Problem Sheet 8

Exercise 1.1

We begin the discussion of the exercise with the functions for the symmetric and upwind numerical differential operators. The field with initial and boundary conditions is initialized using the following function:

```
# Function to create a vector that contains the discrete values of the
    field
def create_field(dimension, left_boundary, right_boundary, length):
  u = np.zeros((dimension + 2, 1))
  for i in range(1, dimension + 1):
    dist = i * dx - length / 2
    if dist < 0:
    u[i] = 1
  u[0] = left_boundary
  u[dimension + 1] = right_boundary
  return u, dimension
```

Function to initialize boundary and initial conditions on the field.

The implementation of the algorithms for the symmetric and upwind operators is shown below:

```
def symmetric_scheme(left_boundary, right_boundary):
  u, dim = create_field(100, left_boundary, right_boundary, L)
  v = velocity(shape='constant', dimension=dim)
  for i in range(0, timesteps):
    for j in range(1, dim):
      u[j] = u[j] - v[j] * (u[j+1] - u[j-1])/(2*dx) * dt
  return u
```

Symmetric numerical derivative code.

```
def upwind_scheme(left_boundary, right_boundary):
  u, dim = create_field(100, left_boundary, right_boundary, L)
  v = velocity(shape='constant', dimension=dim)
  for i in range(0, timesteps):
    for j in range(1, dim):
      u[j] = u[j] - v[j] * (u[j] - u[j - 1]) / dx * dt
  return u
```

Upwind numerical derivative code.

In figure 1, we can see the plots of this functions. One immediately sees the oscillation of the numerical instability for the symmetric schemes. Figure 2 displays the unstable upwind algorithm which was computed using the following function:

```
# Implementing the upwind scheme with the index shift
def upwind_scheme_reversed(left_boundary, right_boundary):
  u, dim = create_field(100, left_boundary, right_boundary, L)
  v = velocity(shape='constant', dimension=dim)
  for i in range(0, timesteps):
```

```
    for j in range(2, dim):
7       u[j] = u[j] - v[j] * (u[j + 1] - u[j]) / dx * dt
  return u
```

Upwind numerical derivative code with shifted indices. This will lead to numerical instabilities.

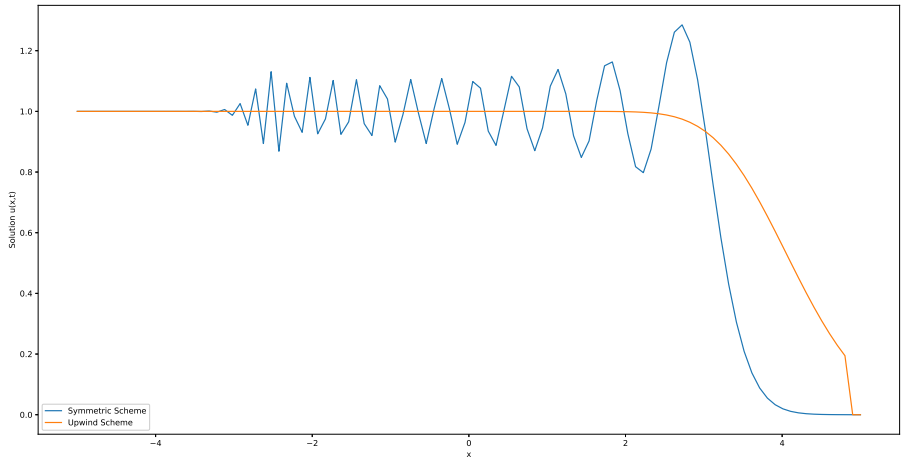

Figure 1: Plot of the behavior of the symmetric and upwind algorithm for 100 timesteps, right boundary = 1 and left boundary = 0.

