C1 - Assignment 4 Report: Parabolic Initial-Boundary Value Problem

Student Number: 1894945

December 27, 2018

Contents

1 Introduction

Intuitively, (Parabolic) Initial-Boundary Value Problems (IVBPs) combine an initial value problem in a distinguished "time-like" direction with a boundary value problem in "space-like" directions. In this assignment, the approximation of solutions to IVBPs for scalar parabolic differential equations (PDEs) will be considered.

Definition 1. Let $I \subset \mathbb{R}$ be an interval, $\Omega \subset \mathbb{R}$ be a simply connected, bounded domain $t_0 \in I$, $\mathcal{L}$ a spatial differential operator of the form $\mathcal{L} = -\Delta + \mathbf{p} \cdot \nabla + q$, for $\mathbf{p}$ and q smooth functions. The parabolic initial boundary-value problem (IBVP) is to find $u(\mathbf{x}, t) : \Omega \times I \rightarrow \mathbb{R}$ for which:

$$\begin{cases} \partial_t u + \mathcal{L}u = f & \text{for } \mathbf{x} \in \Omega, \ \ \forall t \in \ I \\ u = v & \text{for } \mathbf{x} \in \partial\Omega, \ \ \forall t \in I \\ u(\mathbf{x}, 0) = u_0(\mathbf{x}) & \text{for } \mathbf{x} \in \Omega \end{cases} \tag{1}$$

with data $f(\mathbf{x}, t) : \Omega \times I \rightarrow \mathbb{R}$, $v(\mathbf{x}, t) : \partial\Omega \times I \rightarrow \mathbb{R}$, $u_0(\mathbf{x}) : \Omega \rightarrow \mathbb{R}$. Eqn.(1) is a parabolic partial differential equation (PDE).

Generally speaking, one wants the operator $\mathcal{L}$ to be uniformly elliptic in the spatial direction for the whole problem to be parabolic .

While well posedness for ODEs is satisfactorily answered in a generic way with the Picard-Lindelöf theorem, the same is not true for PDEs, not even linear ones. Often, existence and uniqueness are established on a case-by-case basis, and in many cases is not known at all (for example famously for the Navier-Stokes equation describing the motion of incompressible fluids). Nevertheless, there is huge interest from an applications perspective to obtain numerical solutions to PDE problems relevant to physics and engineering. General PDE theory will not be presented here. Instead, methods to combine known methods for IVPs and BVPs to solve problems of the type of problem 1 will be discussed. Clearly, the idea is to discretise the spatial direction by replacing $\mathcal{L}$ with an appropriate discretised operator $\mathcal{L}_h$ and using one of the time integration schemes introduced above to treat the time evolution.

The following simple IVBP represents the benchmark problem for testing numerical schemes for IVBPs in the present assignment.

Definition 2. Let $I = [0, T]$, $\Omega = (0, L)$, where $L \in \mathbb{R}^+$ and $T \in \mathbb{R}^+$, and consider the following IVBP:

$$\begin{cases} \partial_t u - k\partial_x^2 u = f & \text{for } x \in \Omega, \ \ \forall t \in \ I \\ u(0, t) = u(1, t) = 0 & \text{for } x \in \partial\Omega, \ \ \forall t \in I \\ u(x, 0) = u_0(x) & \text{for } x \in \Omega \end{cases} \tag{2}$$

Eqn.(2) is called the heat equation. Specifically, $u(x, t)$ describes the diffusive transport (of matter or heat) in space, with constant diffusivity/conductivity $k \in \mathbb{R}^+$, and where $f(x, t)$ describes the local heat production. The heat equation is well posed.

1.1 Numerical methods to approximate solution to IBVPs

Two different approaches to finding numerical solution to IBVPs can be adopted. The first one results in the so-called *semi-discretisation schemes*, in which a generic IBVP is first discretised in space and then in time or viceversa. Alternatively, a simultaneous discretisation, leading to a *fully discretised scheme*, can be performed.
A detailed presentation of *fully-discretised schemes* can be found in Ref.[1]. In the present report it will only be pointed out that, although a fully-discretised approach appears to be more convenient, in the case of stiff problems approximated by means of a method characterised by a bounded region of absolute stability, the choice of the discretisation time step and of the mesh size cannot be done arbitrarily, since the two are related by the Courant-Friedrichs-Levy conditions (see [1]).

1.1.1 Method of lines

In the present assignment only one of two possible semi-discretisation approaches will be presented. A more general discussion of them can again be found in Ref.[1] and Ref.[2]. In particular, only the so called *method of lines* will be presented.

Let N_x and N_t denote the number of spatial and temporal discretisation points respectively. Also let $(x_i)_{i=0}^{N_x+1}$ be a set of points, that discretise $\Omega = [0, L]$. One has:

$$0 = x_0 < x_1 < \cdots < x_{N_x} < x_{N_x+1} = L$$

Define $I_k := [x_k, x_{k+1}]$, with $|I_k| := h$, for $k = 0, \ldots, N_x$. The variable $h \in \mathbb{R}^+$ is the spatial mesh size. Finally, denote by $u_i : I \to \mathbb{R}$ the approximation of the exact solution u at x_i for $i = 1, \ldots, N_x$. The method of lines takes the position of first discretising in space. Such a discretisation is here performed through second order central finite differences. In the case of Eqn.(2), the resulting problem becomes a system of ordinary differential equations of the form:

$$\frac{du_i}{dt} = \kappa \partial^+ \partial^- u_i = \kappa \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}, \quad i \in \{1, \ldots, N_x\}$$

Such a problem can be equivalently restated in a convenient matrix form. Denoting by $\mathbf{u} = (u_1, \ldots, u_{N_x}) : I^{N_x} \to \mathbb{R}$, one has:

$$\frac{d\mathbf{u}}{dt} = \mathbf{f}(t, \mathbf{u}) = -\frac{\kappa}{h^2} \begin{pmatrix} 2 & -1 & & & & -1 \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & & & -1 \\ -1 & & & & -1 & 2 \end{pmatrix} \mathbf{u} \tag{3}$$

Periodic boundary conditions (PBCs) have been additionally imposed on the problem (see [2]) as required by the assignment instructions[1].

---

[1]In Eqn.(3) the blank entries represent zeros of tre matrix, which has size $N_x \times N_x$.

Hence, the semidiscretisation generates an IVP for a system of ODEs which may be solved by known methods for IVPs. In this formalism, one has:

$$\begin{cases} \frac{du_i}{dt} = \partial^+ \partial^- u_i + f_i & \forall x_i \in \Omega_h \\ u_i = 0 & \forall x_i \in \Gamma_h \end{cases} \tag{4}$$

with initial conditions $u_i(0) = u_0(x_i)$ for $x_i \in \Omega_h$, $i \in \{1 \dots N_x\}$. Here Ω_h represents the set of points in the interior of Ω and Γ_h the set of points on the boundary of Ω, once the spatial discretisation has taken place.

To complete the discretisation a solver for ODEs has to be applied to the system. In order to do so, a time-discretisation needs to be performed. Similarly to what has been previously done for the spatial case, let $(t_n)_{n=0}^{N_t+1}$ denote the discretisation of the temporal space $I = [0, T]$. Hence:

$$0 = t_0 < t_1 < \cdots < t_{N_t} < t_{N_t+1} = T$$

where $|J_k| = |[t_k, t_{k+1}]| := \tau \in \mathbb{R}^+$ is the temporal step size, for $k = 0, \dots, N_t$. Moreover, denote by $\mathbf{u}^i$ the approximation of the exact solution of the IVP for $\mathbf{u}$ at t_i for $i = 1, \dots, N_t$.

1.2 The present case

In the present assignment we aim to solve Eqn.(2) in the case of $L = 1$, $T = 0.1$, $f \equiv 0$, PBCs and the following initial condition:

$$u_0(x) = \begin{cases} 0 & \text{if } x \leq \frac{1}{4}, \ x > \frac{3}{4} \\ 1 & \text{otherwise} \end{cases} \tag{5}$$

System of equations (4) will be solved by means of Runge-Kutta (RK) methods. The implementation of such methods for the present problem will be now discussed. Set $f(t_i, \mathbf{u}^i) = f_i$. An s-stage RK method for system (3) takes the form:

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \tau \mathbf{F}(t_n, \mathbf{u}^n, f, \tau)$$

where the increment function F is defined as

$$\mathbf{F}(t_n, \mathbf{u}^n, f, \tau) = \sum_{i=1}^{s} b_i \mathbf{K}_i$$

$$\mathbf{K}_i = f\left(t_n + c_i \tau, \mathbf{u}^n + \tau \sum_{j=1}^{s} a_{ij} \mathbf{K}_j\right), \quad \text{for } i \in \{1, \dots, s\}$$

and $A = (a_{ij}) \in \mathbb{R}^{s \times s}$, $\mathbf{b}, \mathbf{c} \in \mathbb{R}^s$ completely determine the Runge-Kutta method.

As it is well known, a specific Runge-Kutta method can be denoted by its *Butcher tableau*, which takes the following form:

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b} \end{array}.$$

The forward Euler method, the backward Euler method and the 3-stage Heun method will be used in this assignment. Their Butcher tableaus are reported here:

4

3-stage Heun (Heun3)

Forward Euler (FE)	Backward Euler (BE)

$$
\begin{array}{c|c}
0 & \\
\hline
 & 1
\end{array}
\qquad
\begin{array}{c|c}
1 & 1 \\
\hline
 & 1
\end{array}
\qquad
\begin{array}{c|ccc}
0 & & & \\
1/3 & 1/3 & & \\
2/3 & 0 & 2/3 & \\
\hline
 & 1/4 & 0 & 3/4
\end{array}
$$

It should now be clear that the numerical implementation of the code is a combination of the techniques proposed for the previous assignments and, in particular, extends the code used in Assignment 3 to a "vectorial case".

