Vincent LIU MVA PGM HW2

Noisy Ising

In [1]:
```
import numpy as np
from PIL import Image
from scipy.special import logsumexp
import matplotlib.pyplot as plt

from tqdm import tqdm
```

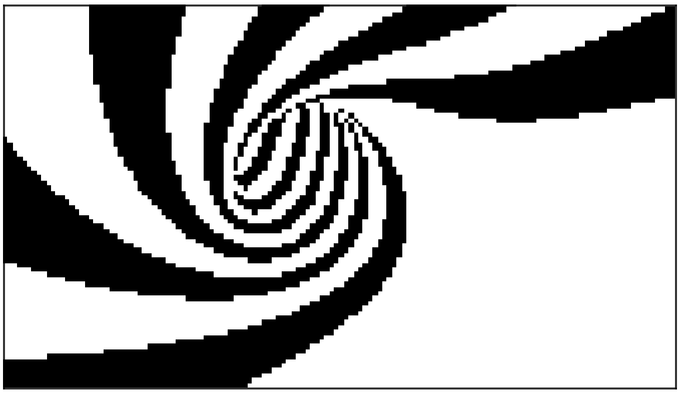
Open original image

We open the image provided by the teacher for this exercise.

We downsample the image preserving aspect ratio, in order to avoid long computation.

In [2]:
```
img = Image.open("spiral.png").resize((140, 80)) # Resize, o.w. computation t

# Plotting
plt.imshow(img, cmap='gray')
plt.title('Original image')
plt.xticks([])
plt.yticks([])
plt.show()
```

Original image



We convert the array representing our RGB image to an array representing a gray scale image with values between 0.0 and 1.0.

In [3]:
```
# Convert image to grayscale
x = np.asarray(img).mean(axis=2)

# Rescale so y = [0, 1]
x = x / 255.
x = x.astype(int)

# Array dims
height, width = x.shape
```

Adding noise

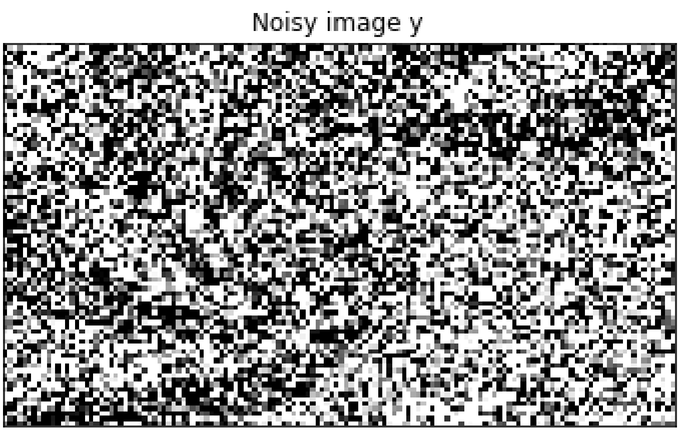
We generate a noisy image using the fact that conditional on $x_i = l$, each y_i are distributed according to a Gaussian distribution $N(\mu_l, 1)$.

We set μ_0 and μ_1 in an arbitrary way.

In [4]:
```
mu0 = 0.2
mu1 = 0.8
```

In [5]:
```
def apply_threshold(y_noisy):
    """
        Therefore, to visualize the image, we threshold the negative value
        to zero and the positive value to one.
    """
    y_noisy_copy = y_noisy.copy()

    y_noisy_copy[y_noisy_copy<0] = 0
    y_noisy_copy[y_noisy_copy>1] = 1
    return y_noisy_copy
```

In [6]:
```
# Generate noise to the original image

y_noisy = np.zeros(x.shape)

idx0 = (x == 0)
idx1 = (x == 1)

y_noisy[idx0] = np.random.normal(mu0, 1, x[idx0].shape)
y_noisy[idx1] = np.random.normal(mu1, 1, x[idx1].shape)

y_noisy_flatten = y_noisy.reshape(-1)

# plotting
plt.imshow(apply_threshold(y_noisy), cmap='gray')
plt.title('Noisy image y')
plt.xticks([])
plt.yticks([])
plt.show()
```



PGM Class

We create a Node class:

- A node instance is identified by its index *idx*.
- It contains a list of its neighbors' index *neighbors*.

We create a PGM class:

- Its purpose its to encapsulate a list of nodes X.
- When creating a PGM instance, we initialize the list of nodes X with all the corresponding neighbors.

In [7]:
```
class Node(object):
    """
        Node object to keep tracks of the node and its neighbors.
    """
    def __init__(self, idx):
        self.neighbors = []
        self.idx = idx

    def __repr__(self):
        return "{classname}({name}, [{neighbors}])".format(
            classname=type(self).__name__,
            name=str(self.idx),
            neighbors=', '.join([str(n.idx) for n in self.neighbors])
        )

    def add_neighbor(self, neighbor):
        self.neighbors.append(neighbor)
```

In [8]:
```
class PGM(object):
    """
        PGM to store data y, nodes X.
    """
    def __init__(self, data):
        """
            Store data and dimensions (height, width).
            Create list of (height x width) nodes.
        """
        self.height, self.width = data.shape

        self.X = []

        # Nodes
        for i in range(self.height * self.width):
            self.X.append(Node(i))

        # Add neighbors
        for h in range(self.height):
            for w in range(self.width):
                i =  w + self.width * h
                if w < width - 1:
                    self.X[i].add_neighbor(self.X[i+1])
                if w > 0:
                    self.X[i].add_neighbor(self.X[i-1])
                if h > 0:
                    self.X[i].add_neighbor(self.X[i-self.width])
                if h < height - 1:
                    self.X[i].add_neighbor(self.X[i+self.width])
```

Question 1: Belief propagation

Answer

We use **loopy belief propagation** to the problem of computing the distribution of a given x_i , conditional on all the y_i's. The reason is that there are loops on the graph. Contrarily to the junction tree algorithm, loopy belief propagation is a **non-exact** algorithm, it allows only to approximate the true distribution. However, according to the professor, it works well in practice !

We write the joint distribution $p(x)$ as a product of potential functions.

$$\begin{aligned} p(x) &= \frac{1}{Z_{\alpha,\beta}} exp\left(\alpha \sum_{i=1}^{n} x_i + \beta \sum_{(i,j)\in E} 1(x_i = x_j)\right) \\ &= \frac{1}{Z_{\alpha,\beta}} \prod_{i=1}^{n} exp(\alpha x_i) \prod_{(i,j)\in E} exp(\beta 1(x_i = x_j)) \\ &= \frac{1}{Z_{\alpha,\beta}} \prod_{i=1}^{n} \Psi_i(x_i) \prod_{(i,j)\in E} \Psi_{i,j}(x_i, x_j) \end{aligned}$$

where we have defined:

$$\begin{aligned} \Psi_i(x_i) &= exp\left(\alpha x_i\right) \\ \Psi_{i,j}(x_i, x_j) &= exp\left(\beta 1(x_i = x_j)\right) \end{aligned}$$

The probability of x_k given all the Y_k's can be written as a sum of $p(x|Y_k, \ldots, Y_n)$ over all variables except x_k:

$$\begin{aligned} p(x_k|Y_1, \ldots, Y_n) &= \sum_{x\backslash\{x_k\}} p(x|Y_1, \ldots, Y_n) \\ &\propto \sum_{x\backslash\{x_k\}} p(x)p(Y_1, \ldots, Y_n|x) \\ &\propto \sum_{x\backslash\{x_k\}} \left(p(x) \prod_{j=1}^{n} p(Y_j|x)\right) \\ &\propto \sum_{x\backslash\{x_k\}} \left(p(x) \prod_{j=1}^{n} p(Y_j|x_j)\right) \\ &\propto \sum_{x\backslash\{x_k\}} \left(p(x) \prod_{j=1}^{n} N(Y_j|\mu_{x_j}, 1)\right) \end{aligned}$$

where we have used Bayes' rules, the fact that the Y_j are independant, and Y_j given x_j is independent of all the others $x_{j'}$.

