Assignment 1 (part I): Line Fitting and other "stuff"

In [ ]:

Problem 1

Prove that any roto-reflective transformation $R^n \rightarrow R^n$ defined by $n \times n$ orthogonal matrix R (s.t. $R^T R = I$) preserves (a) parallel lines and (d) distances between points. Your proof should use only linear algebraic equations. HINTS: you can use line equations of the following format: $p_t = p_o + t\,u$ where p are n-vectors representing points on a line, u is a vector defining the line's direction, and t is a scalar parameter. For distances $d(a,b)$ between points $a, b \in R^n$ you can use $d^2(a,b) = (a-b)^T(a-b)$. NOTE that all your linear algebraic equations in the proof should work for arbitrary $n \times n$ roto-reflective transformations/matrices.

Solution:

To solve this problem, we need to show that a roto-relfective transformation in $\mathbb{R}^n$, defined by an orthogonal matrix R where $R^T R = I$, preserved parallel line and distances between points.

- Orthogonal Matrix: A matrix R is orthogonal if $R^T R = I$ where R^T is the transpose of R and I is the identity matrix. This implies that $R^{-1} = R^T$, meaning the inverse of an orthogonal matrix is its transpose.
- Roto-Reflective Transformation: This is a transformation that can be represented as $R \cdot p$, where p is a vector in $\mathbb{R}^n$ and R is an orthogonal matrix.
- Parallel Lines: Two lines are parallel if they are always the same distance apart and will never meet.
- Distance preservation: A transformation preserves distance if the distance betwene any two points before the transformation is the same as the distance between the transformed points.

(a) Proof for Parallel Lines \ Let's consider two lines L_1 and L_2, where the lines are represented as:

- L_1: $p_t = p_0 + tu$
- L_2: $q_s = q_0 + sv$ \ where t and s are scalar parameters, u and v are direction vectors, and p_0 and q_0 are points on the line. Then, lines L_1 and L_2 are parallel if their direction vectors are scalar multiples of each other, i.e., $u = k \cdot v$, where k is a non-zero scalar.

1. When we apply the transformation R to these lines, we get the transformed lines:

- L_1' represented as $R \cdot p_t$
- L_2' represented as $R \cdot q_s$ \ So,

$$L_1' = R \cdot p_t = R \cdot (p_0 + tu) = R \cdot p_0 + R(tu) \tag{1}$$
$$L_2' = R \cdot q_t = R \cdot (q_0 + sv) = R \cdot q_0 + R(sv) \tag{2}$$

1. Since R is linear and orthogonal, it preserved scalar multiplication, so

$$R(tu) = t(R \cdot u) \tag{3}$$
$$R(sv) = s(R \cdot v) \tag{4}$$

Then the equation for the transformed lines, become:

$$L_1' = R \cdot p_0 + t(R \cdot u) \tag{5}$$
$$L_2' = R \cdot q_0 + s(R \cdot v) \tag{6}$$

Since $u = kv$, applying R gives $R \cdot u = R \cdot (kv)$. Because R preseves scalar multiplication, $R \cdot (kv) = k(R \cdot v)$, which means directional vectors of the transformed lines $R \cdot u$ and $R \cdot v$ are still propostional by the same scalar.

$\therefore$ The transformed lines L_1' and L_2' have direction vectors that are proportional to each other: $R \cdot v$ is to $R \cdot u$ as u is to v, which means L_1' and L_2' are parallel. This completes the proof that orthogonal transformation R preserved the parallelism of lines.

(b) Proof for Distance between points \ Consider two points $a, b \in \mathbb{R}^n$. \ The distance between a and b is given by
$d^2(a,b) = (a-b)^T(a-b) \implies d(a,b) = \sqrt{(a-b)^T(a-b)}$

We apply the roto-reflective transformation R to get the transformed points a' and b', which can be writted as:

$$a' = R \cdot a$$
$$b' = R \cdot b$$

Then the distance between the transformed points can be given as:

$$
\begin{aligned}
d(a', b') &= d(R \cdot a, R \cdot b) && (7)\\
&= \sqrt{((R \cdot a) - (R \cdot b))^T((R \cdot a) - (R \cdot b))} && (8)\\
&= \sqrt{(a^T R^T - b^T R^T)((R \cdot a) - (R \cdot b))} && (9)\\
&= \sqrt{(a^T - b^T)R^T(R \cdot a - R \cdot b)} && (10)\\
&= \sqrt{(a^T - b^T)R^T R(a - b)} && (11)\\
&= \sqrt{(a^T - b^T)I(a - b)} && (12)\\
&= \sqrt{(a^T - b^T)(a - b)} && (13)\\
&= d(a, b) && (14)
\end{aligned}
$$

$\therefore d(a', b') = d(R \cdot a, R \cdot b) = d(a, b)$ as required.

