# exc_02_solution_2

April 30, 2018

## 1 Exercise Sheet 2

### 1.1 Exercise 4: Gibbs sampling

In [29]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from random import random
import h5py as h5

##########################################################################################################
# don't worry if you have not used python extensively before and if this is confusing
# it's only required for the nice progress bar
# if you want to see the progress bar on your computer you need to first run conda install tqd
# (https://github.com/tqdm/tqdm)
try:
    # try to import functions from TQDM
    from tqdm import tqdm_notebook as tqdm
    from tqdm import tnrange
except ModuleNotFoundError:
    # if TQDM is not installed this fallback code will run.
    # it just defines dummy functions that map back to vanilla python
    # *args and *kwargs are "catch-all" arguments, see e.g. http://thepythonguru.com/python-ar
    def tqdm(x, *args, **kwargs):
        return x
    def tnrange(a, *args, **kwargs):
        return range(a)
##########################################################################################################
```

In [30]:
```
def prior_l0(n_states):
    prior = np.ones((n_states, n_states))
    prior[np.where(np.eye(n_states))] = 0
    return prior

def prior_l1(n_states):
    prior = np.zeros((n_states, n_states))
    for i in range(n_states):
        for j in range(n_states):
            prior[i,j] = abs(i-j)
    return prior
```

In [31]:
```
def calc_proba(lattice, x, y, pairwise, unary=None):
    states = []
```

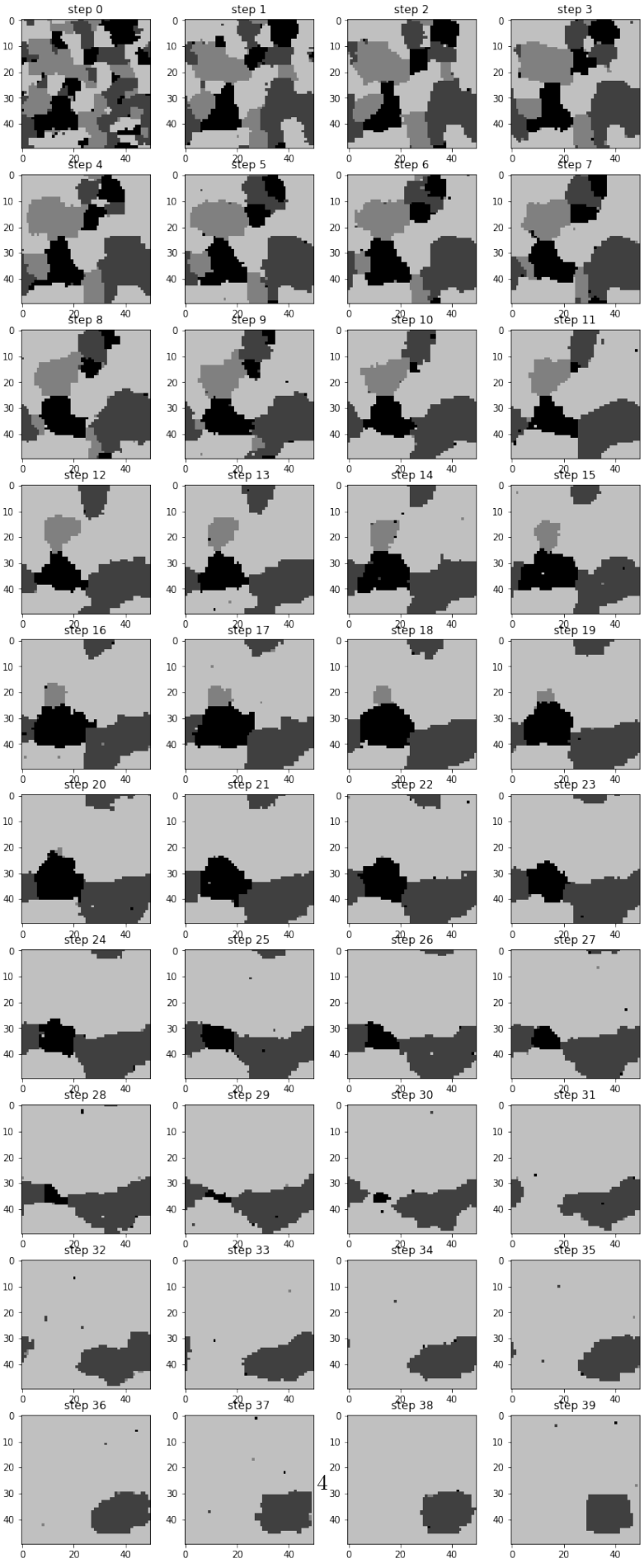
```
            nx, ny = lattice.shape

            state_left = lattice[(x-1)%nx,y]
            state_right = lattice[(x+1)%nx,y]
            state_top = lattice[x,(y-1)%ny]
            state_bottom = lattice[x,(y+1)%ny]

            neighbors = [state_left, state_right, state_bottom, state_top]

            for i in range(pairwise.shape[0]):
                res = 0
                for neighbor in neighbors:
                    res += pairwise[i, neighbor]

                res = np.exp(-res)

                ############ for b) where we sample from the posterior ################
                if not isinstance(unary, type(None)):
                    res *= unary[x, y, i]
                ###########################################################
                states.append(res)

            states /= np.sum(states)
            return states

In [32]: def gibbs_update(lattice, x, y, pairwise, unary=None):
            p = calc_proba(lattice, x, y, pairwise, unary)

            new_state = np.random.choice(pairwise.shape[0], p=p)
            lattice[x,y] = new_state
            #print(p)
            #print(f"update {x},{y} to {new_state}")

In [33]: def sweep_scanlines(lattice, pairwise, unary=None):
            for x in tnrange(lattice.shape[0], desc='x sweep', leave=False, position=1):
                for y in range(lattice.shape[1]):
                    gibbs_update(lattice, x, y, pairwise, unary)

         def sweep_scanlines_rnd(lattice, pairwise, unary=None):
            for x in tqdm(np.random.permutation(range(lattice.shape[0])), desc='x sweep', leave=False,
                for y in np.random.permutation(range(lattice.shape[1])):
                    gibbs_update(lattice, x, y, pairwise, unary)
```

### 1.1.1 a) Sampling from the prior

