Finite Elements in Fluids, Assignment 4, Burgers, Unsteady

Jose Raul Bravo Martinez, MSc Computational Mechanics

March 20, 2019

Part 1. On Unsteady Problems:

-Calculate the Courant Number
The Courant Number is an indicator of the stability of a scheme and is defined as:

$$C = \frac{|a|\Delta t}{h}$$

Where a is the convection velocity, Δt is the time step and h is the mesh size.

-Solve the problem using the Crank-Nicholson scheme in time and linear finite element for the Galerkin scheme in space. Is the solution accurate?

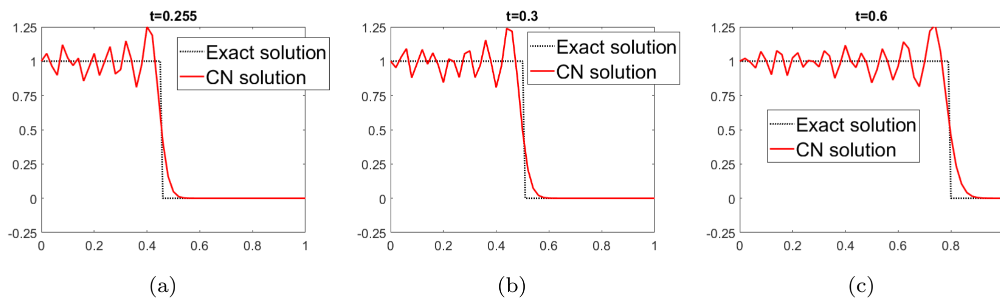
The problem to solve is the following:

$$\begin{cases} u_t + au_x = 0 & x \in (0,1), t \in (0,0.6) \\ u(x,0) = u_0(x) & x \in (0,1) \\ U(0,t) = 1 & t \in (0,0.6] \end{cases}$$

$$u_0 = (x) = \begin{cases} 1 & \text{if } x \leq 0.2, \\ 0 & otherwise \end{cases}$$

$$a = 1, \quad \Delta x = 2 \times 10^-2, \quad \Delta t = 1.5 \times 10^-2$$

The profile to transport is a steep front. Given the domain of [0,1] and the time frame, the following data is entered in the MATLAB code: $a = 1$, $NumberTimeSteps = 40$, $NumberElements = 50$.

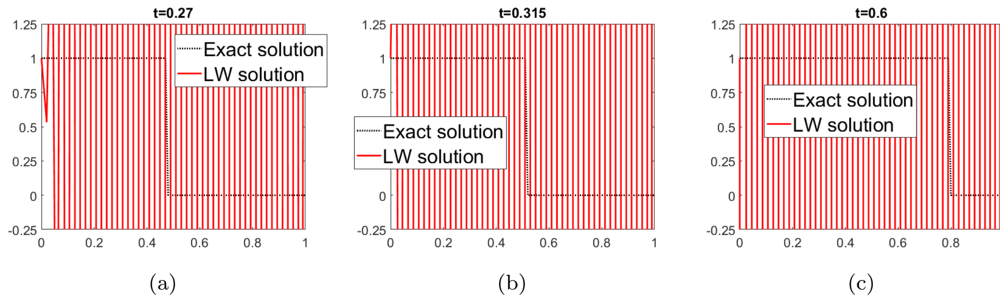


(a) (b) (c)

Figure 1: Crank Nicholson Results for Courant Number 0.75

-Solve the problem using the second-order Lax-Wendroff method. Can we expect the solution to be accurate? If not, what changes are necessary? Comment the results.

Lax-Wendroff is expected to be unstable, since the given example has a Courant number of 0.75, when the stability limit of LW is $\sqrt{\frac{1}{3}} = 0.57735$

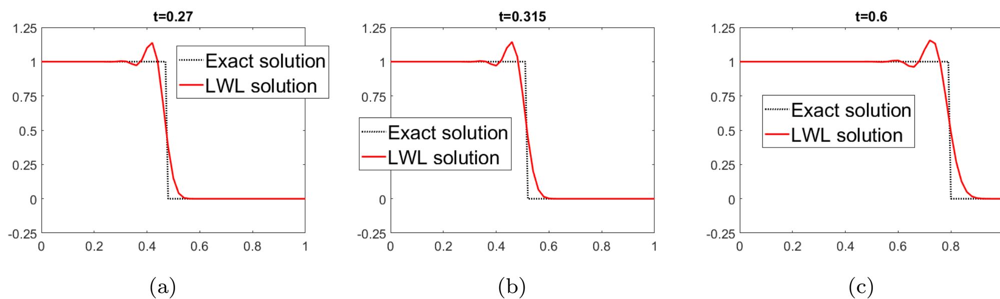


(a) (b) (c)

Figure 2: Lax-Wendroff for Courant Number 0.75

The way to improve Lax-Wendroff's stability range is to use the lumped mass matrix. This is done adding up all of the terms in the rows of the consistent mass matrix, and placing the contribution in the diagonal.

When running the code using LX with lumped mass matrix, the results are expected to be stable.



(a) (b) (c)

Figure 3: Lax-Wendroff with lumped mass matrix for Courant Number 0.75

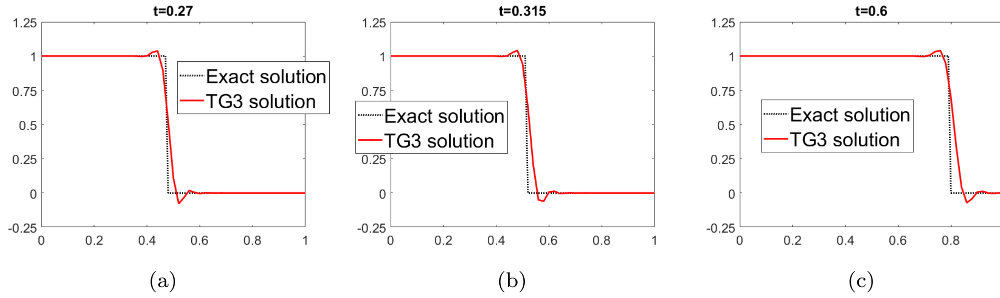
–Solve the problem using the third-order explicit Taylor-Galerkin method. Comment the results.

TG3 is implemented as follows:

```
case 5 %Taylor-Galerkin 3
    A= M + K*(a*a*dt*dt)/6;
    B=-a*dt*C-K*(a*a*dt*dt)/2;
    methodName = 'TG3';
```

Figure 4: Implementation of Taylor-Galerkin 3

The results of running the simulations using TG3 are shown next:



(a) (b) (c)

Figure 5: Taylor-Galerkin third order

Conclusions. Part 1

The implementation of all of this schemes for solving the problem of a steep front transport is not as complicated, due to the lack of boundary and source terms. It was needed only the calculation of the M, C, and K matrices; and then they should be multiplied by the right coefficients, as shown in figure 4.

From the schemes studied Crank-Nicholson is the only one which does not require to fulfill a stability condition and it is a second order method. Lax-Wendroff is also second order, it needs to fulfill the Courant Number lower than $1/\sqrt{3}$, but it has the advantage of being explicit. One can compare figures 1 and 3. Observa how the lumped LW produces less oscillations than the Crank-Nicholson method. This is due to the fact that ...

Finally, Taylor-Galerkin 3, also explicit but 3rd order accurate, is the one that performs better. It has a stability range of Courant$\leq$ 1. In figure 5 one can see how only a small fluctuation near the front are present, and the solution is in general clean away from the front.

Part 2. On Nonlinear Hyperbolic Problems:

The code provided is capable of computing the solution of Burgers equation:

$$\begin{cases} u_t + f_x(u) = 0 \\ u(x,0) = u_0(x) \end{cases}$$

However, it uses a parameter ϵ to smooth, that is, the equation really simulated is:

$$u_t + uu_x = 0 \rightarrow u_t^\epsilon u^\epsilon u_x^\epsilon = \epsilon u_{xx}^\epsilon$$

The first method, Forward Euler, is totally explicit and it can be solved directly, as long as the timestep allows it. For the Backward Euler, an implicit method there are some ways to deal with the implicitness. The first one is Picard method.

Picard method is readily implemented and it consists on solving the following problem:

$$\underbrace{(\boldsymbol{M} + \Delta t(\boldsymbol{C}(\boldsymbol{U}^{n+1}) + \epsilon \boldsymbol{K})}_{\boldsymbol{A}(\boldsymbol{U}^{n+1})} \boldsymbol{U}^{n+1} = \boldsymbol{M}\boldsymbol{U}^n$$

The initial solution is the solution at the previous timestep, and one should iterate until convergence:

$$\boldsymbol{U}_{k+1}^{n+1} = \boldsymbol{A}^{-1}(\boldsymbol{U}_k^{n+1})(\boldsymbol{M}\boldsymbol{U}^n)$$

For this exercise, one is to implement the Newton-Raphson scheme for solving the implicitness. This is done by first calculation a residual $\boldsymbol{f}(\boldsymbol{U}^{n+1}) = \boldsymbol{0}$ with:

$$\boldsymbol{f}(\boldsymbol{U}) = (\boldsymbol{M} + \Delta t \boldsymbol{C}(\boldsymbol{U}) + \epsilon \Delta t \boldsymbol{K})\boldsymbol{U} - \boldsymbol{M}\boldsymbol{U}^n$$

One should use as initial guess the iteration of the last timestep, and iterate until convergence:

$$\boldsymbol{U}_{k+1}^{n+1} = \boldsymbol{U}_k^{n+1} - \boldsymbol{J}^{-1}(\boldsymbol{U}_k^{n+1})\boldsymbol{f}(\boldsymbol{U}_k^{n+1})$$

Where:

$$\boldsymbol{J} = \frac{d\boldsymbol{f}}{d\boldsymbol{U}} = (\boldsymbol{M} + \Delta t \boldsymbol{C}(\boldsymbol{U}) + \epsilon \Delta t \boldsymbol{K}) + (\Delta t \boldsymbol{C})$$

Since $(d\boldsymbol{C}/d\boldsymbol{U})\boldsymbol{U} = \boldsymbol{C}$

The implementation in the code is shown next:

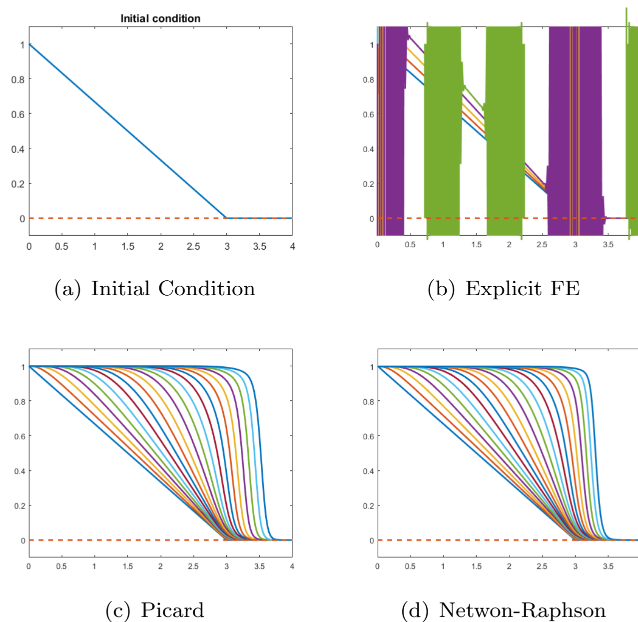
```
for n = 1:nTimeSteps
    %fprintf('\nTime step %d\n', n);
    bccd = [uxa; uxb];
    U0 = U(:,n);
    U1=U0;
    error_U = 1; k = 0;
    while (error_U > 0.5e-5) && k < 20
        C = computeConvectionMatrix(X,T,U(:,n));
        A = M + At*C + At*E*K;
        f=(A*U1) - (M*U(:,n));
        df = A + (At*C);
        U1=U1-(df\f);
        error_U = norm(U1-U0)/norm(U1);
        fprintf('\t Iteration %d, error_U=%e\n',k,error_U);
        U0 = U1;
        k = k+1;
    end
    U(:,n+1) = U1;
```

Figure 6: Newton-Raphson implementation

Testing the code

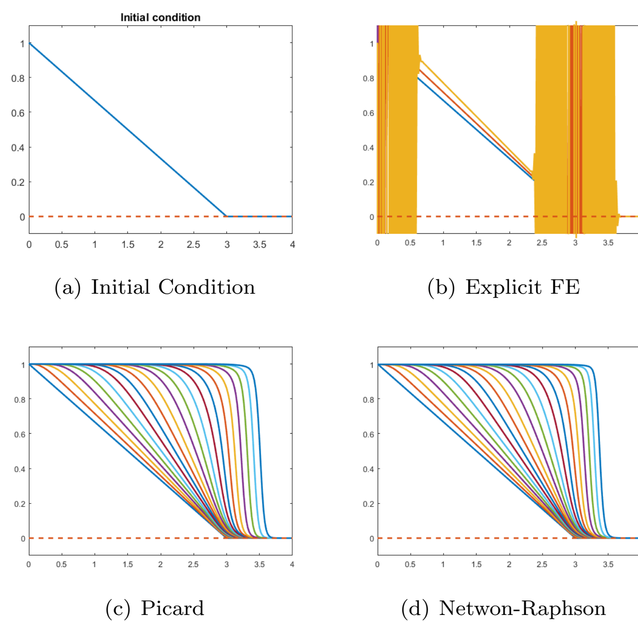
when running the code under different initial conditions, these are the results obtained:

It was proven that the two implicit methods can perform well under condition where the explicit solver has blown up. The NR converges in around 3 iterations to the solution.



(a) Initial Condition (b) Explicit FE (c) Picard (d) Netwon-Raphson

Figure 7: Test 1, $\Delta t = 0.1; \epsilon = 0.01$



(a) Initial Condition (b) Explicit FE (c) Picard (d) Netwon-Raphson

Figure 8: Test 2, $\Delta t = 0.05; \epsilon = 0.01$