Report on results of the students' project about Crank-Nicolson method for advection equation

P. Frolkovič

Faculty of Engineering, Radlinského 11, 81005 Bratislava, Slovakia
peter.frolkovic@stuba.sk

K. Balaj, M. Holý, K. Juhásová, A. Petričko, A. Štuller

Faculty of Informatics and Information Technologies, Ilkovičova 2, 842 16 Bratislava, Slovakia

July 6, 2021

Keywords: advection, Crank-Nicoslon, finite difference method, upwind, numerical solution

Abstract

Our aim is to implement and test some less known numerical methods to solve the linear advection equation. The description is restricted to information necessary to implement and test the presented methods and it is given only for one-dimensional case. We can show that a parametric class of Crank-Nicolson type method can be used conveniently that is second order accurate and stable. The report is accompanied by available code in Python at

`github.com/PeterFrolkovic/Crank_Nicolson_Method_for_Advection`

1 Model

We deal for a simplicity with the one-dimensional advection equation

$$\partial_t \phi + v \partial_x \phi = 0\,. \tag{1}$$

The velocity function $v = v(x)$ is given and (for a simplicity) positive for $x \in [0, L]$. The unknown function $\phi = \phi(x, t)$ shall be determined for $t \in [0, T]$ and $x \in [0, L]$ with two positive real constants T and L given. The partial derivatives with respect to time and space are denoted by $\partial_t \phi$ and $\partial_x \phi$, respectively.

The values of ϕ are prescribed for $t = 0$ by the initial condition

$$\phi(x, 0) = \phi^0(x)\,,\ x \in [0, L] \tag{2}$$

where the initial function ϕ^0 is given.

We consider two types of required boundary conditions. First, the Dirichlet boundary condition is prescribed by

$$\phi(0,t) = \phi_0(t)\,,\ t \in [0,T]\,, \tag{3}$$

where the function ϕ_0 is given. In this case we expect that $\phi^0(0) = \phi_0(0)$ or, otherwise, we prefer to define $\phi(0,0) = \phi_0(0)$.

Alternatively, the periodic boundary condition is prescribed

$$\phi(0,t) = \phi(L,t)\,. \tag{4}$$

In this case we expect that $\phi^0(0) = \phi^0(L)$.

Typical examples with available exact solutions are

1. $v(x) = \bar{v} = \text{const} \ \Rightarrow\ \phi(x,t) = \phi^0(x - \bar{v}t)$
2. $v(x) = 2 + \sin(x) \ \Rightarrow\ \phi(x, 2\pi/\sqrt{3}) = \phi^0(x - 2\pi)$

For the case of constant velocity like given in 1.), one can choose any initial function ϕ_0. Typical choices of smooth and nonsmooth initial profiles can be found in Figure 1 or in [2].

For the example 2.) with variable velocity one can choose again arbitrary initial function, where the exact solution is given only at the specific time $T = 2\pi/\sqrt{3}$.

Note that the initial and boundary condition and the constants T and L shall be chosen in accordance with the form of exact solution.

1.1 Preliminary task

Define in Python an example where the initial function is, e.g., a quadratic function, and the velocity function takes the value 1. Define the exact solution and the function ϕ_0 using the previously defined data. Plot the exact solution for some nonzero time. *Done on May, 6th.*

2 Review of well-known numerical methods

We derive all schemes using the notation of finite difference methods. Let us denote $x_i = ih$ for $i = 0, 1, \ldots, I$ where the positive integer I is chosen and $h = L/I$. Analogously, $t^n = n\tau$ for $n = 0, 1, \ldots, N$ for chosen positive integer N and $\tau = T/N$. The numerical solution is denoted by Φ_i^n and $\Phi_i^n \approx \phi_i^n$ where $\phi_i^n := \phi(x_i, t^n)$. Analogously, $v_i := v(x_i)$. The initial condition is used to define

$$\Phi_i^0 = \phi^0(x_i)\,. \tag{5}$$

The simplest numerical method to solve (1) can be derived as follows. We approximate it at (x_i, t^n) using the forward finite difference for the time derivative