```
In [34]: n_steps_x = 10
         n_steps_y = 4
         n_states = 4
         n_x = 50
         n_y = 50
         alpha = 2

         lattice = np.random.randint(0, n_states, (n_x, n_y))
```

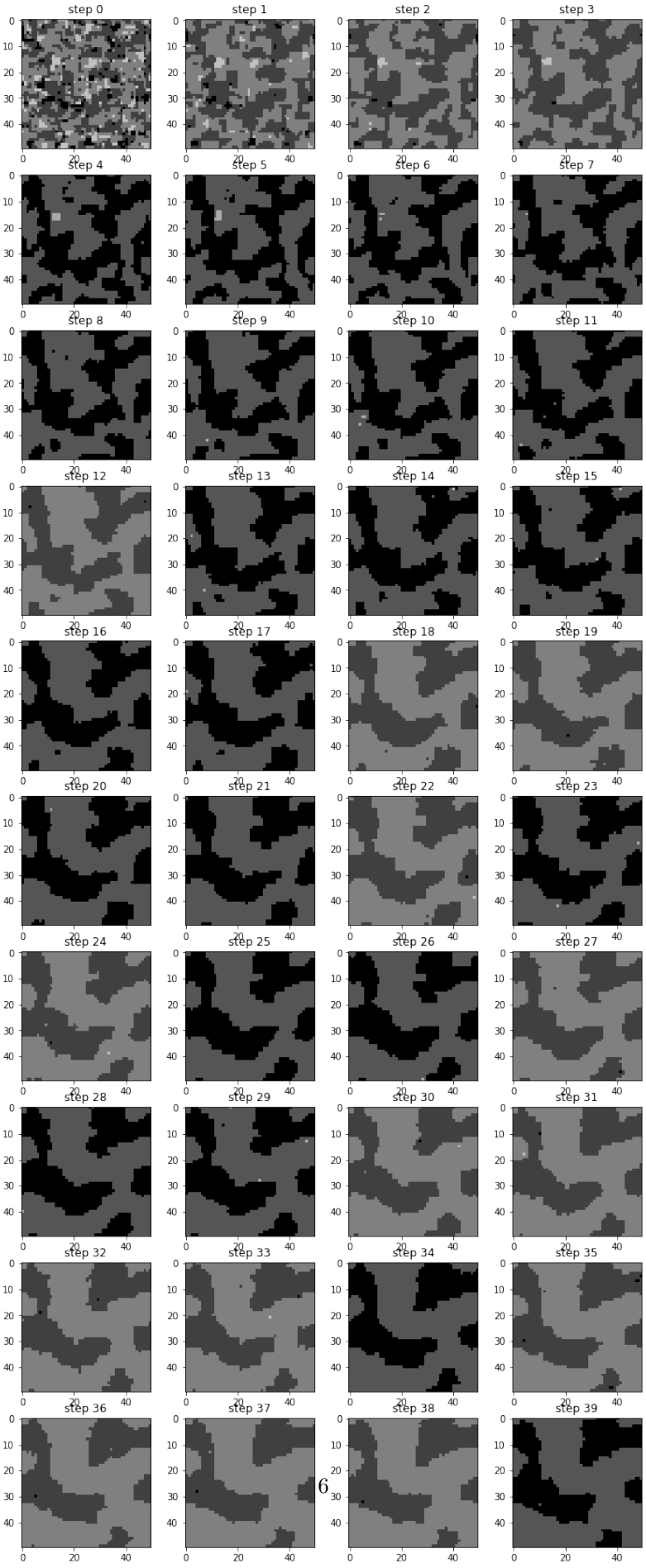
```
        prior = prior_l0(n_states)*alpha
        print(prior)

        f = plt.figure()
        f.set_size_inches((n_steps_y*3, n_steps_x*3))
        for i in tnrange(n_steps_x*n_steps_y):
            for _ in range(10):
                sweep_scanlines_rnd(lattice, prior)
            ax = f.add_subplot(n_steps_x, n_steps_y, i+1)
            ax.set_title("step "+str(i))
            ax.imshow(lattice, cmap='gray', vmax=n_states)
```

```
[[ 0.  2.  2.  2.]
 [ 2.  0.  2.  2.]
 [ 2.  2.  0.  2.]
 [ 2.  2.  2.  0.]]
```


step 0
step 1
step 2
step 3
step 4
step 5
step 6
step 7
step 8
step 9
step 10
step 11
step 12
step 13
step 14
step 15
step 16
step 17
step 18
step 19
step 20
step 21
step 22
step 23
step 24
step 25
step 26
step 27
step 28
step 29
step 30
step 31
step 32
step 33
step 34
step 35
step 36
step 37
step 38
step 39

In [35]:
```
n_steps_x = 10
n_steps_y = 4
n_states = 4
n_x = 50
n_y = 50
alpha = 2

lattice = np.random.randint(0, n_states, (n_x, n_y))
prior = prior_l1(n_states)*alpha
print(prior)

f = plt.figure()
f.set_size_inches((n_steps_y*3, n_steps_x*3))
for i in tnrange(n_steps_x*n_steps_y):
    for _ in range(1):
        sweep_scanlines_rnd(lattice, prior)
    ax = f.add_subplot(n_steps_x, n_steps_y, i+1)
    ax.set_title("step "+str(i))
    ax.imshow(lattice, cmap='gray', vmax=n_states)
```

```
[[ 0.  2.  4.  6.]
 [ 2.  0.  2.  4.]
 [ 4.  2.  0.  2.]
 [ 6.  4.  2.  0.]]
```


step 0
step 1
step 2
step 3
step 4
step 5
step 6
step 7
step 8
step 9
step 10
step 11
step 12
step 13
step 14
step 15
step 16
step 17
step 18
step 19
step 20
step 21
step 22
step 23
step 24
step 25
step 26
step 27
step 28
step 29
step 30
step 31
step 32
step 33
step 34
step 35
step 36
step 37
step 38
step 39

### 1.1.2 b) Sampling from the posterior

```
In [36]: pred_file = h5.File('predictions.h5','r')
         pred = pred_file['test'][...]
         pred_file.close()
         print(pred.shape)
```

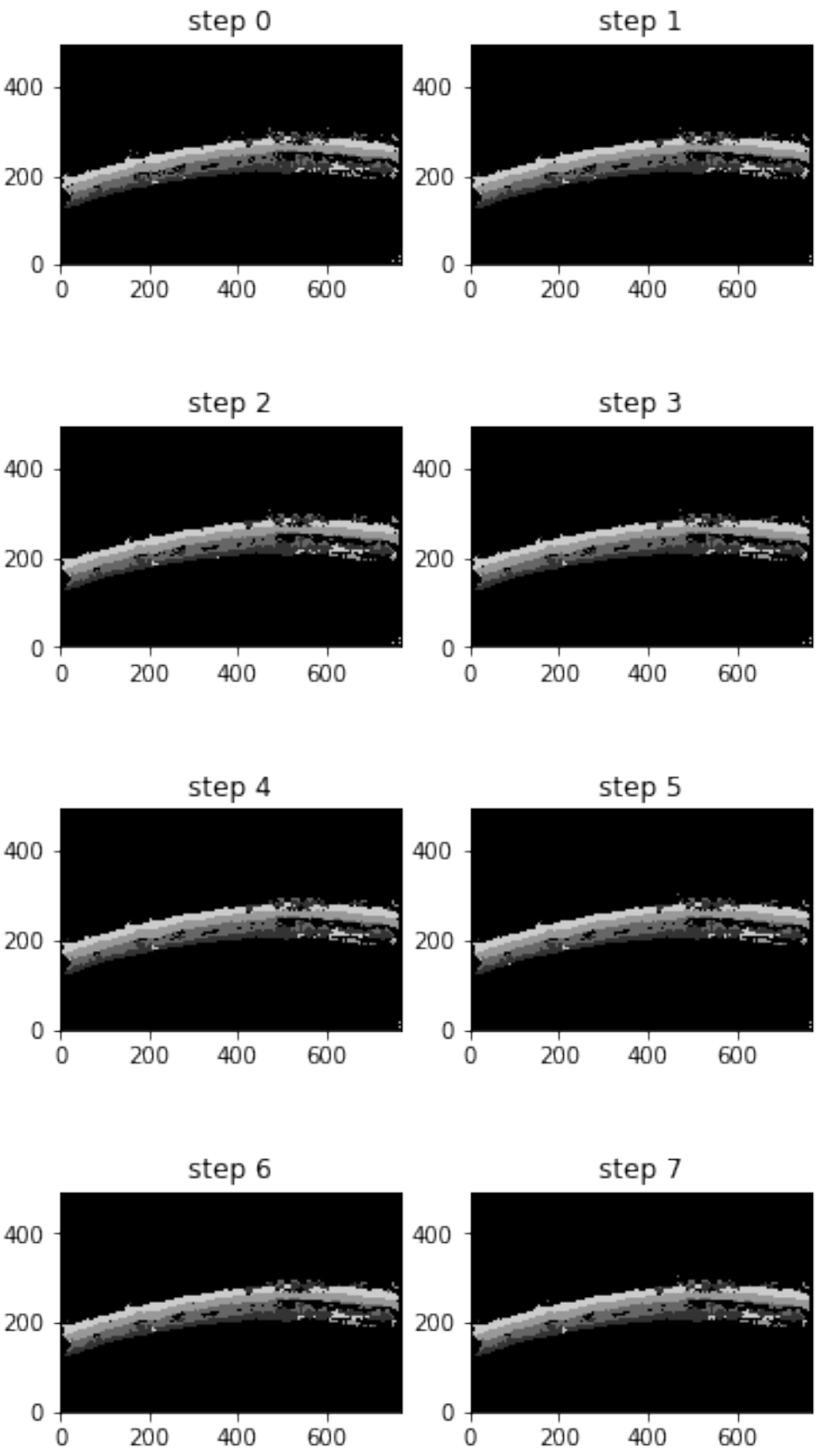
```
(496, 768, 5)
```

**Prior 1)**

