Report of the 4th CFD Project (CFD4012P04)

Submitted to: Dr. M. Pourbagian

Submitted by: Arshia Saffari

Spring of 2023

Contents

[Problem: 3](#_Toc135953341)

[Analytical solution: 3](#_Toc135953342)

[Numerical Solution: 4](#_Toc135953343)

[Using upwind (FTBS) scheme 4](#_Toc135953344)

[Using Lax scheme 4](#_Toc135953345)

[Using Lax-Wendorff 4](#_Toc135953346)

[Code 5](#_Toc135953347)

[How to run 10](#_Toc135953348)

[Project Answer 11](#_Toc135953349)

[Algorithm comparison 17](#_Toc135953350)

[Timestep effect on methods 17](#_Toc135953351)

[Upwind 17](#_Toc135953352)

[Lax 17](#_Toc135953353)

[Lax-Wendroff 18](#_Toc135953354)

[Node count effect on methods 18](#_Toc135953355)

[Upwind 19](#_Toc135953356)

[Lax 20](#_Toc135953357)

[Lax-Wendroff 20](#_Toc135953358)

[Conclusion 21](#_Toc135953359)

[Comparing methods: 22](#_Toc135953360)

[Conclusion 23](#_Toc135953361)

[Figure 1. Analytical Solution 4](#_Toc135953487)

[Figure 2. Upwind using 40 nodes delta t = 1 13](#_Toc135953488)

[Figure 3. Lax using 40 nodes delta t = 1s 14](#_Toc135953489)

[Figure 4. Lax-Wendroff using 40 nodes delta t = 1s. 15](#_Toc135953490)

[Figure 5. Upwind using 40 nodes delta t = 0.5s 15](#_Toc135953491)

[Figure 6. Lax using 40 nodes delta t = 0.5s 16](#_Toc135953492)

[Figure 7. Lax-Wendroff using 40 nodes delta t = 0.5s 16](#_Toc135953493)

[Figure 8. Upwind using 41 nodes. Increasing timesteps 17](#_Toc135953494)

[Figure 9. Lax using 41 nodes. Increasing timesteps 17](#_Toc135953495)

[Figure 10. Lax-Wendroff using 41 nodes. Increasing timesteps 18](#_Toc135953496)

[Figure 11. The timestep was 00001s in the code. it's actually 1s timestep. 19](#_Toc135953497)

[Figure 12. Upwind scheme with increasing node count 19](#_Toc135953498)

[Figure 13. Lax scheme with increasing node count 20](#_Toc135953499)

[Figure 14. Lax-Wendroff scheme with increasing node count 20](#_Toc135953500)

[Figure 15. Different schemes at low node count. 22](#_Toc135953501)

[Figure 16. Different schemes at high node count. 22](#_Toc135953502)

[Figure 17. Different schemes at relatively high node count. Upwind replotted in the second graph. 23](#_Toc135953503)

# Problem:

Consider 1D wave equation:

The boundary conditions are as follows:

1. Create a network with 41 nodes , and solve the equation in times seconds using three methods:
2. Upwind
3. Lax
4. Lax-Wendorff

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

1. Solve the last part using and compare the results.

## Analytical solution:

The analytical solution of the equation

is where c is constant.

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 |

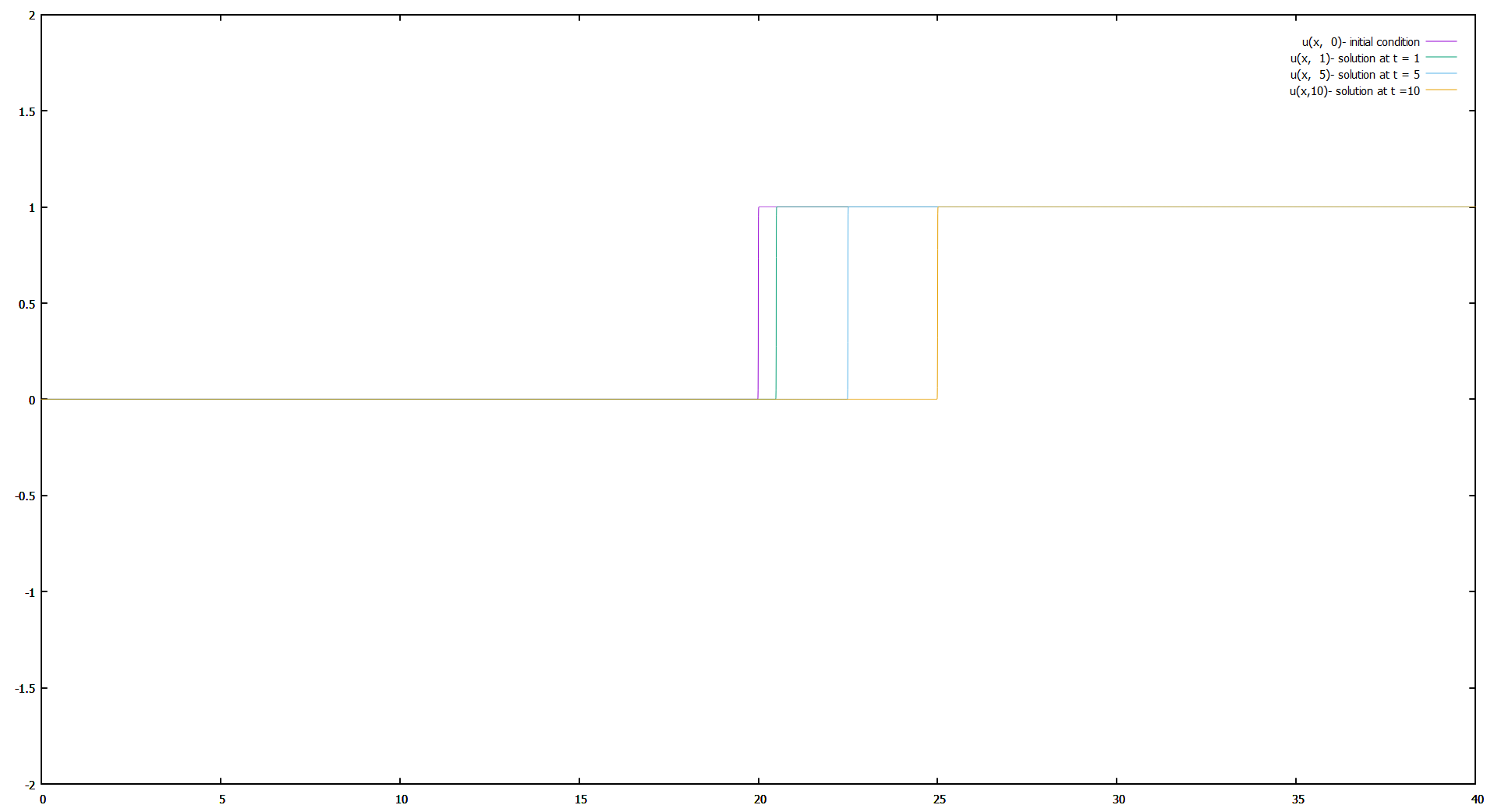


Figure 1. Analytical Solution

## Numerical Solution:

### Using upwind (FTBS) scheme

Stability:

Conditionally stable.

### Using Lax scheme

The same as FTCS but term replaces with its’ spatial average .

Stability:

Conditionally stable.

### Using Lax-Wendorff

Combining Tylor series with respect to time and the equation with its’ derivatives, can be calculated as illustrated below.

Stability:

Conditionally stable.

### 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 and for all values respectively. 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

