Problem Sheet 9

Exercise 1.1

To do the calculations in this exercise, the routine provided on the website was used due to its better implementation and performance. We chose the number of cells $N = 2000$ for all parts of Exercise 1.1.
The whole source code as well as the movie are provided in the attachments. Periodic boundary conditions have been implemented in the following way:

```
# Function to perform an advection step in Ex1.2 sheet 8
def advect(q, v, dx, dt):
  flux = np.zeros_like(v)
  ipos = np.where(v >= 0.)[0]
  ineg = np.where(v < 0.)[0]
  flux[ipos] = q[ipos]*v[ipos]
  flux[ineg] = q[ineg+1]*v[ineg]
  qnew = q.copy()
  qnew[1:-1] -= dt * (flux[1:] - flux[:-1]) / dx
  qnew[0] = qnew[-2]
  qnew[-1] = qnew[1]
  return qnew
```

Modification of the advect() function with periodic boundary conditions.

The following code was used to do solve the 1D-hydrodynamics problem with a fixed step-size.

```
def run_solver_fixed_dt():
  cs = 1  # Speed of sound is set equal to 1
  x = np.linspace(-L/2, L/2, N + 2)
  q = np.zeros((2, N + 2))
  q[0, :] = density_distribution(N, L)[:, 0]
  q[1, :] = initial_flux(N)[:, 0]

  snaps = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
  time = 0
  dt = 0.05
  configure_pyplot()
  for i in range(0, timesteps):
    q = hydro_iso_classic_one_timestep(q, cs, dx, dt)

    snap, new_dt = snapshot(time, snaps, dt)
    if snap:
      plt.plot(x, q[0, :], label='Plot of the density at ' + str(time +
          new_dt) + ' s')
    time += dt
  plt.legend(loc="upper right")
```

Symmetric numerical derivative code.

The snapshot function takes the current time and time-step as input as well as an array of the timestamps where we want to do a snapshot and then checks if the current time is close to a snapshot time and if the next step will overshoot this snapshot time.

```
# Function that calculates whether we want to have to do a snapshot
def snapshot(time, snaptimes, current_dt):
  snaptimes = np.array(snaptimes)
  index = np.where((snaptimes <= (time + current_dt)) & (time <= snaptimes)
      )[0]
  if snaptimes[index].size == 1:
    return True, np.abs(snaptimes[index] - time)[0]
  else:
    return False, 0
```

Function to check whether we have to do a snapshot or not.

In figure 1, we can see the plots of the resulting solution fo the problem. We immediately see the spurious oscillations after some around 70.0 s have passed and the waves are closing in on the boundary of the system. Reason for this is the limited grid resolution.

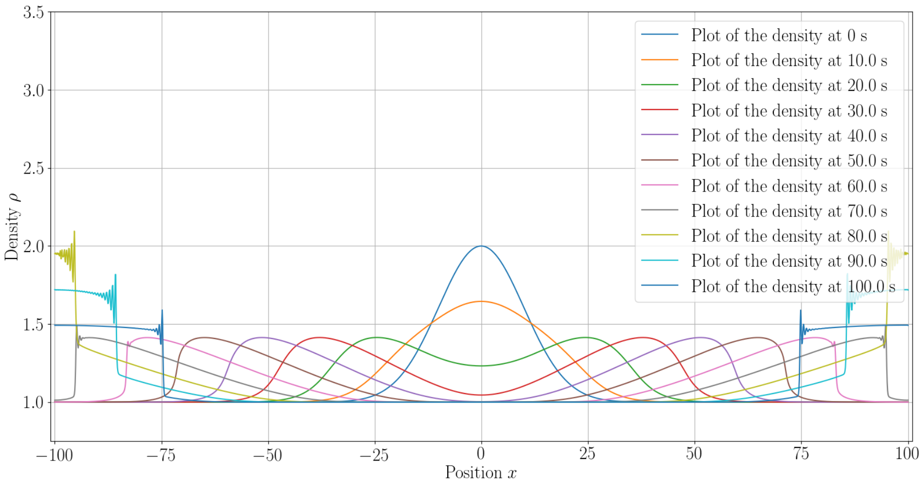

Figure 1: Plot of the solution for a fixed time-step size $dt = 0.1$. The time-step size meets the CFL-condition at all measured times and was determined experimentally.

Next, we implemented a solver for the same problem with a variable time-step.

```
def run_solver_variable_dt():
cs = 1  # Speed of sound is set equal to 1
x = np.linspace(-L/2, L/2, N + 2)
q = np.zeros((2, N + 2))
q[0, :] = density_distribution(N, L)[:, 0]
q[1, :] = initial_flux(N)[:, 0]
snaps = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
# snaps = np.linspace(0, 250, 400)

time = 0
```

```python
dt = 0.01
for i in range(0, timesteps):
  print("Time:", time, dt)
  snap, custom_dt = snapshot(time, snaps, dt)
  if snap:
    # evolve the system to the snapshot time and do a snapshot
    q = hydro_iso_classic_one_timestep(q, cs, dx, custom_dt)
    time += custom_dt
    plt.cla()

    configure_pyplot()

    plt.plot(x, q[0, :], 'b-', label='Plot of the density at ' + str(time)
        [0:5] + ' s')
    plt.legend(loc="upper right")
    plt.savefig('./movie/hydro_' + frame_index(i) + '.png')
    time += calculate_dt(q)
  else:
    q = hydro_iso_classic_one_timestep(q, cs, dx, dt)
    time += dt
  # Calculate next time-step
  dt = calculate_dt(q)
```

Function that runs the simple solver with a self-adjusting time-step.

The function *calculate_dt()* was used to get the time-step size based on the CFL condition:

```python
# Function to calculate the CFL value
def calculate_dt(q):
  if np.amax(np.abs(q[1, :])) != 0:
    cfl = dx / (np.amax(np.abs(q[1, :])))
    dt = 0.4 * cfl
  else:
    # If the velocity field is zero, start with this time-step
    dt = 0.1
  return dt
```

Function that runs the simple solver with a self-adjusting time-step.

The result of this code is depicted in figure 2.

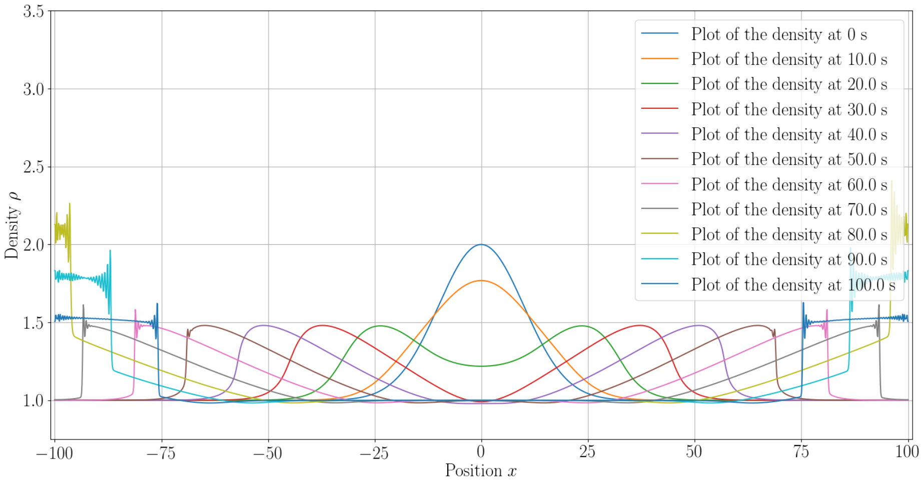

Figure 2: Plot of the algorithm with a time-step size based on the CFL condition.

If we reduce the spacing of the snap-array to a smaller value we can take more pictures and make a movie of them. For example, using

```
snaps = np.linspace(0, 150, 400)
```

we can make a movie with 400 frames evenly distributed over 150 seconds using *ffmpeg*.