```
In [37]: n_steps_x = 4
         n_steps_y = 2
         n_states = 5
         n_x = 40
         n_y = 40
         alpha = 2

         lattice = np.argmax(pred, axis=2).astype(np.int32)
         prior = prior_l0(n_states)*alpha
         print(prior)

         unary = pred

         samples = []


         for i in tnrange(n_steps_x*n_steps_y, desc='outer', position=0):
             sweep_scanlines_rnd(lattice, prior, unary)
             samples.append(lattice.copy())
```

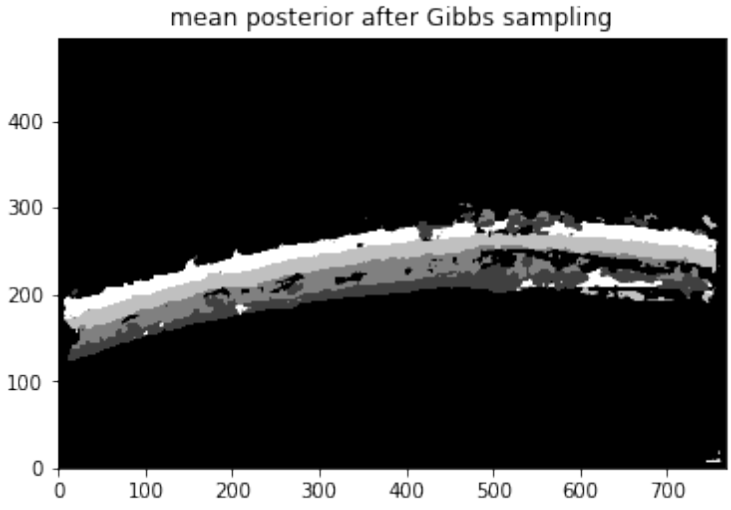
```
[[ 0.  2.  2.  2.  2.]
 [ 2.  0.  2.  2.  2.]
 [ 2.  2.  0.  2.  2.]
 [ 2.  2.  2.  0.  2.]
 [ 2.  2.  2.  2.  0.]]
```

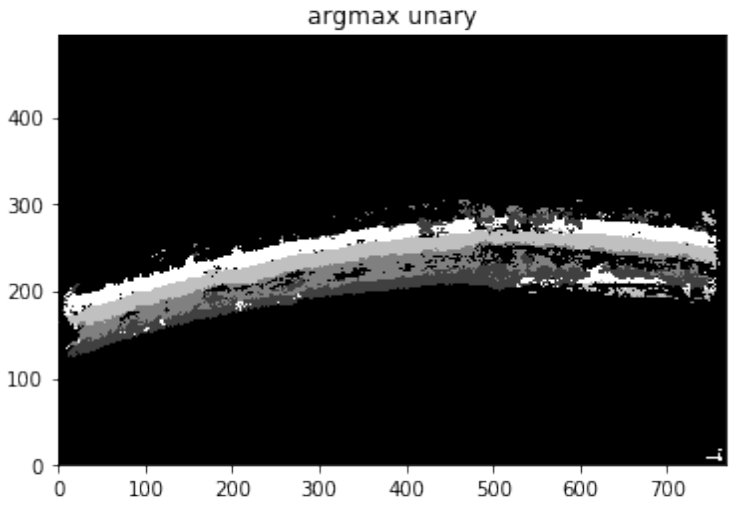
```
In [38]: f = plt.figure()
         f.set_size_inches((n_steps_y*3, n_steps_x*3))
         for i, sample in enumerate(samples):
             ax = f.add_subplot(n_steps_x, n_steps_y, i+1)
             ax.set_title("step "+str(i))
             ax.imshow(sample, cmap='gray', vmax=n_states,origin='lower')
```

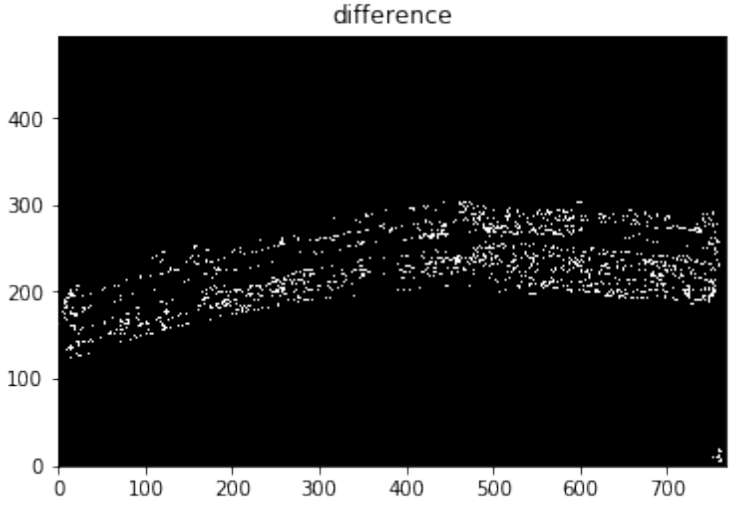

step 0
step 1
step 2
step 3
step 4
step 5
step 6
step 7
0
200
400
600

In [39]:
```
samples = np.array(samples)
samples_mean = np.round(np.mean(samples, axis=0))