This shows that the distance between points a and b is presered under the transformation R.

Hence, the roto-relective transformation R preserves both parallel lines and distances beteeen points in $\mathbb{R}^n$

Problem 2

Prove that affine transformations map lines onto lines. For this, take an arbitrary line in R^2 and show that an arbitrary affine transform maps it onto a set of points that also satisfies a line equation. You should use homogeneous representation of lines, i.e. equations $l^\top x = 0$ where l is a 3-vector of line parameters (so called, homogeneous line representation) and x is a 3-vector (homogeneously) representing a point on the line.

HINT: Use 3x3 matrices A to represent affine transforms, as in Topic 4. Find simple linear-algebraic equation for the transformed line parameters l' given A and the original line l.

Solution: \ An affine transformation in $\mathbb{R}^2$ can be represented as a 3×3 matrix that operates on a point in homogeneous coordinates. The matrix A and the point x in homogeneous coordinates are given by:

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$$

A line in $\mathbb{R}^2$ can be represented in homogeneous coordinates by a vector $l = (l_1, l_2, l_3)^\top$, where the line equation is given by $l^\top x = 0$. Here, x is a point on the in homogenous coordinates $x = (x_1, x_2, 1)^\top$

Proof:

1. Take a point x on the original line, which satisfies the equation $l^\top x = 0$
2. Apply the affine transformation to the point x, giving the new point $x' = Ax$
3. We want the transformed points to satisfy a line eqaution $l^\top x' = 0$.
4. Then by substitution, we get $l^\top (Ax) = 0$
5. For this to hold for all points x on the original line, it must be the case that $l^\top A = l^\top$ which implies that $l = A^{-T} l$ where A^{-T} is the inverse of the transpose of A. The existence of A^{-T} is guaranteed because the affine transformations are invertible.
6. Since, $l^\top (Ax) = (A^{-T} l)^T (Ax) = l^\top (A^{-1} A) x = l^\top x = 0$, we shown that the transformed line is represented by the equation $l^\top x = 0$

∴ The image of the line under the affine transformation is also a line, because the set of points x satisfies the line equatio in homogeneous coordinates.

NOTE: Problems 3-7 below are mostly coding excersices where you should implement and/or test different standard methods for model fitting on examples with synthetic and real data. Part II of this Assignemnt requires homography estimation in a real application (panorama mosaicing). The problems below were primarily designed to help the students learn the basics of model parameter estimation in a much more basic context - simple line models and 2D data points (synthetic or real).

While the provided initial notebook shows synthetic and real examples of 2D data points for line fitting, you might need to restart the nootebook (Kernel->Restart and Clear Output).

Problem 3: least-squares and line fitting in 2D (synthetic data without outliers)

Complete implementation of function $estimate$ of class $LeastSquareLine$ in the second cell below. It should update line parameters a and b correpsonding to line model $y = ax + b$. You can use either SVD of matrix A or inverse of matrix $A^T A$, as mentioned in class. NOTE: several cells below test your code.

In [1]:
```
%matplotlib notebook

import numpy as np
import numpy.linalg as la
import matplotlib
import matplotlib.pyplot as plt
from skimage.measure import ransac
import math
```

TO IMPLEMENT: complete (fix) the code in the following cell. Note that solution has 2-3 lines. You can use svd function in la and/or standard matrix operations from np.

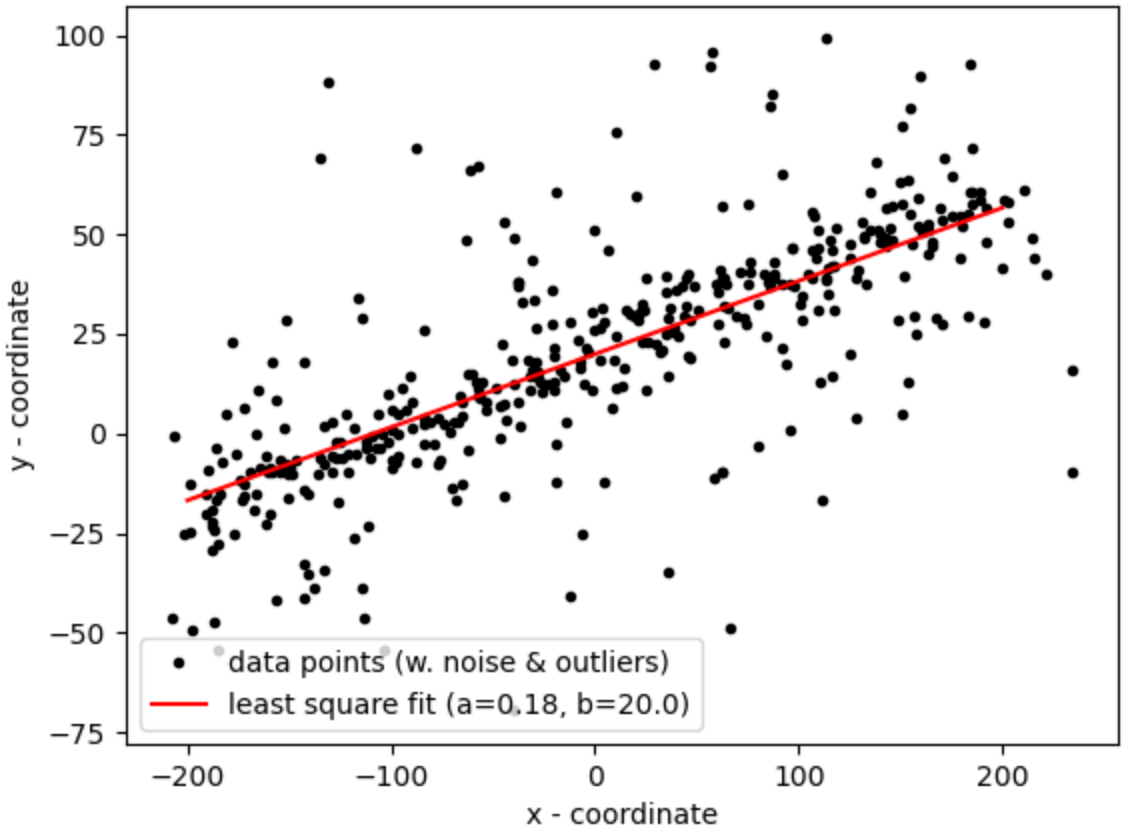
In [2]:
```
class LeastSquareLine:

    def __init__(self):
        self.a = 0.0
        self.b = 0.0

    def estimate(self, points2D):
        B = points2D[:,1]
        A = np.copy(points2D)
        A[:,1] = 1.0

        # Vector B and matrix A are already defined. Change code below
        self.a = 0.0
        self.b = 0.0
        # changes start here
        A_transpose = np.transpose(A)
        A_inverse = la.inv(np.matmul(A_transpose, A))
        B = np.matmul(A_transpose, B)
        X = np.matmul(A_inverse, B)
        self.a = X[0]
```

```
        self.b = X[1]
        # changes end here
        return True

    def predict(self, x): return (self.a * x) + self.b

    def predict_y(self, x): return (self.a * x) + self.b

    def residuals(self, points2D):
        return points2D[:,1] - self.predict(points2D[:,0])

    def line_par(self):
        return self.a, self.b
```

Working code below generates (simulates) data points in $\mathcal{R}^2$ corresponding to noisy observations of a line.

In [3]:
```
np.random.seed(seed=1)

# parameters for "true" line y = a*x + b
a, b = 0.2, 20.0

# x-range of points [x1,x2]
x_start, x_end = -200.0, 200.0

# generate "idealized" line points
x = np.arange(x_start,x_end)
y = a * x + b
data = np.column_stack([x, y])    # staking data points into (Nx2) array

# add gaussian pertubations to generate "realistic" data points (noisy line observations)
noise = np.random.normal(size=data.shape) # generating Gaussian noise (variance 1) for each data point (rows
data += 5 * noise
data[::2] += 10 * noise[::2]  # every second point adds noise with variance 5
data[::4] += 20 * noise[::4] # every fourth point adds noise with variance 20

# IMPORTANT COMMENT!!!!!!!:
# the data now combines line points with different levels of additive Gaussie noise.
# Consider all these points as noisy inliers.
# Do not consider any of these as outliers. The outliers are added in Problem 4.


fig, ax = plt.subplots()
ax.plot(data[:,0], data[:,1], '.k', label='data points (w. noise & outliers)')
ax.set_xlabel('x - coordinate')
```

```
ax.set_ylabel('y - coordinate')
ax.legend(loc='lower left')
plt.show()
```



Use the following code-cell to test your implementation of class $LeastSquareLine$ in Problem 3 for line fitting when observed data is noisy. The estimated line is displayed in the cell above. Note that the initial result shown in Fig.1 of the provided notebook corresponds to the line $a = 0, b = 0$ returned by initial code for $LeastSquareLine$. Your correct solution for $LeastSquareLine$ should return a line very close to the known (ground-truth) line model.

In [4]:
```
LSline = LeastSquareLine() # uses class implemented in Problem 2
print (LSline.estimate(data))
```

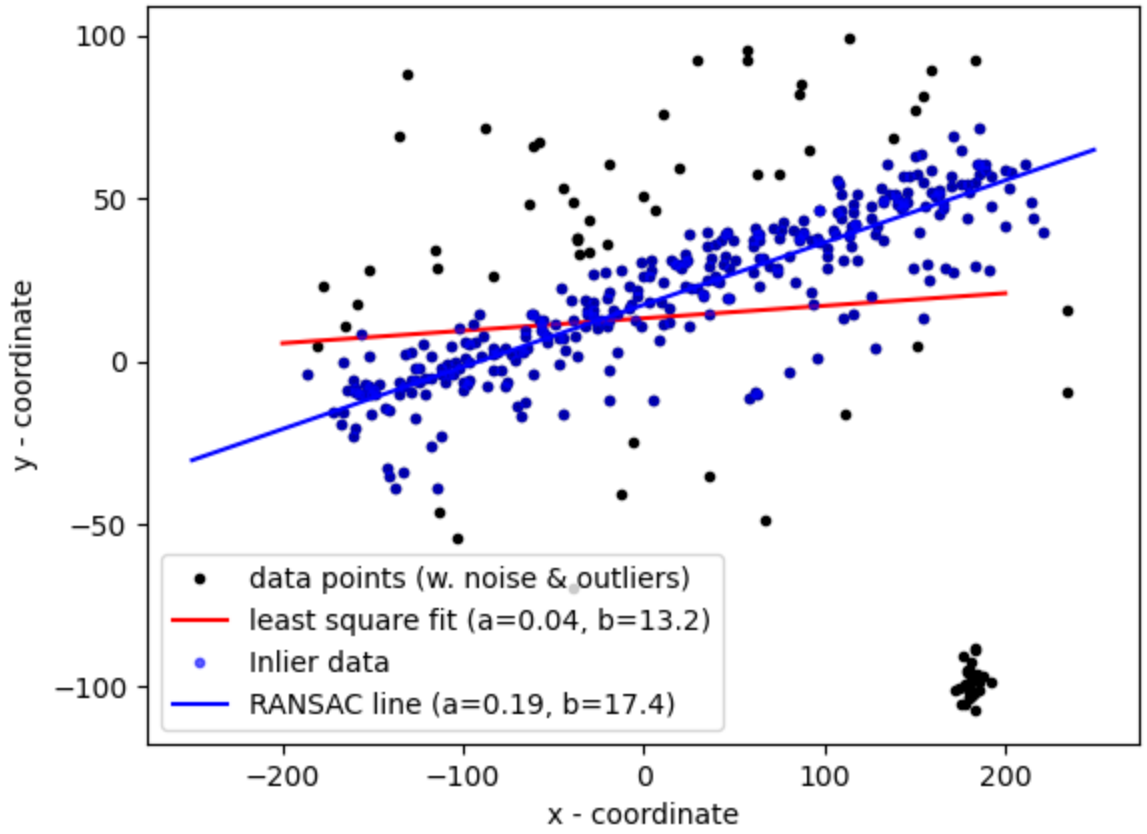
```
a_ls, b_ls = LSline.line_par()
print(f'a: {a_ls}. \nAs you can see the value for a_ls is very close to the true value given as a=0.2') # pri
print(f'b: {b_ls}. \nAs you can see the value for b_ls is very close to the true value given as b=20.0') # pr

# visualizing estimated line
ends = np.array([x_start,x_end])
ax.plot(ends, LSline.predict(ends), '-r', label='least square fit (a={:4.2f}, b={:4.1f})'.format(a_ls,b_ls))
ax.legend(loc='lower left')
plt.show()
```

```
True
a: 0.18353493312620206.
As you can see the value for a_ls is very close to the true value given as a=0.2
b: 19.956828768346003.
As you can see the value for b_ls is very close to the true value given as b=20.0
```

Problem 4: RANSAC for robust line fitting in 2D (synthetic data with outliers)

Working code in the cell below corrupts data with outliers.

In [5]:
```
# add outliers
faulty = np.array(30 * [(180., -100)])  # (30x2) array containing 30 rows [180,-100]  (points)
faulty += 5 * np.random.normal(size=faulty.shape)  # adding Gaussian noise to these points
data[:faulty.shape[0]] = faulty   # replacing the first 30 points in data with faulty (outliers)


fig, ax = plt.subplots()
ax.plot(data[:,0], data[:,1], '.k', label='data points (w. noise & outliers)')
ax.set_xlabel('x - coordinate')
ax.set_ylabel('y - coordinate')
ax.legend(loc='lower left')
plt.show()
```


100
50
0
−50
−100
y - coordinate
−200
−100
0
100
200
x - coordinate
data points (w. noise & outliers)
least square fit (a=0.04, b=13.2)
Inlier data
RANSAC line (a=0.19, b=17.4)

NOTE: As clear from the code for data simulation, (creation of) outliers has nothing to do with the line. In contrast, all other points originate from some points on the true (perfect) line, see Probelm 3. The added Gaussian errors (large or small) simulate complex noise that commonly happens between true (but unknown) model and its real observations (data). The whole point of "model fitting" is to estimate (or restore) a "model" (e.g. line parameters) from its noisy observations (data). If data has no noise, it is trivial, e.g. only two line points would be enough to compute the line parameters exactly. In the presence of noise, one can use methods like least-squres to approximately estimate a line (parameters minimizing the loss functing summing squared L_2 errors), see Problem 3. However, if data is corrupted by outliers that have nothing to do with the model, as in Problem 4, more robust model fitting methods are needed, e.g. RANSAC.

The code below uses your implementation of class $LeastSquareLine$ from Probelm 3 for least-square line fitting when the data is corrupted with outliers. The estimated line is displayed in the cell above. Observe the differences with the result in Problem 3.

In [6]:
```
LSline = LeastSquareLine() # uses class implemented in Problem 2
print (LSline.estimate(data))
a_ls, b_ls = LSline.line_par()
print(f'a:{a_ls}')
print(f'b:{b_ls}')

# visualizing estimated line
ends = np.array([x_start,x_end])
ax.plot(ends, LSline.predict(ends), '-r', label='least square fit (a={:4.2f}, b={:4.1f})'.format(a_ls,b_ls))
ax.legend(loc='lower left')
plt.show()
```

```
True
a:0.038386619910090944
b:13.241985265558936
```

(part a) Assume that a set of $N = 100$ points in $2D$ includes $N_i = 20$ inliers for one line and $N_o = 80$ outliers. What is the least number of times one should sample a random pair of points from the set to get probability $p \geq 0.95$ that in at least one of the sampled pairs both points are inliers? Derive a general formula and compute a numerical answer for the specified numbers.

Solution:

Given:

- N: total number of points, where $N = 100$
- N_i: number of inliner points, where $N_i = 20$
- N_o: number of outlier points, where $N_o = 80$
- s: number of points chosen at each iteration, where $s = 2$ because we are trying to fit a line
- p desired probability we want, where $p \geq 0.95$
- k: The least number of samples or iterations required to achieve probability $p \geq 0.95$

Then,

- p_i: probability of choosing one inliner point is given as $p_i = \frac{N_i}{N}$
- p_o: probability of choosing one outlier point is given as $p_o = \frac{N_o}{N}$

Thus, the probability of choosing both points in a pair being inliners is $p_i^s = p_i^2$, as $s = 2$.

Deriving the formula:

- We want to find at least one set of s inliners in k random samples
- The probability of selecting an inliner on a single draw in p_i
- The probability of selecting a set of s inliners in one sample is p_i^s, assuming that events are independent.
- The probability of not selecting a set of s inliners in one sample is $p_o = 1 - p_i^s$
- The probability of not selecting a set of s inliners in k independed samples is $(1 - p_i^s)^k$

$\therefore$ The probability p of selecting at least one set of s inliners in k samples is the complement of the above given as:

$$p = 1 - (1 - p_i^s)^k$$

Formula:

$$p = 1 - (1 - p_i^s)^k$$

Solving for k:

$$p = 1 - (1 - p_i^s)^k \tag{15}$$
$$(1 - p_i^s)^k = 1 - p \tag{16}$$
$$\log((1 - p_i^s)^k) = \log(1 - p) \tag{17}$$
$$k \log(1 - p_i^s) = \log(1 - p) \tag{18}$$
$$\therefore \boxed{k = \frac{\log(1 - p)}{\log(1 - p_i^s)}} \tag{19}$$

Computing a numerical answer for the given:

- $N = 100$
- $N_i = 20$
- $N_o = 80$
- $p = 0.95$
- $s = 2$

1. Calculate p_i

$$p_i = \frac{N_i}{N} = \frac{20}{100} = 0.20 \tag{20}$$
$$\therefore p_i = 0.20 \tag{21}$$

2. Calculate k

$$k = \frac{\log(1 - p)}{\log(1 - p_i^s)} = \frac{\log(1 - 0.95)}{\log(1 - (0.20)^2)} = \frac{-1.301029996}{-0.01772876696} = 73.38525 \approx 74 \tag{22}$$
$$\therefore \boxed{k = 74} \tag{23}$$

Thus, we need at least $k = 74$ iterations to have $p \geq 0.95$

(part b) Using the knowledge of the number of inliers/outliers in the example at the beginning of Problem 4, estimate the minimum number of sampled pairs needed to get RANSAC to "succeed" (to get at least one pair of inliers) with $p \geq 0.95$. Use your formula in part (a). Show your numbers in the cell below. Then, use your estimate as a value of parameter max_trials inside function $ransac$ in the code cell below and test it. You should

also change $residual\_threshold$ according to the noise level for inliers in the example. NOTE: the result is displayed in the same figure at the beginning of Problem 4.

Your estimates:

- N = length of the data from Problem 3 & 4
- N_o = number of outliers added from few cells above at the beginning of Problem 4
- N_i = $N - N_o$
- s = 2 as we are sampling points for a line
- p = 0.95, which is the desired probability we want

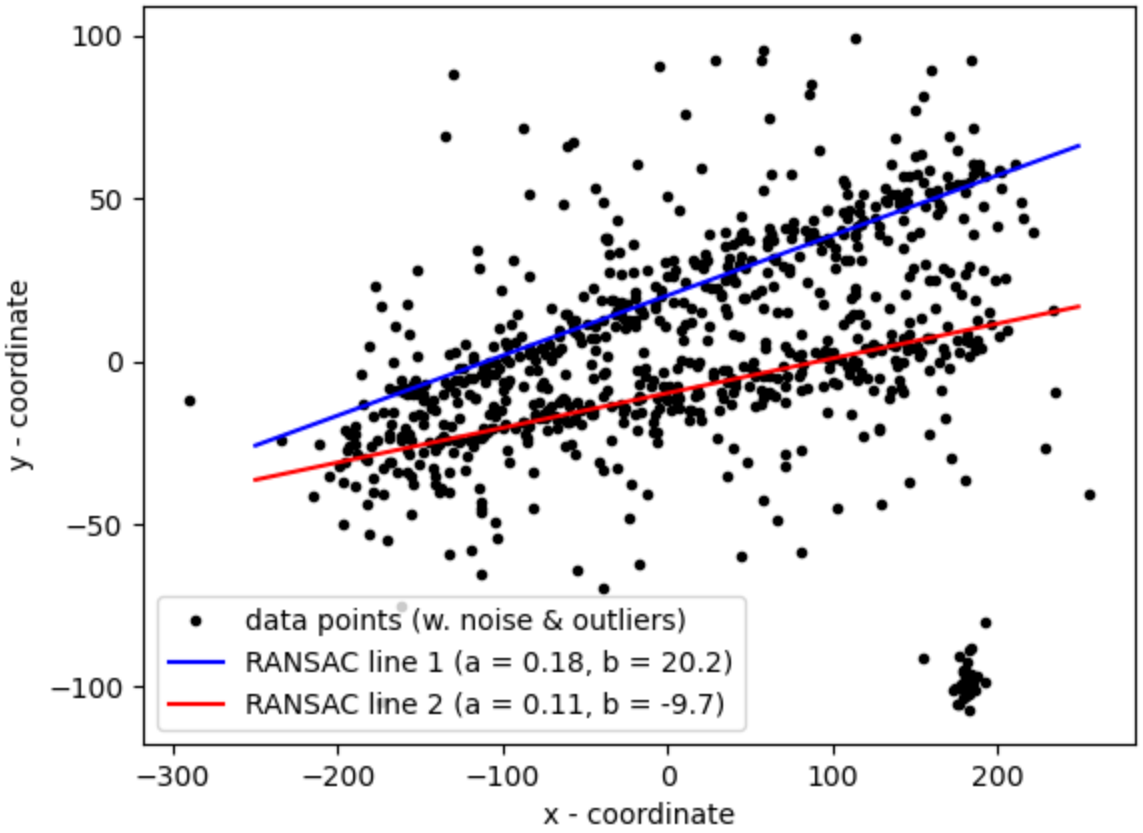
In [7]:
```
# calculates the number of iterations required to achieve the desired probability
def num_iterations(p, s, Ni, N):
    pi = Ni / N
    k = (math.log(1 - p))/(math.log(1 - (pi ** s)))
    return math.ceil(k)
# Given estimates
N = len(data)
No = 30
Ni = N - No
s = 2
p = 0.95

# calculating the number of estimates required
k = num_iterations(p, s, Ni, N)
print(f'Number of trials -> k:{k}')

# robustly fit line using RANSAC algorithm
model_robust, inliers = ransac(data, LeastSquareLine, min_samples=2, residual_threshold=No, max_trials=k)
a_rs, b_rs = model_robust.line_par()
print(f'a: {a_rs}, b:{b_rs}')

# generate coordinates of estimated models
line_x = np.arange(-250, 250)
line_y_robust = model_robust.predict_y(line_x)

#fig, ax = plt.subplots()
ax.plot(data[inliers, 0], data[inliers, 1], '.b', alpha=0.6, label='Inlier data')
ax.plot(line_x, line_y_robust, '-b', label='RANSAC line (a={:4.2f}, b={:4.1f})'.format(a_rs,b_rs))
ax.legend(loc='lower left')
plt.show()
```

```
Number of trials -> k:2
a: 0.19106974092738577, b:17.39318415799239
```

Problem 5: sequential RANSAC for robust multi-line fitting (synthetic data)

Adding data points supporting one more line

In [8]:
```
# parameters for "true" lines y = a*x + b
a2, b2 = 0.1, -10.0

# generate "idealized" line points
y2 = a2 * x + b2
data2 = np.column_stack([x, y2])    # staking data points into (Nx2) array

# add gaussian pertubations to generate "realistic" line data
noise = np.random.normal(size=data.shape) # generating Gaussian noise (variance 1) for each data point (rows
data2+= 5 * noise
data2[::2] += 10 * noise[::2]  # every second point adds noise with variance 5
data2[::4] += 20 * noise[::4] # every fourth point adds noise with variance 20

data = np.concatenate((data,data2)) # combining with previous data

fig, ax = plt.subplots()
ax.plot(data[:,0], data[:,1], '.k', label='data points (w. noise & outliers)')
ax.set_xlabel('x - coordinate')
ax.set_ylabel('y - coordinate')
ax.legend(loc='lower left')
plt.show()
```



Write code below using sequential RANSAC to detect two lines in the data above. Your lines should be displayed in the figure above (in Problem 5).

In [9]:
```
'''
There are a total of 800 data points in the dataset given.
Assuming that the 30 points added above in Problem 4 are outliers for both data set, so 60 outliers.
So,remaining we have 800 - 60= 740 points.
Assuming out of the remaining 740 points, half of them are outliers for each line so, Ni = (800 - 60)/2
'''
# given estimates and values
N = len(data)
Ni = (N - 60)/2
No = N - Ni
```

```
p = 0.95
s = 2
k =  num_iterations(p, s, Ni, N) # number of trials requried
print(f'N = {N}, Ni = {Ni}, No = {No}, Number of trials: {k}')

# for line 1
model_robust_1, inliners_1 = ransac(data, LeastSquareLine, min_samples = 2, residual_threshold = 3, max_trial
a_rs_1, b_rs_1 = model_robust_1.line_par()
print(f'For line 1: a = {a_rs_1}, b = {b_rs_1}')

line_1_x = np.arange(-250, 250)
line_1_robust_y = model_robust_1.predict_y(line_1_x)


# filter out data points that are likely to be part of the first fitted line
line_data =[] # stores the data points for the second line
for i in range(len(data)):
    y = a_rs_1 * data[i][0] + b_rs_1
    if (abs(y - data[i][1]) >= 20): # selected 20 as an arbitary threshold
        line_data.append(data[i])

line_data = np.reshape(line_data, (len(line_data), 2))

# for line 2
model_robust_2, inliners_2 = ransac(line_data, LeastSquareLine, min_samples = 2, residual_threshold = 3, max_
a_rs_2, b_rs_2 = model_robust_2.line_par()
print(f'For line 1: a = {a_rs_2}, b = {b_rs_2}')

line_2_x = np.arange(-250, 250)
line_2_robust_y = model_robust_2.predict_y(line_2_x)
ax.plot(line_1_x, line_1_robust_y, '-b', label = 'RANSAC line 1 (a = {:4.2f}, b = {:4.1f})'.format(a_rs_1, b_
ax.plot(line_2_x, line_2_robust_y, '-r', label = 'RANSAC line 2 (a = {:4.2f}, b = {:4.1f})'.format(a_rs_2, b_
ax.legend(loc = 'lower left')
plt.show()
```

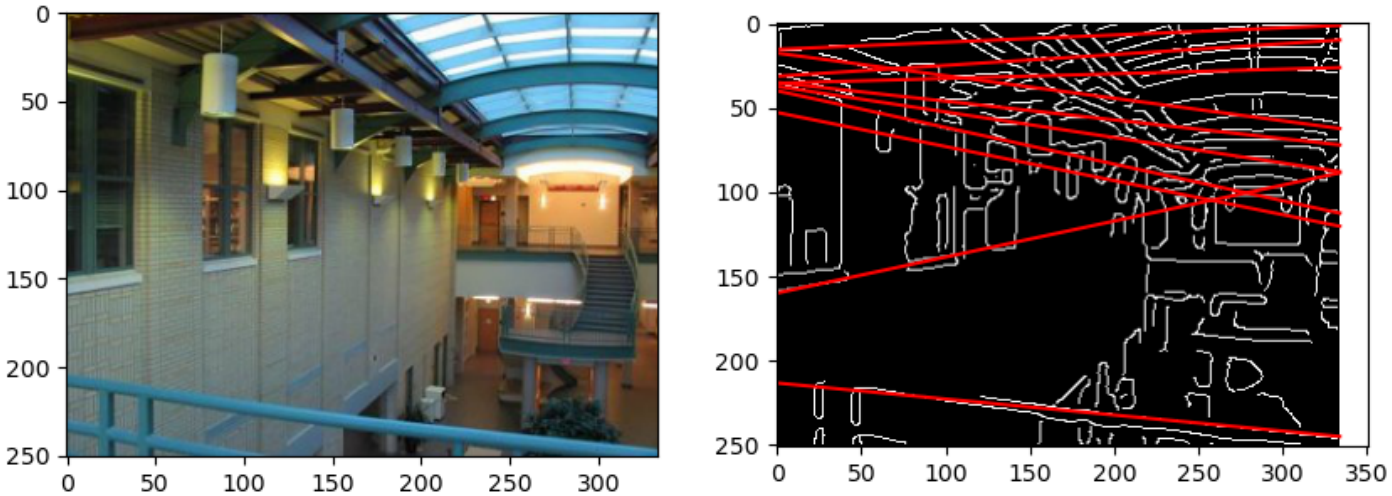
```
N = 800, Ni = 370.0, No = 430.0, Number of trials: 13
For line 1: a = 0.18439347859044966, b = 20.246515509437973
For line 1: a = 0.10658362379124393, b = -9.743834460903415
```

Problem 6: multi-line fitting for real data (Canny edges)

NOTE: while there are real applications that require estimation of lines in images, their detailed description is outside the scope of this assignment. This problem is mostly for fun.

However, it also demostrates a real example of simple 2D feature points (based on contrast edges) that can be used for estimating real geometric models (lines). More advanced real examples of feature points and geometric model estimation can be found in part II of this Assignment.

In [10]:
```
import matplotlib.image as image
from skimage import feature
from skimage.color import rgb2gray

im = image.imread("images/CMU_left.jpg")
imgray = rgb2gray(im)
can = feature.canny(imgray, 2.0)

plt.figure(4,figsize = (10, 4))
plt.subplot(121)
plt.imshow(im)
plt.subplot(122)
plt.imshow(can,cmap="gray")
plt.show()
```



use sequestial-RANSAC to find K lines

In [11]:
```
K = 5
# NOTE 1: write your code using a function that takes K as a parameter.
# NOTE 2: Present visual results for some value of K
# NOTE 3: Your code should visually show detected lines in a figure
#         over the image (either the original one or over the Canny edge mask)
# NOTE 4: You may need to play with parameters of function ransac
#         (e.g. threshold and number of sampled models "max_trials")
#         Also, you can introduce one extra parameter for the minimum number of inliers
#         for accepting ransac-detected lines.

# NOTE: "can" in the cell above is a binary mask with True and False values, e.g.
#print (can[20,30])
#print (can[146,78])

# My code begins here


# Linear model class definition compatible with skimage's ransac()
class LinearModel:
    def estimate(self, data):
        X, y = data[:, 0], data[:, 1]
        X = np.vstack((X, np.ones(len(X)))).T
        self.params_, _, _, _ = la.lstsq(X, y, rcond=None)
        return True

    def residuals(self, data):
        X, y = data[:, 0], data[:, 1]
        X = np.vstack((X, np.ones(len(X)))).T
        residuals = y - X.dot(self.params_)
        return np.abs(residuals)

    def predict_y(self, x):
        a, b = self.params_
        return a * x + b

# Function to perform sequential RANSAC to find K lines
def sequential_ransac(data, k, residual_threshold=4, max_trials=2000):
    lines = []
    data_remaining = data.copy()

    for _ in range(k):
```

```
        model_robust, inliers = ransac(data_remaining, LinearModel, min_samples=2,
                                       residual_threshold=residual_threshold, max_trials=max_trials)
        lines.append(model_robust)
        inliers_mask = np.zeros(len(data_remaining), dtype=bool)
        inliers_mask[inliers] = True
        data_remaining = data_remaining[~inliers_mask]

        if len(data_remaining) < 2:
            break

    return lines

# Read the image and convert to grayscale
im = image.imread("images/CMU_left.jpg")
imgray = rgb2gray(im)

# Apply Canny edge detection
can = feature.canny(imgray, 2.0)

# Extract edge points
y_coords, x_coords = np.where(can)
edge_points = np.column_stack((x_coords, y_coords))

# Set the number of lines you want to find
K = 10

# Apply sequential RANSAC to find K lines
found_lines = sequential_ransac(edge_points, K)

# Plotting the original image and the Canny edges
plt.figure(4, figsize=(10, 4))
plt.subplot(121)
plt.imshow(im)
plt.subplot(122)
plt.imshow(can, cmap="gray")

# Plotting the results with found lines
for line in found_lines:
    x_values = np.linspace(0, can.shape[1], 2)  # Only need two points to define a line
    y_values = line.predict_y(x_values)
    plt.plot(x_values, y_values, '-r')

plt.show() # lines shown in the image above
```

In [ ]: