Sheet 2

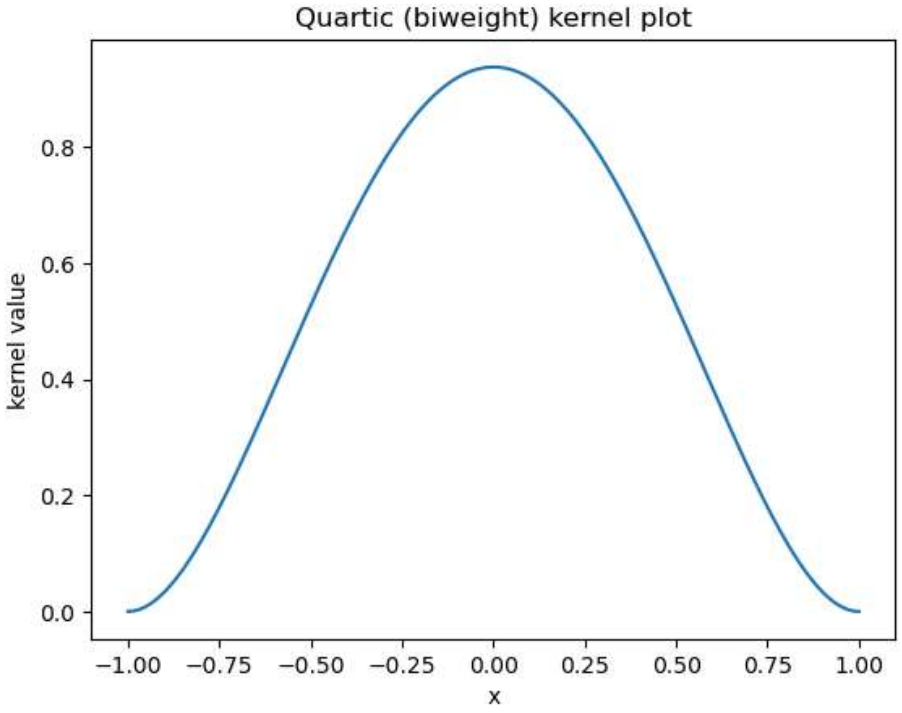
In [1]:
```
import numpy as np
from matplotlib import pyplot as plt
from scipy.stats import gaussian_kde
```

1 Kernel Density Estimation

(a)

In [2]:
```
def biweight(x, mu, w):
    """biweight kernel at mean mu, with bandwidth w evaluated
    if (abs(x - mu) <= w):
        return 15. * (1. - ((x - mu)/w)**2)**2 / (16. * w)
    else:
        return 0
    #TODO: implement the quartic (biweight) kernel
```

In [ ]:
```
# TODO plot the kernel
x = np.linspace(-1, 1, 1000)
y = [biweight(t, mu=0, w=1) for t in x]
fig, ax = plt.subplots()
ax.set_xlabel("x")
ax.set_ylabel("kernel value")
ax.set_title("Quartic (biweight) kernel plot")
ax.plot(x, y)
plt.show()
```



(b)

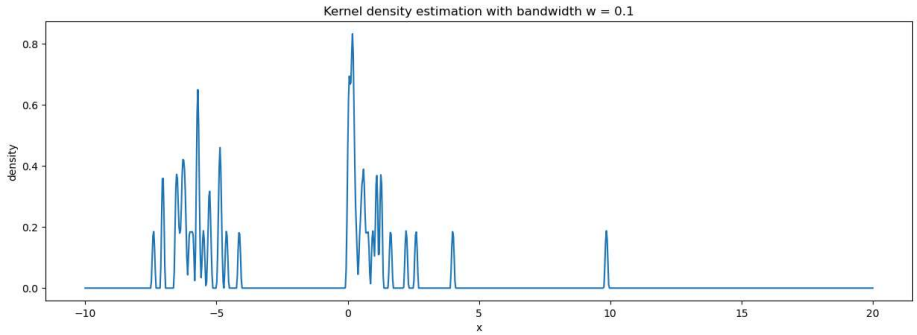
In [ ]:
```
# Load the data
data = np.load("data/samples.npy")
data50 = data[:50]
print(f'{data.shape=}, {data50.shape=}')
```

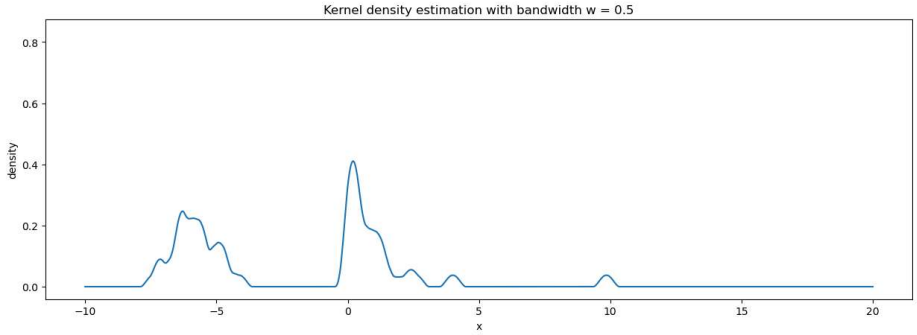
```
data.shape=(10000,), data50.shape=(50,)
```

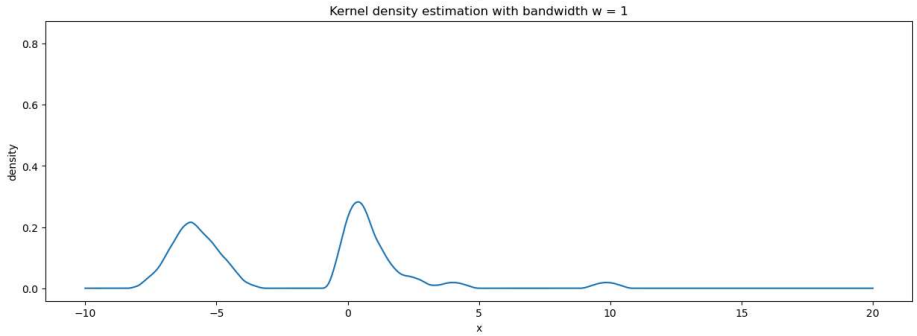
In [15]:
```
def kde(x, obs, w=1):
    return sum(biweight(x, data_x, w) for data_x in obs) / ler
    # TODO: implement the KDE with the biweight kernel
```

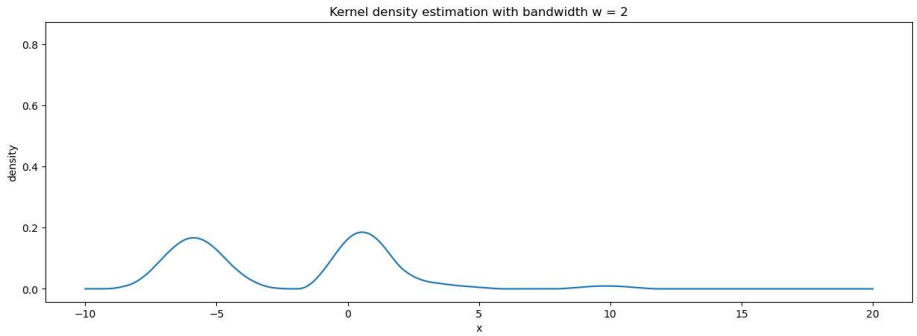
In [16]:
```
# TODO: compute and plot the kde on the first 50 data points
ws = [0.1, 0.5, 1, 2, 3, 5, 7]

fig, ax = plt.subplots(len(ws), 1, sharey='col')

for i, w in zip(range(len(ws)), ws):
    x = np.linspace(-10, 20, 1000)
    y = [kde(graph_x, data50, w) for graph_x in x ]
    ax[i].plot(x, y)
    ax[i].set_xlabel('x')
    ax[i].set_ylabel('density')
    ax[i].set_title(f'Kernel density estimation with bandwidth

fig.set_figwidth(15)
fig.set_figheight(40)
plt.show()

print("Interpretation: The bandwidths 0.1 and 0.5 are clearly
      "because the influence of certain data points is too lar
      "Plots with bandwidths 1, 2 and 3 are much smoother, big
      "Although with the increase of the bandwidth the slope c
      "Bandwidths 5 and 7 seem to be too large,\n"
      "in their respective graphs the underlying structure of
      "The optimal bandwidth is w=2, it preserves the structur
      "while being smooth enough to not have effects from sing
      )


# TODO: explore what happens when you increase the number of p
data_len = [100, 1000, 10000]

fig, ax = plt.subplots(len(ws), len(data_len), sharey='col')

for i, w in enumerate(ws):
    for j, N in enumerate(data_len):
        x = np.linspace(-10, 20, 100)
        data_curr = data[:N]
        y = [kde(graph_x, data_curr, w) for graph_x in x ]
        ax[i, j].plot(x, y)
        ax[i, j].set_xlabel('x')
        ax[i, j].set_ylabel('density')
        ax[i, j].set_title(f'Kernel density estimation\n with

fig.set_figwidth(15)
fig.set_figheight(60)
plt.show()
```

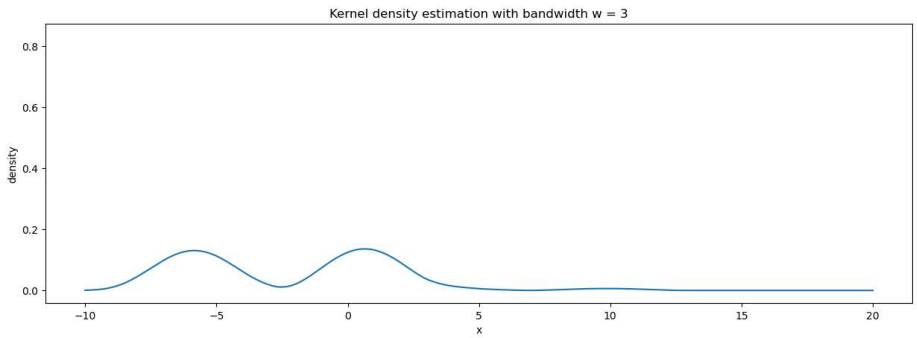
```
print("Interpretation: With the increase in the number of poir
      "For instance with the bandwidth w = 0.5 graph with 100
      "while the graph with 10000 samples with the same bandwi
      "Thus, a bigger number of data samples allows for a smal
      "On the bigger bandwidths the number of samples does not
      )
```

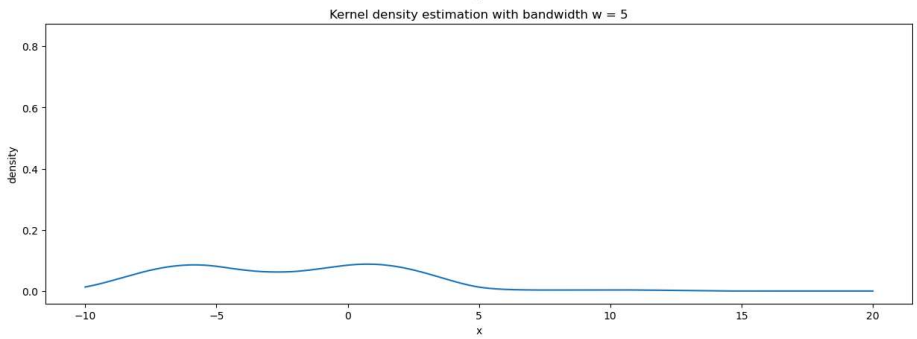

Kernel density estimation with bandwidth w = 0.1
density
x

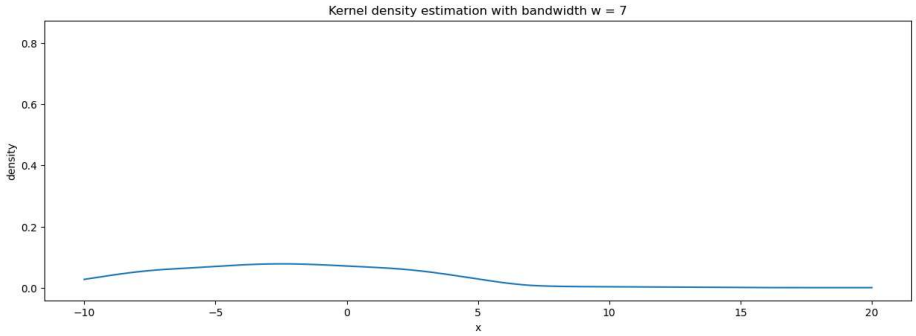

Kernel density estimation with bandwidth w = 0.5
density
x


Kernel density estimation with bandwidth w = 1
density
x

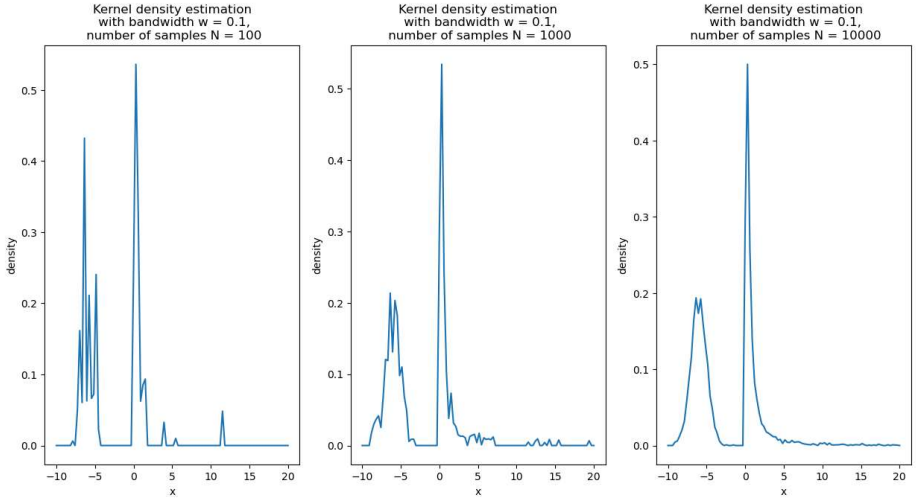

Kernel density estimation with bandwidth w = 2
density
x

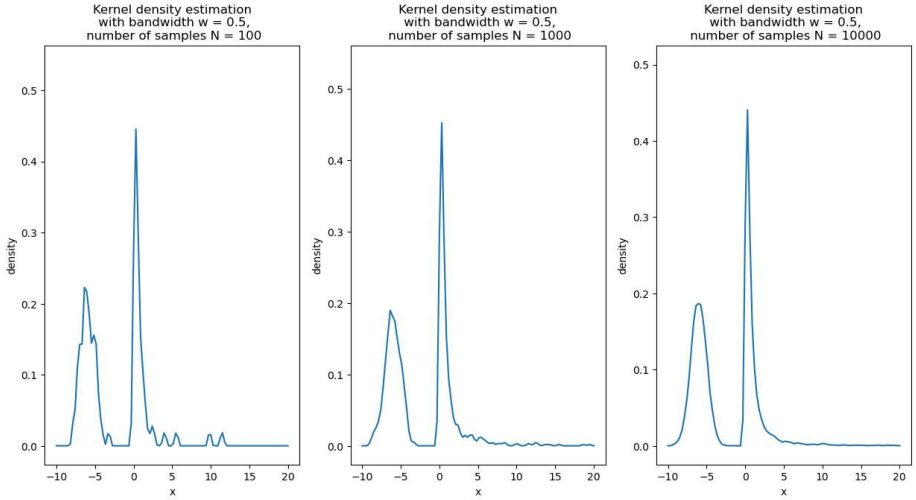

Kernel density estimation with bandwidth w = 3
density
x

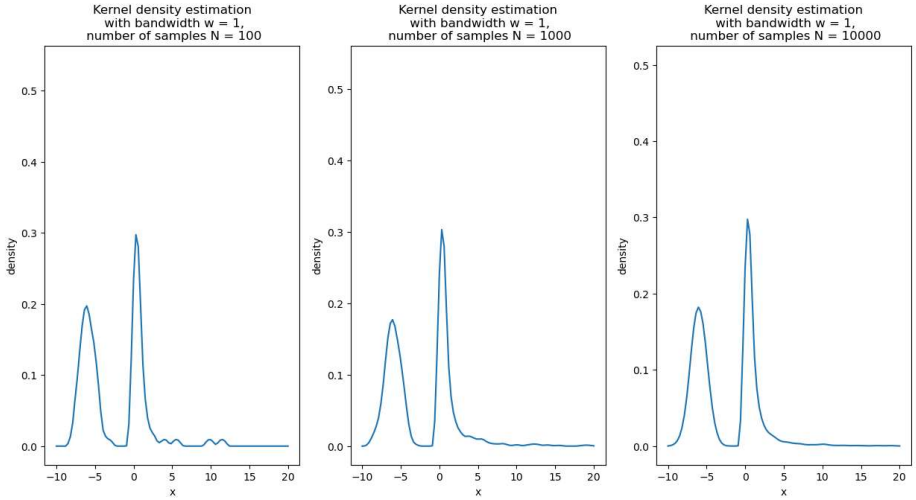

Kernel density estimation with bandwidth w = 5
density
x

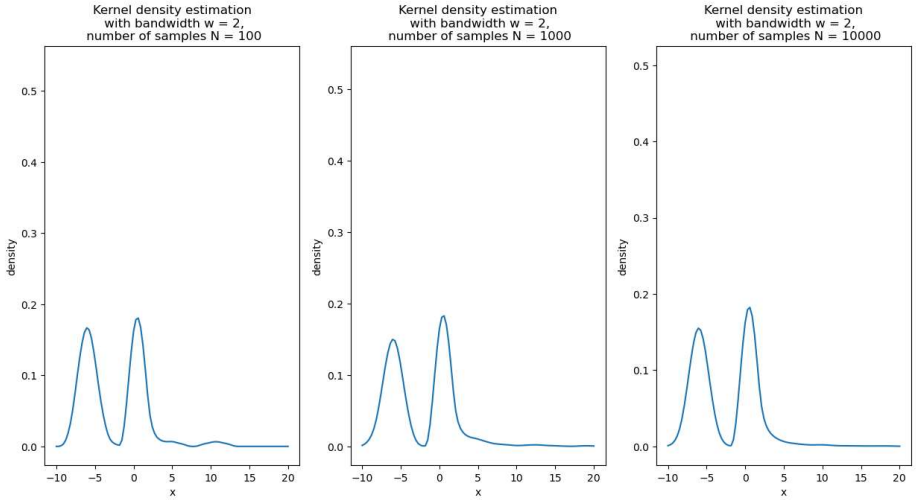


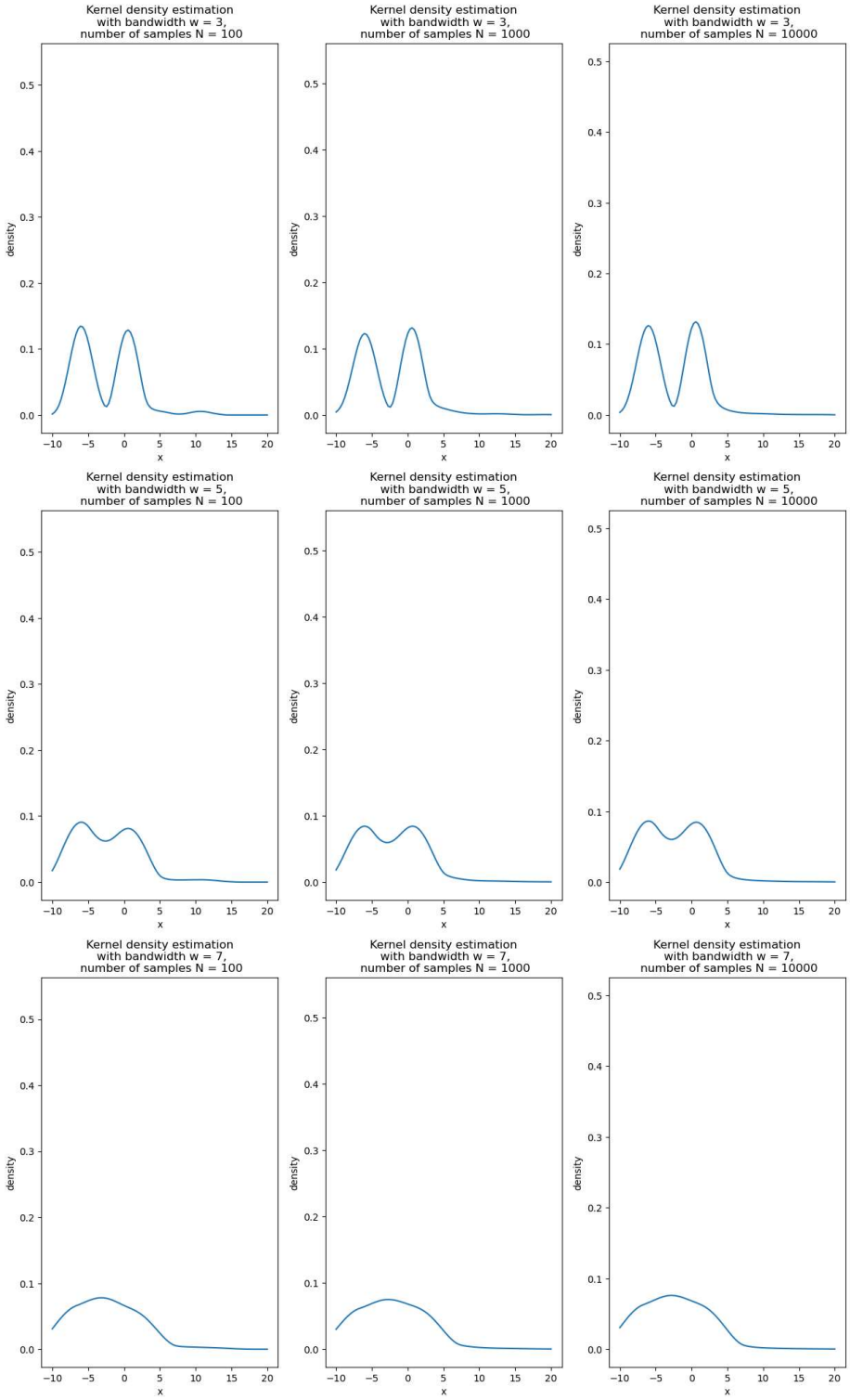
Interpretation: The bandwidths 0.1 and 0.5 are clearly too smal
l since the graphs look to spiky,
because the influence of certain data points is too large.
Plots with bandwidths 1, 2 and 3 are much smoother, bigger band
widths alleviate the effects of data artifacts.
Although with the increase of the bandwidth the slope of the "b
ells" becomes less steep.
Bandwidths 5 and 7 seem to be too large,
in their respective graphs the underlying structure of the dist
ribution (two bells) starts to disappear.
The optimal bandwidth is w=2, it preserves the structure of the
distribution and even the asymmetry of two bells,
while being smooth enough to not have effects from singular dat
a points.


Kernel density estimation
with bandwidth w = 0.1,
number of samples N = 100
Kernel density estimation
with bandwidth w = 0.1,
number of samples N = 1000
Kernel density estimation
with bandwidth w = 0.1,
number of samples N = 10000
density
x


Kernel density estimation
with bandwidth w = 0.5,
number of samples N = 100
Kernel density estimation
with bandwidth w = 0.5,
number of samples N = 1000
Kernel density estimation
with bandwidth w = 0.5,
number of samples N = 10000
density
x


Kernel density estimation
with bandwidth w = 1,
number of samples N = 100
Kernel density estimation
with bandwidth w = 1,
number of samples N = 1000
Kernel density estimation
with bandwidth w = 1,
number of samples N = 10000
density
x


Kernel density estimation
with bandwidth w = 2,
number of samples N = 100
Kernel density estimation
with bandwidth w = 2,
number of samples N = 1000
Kernel density estimation
with bandwidth w = 2,
number of samples N = 10000
density
x



Interpretation: With the increase in the number of points the g
raphs become smoother on smaller bandwidths.
For instance with the bandwidth w = 0.5 graph with 100 samples
still has small spikes,
while the graph with 10000 samples with the same bandwidth is a
lmost completely smooth.
Thus, a bigger number of data samples allows for a smaller band
width, resulting in a more precise estimation.
On the bigger bandwidths the number of samples does not change
the graph.

3 Mean-Shift

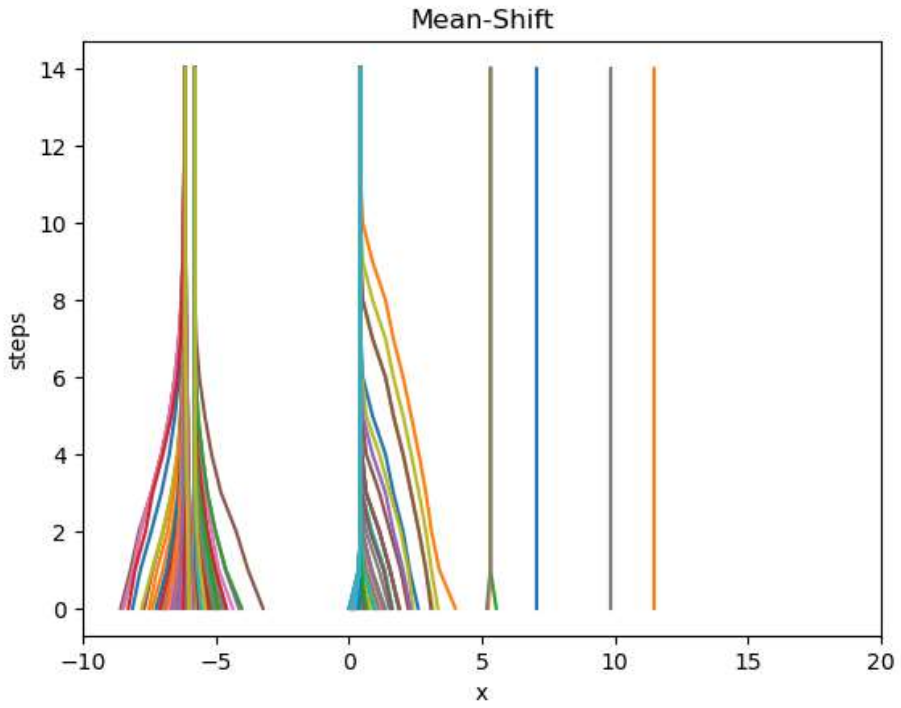
(b)

In [ ]:
```
# For 3(a) see pdf
```

In [54]:
```
# TODO: implement the update to the local mean

def mean_shift_step(x, xt, r=1):
    """
    A single step of mean shift, moving every point in xt to t

    Parameters
    ----------
    x : np.ndarray
        Array of points underlying the KDE, shape (d, N1)
    xt : np.ndarray
        Current state of the mean shift algorithm, shape (d, N
    n_components : int, optional
        Number of requested components. By default returns all

    Returns
    -------
    np.ndarray
        the points after the mean-shift step
    """
    # NOTE: For the excercise you only need to implement this
    #       If you want some extra numpy-practice, implement i

    assert xt.shape[0] == x.shape[0], f'Shape mismatch: {x.sha

    # TODO: start by computing a N by N matrix 'dist' of dista
    #       such that dists[i, j] is the distance between x[i]

    dist = np.zeros((x.shape[0], xt.shape[0]))

    for i, x_i  in enumerate(x):
        for j, xt_j in enumerate(xt):
            dist[i][j] = np.linalg.norm(x_i - xt_j)


    local_means = np.zeros_like(xt)
    for j in range(len(xt)):
        sum_val = 0
        sum_num = 0
        for i in range(len(x)):
            if dist[i][j] < r:
                sum_val += x[i]
                sum_num += 1
        local_means[j] = sum_val / sum_num

    return local_means
```

In [ ]:
```
# Load the data
data = np.load("data/samples.npy")
x = data[:200]  # use e.g. the first 200 points
xt = x
```

```
trajectories = [xt]
max_steps = 100
for step in range(max_steps):

    # TODO: update xt with your mean shift step
    xt = mean_shift_step(x, xt)
    trajectories.append(xt)
    if np.allclose(trajectories[-1], trajectories[-2]):  # bre
        break
trajectories = np.stack(trajectories)
n_steps = len(trajectories) - 1

fig, ax = plt.subplots()

for i in range(trajectories.shape[1]):
    ax.plot(trajectories[:, i], np.array(range(trajectories.sh

ax.set_xlim(-10, 20)
ax.set_xlabel('x')
ax.set_ylabel('steps')
ax.set_title('Mean-Shift')

plt.show()


# TODO: plot the trajectories
```



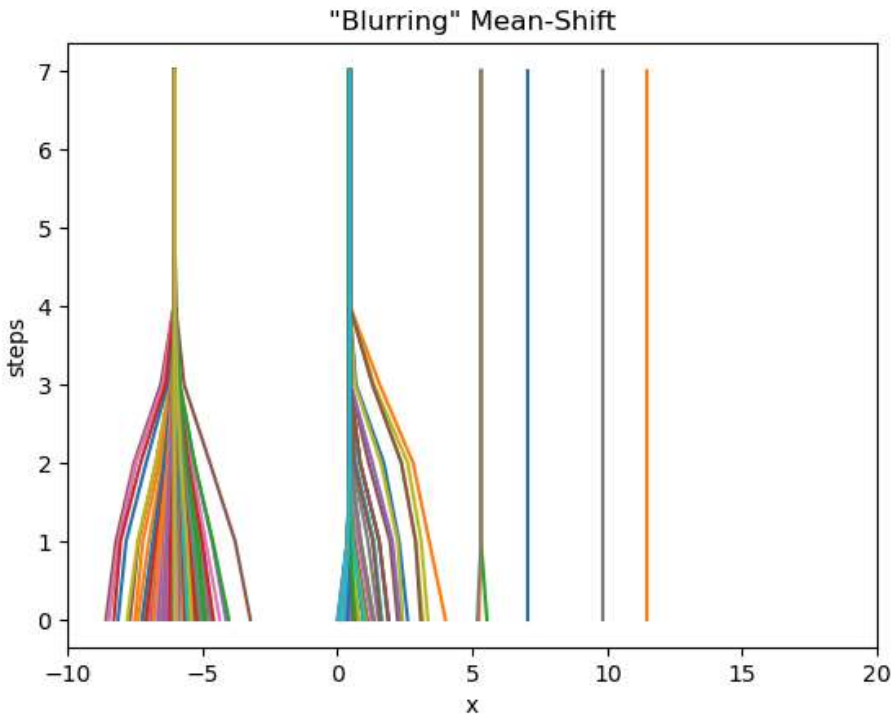
In [ ]:
```
# TODO: implement the update to the local mean

def mean_shift_step_blurry(xt, r=1):
    """
    A single step of mean shift, moving every point in xt to t

    Parameters
    ----------
    xt : np.ndarray
        Current state of the mean shift algorithm, shape (d, N
```

```
    Returns
    -------
    np.ndarray
        the points after the mean-shift step
    """
    # NOTE: For the excercise you only need to implement this
    #       If you want some extra numpy-practice, implement i

    # TODO: start by computing a N by N matrix 'dist' of disto
    #       such that dists[i, j] is the distance between x[i]

    dist = np.zeros((xt.shape[0], xt.shape[0]))

    for i, xt_i  in enumerate(x):
        for j, xt_j in enumerate(xt):
            dist[i][j] = np.linalg.norm(xt_i - xt_j)


    local_means = np.zeros_like(xt)
    for j in range(len(xt)):
        sum_val = 0
        sum_num = 0
        for i in range(len(xt)):
            if dist[i][j] < r:
                sum_val += xt[i]
                sum_num += 1
        local_means[j] = sum_val / sum_num

    return local_means
```

In [64]:

```
# Load the data
data = np.load("data/samples.npy")
x = data[:200]  # use e.g. the first 200 points
xt = x

trajectories = [xt]
max_steps = 100
for step in range(max_steps):
    # TODO: update xt with your mean shift step
    xt = mean_shift_step_blurry(xt)
    trajectories.append(xt)
    if np.allclose(trajectories[-1], trajectories[-2]):  # bre
        break
trajectories = np.stack(trajectories)
n_steps = len(trajectories) - 1

fig, ax = plt.subplots()

for i in range(trajectories.shape[1]):
    ax.plot(trajectories[:, i], np.array(range(trajectories.sh

ax.set_xlim(-10, 20)
ax.set_xlabel('x')
ax.set_ylabel('steps')
ax.set_title('\"Blurring\" Mean-Shift')
plt.show()
```

```
# TODO: plot the trajectories
```



Convergence of the "blurring" mean-shift is faster than the regular mean-shift. In the regular mean-shift the leftmost cluster was split into two, because it had two close local means, the blurring mean-shift, however, handled that and all the points in that cluster converged to one point.

Other than that the clusters are the same, we see two big clusters which coincide with the "bells" of the estimated density function and four small clusters which my be considered as noise/outliers.

3(a)

$$x_j^{t+1} = x_j^t + \alpha_j^t \frac{2}{n} \sum_{i:\|x_i - x_j^t\| < 1} (x_i - x_j^t) \quad \text{— gradient ascent} \quad (1)$$

local mean $\tilde{x}_j^{t+1} = \dfrac{\sum_{i:\|x_i - x_j^t\|<1} x_i}{\sum_{i:\|x_i - x_j^t\|<1} 1}$

$$\frac{\sum_{i:\|x_i - x_j^t\|<1} x_i}{\sum_{i:\|x_i - x_j^t\|<1} 1} = x_j^t + \alpha_j^t \frac{2}{n} \left(\sum_{i:\|x_i - x_j^t\|<1} x_i - x_j^t \cdot \sum_{i:\|x_i - x_j^t\|<1} 1 \right)$$

$$\alpha_j^t = \frac{\sum_{i:\|x_i - x_j^t\|<1} x_i - x_j \sum_{i:\|x_i - x_j^t\|<1} 1}{\sum_{i:\|x_i - x_j^t\|<1} 1} \cdot \frac{1}{\sum_{i:\|x_i - x_j^t\|<1} x_i - x_j^t \sum_{i:\|x_i - x_j^t\|<1} 1} \cdot \frac{n}{2}$$

$$\alpha_j^t = \frac{n}{2 \cdot \sum_{i:\|x_i - x_j^t\|<1} 1}$$

When substituting α_j^t with this fraction, the last term of (1) looks like this:

$$\frac{\sum_{i:\|x_i - x_j^t\|<1} (x_i - x_j^t)}{\sum_{i:\|x_i - x_j^t\|<1} 1}$$

which is the mean vector of the neighbourhood — a sensible choice