41 nodes,

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 |

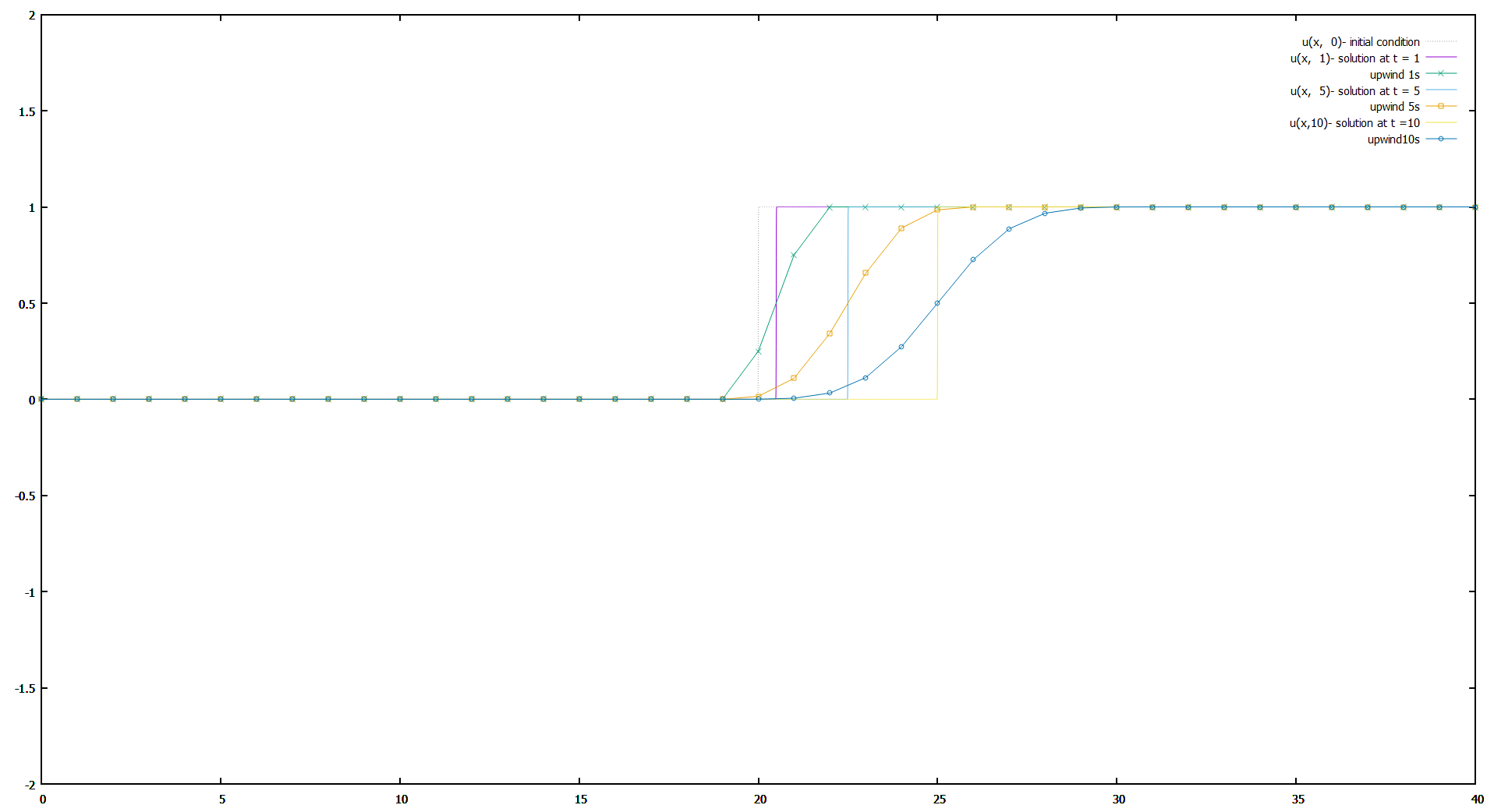


Figure 2. Upwind using 40 nodes delta t = 1

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 |

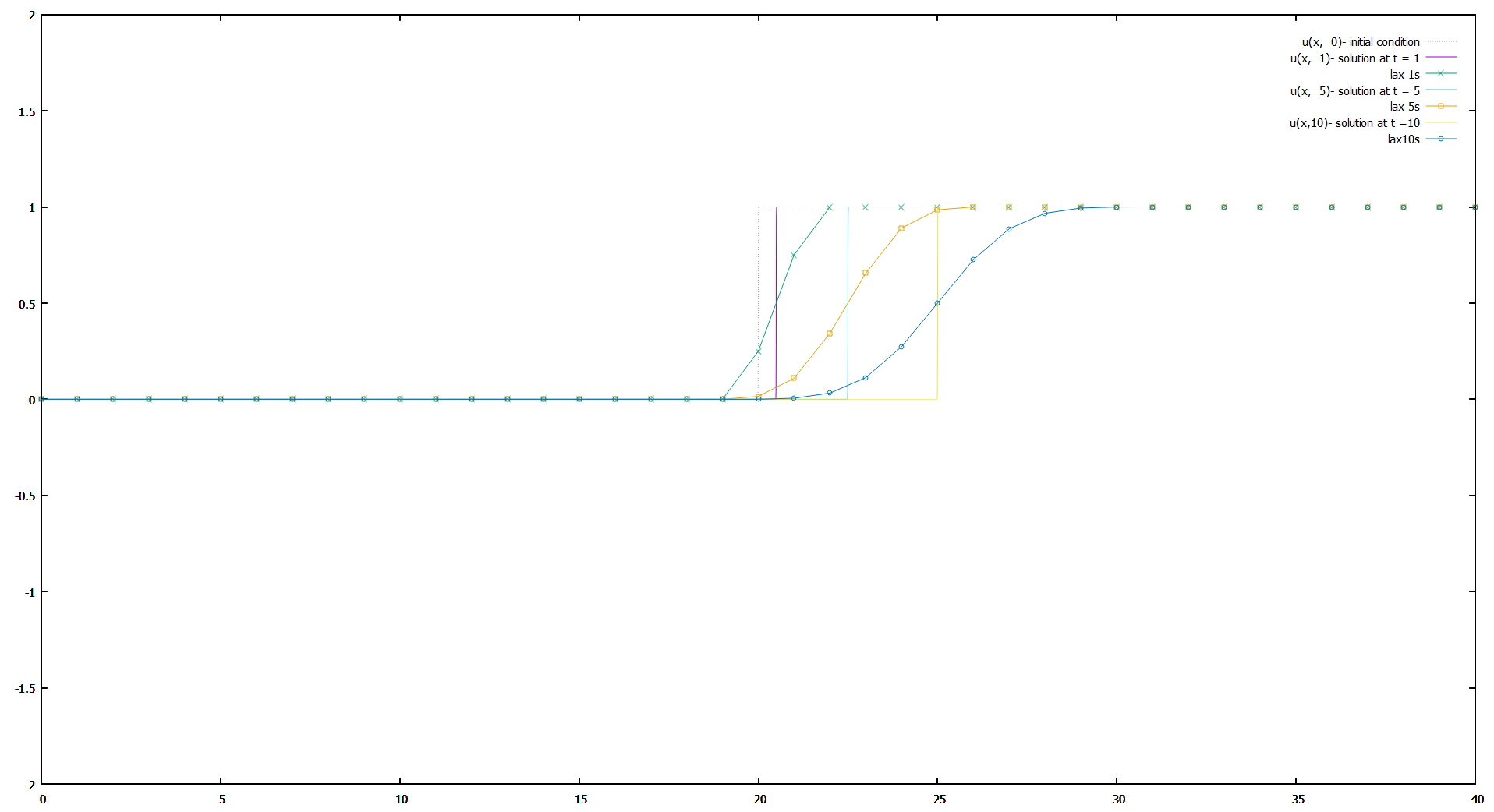


Figure 3. Lax using 40 nodes delta t = 1s

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 |

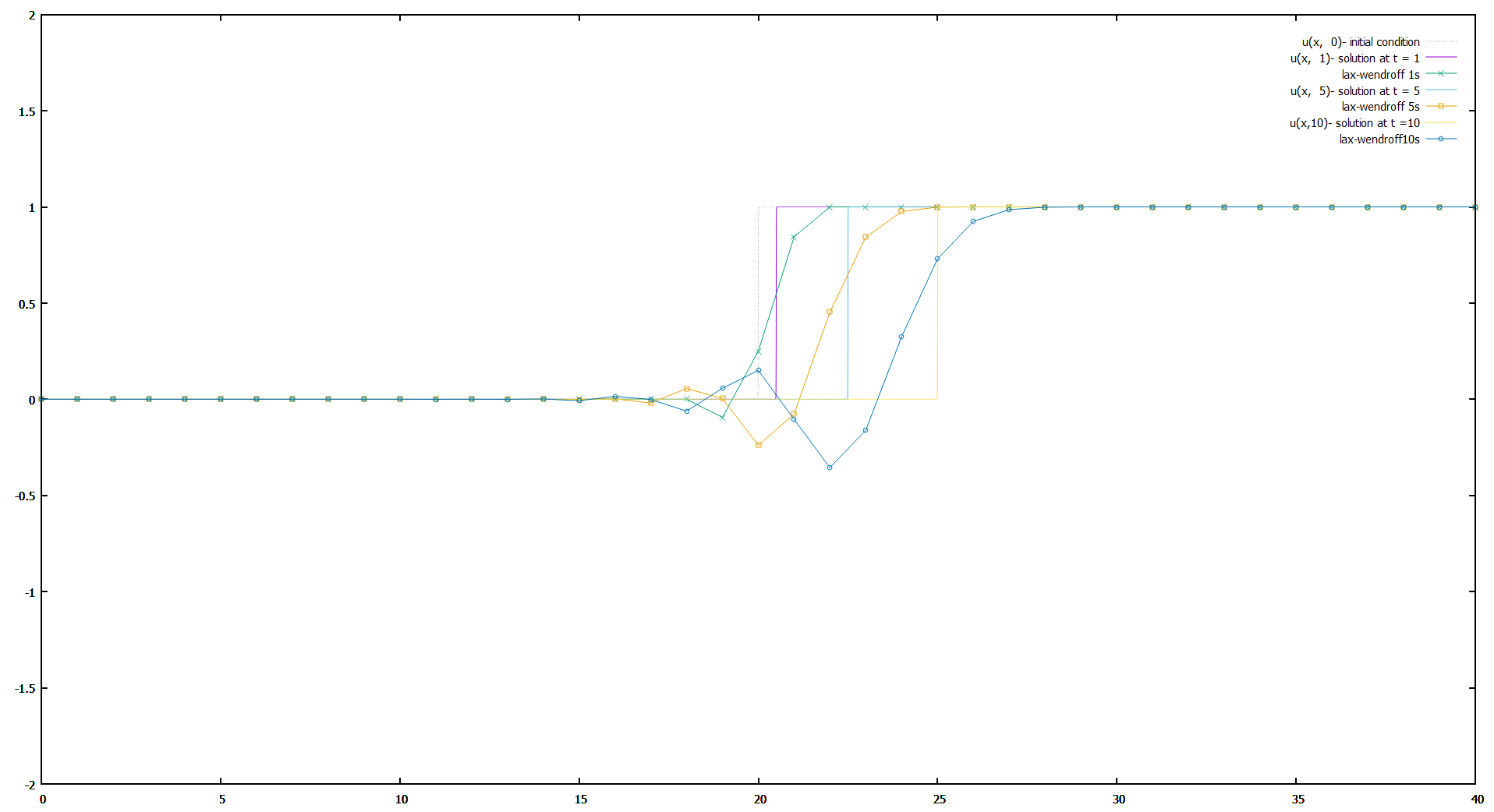


Figure 4. Lax-Wendroff using 40 nodes delta t = 1s.

