Week 2 — September 13, 2018

Homework

- Read chapters 3 and 4 in "Beginning C"
- Read chapters 1 and 2 in "Writing Scientific Software"

Exercises — Part I

1. True or false?
   - The associative property of multiplication holds for floating-point arithmetic, i.e.,
   $$a(bc) = (ab)c$$
   where a, b, and c are floating-point numbers.
   - The distributive property of multiplication holds for floating-point arithmetic, i.e.,
   $$a(b + c) = ab + ac$$
   where a, b, and c are floating-point numbers.
2. Do exercises 3-1 and 4-1 in "Beginning C".
3. Do exercises 2, 3, and 4 (p. 39, chapter 5) in "Writing Scientific Software".
   Remark: exercise 4 should read "If b^2 is **large** compared to ac . . . " (see errata).
4. Modify your code from exercise 2 in "Writing Scientific Software" so that it uses the "round toward zero" rounding mode. Is the output similar?
5. Write a program that prints a table with values `x op y` where
   - `x` and `y` are `-INFINITY`, `-1.0`, `-0.0`, `0.0`, `1.0`, `INFINITY`, or `NAN`
   - `op` is one of the arithmetic operators `*`, `/`, `+`, or `-`, or one of the relational operators `==`, `!=`, `>`, or `<`

y x	$-\infty$	-1.0	-0.0	0.0	1.0	∞	NAN
$-\infty$							
-1.0							
-0.0							
$\vdots$							
NAN							

6. Take this quiz to test your understanding of *if statements*.
7. Take this quiz to test your understanding of *loops*.

Exercises — Part II

Numerical integration

In this exercise, we will consider some basic methods for numerical integration. Specifically, given a function $f : \mathbb{R} \to \mathbb{R}$, we seek to compute or approximate definite integrals of the form

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

where a, b (the limits of integration) are given. Many so-called *rules* exist for approximating such integrals. Here we will focus on two simple rules, namely the *rectangle rule* and the *trapezoidal rule*. For more information on numerical integration, take a quick look at the Wikipedia page about Numerical Integration.

Rectangle rule

The rectangle rule (also known as the midpoint rule) approximates the definite integral by the area of a rectangle that is $b - a$ wide (i.e., the length of the interval $[a, b]$) and with height equal to the value of f at the midpoint $(a + b)/2$ of the interval $[a, b]$, i.e.,

$$\int_a^b f(x)\,dx \approx (b - a)\, f\left(\frac{a + b}{2}\right).$$

Notice that only a single function evaluation is necessary to compute the approximation.

Trapezoidal rule

The trapezoidal rule approximates the definite integral by the area of a trapezoid, i.e.,

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

Notice that the approximation requires two function evaluations.

Repeated/iterated rules

The approximation of the definite integral can be improved by dividing the interval into n subintervals. Specifically, if we define the width of each of these n subintervals as $h = (b-a)/n$, we can express the integral of interest as a sum of integrals over subintervals, i.e.,

$$\int_a^b f(x)\,dx = \sum_{i=1}^{n} \int_{a+(i-1)h}^{a+ih} f(x)\,dx.$$

Repeated rectangle rule

If we apply the rectangle rule to each of the subintervals in the above expression, we obtain the following approximation

$$\int_a^b f(x)\,dx \approx h \sum_{i=0}^{n-1} f(a + 0.5h + ih).$$

Note that this approximation requires n function evaluations.

Repeated trapezoidal rule

The repeated trapezoidal rule follows by applying the trapezoidal rule to each subinterval, i.e.,

$$\int_a^b f(x)\,dx \approx h\left(\frac{f(a)}{2} + \sum_{i=1}^{n-1} f(a+ih) + \frac{f(b)}{2}\right)$$

which requires $n+1$ function evaluations (and not $2n$ since we can reuse function values for adjacent subintervals).

Exercises

8. Write a C program that computes an approximation of the following definite integral

$$\int_a^b e^{-x^2}\,dx.$$

The program should prompt the user to enter the integration limits a and b, the number of subintervals n, and the method of choice (rectangle rule or trapezoidal rule).

You may use the following `main.c` template:

```c
#include <stdio.h>
#include <math.h>

int main(void) {

    /* Insert your code here */

    return 0;
}
```

Compiling your program

You can compile your program using the following command:

```
$ gcc -Wall main.c -lm -o numint1
```

The compiler flag `-Wall` enables warnings, and the flag `-lm` tells the linker to link against the math library (`libm`). Implementations of what is defined in `stdlib.h` and `stdio.h` are included in a system C library (`libc`) which is automatically linked against, so it is not necessary to explicitly include any linking options when including `stdlib.h` and `stdio.h` in your program.

9. Write a new program that (i) prompts the user to enter the integration limits a, b and a positive integer N, and (ii), prints a table with the approximations obtained with the two numerical integration methods for $n = 1, \ldots, N$. For example, the output could look like this:

```
Parameters:

    a = 0.0
    b = 1.0
    N = 50

Results:

    n  Rectangle       Trapezoidal
  -----------------------------------
    1  7.78800783e-01  6.83939721e-01
    2  7.54597944e-01  7.31370252e-01
    3  7.50252350e-01  7.39986475e-01
    :        :               :
   49  7.46836901e-01  7.46798596e-01
   50  7.46836396e-01  7.46799607e-01
```

Optional exercise

Monte Carlo integration is yet another method that can be used to approximate a definite integral of the form

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

Unlike the two methods described above, the Monte Carlo approach is based on randomization and is nondeterminstic.

To explain how Monte Carlo integration works, we'll need a random variable U with a uniform distribution on $[a, b]$, i.e., the probability density function is given by

$$P_U(x) = \begin{cases} \frac{1}{b-a} & a \leq x \leq b \\ 0 & \text{otherwise.} \end{cases}$$

The expectation of $f(U)$ is given by

$$\mathbb{E}[f(U)] = \int_{-\infty}^{\infty} f(x)P_U(x)\,dx = \frac{1}{b-a}\int_a^b f(x)\,dx$$

which is a constant multiple of the integral of interest. The expectation $\mathbb{E}[f(U)]$ may be estimated using a sample average approximation

$$\mathbb{E}[f(U)] \approx \frac{1}{N}\sum_{i=1}^{N} f(U_i)$$

where $U_1, \ldots, U_N$ denote N independent and identically distributed random samples from the uniform distribution on $[a, b]$. Thus, the definite integral of interest can be approximated as

$$\int_a^b f(x)\,dx \approx \frac{b-a}{N}\sum_{i=1}^{N} f(U_i).$$

Implement the Monte Carlo integration method for the function $f(x) = e^{-x^2}$ in C and compare it to your implementation of the two deterministic methods. Investigate numerically how the accuracy depends on the number of samples.

Hints:

- The approximation can be implemented recursively. To see this, let $s_1 = f(U_1)$ and define
$$s_i = s_{i-1} + f(U_i), \quad i = 2, \dots, N,$$
i.e., $s_N = f(U_1) + \cdots + f(U_N)$. Dividing both sides by i yields the equation
$$\begin{aligned}\frac{s_i}{i} &= \frac{s_{i-1}}{i} + \frac{f(U_i)}{i} \\ &= \left(1 - \frac{1}{i}\right)\frac{s_{i-1}}{i-1} + \frac{1}{i}f(U_i),\end{aligned}$$
and hence the approximation after i samples can be expressed recursively as
$$\int_a^b f(x) \approx (b-a)v_i = (b-a)\left[\left(1 - \frac{1}{i}\right)v_{i-1} + \frac{1}{i}f(U_i)\right]$$
where $v_i = \frac{s_i}{i}$, or equivalently,
$$v_i = \left(1 - \frac{1}{i}\right)v_{i-1} + \frac{1}{i}f(U_i)$$
for $i \geq 1$.

- The function `rand()`, which is defined in `stdlib.h`, can be used to generate pseudo-random integers between 0 and `RAND_MAX`, which is a constant that is defined in `stdlib.h`. The random number generator must be initialized with a so-called *seed* in order to produce a new pseudo-random series of numbers each time you run your program. The seed is set using the function `srand()` which takes an `unsigned int` as input. It is common to use the current time as a seed, i.e., to generate two pseudo-random numbers from a uniform distribution on $[a, b]$, we may use the following code:

```c
srand(time(NULL));    // seed random number generator
double rv1 = a + (b-a)*rand()/RAND_MAX;
double rv2 = a + (b-a)*rand()/RAND_MAX;
```

It is only necessary to initialize the random number generator once before subsequent calls to `rand()`. The `time()` function is defined in the `time.h` header file.