One-dimensional traffic flow

Numerical Analysis 1

Yannick Perrenet & Jelle de Jong

November 3, 2018

Contents

1 Introduction

In this report we will model car traffic flow using conservation laws. We start off with a basic model and will analyze and improve the model based on results found from our analysis. The research question we want to answer is: "How long will it take for all cars to have left the road?" Based on an initial distribution and the assumption that no new cars enter the road. In order to answer this question we will create a model for simulation in Python.

To solve the differential equation we will use numerical methods like Euler Forward.

Used definitions, theorems and results can be found in the appendix, as well as the Python code.

2 Analysis

2.1 Problem description

We start by considering the car density $\rho(x,t)$ as a function of time and space, where $x \in [0, L]$ will be one-dimensional (a *street* of length L). We choose a very simple approach for the car velocity $v(\rho) := 1-\rho$. The density in an interval $[x_1, x_2]$ (control volume) can only change in time via the flow across its boundary.
The given description results in a mathematical problem that can be stated as follows,

$$\begin{cases} \frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2} - \frac{\partial}{\partial x}\left(\frac{1}{2}u^2\right) & 0 < x < L,\ t > 0 \\ u(0,t) = u_l(t) & t \geq 0 \\ \frac{\partial u}{\partial x}(L,t) = u_r(t) & t \geq 0 \\ u(x,0) = u^0(x) & 0 < x < L \end{cases} \tag{2.1}$$

where $\nu > 0$ is the viscosity parameter, and the following initial condition,

$$u^0(x) = 0, \qquad 0 < x < L \tag{2.2}$$

and boundary conditions

$$u_l(t) = -1, \qquad u_r(t) = 0$$

are assumed.

2.2 Modeling

We first state the equations we will be discussing.

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(\rho(1-\rho)) = \nu \frac{\partial^2 \rho}{\partial x^2} \qquad \nu \in \mathbb{R}_+ \tag{2.3}$$

and

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\frac{1}{2}u^2\right) = \nu \frac{\partial^2 u}{\partial x^2} \qquad \nu \in \mathbb{R}_+ \tag{2.4}$$

where $u(x,t) = \alpha\rho + \beta$ a linear transformation.
Now we want to find values for α and β such that equation (2.3) can be

transformed into (2.4). This can be done by substituting $u(x,t) = \alpha\rho + \beta$ in (2.4) and applying the chain rule on the partial derivatives, giving us

$$\frac{\partial u}{\partial \rho}\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}\left(\frac{1}{2}((\alpha\rho)^2 + 2\alpha\beta\rho + \beta^2)\right) = \nu\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial \rho}\frac{\partial \rho}{\partial x}\right)$$

rewritten this becomes

$$\alpha\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}\left(\frac{1}{2}((\alpha\rho)^2 + 2\alpha\beta\rho + \beta^2)\right) = \nu\alpha\frac{\partial^2 \rho}{\partial x^2}$$

Next we divide by α, but not without noting that $\alpha \neq 0$ since we otherwise would be left without linear transformation, and compare the resulting term next to $\frac{\partial}{\partial x}$ with $\rho(1-\rho) = -\rho^2 + \rho$. Obtaining

$$\frac{1}{2}\alpha\rho^2 + \beta\rho + \frac{\beta^2}{\alpha} = -\rho^2 + \rho$$

Before we can calculate α and β we first need to note that we can disregard the term $\frac{\beta^2}{\alpha}$ because this constant value will be omitted when differentiating. Now we can easily compute the values, resulting in $\alpha = -2$ and $\beta = 1$.

Before we continue with the analysis we have a look at what the initial and boundary conditions mean in the physical setting.
Since $u(x,t)$ is a linear transformation of ρ and we know what ρ stands for in our model, we rewrite the conditions in terms of ρ.

1. Initial condition
   We have $-2\rho(x,0) + 1 = u(x,0) = 0$. This gives us $\rho(x,0) = \frac{1}{2}$. In a physical setting this tells us that the density is at half its maximum capacity: "the road is half full". Since the density is related to the velocity ($v = 1 - \rho$) we also know that the cars are driving at half of their maximum speed.