Numerical dispersion as well as dissipation is noticeable.

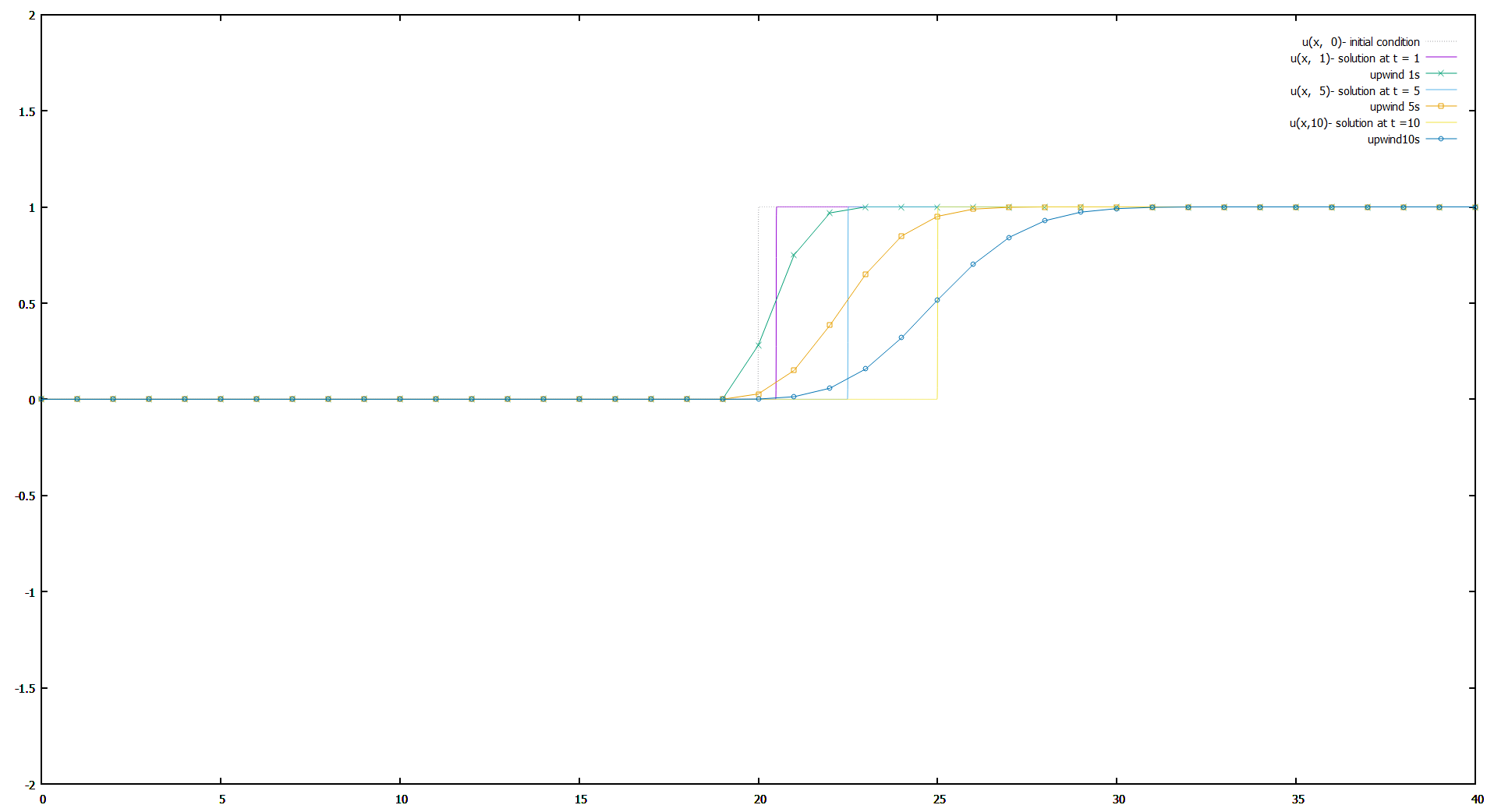


Figure 5. Upwind using 40 nodes delta t = 0.5s

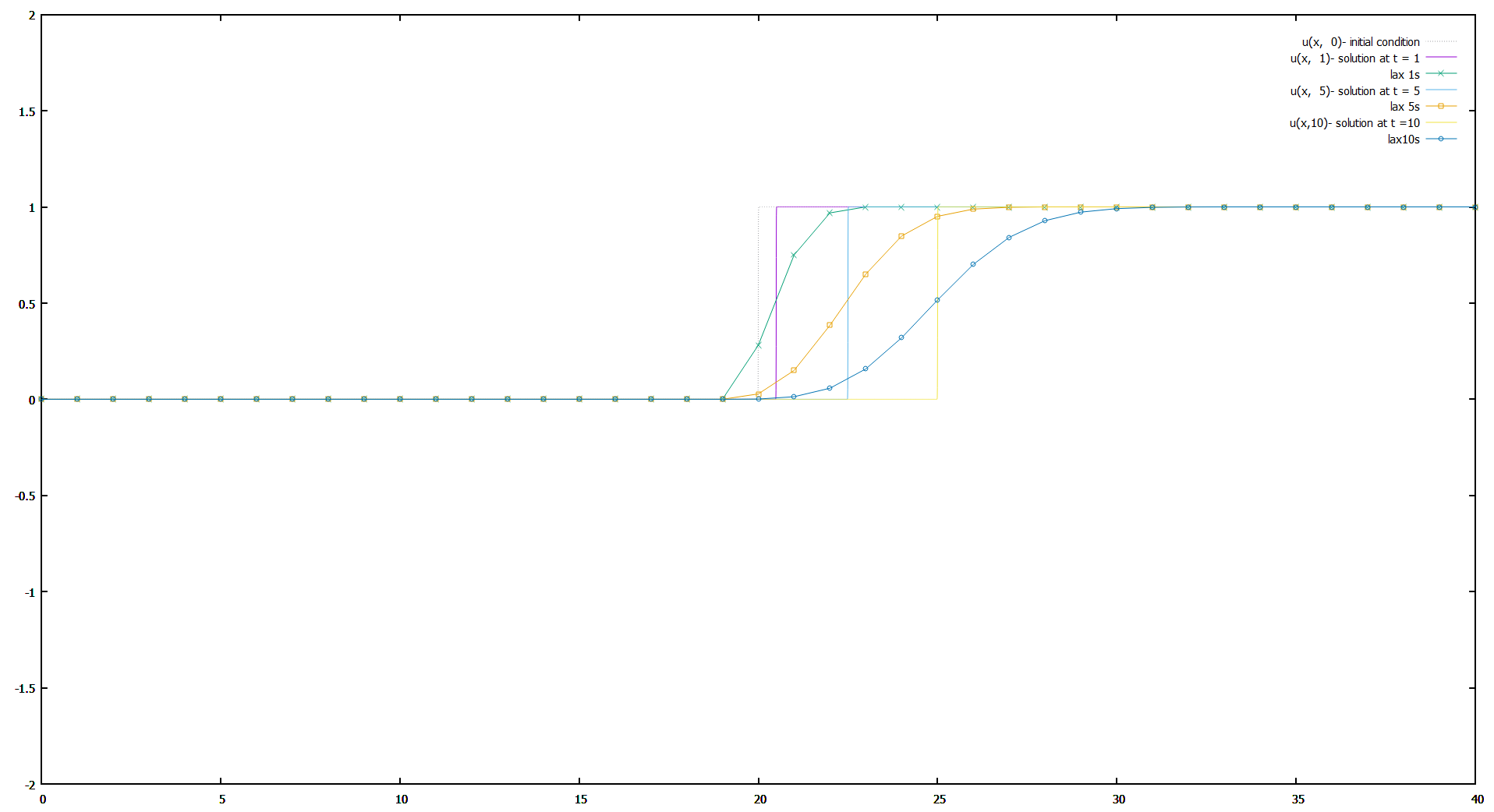


Figure 6. Lax using 40 nodes delta t = 0.5s

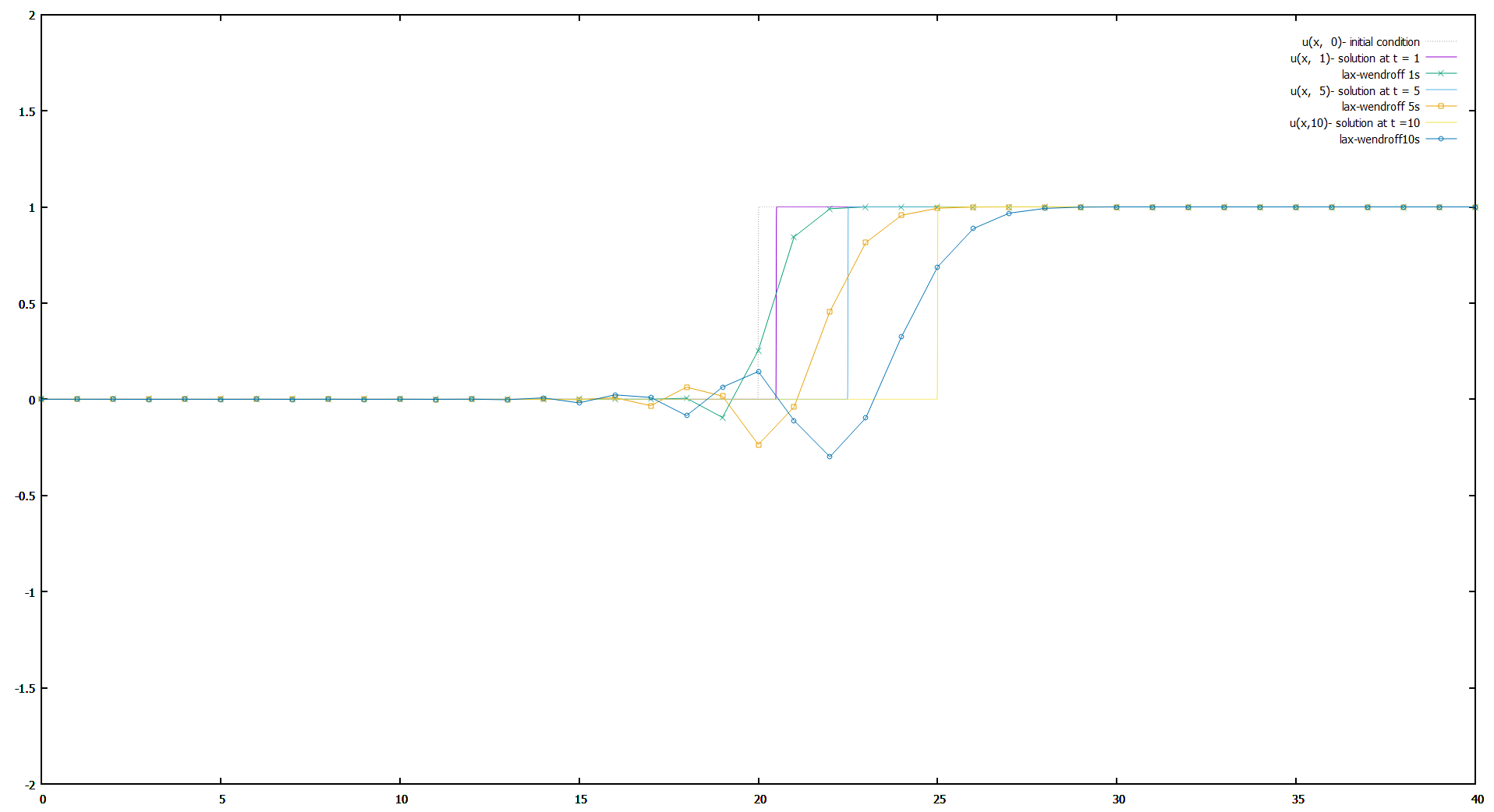


Figure 7. Lax-Wendroff using 40 nodes delta t = 0.5s