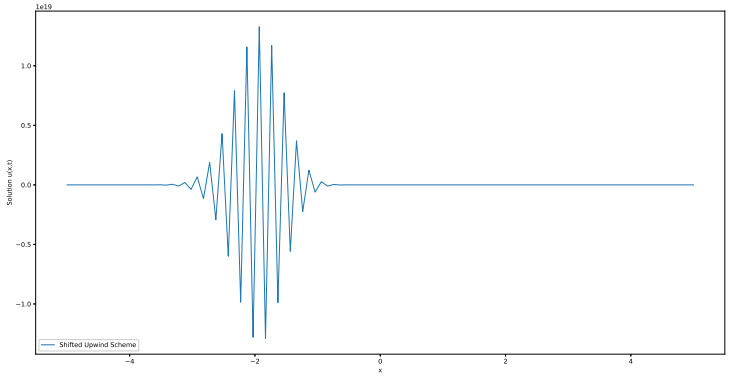

Figure 2: Plot of the unstable upwind algorithm. One can again immediately see the oscillatory pattern with very large amplitudes compared to the initial condition.

Now, we do some further analysis with the stable upwind function. We begin with setting the left boundary condition to $u/(-L/2, t) = 0.5$. After that, we do the same for with the right boundary set to 0.5. The result is shown in 3.
When the left boundary is modified, we see a constant piece at 0.5 followed by a smooth increase to the value 1 and a final decrease to the value 0. The first constant piece at 0.5 is due to the left boundary, because the boundary condition is propagated to the right. We can explain the increase to 1 by the initial condition and its propagation through time. The dip in the end arises from the fact that the initial condition and the information have not been propagated far enough.
In a similar manner, we can explain the constant piece at 1 for the modified right boundary. The peak in the end is due to the boundary condition. The constant piece is again due to the propagation of the initial condition and the boundary condition.

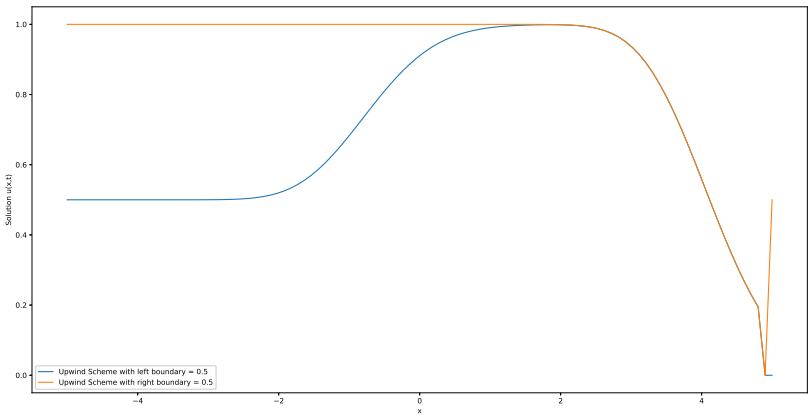

Figure 3: Plot of the stable upwind algorithm with different boundary conditions.

We can analyze both of these boundary cases for different time step amounts. The results are shown below. Figure 4 shows the evolution for 3 seconds sliced into 1000 time steps. We immediately see, that the information was propagated the same distance, which makes sense because the same amount of time has passed, but the result is much smoother, especially in the domain around $t = 3$.

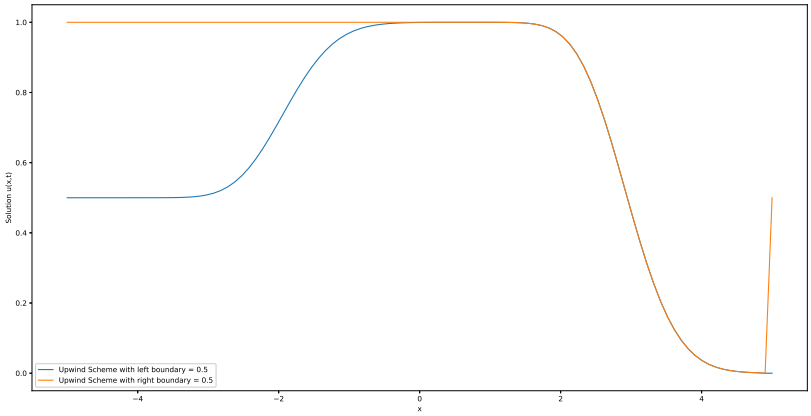

Figure 4: Plot of the stable upwind algorithm with 1000 time steps.