It is worth pointing out how such an extension takes place in the case of implicit RK methods, i.e RK methods where at each stage $\mathbf{K}_i$, for a specific $i \leq s$, $s \in \mathbb{N}$ is determined by solving the implicit equation:

$$
\mathbf{K}_i = \mathbf{f}\left(t_n + \tau c_i, \mathbf{u}^n + \tau \sum_{j=1}^{i-1} a_{ij}\mathbf{K}_j + \tau a_{ii}\mathbf{K}_i\right) \tag{6}
$$

Thus, at each stage i, one has to solve for $\mathbf{K}_i$ through other means, which in this case coincide with the Newton's method. Using the Newton method, one solves for the root of the function $\mathbf{g}$ given by:

$$
\mathbf{g}(\mathbf{K}_i) := \mathbf{f}\left(t_n + \tau c_i, \mathbf{u}^n + \tau \sum_{j=1}^{i-1} a_{ij}\mathbf{K}_j + \tau a_{ii}\mathbf{K}_i\right) - \mathbf{K}_i
$$

$\mathbf{K}_i$, which is the root of g, is approximated through the following iterative equation:

$$
\mathbf{K}_i^{k+1} = \mathbf{K}_i^k - (J_\mathbf{g}(\mathbf{K}_i^k))^{-1}\mathbf{g}(\mathbf{K}_i^k),
$$

$\mathbf{K}_i^k$ being the kth iteration of the approximation of $\mathbf{K}_i$ and $J_\mathbf{g}(\mathbf{K}_i^k)$ is the Jacobian of $\mathbf{g}$ at the point $\mathbf{K}_i$. We note that to perform the inversion of $(J_\mathbf{g}(\mathbf{K}_i^k))$, iterative methods have to be used.

2 The program

2.1 Implementation

The program is built on the classes and struct that can be found in the files *vector.hh*, *vector.cc*, *sparse.hh*, *sparse.cc*, *models.hh* and *schemes.hh*.

In the *vector*-files the class *Vector*, which generalises the *std::vector* class is built. The *SparseMatrix* class construction can be found in the *sparse*-files instead. Being that such files have been provided along with the assignment instructions and that their content was studied for the lessons and for Assignment 1 and Assignment 2, no discussion of them will be here reported. However, it is worth pointing out that a slight modification of the function *Vector::toFile*, provided in the *vector*-files has been performed. Such a modification, which is reported for clarity in the following, eases the printing of the solution to the chosen file.

```
file << std::left << ((double) i)/(size()-1)*L << " " << (*this)[i] << std::endl; // -1
    is added to the provided code to make it print correctly
```

Also, in order to reduce the amount of lines printed on the terminal, one line has been commented out from the *ConjugateGradient* function contained in *sparse.cc*. For clarity reasons, such a line is explicitly reported in this report.

```
if ( (iter % 100) == 0 ) {
   const double resi = (b - A*x).maxNorm();
   // std::cout << "Iteration " << iter << ": residual="  << resi << std::endl;
   if ( resi < tolerance ) {
      return x;
   }
}
```

2.1.1   schemes.hh file

The class DIRK is made of 4 *protected* variables: the matrix A, the vector b, the vector c and the integer s (named stages_ in the code), which fully characterise the Butcher tableau of any RK method. The following instructions, defining the *private* variables of the class, can be found in the schemes.hh file:

```
protected:

   int stages_;

   std::vector<double> a_,b_,c_;
```

It may be worth pointing out that, even though a_- represents a matrix, such a variable is implemented as a vector in the program.
The classes FE (Forward Euler), BE (Backward Euler) and $Heun3$ (3-stage Heun) are implemented as derived classes of $DIRK$.

```
// forward Euler method
class FE: public DIRK
{
   public:

   FE() : DIRK(1)
   {
      a(0,0) = 0.;
      b_[0] = 1.;
      c_[0] = 0.;
```

```cpp
    }

};

// backward Euler method
class BE: public DIRK
{
   public:

   BE() : DIRK(1)
   {
      a(0,0) = 1.;
      b_[0] = 1.;
      c_[0] = 1.;
   }

};

// Heun3 method
class Heun3: public DIRK
{
   public:

   Heun3() : DIRK(3)
   {
      a(1,0) = 1./3;
      a(2,1) = 2./3;
      b_[0] = 0.25;
      b_[1] = 0.;
      b_[2] = 0.75;
      c_[0] = 0;
      c_[1] = 1./3;
      c_[2] = 2./3;
   }
```

The function *evolve*, which returns the evolved approximated solution $\mathbf{u}^{n+1}$ starting from $\mathbf{u}^n$, i.e. it evolves the solution to time $t + \tau$ from time t, can be found in the file as well. For flexibility reasons, the function is built as a *template*. The prototype of the routine, is displayed in the following:

```cpp
template <class Model>
Vector evolve(const Vector &y, double t, double h, const Model &model) const // h = tau
```

Here y represents the approximated solution at the time t, with h standing for τ, i.e. the chosen discretisation stepsize.
As pointed out before, in case an implicit RK method is chosen, an implicit equation for $\mathbf{K}_i$ has to be solved by means of numerical methods: Newton's method has been used here.

```
if (a(s,s) != 0)
{
    int iter = 0; // iter is the counter to stop the while loop after a certain number of
        iterations
    Vector error = (model.f(t + h*c_[s], temp_sum + h*a(s,s)*k[s]) - k[s]);
    count ++; //efficiecy count
    SparseMatrix Jac = model.df(t + h*c_[s], temp_sum + h*a(s,s)*k[s]); //jacobian (see
        models.hh)
    count ++; //efficiency count
    while (iter < 1e6 && error.maxNorm() > 1e-6)
    {
        k[s] = Jac.ConjugateGradient((-1)*model.f(t + h*c_[s], temp_sum + h*a(s,s)*k[s]) +
            k[s], k[s], 1e-6, 1e6) + k[s];
        count ++; //efficiency count
        error = (model.f(t + h*c_[s], temp_sum + h*a(s,s)*k[s]) - k[s]);
        count ++; //efficiency count
        iter++;
    }
}
```

As it can be seen from the listed code above, the program checks if Eqn.(6) is an implicit equation for $\mathbf{K}_i$: if the entry $a_{ss} \neq 0$, then Newton's method is implemented. Inversion of the Jacobian matrix is performed by means of *ConjugateGradient* alogorithm, which has been proven to be faster the *GaussSeidel* one in several cases.

The variable *tol* represents the tolerance and has been here set to 10^{-6}: the method stops once the distance (in L_∞ norm) between the RHS and LHS of Eqn.(6) is smaller than the fixed tolerance or after a given number of iterations (which has been chosen to be equal to 10^6) has been reached.

2.1.2 models.hh file

In this file the class *HeatEquation*, whose elements completely characterise the model at hand, is defined. The class contains the variables *N*, *kappa* and *A*.

```
int N; // N denotes the number of spatial discretisation points
double kappa; // kappa is the thermal diffusivity parameter
SparseMatrix A; // A is the matrix in the matrix IVP
```

A default constructor, setting the values of the variables of the class has been implemented. The other elements of *HeatEquation* are the functions f, df, T and y_0. The prototypes of such functions is reported in the following.

```
HeatEquation(int N_, double kappa_) // Default Constructor
: N(N_), kappa(kappa_), A(SparseMatrix(N))
{
    double h = 1./(double)(N + 1); // Spatial stepsize h
    double UD = kappa/(h*h); // Upper diagonal terms of A
```

```cpp
   double D = -2*kappa/(h*h); // Diagonal terms of A
   double LD = kappa/(h*h); // Lower diagonal terms of A
   for (int i = 0; i < N; ++i)
   {
     for  (int j = 0; j < N; ++j)
     {
       if(j == i + 1) // Upper-diagonal entries
       {
         A.addEntry(i, j, UD);
       }
       else if(j == i) // Diagonal entries
       {
         A.addEntry(i, j, D);
       }
       else if (j == i - 1) // Lower-diagonal entries
       {
         A.addEntry(i, j, LD);
       }
     }
   }
}

// returns f(t,y) in the IVP obtained for method of lines
Vector f(double t,const Vector &y) const
{
   return A*y;
}

// returns the gradient of f, which is the Jacobian matrix. Here is just A
const SparseMatrix& df(double t,const Vector &y) const
{
   return A;
}

// returns the interval length in time
double T() const
{
   return 0.1;
}

// y0 is the initial function at time t = 0
Vector y0() const
{
   double h = 1./(N + 1);
   Vector v(N);
   for (int i = 0; i < N; ++i)
   {
     if (i*h > 0.25 && i*h <= 0.75)
     {
       v[i] = 1;
     }
```

```
    }
    return v;
}
```

The comments associated to the functions completely describe each of them, hence no additional description is required.

It is however worth noticing that the function T simply returns the total integration time without performing any other task. For the sake of generality and flexibility of the code, its function structure has been preserved: as a matter of fact, such a choice eases the extension of the present code to more general cases.

2.1.3 main.cc file

The main file contains the function *solve* which performs the integration of the chosen equation up to time T. Such a task is performed by means of repeated calls to *evolve*.

As it can be seen, the function returns the solution of the whole integration. In order to store such a solution in a suitable file, the function *toFile* has to be called.

A vector of pointers to DIRK is constructed.

```
std::vector<DIRK *> schemes; // Vector of schemes
schemes.push_back(new FE());
schemes.push_back(new BE());
schemes.push_back(new Heun3());
```

By accessing a suitable location of this vector through an appropriate choice of the input parameters, each of the RK methods can be employed. An explicit check on the consistency of the inserted parameters in implemented as well:

```
if ( argc<5 || atoi(argv[1])<1 || atof(argv[2])<0 || ( atoi(argv[3])!=0 &&
    atoi(argv[3])!=1 && atoi(argv[3])!=2 ) || atof(argv[4])<=0 )
{
   std::cerr << "Usage: " << std::endl
   << " " << argv[0] << " <Nx> <kappa> <scheme> <tau>" << std::endl << std::endl
   << "where" << std::endl
   << " <Nx>      spatial resolution (int) -- required to be >2 " << std::endl
   << " <kappa>   diffusivity (double) -- only takes positive values" << std::endl
   << " <scheme>  scheme index (int) -- FE=0, BE=1, Heun3=2" << std::endl
   << " <tau>     temporal time step (double) -- only takes positive values" <<
        std::endl;
   return 1;
}
```

In particular, the program checks if $N_x > 1$, $k > 0$, $tau > 0$ and if the scheme index has been inserted correctly, i.e. if the scheme index corresponds to a scheme implemented in the code. In the

event of a wrong insertion, the program exits with a suitable warning to the user.

2.2   Running the program

The program is run through the Makefile provided. The program accepts 3 arguments (not counting the "./" instruction) from the prompt (here the Makefile) and, whenever an incorrect insertion of them happens, the program exits with an appropriate warning the user.
The current optimisation is set to -Ofast and all following tests have been performed with this optimisation choice. Every possible error or warning has been explicitly checked by means of the instruction -*Wall* -*Wfatal-errors* -*pedantic* before proceeding to performing the different tests: no error appeared. Regarding the warnings, they refer to the provided .hh files and none of them affects the correctness of the code: for these reasons, it has been chosen not to deal with them.
The program generates a fixed number of output files, in which the results of the tests have been stored. Some of these files are used to produce plots by means of different plotscripts.

2.3   Results of the tests

In order to check the correctness of the implementation of the code, the tests requested by the assignment have been performed. In the present assignment, the following choice for the values of k to test on has been made:
$$k_1 = 1.0, \quad k_2 = 0.1$$

In what follows, the results of the more significant tests performed will be reported and briefly discussed.

2.3.1   Choice of temporal stepsize for $N_x = 16$

Having fixed two values of k (k_1 and k_2 respectively) and a moderate spatial resolution ($N_x = 16$), various temporal stepsize τ for the numerical integration have been tested. In the following, we will refer to "unstable scheme" with the meaning of a scheme producing an inconsistent solution to Eqn.(2), i.e a solution growing indefinetely in time (see also §2.3.3).
The obtained results will be now presented:

- FE scheme: the scheme has been shown to be unstable for $\tau = 0.1$ for both values of k. In case the choice of $\tau = 0.01$ is made, the aforementioned unstable behaviour is observed only in the case of $k = 1.0$. For values of $\tau \leq 0.001$, the scheme has been shown to be stable for both k_1 and k_2.

- BE scheme: the scheme has been shown to be stable for $\tau \leq 0.1$. However, in the case of $k = 1.0$ and $\tau = 0.01$, the conjugate gradient method is not able to reach tolerance after 1000000 iterations and the program exits. Further analyses on this particular case may be conducted by means of making the fixed number of iteration the user is willing to wait bigger than 10^6.

- Heun3 scheme: the exact same results of the FE method have been observed.

It has also been shown that the choice of $\tau = 10^{-3}$ guarantees the convergence of all the RK methods employed in the integration of Eqn.2. Such a result is reproduced by default by the program and can be seen explicitly without altering any parameters in the Makefile provided.
As a matter of fact, the choice of smaller stepsizes, up to $\tau = 10^{-6}$, has additionally been seen to work for the FE and $Heun3$ schemes. Finally, the choice of $\tau = 10^{-4}$ has been shown to work for BE scheme and $k = 1.0$.

2.3.2 Comparing solutions over time

As required by the assignment instructions, two different sets of values for k, τ and N_x have been chosen in order to compare the respective solutions after one time step and after 5 different equispaced points in time later on. The following choice has been made:

$$k_1 = 1.0, \quad \tau_1 = 0.001, \quad N_{x_1} = 20 \tag{7}$$

$$k_2 = 0.1, \quad \tau_2 = 0.001, \quad N_{x_2} = 16 \tag{8}$$

Such a choice ensures a fast convergence of all the methods used in the assignment, hence the easy reproducibility of the test performed by a generic user.
Plots for the different set of parameters will be reported in the following. Being that the produced plot are 2D, different colours have been chosen to draw the profile of the approximated solutions at different times. Each colour is associated with a specific time value, where, for convenience reason, the sampling associated with the first time step is denoted by t_0. Every other t_i, $i \in \{1, \ldots, 5\}$, is associated with a sampling at time $\frac{iT}{5}$, as required in the assignment instructions.
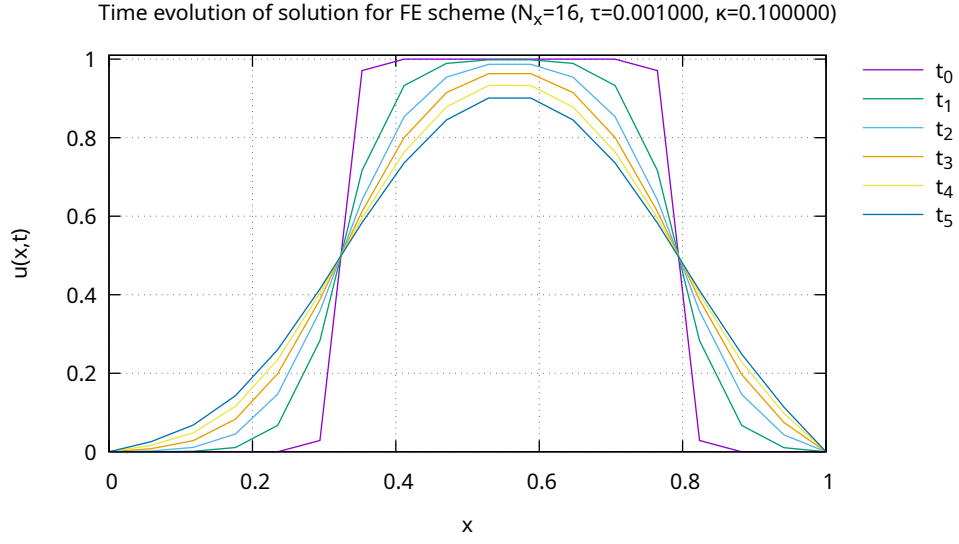
Figure 1: Graphical representation of the evolution over time of the approximated solution to Eqn.(2). The associated parameters are reported in the title of the plot.
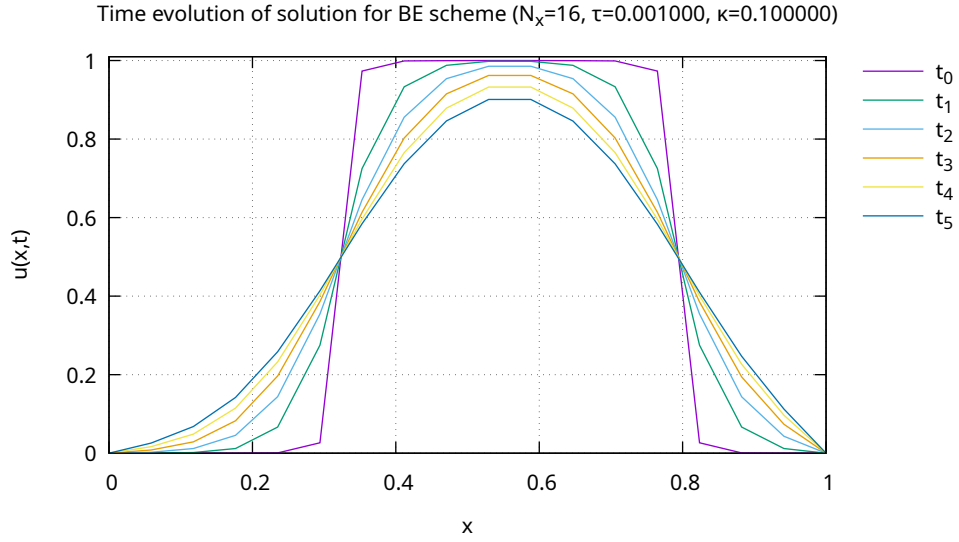

Figure 2: Graphical representation of the evolution over time of the approximated solution to Eqn.(2). The associated parameters are reported in the title of the plot.
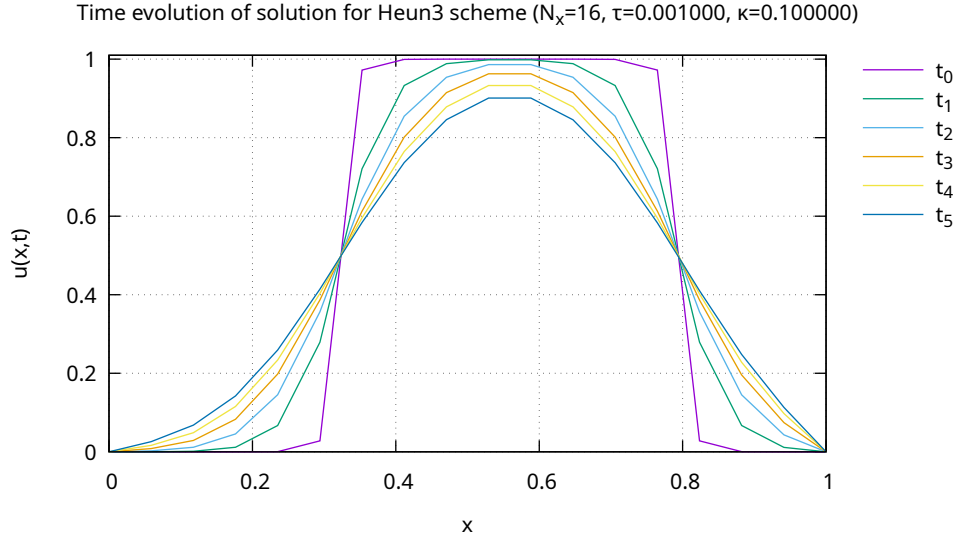
Figure 3: Graphical representation of the evolution over time of the approximated solution to Eqn.(2). The associated parameters are reported in the title of the plot.

For the set of parameters (7) the solution obtained with FE scheme, BE scheme and Heun3 scheme appear to be similar to each other. This is somewhat expected since the test has been conducted on a set of value of k, τ and N_x which guarantees stability of all the used methods.

Solution are also consistent with what one would expect when solving the heat equation analytically: the observed profiles are the approximation of the mollification of the initial condition (5). A small variation of the solution profile over time is indeed expected due to the small value of k, which plays the role of an inverse "characteristic time".

Regarding set of parameters (8), one finds:

Figure 4: Graphical representation of the evolution over time of the approximated solution to Eqn.(2). The associated parameters are reported in the title of the plot.

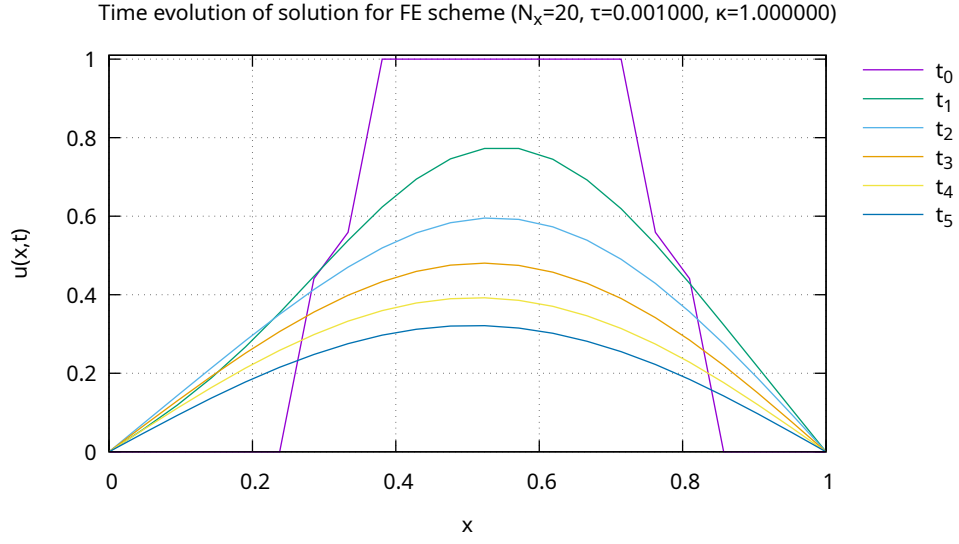Figure 5: Graphical representation of the evolution over time of the approximated solution to Eqn.(2). The associated parameters are reported in the title of the plot.
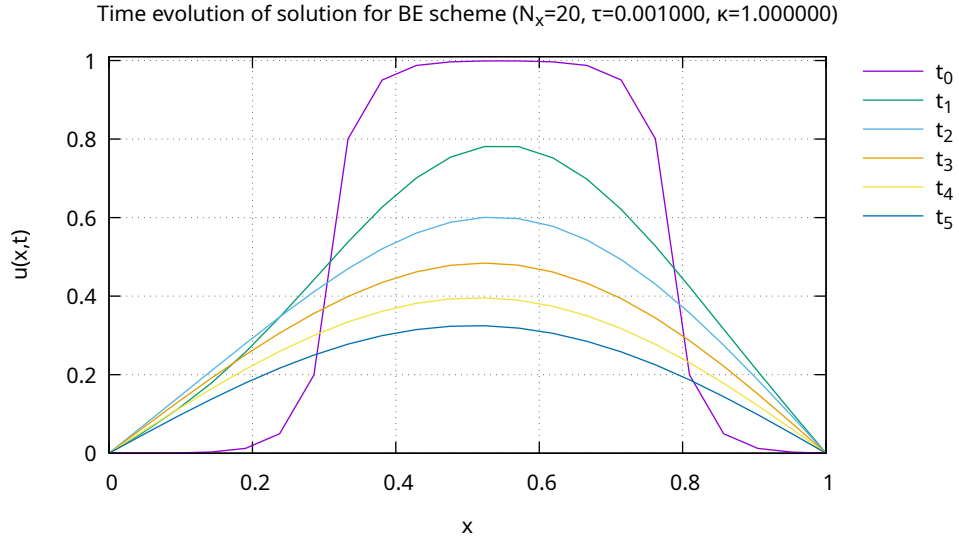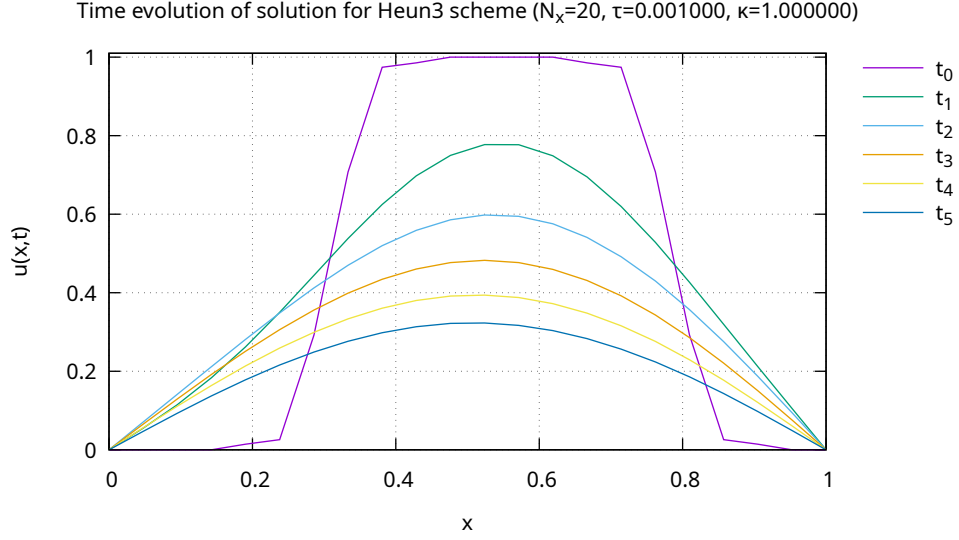
Figure 6: Graphical representation of the evolution over time of the approximated solution to Eqn.(2). The associated parameters are reported in the title of the plot.

As for the previous case, solutions look very similar to each other: this provides an indirect proof of the correctness of the integration performed. However, it may be worth noticing that, even though solutions appear very close to each other for $t \geq t_1$, slightly different curves are observed in the case of τ_0: the profiles associated with the Heun3 and BE methods are smoother than the FE method counterpart.
Differently for the prior case, however, the profile changes quite significantly with time. Once again, the reason of such a behaviour is related to the value of k, which happens to be one order of magnitude greater than in the previous set of parameters. Interpreting k as the inverse of a characteristic time, a faster relaxation of the solution is expected. In this sense, a dominance of one of the slower eigenmode of the solution is observable in the previous plots.

2.3.3 Efficiency analysis

Efficiency of the different methods has been analysed with a technique implemented for the previous assignment. The computational cost of each solution is evaluated in terms of calls to functions f and df contained in *models.hh*. The results of such tests, which are directly printed on the terminal, are reported in the following:

```
----------------------------------------------------------------------------------
-------------------------------------------------- EFFICIENCY
    ----------------------------------------------------------------
----------------------------------------------------------------------------------
-------------------------------------------------- FE
    ----------------------------------------------------------------
```

```
./myProgram 16 1.0 0 0.001
Finished time integration at t=0.099

Number of calls to f and df in scheme 0 is 99
./myProgram 25 1.0 0 0.001
Scheme 0 blows up for N=25, kappa=1.000000, tau=0.001000
Finished time integration at t=0.002
Number of calls to f and df in scheme 0 is 2
--------------------------------------------- BE
    ----------------------------------------------------------------------
./myProgram 16 1.0 1 0.001
Finished time integration at t=0.099
Number of calls to f and df in scheme 1 is 476013
./myProgram 25 1.0 1 0.001
Finished time integration at t=0.099
Number of calls to f and df in scheme 1 is 516483
--------------------------------------------- Heun3
    ---------------------------------------------------------------------
./myProgram 16 1.0 2 0.001
Finished time integration at t=0.099
Number of calls to f and df in scheme 2 is 297
./myProgram 25 1.0 2 0.001
Scheme 2 blows up for N=25, kappa=1.000000, tau=0.001000
Finished time integration at t=0.001
Number of calls to f and df in scheme 2 is 3
```

The listed "code" above clearly shows the efficiency and stability performance of the different methods. It is clear that, measuring computational cost as described above, the FE method appears to be the most efficient of the method used in this assignment.

However, the above analysis shows that both the FE and the Heun3 method generate an inconsistent solution for $N = 25$, $k = 1.0$, $\tau = 0.001$, while the BE method provides the correct solution for such a choice of parameters.

The relevant trade off in numerical integration of Eqn.(2) is thus given by the choice of an explicit or an implicit method. As a matter of fact, if on the one hand an explicit method (like the FE one) guarantees a very short computational cost, on the other, an implicit method (like the BE one) guarantees a good behaviour of the solution, in case the given problem experiences Courant-Friedrichs-Levy's condition type of behaviour. The usage of implicit methods is strongly advised when dealing with similar problems (see [2], [1]).

The "solution-blow-up" analysis just described has been performed as suggested in the assignment instructions, i.e. by means of setting a target wallclock of 10 seconds and looking at which solution experiences a blow up within that amount of time. Such a target wallclock has been implemented through the following instructions:

```
double timeStart = clock(); //this is for initialising the target wallclock
if (y.maxNorm()>1 || (clock() - timeStart) / CLOCKS_PER_SEC >= 10) // time in seconds.
    Tarhget wallclock fixed as required
{
    std::cout << "Scheme " + std::to_string(schemeNumber) + " blows up for N=" +
```

```
        std::to_string(model.N) + ", kappa=" + std::to_string(model.kappa) + ", tau=" +
        std::to_string(tau) << std::endl;
    break;
}
```

3 Conclusive remarks

In this section some additional comments and observation are presented.

3.1 Memory

As for the previous assignments, possible memory leaks have been checked.with the help of the software *valgrind*, To do this, it is necessary to compile with *-g* and *-O1* optimisation. After having compiled the program, the following instruction has been used:

```
$ valgrind --leak-check=yes ./MyProgram 16 1.0 0
```

In the present case the parameters refer to the case of $N_x = 16$, $k = 1.0$ and FE scheme.

The following output has been produced on the terminal:

```
==18980== Memcheck, a memory error detector
==18980== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==18980== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==18980== Command: ./MyProgram 16 1.0 0
==18980==
Finished time integration at t=0.099
==18980==
==18980== HEAP SUMMARY:
==18980==     in use at exit: 0 bytes in 0 blocks
==18980==   total heap usage: 814 allocs, 814 frees, 163,932 bytes allocated
==18980==
==18980== All heap blocks were freed -- no leaks are possible
==18980==
==18980== For counts of detected and suppressed errors, rerun with: -v
==18980== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

One can clearly see that no memory has been lost. Hence, the instruction:

```
for (const auto& r : (schemes)) //freeing memory
{
    delete r;
}
```

which is contained in the main file, works correctly.

3.2 Performances

Performances can be checked through the following commands:

```
$ valgrind --tool=callgrind ./MyProgram 16 1.0 0
$ kcachegrind
```

where the parameters inserted from terminal have to be changed according to the desired model, scheme, initial step size and number eocs computed. In the present case the parameters refer o the case of $N_x = 16$, $k = 1.0$ and FE scheme.
The first command analyses the perfomances in the terms of load distrubution, while *kcachegrind* is used to visualise the load distribution results.

Such an analysis shows that the load is well distributed among the different functions of the program.

References

[1] A. Quateroni, R. Sacco, F. Saleri; *Numerical Mathematics*, Vol.37, Springer Verlag, (2007).

[2] T. Grafke; *Scientific Computing*, Lecture Notes, University of Warwick, (2018).