2. Boundary conditions
   Here we have $-2\rho(0,t) + 1 = u(0,t) = -1$ and $-2\frac{\partial \rho}{\partial x} = \frac{\partial u}{\partial x}(L,t) = 0$, this gives us resp. $\rho(0,t) = 1$ and $\frac{\partial \rho}{\partial x} = 0$. The first condition means that at $x = 0$ the density is maximal for every time step: "the maximal amount of cars is driving into our street". The second condition means that the density at the end of the street is constant: "a constant number of cars is leaving the street".

2.3 Spatial discretization and matrix formulation

We derive a semi-discrete matrix formulation of our model, by discretizing (2.1) in the spatial x-direction by means of the finite-difference method. The

interval $[0, L]$ is divided into $N+1$ equal parts with length Δx, where the nodes are given by $x_i = i\Delta x$, $i = 0, \ldots, N+1$. The vector $\mathbf{u}(t)$ is given by $\mathbf{u}(t) = (u_1(t), \ldots, u_{N+1})^T$; it has $N+1$ elements because there are $N+1$ unknowns due to the Neumann boundary at $x_{N+1} = L$. The value at $x_0 = 0$ is omitted, because it is known from the Dirichlet boundary condition.

2.3.1 Finite-difference method

Since the finite-difference method uses central differences to approximate the first and second derivative, we first state the central difference formula for the first derivative of a certain function $g(x)$

$$Q_1(h) = \frac{g(x+h) - g(x-h)}{2h} \tag{2.5}$$

and for the second derivative

$$Q_2(h) = \frac{g(x+h) - 2g(x) + g(x-h)}{h^2} \tag{2.6}$$

Applying these formulae to our boundary-value problem (2.1) we obtain

$$\frac{\partial u_j}{\partial t} = \nu \frac{u_{j+1} - 2u_j + u_{j-1}}{\Delta x^2} - \frac{1}{2}\frac{(u_{j+1})^2 - (u_{j-1})^2}{2\Delta x} \qquad 2 \leq j \leq N \tag{2.7}$$

For $j = 1$ we have $u_{j-1} = u_0 = u_l(t) = -1$, which is the nonhomogeneous Dirichlet boundary condition. This results in the following equation

$$\frac{\partial u_1}{\partial t} = \nu \frac{u_2 - 2u_1 + (-1)}{\Delta x^2} - \frac{1}{2}\frac{(u_2)^2 - (-1)^2}{2\Delta x} \tag{2.8}$$

$$= \nu \frac{u_2 - 2u_1}{\Delta x^2} - \frac{(u_2)^2}{4\Delta x} + \left(\frac{1}{4\Delta x} - \nu\frac{1}{\Delta x^2}\right) \tag{2.9}$$

As was previously said, u_{N+1} isn't known due to the Neumann boundary. Thus it is included in the discretization. Note that for $j = N+1$ equation (2.7) contains the value u_{N+2}, which does not exists since it falls out of the interval. In order to approximate u_{N+1} properly, a virtual point $x_{N+2} = (N+2)\Delta x = L + \Delta x$ is introduced. The value of u_{N+2} follows from discretization of the Neumann boundary condition using central difference equation (2.5)

$$\frac{u_{N+2} - u_N}{2\Delta x} = 0 \tag{2.10}$$

This implies $u_{N+2} = u_N$. We substitute this into (2.7) to obtain

$$\frac{\partial u_{N+1}}{\partial t} = \nu \frac{u_N - 2u_{N+1} + u_N}{\Delta x^2} - \frac{1}{2}\frac{(u_N)^2 - (u_N)^2}{2\Delta x} \tag{2.11}$$

$$= \nu \frac{2u_N - 2u_{N+1}}{\Delta x^2} \tag{2.12}$$

2.3.2 Semi-discrete matrix formulation

By combining the results found in equations: (2.7), (2.9) and (2.12), we derive a semi-discrete matrix formulation of the form

$$\dot{\mathbf{u}} = \nu \mathbf{K}\mathbf{u} - \mathbf{f}(\mathbf{u}) + \mathbf{r}(\mathbf{u}) \tag{2.13}$$

with

$$\mathbf{K} = \frac{1}{\Delta x^2} \begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 2 & -2 \end{pmatrix}, \tag{2.14}$$

$$\mathbf{f}(\mathbf{u}) = \frac{1}{4\Delta x} \begin{pmatrix} 0 & 1 & & & \\ -1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 0 & 1 \\ 0 & \dots & \dots & \dots & 0 \end{pmatrix} \begin{pmatrix} (u_1)^2 \\ (u_2)^2 \\ \vdots \\ (u_N)^2 \\ (u_{N+1})^2 \end{pmatrix}, \tag{2.15}$$

$$\mathbf{r}(\mathbf{u}) = \left(\frac{1}{4\Delta x} - \nu \frac{1}{\Delta x^2} \right) \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}, \tag{2.16}$$

$\mathbf{u}(t) = (u_1(t), \dots, u_{N+1})^T$ and $\dot{\mathbf{u}} = (\frac{\partial u_1}{\partial t}, \dots, \frac{\partial u_{N+1}}{\partial t})^T$.

