Technische Universität München

Informatics I for MSE

MSE IN8011
Prof. Bjoern Menze, Giles Tetteh
Philipp Kaeß

WS 19/20
Homework 5

Due Date: 08.12.19, 23:59

Release Date: 29.11.19

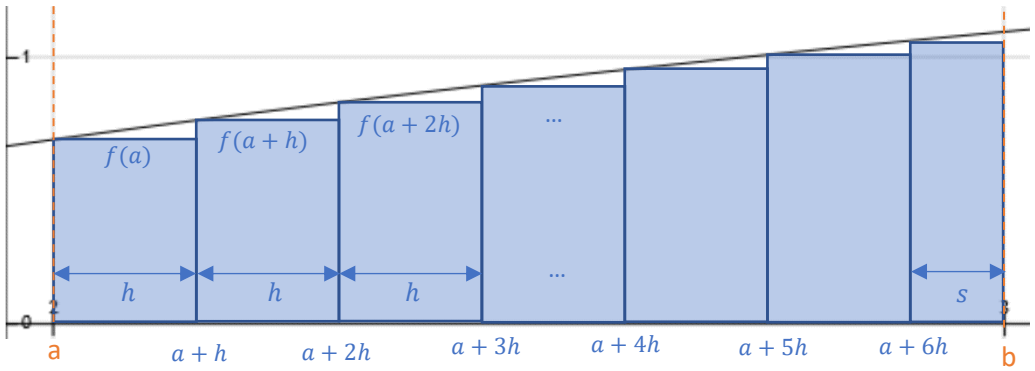
Homework 5: Numerical Integration

Topics: Numerical Methods, Multiple Files, Headers, Libraries, Functions as Parameters

Introduction
Integration is one of the main topics in maths in high school. Even though the techniques and formulas you learned enable you to integrate many functions, there are some cases where none of them will work. Probably the most famous one is the so-called Gauss Distribution Function (here without parameters for simplicity): $f_{gauss}(x) = e^{-x^2}$

You won't find a way to integrate this function, as any known technique fails here. This is because no analytical solution is existing, the indefinite integral ("Stammfunktion") can't be expressed by a mathematical function. In such cases, the only possibility to integrate is by application of numerical methods.

Left Riemann Method:
Let us suppose we want to integrate $f(x) = \ln(x)$ (plotted below) from $a = 2$ to $b = 3$. We know from school that the area under the graph equals the value of the desired integral. Therefore, we use an approximation of the area under the graph to estimate the integral. This approximation is called Riemann Sum.



Left Riemann Sum for $f(x) = \ln(x)$ from $a = 2$ to $b = 3$

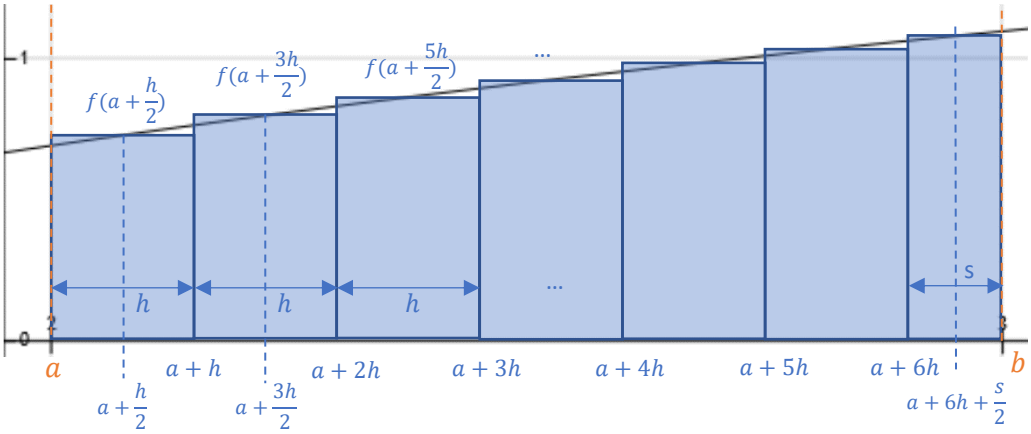
To estimate the area, we divide the x-Axis into equally small intervals, here each of length $h = 0.15$, where we assume constant behavior of the function we want to integrate (here $\ln(x)$). For the constant value (= height of the rectangle) we assume the real function value at the left boundary. This is why this method is called **Left Riemann Sum**. In general the width of the last slice is $s < h$, as the integration boundary b "cuts" in. In this case, we can see that $s = b - (a + 6h)$. For the final approximation, the single areas have to be added:

$$LRS = hf(a) + hf(a + h) + hf(a + 2h) + \cdots + hf(a + 5h) + sf(a + 6h).$$

The smaller we choose the element length h, the more accurate our approximation will be, and the more computations have to be done. But in general, we can approximate any integral of (continuous) functions as exactly as we like.

Midpoint Rule:

In the graphic of LRS we can see that the calculated value will in this specific case always be smaller than the real integral, as the function is continuously rising, and we are approximating with constant steps. Another, usually (but not always!) better method is the **Midpoint Rule** (also by Riemann). It works exactly the same, but the value of an interval (= height of the rectangle) is not the function value at the left boundary, but at the middle point of the interval.



Midpoint Rule for $f(x) = \ln(x)$ from $a = 2$ to $b = 3$

With the Midpoint Rule, our approximated integral in this case would be

$$MP = hf\left(a+\frac{h}{2}\right) + hf\left(a+\frac{3h}{2}\right) + hf\left(a+\frac{5h}{2}\right) + \cdots + hf\left(a+\frac{11h}{2}\right) + sf\left(a+6h+\frac{s}{2}\right).$$

Note: In this case, the numerical integration does not really make sense, as we could simply analytically integrate $\ln(x)$, and for our respective integration boundaries we would get the analytical solution $3ln(3) - 2ln(2) - 1$. Nevertheless, it is a good way to test the accuracy of our numerical approximation.

Numerical differentiation:

For the numerical approximation of the derivative, we introduce the **symmetric difference quotient**. To calculate the value of the derivative (= slope) of a function $f(x)$ at a specific point x_0, we assume linear behavior in a very small area around x_0, namely from $x = x_0 - h$ to $x = x_0 + h$. The slope of this linear part can then easily be calculated as known from school:

$$SDQ = \frac{f(x_0+h) - f(x_0-h)}{2h}$$

Once again, the smaller we choose h, the more accurate this approximation will be. For $lim_{h\to 0}$ we get the definition of the (symmetric) derivative itself.

Specifications

The function `differentiate(double (*f)(double x), double x, double h)` with the input parameters

- `f()` , the function that should be differentiated (see Tool Box)
- `x` , the point on the x-axis where `f()` should be differentiated
- `h` , the size of the infinitesimal step in both directions (as described above)

should return the above defined approximation to the derivative of `f` at `x` as a double.

The function `integrate_left(double (*f)(double x), double a, double b, double h)` with the input parameters

- `f()` , the function that should be integrated
- `a` , the left bound (start point) of the integration
- `b` , the right bound (end point) of the integration (you may assume $b \geq a$)
- `h` , the size of the infinitesimal intervals (as described above)

should return the above defined **Left Riemann** approximation to the integral over `f` from `a` to `b` as a double.

The function `integrate_midpoint(double (*f)(double x), double a, double b, double h)` (same input parameters as `integrate_left()`) should return the above defined **Midpoint Rule** approximation to the integral over `f` from `a` to `b` as a double.

Tasks

You are provided with the template file ***riemann.c*** on Moodle. Complete the functions `differentiate()`, `integrate_left()` *and* `integrate_midpoint()`, so that the program works as specified above. Please note:

- You are only allowed to write code in the spots indicated in the functions. Do not change anything else. Also do not include or remove any libraries or headers.