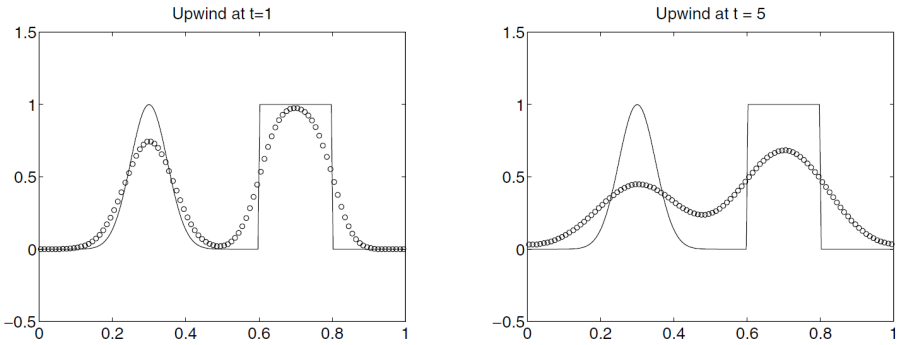

Figure 1: The exact (bold) and numerical (circles) solution for an example with constant velocity and periodic boundary condition after one and five periods taken from [1]. The numerical results were obtained with the method (7) using Courant number $C = 0.8$

$\partial_t \phi_i^n$ and the backward finite difference to approximate $\partial_x \phi_i^n$. Consequently, we get for $n = 0, 1, \ldots, N-1$ and $i = 1, 2, \ldots, I$

$$\frac{\Phi_i^{n+1} - \Phi_i^n}{\tau} + v_i \frac{\Phi_i^n - \Phi_{i-1}^n}{h} = 0 \tag{6}$$

or equivalently,

$$\Phi_i^{n+1} = (1 - C_i)\Phi_i^n + C_i \Phi_{i-1}^n \tag{7}$$

where we use the notation for so called (grid) Courant number

$$C_i = \frac{\tau v_i}{h} \tag{8}$$

In the case of Dirichlet boundary condition we use

$$\Phi_0^{n+1} = \phi_0(t^{n+1}) \,. \tag{9}$$

In the case of periodic boundary conditions we use

$$\Phi_0^{n+1} = \Phi_I^{n+1} \,. \tag{10}$$

Note that the Dirichlet boundary conditions can be implemented "in advance" (separately), similarly to the initial conditions (5), before using (7). In the case of periodic boundary conditions one shall use (10) for some $n+1$ only after the scheme (7) has been used for $i = I$.

The scheme (7) is only first order accurate and it is conditionally stable with the stability restriction $C_i \leq 1$. A typical results obtained with this method are plotted in Figure 1.

To avoid the stability restriction we can derive the following scheme. Now we approximate the advection equation (1) at (x_i, t^{n+1}) using the backward finite

difference for the both partial derivatives, so

$$\frac{\Phi_i^{n+1} - \Phi_i^n}{\tau} + v_i \frac{\Phi_i^{n+1} - \Phi_{i-1}^{n+1}}{h} = 0\,. \tag{11}$$

Consequently we get for $n = 0, 1, \ldots, N-1$ and $i = 1, 2, \ldots, I$

$$(1 + C_i)\Phi_i^{n+1} - C_i \Phi_{i-1}^{n+1} = \Phi_i^n\,. \tag{12}$$

The boundary conditions are treated equivalently as defined before.

There is no stability restriction for this implicit scheme. To use it one has to solve the system of linear algebraic equation with a two-diagonal matrix.

2.1 Preliminary task

2.1.1 Explicit method

Add to the previous program from task 1.1 the definition of input parameters for numerical methods and the initialization of variables x_i, t^n, v_i, Φ_i^0, and Φ_0^n. Implement the method (7), if possible in a vectorised form. Compare the numerical solution with the exact one visually and by computing the error as written in (16) later. Test the behaviour of the method for the Courant number less, equal or greater than 1. *Finished on May, 11th.*

2.1.2 Implicit method

Rewrite the method (12) by collecting all coefficients for two unknowns Φ_i^{n+1} and Φ_{i-1}^{n+1} using (8). Implement then the method (12) by defining a two-diagonal (i.e. a banded) matrix and the vector of right hand sides corresponding to (12) and find the solution by calling appropriate procedure in Python. Again, compare the numerical solution visually and by computing the error (16). Test the behaviour of the method for the Courant number less, equal or greater than 1. *Done on May, 11th.*

3 Crank-Nicolson methods

The task of this project is to implement the so-called Crank-Nicolson type of time discretization that should give the second order accuracy for the approximation of time derivative. To do it in a correct way, we have to suggest also a second order accurate approximation of the space derivative.

To do so, we denote such approximation with one free parameter $\kappa \leq 1$ that can be chosen in general different for each i,

$$\partial_x^\kappa \Phi_i^n = \frac{1}{4h}\left((1-\kappa)(3\Phi_i^n - 4\Phi_{i-1}^n + \Phi_{i-2}^n) + (1+\kappa)(\Phi_{i+1}^n - \Phi_{i-1}^n)\right)\,, \tag{13}$$

Typical choices for the free parameter are $\kappa = 1, 0, -1$ that represents three basic finite difference methods to approximate the first derivative in (13). In

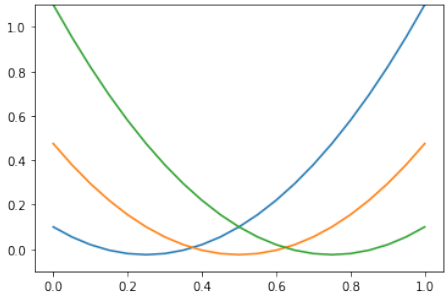

Figure 2: The numerical solution for three time steps with the Dirichlet boundary conditions and $\kappa = 0$ for the equation 15 if $a = 0.1, b = -1, c = 2, d = 0$ with 10 time steps and 10 spatial steps for $L = 1$ and $T = 0.5$. The error is zero.

fact, the choice $\kappa = 1/3$ is the third order accurate. Note that the choices $\kappa = -1$ and $\kappa = 1$ requires only the values of ϕ at three different points.

Having such definition, the Crank-Nicolson scheme can be given by

$$\Phi_i^{n+1} + \frac{\tau v_i}{2} \partial_x^\kappa \Phi_i^{n+1} = \Phi_i^n - \frac{\tau v_i}{2} \partial_x^\kappa \Phi_i^n \tag{14}$$

Concerning the boundary conditions, the treatment does not change. For the grid points x_1 and x_I one can use a special choice of κ in (13) to reduce the stencil conveniently.

The task of this project to implement and test this method. In fact, it is not easy to find it in literature except for the case $\kappa = 1$.

To test a numerical method for the solution of a differential equation, the following steps are suggested:

1. Simpler methods like in Section 2 can be tried first. *Done*

2. Test examples for which the numerical solution is known. In our case, the choice of constant velocity $\bar{v}$ with a polynomial function

   $$\phi(x,t) = a + b\,(x - \bar{v}t) + c\,(x - \bar{v}t)^2 + d\,(x - \bar{v}t)^3\,, \quad a, b, c, d \in R \tag{15}$$

   can be used. The numerical solution for the first order accurate methods like (7) or (12) must produce no error for arbitrary steps τ and h if $c = 0$ and $d = 0$. The Crank-Nicolson method shall be exact even if $c \neq 0$, but $d = 0$.

   Done. Solutions can be seen in the Figure 2 and Figure 3.

3. Test the second order accurate methods with (15) and $d \neq 0$ when the numerical error should be one quarter smaller when the steps τ and h are halved.

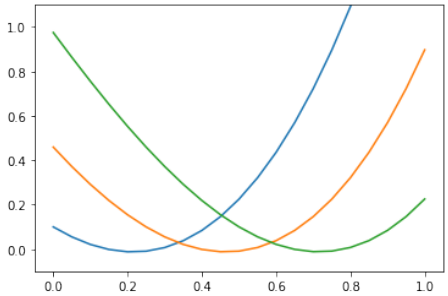

Figure 3: The numerical solution with the Dirichlet boundary conditions with $\kappa = 0$ for the equation 15. The example with $a = 0.1, b = -1, c = 2, d = 1$ with 10 time steps and 10 spatial steps for $L = 1$ and $T = 0.5$. The error based on the definition 16 and 17 is $E \approx 0.000104$ and $E_T \approx 0.000313$.

Done. The prove is in the Colab in the ***Error measurement section*** *in the* ***Testing accuracy with the known exact function****. However in case of $\kappa = 0$, the error is not one quarter smaller, but is quartered when when the τ and h are halved. Also this is only in the case of the error in the equation 16.*

4. Test examples for nontrivial exact solutions like the one in section 1 for which the experimental order of convergence shall be checked. Typical errors between the numerical solution Φ_i^n and the exact ones ϕ_i^n is given by

$$E = E(I, N) = \tau h \sum_{n=0}^{N} \sum_{i=0}^{I} |\Phi_i^n - \phi_i^n| \,. \tag{16}$$

or, if the error is computed only in the last time,

$$E_T = E_T(I, N) = h \sum_{i=0}^{I} |\Phi_i^N - \phi_i^N| \,. \tag{17}$$

5. Test examples for which the exact solution is not known, but it can be expected to exist. In such case one can check the behaviour of the method comparing two numerical solutions on two related discretization steps, see also `FIIT01.ipynb`. Let $\bar{\Phi}_i^n$ be numerical solution for some choice of $\bar{h} = L/\bar{I}$ and $\bar{\tau} = T/\bar{N}$ and Φ_i^n the numerical solution for $I = 2\bar{I}$ and $N = 2\bar{N}$, then one can check the difference

$$\bar{\tau}\bar{h} \sum_{n=0}^{\bar{N}} \sum_{i=0}^{\bar{I}} |\bar{\Phi}_i^n - \Phi_{2i}^{2n}| \tag{18}$$

Done. The ***Error measurement section*** *on the bottom of the colab in the subsection* ***Checking error when unknown exact solution****.*

3.1 Task 1

Implement the method (14) together with the Dirichlet boundary conditions (9). To do so, define the matrix and the right hand side of this system of linear algebraic equations to use a procedure in Python to solve it. Use a format of banded matrix that is available in SciPy. Test the program on several examples as defined in Section 3 and for several values of κ. Follow the approach described in section 3.

3.2 Task 2

Implement the method (14) together with the periodic boundary conditions (4). To do so, define the matrix and the right hand side of this system of linear algebraic equations to use a procedure in Python to solve it. Note that the resulting matrix is no more banded one, therefore use a sparse format that is available in SciPy. Test the program on several examples as defined in Section 3 and for several values of κ. Follow the approach described in section 3.

3.3 Task 3

Implement an efficient solver for 4 diagonal matrix based on Gauss elimination and use it for the Task 1.

Do it in two steps. First, eliminate the upper diagonal in a backward manner, i.e. for $i = I-1, I-2, \ldots, 1$. Second, use a forward substitution to solve the resulting three-diagonal system directly.

4 The results

4.1 Task 3

To implement the solver, we need to define the size of the matrix, its diagonals and the right hand side of linear system of equations. Let the value of I represent the number of rows (and columns) in the matrix. Variables a, b, c, d describe the diagonals, see (19) for an illustration, and z represents the right side of the equations.

The solver consists of two parts:

1. an elimination of the upper diagonal d so that we get the lower triangular matrix,
2. a forward substitution to solve the resulting triangular matrix.

With $I = 4$ the initial matrix is:

$$\begin{bmatrix} c_0 & d_0 & 0 & 0 \\ b_1 & c_1 & d_1 & 0 \\ a_2 & b_2 & c_2 & d_2 \\ 0 & a_3 & b_3 & c_3 \end{bmatrix} \tag{19}$$

Note that the first line in the matrix has the index 0 and the last one the index $I-1$.

The elimination of the upper diagonal is performed by Gauss elimination method for $i = I-1, I-2, \ldots, 1$:

$$\begin{aligned} k &= d_{i-1}/c_i \\ c_{i-1} &= c_{i-1} - k * b_i \\ b_{i-1} &= b_{i-1} - k * a_i \end{aligned} \tag{20}$$

In the process above, we can't forget to modify z as well:

$$z_{i-1} = z_{i-1} - k * z_i \tag{21}$$

To compute the solution x, the first two values have to be calculated separately. The rest of the values can be then calculated in a loop:

$$\begin{aligned} x_0 &= z_0/c_0 \\ x_1 &= (z_1 - b_1 * x_0)/c_1 \\ x_i &= (z_i - b_i * x_{i-1} - a_i * xi - 2)/c_i \end{aligned} \tag{22}$$

where $i = 2, 3, ..., I-1$. Python code below contains the implementation of the described method. To test it the solution obtained by the implemented method is compared with the result of scipy function. In both cases one obtains $x = [3, 1, -2, 1]$.

```
import numpy as np
from scipy.sparse.linalg import spsolve

I = 4
a = np.array([0, 0, 3, 4], dtype=float)
b = np.array([0, 12, -13, 1], dtype=float)
c = np.array([6, -8, 9, -18], dtype=float)
d = np.array([-2, 6, 3, 0], dtype=float)
z = np.array([16, 16, -19, -16], dtype=float)

A = np.array(
    [[6, -2, 0, 0],
    [12, -8, 6, 0],
    [3, -13, 9, 3],
    [0, 4, 1, -18]], dtype=float)

print("Solution from scipy package: ", np.linalg.solve(A, z))

def solve_4_diagonal_matrix(a, b, c, d, z, I):
    for i in range(I - 1, 0, -1):
        k = d[i - 1] / c[i]

        d[i - 1] = d[i - 1] - k * c[i]
        c[i - 1] = c[i - 1] - k * b[i]
        b[i - 1] = b[i - 1] - k * a[i]

        z[i - 1] = z[i - 1] - k * z[i]

```

```
    x = np.zeros(I)

    x[0] = z[0] / c[0]
    x[1] = (z[1] - b[1] * x[0]) / c[1]

    for i in range(2, I):
        x[i] = (z[i] - b[i] * x[i - 1] - a[i] * x[i - 2]) / c[i]

    return x

x = solve_4_diagonal_matrix(a, b, c, d, z, I)
print("Our solution: ", x)
```

Listing 1: 4-diagonal matrix solver

The output of the code above is:

```
Solution from scipy package:  [ 3.  1. -2.  1.]
Our solution:  [ 3.  1. -2.  1.]
```

5 Additional remarks

In general, the scheme (14) for $i = 1, 2, \ldots, I$ takes the form

$$\Phi_i^{n+1} + \sum_{k=-2}^{1} \alpha_{i+k} \Phi_{i+k}^{n+1} = \Phi_i^n + \sum_{k=-2}^{1} \beta_{i+k} \Phi_{i+k}^n \,. \tag{23}$$

where the coefficients α_* and β_* can be determined from (14). Note that these coefficients depend in our case only on C_i and κ.

The stability of schemes taking the form (23) can be found from investigating the values of so called amplification factor S that is a function of $x \in [0, 2\pi]$

$$S = \left| \left(1 + \sum_{k=-2}^{1} \beta_{i+k} \exp(\imath k x) \right) \left(1 + \sum_{k=-2}^{1} \alpha_{i+k} \exp(\imath k x) \right)^{-1} \right| \tag{24}$$

where $\imath$ denotes the imaginary number. Consequently, although S is a real parameter as it is defined with the absolute value $|.|$, to compute it one has to use an arithmetic with complex numbers. Of course, due to the presence of coefficients α and β in (24), the parameter S depends indirectly on C_i and κ.

References

[1] Randall J LeVeque et al. *Finite volume methods for hyperbolic problems*, volume 31. Cambridge university press, 2002.

[2] Hans Petter Langtangen and Svein Linge. *Finite difference computing with PDEs: a modern software approach*. Springer Nature, 2017.