2.4 Time integration

In the previous section we found a semi-discrete matrix formulation for our problem (2.1). With this formulation we can make a simulation[1] to examine the traffic flow.
Remember from section 2.2 that we had $u(x,t) = -2\rho + 1$. We will be simulating both the solutions, since we have a way to calculate u, but ρ is closer related to the density on the road (as well as the velocity of the cars).
As for the numerical time-integration method, we used Euler Forward and we took the parameter configuration as shown in table 2.1.
To give you a little taste of the simulation, we have included the final frame in figure 2.1.

[1] The Python code can be found in section 5.1.1.

ν	N	L	t_e	Δt
0.5	100	3.0	5.0	0.0001

Table 2.1: Parameter configuration.

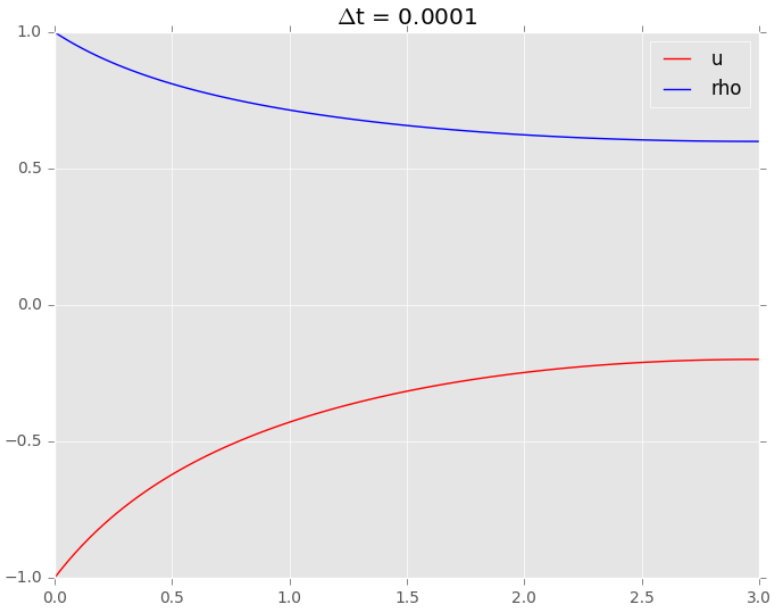

Figure 2.1: Final frame of the simulation.

2.5 Stability analysis of the linearized equation

The simulation in the previous section was quite slow. One of the things we can do to speed it up is choosing an as big as possible time-step, but still have a stable approximation method. In order to find this time-step we compute the Jacobian matrix at the initial condition (2.2) and use the stability region of Euler forward, which we used as the integration method in the simulation.
Before we can compute the Jacobian we have to rewrite (2.24) into a single

vector

$$\dot{\mathbf{u}} = \nu \frac{1}{\Delta x^2} \begin{pmatrix} -2u_1 + u_2 \\ u_1 - 2u_2 + u_3 \\ \vdots \\ u_{N-1} - 2u_N + u_{N+1} \\ 2u_N - 2u_{N+1} \end{pmatrix} - \frac{1}{4\Delta x} \begin{pmatrix} (u_2)^2 \\ -(u_1)^2 + (u_3)^2 \\ \vdots \\ -(u_{N-1})^2 + (u_{N+1})^2 \\ 0 \end{pmatrix} + \mathbf{r}(\mathbf{u})$$

$$= \begin{pmatrix} \frac{\nu(-2u_1+u_2)}{\Delta x^2} - \frac{(u_2)^2}{4\Delta x} + \frac{1}{4\Delta x} - \nu\frac{1}{\Delta x^2} \\ \frac{\nu(u_1-2u_2+u_3)}{\Delta x^2} - \frac{-(u_1)^2+(u_3)^2}{4\Delta x} \\ \vdots \\ \frac{\nu(u_{N-1}-2u_N+u_{N+1})}{\Delta x^2} - \frac{-(u_{N-1})^2+(u_{N+1})^2}{4\Delta x} \\ \frac{\nu(2u_N-2u_{N+1})}{\Delta x^2} \end{pmatrix}$$