Since the Y_k's are observed, we can either use clamping or substitute Y_k by its observed value $\hat{y}_k$ (node removing method). We have:

$$\begin{aligned} p(x_k|Y_1 = \hat{y}_1, \ldots, Y_n = \hat{y}_n) &\propto \left(\sum_{x\backslash\{x_k\}} \prod_{j=1}^{n} \Psi_j(x_j) \prod_{(i,j)\in E} \Psi_{i,j}(x_i, x_j) \prod_{j=1}^{n} N(Y_j = \hat{y}_j|\mu_{x_j}, 1)\right) \\ &\propto N(Y_k = \hat{y}_k|\mu_{x_k}, 1)\psi_k(x_k) \sum_{x\backslash\{x_k\}} \left(\prod_{j\neq k} \Psi_j(x_j) \prod_{(i,j)\in E} \Psi_{i,j}(x_i, x_j) \prod_{j\neq k} N(Y_j = \hat{y}_j|\mu_{x_k}, 1)\right) \\ &\propto N(Y_k = \hat{y}_k|\mu_{x_k}, 1)\psi_k(x_k) \prod_{j\in ne(k)} \mu_{j->k}(x_k) \end{aligned}$$

where we have defined the message from node j to node k as follows:

$$\mu_{j->k}(x_k) = \sum_j \Psi_{j,k}(x_j, x_k)\Psi_j(x_j)N(Y_j = \hat{y}_j|\mu_{x_j}, 1) \prod_{i \in ne(j)\setminus\{k\}} \mu_{i->j}(x_j)$$

As a side note, we can write explicitly the discrete values taken by the potential functions $\Psi_i(x_i)$ and $\Psi_{i,j}(x_i, x_j)$:

$$\Psi_i(x_i = 0) = exp(0)$$
$$\Psi_i(x_i = 1) = exp(\alpha)$$

and

$$\Psi_{i,j}(x_i = 0, x_j = 0) = exp(\beta)$$
$$\Psi_{i,j}(x_i = 0, x_j = 1) = exp(0)$$
$$\Psi_{i,j}(x_i = 1, x_j = 0) = exp(0)$$
$$\Psi_{i,j}(x_i = 1, x_j = 1) = exp(\beta)$$

In [9]:

```
def psi_j(alpha):
    """
        Potential function associed to a singe node (j).
        Xsi_j(x_j=0) = exp(0)
        Xsi_j(x_j=1) = exp(alpha)
        [0, alpha]
    """
    return np.array([0, alpha])

def psi_jk(beta):
    """
        Potiential function associed to a pair of nodes (j, k)
                    (x_j=0)         (x_j=1)
        (x_k=0)     exp(beta)       exp(0)
        (x_k=1)     exp(0)          exp(beta)
    """
    return np.array([[beta, 0],
                     [0, beta]])

def send_messages(y_noisy_flatten, message_history, sender, receiver, alpha,
    """
        Send a message from node 'sender' j to node 'receiver' k.
        message_{j->k}(k) =
        Sum_j Xsi_j(x_j) * Xsi_jk(x_j, x_k) * Prod(message_{i->j}s)
    """
    incoming_msg = []
    for neighbor in sender.neighbors:
        if neighbor.idx != receiver.idx:
            msg_name = (neighbor.idx, sender.idx)
            msg = message_history[msg_name]
            incoming_msg.append(msg)

    new_message = np.sum(incoming_msg, axis=0)
    new_message = new_message + psi_j(alpha) + psi_jk(beta)
    new_message = new_message - 0.5 * (y_noisy_flatten[sender.idx] - mu) ** 2

    # Remarques du cours
    # log(e**a + e**b) = a + np.log(1+np.exp(b-a))  if a > b
    #                  = b + np.log(np.exp(a-b)+1)  if b < a

    return logsumexp(new_message, axis=1)
```

```
# https://stackoverflow.com/questions/42599498/numercially-stable-softmax
def stable_softmax(x):
    z = x - max(x)
    numerator = np.exp(z)
    denominator = np.sum(numerator)
    softmax = numerator/denominator

    return softmax

def belief_propagation(model, y_noisy, n_iters=4, alpha=0.1, beta=0.6, mu=[0.
    """
        Loopy Belief propagation.
        Create a message history to store the messages.
        Initialize the messages.
        Loop for n_iters to exchange messages.
    """

    height, width   = y_noisy.shape
    y_noisy_flatten = y_noisy.reshape(-1)

    # Output: probability distribution of the image
    P_X_given_y = np.zeros((width * height, 2))

    # Each element is a key-value pair representing a message i->j.
    # The keys are a tuple of int idx (i, j).
    # The value is a numpy array corresponding LOG probability distribution (
    message_history = {}

    # For each node, we iterate over its neighbors to initialize each message
    for node in model.X:
        for neighbor in node.neighbors:
            message_name = (node.idx, neighbor.idx)
            message_history[message_name] = np.array([0.1, 0.1])

    # Loopy Message Passing
    for i in range(n_iters):
        # At each iteration,
        # we create a dictionary of the history of the message passing
        new_message_history = dict()
        # For over each node,
        # we iterate over its neighbors in order to send a message
        for sender in tqdm(model.X):
            for receiver in sender.neighbors:
                message_name = (sender.idx, receiver.idx)
                # We use the message_history from previous iteration to send
                # We update the new message_history to be used in the next it
                new_msg = send_messages(y_noisy_flatten, message_history,
                                        sender, receiver, alpha, beta, mu)
                new_message_history[message_name] = new_msg
        # Replace by the new_message_history
        message_history = new_message_history

    # Compute the probability distribution p
    for idx, x in enumerate(model.X):
        # Sum over incoming messages
        incoming_messages = []
        for neighbor in x.neighbors:
            message_name = (neighbor.idx, x.idx)
            incoming_messages.append(message_history[message_name])

        p = np.sum(incoming_messages, axis=0)

        # Compute probability of P(X_k=0) and P(X_k=1)
        p = p +  psi_j(alpha) - 0.5 * (y_noisy_flatten[idx] - mu) ** 2
```

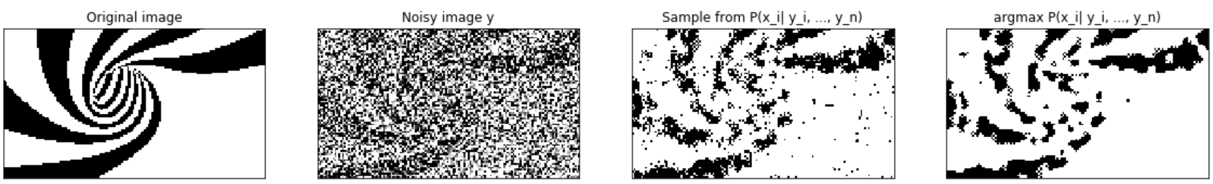
```
        P_X_given_y[idx, :] = stable_softmax(p)
    return P_X_given_y
```

In [10]:
```
def sample_from(P):
    """
        Sample from P.
    """
    N = P.shape[0]
    image_flatten = np.zeros((N))
    for i in range(N):
        image_flatten[i] = np.random.binomial(1, P[i, 1])
    return image_flatten

def sample_max_from(P):
    """
        Taking the most probable value from P.
    """
    N = P.shape[0]
    image_flatten = np.zeros((N))
    for i in range(N):
        image_flatten[i] = np.argmax(P[i, :])
    return image_flatten
```

In [11]:
```
model       = PGM(data=y_noisy)
P_X_given_y = belief_propagation(model, y_noisy, n_iters=4, alpha=0.1, beta=1
```

```
100%|██████████| 11200/11200 [00:04<00:00, 2325.43it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2309.99it/s]
100%|██████████| 11200/11200 [00:06<00:00, 1764.96it/s]
100%|██████████| 11200/11200 [00:05<00:00, 1955.13it/s]
```

In [12]:
```
sample     = sample_from(P_X_given_y)
sample_max = sample_max_from(P_X_given_y)

f, ax = plt.subplots(1, 4, figsize=(20, 8))

ims = [img,
       apply_threshold(y_noisy),
       sample.reshape((model.height, model.width)),
       sample_max.reshape((model.height, model.width)),]

titles = ['Original image',
          'Noisy image y',
          'Sample from P(x_i| y_i, ..., y_n)',
          'argmax P(x_i| y_i, ..., y_n)']

for i, (im, title) in enumerate(zip(ims, titles)):
    plt.sca(ax[i])
    plt.imshow(im, cmap='gray')
    plt.title(title)
    plt.xticks([])
    plt.yticks([])

plt.show()
```

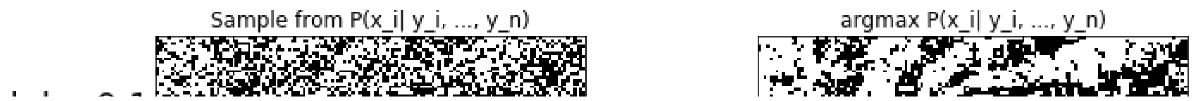


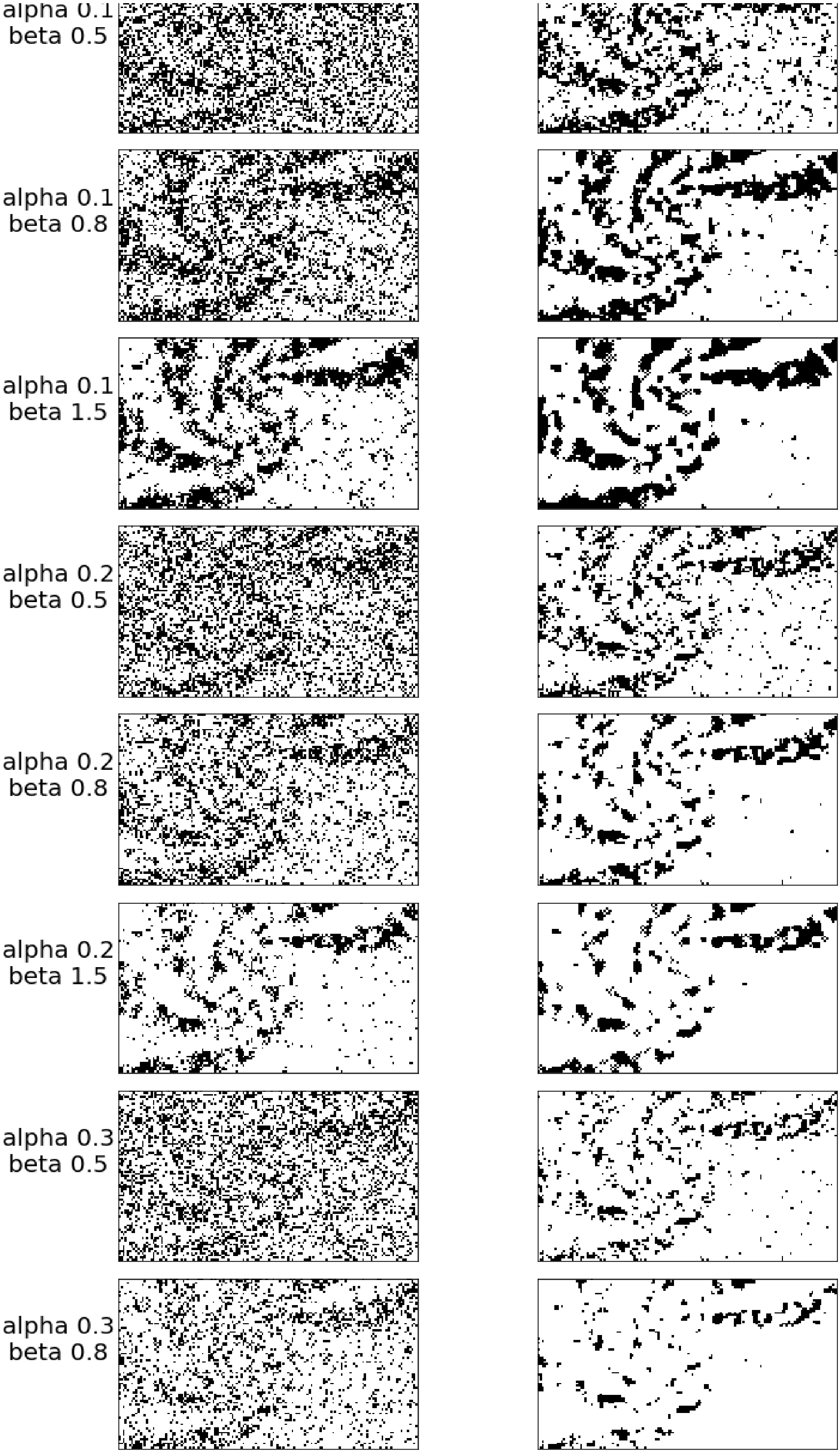
Other examples.

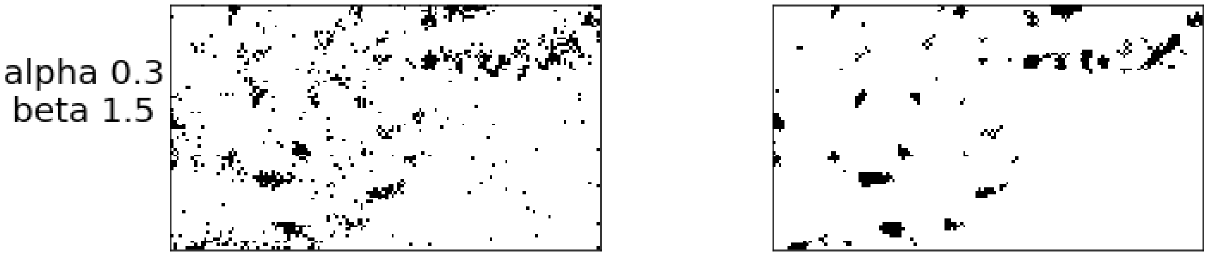
In [13]:
```
titles = ['Sample from P(x_i| y_i, ..., y_n)',
```

```
          'argmax P(x_i| y_i, ..., y_n)']

alphas = [0.1, 0.2, 0.3]
betas  = [0.5, 0.8, 1.5]

f, ax = plt.subplots(9, 2, figsize=(12, 20))

i = 0
for alpha in alphas:
    for beta in betas:
        model       = PGM(data=y_noisy)
        P_X_given_y = belief_propagation(model, y_noisy, n_iters=3,
                                         alpha=alpha, beta=beta, mu=[0.2, 0.8

        sample     = sample_from(P_X_given_y)
        sample_max = sample_max_from(P_X_given_y)

        for j, im in enumerate([sample.reshape((model.height, model.width)),
                                sample_max.reshape((model.height, model.width)),

            plt.sca(ax[i, j])
            plt.imshow(im, cmap='gray')
            plt.xticks([])
            plt.yticks([])

            if i == 0:
                plt.title(titles[j])

            if j == 0:
                plt.ylabel('alpha {}\nbeta {}'.format(alpha, beta),
                           rotation=0,  fontsize=20, labelpad=50)

        i += 1

plt.tight_layout()
plt.show()
```

```
100%|██████████| 11200/11200 [00:04<00:00, 2576.04it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2629.67it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2608.76it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2617.42it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2608.61it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2574.69it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2641.49it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2563.34it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2627.75it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2630.84it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2644.37it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2545.94it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2638.35it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2478.46it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2637.63it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2504.64it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2614.97it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2616.37it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2597.86it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2611.21it/s]
100%|██████████| 11200/11200 [00:05<00:00, 2063.29it/s]
100%|██████████| 11200/11200 [00:05<00:00, 1887.26it/s]
100%|██████████| 11200/11200 [00:06<00:00, 1642.60it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2611.00it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2583.98it/s]
100%|██████████| 11200/11200 [00:04<00:00, 2604.23it/s]
100%|██████████| 11200/11200 [00:09<00:00, 1216.55it/s]
```




alpha 0.1
beta 0.5
alpha 0.1
beta 0.8
alpha 0.1
beta 1.5
alpha 0.2
beta 0.5
alpha 0.2
beta 0.8
alpha 0.2
beta 1.5
alpha 0.3
beta 0.5
alpha 0.3
beta 0.8



We notice that we need a low value for alpha (0.1 or 0.2) and a higher value for beta (greater than 0.8) in order to perform denoising.

Question 2: MCMC

Answer

We use **MCMC Gibbs sampling**.

In this setting, we initialize a vector $x^{(0)}$. For a given time step t, we iteratively sample each component $x_k^{(t)}$ from $p(.\,|x_{-k}^{(t-1)}, Y)$. It is easier to sample from this distribution since $p(.\,|x_{-k}^{(t-1)}, Y)$ = $p(.\,|x_{ne(k)}^{(t-1)}, Y)$. Moreover, we can either use clamping or substitute Y by its observed value $\hat{y}$.

The joint distribution is:

$$\begin{aligned} p(x, Y=\hat{y}) &= p(x)p(Y=\hat{y}|x) \\ &= p(x_k, x_{-k})p(Y=\hat{y}|x) \\ &= \frac{1}{Z_{\alpha,\beta}} exp\left(\alpha \sum_{i=1}^{n} x_i + \beta \sum_{(i,j)\in E} 1(x_i = x_j)\right) \prod_{i=1}^{n} N(Y_i = \hat{y}_i|\mu_{x_i}, 1) \\ &= \frac{1}{Z_{\alpha,\beta}} \frac{1}{(2\pi)^{N/2}} exp\left(\alpha x_k + \beta \sum_{i\in ne(k)} 1(x_i = x_k) - \frac{1}{2}(\hat{y}_k - \mu_{x_k})^2 + \right. \\ &\qquad \left. \sum_{i\neq k} \alpha x_i + \beta \sum_{\substack{(i,j)\in E \\ \text{except } k}} 1(x_i = x_j) - \frac{1}{2}\sum_{i\neq k}(\hat{y}_i - \mu_{x_i})^2 \right) \end{aligned}$$

We sum the joint distribution over every state of x_k to obtain $p(x_{-k}, Y=\hat{y})$:

$$p(x_{-k}, Y=\hat{y}) = \sum_{x_k=0}^{1} p(x)p(Y=\hat{y}|x)$$

The two previous steps allow us to use Bayes' rules to obtain $p(x_k|x_{-k}, Y=\hat{y})$ (the distribution used in Gibbs sampling).

The terms that do not depend on x_k cancel each other.

$$\begin{aligned}
p(x_k|x_{-k}, Y=\hat{y}) &= \frac{p(x_k, x_{-k}, Y=\hat{y})}{p(x_{-k}, Y=\hat{y})} \\
&= \frac{exp\left(\alpha x_k + \beta \sum_{i\in ne(k)} 1(x_i = x_k) - \frac{1}{2}(\hat{y}_k - \mu_{x_k})^2\right)}{\sum_{x_k=0}^{1} exp\left(\alpha x_k + \beta \sum_{i\in ne(k)} 1(x_i = x_k) - \frac{1}{2}(\hat{y}_k - \mu_{x_k})^2\right)} \\
&= softmax\left(\alpha x_k + \beta \sum_{i\in ne(k)} 1(x_i = x_k) - \frac{1}{2}(\hat{y}_k - \mu_{x_k})^2\right)
\end{aligned}$$

Since we have only two states for x_k, we can use sigmoid but softmax is more general.

We remark that $p(x_k|x_{-k}, Y)$ does depend only on x_k neighbors and itself and can be interpreted as a bernoulli.

In [14]:
```
def P_X_i_given_markov_blanket(sample, node, y_noisy_flatten, alpha, beta, mu
    """
        Sample from P(x_i | x_{-i}, y) = P(x_i | x_{n(i)}, y).
    """
    # There is two states of x_i, either x_i = 0 or x_i = 1 so shape is 2 her
    p = np.zeros(2)

    # Recall that Neighbors = Markov Blanket in undirected graph.
    # Add the term depending on the neighbors.
    for neighbor in node.neighbors:
        p = p + beta * np.array([(sample[neighbor.idx] == 0),
                                 (sample[neighbor.idx] == 1)]).astype(i
    # Add the term depending on y_k.
    p = p - 0.5 * (y_noisy_flatten[node.idx] - mu)**2
    # Add own contribution.
    p = p + alpha * np.array([0, 1])
    return stable_softmax(p)

def Gibbs_Sampling(model, y_noisy, sample_init, n_iters,
                   alpha, beta, mu, return_history=False):
    """
        Sample from P(x) using a linear scan gibbs sampling.
    """

    height, width   = y_noisy.shape
    y_noisy_flatten = y_noisy.reshape(-1)

    P_X_given_y = np.ones((width*height, 2))
    sample      = (sample_init > 0.5).astype(int)

    if return_history:
        history = []
    else:
        history = None

    # Linear scan of n_iters iterations
    for _ in range(n_iters):
        for k, node in enumerate(model.X):
            P_X_given_y[k, :] = P_X_i_given_markov_blanket(sample,
                                                           node,
                                                           y_noisy_flatten,
                                                           alpha,
                                                           beta,
                                                           mu)
```

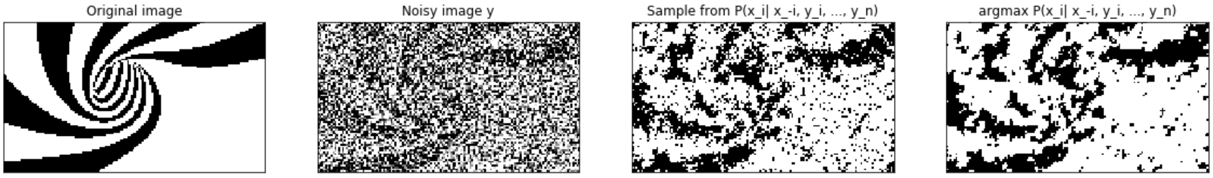
```
            # Sample from p(. | x^{(t-1)}_{-k}, y)
            sample[k]          = np.random.binomial(1, P_X_given_y[k, 1])
        if return_history and _ % 40 == 0:
            history.append(P_X_given_y.copy())
    return P_X_given_y, history
```

In [15]:

```
model                = PGM(data=y_noisy)
P_X_given_y, history = Gibbs_Sampling(model,
                                      y_noisy,
                                      sample_init=y_noisy_flatten.copy(),
                                      n_iters=180,
                                      alpha=0.05,
                                      beta=0.8,
                                      mu=[0.2, 0.8],
                                      return_history=True)
```

In [16]:

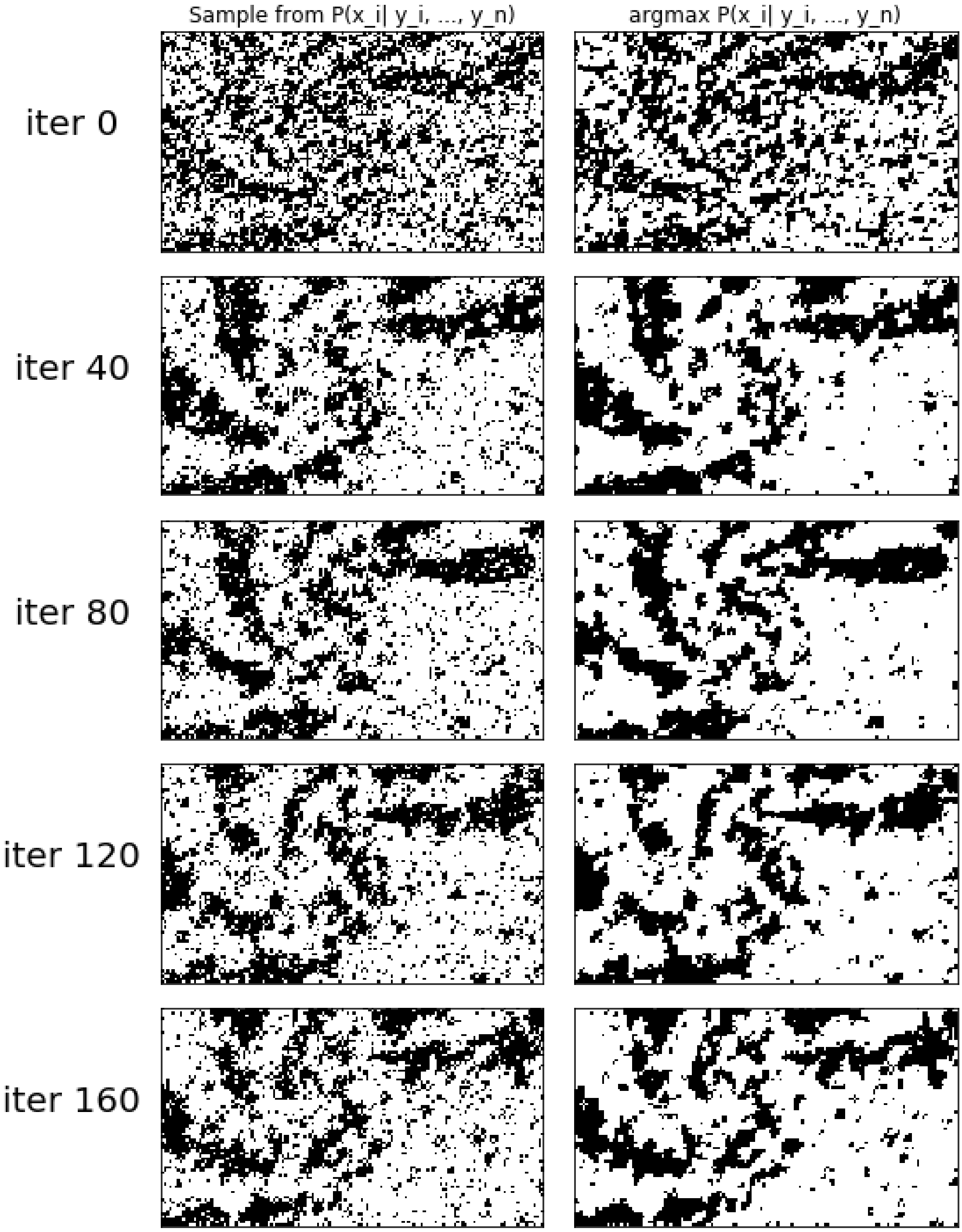
```
sample     = sample_from(P_X_given_y)
sample_max = sample_max_from(P_X_given_y)

f, ax = plt.subplots(1, 4, figsize=(20, 8))

ims = [img,
       apply_threshold(y_noisy),
       sample.reshape((model.height, model.width)),
       sample_max.reshape((model.height, model.width)),]

titles = ['Original image',
          'Noisy image y',
          'Sample from P(x_i| x_-i, y_i, ..., y_n)',
          'argmax P(x_i| x_-i, y_i, ..., y_n)']

for i, (im, title) in enumerate(zip(ims, titles)):
    plt.sca(ax[i])
    plt.imshow(im, cmap='gray')
    plt.title(title)
    plt.xticks([])
    plt.yticks([])

plt.show()
```



In [17]:

```
titles = ['Sample from P(x_i| y_i, ..., y_n)',
          'argmax P(x_i| y_i, ..., y_n)']

f, ax = plt.subplots(len(history), 2, figsize=(8, 10))

i = 0
for P_X_given_y in history:
    sample     = sample_from(P_X_given_y)
    sample_max = sample_max_from(P_X_given_y)

    for j, im in enumerate([sample.reshape((model.height, model.width)),
                            sample_max.reshape((model.height, model.width)),]):

        plt.sca(ax[i, j])
        plt.imshow(im, cmap='gray')
        plt.xticks([])
        plt.yticks([])
```

```
        if i == 0:
            plt.title(titles[j])

        if j == 0:
            plt.ylabel('iter {}'.format(i * 40),
                       rotation=0,  fontsize=20, labelpad=50)

    i += 1

plt.tight_layout()
plt.show()
```



Question 3: EM

α, β fixed. We want to learn μ_0, μ_1.

Answer

The latent variable are the X_i. Since we do not observe the X_i, our knowledge over X is only given by the posterior $p(X|Y)$.

The complete data lkelihood is:

$$p(X,Y) = \frac{1}{Z_{\alpha,\beta}} \frac{1}{(2\pi)^{N/2}} exp\left(\alpha \sum_{i=1}^{n} x_i + \beta \sum_{(i,j)\in E} 1(X_i = X_j) - \frac{1}{2}\sum_{i=1}^{n}(\hat{y}_i - \mu_{X_i})^2\right)$$

Therefore, the log likelihood is:

$$log(p(X,Y)) = \alpha \sum_{i=1}^{n} X_i + \beta \sum_{(i,j)\in E} 1(X_i = X_j) - \frac{1}{2}\sum_{i=1}^{n}(\hat{y}_i - \mu_{x_i})^2 + cst(\alpha,\beta)$$

In the E step, we compute the expectation of the complete data log likelihood under the posterior distribution over latent variable $p(x|y)$.

$$E\left(log(p(X,Y))\right) = \sum_{X} log(p(X,Y))p(X|Y)$$

In the M step, we maximize this quantity over the parameters μ_0 and μ_1.

Let $k = \{0,1\}$, we set the derivative w.r.t. μ_k to zero.

$$\nabla_{\mu_k} E\left(log(p(X,Y))\right) = 0$$
$$\nabla_{\mu_k}\left(-\frac{1}{2}\sum_{i=1}^{n}\sum_{x_i=0}^{1}(\hat{Y}_i - \mu_{x_i})^2 p(X_i = x_i|Y=\hat{y})\right) = 0$$
$$\sum_{i=1}^{n}(\hat{y}_i - \mu_k)p(X_i = k|Y=\hat{y}) = 0$$

Rearranging the terms, it gives:

$$\mu_k = \frac{\sum_{i=1}^{n}\hat{y}_i p(X_i = k|Y=\hat{y})}{\sum_{i=1}^{n} p(X_i = k|Y=\hat{y})}$$

We can use gibbs sampling to sample from $p(X_i = k|Y=\hat{y})$.

In [18]:
```
def expectation_complete_data_likelihood(y_noisy_flatten, P_X_given_y, mu):
    """
        Compute the expectation of the complete data likelihood.
        P(X|)
    """

    likelihood = 0
    for i in range(len(y_noisy_flatten)):
        likelihood -= (P_X_given_y[i, 0] * (y_noisy_flatten[i] - mu[0])**2) +
    return 0.5 * likelihood
```

In [19]:
```
def EM(model, y_noisy, n_iters_em=5, n_iters_gibbs=10, alpha=0.2, beta=1):
    """
        Computes Expectation propagation given alpha and beta.
    """
    height, width   = y_noisy.shape
```

```
    y_noisy_flatten = y_noisy.reshape(-1)

    P_X_given_y = np.ones((width*height, 2))
    likelihood_history = []
    checkpoints = np.zeros((n_iters_em, width*height, 2))
    mu_history = []

    # Initialization
    mu = np.array([0.25, 0.4])

    for i in tqdm(range(5)):
        # Expectation step
        P_X_given_y, _ = Gibbs_Sampling(model, y_noisy, y_noisy_flatten.copy(
        # Maximization step
        mu = np.dot(y_noisy_flatten, P_X_given_y) / P_X_given_y.sum(axis=0)
        # Lookup for likelihood
        likelihood = expectation_complete_data_likelihood(y_noisy_flatten, P_
        # stock results
        checkpoints[i, :] = P_X_given_y
        likelihood_history.append(likelihood)
        mu_history.append(mu)
    return checkpoints, likelihood_history, mu_history
```

In [20]:
```
model = PGM(data=y_noisy)
checkpoints, likelihood_history, mu_history = EM(model, y_noisy, n_iters_em=5
```

```
100%|██████████| 5/5 [09:11<00:00, 110.30s/it]
```

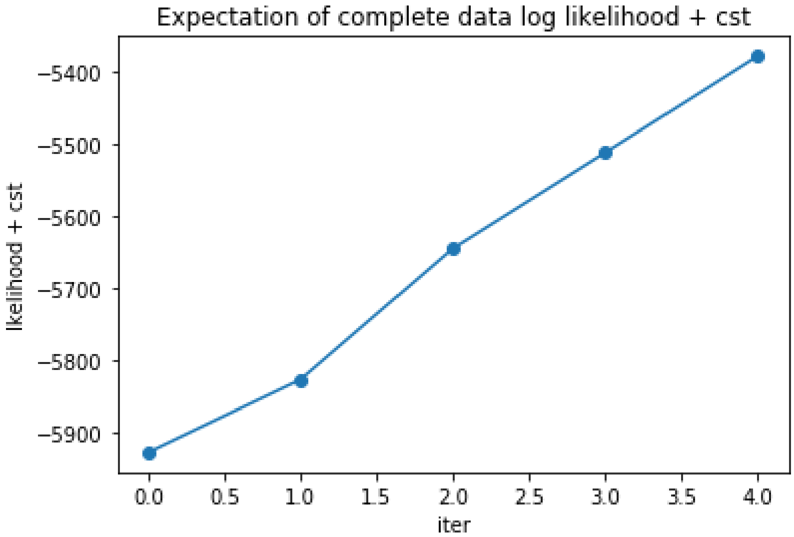
The estimation of the parameters is close to the real values.

In [21]:
```
mu_history
```

Out[21]:
```
[array([0.40334831, 0.64799045]),
 array([0.33319701, 0.70248885]),
 array([0.25819071, 0.77504167]),
 array([0.18649122, 0.8014005 ]),
 array([0.13827477, 0.83186635])]
```

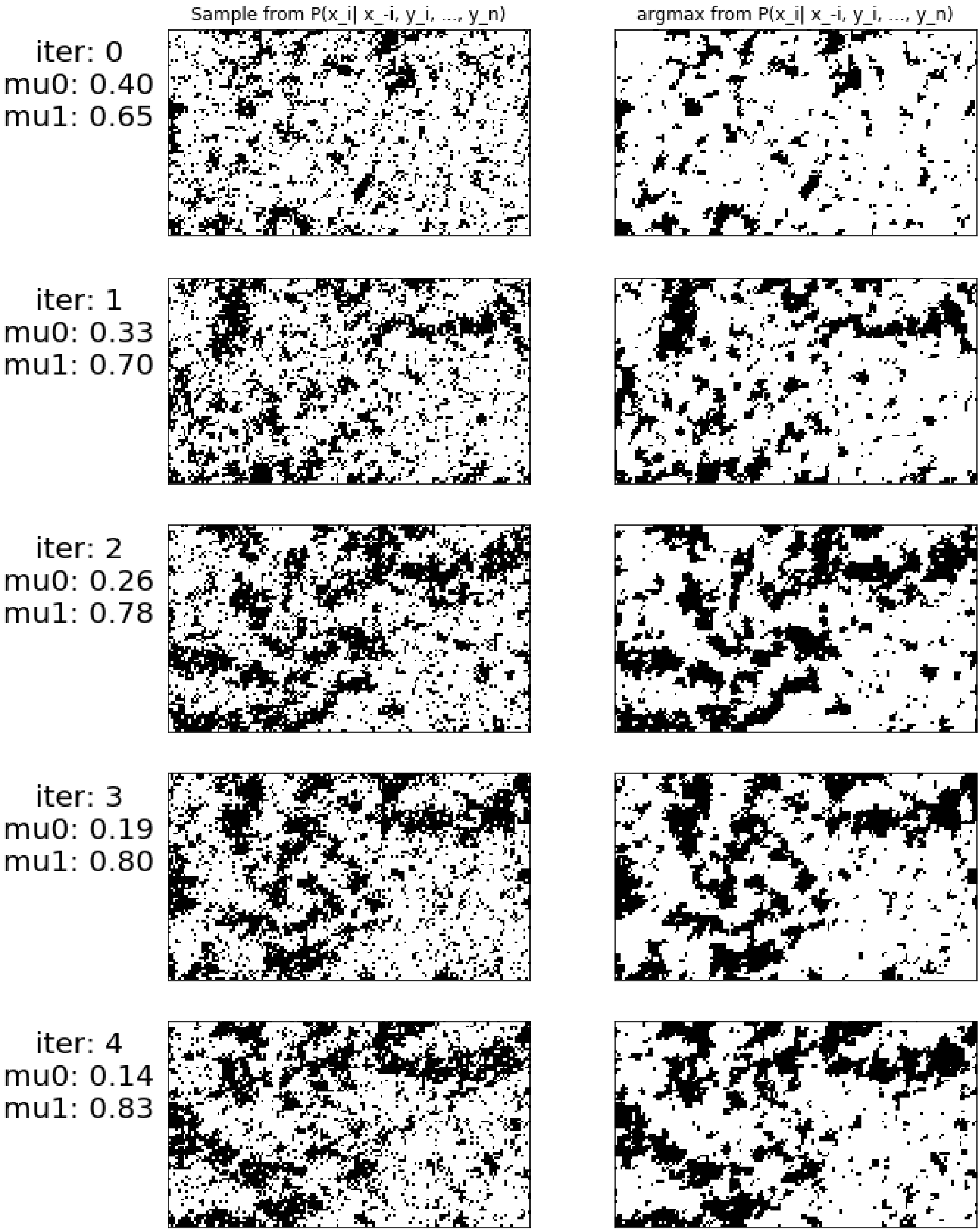
The complete data log likelihood is increasing at each iteration.

In [22]:
```
plt.plot(likelihood_history, 'o-')
plt.title('Expectation of complete data log likelihood + cst')
plt.xlabel('iter')
plt.ylabel('lkelihood + cst')
plt.show()
```



In [23]:
```
f, ax = plt.subplots(len(checkpoints), 2, figsize=(10, 15))
```

```
titles  = ['Sample from P(x_i| x_-i, y_i, ..., y_n)', 'argmax from P(x_i| x_-

for i, (ckpt, mu) in enumerate(zip(checkpoints, mu_history)):
    sample     = sample_from(ckpt)
    sample_max = sample_max_from(ckpt)
    for j, im in enumerate([sample, sample_max]):
        plt.sca(ax[i, j])
        plt.imshow(im.reshape((model.height, model.width)), cmap='gray')

        plt.xticks([])
        plt.yticks([])

        if i == 0:
            plt.title(titles[j])

        if j == 0:
            plt.ylabel('iter: {0:d}\nmu0: {1:0.2f}\nmu1: {2:0.2f}'.format(i,

plt.show()
```



Question 4

Answer

α appears in $\sum_{i=1}^{n} \alpha x_i$ and in $cst(\alpha)$.

β appears in $\sum_{i=1}^{n} \beta 1(x_i = x_n)$ and in $cst(\beta)$.

We may need to introduce a prior probability distribution over α, β.

We need then to sample from α, β.

There may be no close solution when taking the derivatives w.r.t α, β.

Question 5

Answer

We use Gibbs sampling to sample.

We sample iteratively from
$p(x_k|x_{-k}, \alpha, \beta, \mu_0, \mu_1, y), p(\alpha|x, \beta, \mu_0, \mu_1, y), p(\beta|x, \alpha, \mu_0, \mu_1, y), p(\mu_0|x, \alpha, \beta, \mu_1, y), p(\mu_1|$

Sample from mu

The joint probability is:

$$\begin{aligned} p(x, y, \mu_0, \mu_1, \alpha, \beta) &= p(\mu_0, \mu_1, \alpha, \beta)p(x, y|\mu_0, \mu_1, \alpha, \beta) && (1) \\ &= p(\mu_0, \mu_1, \alpha, \beta)p(x|\mu_0, \mu_1, \alpha, \beta)p(y|x, \mu_0, \mu_1) && (2) \end{aligned}$$

Therefore,

$$\begin{aligned} p(\mu_0|x, y, \mu_1, \alpha, \beta) &\propto p(x, y, \mu_0, \mu_1, \alpha, \beta) && (3) \\ &\propto N(\mu_0|m, s^2)\prod_{i=1}^{n} N(y_i|\mu_{x_i}, 1) && (4) \end{aligned}$$

In log scale:

$$log(p(\mu_0|x, y, \mu_1, \alpha, \beta)) + cst = \qquad (5)$$

$-\frac{1}{2}(\mu_0 - m)^2 - \frac{1}{2}\sum_{i=1}^{n}(y_i - \mu_{x_i})^2 =$
$-\frac{1}{2}(\mu_0 - m)^2 - \frac{s^2}{2s^2}\sum_{i=1}^{n}(1-x_i)(y_i - \mu_{x_i})^2 =$
$-\frac{1}{2s^2}\left(\mu_0^2 - 2\mu_0 m + m^2 + \sum_{i=1}^{n}(1-x_i)y_i^2 s^2 - 2\sum_{i=1}^{n}(1-x_i)y_i\mu_0 s^2 + \sum_{i=1}^{n}(1-x_i)\mu_0^2 s^2\right.$
$-\frac{1}{2s^2}\left(\mu_0^2\left(1 + \sum_{i=1}^{n}(1-x_i)s^2\right) - 2\mu_0\left(m + \sum_{i=1}^{n}(1-x_i)y_i\right) + cst\right) =$
$-\frac{1+\sum_{i=1}^{n}(1-x_i)s^2}{2s^2}\left(\mu_0^2 - 2\mu_0\frac{m+\sum_{i=1}^{n}(1-x_i)y_i}{1+\sum_{i=1}^{n}(1-x_i)s^2} + cst\right) =$

Using the complete the square trick, we can conclude without developing the part that does not depend on μ_0.

In order to sample μ_0, we have to draw a gaussian with mean:

$$\frac{m + \sum_{i=1}^{n}(1-x_i)y_i}{1 + \sum_{i=1}^{n}(1-x_i)s^2}$$

and variance:

$$\frac{s^2}{1 + \sum_{i=1}^{n}(1-x_i)s^2}$$

.

Similarly, in order to sample μ_1, we have to draw a gaussian with mean:

$$\frac{m + \sum_{i=1}^n x_i y_i}{1 + \sum_{i=1}^n x_i s^2}$$

and variance:

$$\frac{s^2}{1 + \sum_{i=1}^n x_i s^2}$$

.

Sample from alpha, beta

For α, we have:

$$\begin{align} p(\alpha|x, y, \mu_0, \mu_1, \beta) &\propto p(x, y, \mu_0, \mu_1, \alpha, \beta) \tag{6} \\ &\propto 1_{[0,a]}(\alpha) exp\left(\alpha \sum_{i=1}^n x_i\right) \tag{7} \end{align}$$

It can be written as follows:

$$\begin{align} p(\alpha|x, y, \mu_0, \mu_1, \beta) &= \frac{1_{[0,a]}(\alpha) exp\left(\alpha \sum_{i=1}^n x_i\right)}{\int 1_{[0,a]}(\alpha) exp\left(\alpha \sum_{i=1}^n x_i\right) d\alpha} \tag{8} \\ &= \frac{1_{[0,a]}(\alpha) exp\left(\alpha \sum_{i=1}^n x_i\right)}{\frac{exp(a \sum x_i) - 1}{\sum x_i}} \tag{9} \end{align}$$

We derive the cdf and the inv cdf, we will use inverse transform sampling.

The cdf can be written as:

$$\begin{align} F(y) &= \int_{-\infty}^{y} p(\alpha|x, y, \mu_0, \mu_1, \beta) d\alpha \tag{10} \\ &= \int_{-\infty}^{y} \frac{1_{[0,a]}(\alpha) exp\left(\alpha \sum_{i=1}^n x_i\right)}{\frac{exp(a \sum x_i) - 1}{\sum x_i}} d\alpha \tag{11} \\ &= \frac{\sum x_i}{exp(a \sum x_i) - 1} \int_0^{min(y,a)} exp\left(\alpha \sum_{i=1}^n x_i\right) d\alpha \tag{12} \\ &= \frac{1}{exp(a \sum x_i) - 1} \left[exp\left(\alpha \sum_{i=1}^n x_i\right)\right]_0^{min(y,a)} \tag{13} \\ &= \frac{exp\left(min(y,a) \sum_{i=1}^n x_i\right) - 1}{exp(a \sum x_i) - 1} \tag{14} \end{align}$$

The pseudo inverse cdf can be then derived:

$$\begin{align} U &= \frac{exp\left(min(y,a) \sum_{i=1}^n x_i\right) - 1}{exp(a \sum x_i) - 1} \tag{15} \\ F'(U) &= min\left[a, \frac{1}{\sum x_i} log\left(1 + U\left[exp(a \sum x_i) - 1\right]\right)\right] \tag{16} \end{align}$$

I am not sure about min(a, .) here.

We use the same method to sample β, the difference is that we interchange the hyper parameters a and b, and take $\sum_{(i,j)\in E} 1(x_i = x_j)$ instead of $\sum x_i$.

We use this prior in order to cancel out $Z_{\alpha,\beta}$ and prevent α, β to take too large value.

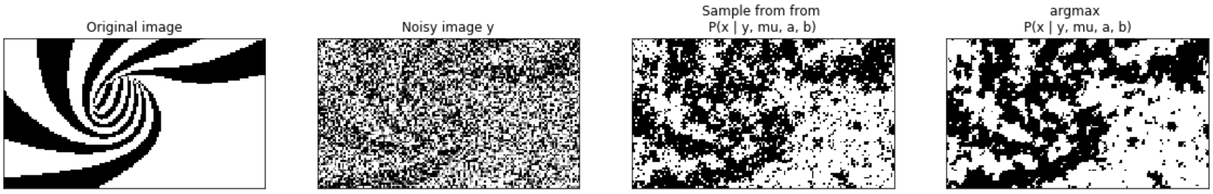
In [24]:
```
def sample_alpha(u, a, sample_init, model):
    return min(a, np.log(1 + u * (np.exp(a * np.sum(sample_init))-1)) / np.su

def sample_beta(u, b, sample_init, model):
    s = 0
    for node in model.X:
        for neighbor in node.neighbors:
            s += (sample_init[node.idx] == sample_init[neighbor.idx]).astype(
    s = s // 2
    return min(b, np.log(1 + u * (np.exp(b * s) - 1)) / s)

def Gibbs_Sampling_Revisited(model, y_noisy, sample_init, n_iters, s, m, a, b
    """
        Sample from P(x) using a linear scan gibbs sampling.
        Using prior knowledge pi(s, m, a, b).
    """

    height, width   = y_noisy.shape
    y_noisy_flatten = y_noisy.reshape(-1)

    P_X_given_y = np.ones((width*height, 2))
    sample      = (sample_init > 0.5).astype(int)

    if return_history:
        history = []
        history_parameters = []
    else:
        history = None
        history_parameters = None

    # Linear scan of n_iters iterations
    for _ in range(n_iters):
        # Sample mu_0 and mu_1
        mean0 = (m + np.dot(1-sample, y_noisy_flatten)) / (1 + np.sum(1-sampl
        mean1 = (m + np.dot(sample, y_noisy_flatten))   / (1 + np.sum(sample)

        var0  = s ** 2 / (1 + np.sum(1-sample) * s**2)
        var1  = s ** 2 / (1 + np.sum(sample) * s**2)

        mu = np.zeros(2)
        mu[0] = np.random.normal(mean0, var0**2)
        mu[1] = np.random.normal(mean1, var1**2)
        # Inverse transform sampling
        u = np.random.rand()

        alpha = sample_alpha(u, a, sample, model)
        beta  = sample_beta(u, b,  sample, model)

        for k, node in enumerate(model.X):
            P_X_given_y[k, :] = P_X_i_given_markov_blanket(sample, node, y_no
            # Sample from p(. | x^{(t-1)}_{-k}, y)
            sample[k]         = np.random.binomial(1, P_X_given_y[k, 1])

        if return_history and _ % 40 == 0:
            history.append(P_X_given_y.copy())
```

```
            history_parameters.append((mu[0], mu[1], alpha, beta))
    return P_X_given_y, history, history_parameters
```

In [25]:
```
model        = PGM(data=y_noisy)
P_X_given_y, history, history_parameters = Gibbs_Sampling_Revisited(model, y_
                                                                      n_iters=2
                                                                      s=1,
                                                                      m=0.05,
                                                                      a=0.001,
                                                                      b=0.8,
                                                                      return_hi
```

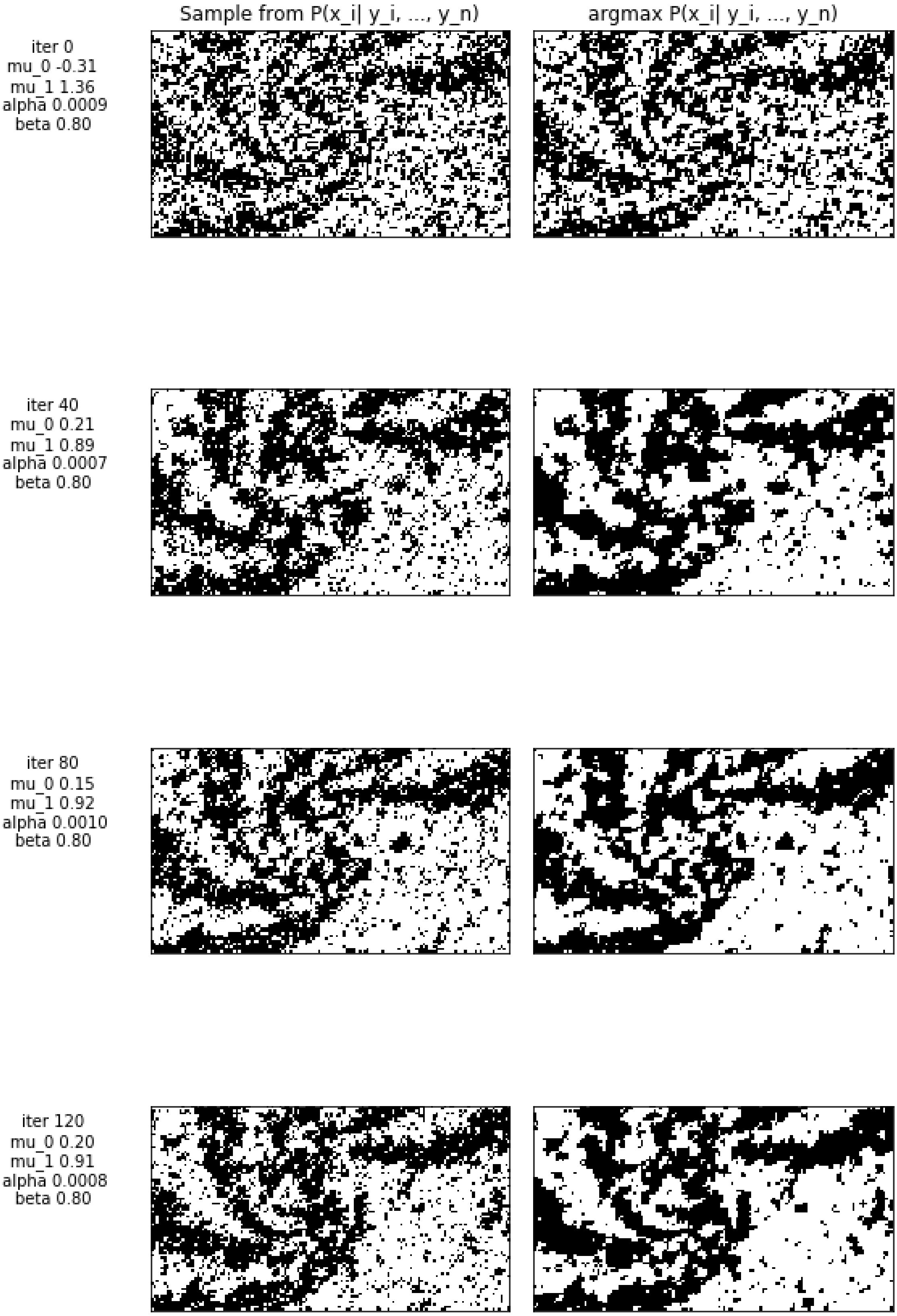
```
/home/v/.local/lib/python3.6/site-packages/ipykernel_launcher.py:10: RuntimeW
arning: overflow encountered in exp
  # Remove the CWD from sys.path while we load stuff.
```

In [26]:
```
sample     = sample_from(P_X_given_y)
sample_max = sample_max_from(P_X_given_y)

f, ax = plt.subplots(1, 4, figsize=(20, 8))

ims = [img,
       apply_threshold(y_noisy),
       sample.reshape((model.height, model.width)),
       sample_max.reshape((model.height, model.width)),]

titles = ['Original image',
          'Noisy image y',
          'Sample from from \nP(x | y, mu, a, b)',
          'argmax \nP(x | y, mu, a, b)']

for i, (im, title) in enumerate(zip(ims, titles)):
    plt.sca(ax[i])
    plt.imshow(im, cmap='gray')
    plt.title(title)
    plt.xticks([])
    plt.yticks([])

plt.show()
```

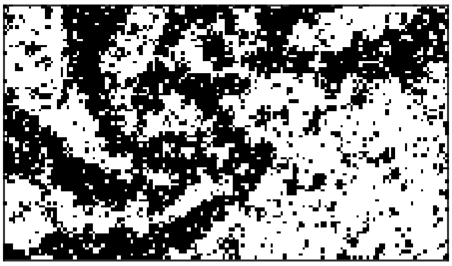


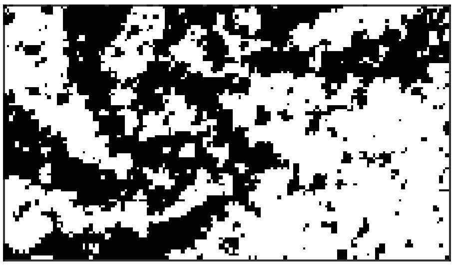
In [27]:
```
titles = ['Sample from P(x_i| y_i, ..., y_n)',
          'argmax P(x_i| y_i, ..., y_n)']

f, ax = plt.subplots(len(history), 2, figsize=(8, 16))

i = 0
for P_X_given_y, parameters in zip(history, history_parameters):
    sample     = sample_from(P_X_given_y)
    sample_max = sample_max_from(P_X_given_y)

    for j, im in enumerate([sample.reshape((model.height, model.width)),
                            sample_max.reshape((model.height, model.width)),]):

        plt.sca(ax[i, j])
        plt.imshow(im, cmap='gray')
        plt.xticks([])
        plt.yticks([])
```

```
        if i == 0:
            plt.title(titles[j])

        if j == 0:
            mu0_, mu1_, alpha_, beta_ = parameters
            plt.ylabel('iter {0:d}\nmu_0 {1:0.2f}\nmu_1 {2:0.2f}\n alpha {3:0

    i += 1

plt.tight_layout()
plt.show()
```



iter 160
mu_0 0.19
mu_1 0.91
alpha 0.0008
beta 0.80





In [ ]: