1)

In [2]:

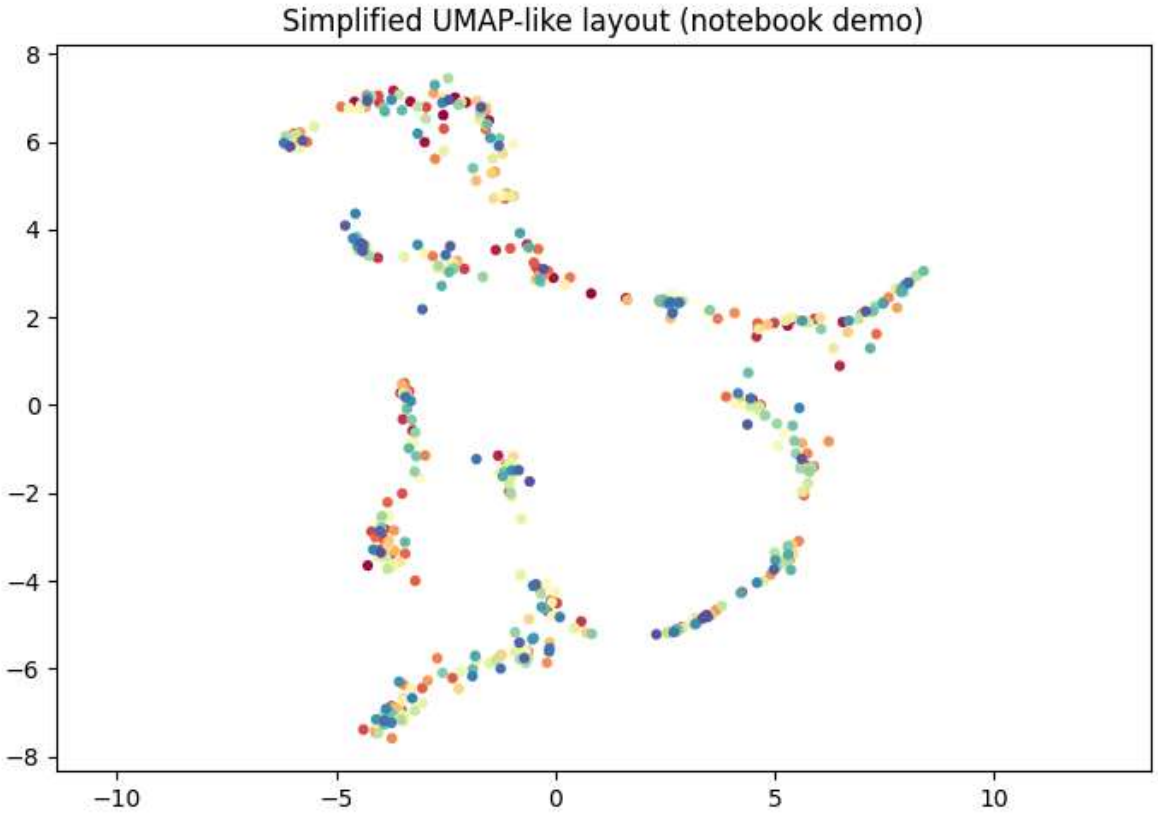
```python
# Implementation: simplified UMAP-like layout (analytic gradients)

from __future__ import annotations

import math
import random
from typing import Tuple

import numpy as np
from sklearn.neighbors import NearestNeighbors


def build_symmetrized_knn(X: np.ndarray, k: int = 15) -> np.ndarray:
    """Compute k nearest neighbor indices for each point and return indices arra

    Note: the returned array does not include the point itself.
    """
    nbrs = NearestNeighbors(n_neighbors=k + 1, algorithm="auto").fit(X)
    distances, indices = nbrs.kneighbors(X)
    # drop self at column 0
    return indices[:, 1:]


def symmetrize_edges(knn_idx: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
    """Symmetrize directed kNN edges and return arrays of endpoints (i_idx, j_id

    Each undirected edge appears once with i < j.
    """
    N, k = knn_idx.shape
    edges = set()
    for i in range(N):
        for j in knn_idx[i]:
            a, b = (i, int(j))
            if a == b:
                continue
            if a > b:
                a, b = b, a
            edges.add((a, b))
    edges = sorted(edges)
    if not edges:
        return np.zeros((0,), dtype=int), np.zeros((0,), dtype=int)
    i_idx = np.array([e[0] for e in edges], dtype=int)
    j_idx = np.array([e[1] for e in edges], dtype=int)
    return i_idx, j_idx


def sample_random_pairs(N: int, m: int, exclude_pairs: set | None = None, rng: r
    """Sample m random unordered pairs (i<j) from range(N), avoiding exclude_pai

    Returns two integer arrays (i_idx, j_idx) each of length <= m.
    """
    if rng is None:
        rng = random
    exclude_pairs = exclude_pairs or set()
    pairs = set()
```

```
    max_pairs = N * (N - 1) // 2
    if m >= max_pairs - len(exclude_pairs):
        # take all possible pairs except excluded
        for i in range(N):
            for j in range(i + 1, N):
                if (i, j) in exclude_pairs:
                    continue
                pairs.add((i, j))
        pairs = list(pairs)
        rng.shuffle(pairs)
        pairs = pairs[:m]
        if not pairs:
            return np.zeros((0,), dtype=int), np.zeros((0,), dtype=int)
        i_idx = np.array([p[0] for p in pairs], dtype=int)
        j_idx = np.array([p[1] for p in pairs], dtype=int)
        return i_idx, j_idx

    attempts = 0
    while len(pairs) < m:
        i = rng.randrange(0, N)
        j = rng.randrange(0, N - 1)
        if j >= i:
            j = j + 1
        a, b = (i, j) if i < j else (j, i)
        if (a, b) in exclude_pairs:
            attempts += 1
            if attempts > m * 10:
                # fallback to enumerating remaining pairs
                for ii in range(N):
                    for jj in range(ii + 1, N):