When we begin to decrease the step amount to 10 steps, we find an entirely different behavior, which is displayed for both boundary conditions in figures 5 and 6. In the domain around $t = 3$, we get and exponential increase(decrease) for the solution.
It is blowing up. The reason for this is that the *Courant-Friedrichs-Levy* criterion is no longer satisfied, because we have $h = \frac{L}{N}$, where $N = 100$ is the cell size and $L = 10$ is the length of the system. Then we see that with $v = 1$, the CFL-limit is given as $\Delta t \ll \frac{h}{v} = \frac{1}{10}$, while the systems time step size equals $\frac{3}{10}$.

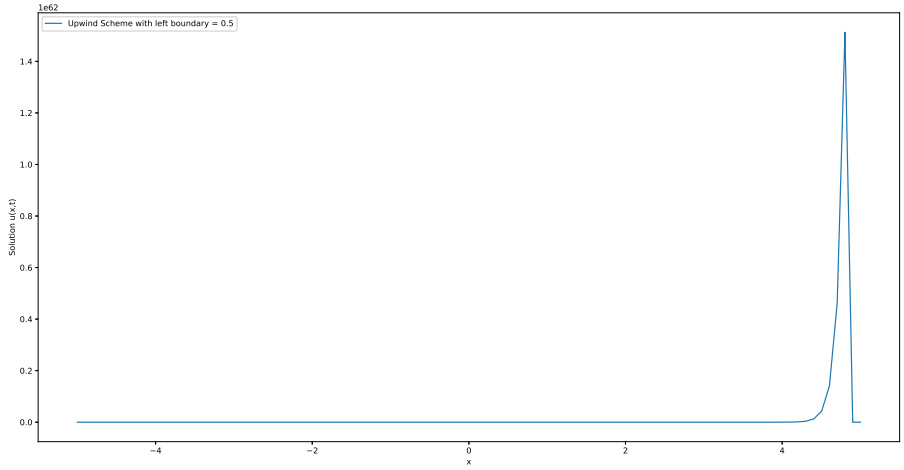

Figure 5:

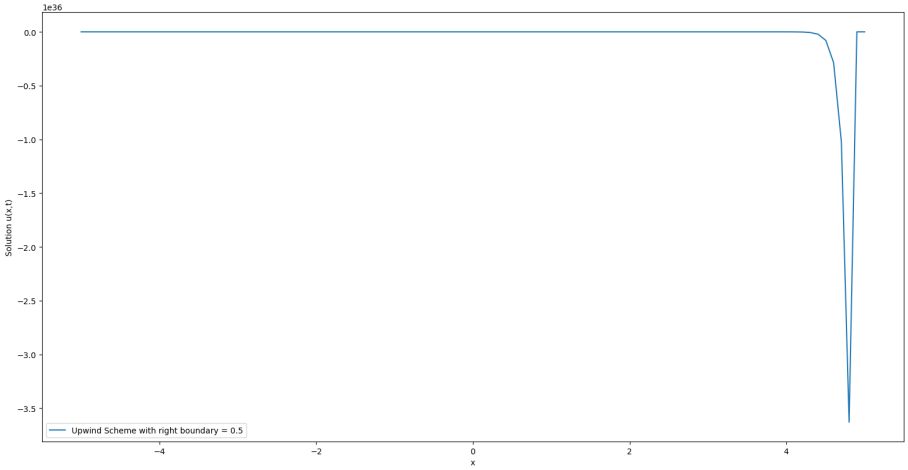
1e36
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
Solution u(x,t)
Upwind Scheme with right boundary = 0.5
−4
−2
0
2
4
x

