I basically just implemented what is explained here: https://www.youtube.com/watch?v=REypj2sy_5U&list=PLBv09BD7ez_4e9LtmK626Evn1ion6ynrt (https://www.youtube.com/watch?v=REypj2sy_5U&list=PLBv09BD7ez_4e9LtmK626Evn1ion6ynrt)

Imports and global variables

In [1]:
```python
import numpy as np
from numpy.random import randn
from numpy.random import seed
import math

import matplotlib.pyplot as plt
from pylab import *

from PIL import Image

# Should be an even number
n = 6

seed(1)
max_iters = 3
```

Table of contents

Step 0: Task for assignment 2

Implement the expectation-maximization method for the generative model of the mixture of two Gaussian's in one dimension, as developed in the lecture. Demonstrate that your code works for some simple surrogate dataset of your choice and design. Plot the final pdf shapes and show convergence of parameters by monitoring the values at each step.

Step 1: Surrogate data generation

In [2]:

```
mu_1 = 5
mu_a_true = mu_1
mu_2 = -6
mu_b_true = mu_2
sigma_1 = 1
sigma_a_true = sigma_1
sigma_2 = 2
sigma_b_true = sigma_2

x_1 = np.random.normal(mu_1, sigma_1, int(n/2))
x_2 = np.random.normal(mu_2, sigma_2, int(n/2))

list_x = []
list_x.extend(x_1)
list_x.extend(x_2)
```

Step 2: Expectation-maximization method for the generative model of the mixture of two Gaussians in one dimension

In [3]:

```
pil_im = display(Image.open('P(x_i|b).png'))
def probability_x_i_given_b(sigma_square, x, mu):

    result = (
        1 / math.sqrt(2 * math.pi * sigma_square)
    ) * math.exp(
        -1*(
            (x-mu)**2 / 2*sigma_square
        )
    )

    return result
```

$$P(x_i \mid b) = \frac{1}{\sqrt{2\pi\sigma_b^2}} \exp\left(-\frac{(x_i - \mu_b)^2}{2\sigma_b^2} \right)$$

In [4]:

```
pil_im = display(Image.open('mu_a.png'))
pil_im = display(Image.open('mu_b.png'))
def mu_a_or_b(list_a, list_x):
    numerator = 0
    denominator = 0
    for index, element in enumerate(list_a):
        numerator = numerator + (element * list_x[index])
        denominator = denominator + element
    mu = numerator / denominator

    return mu
```

$$\mu_a = \frac{a_1x_1 + a_2x_2 + \ldots + a_nx_n}{a_1 + a_2 + \ldots + a_n}$$

$$\mu_b = \frac{b_1x_1 + b_2x_2 + \ldots + b_nx_n}{b_1 + b_2 + \ldots + b_n}$$

In [5]:

```
pil_im = display(Image.open('P(a|x_i).png'))
pil_im = display(Image.open('P(b|x_i).png'))
def probability_a_or_b_given_x_i(sigma_square_a, sigma_square_b, x, mu_a, mu_
    p_x_i_giv_b = probability_x_i_given_b(sigma_square=sigma_square_b, x=x,
    p_x_i_giv_a = probability_x_i_given_b(sigma_square=sigma_square_a, x=x,

    result = (p_x_i_giv_b * probability_b) / ((p_x_i_giv_b * probability_b)

    return result
```

$$a_i = P(a \mid x_i) = 1 - b_i$$

$$b_i = P(b \mid x_i) = \frac{P(x_i \mid b)P(b)}{P(x_i \mid b)P(b) + P(x_i \mid a)P(a)}$$

In [6]:

```python
pil_im = display(Image.open('sigma_a^2.png'))
pil_im = display(Image.open('sigma_b^2.png'))
def estimated_sigma_square(list_a, list_x, mu):
    numerator = 0
    denominator = 0
    for index, element_a in enumerate(list_a):
        numerator = numerator + element_a * ((list_x[index] - mu)**2)
        denominator = denominator + element_a

    estimated_sigma_square = numerator / denominator

    return estimated_sigma_square
```

$$\sigma_a^2 = \frac{a_1(x_1-\mu_1)^2+\ldots+a_n(x_n-\mu_n)^2}{a_1+a_2+\ldots+a_n}$$

$$\sigma_b^2 = \frac{b_1(x_1-\mu_1)^2+\ldots+b_n(x_n-\mu_n)^2}{b_1+b_2+\ldots+b_n}$$

Step 3: Plot pdf shapes and show convergence of parmeters by monitoring the values at each step

In [7]:

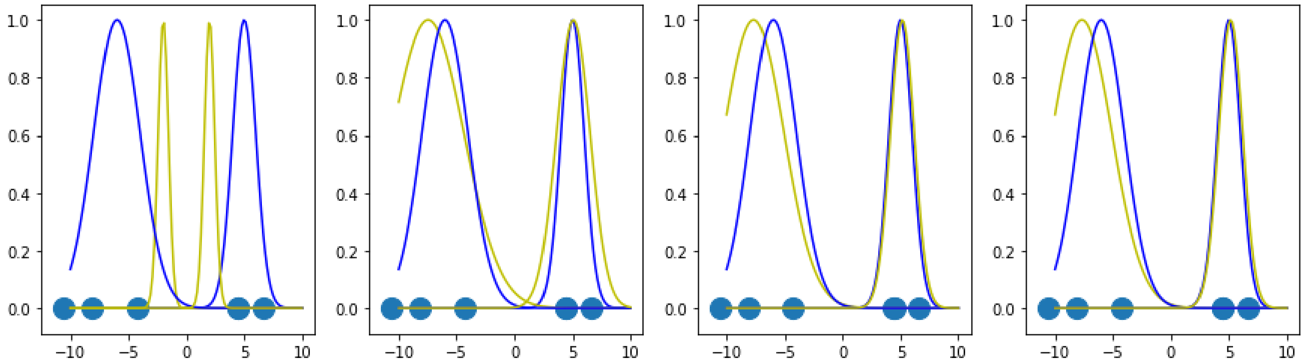
```python
def gaussian(x, mu, sig):
    return np.exp(-np.power(x - mu, 2.) / (2 * np.power(sig, 2.)))

def plot_true_and_estimated_distributions(list_of_tuples_with_parameters):
    x_values = np.linspace(-10, 10, 120)
    for mu, sig in list_of_tuples_with_parameters:
        plt.plot(x_values, gaussian(x_values, mu[0], sig[0]), c='b')
        plt.plot(x_values, gaussian(x_values, mu[1], sig[1]), c='y')

    list_y = []
    for x_i in range(0, len(list_x)):
        list_y.append(0)

    plt.scatter(list_x, list_y, s=200)
```

In [8]:

```python
def expectation_maximization(list_x):
    # Initializations
    sigma_square_a = 0.2
    sigma_square_b = 0.2
    mu_a = -2
    mu_b = 2

    # Posterior probabilities (Bayes)
    probability_a = 0.5
    probability_b = 0.5

    fig, ax = plt.subplots(figsize=(15, 4))
    subplot_number = 0

    # Loop
    for iteration in range(1, max_iters + 1):
        subplot_number = subplot_number + 1
        subplot(1, max_iters + 1, subplot_number)
        plot_true_and_estimated_distributions(
            [
                ([mu_a_true, mu_a], [sigma_a_true, math.sqrt(sigma_square_a)
                ([mu_b_true, mu_b], [sigma_b_true, math.sqrt(sigma_square_b)
            ]
        )

        list_a = []
        list_b = []
        for index, i in enumerate(range(0, n)):
            # Coloring
            x_i = list_x[index]
            b_i = probability_a_or_b_given_x_i(sigma_square_a, sigma_square_
            list_b.append(b_i)
            list_a.append(1 - b_i)

        # Reestimation of Gaussian parameters
        mu_b = mu_a_or_b(list_b, list_x)
        sigma_square_b = estimated_sigma_square(list_a=list_b, list_x=list_x

        mu_a = mu_a_or_b(list_a, list_x)
        sigma_square_a = estimated_sigma_square(list_a=list_a, list_x=list_x

    subplot_number = subplot_number + 1
    subplot(1, max_iters + 1, subplot_number)
    plot_true_and_estimated_distributions(
        [
            ([mu_a_true, mu_a], [sigma_a_true, math.sqrt(sigma_square_a)]),
            ([mu_b_true, mu_b], [sigma_b_true, math.sqrt(sigma_square_b)])
        ]
    )

    return mu_a, math.sqrt(sigma_square_a), mu_b, math.sqrt(sigma_square_b)

expectation_maximization(list_x)
```

Out[8]:

```
(-7.673474472073881, 2.6071601631712475, 5.1594191503914955, 1.044169100352003)
```



In [ ]: 1