ECE26400 Programming Exercise #4

In this exercise, you will implement yet another way to perform numerical integration of a function that is unknown to you. This exercise is related to PE03 and PA02.

The main learning goals are:

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

1 Getting started

You should unzip on eceprog.ecn.purdue.edu the zip file `pe04_files.zip` using the following command:

```
unzip pe04_files.zip
```

The zip file `pe04_files.zip` contains a folder, named `PE04`, and five files within the folder:

1. `answer04.c`: *This is the file that you have to modify and then submit.* You must implement the new numerical integration method in this file.
2. `answer04.h`: This is a "header" file and it declares the functions you will be writing for this exercise. You also have to define the structure required. ***You have to submit this file.***
3. `pe04.c`: You should use this file to write the main function that would call the appropriate numerical integration function with correct parameter(s). *You will also submit this file.*
4. `aux04.h`: This is an include file that declares a few unknown functions to be integrated depending on the arguments provided to the executable.
5. `aux04.o`: This file provides the object code for the unknown functions to be integrated.

To get started, read this document in its entirety. You will be writing code in the `answer04.c` file. You will also write code in the `pe04.c` file to call the correct functions in `answer04.c` file. You should also read both `answer04.c` and `pe04.c` to figure out the structure that you have to define in `answer04.h`.

2 Numerical integration

Please read the part about numerical integration in PE03. Now we focus on a new integration method: Simpson's rule.

2.1 Simpson's rule

Consider the approximation of

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

Let $m = (a+b)/2$ be the mid-point of the bounded interval. The Simpson's rule approximates the integration of $f(x)$ by considering a quadratic function that passes through three points: $(a, f(a))$, $(m, f(m))$, and $(b, f(b))$. The integration of the quadratic function over $[a,b]$ is the approximation of the original integrand. The following expression is the integration of the quadratic function over $[a,b]$:

$$\frac{(b-a)}{6}\left(f(a)+4f(m)+f(b)\right).$$

Therefore, the integration is approximated as

$$\int_a^b f(x)dx \approx \frac{(b-a)}{6}\left(f(a)+4f(m)+f(b)\right).$$

Again, 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

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

We can apply the Simpson's rule to each of the intervals. The sum of all approximations is an approximation to $\int_a^b f(x)dx$.

3 Functions/Structure you have to define

You have to define a structure in `answer04.h`, define a function in `answer04.c`, and define the main function in `pe04.c`.

In PE03, there is a limitation that the two integration functions are defined with a particular function to be integrated (called `function_to_be_integrated`). If you want to provide a more general integration function that could be used for any functions to be integrated, we have to define the integration function to accept any function that takes a `double` as an input parameter and returns a `double`.

We will try to fix that with this exercise.

3.1 Defining the structure

In PE03, we pass three parameters to the integration function: `double lower_limit`, `double upper_limit`, and `int n_intervals`.

Moreover, the `function_to_be_integrated` is "hard-wired" within the two integration functions. To relax that such that the integration function can be used for any function that accepts a `double` as an input parameter and returns a `double`, we must also pass the address of such a function to the integration function.

In PE04, we are required to pass a structure `integrand` to the integration function.

```
double simpson_numerical_integration(integrand intg_arg);
```

You task is to define such a structure in `answer04.h`. In this structure, you must store `double lower_limit`, `double upper_limit`, and `int n_intervals`. Moreover, you must store the address of the function to be integrated. The structure is partially defined. You have to complete the definition.

You may ask what is the type of the address of the function to be integrated. Consider the following:

```
int (*func)(double, int);
```

(Note that this is NOT the type that you should use in your structure.) Here, it says that `func` is the name of the location storing something. The `*` to the left of `func` says that `func` stores an address. It is important that `*func` is enclosed by a pair of parentheses. Without the parentheses around `*func`, it becomes a declaration of a function called `func` and `func` returns `int *`.

To the right of `(*func)`, the expression enclosed in the pair of parentheses indicates that the function whose address is stored in `func` expects two input variables, the first being a `double` and the second being an `int`. To the left of `(*func)`, `int` indicates that the function returns an `int`.

Essentially, the statement declares `func` to be a variable that would store an address pointing to a function that accepts a `double` and an `int`, and returns an `int`.

Here, **you want to define the** `integrand` **structure such that it contains a field called** `func_to_be_integrated` **to store the address of a function that accepts a** `double` **as an input parameter and returns a** `double`**. What should be the type of the field** `func_to_be_integrated` **in the structure?**

The answer to that would allow you to define in `answer04.h` your structure called `integrand`, which should contain the fields

- `lower_limit`
- `upper_limit`
- `n_intervals`
- `func_to_be_integrated` (whose type is to be determined by you)

Use these field names exactly and in the same order as listed in your structure. The field `lower_limit` is already defined in the structure. **Do not introduce additional fields.** Otherwise, we will not be able to evaluate your submission properly.

3.2 Numerical integration using Simpson's rule

The function implementing the numerical integration method based on the Simpson's rule is declared in `answer04.h` as

```
double simpson_numerical_integration(integrand intg_arg);
```

The parameter `intg_arg` passed into the function contains the following information: `intg_arg.lower_limit` and `intg_arg.upper_limit` correspond to the limits of the bounded interval $[a,b]$ of

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

with $f(x)$ being `intg_arg.func_to_be_integrated(x)`. Moreover, `intg_arg.n_intervals` corresponds to the number of intervals we divide the bounded interval $[a,b]$. You may assume that `intg_arg.n_intervals` ≥ 1 for this function. The caller function has to ensure that `intg_arg.n_intervals` ≥ 1 before calling the integration function.

You are required to implement in `answer04.c` the numerical integration method based on the Simpson's rule.

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

3.3 `main` function

We provide three unknown functions in `aux04.h` and `aux04.o`. In `aux04.h`, we declare three functions: `unknown_function_1`, `unknown_function_2`, and `unknown_function_3`. `aux04.o` contains the object code for these three functions.

If the executable is provided with correct arguments, it should perform numerical integration of one of these three functions.

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

The first argument to the executable specifies which of the three functions in `aux04.o` and `aux04.h` you are supposed to integrate. If the first argument to the executable is `"1"`, you should use the Simpson's rule based method to perform the numerical integration of `unknown_function_1`. If the first argument is `"2"`, you should use the Simpson's rule based method to perform the numerical integration of `unknown_function_2`. If the first argument is `"3"`, you should use the Simpson's rule based method to perform the numerical integration of `unknown_function_3`.

If the first argument does not match `"1"`, `"2"`, or `"3"`, the executable 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 executable should return `EXIT_FAILURE`. Moreover, if any of the two limits is `inf` (infinity) or `-inf` (negative infinity), the executable should return `EXIT_FAILURE`. You can use the function `isinf` or from `math.h` 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` 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` 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`.

At this point, you should be able to declare and initialize the fields of a variable `intg_arg` (of type `integrand`) properly, and pass that parameter to the 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 result in a lower score for your submission. 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 `answer04.c` and `pe04.c` should be printed to `stderr`.

You may define your own helper functions in `pe04.c` and `answer04.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.

Unlike any of the earlier exercises, you should modify and submit the `answer04.h`.

You 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 PE03 to compile the program for debugging purpose.

```
gcc -std=c99 -Wall -Wshadow -Wvla -Werror -g -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pe04.c answer04.c aux04.o -o pe04 -lm
```

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 pe04.c answer04.c aux04.o -o pe04 -lm
```

You are recommended to copy the `Makefile` from PE01 and modify it appropriately for PE04.

5 Running and testing your program

To run your program, supply four appropriate arguments to the executable. For example,

```
./pe04 1 0.0 10.0 5
```

As it is, i.e., if you do not make any changes to `answer04.h`, `answer04.c`, and `pe04.c`, this would simply print to `stdout`

```
0.0000000000e+00
```

See the write-up in PE03 on how you can test your program. Here, assume that you write your own `unknown_function_1`, `unknown_function_2`, and `unknown_function_3` in a file called `my_aux04.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 pe04.c answer04.c my_aux04.c -o pe04 -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.

6 Running `./pe04` under `valgrind`

You should also run `./pe04` 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 ./pe04 1 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 ./pe04 1 0.0 10.0 5
```

Please see PE01 description about `valgrind`.

7 Submission

You must submit a zip file called `PE04.zip`, which contains three files:

1. `answer04.h`
2. `answer04.c`
3. `pe04.c`

Assuming that you are in the folder that contains `answer04.h`, `answer04.c`, and `pe04.c`, use the following command to zip your files:

```
zip PE04.zip answer04.h answer04.c pe04.c
```

Make sure that you name the zip file as `PE04.zip`. Moreover, the zip file should not contain any folders. Submit `PE04.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 `pe04.c`, it should be compilable with your `answer04.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 `integrand` structure accounts for 30%, the `simpson_numerical_integration` function accounts for 40%, and 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 must modify `answer04.h` as specified and submit the modified file.

You are not submitting `aux04.h`. Therefore, you should not make changes to `aux04.h`.

You can declare and define additional static functions that you have to use in `pe04.c` and `answer04.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.

Appendix: Derivation of Simpson's rule

This appendix is not needed for the completion of the exercise. It is put here for those who want to understand the background of Simpson's rule.

Let $g(x)$ be a quadratic function that passes through three points $(a, f(a))$, $(m, f(m))$, and $(b, f(b))$, where $m = (a+b)/2$. Assume that $a \neq b$, $g(x)$ satisfies the following form:

$$g(x) = f(a)\frac{(x-m)(x-b)}{(a-m)(a-b)} + f(m)\frac{(x-a)(x-b)}{(m-a)(m-b)} + f(b)\frac{(x-a)(x-m)}{(b-a)(b-m)}$$

If you evaluate g(x) at a, m, and b, you will see that the expression evaluates to $f(a)$, $f(m)$, and $f(b)$, respectively.

Let $h = (b-a) \neq 0$, $m = a+h/2 = b-h/2$. Now, we consider a translation of the quadratic equation such that $G(x-m) = g(x)$. $G(x)$ must pass through the three points $(-h/2, f(a))$, $(0, f(m))$, $(h/2, f(b))$. $G(x)$ satisfies the following expression:

$$\begin{aligned} G(x) &= f(a)\frac{(x)(x-h/2)}{(-h/2)(-h)} + f(m)\frac{(x+h/2)(x-h/2)}{(h/2)(-h/2)} + f(b)\frac{(x+h/2)(x)}{(h)(h/2)} \\ &= \frac{2}{h^2}\left(f(a)x(x-h/2) - 2f(m)(x+h/2)(x-h/2) + f(b)(x+h/2)x\right) \end{aligned}$$

Let $G(x) = Ax^2 + Bx + C$, we can write down the expressions of A, B, and C as follows:

$$\begin{aligned} A &= \frac{2}{h^2}\left(f(a) - 2f(m) + f(b)\right) \\ B &= \frac{2}{h^2}\left(f(a)(-h/2) + f(b)(h/2)\right) = \frac{1}{h}(f(b) - f(a)) \\ C &= \frac{2}{h^2}\left(-2f(m)(-h/2)(h/2)\right) = f(m) \end{aligned}$$

$$
\begin{aligned}
\int_a^b g(x)dx &= \int_{a-m}^{b-m} G(x)dx \\
&= \int_{-h/2}^{h/2} G(x)dx \\
&= \frac{Ax^3}{3} + \frac{Bx^2}{2} + Cx \Big|_{-h/2}^{h/2} \\
&= \frac{Ah^3}{12} + Ch \\
&= \frac{h}{6}(f(a) - 2f(m) + f(b)) + f(m)h \\
&= \frac{h}{6}(f(a) + 4f(m) + f(b)) \\
&= \frac{(b-a)}{6}(f(a) + 4f(m) + f(b))
\end{aligned}
$$