Writing $\dot{\mathbf{u}} = \mathbf{g}(\mathbf{u})$ we can now find the Jacobian via $[\mathbf{J}]_{i,j} = \frac{\partial g_i}{\partial u_j}$. Then we substitute the initial condition $\mathbf{u}^0 = \mathbf{0}$, giving

$$\mathbf{J}(\mathbf{u}^0) = \begin{pmatrix} -\frac{2\nu}{\Delta x^2} & \frac{\nu}{\Delta x^2} & & & \\ \frac{\nu}{\Delta x^2} & -\frac{2\nu}{\Delta x^2} & \frac{\nu}{\Delta x^2} & & \\ & \ddots & \ddots & \ddots & \\ & & \frac{\nu}{\Delta x^2} & -\frac{2\nu}{\Delta x^2} & \frac{\nu}{\Delta x^2} \\ & & & \frac{2\nu}{\Delta x^2} & -\frac{2\nu}{\Delta x^2} \end{pmatrix} \tag{2.17}$$

$$= \frac{\nu}{\Delta x^2} \begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 2 & -2 \end{pmatrix} \tag{2.18}$$

$$= \nu\mathbf{K} \tag{2.19}$$

where $\mathbf{K}$ as in section 2.3.2.
Before we can use the stability condition for Euler forward[2]: $\Delta t \leq -\frac{2}{\lambda}$, we need to compute the extreme eigenvalues of the matrix $\nu\mathbf{K}$. A good estimate can be found using the circle theorem of Gershgorin[3] as you can see here

$$|\lambda + \frac{2\nu}{\Delta x^2}| \leq 2\frac{\nu}{\Delta x^2}$$

which implies $-\frac{4\nu}{\Delta x^2} \leq \lambda \leq 0$. Now we obtained a maximal step-size for Euler forward

$$\Delta t \leq \frac{\Delta x^2}{2\nu} \tag{2.20}$$

[2]See table 5.1 in section 5.2.3 to find out more about stability conditions.
[3]See section 5.2.2 theorem (1).

2.6 Simulation

Now that we have found an expression for the biggest possible time-step, whilst having stability for our time-integration method, we would like to use it in our simulation to speed things up. To check that no mistakes were made, we increase the derived step-size in (2.20) with 1% and ran the simulation[4] with the parameters as mentioned in table 2.1 (except for the time-step of course).
As you can see in figure 2.2: the approximation explodes after some time steps, just as expected. It can be concluded that the approximation is *sharp*.

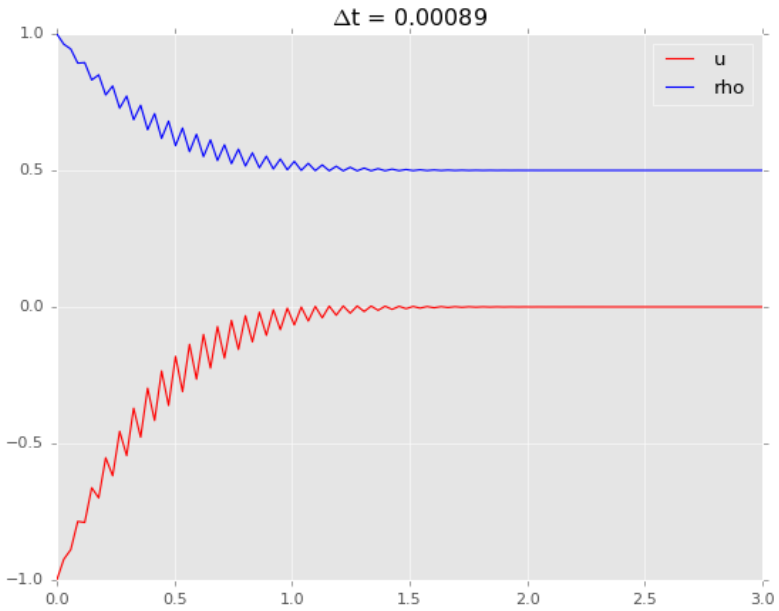

Figure 2.2: Exploding approximation.