# Algorithm comparison

## Timestep effect on methods

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

### Upwind

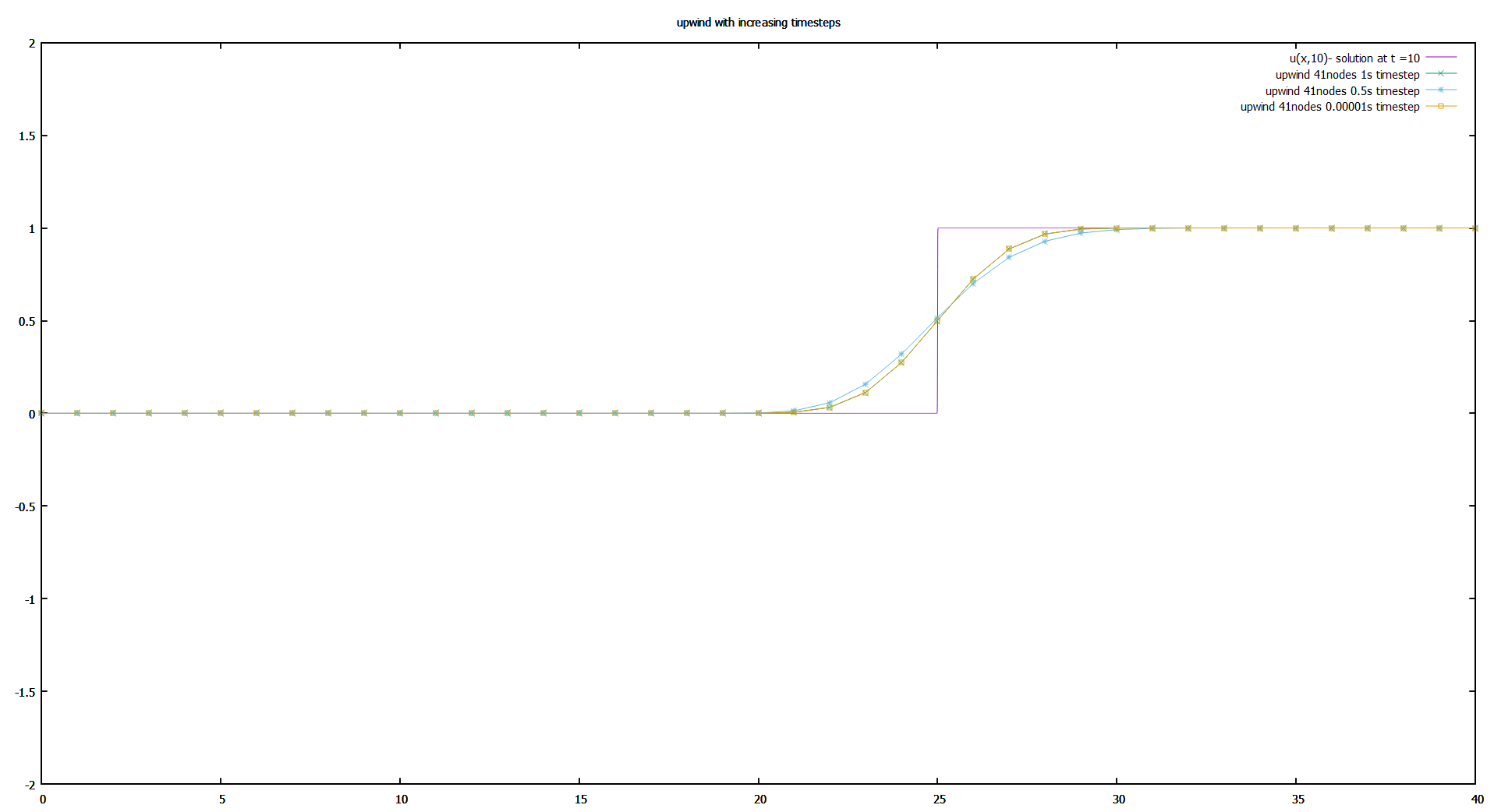


Figure 8. Upwind using 41 nodes. Increasing timesteps

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

### Lax

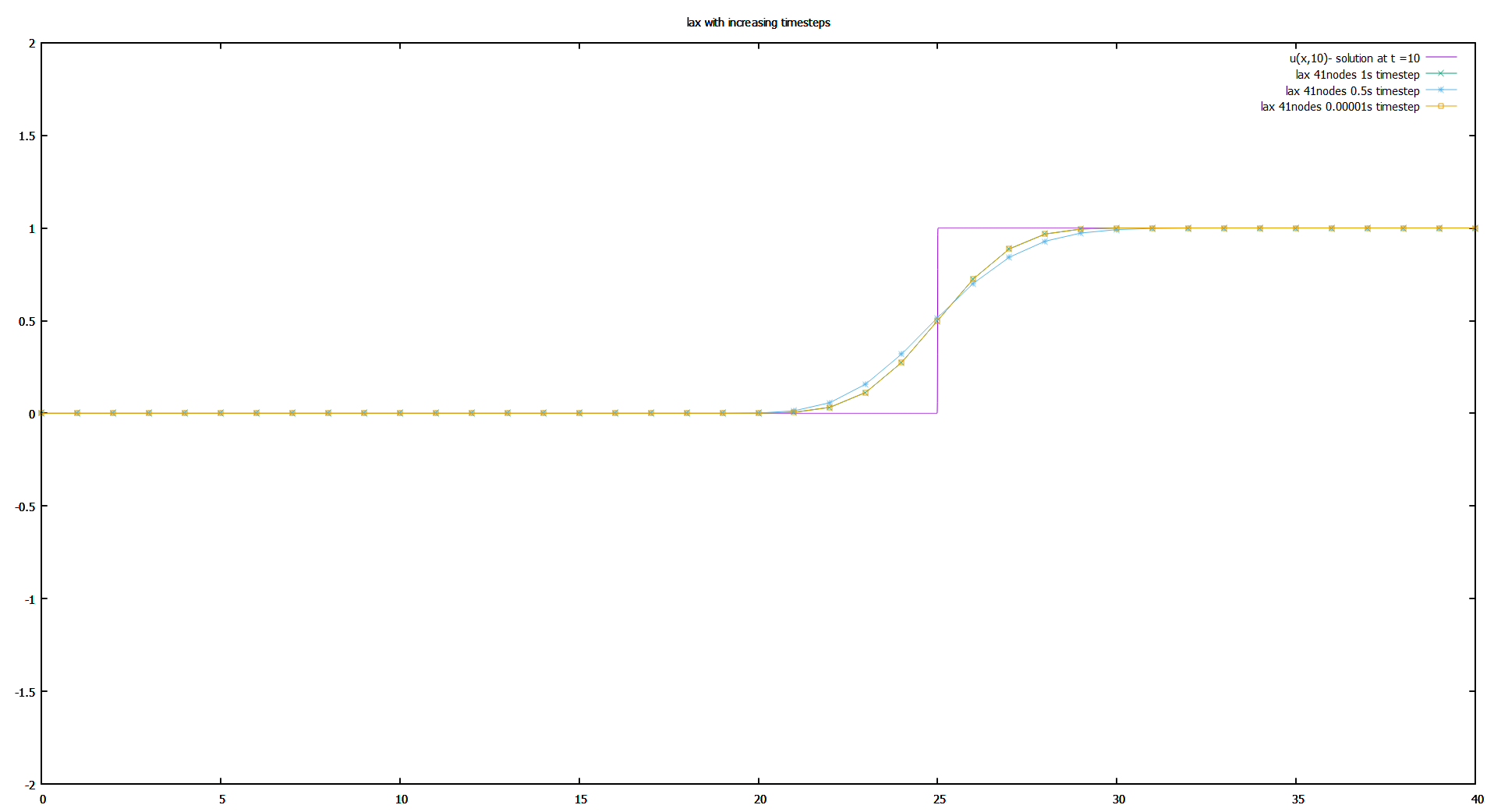


Figure 9. Lax using 41 nodes. Increasing timesteps

