Advanced C Programming

Fall 2017 :: ECE 264 :: Purdue University

This is for Fall 2017 (2 years ago) only.

Numerical integration #2 (E)

Due 9/13

In this exercise, you will implement yet another way to perform numerical integration of a function (unknown to you). This exercise is related to HW03 and HW05.

Learning goals

You should learn:

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

Getting Started

Start by getting the files. Type `264get hw04` and then `cd hw04` from bash.

You will get the following files:

1. **numint.c**: This has the descriptions of two numerical integration methods in it. You must implement the numerical integration methods in these functions. You have to hand in this file.
2. **numint.h**: This is a "header" file. It declares the functions you will be writing for this assignment. You also have to define the structure required. You have to hand in this file.
3. **test_numint.c**: You should use this file to write the main function that would properly initialize the structure that would be passed into the numerical integration function. You have to hand in this file.
4. **aux.h**: This is another “header” file. It declares a few unknown functions to be integrated depending on the arguments provided to the executable.
5. **aux.o**: This is the “object code” for the unknown functions to be integrated.

To get started, read this homework description in its entirety. Browse through the numint.h and numint.c files to see what code needs to be written. You will be writing code in the numint.c file. You will also write code in the test_numint.c file to call the functions in numint.c. Both numint.c and test_numint.c contain comments that tell you the code that needs to be written in numint.c and test_numint.c, respectively. You should also read both numint.c and test_numint.c to figure out the structure that needs to be defined in numint.h.

Follow the discussions below on how to compile and run your code, as well as test and submit it.

1. Numerical integration

Please read the part about numerical integration in HW03. Now we focus on a new integration method: Simpson's rule.

1a. Simpson's rule

Consider the approximation of $\int_a^b f(x)dx$
The Simpson's rule approximates the integration by using three points that pass through the function: $(a, f(a)), (\frac{a+b}{2}, f(\frac{a+b}{2})), (b, f(b))$.

A quadratic function is formed to pass through the same three points, and the integraton 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 $[a, b]$:

$$\frac{b-a}{6} * (f(a) + 4 * f(\frac{a+b}{2}) + f(b))$$

Therefore, the integration is approximated as

$$\int_a^b f(x)dx \approx \frac{b-a}{6} * (f(a) + 4 * f(\frac{a+b}{2}) + f(b))$$

Again, the accuracy may be improved if we divide the range into many intervals. We can apply the Simpson's rule to each of the intervals. The sum of all approximations is an approximation to $\int_a^b f(x)dx$.

2. Functions/Structure you have to define

You are to define a structure in numint.h, define a function in numint.c, and define the main function in test_numint.c.

In HW03, there was a limitation that the two integration functions are defined with a particular `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 accepts a double as an input parameter and returns a double as output.

We will try to fix that with this exercise.

2a. Defining the structure

In HW03, we passed three parameters to the integration function: `double lower_limit`, `double upper_limit`, and `int n_intervals`. Moreover, the `function_to_be_integrated` is fixed 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 as output, we must also pass the address of such a function to the integration function.

In HW04, we are allowed to pass only a structure to the integration function.

```
double simpson_numerical_integration(Integrand intg_arg);
```

You task is to define such a structure in numint.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. Let's understand the meaning of

```
    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 * in front of func says that func is going to store an address. It is important that the parentheses enclose "`*func`". Without the parentheses around "`*func`", it becomes a declaration of a function called func and func returns "`int *`".

The parentheses following `(*func)` indicate that func is going to store an address pointing to a function, and that function expects two input variables, the first being a `double` and the second being an `int`. Now, the int before `(*func)` indicates that the function returns an `int`.

Essentially, the statement declare `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.

You should ask yourself if you want to declare in the `Integrand` structure 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` as output, how you should declare the type of `func_to_be_integrated`.

The answer to that would allow you to define in numint.h your structure called `Integrand`, which should contain the fields `lower_limit`, `upper_limit`, `n_intervals`, and `func_to_be_integrated`. Use these names exactly. Otherwise, we may not be to evaluate your submission properly.

You need only **FOUR** fields in your structure, one of which is already defined for you. **DO NOT** include other fields in your structure.

2b. Simpson's rule based numerical integration

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