- Do not print anything

- You are provided with additional files:
  - ***funct_lib.c*** contains various mathematical functions. All functions receive a double value as `x` and return the respective `y` (also as double). These are the functions we will integrate / differentiate. Please don't change anything in this file!
  - ***funct_lib.h*** is the respective header file. When we need to use the mathematical functions in another C-File, we include this header.
  - ***main_riemann.c*** is a main-file that we provide you to test your functions. We will test the general accuracy of your functions, **and** if you really implemented the above defined methods (correct h, left vs. midpoint, correct handling of last (shorter) element etc.), so please stick **exactly** to the definitions above and do **not** e.g. make the steps smaller than the parameter value to improve accuracy artificially

- As the main file needs to use mathematical functions as well as your Riemann functions, the headers of both ***funct_lib.c*** and ***riemann.c*** are included at the beginning of the main file. We will not provide ***riemann.h***, you will have to create it on your own.

- When you are finished, create a library containing ***funct_lib.c*** and ***riemann.c*** called ***calculus.a***. This library has to compile and run together with the main file. (`gcc -Wall -Werror main_riemann.c calculus.a`)

- When you are finished, put ***calculus.a*, *riemann.c* and *riemann.h*** directly in a zip-File named "HW5_Mustermann_Max" (with your name of course) and upload this file to Moodle.

- We will provide test inputs so that the range of `double` will be sufficient. Therefore, use `double` instead of `float` for all your calculations to avoid accuracy-losses.

Example Output

After you compiled and executed the main file and your library (or your C-Files) together, your terminal should look like this:

```
Testing integrate_midpoint...
Testing integrate_left...
Testing differentiate...
Testing implementation...


TOTAL SCORE: 20 /  20
Please note that these testcases should only give you an orientation, and are NOT
representative for the correction test cases.
```

If a test case fails, it means that your approximation error is higher than the allowed threshold. If you correctly implement the above defined methods, your errors will be way smaller than the threshold.

Tool Box

The explanations below **may** help you to solve this problem.

FUNCTIONS AS PARAMETERS

The integration process works the same for all mathematical functions. Therefore, it makes sense to define the integration generally, and provide the specific mathematical function as a parameter of the integration function. An example for a function as parameter in another context is given below:

```
void print_number(int number){
    printf("%d\n",number);
}
void process_sum(void (*f)(int number), int a, int b){
    f(a+b); //Here f is a space holder for print_number
}
int main(){
    process_sum(print_number, 2,3);//prints '5'
}
```

For now, you don't need to understand "`void (*f)(int number)`", you only need to know that if you call `f()`, you actually call the function you get as parameter. So `f` is basically nothing but a space holder.

MULTIPLE FILES

Until now, if we needed to access a function that is defined in another C-File, we simply included this file. This bad coding style and won't be allowed now and in future assignments. We need to create a header for the desired file, and include this header instead. We then have to compile the two C-Files together, as the header only contains the declarations, and not the definitions.
Example: We want to use the mathematical functions from ***funct_lib.c*** as well as our self-written integration functions from ***riemann.c*** to test our functions in the main file. To do that, we have to include the headers of both files into the main file, and compile the three files together (list files in order of descending dependence):

```
gcc -Wall -Werror -o test main_riemann.c riemann.c funct_lib.c
```

This will create an executable that we can run. Alternatively, we can create a library out of the two desired files, and compile the main file together with this library. Therefore, we first create the object files ***funct_lib.o*** and ***riemann.o***:

```
gcc -c funct_lib.c riemann.c
```

These object files can then be converted into a library, here called ***calculus.a***:

```
ar -rcs calculus.a funct_lib.o riemann.o
```

When compiling the main file, we then simply put this library at the end of the command to create an executable:

```
gcc -Wall -Werror -o test main_riemann.c calculus.a
```

Threshold

To pass this homework, your program has to fulfill the following conditions:

- Your C-Files must be compilable and runnable together with the main file without any errors and with the flags -Wall -Werror enabled
  (`gcc -Wall -Werror -o test main_riemann.c riemann.c funct_lib.c`)

- Your files ***riemann.c*** and ***riemann.h*** have to be the files you used to create the library (do not change anything in these files after creating your final library).

- Your functions will be tested on a total of 35 cases (accuracy + implementation). To pass an accuracy case your approximation error must not be higher than the parameter h. If you correctly implement the above defined methods, this is fulfilled. To pass an implementation case, your implementation needs to be exactly as defined above (correct handling of boundaries etc.). To pass this assignment, you have to pass at least 80% (=28) of the test cases.