a floating representation is "not a number."

The fourth argument provides the number of intervals (`int`) you should use for the approximation. **The argument should be provided in the base 10 format (see PA01).** You could use `strtol` (from `stdlib.h`) to convert the fourth argument into a `long` (and then store in an `int`). If the argument is of the wrong format or out of range, the executable should return `EXIT_FAILURE`. Moreover, if the conversion of the fourth argument results in a value that is less than 1 or greater than `INT_MAX`, you should return `EXIT_FAILURE`.

The successfully converted values from second, third, and fourth arguments should be supplied to the appropriate integration function.

Upon the successful completion of the numerical integration, print the approximation using the format `"%.10e\n"` to `stdout` using the function `fprintf`. This is the only output printed to `stdout` in the entire exercise. Any other messages printed to `stdout` will render the evaluation of your submission impossible. All other messages should be printed to `stderr`.

After printing the results of the numerical integration to `stdout`, return `EXIT_SUCCESS` from the main function.

You may have to include more `.h` files.

3.4 Printing, helper functions and macros

All debugging, error, or log statements in `answer03.c` and `pe03.c` should be printed to `stderr`.

You may define your own helper functions in `pe03.c` and `answer03.c`. All these functions should be declared and defined as static functions. In other words, the scope of these functions are only within the files that they are found. Declaring and defining these functions as static eliminate the conflicts in function names. You should not modify any of the `.h` files.

You also should not use macros that start with `T_` (Upper case T and underscore) in their names. We will use such macros in the evaluation of your submissions. If you use such macros, we may not be able to evaluate your submission properly.

4 Compiling your program

We use the same flags introduced in PE01 to compile the program for debugging purpose.

```
gcc -std=c99 -Wall -Wshadow -Wvla -Werror -g -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pe03.c answer03.c aux03.o -o pe03 -lm
```

Here, the linker option `-lm` specifies that we want to link with the math library. Although the function to be integrated in `aux03.o` does not require the math library, your testing of the numerical methods may involve math functions in the library (see Section 5).

When we evaluate your program, we also use the optimization flag `-O3` (uppercase letter `O` and number three) instead of the `-g` flag as follows:

```
gcc -std=c99 -Wall -Wshadow -Wvla -Werror -O3 -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pe03.c answer03.c aux03.o -o pe03 -lm
```

You are recommended to copy the `Makefile` from PE01 and modify it appropriately for PE03.

5 Running and testing your program

To run your program using the mid-point rule-based integration method, you can use for example,

```
./pe03 -m 0.0 10.0 5
```

As it is, i.e., if you do not make any changes to `answer03.c` and `pe03.c`, this would simply print to `stdout`

```
0.0000000000e+00
```

How do you know whether your implementation is correct when you have no idea what function you are integrating? Well, the integration methods are supposed to work for all functions, under the assumption that the function is well-defined within the bounded interval. (We will evaluate your implementation with a function that is well-defined within the bounded interval supplied to the `main` function.)

5.1 Testing with known functions

You can test your implementation on some known functions (and functions whose integrands you are familiar with).

You can write your own `function_to_be_integrated` in a different file. Let's call that file `my_aux03.c`. Now, you compile with the following command:

```
gcc -std=c99 -Wall -Wshadow -Wvla -Werror -g -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pe03.c answer03.c my_aux03.c -o pe03 -lm
```

Here, we assume that you are using some functions in `math.h`. Therefore, the `-lm` option is still being used so that we can link to the math library.

Your implementation of `function_to_be_integrated` could be any of the following simple functions such as:

$$f(x) = 1,$$

or

$$f(x) = x.$$

You can try to use piecewise linear function, such as:

$$f(x) = \begin{cases} 0 & \text{if } x < 1, \\ (x-1) & \text{if } x \geq 1. \end{cases}$$

You can try quadratic functions or functions with higher order. You can also try functions available in `math.h` (that is the reason we included the `-lm` option in the `gcc` command).

In fact, that is how we are going to evaluate your implementation, by using different implementations of `function_to_be_integrated`.

For each known function that you have implemented in `my_aux03.c`, you should try to perform integration with only 1 interval, and then 2 intervals, and perhaps some other numbers of intervals. You should choose a number of intervals that is easy for you to verify the correctness of your implementation. You may want to choose the number of intervals together with an appropriate pair of lower and upper limits.

For example, if you choose 3 as the number of intervals, it would be easier for you to work out the expected solution by hand if difference of upper limit and lower limit is divisible by 3.

Also, pick the lower and upper limits so that you can verify the results by hand easily. For example, if the number of intervals is 10, and if the method for integration is based on the mid-point rule, it may be better to use a lower limit of 0.5 and an upper limit of 10.5 because the mid-points for the 10 intervals would be integers, which might be easier for you to evaluate. On the other hand, if the method is based on the trapezoidal rule, it may be better to use a lower limit of 0 and an upper limit of 10 because the left and right end points of the intervals would be integers.

You can also use your implementation to check against your implementation. Let's assume that you have verified that your implementation is correct when you use only 1 bounded interval for integration. Let's pick a lower limit of 0 and an upper limit of 10, and you use 10 bounded intervals. You can run the case for integration with 1 bounded interval 10 times, each time with different bounded intervals: $[0,1]$, $[1,2]$, ..., $[9,10]$. The results you get from all 10 ranges should be summed and compared to the result when you run it on the case of 10 intervals with 0 and 10 being the lower and upper limits.

5.2 Roundoff Errors

In this exercise, you are dealing with floating point representation (in `double`) in this exercise. The representation of a real number using `double` (or `float` or `long double`) is not exact. (We have to use 64 bits in a `double` to represent real numbers, which are uncountable.) Therefore, there will be roundoff errors when we perform mathematical operations on floating point representation. The order in which your implementation performs arithmetic operations is likely different from the orders in your classmates' implementations and our implementation. Therefore, your roundoff errors will be different from the your classmates' roundoff errors and our roundoff errors.

You should not expect your `stdout` output to match others even with the same pair of limits and the same number of intervals. Similarly, if you use your implementation to test against your implementation, you should not expect the sum of the 10 results to match the result from a single run exactly. The reason is that when a double is printed through the format `"%.10e\n"`, there is some loss in accuracy in the printed `double` because the format prints only the first 10 significant numbers after the decimal point. Therefore, some reasonable amount of roundoff error is acceptable.

When a mathematical operation results in overflow or underflow, you may get `inf` or `-inf` printed to `stdout`, respectively. When a mathematical operation results in undefined value, you may have `nan` (not a number) or `-nan` printed to `stdout`. We will not use test cases that would result in such `stdout` output for a correct implementation when we evaluate your submission.

6 Running `./pe03` under `valgrind`

You should also run `./pe03` with arguments under `valgrind`. To do that, you may use, for example, the following command:

```
valgrind --tool=memcheck --leak-check=yes --log-file=memcheck.log ./pe03 -m 0.0 10.0 5
```

Although it is unlikely that you will have memory problems in this exercise, it is a good habit to cultivate. It is possible to run valgrind with the simple command below.

```
valgrind ./pe03 -m 0.0 10.0 5
```

Please see PE01 description about `valgrind`.

7 Submission

You must submit a zip file called `PE03.zip`, which contains two files:

1. `answer03.c`
2. `pe03.c`

Assuming that you are in the folder that contains `answer03.c` and `pe03.c`, use the following command to zip your files:

```
zip PE03.zip answer03.c pe03.c
```

Make sure that you name the zip file as `PE03.zip`. Moreover, the zip file should not contain any folders. Submit `PE03.zip` through Brightspace. You may submit as many versions as you like. Brightspace will keep only the most recent submission, and we will grade only the final submission.

8 Grading

All debugging messages should be printed to `stderr`. We expect only a single `double` returned from the appropriate numerical integration function be printed to `stdout`. It is important that if the instructor has a working version of `pe03.c`, it should be compilable with your `answer03.c` to produce an executable. For evaluation purpose, we will use different combinations of your submitted `.c` files and our `.c` files to generate different executables. If a particular combination does not allow an executable to be generated, you do not get any credit for the function(s) that the executable is supposed to evaluate. We use both `-g` and `-O3` flags (separately) to compile your submission. Your submission is evaluated only when compilation using each of the flags is successful.

Be aware that we set a time-limit for each test case based on the size of the test case. If your program does not complete its execution before the time limit for a test case, it is deemed to have failed the test case. We will not announce the time-limits that we will use. They will be very reasonable.

The `mid_point_numerical_integration` function and the `trapezoidal_numerical_integration` function account for 35% each. The `main` function accounts for 30%. In our evaluation of your implementation, some reasonable amount of roundoff error is acceptable. We will not announce the amount of roundoff error that we would tolerate.

The occurrence of any memory issues (memory errors or memory leaks flagged in a `valgrind` report) will result in 50-point penalty.

9 A few points to remember

All debugging messages should be printed to `stderr`. Other than the approximated integral, you should not be printing anything else to `stdout`. If the output of your program is not as expected, you get 0 for that test case.

You are not submitting `answer03.h` and `aux03.h`. Therefore, you should not make changes to `answer03.h` and `aux03.h`.

You can declare and define additional static functions that you have to use in `pe03.c` and `answer03.c`.

You should not use macros that start with `T_`.

Grading of programming exercises and assignments is performed on machines with similar setup as eceprog.ecn.purdue.edu. You should perform testing of your work on eceprog.ecn.purdue.edu before submission. Correct output on your computer does not translate into correct output on eceprog.ecn.purdue.edu.