2.7 Upwind scheme

To answer our research question we change things up a bit and consider the new initial condition:

$$u^0(x) = \begin{cases} 1, & 0 \leq x \leq L/3 \\ 2 - (3/L)x, & L/3 \leq x \leq 2L/3 \\ 0, & x \geq 2L/3 \end{cases} \tag{2.21}$$

[4]The code can be found in section 5.1.2.

together with new boundary conditions

$$u_l(t) = +1, \qquad u_r(t) = 0, \qquad t \geq 0$$

The only way in which the model changes in regard to the new boundary conditions is that in stead of -1, we now substitute 1 in equation (2.7) Since we already have a simulation; it would be a waste not use it. We add and change a couple of lines in the Python code to get a new simulation[5]. Besides the conditions we also use a new parameter configuration to make things more realistic. During the simulation something strange happens as

ν	N	L	t_e	Δt
0.01	100	3.0	5.0	0.003

Table 2.2: More realistic parameter configuration.

you can see in figure 3.1, namely ρ is smaller than zero and u is greater than one, which should be impossible considering our assumptions. To avoid the

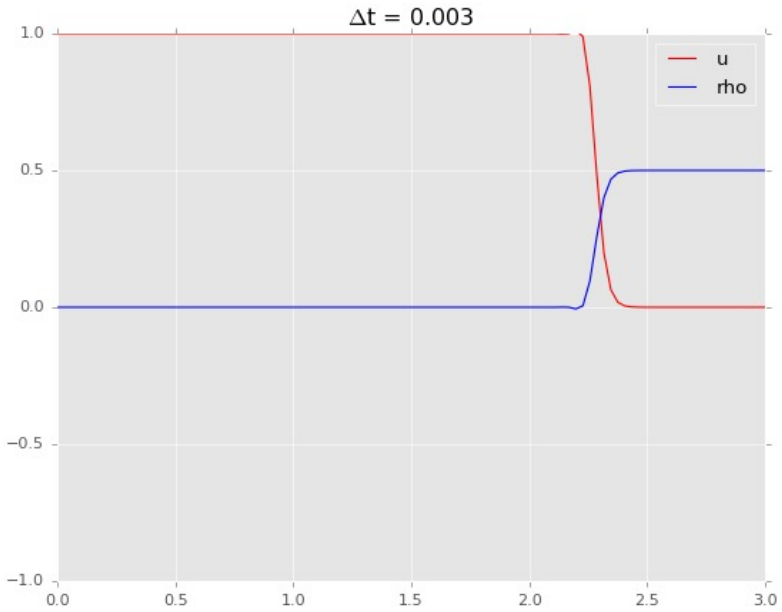

Figure 2.3: Both graphs falling outside of their boundaries.

strangeness we note that $u > 0$ and replace the central discretization for the first derivative in the first equation in (2.1) by an *upwind scheme*.

[5]This new simulation can be found in section 5.1.3.

We use the same approach as in section 2.3, but now replacing equation (2.5) by

$$Q_1(h) = \frac{g(x) - g(x-h)}{h} \tag{2.22}$$

giving us the following equations

$$\frac{\partial u_j}{\partial t} = \begin{cases} \nu\frac{u_2-2u_1}{\Delta x^2} - \frac{(u_1)^2}{\Delta x} + \left(\frac{1}{2\Delta x} + \nu\frac{1}{\Delta x^2}\right) & j = 1 \\ \nu\frac{u_{j+1}-2u_j+u_{j-1}}{\Delta x^2} - \frac{1}{2}\frac{(u_j)^2-(u_{j-1})^2}{\Delta x} & 2 \leq j \leq N \\ \nu\frac{2u_N-2u_{N+1}}{\Delta x^2} - \frac{1}{2}\frac{(u_{N+1})^2-(u_N)^2}{\Delta x} & j = N+1 \end{cases} \tag{2.23}$$

By combining the results from equation (2.23), we again derive a matrix formulation of the form

$$\dot{\mathbf{u}} = \nu\mathbf{K}\mathbf{u} - \mathbf{f}(\mathbf{u}) + \mathbf{r}(\mathbf{u}) \tag{2.24}$$

with

$$\mathbf{K} = \frac{1}{\Delta x^2}\begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 2 & -2 \end{pmatrix}, \tag{2.25}$$

$$\mathbf{f}(\mathbf{u}) = \frac{1}{2\Delta x}\begin{pmatrix} 1 & & & & \\ -1 & 1 & & & \\ & \ddots & \ddots & & \\ & & -1 & 1 & \\ & & & -1 & 1 \end{pmatrix}\begin{pmatrix} (u_1)^2 \\ (u_2)^2 \\ \vdots \\ (u_N)^2 \\ (u_{N+1})^2 \end{pmatrix}, \tag{2.26}$$

$$\mathbf{r}(\mathbf{u}) = \left(\frac{1}{2\Delta x} + \nu\frac{1}{\Delta x^2}\right)\begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}, \tag{2.27}$$