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

### Lax-Wendroff

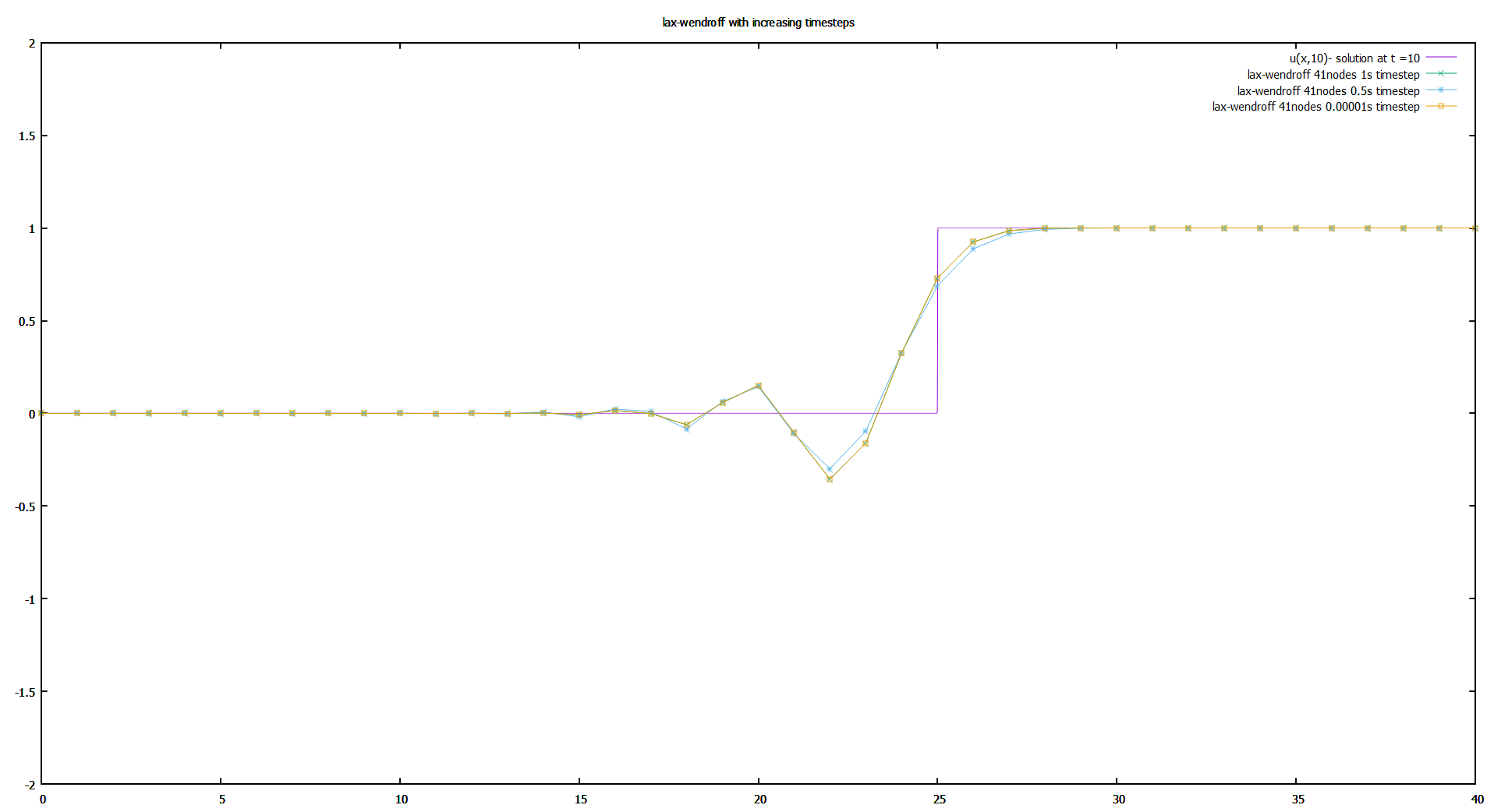


Figure 10. Lax-Wendroff using 41 nodes. Increasing timesteps

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.

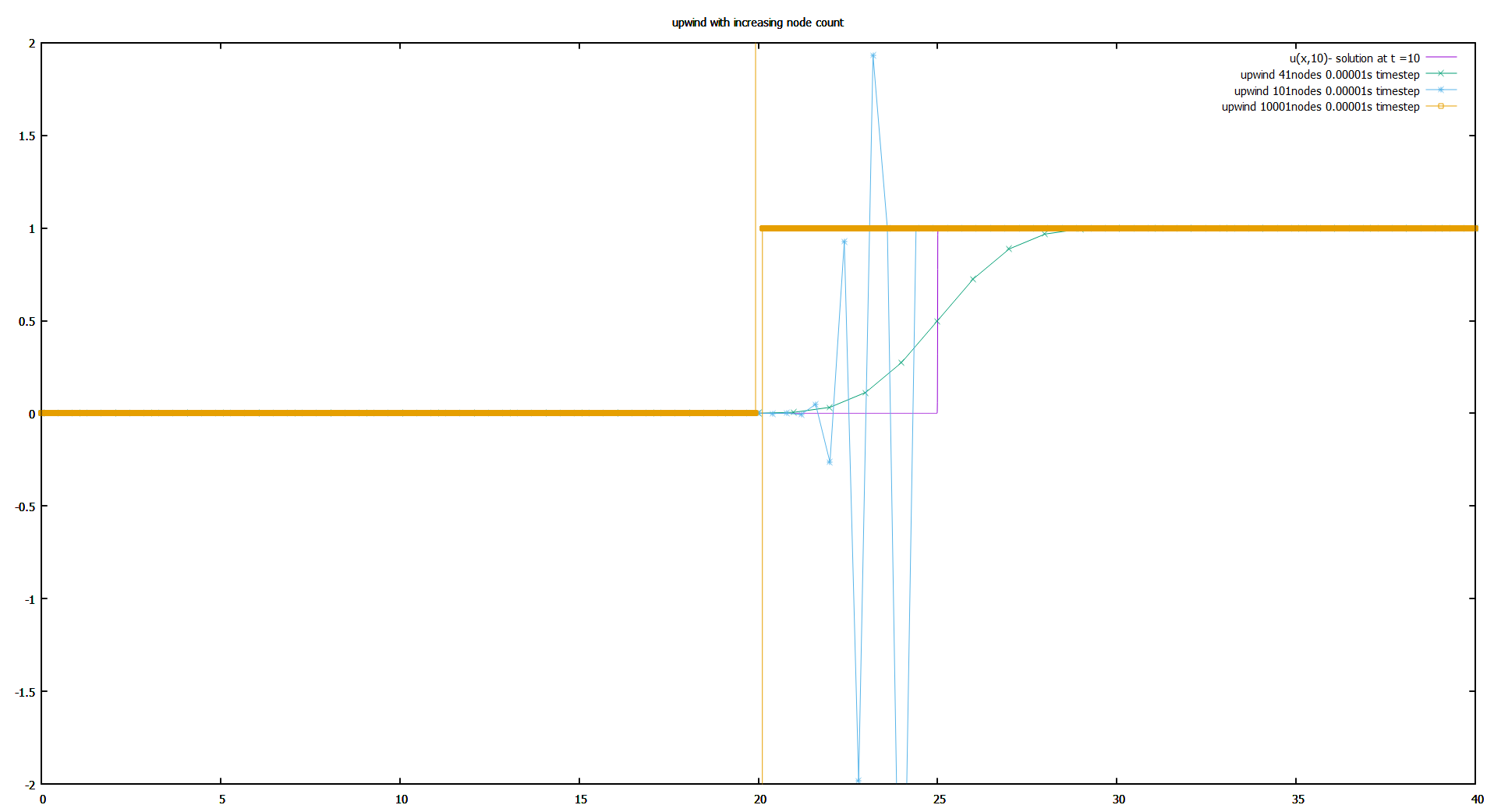


Figure 11. The timestep was 00001s in the code. it's actually 1s timestep.

