Report of the 4th CFD Project

(CFD4012P04)

Submitted to: Dr. M. Pourbagian

Submitted by: Arshia Saffari

Spring of 2023

Contents

Problem:

Consider 1D wave equation:

$$\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial u}{\partial x} = 0$$

The boundary conditions are as follows:

$$u_0(x) = \frac{1}{2}(1 + \tanh[250(x - 20)]), \quad 0 \leq x \leq 40$$

1. Create a network with 41 nodes ($\Delta x = 1$), $\Delta t = 1$ and solve the equation in times $t = 1, t = 5, t = 10$ seconds using three methods:
   I. Upwind
   II. Lax
   III. Lax-Wendorff

   Then, compare the results with each other and analytical result.

2. Solve the last part using $\Delta t = 0.5$ and compare the results.

Analytical solution:

The analytical solution of the equation

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0$$

is $u(x,t) = u_0(x - c\,t)$ where c is constant.

$$u(x,t) = u_0\left(x - \frac{1}{2}t\right)$$

Plot is generated using gnuplot.

```
set multiplot
set xrange [0:40]
set yrange [-2:2]
set sample 10000
u0(x) = 0.5*(1.0 + tanh(250.0 * (x-20.0)))
u(x,t)= u0(x-0.5*t)
plot u(x,  0) ls 1 title 'u(x,  0)- initial condition' at 0.95,0.95
plot u(x,  1) ls 2 title 'u(x,  1)- solution at t = 1' at 0.95,0.93
plot u(x,  5) ls 3 title 'u(x,  5)- solution at t = 5' at 0.95,0.91
plot u(x,10) ls 4 title 'u(x,10)- solution at t =10' at 0.95,0.89
```

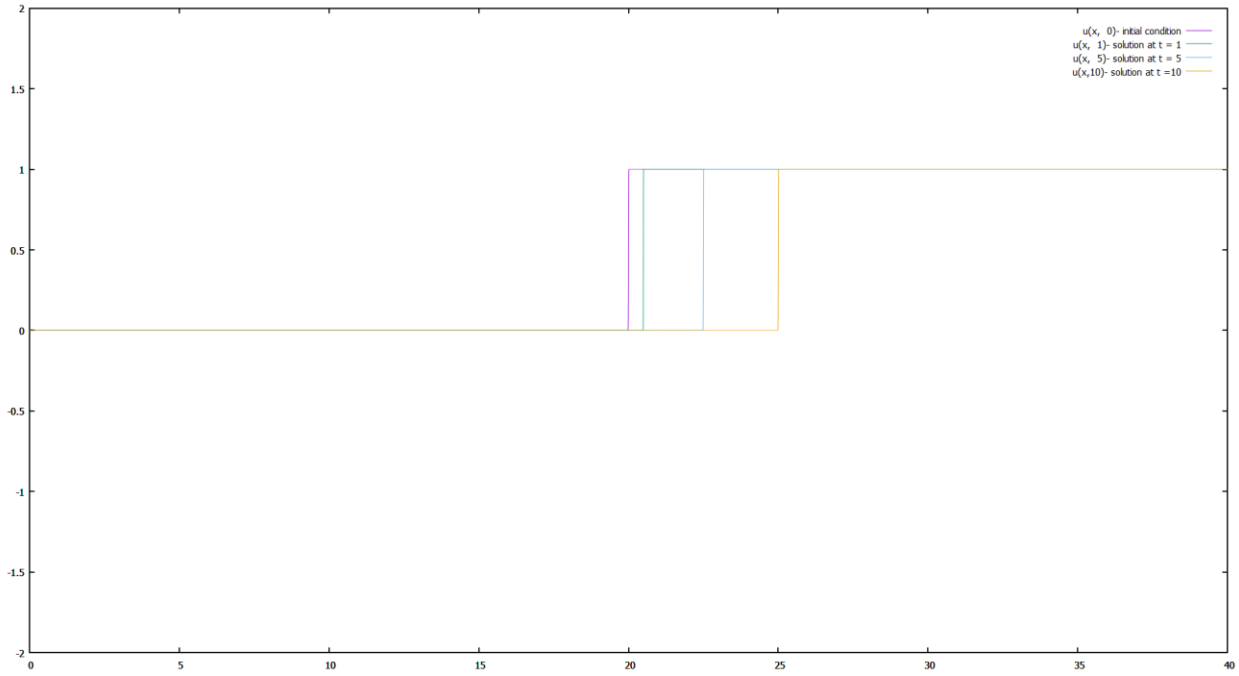

Figure 1. Analytical Solution

Numerical Solution:

Using upwind (FTBS) scheme

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + c\frac{u_i^n - u_{i-1}^n}{\Delta x} = 0$$

$$u_i^{n+1} = u_i^n - c\frac{u_i^n - u_{i-1}^n}{\Delta x}(\Delta t)$$

Stability:

Conditionally stable. $\frac{c\Delta t}{\Delta x} \leq 1$

Using Lax scheme

The same as FTCS but term u_i^n replaces with its' spatial average $\frac{u_{i+1}^n + u_{i-1}^n}{2}$.

$$\frac{u_i^{n+1} - \frac{1}{2}(u_{i+1}^n + u_{i-1}^n)}{\Delta t} + c\frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} = 0$$

$$u_i^{n+1} = \frac{1}{2}(u_{i+1}^n + u_{i-1}^n) - c(\Delta t)\frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x}$$

Stability:

Conditionally stable. $\frac{c\Delta t}{\Delta x} \leq 1$

Using Lax-Wendorff

Combining Tylor series with respect to time and the equation with its' derivatives, u_i^{n+1} can be calculated as illustrated below.

$$u_i^{n+1} = u_i^n + c\Delta t\frac{\partial u}{\partial t}_{x=i} + c\frac{(\Delta t)^2}{2!}\frac{\partial^2 u}{\partial t^2}_{x=i} + \frac{(\Delta t)^3}{3!}\frac{\partial^3 u}{\partial t^3}_{x=i} + \text{H.O.T.}, \quad \frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial u}{\partial x} = 0, \text{ its' derivatives}$$

$$u_i^{n+1} = u_i^n - c\Delta t\left(\frac{{u_{i+1}}^n - {u_{i-1}}^n}{2\Delta x}\right) + \frac{c^2(\Delta t)^2}{4}\left(\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{(\Delta x)^2}\right)$$

Stability:

Conditionally stable. $\frac{c\Delta t}{\Delta x} \leq 1$

Code

For better programming practices, the wave equation is presented as a class.

The header file begins with some includes:

```
#pragma once
#include <iostream>
#include <sstream>
#include <vector>
#include <functional>
#include <execution>
#include <algorithm>
#include <unordered_map>
#include <any>
#include <fstream>
#include <omp.h>

#include <chrono>
```

For timing loops easier a timer struct is created. The constructor stores start time inside variable “start” and the deconstructor stores end time in “end” variable. The difference is then calculated and printed to the console. Basically, the timer starts when an instance is created and ends when the object goes out of scope (its lifetime expires).

```
struct Timer {
    std::chrono::time_point<std::chrono::high_resolution_clock> start, end;
    std::chrono::duration<float> duration;
    std::string message;

    Timer(const std::string& msg) : message(msg) {
        start = std::chrono::high_resolution_clock::now();
    }

    ~Timer() {
        end = std::chrono::high_resolution_clock::now();
        duration = end - start;
        float ms = duration.count() * 1000.0f;
        std::cout << message << " took " << ms << "ms\n";
    }
};
```

To code easier, the class has a default constructor but it should be noted that setters should be called in specific order if the default constructor is used.

Constructors:

```
WaveEquation1D();

WaveEquation1D(std::pair<double, double> xLimits, size_t nodeCounts
        , double deltaT, double deltaX, double c, std::function<double(double)>
initialConditionFunction);
```

The class has private members and attributes:

```
    std::pair<double, double> m_xLimits;
```

```
    double m_deltaX;
    double m_deltaT;
    double C;

    std::function<double(double)> m_initialConditionFunctionOfX;
    double m_leftBoundaryValue;
    double m_rightBoundaryValue;
    size_t m_nodeCount;

    std::vector<double> m_uNextValues;
    std::vector<double> m_uPrevValues;
    std::vector<double> m_uCurrValues;
    std::vector<double> m_xValues;
    std::vector<double> m_tValues;

    size_t m_currentTimeStep = 0;

    //double (*m_initialCondition)(double _x);


    // Allocates enough memory for all vectors
    void allocateMemory();
    // Sets m_xValues with proper x values
    void discretizeDomain();
    // Sets m_uCurrValues.at(i) with m_initialConditionFunctionOfX(m_xValues.at(i))
    void setInitialValues();

    // Unorderes map to store all schemes.
    //The key is method names as strings.
    //The value is a function taking current u values returning next u values.
    std::unordered_map<std::string, std::function<std::vector<double>(std::vector<double>)> >
m_schemes;


    //Upwind scheme. Takes Current u values and returns next u values.
    inline std::vector<double> scheme_upwind(const std::vector<double>);
    //Lax scheme. Takes Current u values and returns next u values.
    inline std::vector<double> scheme_lax(const std::vector<double>);
    //Lax-Wendroff scheme. Takes Current u values and returns next u values.
    inline std::vector<double> scheme_lax_wendroff(const std::vector<double>);

    inline std::vector<double> scheme_leapfrog(const std::vector<double>);
```

Variable "`m_xLimits`" stores where the computational domain starts and ends.

Variable "`m_deltaT`" stores timestep.

Variable "`m_deltaX`" stores spatial step in x direction.

Variable "c" stores wave speed.

Variable "`m_initialConditionFunctionOfX`" stores the initial condition as a function of x.

Variables "`m_uNextValues`", "`m_uPrevValues`", "`m_uCurrValues`" and "`m_xValues`" store values of $u_i^{n+1}, u_i^n, u_i^{n-1}$ and x_i for all i values respectively. n is the current time step.

Variable "`m_currentTimeStep`" stores the current time step.

Method "`void allocateMemory()`" creates vectors of size node count and initializes them with zero.

Method "`void discretizeDomain()`" fills the vector "`m_xValues`".

Method "`void setInitialValues()`" fills "`m_uCurrValues`" before the first iteration (timestep n = 0).

Variable "`std::unordered_map<std::string, std::function<std::vector<double>(std::vector<double>)> > m_schemes`" stores scheme names as the key paired with numerical schemes as the value. This approach makes it possible to implement more methods easier in code and even add and run more schemes in runtime. Although the required methods to add schemes are not implemented yet in this project.

Methods “`inline std::vector<double> scheme_upwind(const std::vector<double>)`”, “`inline std::vector<double> scheme_lax(const std::vector<double>)`”,” `inline std::vector<double> scheme_lax_wendroff(const std::vector<double>)`”and” `inline std::vector<double> scheme_leapfrog(const std::vector<double>)`” are numerical methods that calculate u values of the next timestep for middle nodes.

There are also accessors for these variables.

```
//Accessors

    double getValue(double _x, double _t);
    // Returns std::pair object with firsr = x_min and second = x_max
    std::pair<double, double> getXLimits() const;
    // Returns total node count
    size_t getNodeCount() const;
    // Returns delta t (time step)
    double getDeltaT() const;
    // Returns delta x (x step)
    double getDeltaX() const;
    // Returns the left boundary value
    double getLeftBoundaryValue() const;
    // Returns the right boundary value
    double getRightBoundaryValue() const;
    // Returns wave speed c (coeff. of partial(u)/partial(x))
    double getC() const;
    // Returns initial condition as a function of x
    std::function<double(double)> getInitialConditionFunctionOfX() const;
    // Saves u(x,t) for all x values at current time step in proper format to be plotted by
gnuplot
    void saveToFileGnuplot(const std::string&);

    //Set as std::make_pair(x_min, x_max)
    void setXLimits(std::pair<double, double>);

    void setRightBoundaryValue(double);

    void setLeftBoundaryValue(double);
    //Set node count
    void setNodeCount(size_t);
    //Set delta t (time step)
    void setDeltaT(double);
    //Set delta x (x step)
    void setDeltaX(double);
    //Set wave speed c (coeff. of partial(u)/partial(x))
    void setC(double);
    //Set initial condition as a function of x
    void setInitialConditionFunctionOfX(std::function<double(double)>);
    //void setInitialCondition(double(*)(double));
```

Code for gnuplot output:

```
void WaveEquation1D::saveToFileGnuplot(const std::string& filename) {
    std::ofstream outputFile(filename);
    if (!outputFile.is_open()) {
        throw std::runtime_error("Unable to open file: " + filename);
    }

    // Write the output to the file
    for (size_t i = 0; i < m_nodeCount; i++) {
        outputFile << m_xValues[i] << ' ' << m_uCurrValues[i] << '\n';
    }

    // Close the file
    outputFile.close();
}
```

Method “`void initializeDefaultSchemes()`” fills the “`m_schemes`” unordered map with schemes stated above.

Methods “” and “” are the most common methods in CFD. Here is the code:

```
void WaveEquation1D::discretizeDomain() {
    //Discretize by x_i = a + i * deltaX
    //This function sets m_deltaX as well as m_xValues
```

```
    m_deltaX = (m_xLimits.second - m_xLimits.first) / (double(m_nodeCount) - 1);

    for (int i = 0; i < m_nodeCount; i++) {
        m_xValues.at(i) = m_xLimits.first + i * m_deltaX;
    }
}

void WaveEquation1D::setInitialValues() {
    if (!m_initialConditionFunctionOfX) {
        throw std::runtime_error("No initial condition function set.");
    }

    for (size_t i = 0; i < m_nodeCount; ++i) {
        double x = m_xValues.at(i);
        m_uCurrValues.at(i) = m_initialConditionFunctionOfX(x);
    }
}
```

Last method is "`void WaveEquation1D::updateU(const size_t, const std::string)`" which does the calculation.

```
void WaveEquation1D::updateU(const size_t _update_time_in_timestep, const std::string
_numerical_scheme) {

    Timer time("updateU()");

    // n_new = n_current + _tSteps
    std::cout << "Looking for scheme: " << _numerical_scheme << std::endl;
    auto schemeIter = m_schemes.find(_numerical_scheme);
    if (schemeIter == m_schemes.end()) {
        throw std::runtime_error("Scheme not found: " + _numerical_scheme);
    }
    else { std::cout << "method formula found\n";}

    for (size_t tSteps = 0; tSteps < _update_time_in_timestep; tSteps++) {
        // Update all nodes between using the specified numerical scheme
        m_uNextValues = schemeIter->second(m_uCurrValues);
        // Update first node (m_uNextValues.at(0)) using left boundary condition
        m_uNextValues.at(0) = m_leftBoundaryValue;
        // Update last node (m_uNextValues.at(m_nodeCount - 1)) using right boundary condition
        m_uNextValues.at(m_nodeCount - 1) = m_rightBoundaryValue;
        // Move data
        std::copy(std::execution::par, m_uCurrValues.begin(), m_uCurrValues.end(),
m_uPrevValues.begin());
        std::copy(std::execution::par, m_uNextValues.begin(), m_uNextValues.end(),
m_uCurrValues.begin());
        m_currentTimeStep++;
    }

};
```

Application:

The application.cpp (renamed `CFD4012P04.cpp`) is an interactive application to solve wave equation problems.

```
#include <iostream>
#include <cmath>
#include "WaveEquation1D.h"
#include <exprtk.hpp>
```

Inside the main loop:

```
// Create WaveEquation1D object
WaveEquation1D waveEquation;
// Loop to accept user input
while (true)
{
```

Input simulation parameters:

```
// Prompt user for simulation parameters
```

```
        std::cout << "Enter simulation parameters (leftBoundaryValue rightBoundaryValue nodeCount
domainStart domainEnd simulationTime timestep wavespeed): \n";
        double leftBoundaryValue, rightBoundaryValue, domainStart, domainEnd, simulationTime,
timeStep, c;
        size_t nodeCount;
        std::cin >> leftBoundaryValue >> rightBoundaryValue >> nodeCount >> domainStart >>
domainEnd >> simulationTime >> timeStep >> c;
```

Input initial condition:

This code takes the function form user as string. Using EprTk library the string is turned into a function.

First white spaces are removed from string input and stored in a variable.

Then the object symbolic_table is created which states what variables are inside the sting as well as what identifier is used for them inside code.

Then the object expression is created taking in the symbolic table.

Using object parser which takes the string expression as well as expression object, the parser is compiled.

By setting x variable as the desired value, the expression method "`exprtk::expression<double>::value()`" calculates the function and returns the output.

Lastly a lambda function is passed using "`std::function<double(double)>`" wrapper to be used as the initial condition function.

```
// Prompt user for initial condition function expression
std::cout << "Enter initial condition function expression (e.g. sin(x), x^2, etc.): ";
// Create function string
std::string expression_string;
// Remove white spaces and store the function entered by user in expression_string .
std::getline(std::cin >> std::ws, expression_string);
// Create the function from string

double x;
exprtk::symbol_table <double> symbol_table;
symbol_table.add_variable("x", x);
symbol_table.add_constants();
exprtk::expression<double> expression;
expression.register_symbol_table(symbol_table);
exprtk::parser<double> parser;
if (!parser.compile(expression_string, expression)) {
    std::cerr << "Error: Failed to compile expression '" << expression_string << "'." <<
std::endl;
        }

std::function<double(double)> initialFunc = [&](double _x) {
     x = _x;
     return expression.value();
};
```

Parameters are set.

```
waveEquation.setInitialConditionFunctionOfX(initialFunc);

// Set simulation parameters
waveEquation.setLeftBoundaryValue(leftBoundaryValue);
waveEquation.setRightBoundaryValue(rightBoundaryValue);
waveEquation.setNodeCount(nodeCount);
waveEquation.setXLimits(std::make_pair(domainStart, domainEnd));
waveEquation.setDeltaT(timeStep);
waveEquation.setC(c);
```

The solver method is specified as user is prompted for another simulation or termination of the program.

```
// Prompt user for numerical scheme and number of time steps
std::cout << "Enter numerical scheme (upwind/leapfrog/lax-wendroff/lax): ";
std::string numericalScheme;
std::cin >> std::ws >> numericalScheme;

// Prompt user for output file name
```

```
std::cout << "Enter output file name (press enter for default name): ";
std::string outputFileName;
std::getline(std::cin >> std::ws, outputFileName);
if (outputFileName.empty()) {
    outputFileName = "output.txt";
}

waveEquation.initializeDefaultSchemes();
// Solve the wave equation
waveEquation.updateU(size_t(simulationTime/waveEquation.getDeltaT()), numericalScheme);

// Output results to file
waveEquation.saveToFileGnuplot(outputFileName);

std::cout << "Simulation completed. Output saved to " << outputFileName << "." << std::endl;

// Prompt user to repeat simulation or exit
std::cout << "Do you want to run another simulation? (Y/N): ";
char repeat;
std::cin >> repeat;
if (repeat != 'Y' && repeat != 'y') {
     brake;
}
```

How to run

The executable file is compiled for both 64bit and 32bit using MSVC compiler and the 64bit version is used to generate results.