unary_argmax =  np.argmax(pred, axis=2)
difference = np.zeros_like(unary_argmax)
difference[np.where(unary_argmax != samples_mean)] = 1
f = plt.figure()
f.set_size_inches(6,18)
ax = f.add_subplot(3,1,1)
ax.imshow(samples_mean, cmap='gray',origin='lower')
ax.set_title("mean posterior after Gibbs sampling")
ax = f.add_subplot(3,1,2)
ax.imshow(unary_argmax, cmap='gray',origin='lower')
ax.set_title("argmax unary")
ax = f.add_subplot(3,1,3)
ax.imshow(difference, cmap='gray',origin='lower')
ax.set_title("difference")
```

Out[39]:
```
<matplotlib.text.Text at 0x127ab99e8>
```

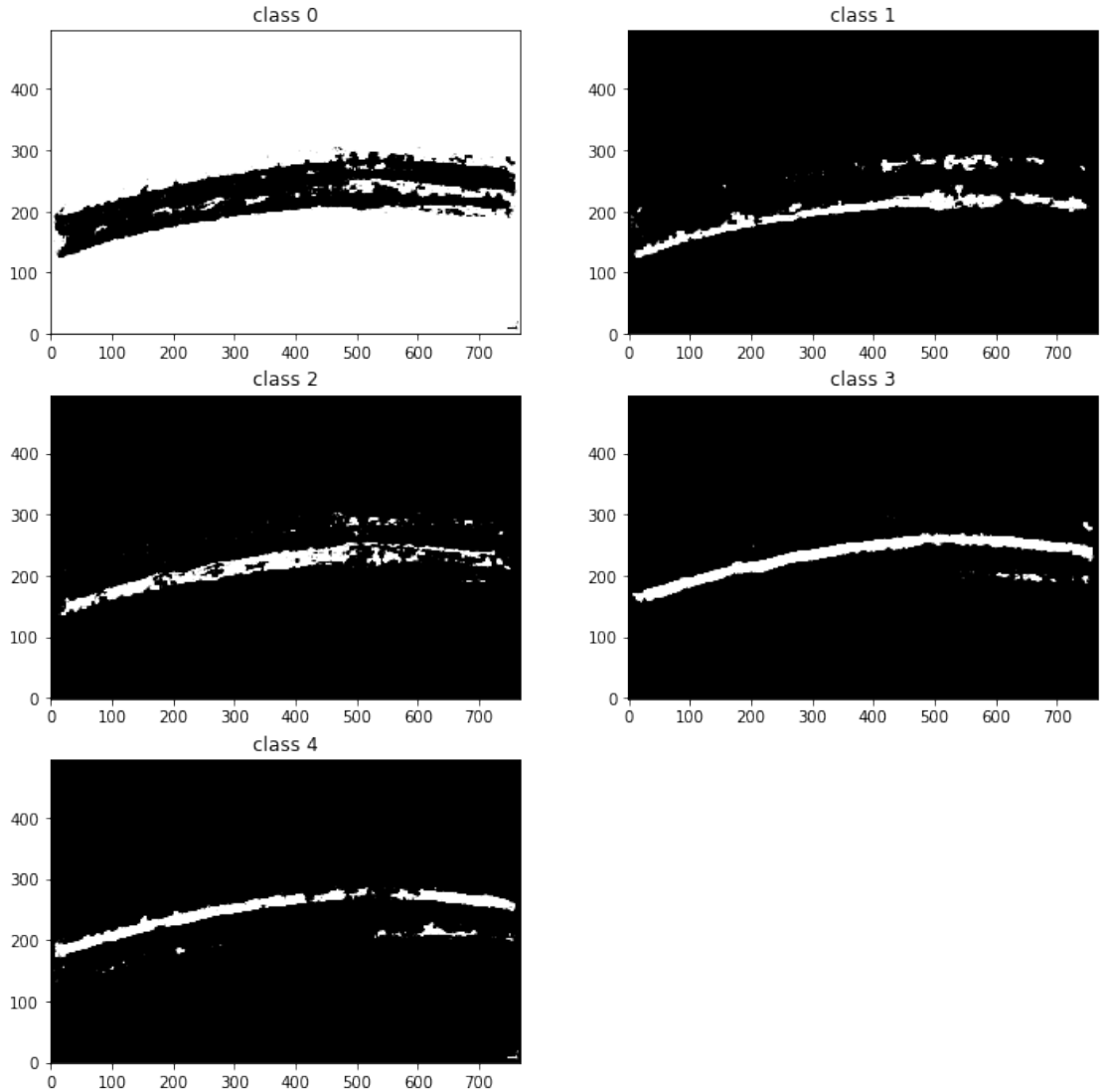

mean posterior after Gibbs sampling
0
100
200
300
400
0
100
200
300
400
500
600
700


argmax unary
0
100
200
300
400
0
100
200
300
400
500
600
700


difference
0
100
200
300
400
0
100
200
300
400
500
600
700

In [40]:
```
posterior_proba = np.zeros_like(unary, dtype=np.uint32)

for x in tnrange(posterior_proba.shape[0], desc='summing over rows'):
    for y in range(posterior_proba.shape[1]):
        for i in range(samples.shape[0]):
            posterior_proba[x,y, samples[i,x,y]] += 1
```

In [41]:
```
f = plt.figure()
f.set_size_inches((12, 12))
for i in range(5):
    ax = f.add_subplot(3,2,i+1)
    ax.imshow(posterior_proba[:,:,i], cmap='gray', vmax=posterior_proba.max(),origin='lower')
    ax.set_title("class "+str(i))
```

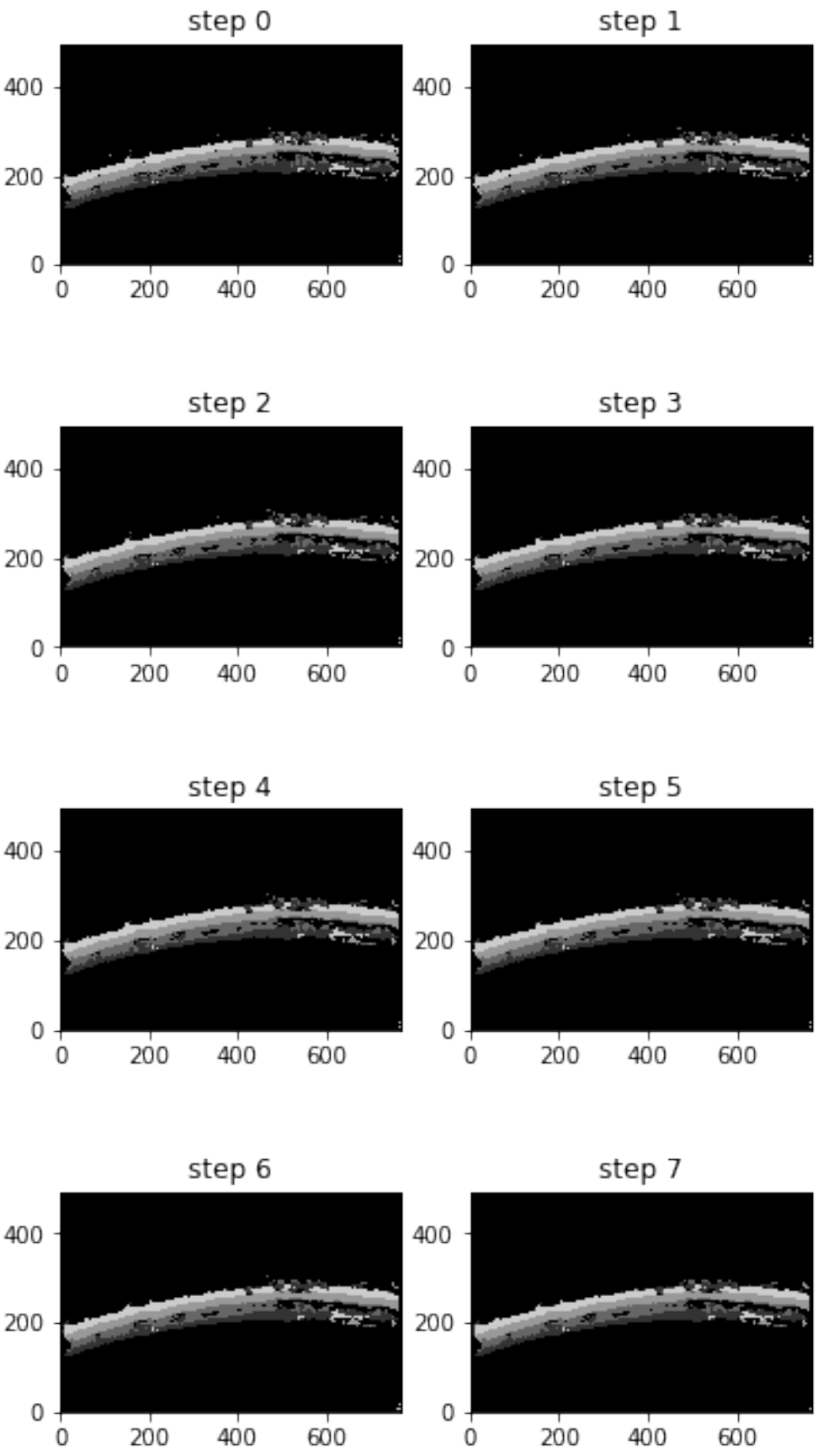


**Prior 2)**

In [42]:
```
n_steps_x = 4
n_steps_y = 2
n_states = 5
n_x = 40
n_y = 40
alpha = 2

lattice = np.argmax(pred, axis=2).astype(np.int32)
prior = prior_l1(n_states)*alpha
print(prior)

unary = pred

samples = []


for i in tnrange(n_steps_x*n_steps_y, desc='outer', position=0):
    sweep_scanlines_rnd(lattice, prior, unary)
    samples.append(lattice.copy())
```

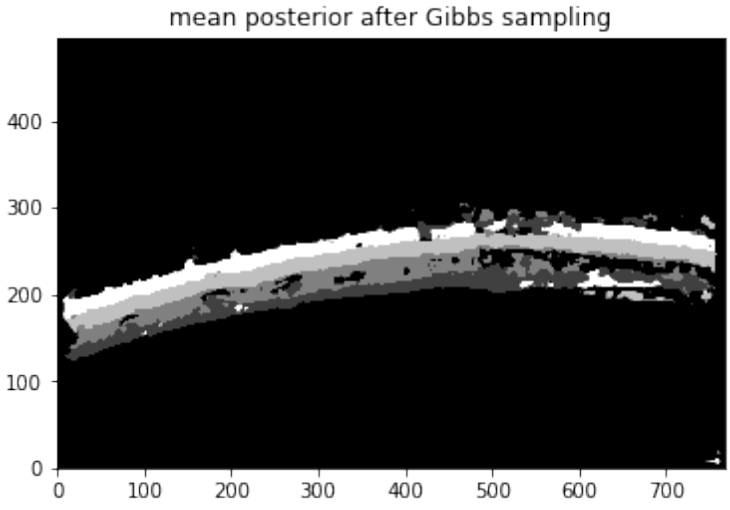
```
[[ 0.  2.  4.  6.  8.]
 [ 2.  0.  2.  4.  6.]
 [ 4.  2.  0.  2.  4.]
 [ 6.  4.  2.  0.  2.]
 [ 8.  6.  4.  2.  0.]]
```

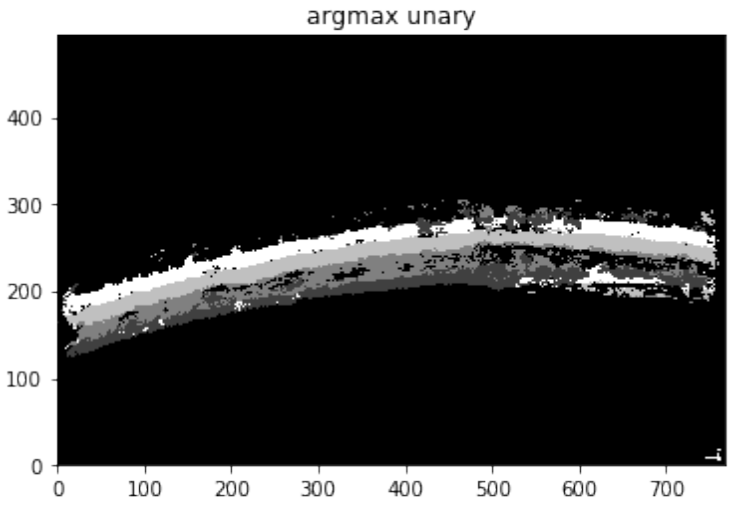
In [43]:
```
f = plt.figure()
f.set_size_inches((n_steps_y*3, n_steps_x*3))
for i, sample in enumerate(samples):
    ax = f.add_subplot(n_steps_x, n_steps_y, i+1)
    ax.set_title("step "+str(i))
    ax.imshow(sample, cmap='gray', vmax=n_states,origin='lower')
```

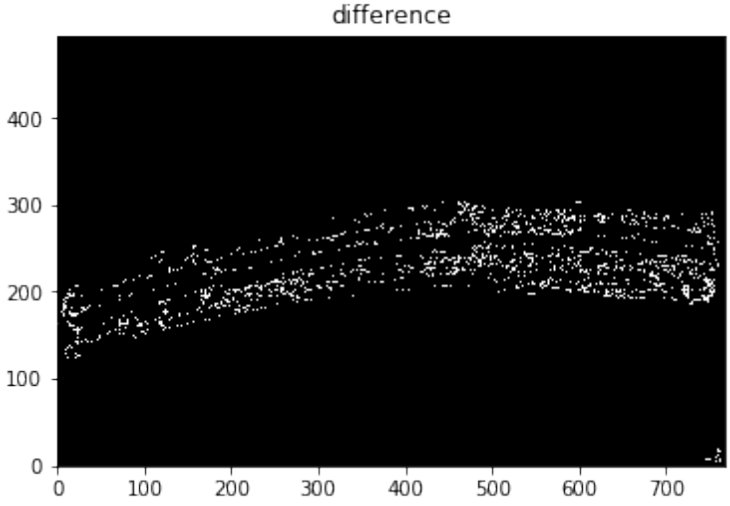

step 0
step 1
step 2
step 3
step 4
step 5
step 6
step 7
0
200
400
600

In [44]:
```
samples = np.array(samples)
samples_mean = np.round(np.mean(samples, axis=0))