$\mathbf{u}(t) = (u_1(t), \ldots, u_{N+1})^T$ and $\dot{\mathbf{u}} = (\frac{\partial u_1}{\partial t}, \ldots, \frac{\partial u_{N+1}}{\partial t})^T$.

2.7.1 Threshold

In order to answer our research question we will have to integrate the car density ρ with respect to x at all time steps. We will use the Trapezoidal rule in order to do so. Since this is a numerical approach we will need to set a threshold on ρ close to zero. We take a threshold of 0.001, and thus once

this threshold is met; all cars will have left the road.
Integrating the car density at a certain time step yields the following

$$\begin{aligned}\int_{x_0}^{x_L} \rho(x,t)dx &= \sum_{j=0}^{N} \int_{x_j}^{x_{j+1}} \rho(x,t)dx \\ &\approx \sum_{j=0}^{N} \frac{\Delta x}{2} \left(\rho_j + \rho_{j+1}\right) \\ &= \frac{\Delta x}{2}(\rho_1 + \rho_{N+1}) + \Delta x(\rho_2 + \rho_3 + \ldots \rho_N)\end{aligned}$$

Adding this to our simulation gives us the ability to answer our research question.

3 Numerical results

Altering the Python code for the simulation[1] such that it now implements an upwind scheme, we choose the parameter configuration from table 2.2 and have a look at the resulting plot We see that both the graphs in our

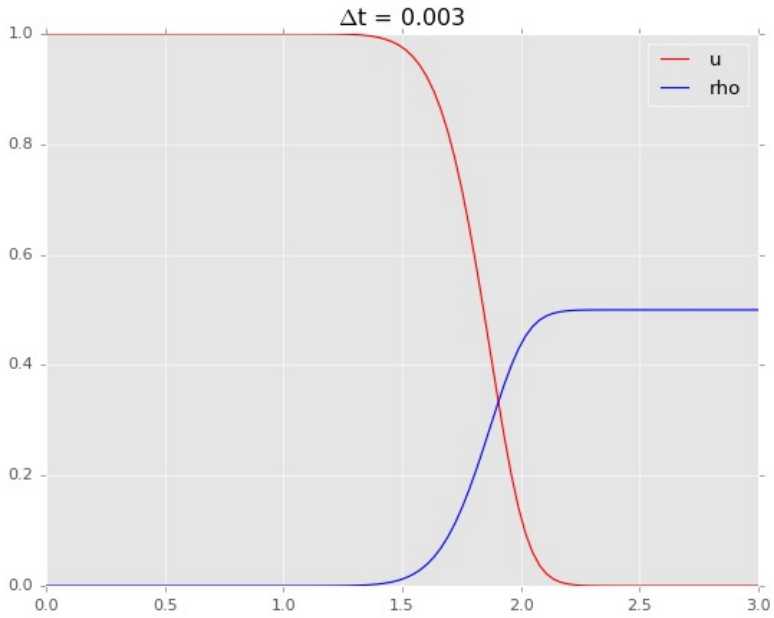

Figure 3.1: Showing a frame from the final simulation.

figure are smooth and do not violate any of the assumptions. If you run the simulation you will see that *rho* will decrease as time passes by at every x until it hits zero. This is easy to explain since we have the assumption that no cars will enter the street.

You will notice that running the simulation will take a considerable amount of time. You could speed this up by increasing the time step such the method is still stable, but we will not cover this here.

[1]The Python code can be found in section 5.1.4.

4 Conclusion

Remember that we wanted to answer the question: "How long will it take for all cars to have left the road?" To answer this question we implement our findings from section 2.7.1 in our simulation[1]. Then running the simulation with the parameter configuration from table 2.2 gives us an answer: "The road will be empty after 3.189 time units."

Note that the answer does not depend on the time step you take.

[1]The Python code can be found in section 5.1.5.