Just double click and the simple interactive CLI application starts.

You can also generate different simulation parameters and past all of them at once as illustrated below.

To compile the source code, add ExprTk library path to include path. For MSVC compiler use option “/bigobj”.

Project Answer

$u_0{}^n = 0, u_{40}{}^n = 1$, 41 nodes,xrange $= [0,\ 40], t_{end} = 1, 5, 10, \Delta t = 1, \Delta x = 1, c = 0.5$.

Using text below as input:

```
0.0
1.0
41
0.0
40.0
1
1
0.5
1.0/2.0 * (1 + tanh(250*(x - 20)))
upwind
upwind01_s.txt
y
0.0
1.0
41
0.0
40.0
5
1
0.5
1.0/2.0 * (1 + tanh(250*(x - 20)))
upwind
upwind05_s.txt
y
0.0
1.0
41
0.0
40.0
10
1
0.5
1.0/2.0 * (1 + tanh(250*(x - 20)))
upwind
upwind10_s.txt
y
0.0
1.0
41
0.0
40.0
1
1
0.5
1.0/2.0 * (1 + tanh(250*(x - 20)))
lax
lax01_s.txt
y
0.0
1.0
41
0.0
40.0
5
1
0.5
1.0/2.0 * (1 + tanh(250*(x - 20)))
lax
lax05_s.txt
y
```

```
0.0
1.0
41
0.0
40.0
10
1
0.5
1.0/2.0 * (1 + tanh(250*(x - 20)))
lax
lax10_s.txt
y
0.0
1.0
41
0.0
40.0
1
1
0.5
1.0/2.0 * (1 + tanh(250*(x - 20)))
lax-wendroff
lax-wendroff01_s.txt
y
0.0
1.0
41
0.0
40.0
5
1
0.5
1.0/2.0 * (1 + tanh(250*(x - 20)))
lax-wendroff
lax-wendroff05_s.txt
y
0.0
1.0
41
0.0
40.0
10
1
0.5
1.0/2.0 * (1 + tanh(250*(x - 20)))
lax-wendroff
lax-wendroff10_s.txt
y
```

The minimum calculation time is 0. 1566ms and the maximum time is 0.4725ms.

```
set multiplot

set xrange [0:40]
set yrange [-2:2]
set sample 10000

u0(x) = 0.5*(1.0 + tanh(250.0 * (x-20.0)))
u(x,t)= u0(x-0.5*t)
plot u(x,  0) ls 0 title 'u(x,  0)- initial condition' at 0.95,0.95



plot u(x,  1) ls 1 title 'u(x,  1)- solution at t = 1' at 0.95,0.93
plot 'upwind01_s.txt' w lp ls 2 ti 'upwind 1s'        at 0.95,0.91

plot u(x,  5) ls 3 title 'u(x,  5)- solution at t = 5' at 0.95,0.89
plot 'upwind05_s.txt' w lp ls 4 ti 'upwind 5s'        at 0.95,0.87
```

```
plot u(x,10) ls 5 title 'u(x,10)- solution at t =10' at 0.95,0.85
plot 'upwind10_s.txt' w lp ls 6 ti 'upwind10s'          at 0.95,0.83
```

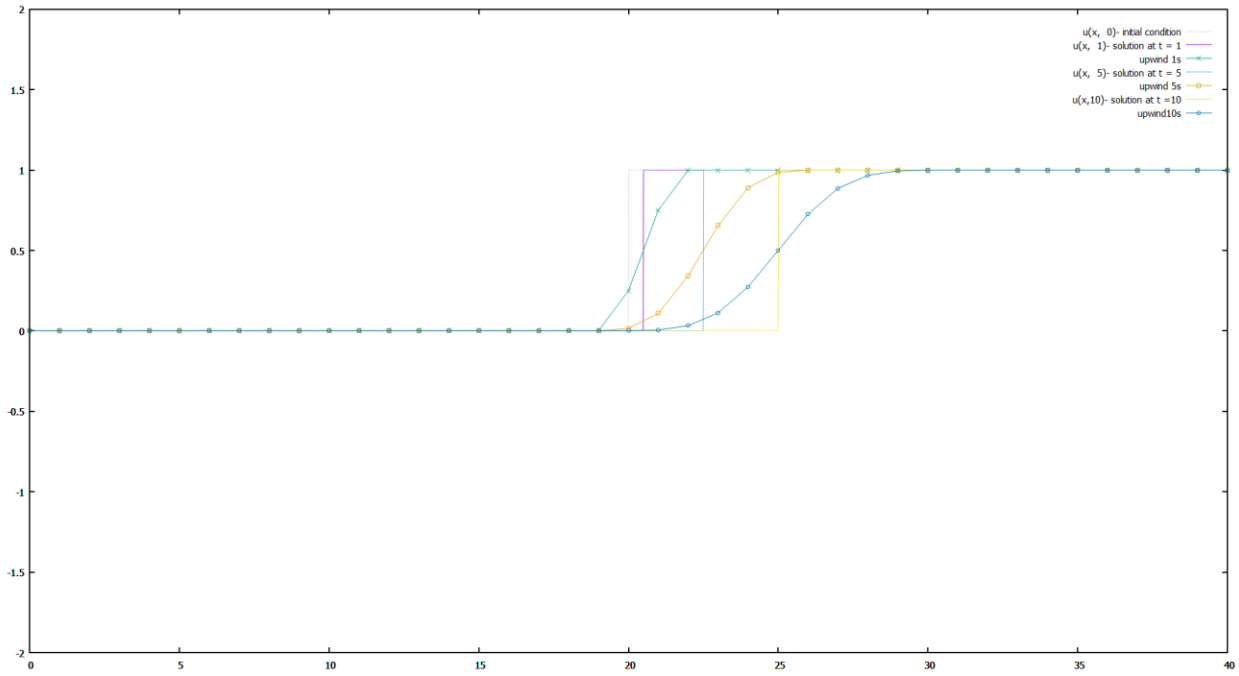

Figure 2. Upwind using 40 nodes delta t = 1

Numerical dissipation error is notable (aka false diffusion).

```
set multiplot

set xrange [0:40]
set yrange [2:-2]
set multiplot

set xrange [0:40]
set yrange [-2:2]
set sample 10000

u0(x) = 0.5*(1.0 + tanh(250.0 * (x-20.0)))
u(x,t)= u0(x-0.5*t)
plot u(x,  0) ls 0 title 'u(x,  0)- initial condition' at 0.95,0.95


plot u(x,  1) ls 1 title 'u(x,  1)- solution at t = 1' at 0.95,0.93
plot 'lax01_s.txt' w lp ls 2 ti 'lax 1s'          at 0.95,0.91

plot u(x,  5) ls 3 title 'u(x,  5)- solution at t = 5' at 0.95,0.89
plot 'lax05_s.txt' w lp ls 4 ti 'lax 5s'          at 0.95,0.87

plot u(x,10) ls 5 title 'u(x,10)- solution at t =10' at 0.95,0.85
plot 'lax10_s.txt' w lp ls 6 ti 'lax10s'          at 0.95,0.83
```

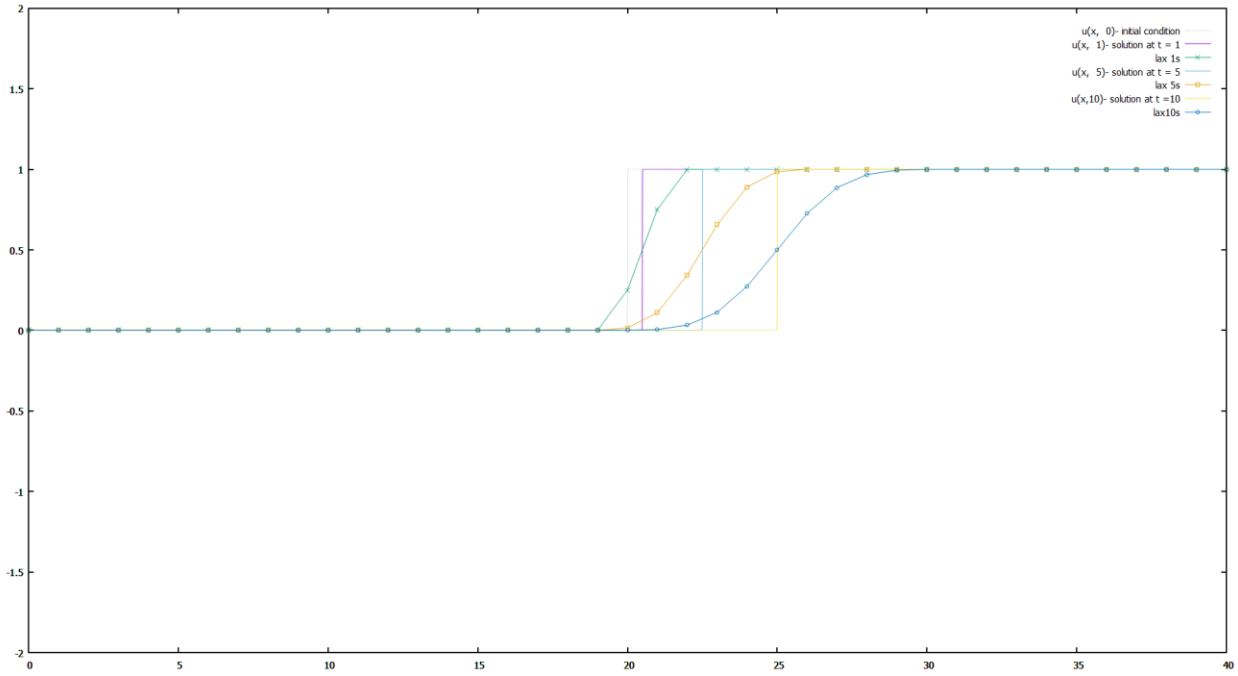

Figure 3. Lax using 40 nodes delta t = 1s

Numerical dissipation error is notable (aka false diffusion).

```
set multiplot

set xrange [0:40]
set yrange [-2:2]
set sample 10000

u0(x) = 0.5*(1.0 + tanh(250.0 * (x-20.0)))
u(x,t)= u0(x-0.5*t)
plot u(x,  0) ls 0 title 'u(x,  0)- initial condition' at 0.95,0.95


plot u(x,  1) ls 1 title 'u(x,  1)- solution at t = 1' at 0.95,0.93
plot 'lax-wendroff01_s.txt' w lp ls 2 ti 'lax-wendroff 1s'         at 0.95,0.91

plot u(x,  5) ls 3 title 'u(x,  5)- solution at t = 5' at 0.95,0.89
plot 'lax-wendroff05_s.txt' w lp ls 4 ti 'lax-wendroff 5s'         at 0.95,0.87

plot u(x,10) ls 5 title 'u(x,10)- solution at t =10' at 0.95,0.85
plot 'lax-wendroff10_s.txt' w lp ls 6 ti 'lax-wendroff10s'         at 0.95,0.83
```

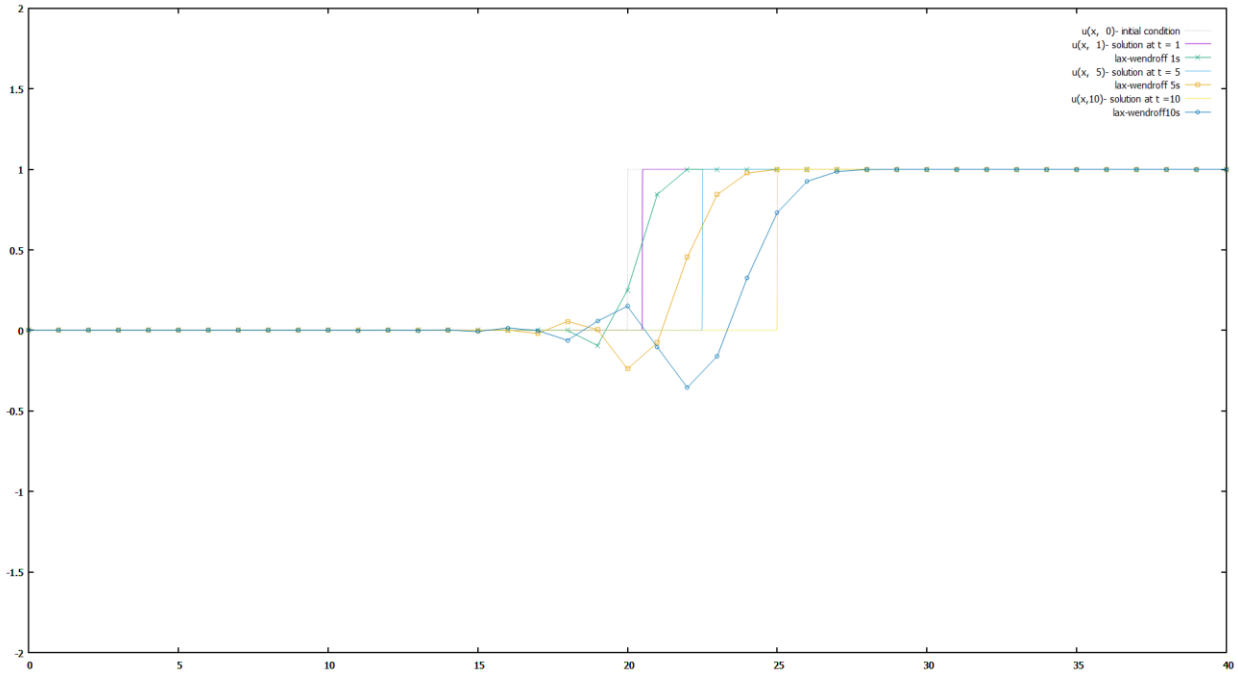

Figure 4. Lax-Wendroff using 40 nodes delta t = 1s.

Numerical dispersion as well as dissipation is noticeable.

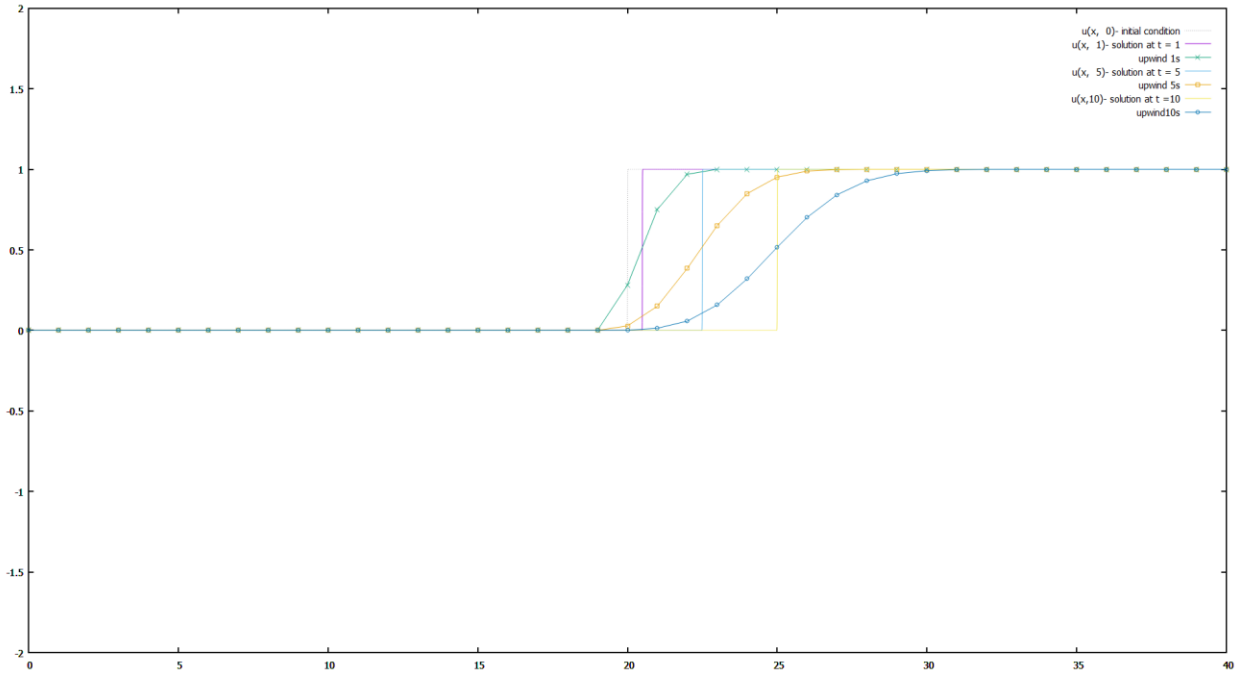

Figure 5. Upwind using 40 nodes delta t = 0.5s

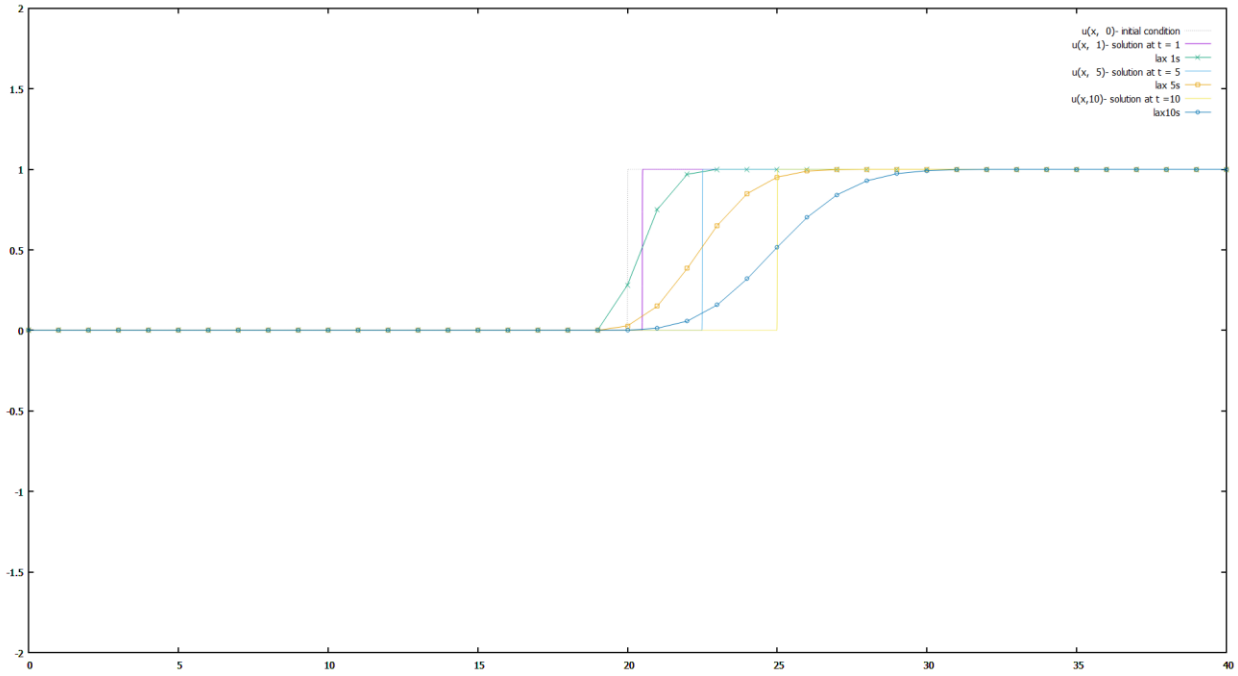

Figure 6. Lax using 40 nodes delta t = 0.5s

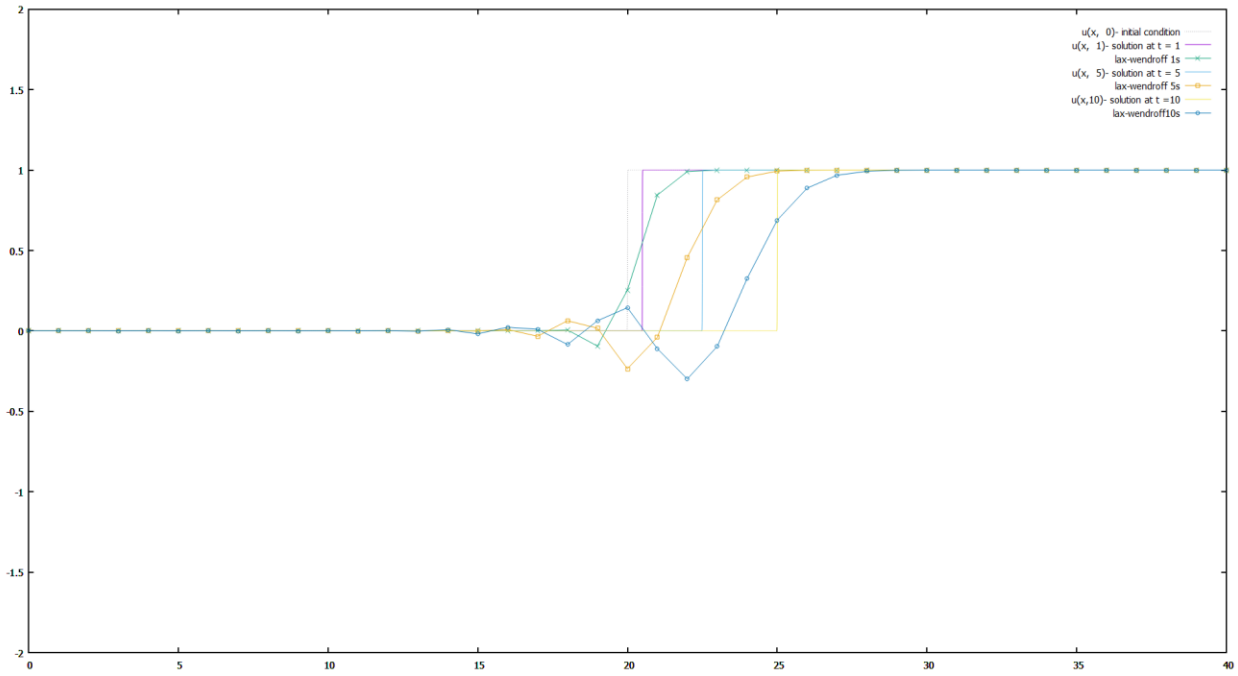

Figure 7. Lax-Wendroff using 40 nodes delta t = 0.5s

Algorithm comparison

Timestep effect on methods

In this section all methods are compared using the same node count but with increasing steps.

Upwind

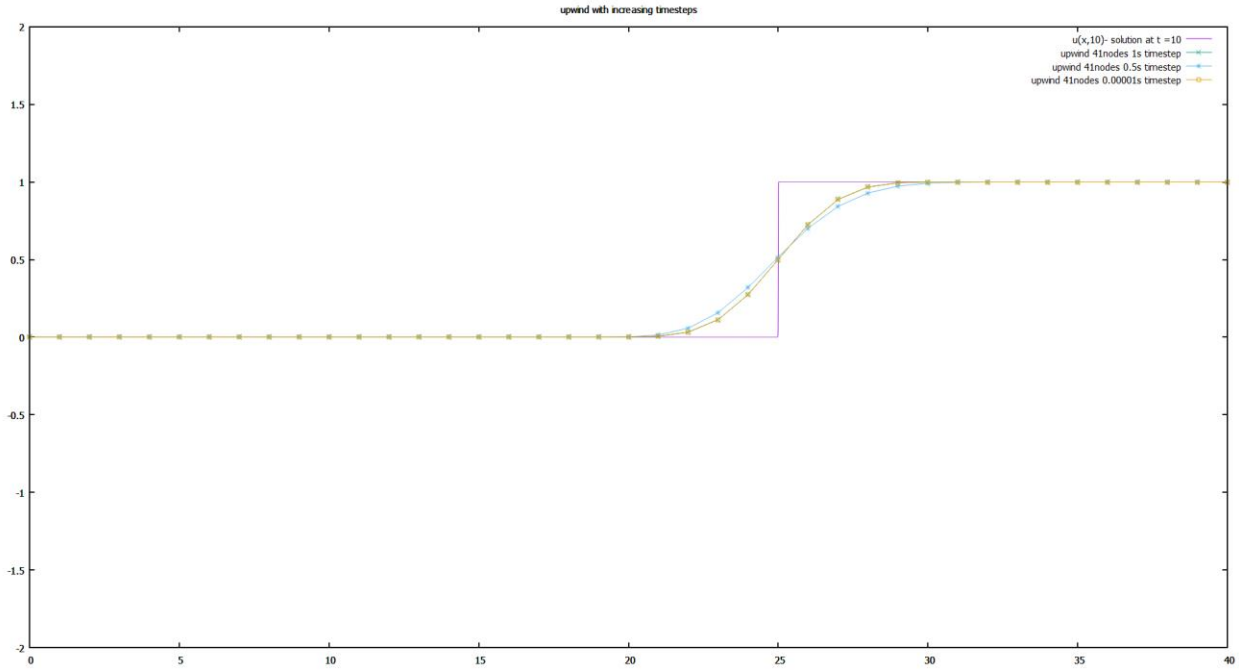

Figure 8. Upwind using 41 nodes. Increasing timesteps

With a huge increase in timestep the error doesn't get much smaller. Not enough to justify the computational cost increase.

Lax

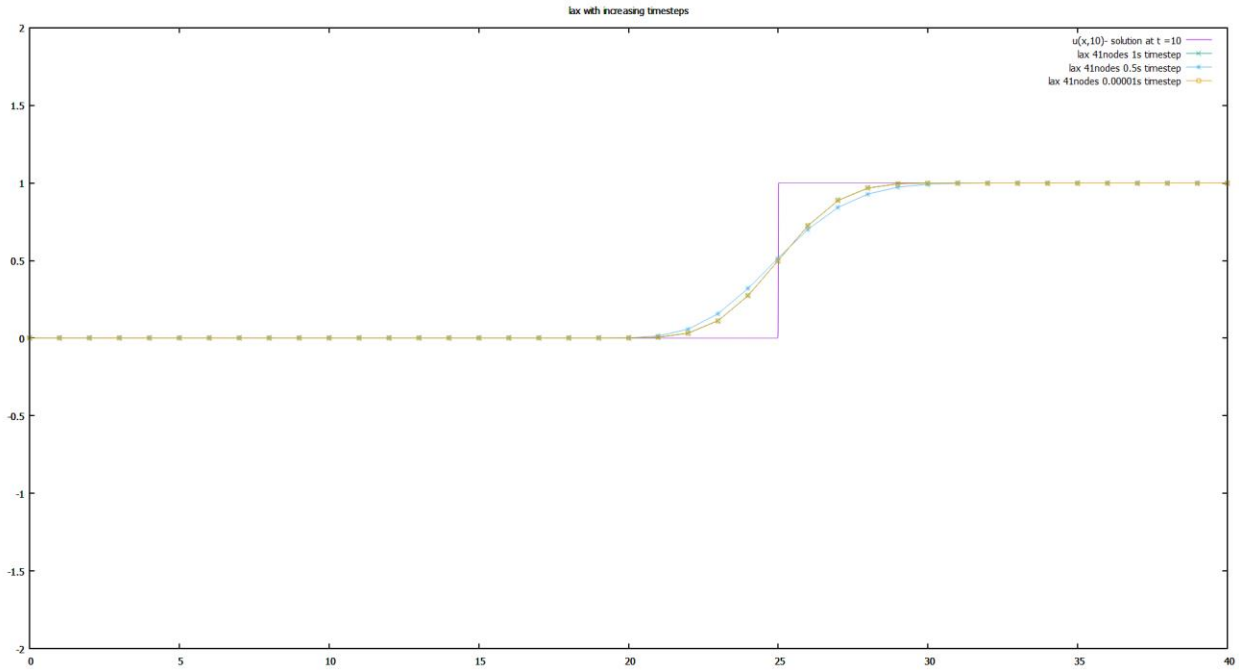

Figure 9. Lax using 41 nodes. Increasing timesteps

With a huge increase in timestep the error doesn't get much smaller. Not enough to justify the computational cost increase.

Lax-Wendroff

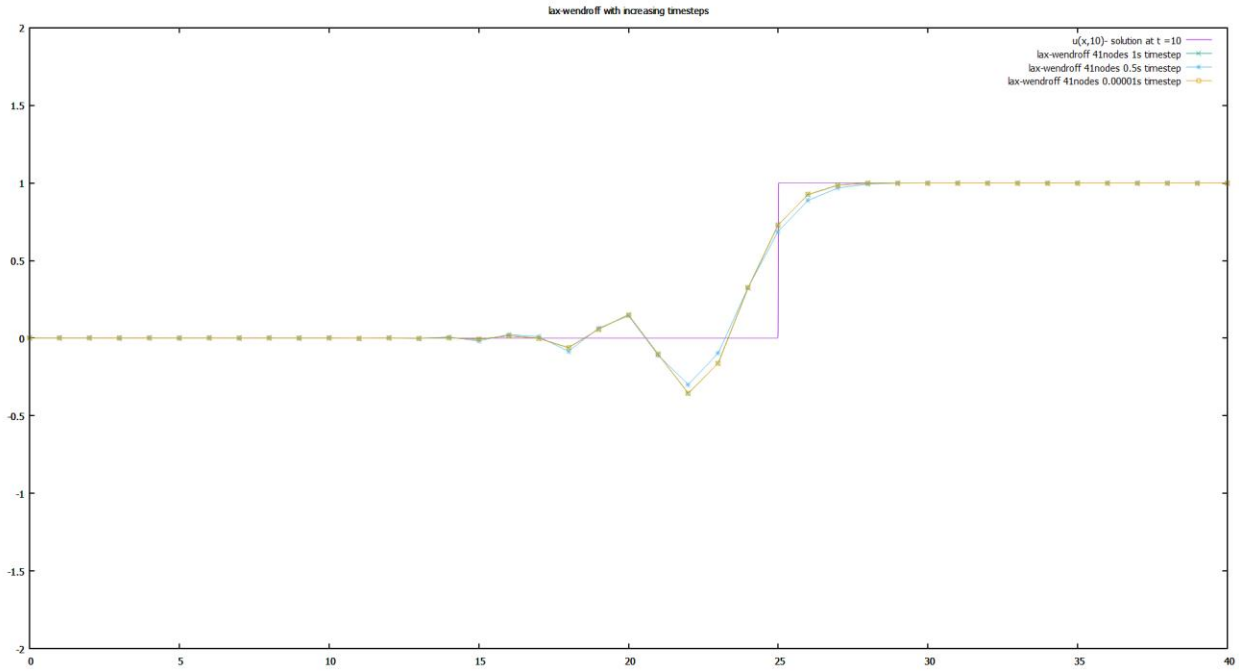

Figure 10. Lax-Wendroff using 41 nodes. Increasing timesteps

With a huge increase in timestep the error doesn't get much smaller. Not enough to justify the computational cost increase. The solution is still very wiggly.

Node count effect on methods

In this section all methods are compared using the same timestep but with increasing x steps.

Just remember the stability condition or answers will not converge.

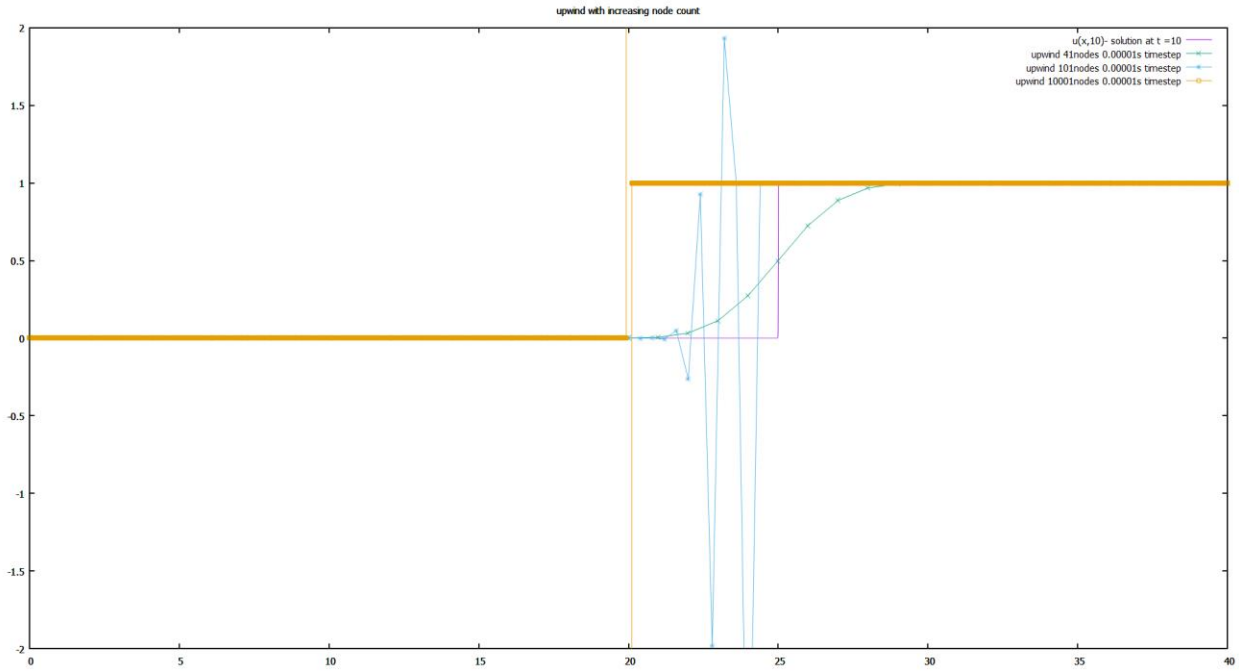

Figure 11. The timestep was 00001s in the code. it's actually 1s timestep.

It's quite interesting that the WRONG solution of 10001 node grid is actually very similar to boundary condition.

Upwind

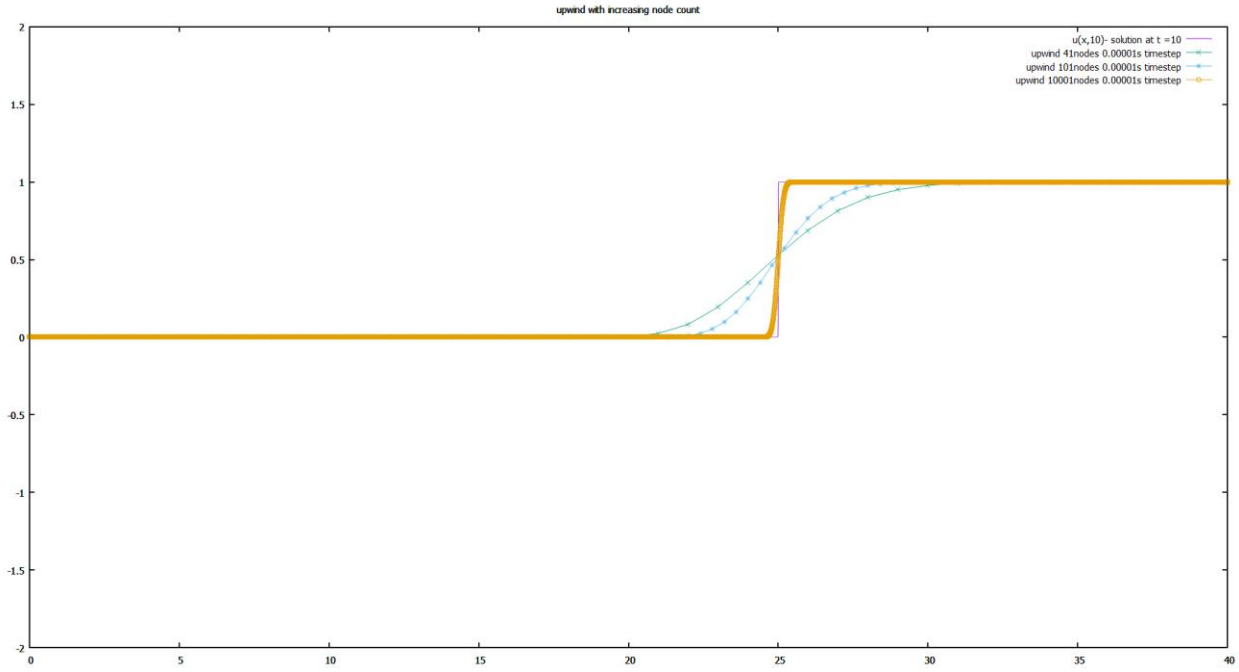

Figure 12. Upwind scheme with increasing node count

The error substantially decreases with increasing node count.

Lax

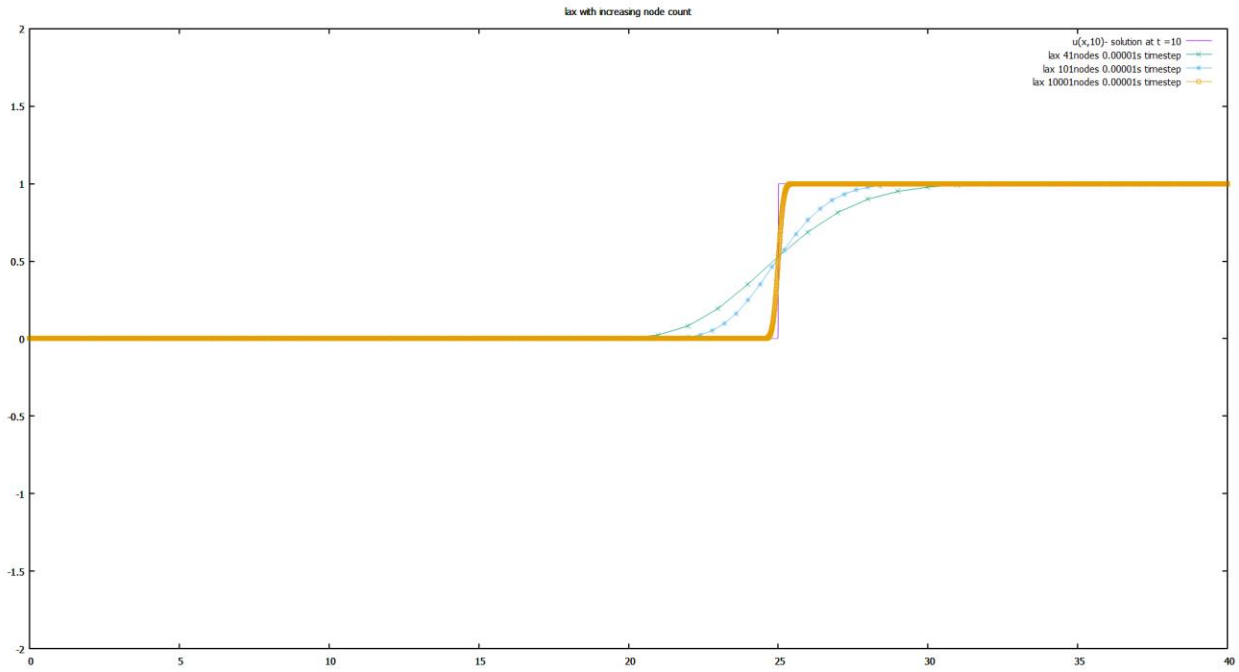

Figure 13. Lax scheme with increasing node count

The error substantially decreases with increasing node count.

Lax-Wendroff

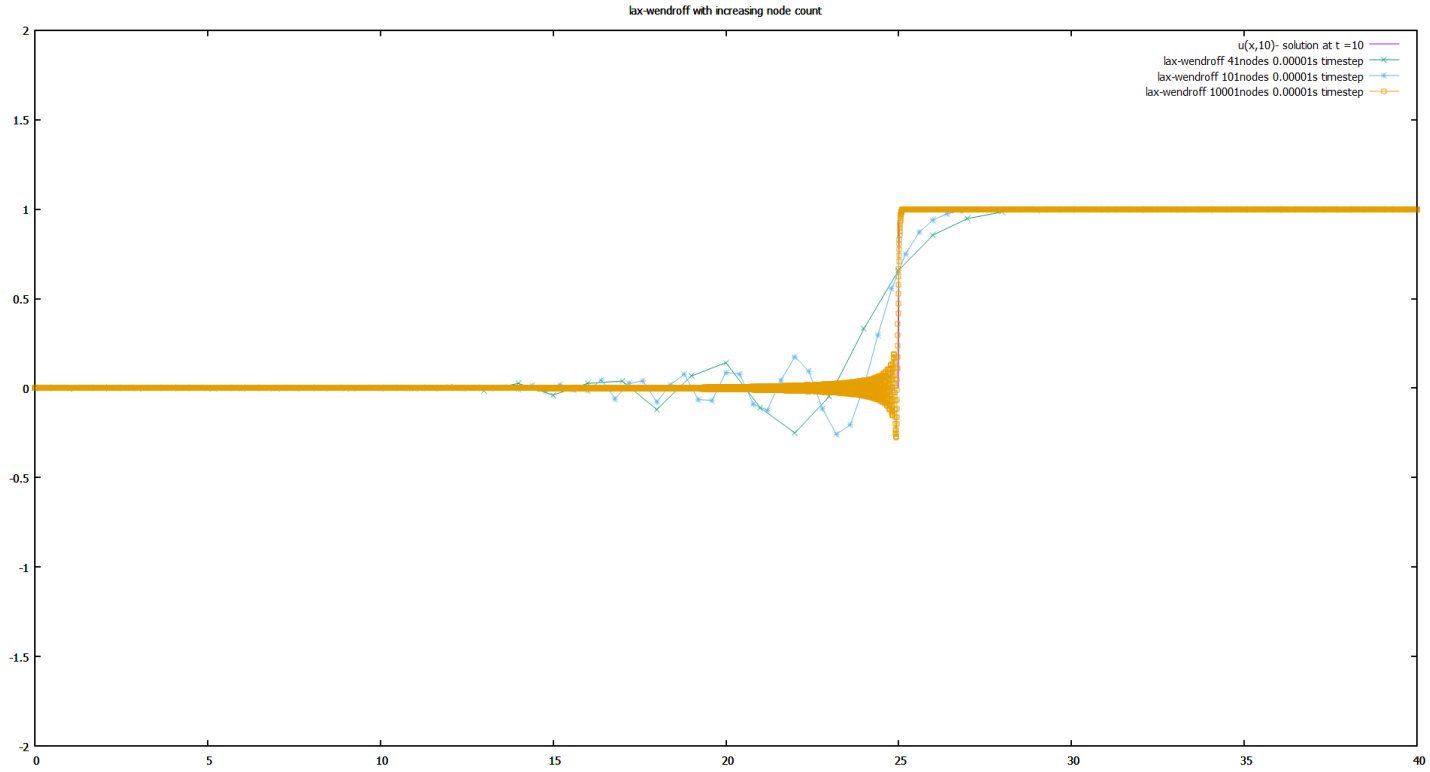

Figure 14. Lax-Wendroff scheme with increasing node count

The error substantially decreases with increasing node count but the wiggle is still there. Although a little lesser in magnitude, the frequency of the wiggles is increased considerably.

Conclusion

1. Increasing timesteps beyond the stability condition is mostly unnecessary since it increases computational cost substantially with small decrease in error. It cannot compensate numerical dispersion.
2. Increasing node count will increase accuracy but also requires an equal increase in timesteps to be stable. It cannot compensate numerical dispersion.
3. If a solution is dispersive, the scheme should be changed and increasing nodes or timesteps will not solve the problem.

Comparing methods:

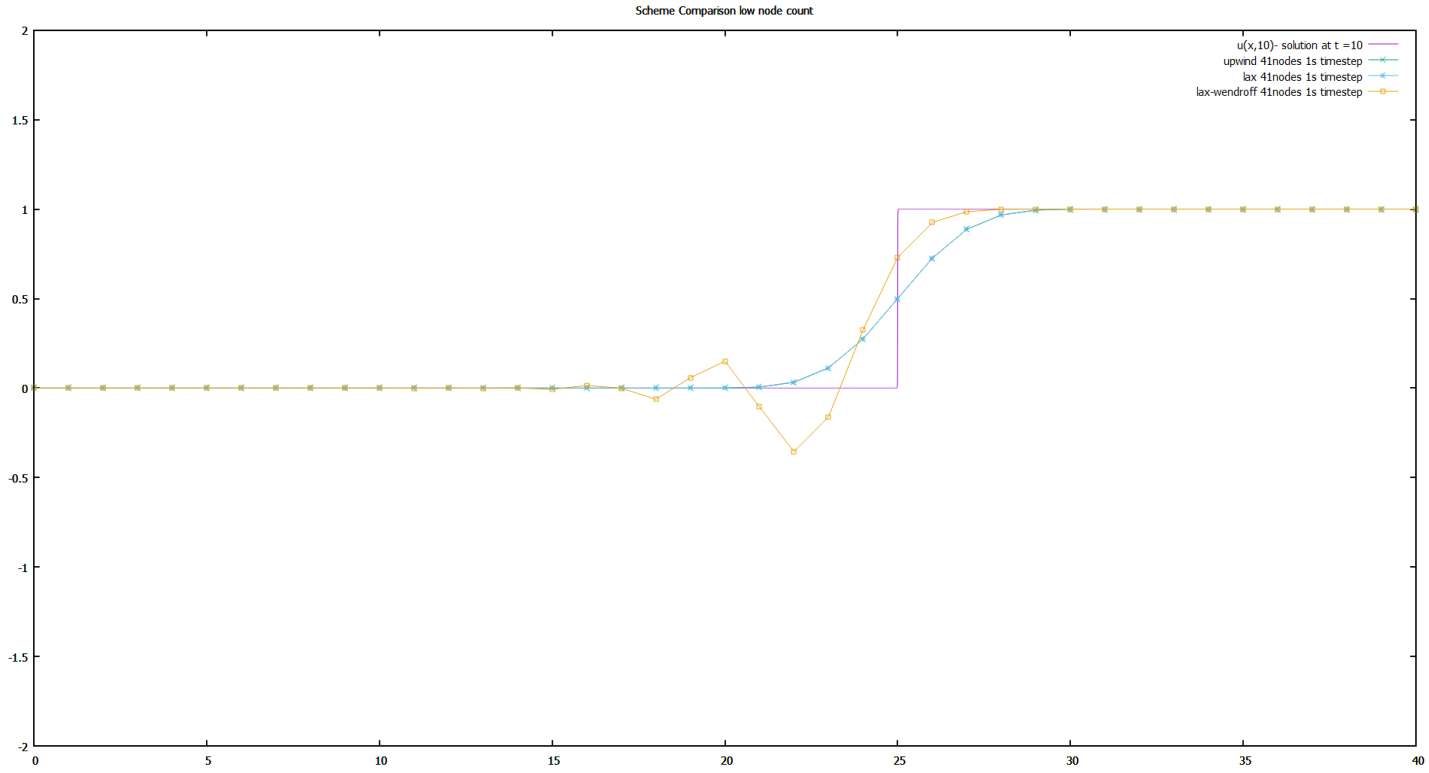

Figure 15. Different schemes at low node count.

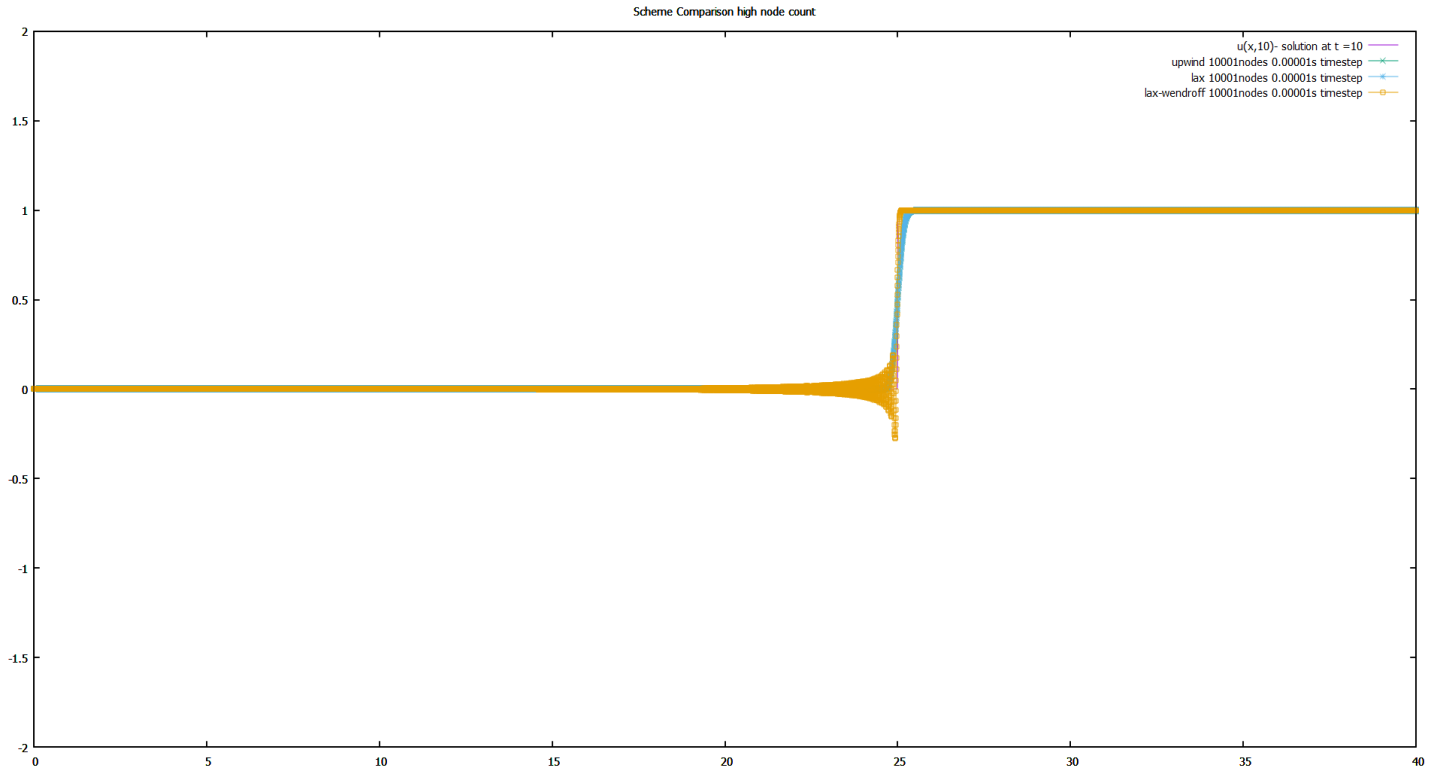

Figure 16. Different schemes at high node count.

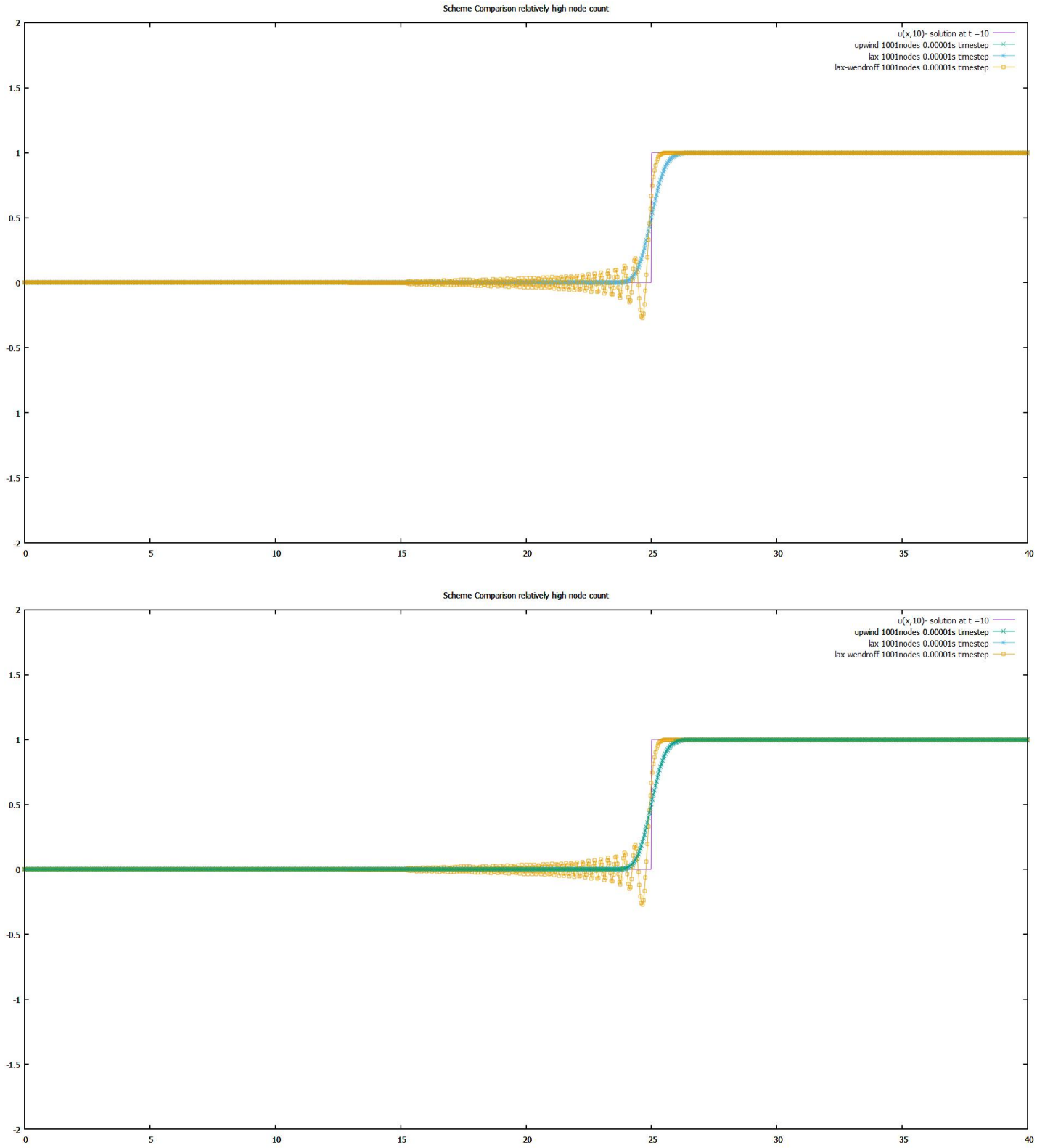

Figure 17. Different schemes at relatively high node count. Upwind replotted in the second graph.

Conclusion

1. Upwind and Lax are basically the same. They are so close that they cover each other so mush so that there are two copies of relatively high node graph showing that thy are actually two lines on top of each other. Even computation time is about the same (1486.43ms vs 1557.65ms respectively).
2. Lax method is dispersive and not suited for this problem.