unary_argmax =  np.argmax(pred, axis=2)
difference = np.zeros_like(unary_argmax)
difference[np.where(unary_argmax != samples_mean)] = 1
f = plt.figure()
f.set_size_inches(6,18)
ax = f.add_subplot(3,1,1)
ax.imshow(samples_mean, cmap='gray',origin='lower')
ax.set_title("mean posterior after Gibbs sampling")
ax = f.add_subplot(3,1,2)
ax.imshow(unary_argmax, cmap='gray',origin='lower')
ax.set_title("argmax unary")
ax = f.add_subplot(3,1,3)
ax.imshow(difference, cmap='gray',origin='lower')
ax.set_title("difference")
```

Out[44]:
```
<matplotlib.text.Text at 0x126fbd208>
```

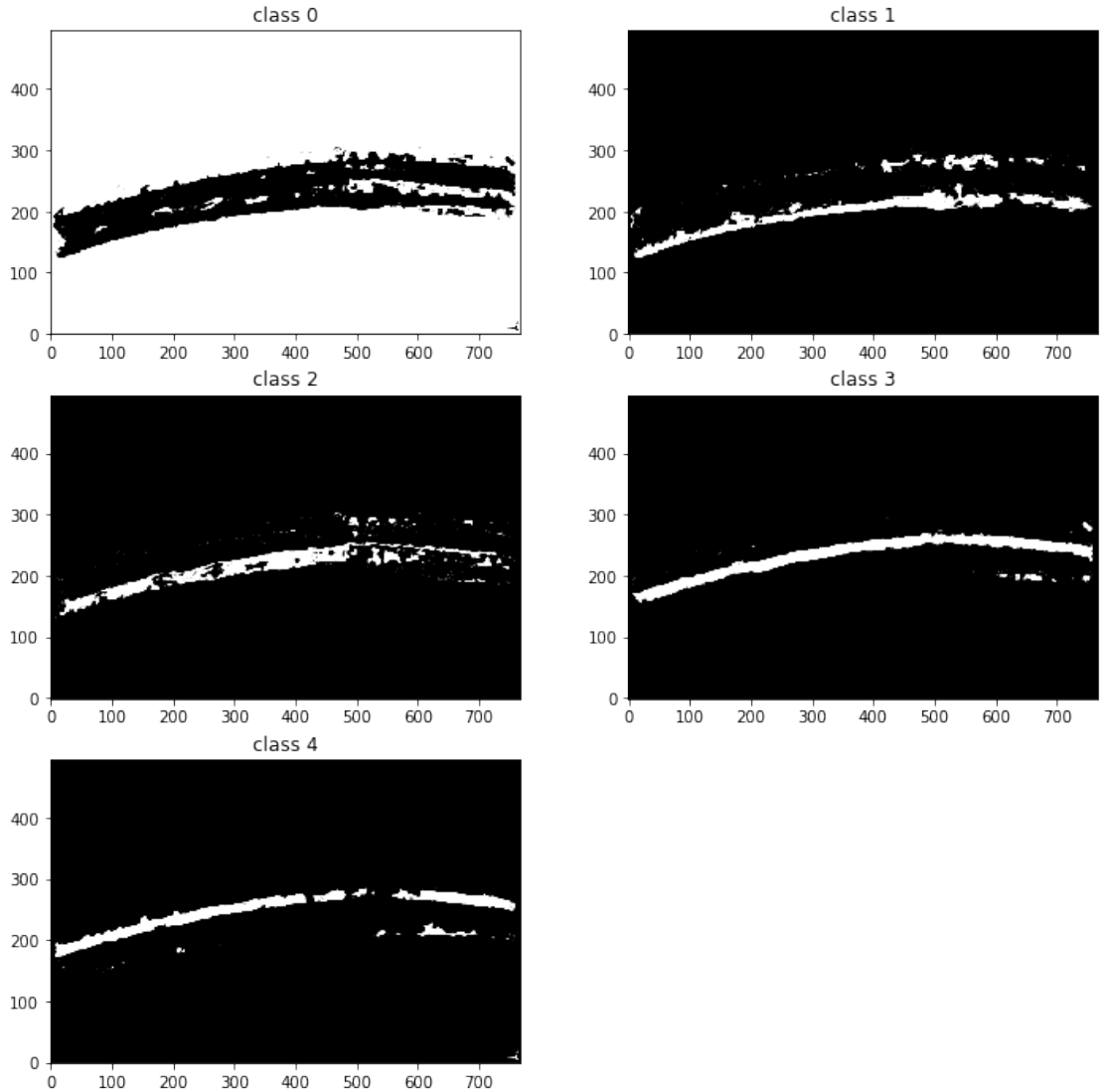

mean posterior after Gibbs sampling
0
100
200
300
400
0
100
200
300
400
500
600
700


argmax unary
0
100
200
300
400
0
100
200
300
400
500
600
700


difference
0
100
200
300
400
0
100
200
300
400
500
600
700

In [45]:
```
posterior_proba = np.zeros_like(unary, dtype=np.uint32)

for x in tnrange(posterior_proba.shape[0], desc='summing over rows'):
    for y in range(posterior_proba.shape[1]):
        for i in range(samples.shape[0]):
            posterior_proba[x,y, samples[i,x,y]] += 1
```

In [46]:
```
f = plt.figure()
f.set_size_inches((12, 12))
for i in range(5):
    ax = f.add_subplot(3,2,i+1)
    ax.imshow(posterior_proba[:,:,i], cmap='gray', vmax=posterior_proba.max(),origin='lower')
    ax.set_title("class "+str(i))
```



### 1.1.3 ⋆ Gibbs sampling on a chessboard

In [47]:
```
# convert the image to have at each pixel a one-hot vector indicating the current class
def convert_lattice(lattice,n_states):
    converted_lattice = np.zeros((lattice.shape[0],lattice.shape[1],n_states))
    for i in range(n_states):
        converted_lattice[np.where(lattice ==i)[0],np.where(lattice==i)[1],i] = 1
    extended_lattice = np.pad(converted_lattice, ((1,1),(1,1),(0,0)), "constant")
    return(extended_lattice)

def reconvert_lattice(c_lattice):
    lattice = c_lattice[1:-1,1:-1,:]
    labels = np.argmax(lattice,axis=2)
    return(labels)
```

In [48]:
```
# define masks to get the checkerboard and the neighbors corresponding to the white and black

def get_all_masks(lattice):
    all_neigh_white = []
    # white mask
    a = np.zeros((lattice.shape[0],lattice.shape[1]))
    a[1::2,1::2] = 1
    a[::2,::2] = 1
    white_mask = np.pad(a, ((1,1),(1,1)), "constant")
    all_neigh_white.append(np.pad(a, ((0,2),(1,1)), "constant"))
    all_neigh_white.append(np.pad(a, ((2,0),(1,1)), "constant"))
    all_neigh_white.append(np.pad(a, ((1,1),(0,2)), "constant"))
    all_neigh_white.append(np.pad(a, ((1,1),(2,0)), "constant"))

    all_neigh_black = []
    # black mask
    a.fill(0)
    a[::2,1::2] = 1
    a[1::2,::2] = 1
    black_mask = np.pad(a, ((1,1),(1,1)), "constant")
    all_neigh_black.append(np.pad(a, ((0,2),(1,1)), "constant"))
    all_neigh_black.append(np.pad(a, ((2,0),(1,1)), "constant"))
    all_neigh_black.append(np.pad(a, ((1,1),(0,2)), "constant"))
    all_neigh_black.append(np.pad(a, ((1,1),(2,0)), "constant"))

    return(white_mask, all_neigh_white, black_mask, all_neigh_black)
```

In [49]:
```
def compute_probs(c_lattice, unary, pairwise, mask, all_neigh_mask):
    # get the labels of the neighboring pixels
    neigh_labels_list = []
    for neigh_mask in all_neigh_mask:
        neigh = c_lattice[np.logical_not(1-neigh_mask),:]
        neigh_labels_list.append(neigh)
    neigh_labels = np.stack(neigh_labels_list,axis=0)