Figure 6:

Exercise 1.2

The following scheme was used to allow for velocity fields with spatial dependence:

```
# Function to create the velocity field for Ex 1.2 with given initial
    conditions
def velocity(shape, dimension):
  v = np.zeros((dimension + 1, 1))
  if shape == 'constant':
    v = np.ones((dimension + 1, 1))
  if shape == 'linear_convergence':
    for i in range(0, dimension + 1):
  v[i] = -2*(i*dx - L/2) / L
  return v
```

Function that returns the velocity fields for the different cases.

```
# Function to perform an advection step in Ex1.2
def advect(u, dim, shape):
  v = velocity(shape='constant', dimension=dim)
  for j in range(1, dim):
    u[j] = u[j] - v[j] * (u[j] - u[j - 1]) / dx * dt
  return u
```

Implementation of a single advection step.

```
# Function to run the advection subroutine
def run_advection(shape, initial_condition=None):
  u, dim = create_field(100, 1, 0, L)
  if initial_condition == 'top_hat':
    u, dim = create_top_hat_field(100, L)
  for i in range(0, timesteps):
    u = advect(u, dim, shape)
  return u
```

Function to perform the advection steps.

If we use the initial and boundary conditions given in exercise 1.1, we arrive at the same output as can be seen in figure 7.

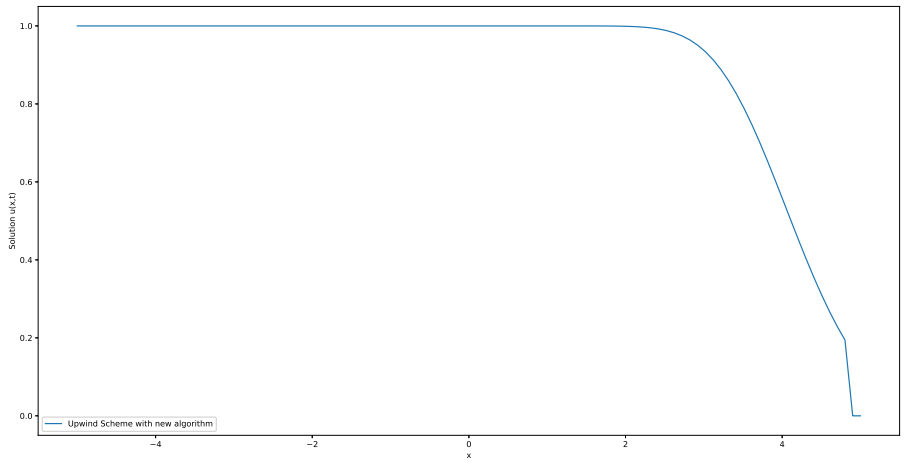

Figure 7: Ouput of the new general-purpose advection function for boundary conditions from exercise 1.1. It is the same as in figure 1.

Using the initial and boundary conditions given in exercise 1.2, we arrive at the time-evolution that is displayed in figures 8, 9 and 10. It shows that the top-hat function is evolved to the right in time. This is not a perfect fit though as the edges of the function are smoothed out. This smoothing-out effect comes from the leftmost and rightmost part of the initial condition, where the function is zero in the beginning. This information is also propagated upstream and causes the smoothing.

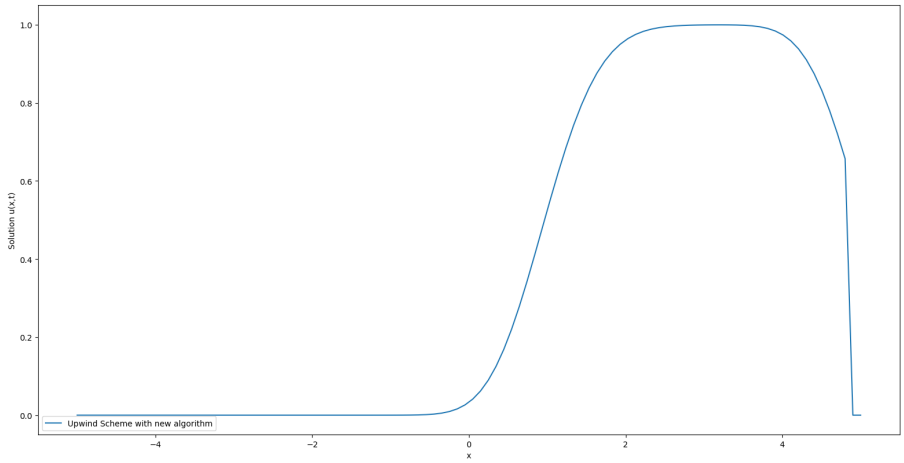

Figure 8: Plot of the general purpose advection subroutine shortly after the beginning.

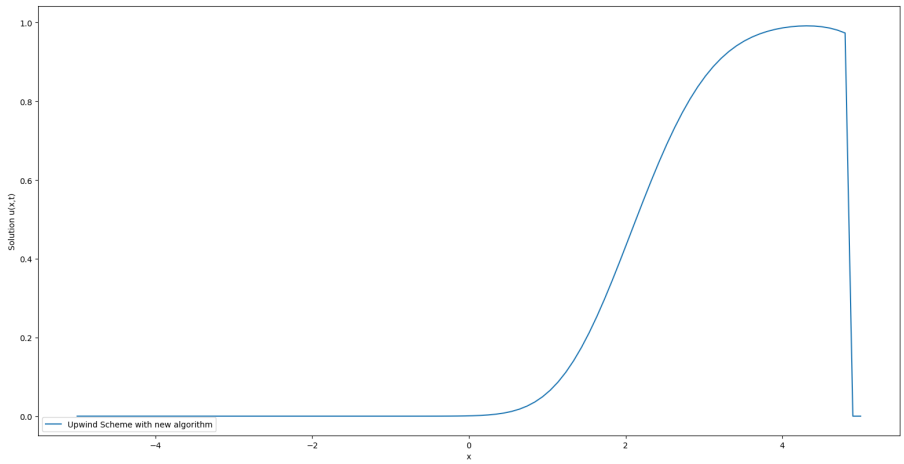

Figure 9: Plot of the general purpose advection subroutine after some time has passed.

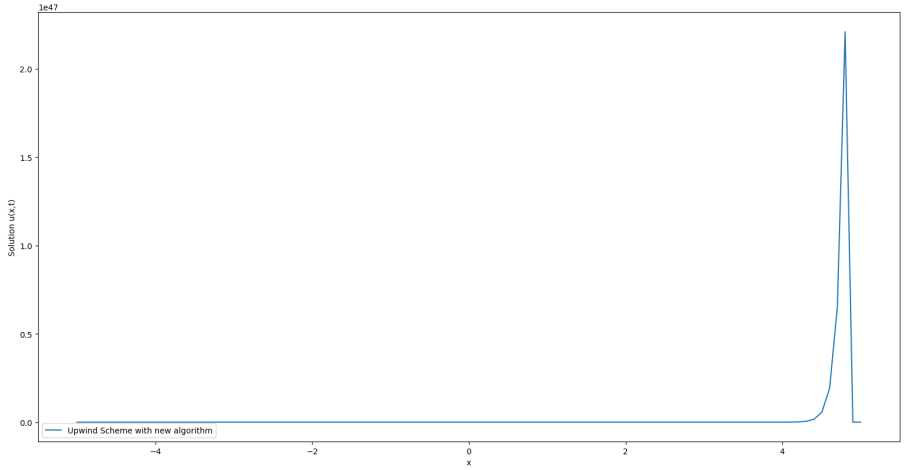

Figure 10: Plot of the general purpose advection subroutine shortly before it vanishes.