5 Appendix

5.1 Python code

All the code is written following the coding conventions in PEP 8 – Style Guide for Python Code.[1]

5.1.1 Framework

This is the framework. In all the other sections we only include the parts of the python code you need to add to or change in the framework. We do assume that you work your way from top to bottom, meaning that the code in the last section will not work if you haven't added the code from the sections before.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import matplotlib.animation as animation
4  from scipy.sparse import diags
5
6  # makes the plots prettier
7  plt.style.use('ggplot')
8
9
10 def next_step_approximation(u):
11     """ Applying a semi-discrete matrix formulation on a vector u
12     to compute a partial derivative of u with respect to t.
13
14     Args:
15         u (np.array): vector containing values at time t
16
17     Returns:
18         res (np.array): vector with computed values at time (t + 1)
19     """
20     u_squared = np.array([elt**2 for elt in u])
21
22     # constructs f(u)
23     f = W.dot(u_squared)
24
25     # applies r(u)
```

[1] The complete guide can be found at: `https://www.python.org/dev/peps/pep-0008/`

```
26      f[0] += (1 / (4 * dx)) - v * (1 / (dx**2))
27
28      # Constructs the end result.
29      res = v * K.dot(u) + f
30
31      return res
32
33
34  def euler_forward(u_init, dt, t0, t_end):
35      """ Applying euler_forward to find a numerical approximation
36      starting at t0 and ending at t_end with time-step dt.
37
38      Args:
39          u_init (np.array): the initial value at t0
40          dt (int): step-size timewize
41          t0 (int): starting time
42          t_end (int): ending time
43
44      Yields:
45          u (np.array): numerial approximation on a certain time
46      """
47      # Copies the initial condition to u
48      u = u_init[:]
49
50      # Computes an approximation for every time-step.
51      for t in np.arange(t0, t_end + dt / 3, dt):
52          # Only uses a couple of time-steps to create a good looking
53          # animation.
54          if not t % (34 * dt):
55              yield u
56
57          # Computes the approximation at time t.
58          u = u + dt * next_step_approximation(u)
59
60      yield u
61
62
63  def plotting():
64      """ Creating and running the simulation. """
65      def update_line(u):
66          """ Replacing the previous values of u at time t, with the
67          new values of u at time (t + 1). """
68          # Adds the known value of the Dirichlet boundary.
69          u = np.insert(u, 0, -1)
70
```

```
71          # Calculates rho with the predetermined values alpha and beta.
72          rho = (u - 1) / -2
73
74          # Sets the new values to plot.
75          line_u.set_ydata(u)
76          line_rho.set_ydata(rho)
77
78          return line_u, line_rho,
79
80      # Creates and initializes the figure.
81      fig1 = plt.figure()
82      plt.xlim(0, L)
83      plt.ylim(-1, 1)
84
85      # Specifies the lines that will be updated and plotted.
86      line_u, = plt.plot([], [], 'r-', label='u')
87      line_u.set_xdata(street)
88      line_rho, = plt.plot([], [], 'b-', label='rho')
89      line_rho.set_xdata(street)
90
91      # Creates the animation using the matplotlib.animation module.
92      line_ani = animation.FuncAnimation(fig1, update_line,
93                                         frames=euler_forward(u, dt, 0, te),
94                                         interval=100, blit=True)
95
96      # uncomment the line beneath to save the figure
97      # line_ani.save("movie.mp4", metadata={"artist": "goblin_slayer"})
98
99      # shows the animation
100     plt.legend()
101     plt.title("$\Delta$t = " + str(round(dt, 5)))
102     plt.show()
103
104
105 if __name__ == "__main__":
106     # initialize parameters
107     v, N, L, te = 0.5, 100, 3.0, 5
108
109     # determines spatial step-size
110     dx = L / (N + 1)
111
112     # sets time-step
113     dt = 0.0001
114
115     # builds the street
```

```
116     street = np.arange(0, L + dx / 3, dx)
117
118     # constructs a sparse matrix W to calculate f(u)
119     diagonals_w = np.array([(N - 1) * [-1] + [0], (N + 1) * [0], N * [1]])
120     W = -(1 / (4 * dx)) * diags(diagonals_w, [-1, 0, 1])
121
122     # constructs sparse matrix K
123     diagonals_k = np.array([(N - 1) * [1] + [2], (N + 1) * [-2], N * [1]])
124     K = (1 / dx**2) * diags(diagonals_k, [-1, 0, 1])
125
126     # sets up the initial value of the problem
127     u = np.array((N + 1) * [0])
128
129     # creates the animation
130     plotting()
```

5.1.2 New time-step

As opposed to the framework, we have increased the time-step size just outside of the stability region. You only have to change line 113 into the following two.