    # sum over all neighbors
    sum_neighbors = np.sum(neigh_labels,axis=0)

    # compute the product of the prior matrix  with the sum of the neighboring states for each
    energy_from_neighbors = np.matmul(pairwise, sum_neighbors.T).T
```

```
        probs_from_neighbors = np.exp(-energy_from_neighbors)

        full_probs = probs_from_neighbors * unary[np.logical_not(1-mask),:]
        norm_ = np.sum(full_probs,axis=1).reshape(full_probs.shape[0],1)
        full_probs /= norm_

        return(full_probs)
```

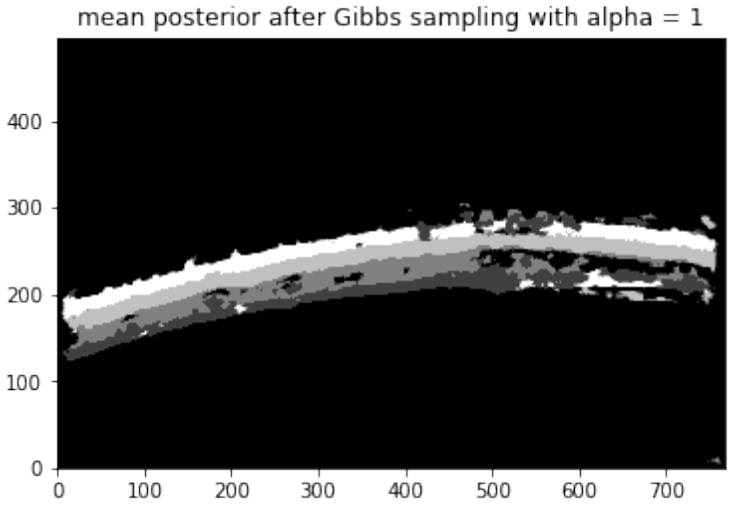
In [50]:
```
def checkerboard_sampler(c_lattice, unary, prior, black_mask, all_neigh_black, white_mask, all
    # sample the black fields conditioned on the white fields

    # compute probs for the updates of the black fields conditioned on their neighbors
    p = compute_probs(c_lattice, unary, prior, black_mask, all_neigh_black)

    # sample from p to update the black fields
    # use reparametrization of the discrete distribution given by p we want to sample from
    U = np.random.random(p.shape)
    G = np.log(p) - np.log(-np.log(U))
    G[np.where(np.logical_not(np.isfinite(G)))] = -np.inf

    new_states = np.argmax(G,axis=1)
    c_new_states = np.zeros((new_states.shape[0],n_states))
    for i in range(n_states):
        c_new_states[np.where(new_states ==i)[0],i] = 1

    # write the new labels to the c_lattice
    c_lattice[np.logical_not(1-black_mask)] = c_new_states

    # sample the white fields conditioned on the black fields

    # compute probs for the updates of the black fields conditioned on their neighbors
    p = compute_probs(c_lattice, unary, prior, white_mask, all_neigh_white)

    # sample from p to update the black fields
    # use reparametrization of the discrete distribution given by p we want to sample from
    U = np.random.random(p.shape)
    G = np.log(p) - np.log(-np.log(U))
    G[np.where(np.logical_not(np.isfinite(G)))] = -np.inf

    new_states = np.argmax(G,axis=1)
    c_new_states = np.zeros((new_states.shape[0],n_states))
    for i in range(n_states):
        c_new_states[np.where(new_states ==i)[0],i] = 1

    # write the new labels to the c_lattice
    c_lattice[np.logical_not(1-white_mask)] = c_new_states

    return(c_lattice)
```

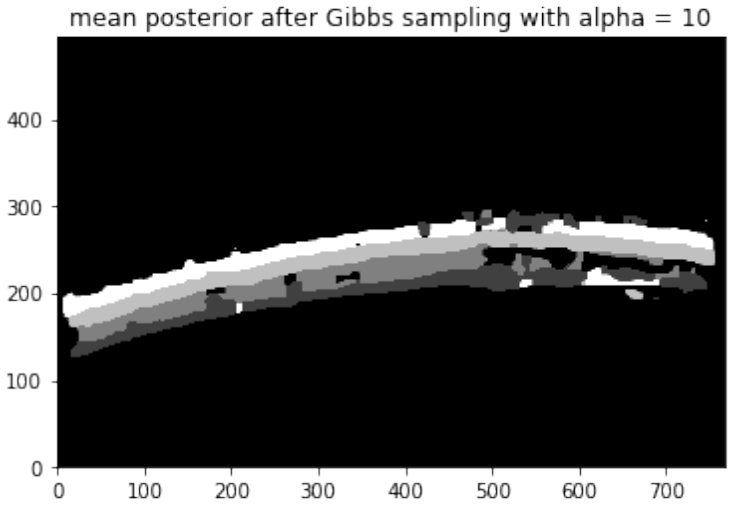
In [51]:
```
n_steps = 100
n_scip = 10
n_states = 5
alpha = [1,10,100]
```

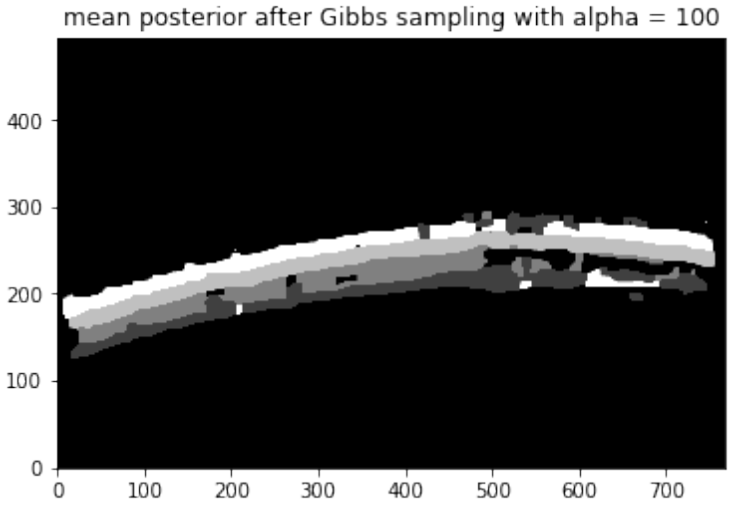
```
        lattice = np.random.randint(0,n_states,np.argmax(pred, axis=2).shape)

        # get masks for black and white fields and their neighbors
        white_mask, all_neigh_white, black_mask, all_neigh_black = get_all_masks(lattice)

        # extend the lattice by padding with a zero on each side and convert class label into one-hot
        c_lattice = convert_lattice(lattice, n_states)

        # extend the unary term by a row of zeros on each side
        unary = np.pad(pred, ((1,1),(1,1),(0,0)), "constant")



        samples = {}
        for al in alpha:
            samples[al] = []
            prior = prior_l0(n_states)*al



            for i in tnrange(n_steps, desc='outer', position=0):
                for j in range(n_scip):
                    sample = reconvert_lattice(checkerboard_sampler(c_lattice, unary, prior, black_mask
                samples[al].append(sample.copy())
```

```
//anaconda/envs/python3/lib/python3.5/site-packages/ipykernel/__main__.py:10: RuntimeWarning: divide by z
//anaconda/envs/python3/lib/python3.5/site-packages/ipykernel/__main__.py:29: RuntimeWarning: divide by z
```

In [52]:
```
        f = plt.figure()
        f.set_size_inches(6,18)

        i = 0
        for al in alpha:
            samples_ = np.array(samples[al])
            samples_mean = np.round(np.mean(samples_, axis=0))

            i += 1
            ax = f.add_subplot(3,1,i)
            ax.imshow(samples_mean, cmap='gray',origin='lower')
            ax.set_title("mean posterior after Gibbs sampling with alpha = %i"%al)
```


mean posterior after Gibbs sampling with alpha = 1
0
100
200
300
400
0
100
200
300
400
500
600
700

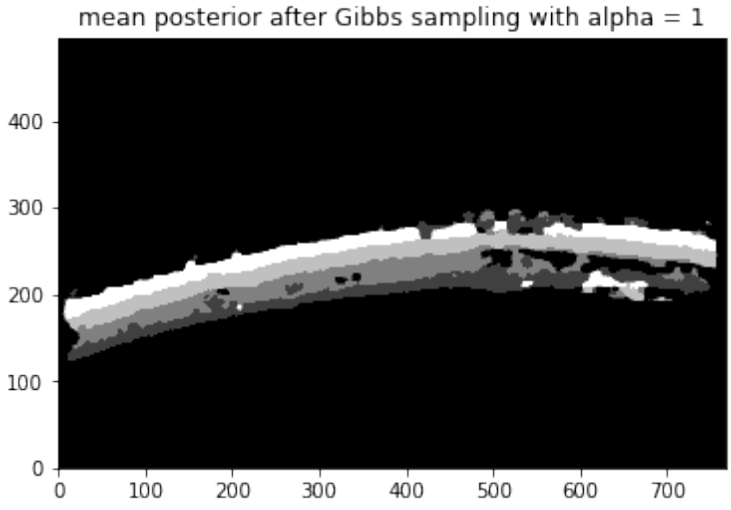

mean posterior after Gibbs sampling with alpha = 10
0
100
200
300
400
0
100
200
300
400
500
600
700

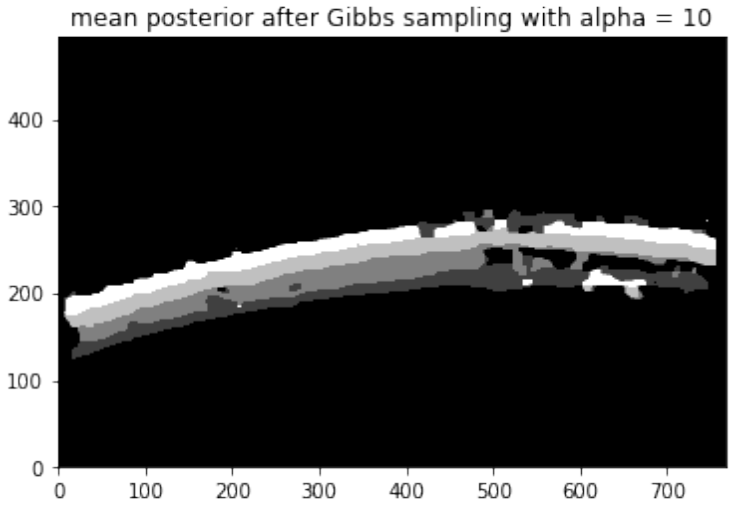

mean posterior after Gibbs sampling with alpha = 100
0
100
200
300
400
0
100
200
300
400
500
600
700

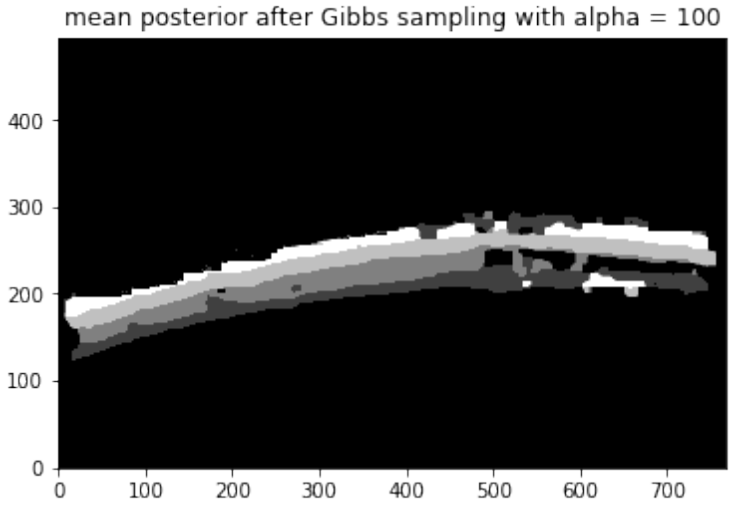
In [53]:
```
lattice = np.random.randint(0,n_states,np.argmax(pred, axis=2).shape)


        # extend the lattice by padding with a zero on each side and convert class label into one-hot
        c_lattice = convert_lattice(lattice, n_states)



        samples = {}
        for al in alpha:
            samples[al] = []
            prior = prior_l1(n_states)*al



            for i in tnrange(n_steps, desc='outer', position=0):
                for j in range(n_scip):
                    sample = reconvert_lattice(checkerboard_sampler(c_lattice, unary, prior, black_mask
                samples[al].append(sample.copy())
```

```
//anaconda/envs/python3/lib/python3.5/site-packages/ipykernel/__main__.py:10: RuntimeWarning: divide by z
//anaconda/envs/python3/lib/python3.5/site-packages/ipykernel/__main__.py:29: RuntimeWarning: divide by z



//anaconda/envs/python3/lib/python3.5/site-packages/ipykernel/__main__.py:19: RuntimeWarning: invalid val
```

In [54]:
```
f = plt.figure()
f.set_size_inches(6,18)

i = 0
for al in alpha:
    samples_ = np.array(samples[al])
    samples_mean = np.round(np.mean(samples_, axis=0))

    i += 1
    ax = f.add_subplot(3,1,i)
    ax.imshow(samples_mean, cmap='gray',origin='lower')
    ax.set_title("mean posterior after Gibbs sampling with alpha = %i"%al)
```


mean posterior after Gibbs sampling with alpha = 1
0
100
200
300
400
0
100
200
300
400
500
600
700


mean posterior after Gibbs sampling with alpha = 10
0
100
200
300
400
0
100
200
300
400
500
600
700


mean posterior after Gibbs sampling with alpha = 100
0
100
200
300
400
0
100
200
300
400
500
600
700

In [ ]: