UNIVERSITÉ DE MONTPELLIER

FACULTÉ DES SCIENCES MONTPELLIER

PROGRAMMATION 1 - HAX816X - 2021/2022
-
EVALUATION FINALE

les fichiers à rendre à l'issue de cette évaluation sont indiqués en rouge. merci de respecter la dénomination demandée. Merci de ne déposer sur l'ENT qu'un seul fichier (dossier compressé) contenant l'ensemble de vos unités de compilation.

Exercice 1. Numerical methods for Ordinary Differential Equations (ODEs)

Many physical problems are modeled by differential equations. In this evaluation, we focus on some basic numerical methods for solving ODEs. A first-order ordinary differential equation with a given initial value is usually written in the form:

$$x'(t) = f(t, x(t)), \quad t_0 \leq t \leq T,$$
$$x(t_0) = x_0,$$

where $x(t)$ is the unknown function depending on t (often representing time) and $f(t, x)$ is a given (source) function depending on t and x. The exact solution $x(t)$ can not be found in general.
Instead, approximations to the solution $x(t)$ are sought at certain points $t_0 < t_1 < \ldots < t_N = T$ for a positive integer N. Let the approximate solution at $t = t_k$ be denoted by x_k. The objective of a numeric method is to find values $x_1, x_2, \ldots, x_n$, which approximate the exact solution values $x(t_1), x(t_2), \ldots, x(t_N)$, respectively. Denoting $\delta_t = T/N$, one simple such method is called explicit Euler's method:

$$x_{k+1} = x_k + \delta_t f(t_k, x_k).$$

Since x_0 is given, x_1 can be obtained directly from the formula with $k = 0$. Then x_2 can be solved by letting $k = 1$ and using x_1. This procedure can go on until x_N is obtained.

In the following, we want to create a small class dedicated to the approximation of first-order ODEs solutions based on such ideas.

(1) in files named ode.hpp and ode.cpp, declare and define a class named ode, with the following **private** members:

- t_ini for the initial time (generally equal to zero, but it may varies as well ...),
- x_0 for the initial value,
- t_end for the end time,
- **double** (*sfn) (**double** t, **double** x); for the source function f,

and the following **public** members:

- ode(**double** tini, **double** xini, **double** tend, **double** (*f) (**double**, **double**)) as a constructor,
- **double*** euler(**int** n) **const**;

for the implementation of the Euler method with N points, as described above. The euler method returns a one-dimensional array of **double**s, storing the N values x_k, hence the **double*** return type.

(2) test your class with the provided main_ode.cpp program, using the following source function:
`double f(double t, double x) {return x*(1 - exp(t)/(1 + exp(t);}`

(3) the exact solution of the ODE corresponding to the previous source function is

$$x(t) = \frac{12\exp(t)}{(1+\exp(t))^2}.$$

In the main file, add the computation of the error in ∞-norm between the numerical approximation and the exact solution. As a consequence, the main_ode.cpp file should be placed in your zip files ...

Exercice 2. Matrices and Purely templated approach

We aim at defining a function sum() that adds all entries of a matrix. We'd like this function to operate on various matrix types, let say for full matrices that stores all entries of a matrix and symmetric matrices that stores only the lower triangular part of a matrix to save memory.

The overall goal is thus to give only one version of sum() that can be called for full and symmetric matrices, and *without using virtual functions*.

In this Exercice, **we do not use inheritance at all, but only use templates**. A class definition (full .hpp, full .cpp) is provided in /CC_FINAL.

(1) given the definition of the class FullMtx, define a global (free) template function sum() whose template parameter can be any type that overload the **operator**() with two parameters to return the (i, j) entry of the matrix. This function should sum all the entries of its template parameters (which is then assumed here to be of a matrix type). This function should be provided in a separate sum.hpp file.

(2) declare and define (in new files symmetric.hpp and symmetric.cpp) a new class SymmetricMtx that only stores the lower triangular part of symmetric square matrices to save memory. As for the class FullMtx, such a class should provide a constructor that allocates and initializes to 0 all $\frac{n(n+1)}{2}$ entries, and overload the **operator**() to return the (i, j) entry, so that this class can be passed as a template parameters to sum().

(3) using your sum() function and your SymmetricMtx class, the provided main_sum.cpp program should compile, run and give the expected result. Provide the compilation instructions in a separate file compile.txt or a Makefile.

Fin du sujet.