```
1  dt = dx**2 / (2 * v)
2  dt *= 1.01
```

5.1.3 New initial and boundary condition

There are a couple of lines that need to be changed, with respect to the framework. To make things clearer we included the comments, so when the comments match you can change the code with following. The lines that have to be changed are: 26, 69, 107, 113 and 127.

```
1  # applies r(u)
2  f[0] += (1 / (4 * dx)) + v * (1 / (dx**2))
3
4  # Adds the known value of the Dirichlet boundary.
5  u = np.insert(u, 0, 1)
6
7  # initialize parameters
8  v, N, L, te = 0.01, 100, 3.0, 5
9
10 # sets time-step
11 dt = 0.003
```

```
12
13  # sets up the initial value of the problem
14  u = np.array([])
15  for x in np.arange(dx, L + dx / 3, dx):
16      if x <= L / 3:
17          u = np.append(u, 1)
18      elif L / 3 <= x <= 2 * L / 3:
19          u = np.append(u, 2 - (3 / L) * x)
20      else:
21          u = np.append(u, 0)
```

5.1.4 Upwind scheme

In order to implement an upwind scheme we only have to alter the matrices used to compute the approximation. Thus change the following lines 118–124 into

```
1  # constructs a sparse matrix W to calculate f(u)
2  diagonals_w = np.array([N * [-1], (N + 1) * [1], N * [0]])
3  W = -(1 / (2 * dx)) * diags(diagonals_w, [-1, 0, 1])
4
5  # constructs sparse matrix K
6  diagonals_k = np.array([(N - 1) * [1] + [2], (N + 1) * [-2], N * [1]])
7  K = (1 / dx**2) * diags(diagonals_k, [-1, 0, 1])
```

5.1.5 Determining threshold

To determine at which time a certain threshold is reached, put this code underneath all the other code you have so far and delete line 54.

```
1  # Determining when the threshold is hit.
2  threshold = 0.001
3  for i, value in enumerate(euler_forward(u, dt, 0, te)):
4      # We want to calculate the threshold for rho, and euler_forward
5      # returns a value for u thus we transform.
6      value = (value - 1) / -2
7
8      # Applies the Trapezium rule.
9      ans = sum(dx * value) - dx * (value[0] + value[-1]) / 2
10
11     if ans < threshold:
12         # Print the time at which the threshold is reached.
13         print(i * dt)
```

```
14
15          break
```

5.1.6 Download the resulting file

For the sake of completion we put the resulting Python file in a github repo, which you can find here: `https://github.com/yannickperrenet/numeriek`. This includes the upwind scheme as well as the threshold feature.

5.2 Definitions and theorems

5.2.1 Definitions

5.2.2 Theorems

We did not include the proofs to these theorems, but there are several books[2] that do.

Theorem 1 (Gershgorin circle theorem)**.** *The eigenvalues of a general $n \times n$ matrix* $\mathbf{A}$ *are located in the complex plane in the union of circles*

$$|z - a_{ii}| \leq \sum_{\substack{j \neq i \\ j=1}}^{n} |a_{ij}| \qquad \text{where } z \in \mathbb{C}$$

5.2.3 Results

To minimize calculations we used the following results.[3]

Result (Stability conditions)*.* In table 5.1 you can find the stability conditions of the used numerical time-integration methods.

method	stability condition ($\lambda \in \mathbb{R}, \lambda < 0$)
Euler forward	$\Delta t \leq -\frac{2}{\lambda}$

Table 5.1: Stability conditions.

Result (Trapezoidal rule)*.* Let $f \in C[x_L, x_R]$ then the following approximation holds

$$\int_{x_L}^{x_R} f(x)dx \approx \frac{x_R - x_L}{2} \left(f(x_L) + f(x_R)\right)$$

[2] One of these books is [1], a personal favorite if I might add.
[3] The calculations leading up to these results can be found in [1].

Bibliography

[1] C. Vuik, F.J. Vermolen, M.B. Gijzen, and M.J. Vuik. *Numerical Methods for Ordinary Differential Equations*. DAP, Delft Academic Press, 2015.