Advanced C Programming

Fall 2017 :: ECE 264 :: Purdue University

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

Numerical integration #1 (E)

Due 9/7

In this exercise, you will implement two different ways to perfrom numerical integration of a function (unknown to you).

This exercise is related to HW04 and HW05.

Learning goals

You should learn:

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

Getting started

To get the files, type `264get hw03` from bash. Then type `cd hw03`.

You will get the following files:

1. **numint.c**: this is the file that you hand in. It has the descriptions of two numerical integration methods in it, and you must implement the numerical integration methods in these functions.
2. **numint.h**: this is a "header" file and it declares the functions you will be writing for this exercise.
3. **test_numint.c**: You should use this file to write the main function that would call the appropriate numerical integration function.
4. **aux.h**: an include file to declare the function to be integrated
5. **aux.o**: provide the object code for the function 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 correct functions in numint.c. Both numint.c and test_numint.c contain comments telling you the code that needs to be written in numint.c and test_numint.c, respectively.

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

Numerical integration

We assume that you understand what it means to integrate a function over a range. Given a function $f(x)$, the integration of the function over a range $[a, b]$ is represented as $\int_a^b f(x)dx$

For example, $\int xdx = \frac{1}{2}x^2 + C$, where C is a constant.
With the formula given above, we can integrate over a range: $\int_2^{10} xdx = \frac{10*10}{2} - \frac{2*2}{2} = 48$
If the order of the limits of the range is reversed, $\int_{10}^2 xdx$ = $\frac{2*2}{2} - \frac{10*10}{2} = -48$

In the preceding examples, we know the analytical form of the integral, therefore, we can calculate the integral over the range $[2, 10]$ or $[10, 2]$ 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 x's. In the engineering world, we encounter that frequently when we use sensors to measure certain aspects of the environment. In that case, x is the time and $f(x)$ is the sensed datum.

The integral of the function (called integrand) may be too complicated or impossible to calculate. There are also cases that it is impossible to write down in analytical form the integrand.

Therefore, we have to use numerical integration to approximate the integrand. There are many different numerical integration methods. We will focus on two in this exercise: the **mid-point rule** and the **trapezoidal rule**. You will work on the Simpson's rule in HW04.

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 $m = \frac{a+b}{2}$, 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:

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

Of course, this may not be accurate. The accuracy may be improved if we divide the range into many intervals. Let n be the number of intervals. The step size is defined to be $\frac{b-a}{n}$. We can define

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

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

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

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

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

Requirements

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

file	contents	
numint.c	functions	**`mid_point_numerical_integration`**`(double lower_limit, double upper_limit, int n_intervals)` → *return type:* double • The parameters `lower_limit` and `upper_limit` correspond to the limits of the range $[a, b]$ 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 range $[a, b]$. You may assume that `n_intervals` ≥ 1 for this function. • The caller function has to pass in an int greater or equal to 1. • For this homework, $f(x)$ is called `function_to_be_integrated(double x)`, which is declared in aux.h, and defined in aux.c. However, you are **not** provided aux.c. Instead, you are given the object code aux.o. You have to include aux.h in your numint.c file and call `function_to_be_integrated(…)` in `mid_point_numerical_integration(…)`. • You are required to implement in numint.c the numerical integration method based on the mid-point rule, with the range [`lower_limit`, `upper_limit`] divided into `n_intervals` intervals. • The sum of the approximations for all intervals should be returned.
		`trapezoidal_numerical_integration``(double lower_limit, double upper_limit, int n_intervals)` → *return type:* double • You are required to implement the numerical integration method based on the trapezoidal rule, with the range [`lower_limit`, `upper_limit`] divided into `n_intervals` intervals.

file	contents	

test_numint.c	functions	**main**(int argc, char ✶✶ argv)
		→ *return type:* int
		• The executable of this homework expects 4 arguments. If the executable is not supplied with exactly 4 arguments, 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 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 (int) 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.
		• The converted values from second, third, and fourth arguements should be supplied to the appropriate 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, return EXIT_SUCCESS from the main function.

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

Compilation and testing

You should compile your program with the following command:

```
gcc test_numint.c numint.c aux.o -o hw03
```

Running your program

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

```
./hw03 -m 0.0 10.0 5
```

At it is, this would simply print to the screen `0.0000000000e+00`

Testing your program

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 on all functions. What you can do is to 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_aux.c. Now, you compile with the following command:

```
gcc test_numint.c numint.c my_aux.c -o hw03
```

Your implementation of `function_to_be_integrated(…)` should include simple functions such as:

$f(x) = 1, f(x) = x$

In these cases, the numerical integration should be exact because we are integrating a constant function or a linear function.

You can try to use piece-wise linear function, such as:

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

You can try quadratic functions or functions with higher order. You can also try functions available in math.h. However, in that case, you will have to compile with the -lm option:

```
gcc test_numint.c numint.c my_aux.c -o hw03 -lm
```

Note that this is in fact 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_aux.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 interval for integration. Let's pick a lower limit of 0 and an upper limit of 10, and you use 10 intervals. You can run the case for integration with 1 interval 10 times, each time with different ranges: 0 and 1, 1 and 2, 2 and 3, ..., 9 and 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 limits.

Be aware that you are dealing with floating point representation (in double) in this exercise. The order of arithmetic operations performed in your implementation is likely different from your classmates' implementations and my implementation. You should not expect your printed 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 on the screen because the format prints only the first 10 significant numbers after the decimal point.

4. Running ./hw03 Under Valgrind

You should also run ./hw03 with arguments under valgrind. To do that, you have to use for example the following command:

```
valgrind --log-file=memcheck.log ./hw03 -m 0.0 10.0 5
```

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

```
valgrind ./hw03 -m 0.0 10.0 5
```

You will get the log messages from valgrind on the screen.

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.

Bottomline: You should not use `printf(…)` to debug.

You are not submitting numint.h and aux.h. Therefore, you should not make changes to these .h files.

You can declare and define additional functions that you have to use in test_numint.c and numint.c. Make sure that these functions are declared before they are called in any other functions.

Summary

1. **Compile**

```
gcc test_numint.c numint.c aux.o -o hw03
```

2. **Run**

```
./hw03 -m 0.0 10.0 5
```

3. **Run under valgrind**

```
valgrind --log-file=memcheck.log ./hw03 -m 0.0 10.0 5
```

4. Don't forget to **LOOK** at the log-file "memcheck.log"

Pre-tester ●

The pre-tester for HW03 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 hw03
```

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.)

Submit

To submit HW03, type `264submit HW03 numint.c test_numint.c` from inside your hw03 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 hw03`).

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.

Updates

9/5/2017 Pre-tester instructions added 9/8/2017 Fixed typo in submission command