```
    double simpson_numerical_integration(Integrand intg_arg);
```

`intg_arg.lower_limit` and `intg_arg.upper_limit` correspond to the limits of the range $[a, b]$ of $\int_a^b f(x)dx$, where $f(x)$ is `intg_arg.func_to_be_integrated(x)`.

`intg_arg.n_intervals` corresponds to the number of intervals we divide the range $[a, b]$. You may assume that `intg_arg.n_intervals` ≥ 1 for this function. The caller function has to set `intg_arg.n_intervals` to be greater or equal to 1.

2c. The `main(…)` function

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

The first argument specifies which of the three functions you are supposed to be integrate. If the first argument 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 exit and return `EXIT_FAILURE`.

The second argument provides the lower limit (double) of the integral. You should use `atof(…)` (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 `atof(…)` to convert the third argument into a double.

The fourth argument provides the number of intervals you should use for the approximation. You should use `atoi(…)` (from stdlib.h) to convert the fourth argument into an int. If the conversion of the fourth argument results in an int that is less than 1, you should supply 1 (numeric one) as the number of intervals for approximation.

You should declare and initialize the fields of a variable `intg_arg` (of type `Integrand`), i.e., `intg_arg.lower_limit, intg_arg.upper_limit`, and `intg_arg.n_intervals`, and `intg_arg.func_to_be_integrated`, with the lower limit, the upper limit, the number of intervals, and the appropriate unknown function.

The variable `intg_arg` should be passed to the Simpson's rule based integration function.

Upon the successful completion of the numerical integration, print the approximation using the format "%.10e\n" using the function `printf(…)`. (That is the format to be used, and this is the only `printf(…)` statement in the entire exercise. If you print other messages, your exercises will most likely receive a lower score.)

After printing, the `main(…)` function returns `EXIT_SUCCESS`.

Requirements

1. Your submission must contain each of the following files, as specified:

file	contents

file	contents	
numint.c	functions	`double` **`simpson_numerical_integration`**`(Integrand intg_arg)` → *return type:* double • Given `intg_arg`, this function performs numerical integration of the function `intg_arg.func_to_be_integrated` over the range `intg_arg.lower_lilmit` to `intg_arg.upper_limit` • The range is divided into `intg_arg.n_intervals` uniform intervals, where the left-most interval has a left boundary of `intg_arg.lower_limit` and the right-most interval has a right boundary of `intg_arg.upper_limit` (assuming `intg_arg.lower_limit` ≤ `intg_arg.upper_limit`). If `intg_arg.lower_limit` ≥ `intg_arg.upper_limit`, the right-most interval has a right boundary of `intg_arg.lower_limit` and the left-most interval has a left boundary of `intg_arg.upper_limit`. • The Simpson's rule is used to perform the integration for each interval. In the Simpson's rule, three points are used to approximate the `intg_arg.func_to_be_integrated`. • The three points are: ◦ (left boundary, `intg_arg.func_to_be_integrated(left boundary)`) ◦ (right boundary, `intg_arg.func_to_be_integrated(right boundary)`) ◦ (mid-point, `intg_arg.function_to_be_integrated(mid-point)`) mid-point is the middle of the left and right boundary. A quadratic equation that passes through these three points is used to approximate the `intg_arg.func_to_be_integrated` The integration of the quadratic equation yields $\frac{width\ of\ interval}{6} * (f(left) + 4 * f(mid-point) + f(right))$ Here, f is short for `intg_arg.func_to_be_integrated` The width of the interval is (interval boundary closer to `intg_arg.upper_limit` - interval boundary closer to `intg_arg.lower_limit`). Note that width could be negative if `intg_arg.upper_limit` <`intg_arg.lower_limit` • The integral is the sum of the integration over all intervals. The caller function has to make sure that `intg_arg.n_intervals` ≥ 1 Therefore, this function may assume that `intg_arg.n_intervals` ≥ 1
numint.h	declarations	Please follow the instructions in this file.
test_numint.c	functions	**`main`**`(int argc, char ** argv)` → *return type:* int • Fill in the correct statements to complete the main function we expect four arguments: the first argument is 1 character from the sets {"1", "2", "3"}, it specifies the unknown function to be integrated ◦ `unknown_function_1(…)` ◦ `unknown_function_2(…)` ◦ `unknown_function_3(…)` otherwise: return `EXIT_FAILURE` • To integrate any of the three functions, expect the next three arguments to be the lower limit of the integration (double), the upper limit of the integration (double), and the number of intervals for the integration (int) if the number of steps is less than 1, set it to 1 • Use `atof(…)` to convert an argument to a `double`. • Use `atoi(…)` to convert an argument to an `int`. • You should declare a variable of type `Integrand`. You should use the four arguments to initialize the structure, and pass the structure to the simpson's method after printing the integral, return `EXIT_SUCCESS`.

2. Submissions must meet the code quality standards and the course policies on homework and academic integrity.

Compile

Compile your program with the following command:

```
gcc test_numint.c numint.c aux.o -o numint -lm
```

Note that -lm is required because the unknown functions contain function calls to math functions declared in math.h.

3a. Running your program

To numerically integrate unknown_function_1, you can use for example,

```
./numint 1 0.0 10.0 5
```

As it is, the executable would simply print to the screen
0.0000000000e+00

3b. Testing your program

See the write-up in HW03.
Here, assume that you write your own `unknown_function_1(…)`, `unknown_function_2(…)`, and `unknown_function_3(…)` in a file called my_aux.c . Now, you compile with the following command:

```
gcc test_numint.c numint.c my_aux.c -o numint -lm
```

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

Warning

Other than the approximated integral, you should not be printing anything else. If the output of your program is not as expected, you get 0 for that test case.

Bottom line: Do not use `printf(…)` to debug.

Although you can make changes to numint.h (since you are submitting the file), the only changes you are allowed to make is to define the type `Integrand` in numint.h. You should not make other changes to numint.h.

You can declare and define additional functions that you have to use in test_numint.c and numint.c.

In numint.c and test_numint.c, the first few lines of the files include the following statements:

```
// do not change this part, if you do, your code may not compile
  //
  /* test for structure defined by student */
  #ifndef NTEST_STRUCT
  #include "numint.h"
  #else
  #include "numint_key.h"
  #endif /* NTEST_STRUCT */
  //
  // do not change above this line, if you do, your code may not compile
```

`#ifndef` states that if the macro `NTEST_STRUCT` is not defined, the file numint.h is included. Otherwise, the file numint_key.h is included. (Note that numint_key.h is withheld from you. You do not have this file.) This is to facilitate grading so that if your numint.h does not work, I can still use numint_key.h (provided by the instructor) to compile and test your code. **Do not** make changes to these lines.

Submit

To submit HW04, type `264submit HW04 numint.h numint.c test_numint.c` from inside your hw04 directory.

In general, to submit any assignment for this course, you will use the following command:

```
264submit ASSIGNMENT FILES…
```

Submit often and early, even well before you are finished. Doing so creates a backup that you can retrieve in case of a problem (e.g., accidentally deleted your files).

To retrieve your most recent submission, type `264get --restore ASSIGNMENT` (e.g., `264get --restore hw04`).

To retrieve an earlier submission, first type `264get --list ASSIGNMENT` to view your past submissions and find the timestamp of the one you want to retrieve. Then, type `264get --restore -t TIMESTAMP ASSIGNMENT`.

Scores will be posted to the Scores page after the deadline for each assignment.

Pre-tester ●

The pre-tester for HW04 has been released and is ready to use.

Using the pretester

The pretester is a tool for checking your work after you believe you are done, and before we have scored it. It is not a substitute for your own checking, but it may help you avoid big surprises by letting you know if your checking was not adequate. To use the pre-tester, first submit your code. Then, type the following command. (*Do this only after you have submitted, and only after you believe your submission is perfect.*)

```
264test hw04
```

Do not ask TAs or instructors which tests you failed.

Keep in mind:

- Pre-testing is intended only for those who believe they are done and believe their submission is perfect.
- The pre-tester is not part of the requirements of this or any other assignment.
- You are responsible for reading the assignment carefully, and ensuring that your code meets all requirements.
- Feedback is limited, to ensure that everyone learns to test their own code.
- If your code is failing some tests, review *your tests* and make sure they are comprehensive enough to catch any bugs (deviations from requirements). Follow the tips given by the pre-tester.

- Code quality issues are not reported by the pre-tester; writing clean code is something you must learn to do from the start, not a clean-up step to do at the end.

Logistics:

- If we discover that we have not checked some significant part of the assignment requirements, we may add additional tests at any time up to the point when scores are released.
- The pre-tester will only be enabled after much of the class has submitted the assignment, and at least a few people have submitted perfect submissions. This is to allow us to test the pre-tester.
- The pre-tester checks your most recent submission. You must submit first.
- You may be limited to running the pre-tester ≤24 times in a 24-hour period. (This is not implemented yet but will be added.)

Q&A

Answers to common questions may be posted here later.