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

### Upwind

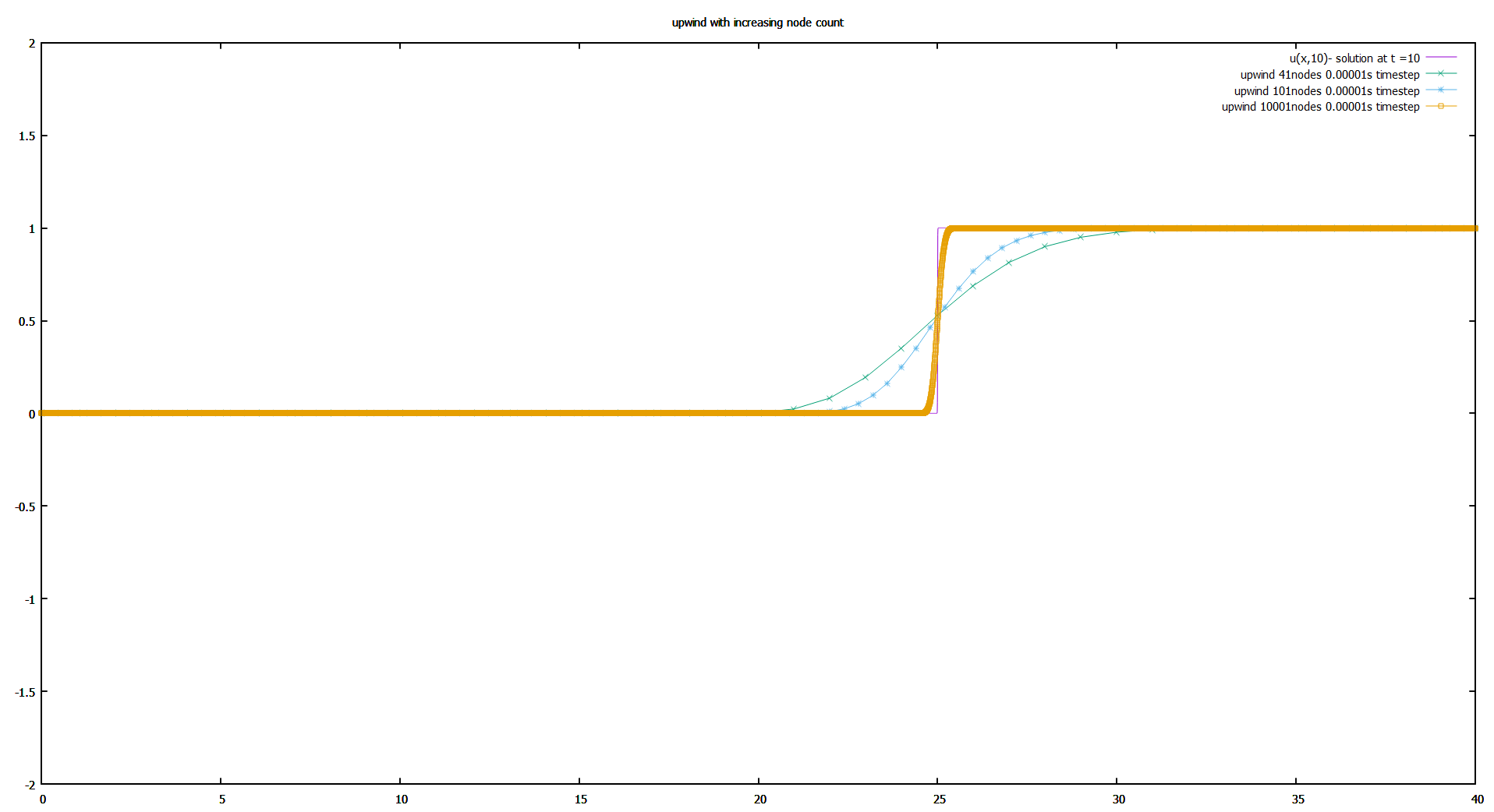


Figure 12. Upwind scheme with increasing node count

The error substantially decreases with increasing node count.

### Lax

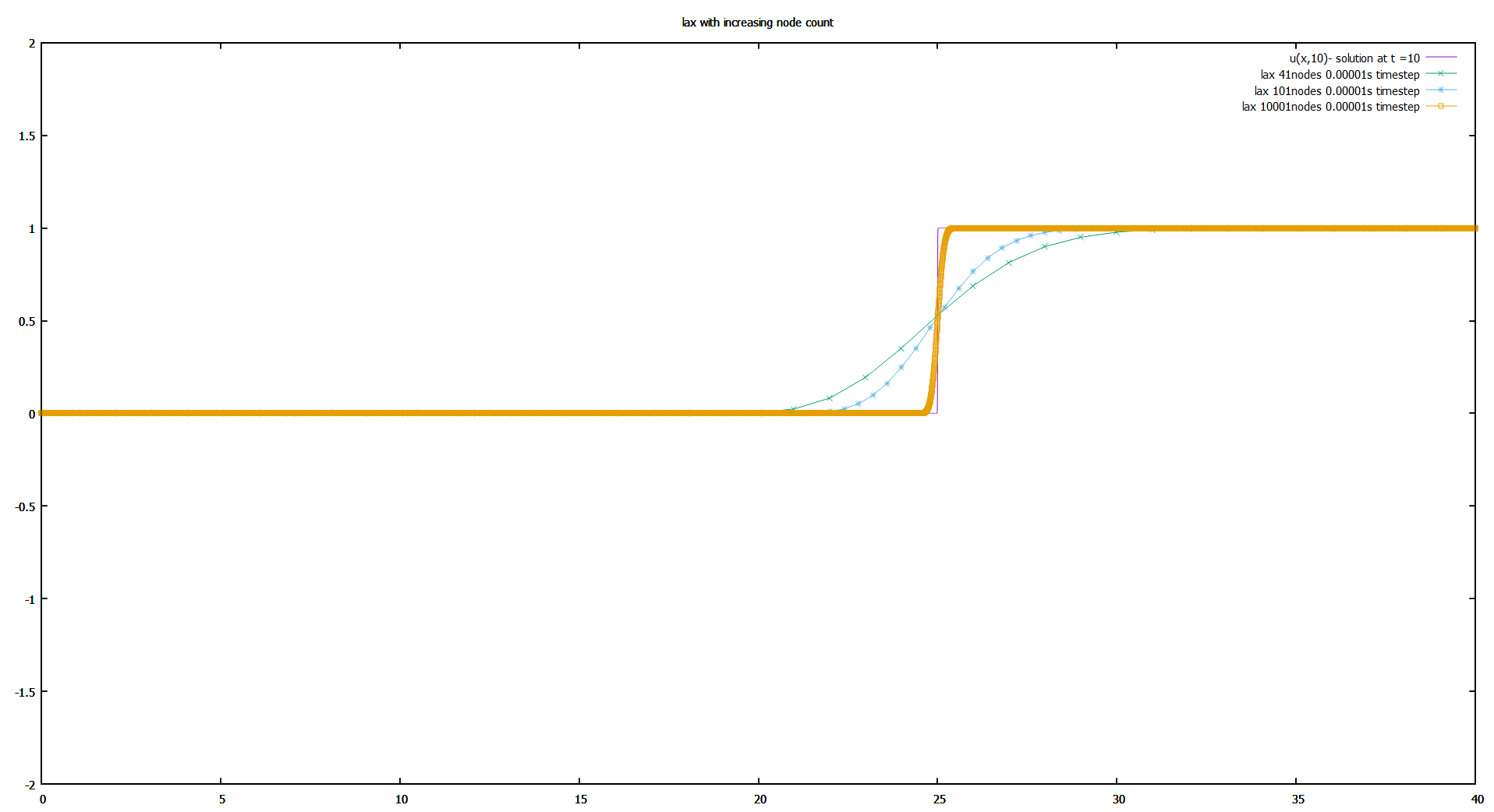


Figure 13. Lax scheme with increasing node count

The error substantially decreases with increasing node count.

### Lax-Wendroff

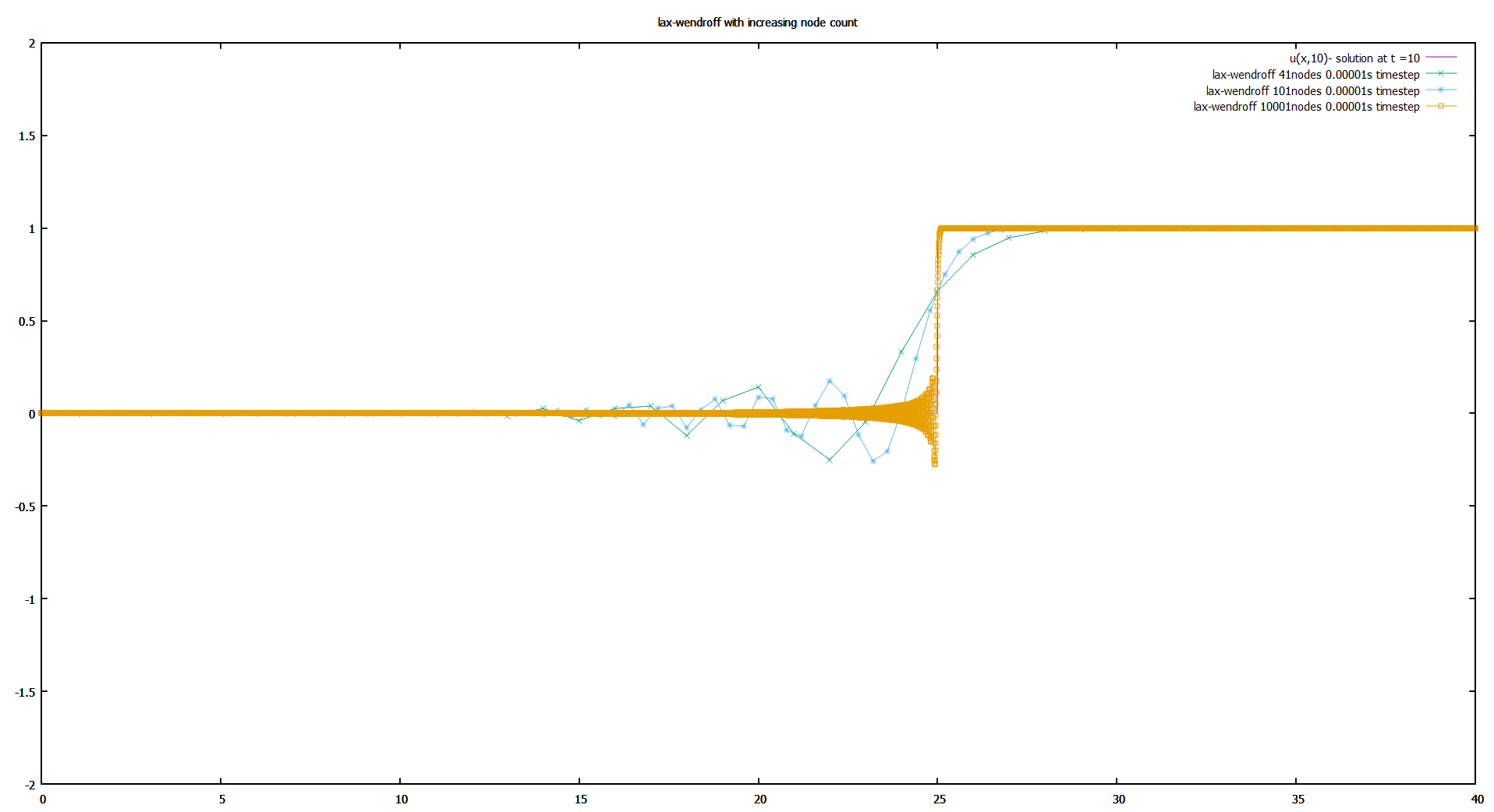


Figure 14. Lax-Wendroff scheme with increasing node count

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:

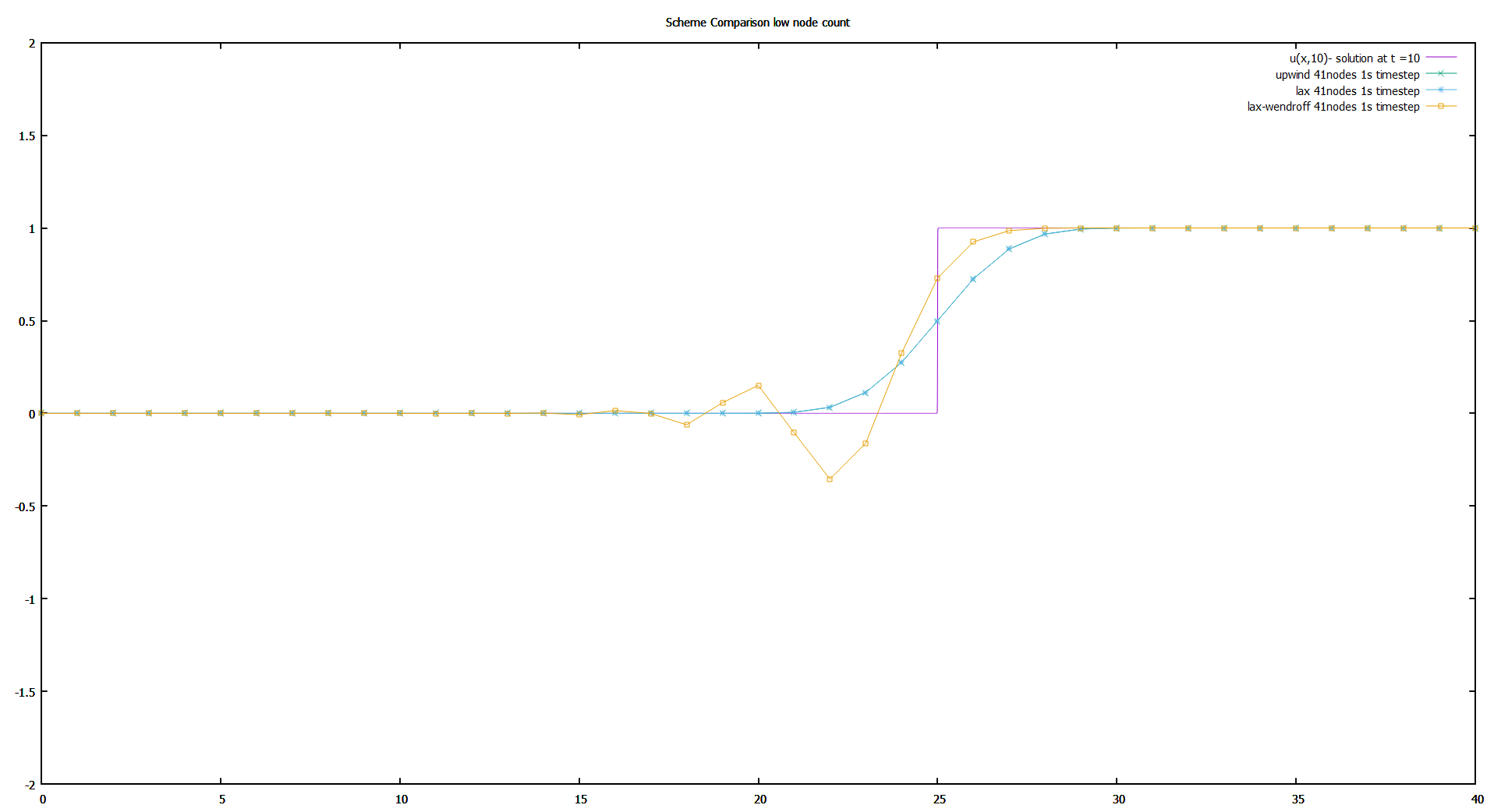


Figure 15. Different schemes at low node count.

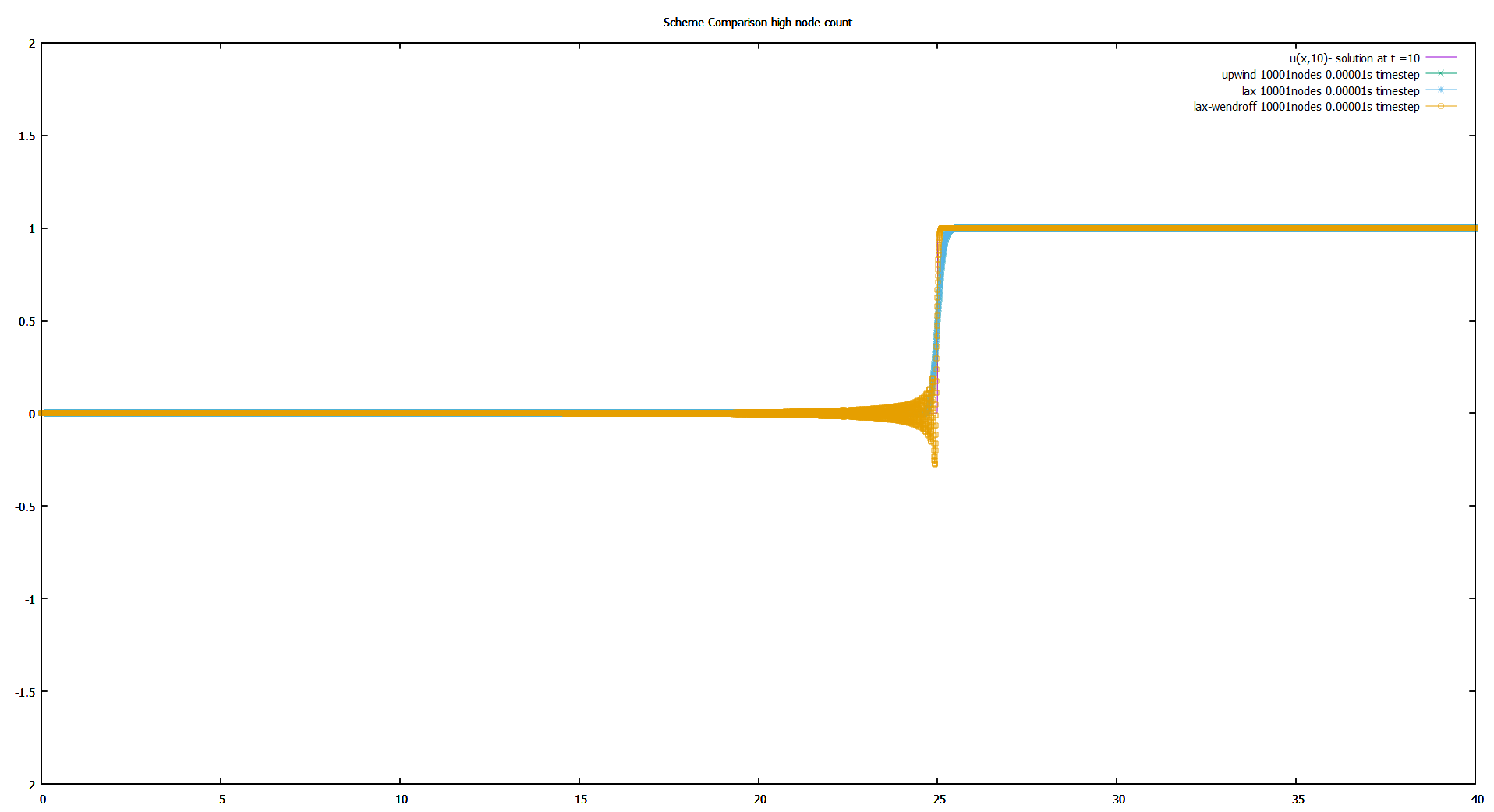
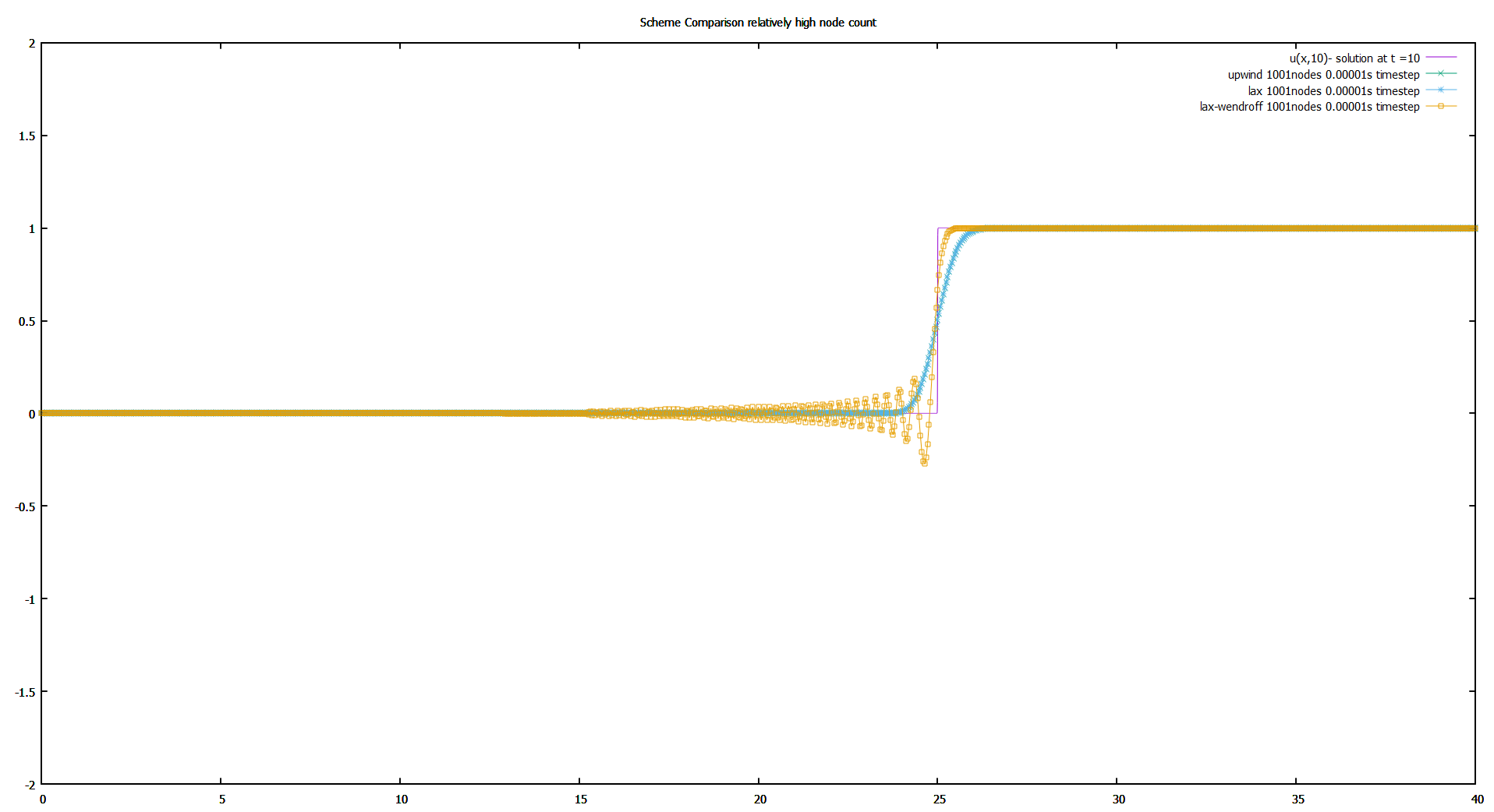


Figure 16. Different schemes at high node count.



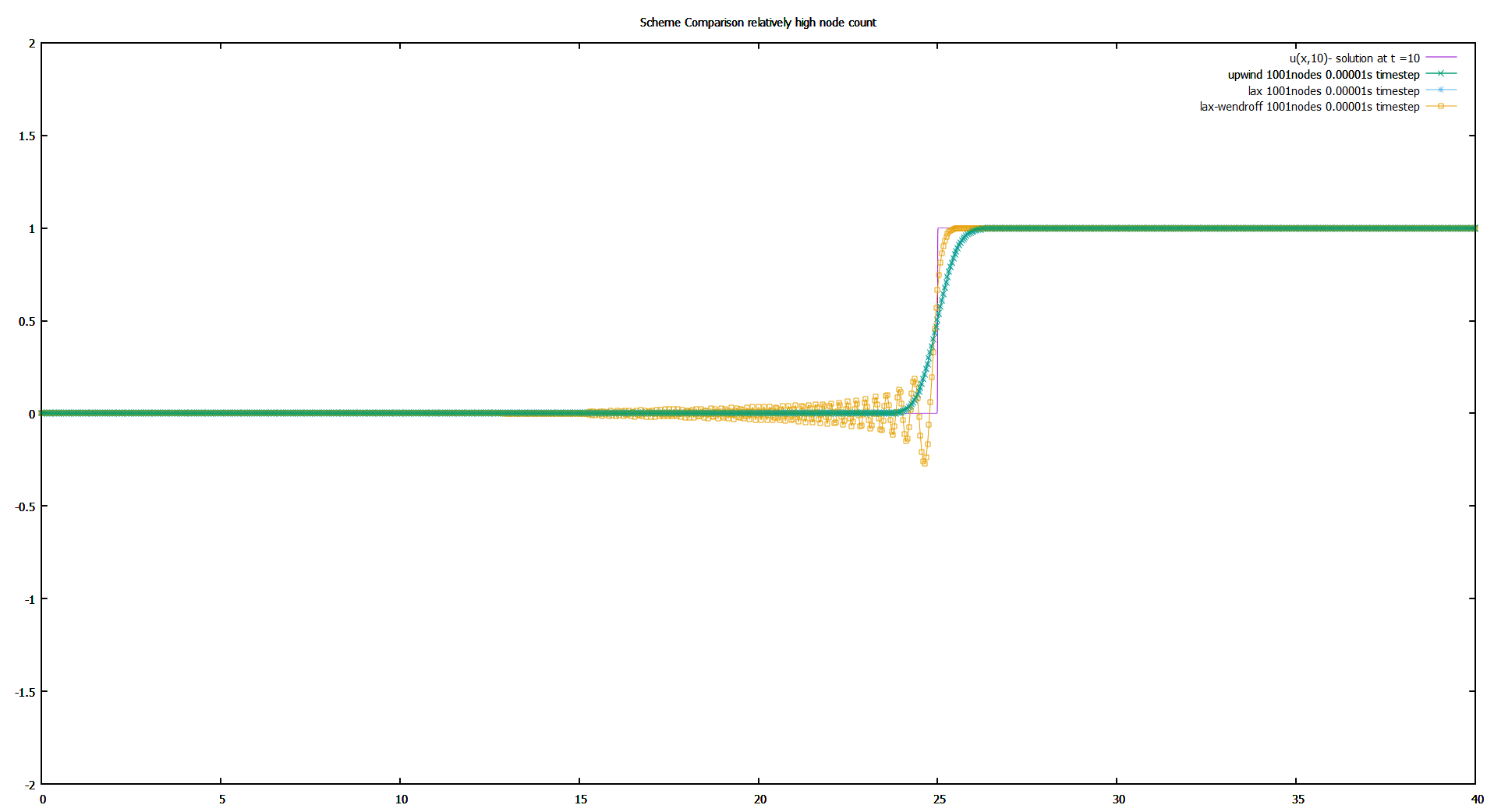


Figure 17. Different schemes at relatively high node count. Upwind replotted in the second graph.

### 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.