                        if (ii, jj) in exclude_pairs:
                            continue
                        pairs.add((ii, jj))
                        if len(pairs) >= m:
                            break
                    if len(pairs) >= m:
                        break
                break
            continue
        pairs.add((a, b))

    pairs = list(pairs)
    i_idx = np.array([p[0] for p in pairs], dtype=int)
    j_idx = np.array([p[1] for p in pairs], dtype=int)
    return i_idx, j_idx


def layout_force_directed(
    X: np.ndarray,
    dim: int = 2,
    k: int = 15,
    R_mult: int = 5,
    c: float = 10.0,
    epochs: int = 400,
    lr: float = 1.0,
    seed: int | None = None,
    verbose: bool = True,
) -> np.ndarray:
    """Run the simplified UMAP-like optimization and return low-dim positions Z.
```

```
    The implementation uses analytic gradients and resamples R each epoch.
    """
    rng = np.random.default_rng(seed)
    N = X.shape[0]
    knn_idx = build_symmetrized_knn(X, k=k)
    i_e, j_e = symmetrize_edges(knn_idx)

    edge_set = set((int(a), int(b)) for a, b in zip(i_e, j_e))

    Z = 0.01 * rng.standard_normal((N, dim))

    R_size = int(R_mult * N)
    if verbose:
        print(f"N={N}, edges={len(i_e)}, R_size={R_size}, epochs={epochs}")

    for epoch in range(epochs):
        t = epoch / max(1, epochs - 1)
        eta = lr * (1.0 - 0.9 * t)

        grads = np.zeros_like(Z)

        # Attractive gradients
        if len(i_e) > 0:
            diffs = Z[i_e] - Z[j_e]
            d2 = np.sum(diffs * diffs, axis=1)
            denom = 1.0 + d2
            coeff = 2.0 / denom
            coeff = coeff.reshape(-1, 1)
            gi = coeff * diffs
            np.add.at(grads, i_e, gi)
            np.add.at(grads, j_e, -gi)

        # Repulsive pairs resampled each epoch
        ri, rj = sample_random_pairs(N, R_size, exclude_pairs=edge_set, rng=rand
        if len(ri) > 0:
            rdiffs = Z[ri] - Z[rj]
            rd2 = np.sum(rdiffs * rdiffs, axis=1)
            rden = (1.0 + rd2) ** 2
            rcoeff = -2.0 * c / rden
            rcoeff = rcoeff.reshape(-1, 1)
            rgi = rcoeff * rdiffs
            np.add.at(grads, ri, rgi)
            np.add.at(grads, rj, -rgi)

        Z -= eta * grads

        if verbose and (epoch % max(1, epochs // 10) == 0 or epoch == epochs - 1
            E_attr = 0.0
            if len(i_e) > 0:
                d2 = np.sum((Z[i_e] - Z[j_e]) ** 2, axis=1)
                E_attr = np.sum(np.log1p(d2))
            if len(ri) > 0:
                rd2 = np.sum((Z[ri] - Z[rj]) ** 2, axis=1)
                E_rep = np.sum(c / (1.0 + rd2))
            else:
                E_rep = 0.0
            print(f"epoch {epoch+1}/{epochs}: eta={eta:.4f}, E_attr={E_attr:.4f}

    return Z
```

In [3]:
```
# Demo: generate data, run layout, and plot result

# matplotlib inline for Jupyter
%matplotlib inline

from sklearn.datasets import make_swiss_roll
import matplotlib.pyplot as plt

# Create a moderate-size dataset for the notebook demo
X, _ = make_swiss_roll(n_samples=500, noise=0.1)

# Run simplified UMAP layout (this cell may take a little time depending on epoc
Z = layout_force_directed(X, dim=2, k=15, R_mult=5, c=10.0, epochs=200, lr=0.5,

# Plot
plt.figure(figsize=(7, 5))
plt.scatter(Z[:, 0], Z[:, 1], s=12, c=np.arange(Z.shape[0]), cmap="Spectral")
plt.title("Simplified UMAP-like layout (notebook demo)")
plt.axis('equal')
plt.tight_layout()
plt.show()
```

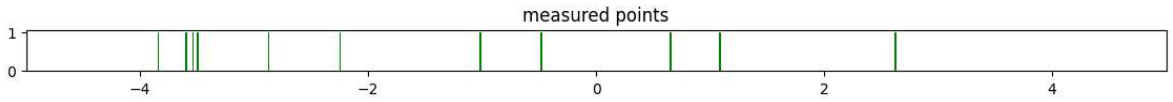
```
N=500, edges=4318, R_size=2500, epochs=200
epoch 1/200: eta=0.5000, E_attr=5146.5979, E_rep=7845.5083
epoch 21/200: eta=0.4548, E_attr=9260.1773, E_rep=1251.2211
epoch 41/200: eta=0.4095, E_attr=8319.2187, E_rep=986.9747
epoch 61/200: eta=0.3643, E_attr=7703.3698, E_rep=989.0607
epoch 81/200: eta=0.3191, E_attr=7524.9539, E_rep=804.1073
epoch 101/200: eta=0.2739, E_attr=6721.4429, E_rep=843.7179
epoch 121/200: eta=0.2286, E_attr=6228.2542, E_rep=851.4307
epoch 141/200: eta=0.1834, E_attr=5215.0602, E_rep=847.7466
epoch 161/200: eta=0.1382, E_attr=4189.6373, E_rep=925.2879
epoch 181/200: eta=0.0930, E_attr=3143.0645, E_rep=960.6862
epoch 200/200: eta=0.0500, E_attr=2169.9199, E_rep=1079.6674
```

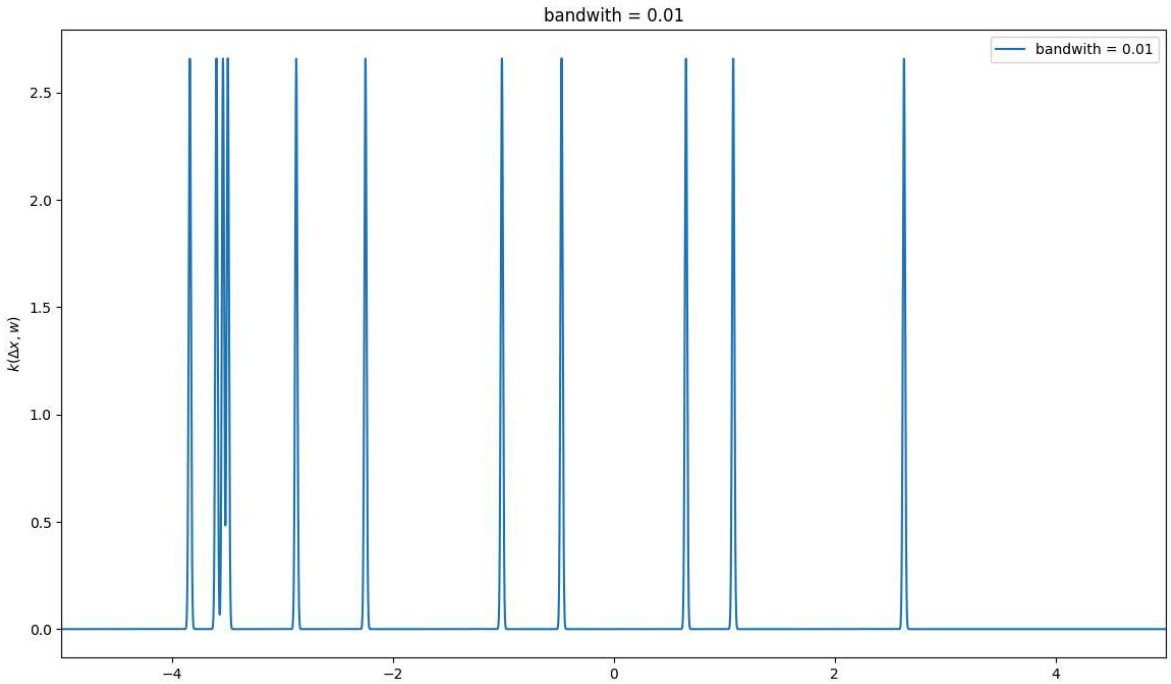


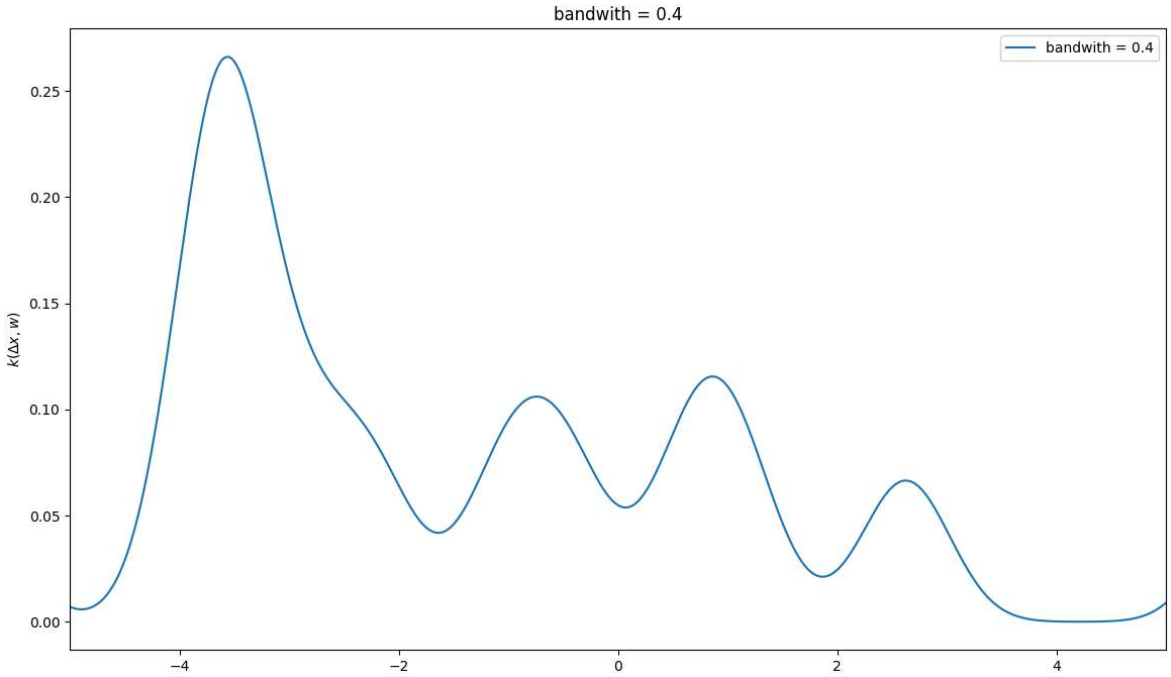
2 Kernel Density Estimation

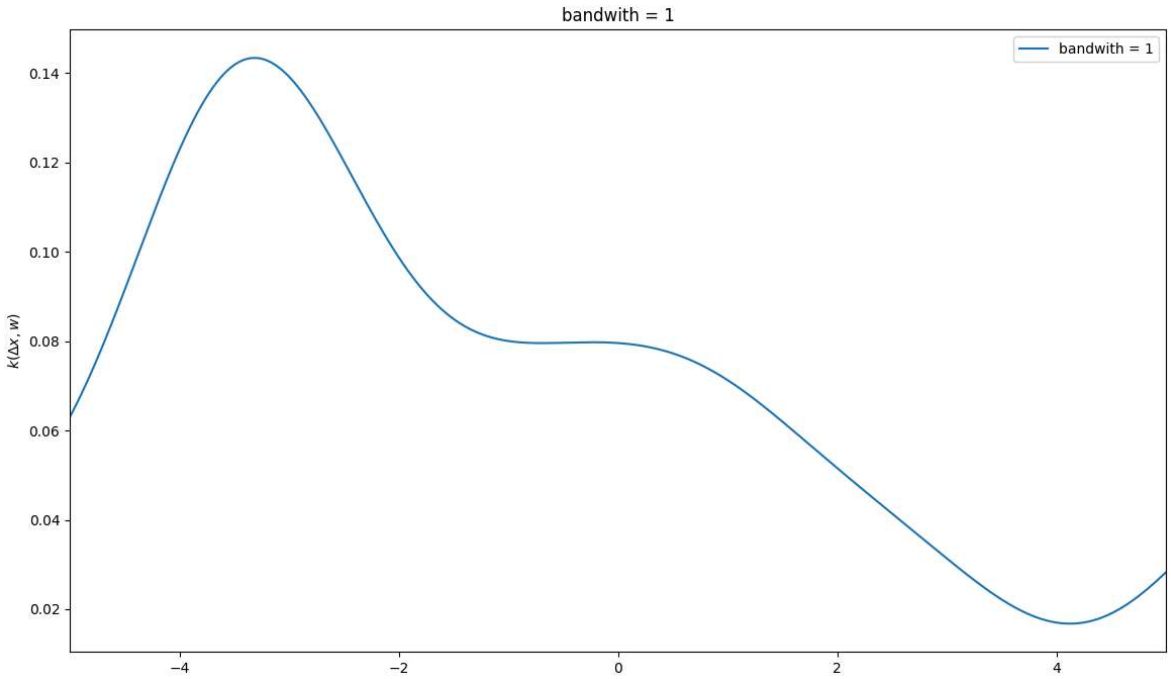
In [4]:
```
import matplotlib.pyplot as plt
def k(x_line, x_i,w):
    x_diff = x_line[:, None] - x_i[None, :]
    return 1/(w*np.sqrt(2*np.pi))*np.exp(-(x_diff)**2/(2*w**2))
```

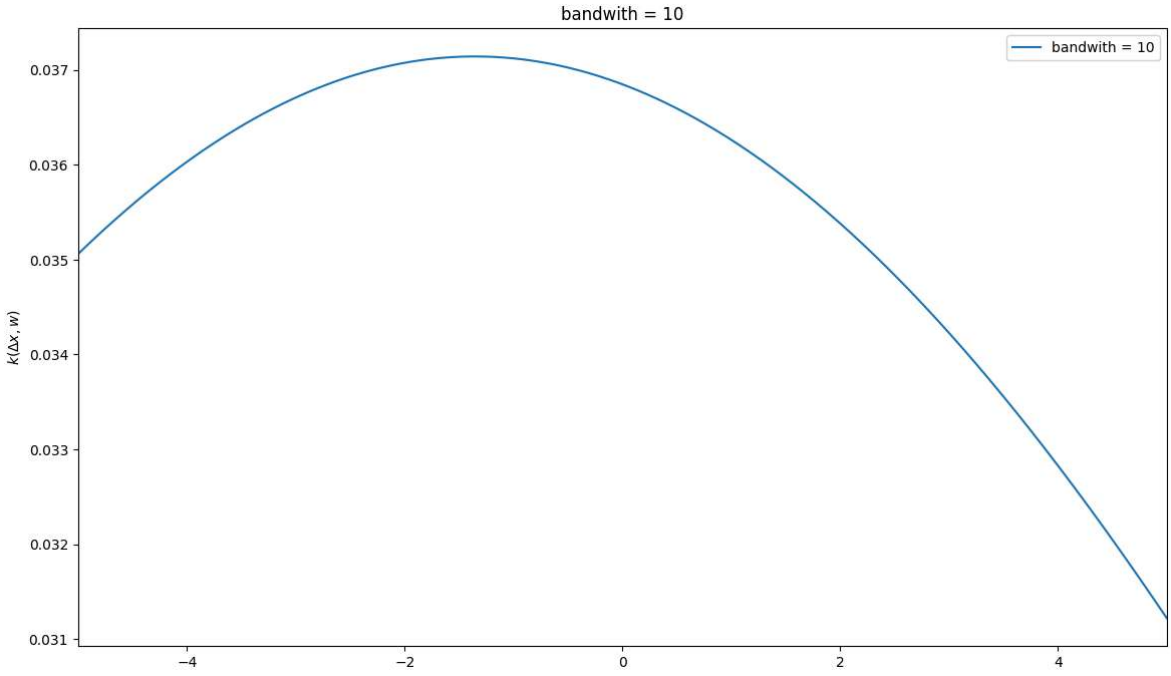
In [7]:
```
## 2)

#generate N=15 measurement points, similar to those in exercise


data = np.load('kde_1d_data.npy')
N=len(data)
#plot it
plt.figure(figsize=(14,.5))
plt.title('measured points')
plt.xlim(-5,5)
plt.hist(x,bins=1000, color='g')

#use kernel density estimation, to estimate probability density,
#with normnal distribution
x_line = np.linspace(-5,5,10000)
k_bw = lambda bw: k(x_line, x, bw)


bandwidth = [0.01,0.4,1,10]
for w in bandwidth:
    plt.figure(figsize=(14,8))
    plt.title(f'bandwith = {w}')
    plt.xlim(-5,5)
    plt.ylabel(r'$k(\Delta x, w)$')
    kvals= k_bw(w)
    density_est = kvals.mean(axis=1)
    plt.plot(x_line, density_est, label=f'bandwith = {w}')
    plt.legend()
```




bandwith = 0.01
bandwith = 0.01
$k(\Delta x, w)$
2.5
2.0
1.5
1.0
0.5
0.0
−4
−2
0
2
4


bandwith = 0.4
bandwith = 0.4
$k(\Delta x, w)$
0.25
0.20
0.15
0.10
0.05
0.00
−4
−2
0
2
4





a)

As we would expect, narrow bandwiths (f.ex. $w = 0.01$) lead to a overfit wherethe set of datapoint are too dominant, while too wide bandwiths tend to generalize too much, such that the KDF dominates.

b)

To locally adjust bandwith, we have to adapt it to the density of the given n data points, in a way, that high densities don't dominate too much. We could use the following scheme:

1. We start with kNN algorithm, to give each datapoint a weighted bandwith w_i^*. Calulate the weight vector b^* by applying kNN on each point and returning the mean distance of its k nearest neighbors. This vector can further be normalized by

dividing it by the mean or median of all sampled distances. WIth a more robust version, taking the median. $b^* = b^*/\text{median}(b^*)$

2.

use adapted bandwiths $w*_i$ such that overfitting will be prevented. Therefore we take $w^* = w\mathbb{1}_n b^*$ where w is a predetermined bandwith, $\mathbb{1}_n$ is a quadratic identity matrix with the rank of our data vector and b^* is the kNN weight.#

3.

Apply it now onto the KDF such that we get the output:

$$k(w^*, x_{\text{data}}, x) = \sum_i \frac{1}{w_i^* \sqrt{2\pi}} \exp\left(\frac{(x-x_i)^2}{2w_i^{*2}}\right)$$

3.

a) $$\hat{\beta} = \underset{\beta}{\operatorname{argmax}} \sum_{n=1}^{N} \log \mathcal{N}(y_n \mid \beta^T x, \sigma^2)$$

$$= \underset{\beta}{\operatorname{argmax}} \sum_{n=1}^{N} \log\left(\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(y_n - \beta^T x)^2}{2\sigma^2} \right) \right)$$

$$= \underset{\beta}{\operatorname{argmax}} \sum_{n=1}^{N} -\log(\sqrt{2\pi}\sigma) - \frac{(y_n - \beta^T x)^2}{2\sigma^2}$$

$$= \underset{\beta}{\operatorname{argmax}} \; -\log(\sqrt{2\pi}\sigma)\cdot N - \underbrace{\sum_{n=1}^{N} \frac{(y_n - \beta^T x_n)^2}{2\sigma^2}}_{=: B} \qquad \underbrace{\frac{1}{2\sigma^2} \cdot \sum (y_n - \beta^T x)^2}$$

ignore non β-dependent part

$$\Rightarrow = \underset{\beta}{\operatorname{argmin}} \underbrace{\sum_{n=0}^{N} (y_n - \beta^T x)^2}_{= SSQ}$$

To find $\hat{b}$ we find the minimum of $SSQ: \|\vec{y} - \beta^T X\|_2^2$

$$X = \begin{bmatrix} x_1^T \\ \vdots \\ x_N^T \end{bmatrix} \in \mathbb{R}^{N \times N} \hookrightarrow X\beta = \begin{bmatrix} x_0^T \\ \vdots \\ x_N^T \end{bmatrix} \beta = \begin{bmatrix} x_0^T \beta \\ \vdots \\ x_N^T \beta \end{bmatrix}$$

as $x_i^T \beta$ is a scalar we the can write it as $X\beta = \begin{bmatrix} \beta^T x_0 \\ \vdots \\ \beta^T x_N \end{bmatrix}$

$$\Rightarrow \sum_{n=0}^{N} (y_n - \beta^T x_n)^2 = \|\vec{y} - X\beta\|_2 = (\bar{y} - X\beta)^T (\bar{y} - X\beta) \equiv Q$$

$$Q = [y^T - (X\beta)^T](\bar{y} - X\beta) = y^T y - \underbrace{\beta^T X^T y}_{\text{is scalar s.t. } \beta^T X^T y = (\beta^T X^T y)^T = y^T X\beta} - y^T X\beta + \beta^T X^T X\beta$$

$$= y^T y - 2\beta^T X^T y + \beta^T X^T X\beta$$

$\frac{\partial}{\partial\beta} \beta^T a = a \equiv X^T y$

$\partial_\beta \beta^T \underbrace{X^T X}_{\equiv S \in \mathbb{R}^{n \times n}} \beta = (S + S^T)\beta$

$$\Rightarrow \frac{\partial Q}{\partial \beta} = -2X^T y + 2X^T X\beta \stackrel{!}{=} 0$$

$$\Rightarrow X^T X\beta = 2X\bar{y} \quad \text{as } X \text{ should be positive definit}$$

$$\hat{\beta} = (X^T X)^{-1} X\bar{y} \quad \text{this equals the least-squares estimator}$$

Having this we now can calculate the σ-estimator $\hat{\sigma}$:

$$\hat{\sigma}^2 = \underset{\sigma^2}{\operatorname{argmax}} \sum_{n=1}^{N} \log \mathcal{N}(y_n \mid X^T x, \sigma^2) =$$

$$= \underset{\sigma^2}{\operatorname{argmax}} \; -\log(\sqrt{2\pi}\sigma)N - \frac{\|\bar{y} - X\hat{\beta}\|}{2\sigma^2}$$

$$= \underset{\sigma^2}{\operatorname{argmin}} \; \log(\sqrt{2\pi}\sigma)N - \frac{1}{2\sigma^2}\|\bar{y} - X\hat{\beta}\|$$

$$\Rightarrow 0 \stackrel{!}{=} N\frac{\partial}{\partial\sigma}\log\sigma + \frac{\partial}{\partial\sigma}\frac{1}{\sigma^2} \cdot \frac{\|\bar{y} - X\hat{\beta}\|}{2}$$

$$0 = \frac{N}{\sigma} - \frac{2}{\sigma^3}\frac{\|\bar{y} - X\hat{\beta}\|}{2}$$

$$N\sigma^2 = \|y - X\hat{\beta}\| \Rightarrow \hat{\sigma}^2 = \frac{SSQ}{N} =$$

b) apply SSQ problem on $y_n = \beta^T x_n + \varepsilon_n$

$\rightsquigarrow$ $\|y_n - (X\beta + \varepsilon)\|$ $\Rightarrow$ we get an additional term

$= (\bar{y} - (X\beta + \bar{\varepsilon}))^T (\bar{y} - (X\beta + \bar{\varepsilon}))$ $= y^T y - (X\beta + \varepsilon)^T y - y^T (X\beta + \varepsilon) + \overbrace{(X\beta + \varepsilon)^T (X\beta + \varepsilon)}$ $= \beta^T X^T X \beta + \beta^T X^T \varepsilon + \varepsilon^T X \beta + \varepsilon^T \varepsilon$

ignore terms w.o. β

$= -\beta^T X^T y - y^T X \beta + \beta^T X^T X \beta + \beta^T X \varepsilon + \varepsilon^T X \beta \equiv Q$

apply logic that $\beta^T X \varepsilon$ is a scalar we say: $\|\beta^T X \varepsilon\| = \|(\beta^T X \varepsilon)^T\| = \|\varepsilon^T X^T \beta\|$

$\partial_\beta Q = -2\beta^T X^T y + 2\beta^T X \varepsilon + \beta^T X^T X \beta$

$0 \stackrel{!}{=} -2(X^T y - X\varepsilon) + 2X^T X \beta$

$\hat{\beta}'(\varepsilon) = (X^T X)^{-1}(X^T y - X\varepsilon) = \hat{\beta} - (X^T X)^{-1} X \varepsilon$