Problem Sheet 5

Exercise 1.1

Comment of the author:
When programming this exercise, I realized, that it is much easier to consider a 30×30 grid reaching from 0 to $2H$ because this is much simpler to represent in Python.

Before we can start the exercise, we have to devise a scheme to determine the cell where the particle with position (X_i, Y_i) can be found. To do this, we use the relations

$$x_k = k + \frac{H}{K} \tag{1}$$

$$y_l = l + \frac{H}{K} \tag{2}$$

for the position of the center (x_k, y_l) of the cell (k, l). We can reshape these relations and use the floor() function to obtain the cell indices.

$$k = \left\lfloor X_i - \frac{H}{K} \right\rfloor \tag{3}$$

$$l = \left\lfloor Y_i - \frac{H}{K} \right\rfloor \tag{4}$$

This can be simplified further by assuming that the cell is enumerated by the lower left vertex of the cell, i.e. $(15.3, 4.9)$ is in the cell with the position $(15, 4)$. In that case, we can drop the $\frac{H}{K}$-part and write

$$k = \left\lfloor X_i \right\rfloor \tag{5}$$

$$l = \left\lfloor Y_i \right\rfloor \tag{6}$$

Or as python code:

```
def indices(X, Y):
  k = np.floor(X)
  j = np.floor(Y)
  return [int(k), int(j)]
```

Function to calculate the index.

Now that we know the cells where our particles are located, we obtain the numerical form of $\rho(x, y)$ by calculating

$$\rho_{kl} = \sum_{i=1}^{N} W_{kl}(X_i, Y_i) \tag{7}$$

where the weight functions are given through

$$W_{kl}(x, y) = \iint dxdy \Pi\Big(\frac{x - x_k}{2H/K}\Big) \Pi\Big(\frac{y - y_k}{2H/K}\Big) \delta(x - x_i)\delta(y - y_i) \tag{8}$$

$$= \Pi\Big(\frac{x_i - x_k}{2H/K}\Big) \Pi\Big(\frac{y_i - y_k}{2H/K}\Big) \tag{9}$$

This function is equal to 1 only when the particle is in th respective cell and 0 elsewhere.

Exercise 1.2

In the generalization to a 3×3-stencil, we can simply write the 0^{th} order as

$$\mathrm{w}[:,:] = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}. \tag{10}$$

Before we continue with the derivation for the 1^{st}- and 2^{nd}-order stencil, we introduce the following notation to make the derivation easier:

$$\epsilon_x = \frac{X_i - x_{k-1/2}}{x_{k+1/2} - x_{k-1/2}} \tag{11}$$

Exercise 1.2

and the same for ϵ_y in the y-direction. We will assume a cell length of 1 to increase readability. The following sketch 1 shows the geometric interpretation of this. We now

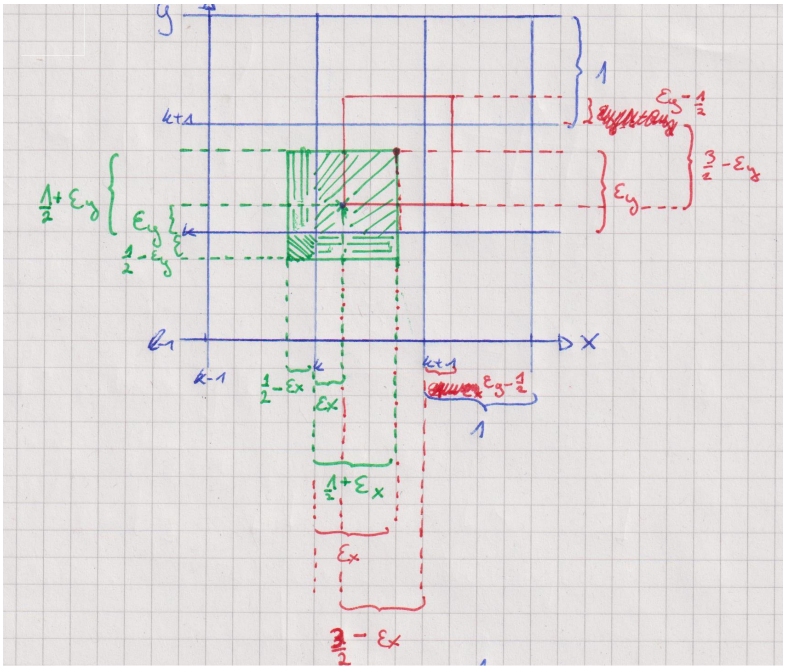

Figure 1: Geometric interpretation for the devised algortihm. The green triangle satisfies $\epsilon_x, \epsilon_y < \frac{1}{2}$, while the red triangle satisfies $\epsilon_x, \epsilon_y > \frac{1}{2}$. Depending on the size of the epsilons, different cells are filled. We also see that the formulas for the areas in the squares depend on the epsilons. The height of the cubucle is one and thus ignored.

begin with the 1^{st}-order derivation. To do this, we replace the δ-distribution of the 0-th order by a top-hat function. Thus the integral in (8) gives as a result the overlap

between cloud cubicle of the particle and the adjacent cells. The overlap can be simply expressed in terms of ϵ_x and ϵ_y. We immediately see that the cloud can overlap with 4 cells at most. The overlap for all four non-zero weights in the sketch reads:

$$
\begin{aligned}
W_{k,l} &= \left(\frac{1}{2} + \epsilon_x\right)\left(\frac{1}{2} + \epsilon_y\right) \\
W_{k-,l-1} &= \epsilon_x \epsilon_y \\
W_{k,l-1} &= \left(\frac{1}{2} + \epsilon_x\right)\left(\frac{1}{2} - \epsilon_y\right) \\
W_{k-1,l} &= \left(\frac{1}{2} - \epsilon_x\right)\left(\frac{1}{2} + \epsilon_y\right)
\end{aligned}
$$

All other weights are zero. The weights satisfy the normalization condition for their sum to be equal to unity. This works of course only if $\epsilon_x, \epsilon_y < \frac{1}{2}$. For the case of one or both of them being bigger, we have to replace some of the weights. To see how this is done, consider figure (). This algorithm is implemented in the following way:

```
def compute_1_order(X, Y, k, l):
  W = np.ones((3, 3))
  ex = calculate_e(X, k)
  ey = calculate_e(Y, l)
  if ex < 0.5:
    W[:, 0] *= H/K - ex
    W[:, 1] *= ex + H/K
    W[:, 2] *= 0
  else:
    W[:, 0] *= 0
    W[:, 1] *= 3*H/K - ex
    W[:, 2] *= ex - H/K

  if ey < 0.5:
    W[0, :] *= H/K - ey
    W[1, :] *= ey + H/K
    W[2, :] *= 0
  else:
    W[0, :] *= 0
    W[1, :] *= 3*H/K - ey
    W[2, :] *= ey - H/K
  return W
```

Function to calculate the 1st order.

In a similar manner, we can derive the solution for the 2^{nd}-order algorithm. First, we realize, that we no longer have a cubical cloud shape with base 1 but instead a regular pyramid with baseline 2 and height 1. Thus, when calculating the volume of the cloud contained in a specific cell we have to take this into account.
The following sketch 2 shows again how to derive the weight matrix for the 2^{nd}-order method. We simply project the x and the y-part of the pyramid into a 2D plane. There we find the area, that contributes to the respective square to be width times height or length times height.

Figure 2: Geometric interpretation for the second-order method. This time, the height depends on the epsilons. and we have to add an factor of $\frac{1}{2}$ when calculating the area projected into the x or y-direction due to the triangular shape.

To get the volume contained in the stencil cell, we multiply these two areas. For example, we multiply all cells in the row with index $k-1$ with the value $\frac{(1-\epsilon_x)^2}{2} = \frac{1}{2} - \epsilon_x + \frac{\epsilon_x^2}{2}$ and their respectiv y-direction contribution. From this, we can write an algorithm that calculates the stencil using:

```
def compute_2_order(X, Y, k, l):
  ex = calculate_e(X, k)
  ey = calculate_e(Y, l)
  W = np.ones((3, 3))
  W[:, 0] *= 0.5 - ex + 0.5*ex**2
  W[:, 1] *= 0.5 + ex - ex ** 2
  W[:, 2] *= 0.5 * ex**2
  W[0, :] *= 0.5 - ey + 0.5 * ey ** 2
  W[1, :] *= 0.5 + ey - ey ** 2
  W[2, :] *= 0.5 * ey ** 2
  return W
```

Function to calculate the 2nd order.

Exercise 1.3

After we obtained functions for the different methods, we use the *density_map()* function to calculate the density map:

```
def density_map(particle_list, order):
  d_map = np.zeros((K, K))
    for k in range(1, K-1):
      for j in range(1, K-1):
        for particle in particle_list:
        x, y = particle[0], particle[1]  # particle position
        if indices(x, y) == [k, j]:
          W = compute_order(x, y, k, j, order)
          d_map[np.ix_([j-1, j, j+1], [k-1, k, k+1])] += M/N * W
  return d_map
```

Function to calculate the 2nd order.

This gives a Numpy array that contains the density map. To check whether everything is centered and calculated accordingly, we check some plots for the different functions in 3, 4 and 5. We see that the particles are placed accordingly.

Next, we plot the same Gaussian distribution of 100 particles on the grid for different methods(see figures 6, 10 and 11). We immediately see the softening, that occurs when using higher orders.

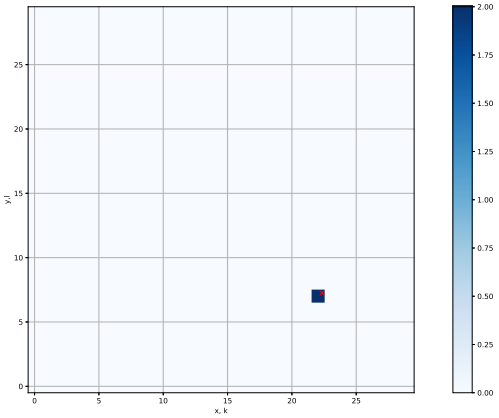

Figure 3: One particle on the grid with the 0^{th}-order method.

Lastly, we analyze the same for 10000 particles (figures 9, 10 and 11). When calculating the sum over the density matrix, we got the following results:

$$\sum_{kl} \rho^0_{kl} \approx 1.985 \tag{12}$$

$$\sum_{kl} \rho^1_{kl} \approx 1.996 \tag{13}$$

$$\sum_{kl} \rho^2_{kl} \approx 1.998 \tag{14}$$

This is in good agreement with expected value of 2.

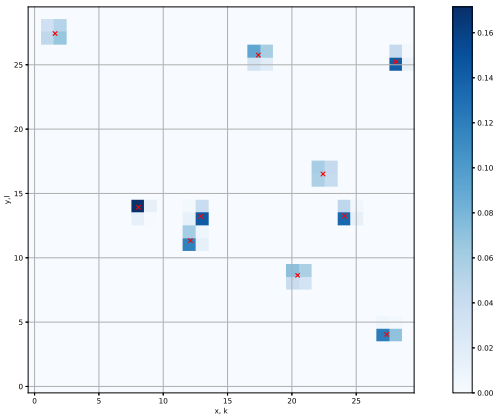

Figure 4: Ten particles on the grid with the 1^{st}-order method.

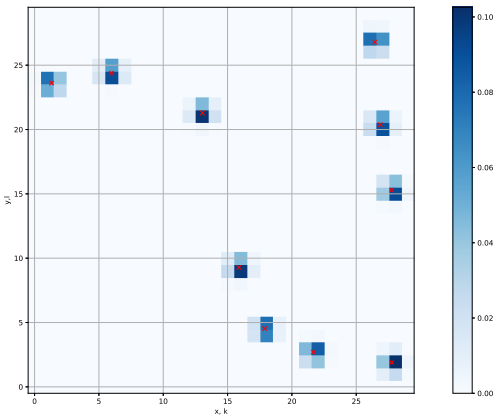

Figure 5: Ten particles on the grid with the 2^{nd}-order method.

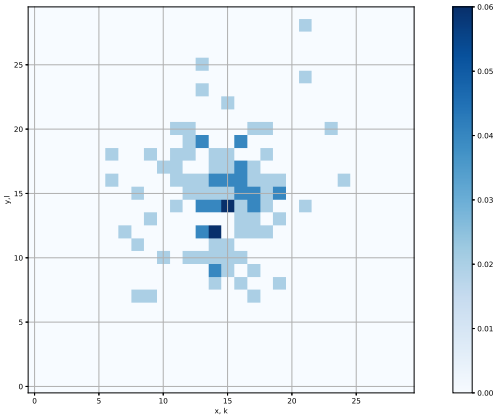

Figure 6: 100 particles on the grid with the 0^{th}-order method.

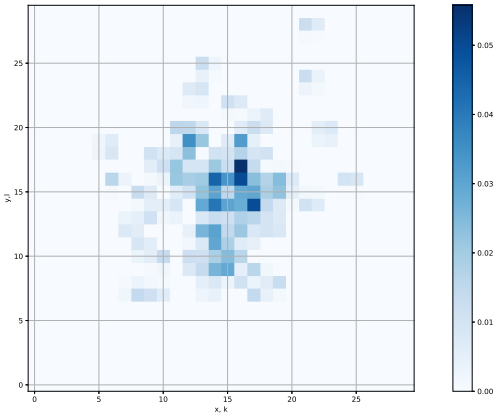

Figure 7: 100 particles on the grid with the 1^{st}-order method.

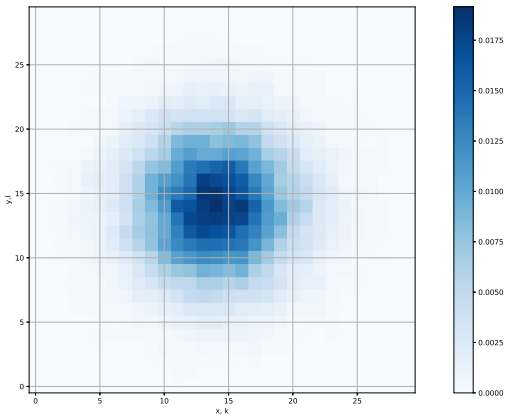

Figure 8: 100 particles on the grid with the 2^{nd}-order method.

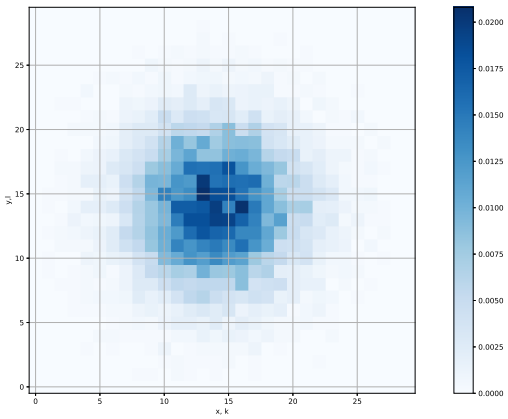

Figure 9: 10000 particles on the grid with the 0^{th}-order method.

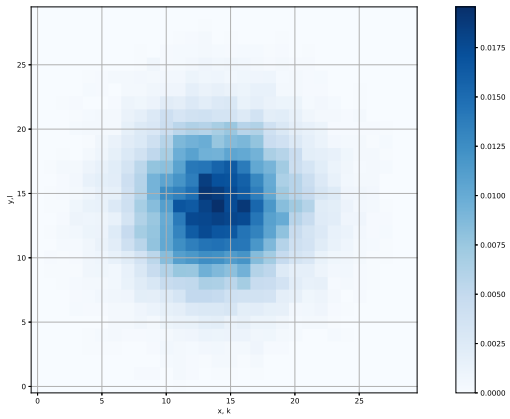

Figure 10: 10000 particles on the grid with the 1^{st}-order method.

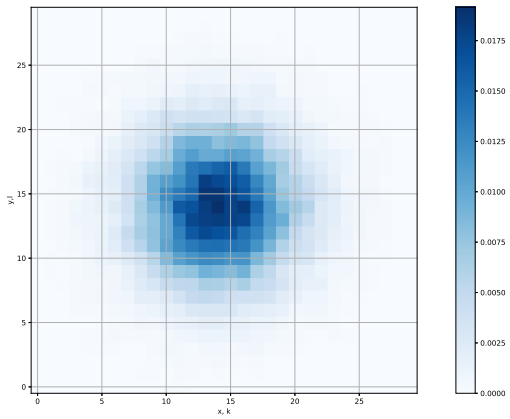

Figure 11: 10000 particles on the grid with the 2^{nd}-order method.