(a) Initialization

```python
# initialize nodes
nodes = {}
for i in range(size):
    for j in range(size):
        if initial_value != 1 and initial_value != -1:
            nodes[(i, j)] = 2 * np.random.randint(2) - 1
        else:
            nodes[(i, j)] = initial_value

# set up neighbor information
node_neighbors = {}
for i in range(size):
    for j in range(size):
        neighbors = []
        if i >= 1:
            neighbors.append((i - 1, j))
        if j >= 1:
            neighbors.append((i, j - 1))
        if i < size - 1:
            neighbors.append((i + 1, j))
        if j < size - 1:
            neighbors.append((i, j + 1))
        node_neighbors[(i, j)] = neighbors

# set up edge potential
edge_potential = {}
edge_potential[(1, 1)] = np.exp(theta)
edge_potential[(1, -1)] = np.exp(- theta)
edge_potential[(-1, 1)] = np.exp(- theta)
edge_potential[(-1, -1)] = np.exp(theta)
```

(b) Gibbs node sampler

```python
def sample(nodes, node_neighbors, edge_potential, traversal_order):
    '''
    nodes: list of (i, j) nodes
    nodes: dictionary node (i, j) -> neighbors of node (i, j)
    edge_potentials: dictionary ((i, j), (k, l)) -> potential map
    traversal_order: list of [(i, j)] for message passing
    '''
    node_potentials = {}
    tree_edge_potentials = {}

    last_node = set()
    for (i, j) in traversal_order:
        node_potentials[(i, j)] = {-1: 1, 1: 1}
        for n in node_neighbors[(i, j)]:
            # set up node potential based on observation on the other comb
            # those nodes shouldn't be on the traversal path
            if n not in traversal_order:
                node_potentials[(i, j)][-1] *= edge_potential[-1, nodes[n]]
                node_potentials[(i, j)][1] *= edge_potential[1, nodes[n]]
        # set up edge potential along the tree path
        assert len(last_node) <= 2
        for n in list(last_node):
            if n in node_neighbors[(i, j)]:
                tree_edge_potentials[(n, (i, j))] = edge_potential
                last_node.remove(n)
        last_node.add((i, j))

    # get message from one pass of belief propagation
    messages = belief_propagation(
        node_potentials, tree_edge_potentials, traversal_order)

    # reverse the traversal order, perform sampling
    samples = OrderedDict()  # need O(1) query and insertion order
    for (i, j) in reversed(traversal_order):
        ep_positive = 1
        ep_negative = 1
        for n in node_neighbors[(i, j)]:
            if n in traversal_order:
                if n in samples:
                    # already sampled
                    ep_positive *= edge_potential[1, samples[n]]
                    ep_negative *= edge_potential[-1, samples[n]]
                else:
                    # need to use message
                    ep_positive *= messages[(n, (i, j))][1]
                    ep_negative *= messages[(n, (i, j))][-1]

        # sample
        p = np.random.rand()

        if p < ep_positive / (ep_positive + ep_negative):
            samples[(i, j)] = 1
        else:
            samples[(i, j)] = -1
```

(c) Gibbs node sampling loop

```python
# gibbs block sampling
mean_vals = []
mean_vals.append(np.mean(list(nodes.values())))
for iteration in range(iterations):
    # sweep through bloack A and B
    for traversal_order in [traversal_order_a, traversal_order_b]:
        samples = sample(nodes, node_neighbors, edge_potential, traversal_order)
        for n in samples:
            nodes[n] = samples[n]

    # record the mean values
    mean_vals.append(np.mean(list(nodes.values())))

    if iteration % output_interval == 0:
        # visualization
        visualize(size, nodes, results_folder +
            file_prefix + '_iter_' + str(iteration) + '.png')
        print('iteration ', iteration)

# plot mixing behavoir
plot_mixing(mean_vals, results_folder + file_prefix + '_mixing')
```

(d) Message update

```python
def get_msg(i, j, node_potential, edge_potential, messages, neighbors, normalize=True):
    # get msg_{i->j}(var)
    distant_msg = {-1: 1, 1: 1}
    for k in neighbors[i]:
        if k != j:
            distant_msg[-1] *= messages[(k, i)][-1]
            distant_msg[1] *= messages[(k, i)][1]

    msg = {}
    msg[-1] = node_potential[i][-1] * ep(edge_potential, i, j, -1, -1) * distant_msg[-1] \
        + node_potential[i][1] * ep(edge_potential, i, j, 1, -1) * distant_msg[1]
    msg[1] = node_potential[i][-1] * ep(edge_potential, i, j, -1, 1) * distant_msg[-1] \
        + node_potential[i][1] * ep(edge_potential, i, j, 1, 1) * distant_msg[1]

    if normalize:
        s = msg[-1] + msg[1]
        msg[-1] /= s
        msg[1] /= s

    return msg
```

(e) Serial belief propagation

```python
def belief_propagation(node_potential, edge_potential, traversal_order):
    '''
    node_potential: {i -> node_potential}
    edge_potential: {(i, j) -> edge_potential}
    output: {i -> marginal}
    '''

    # find neighbor nodes for each node
    neighbors = {}
    for i in node_potential:
        neighbors[i] = set()
    for (i, j) in edge_potential:
        neighbors[i].add(j)
        neighbors[j].add(i)

    # initialize random message
    messages = {}
    init_msg = {-1: 1, 1: 1}
    for (i, j) in edge_potential:
        messages[(i, j)] = init_msg
        messages[(j, i)] = init_msg

    # traverse following the order
    last_node = None
    for n in traversal_order:
        if last_node is not None and \
          (last_node, n) in edge_potential:
            msg = get_msg(
                last_node, n,
                node_potential,
                edge_potential,
                messages,
                neighbors)
            messages[(last_node, n)] = msg
        # advance a node
        last_node = n

    return messages
```

(f) Traversal order

```python
# find traversal order for block A
traversal_order_a = OrderedSet()
for j in range(size):
    if j % 2 == 0:
        for i in range(size - 2, -1, -1):
            traversal_order_a.add((i, j))
    else:
        traversal_order_a.add((0, j))
```

```python
# find traversal order for block B
traversal_order_b = OrderedSet()
for j in range(size):
    if j % 2 == 0:
        traversal_order_b.add((size - 1, j))
    else:
        for i in range(1, size, 1):
            traversal_order_b.add((i, j))
```

(g) Gibbs block sample

```python
def sample(nodes, node_neighbors, edge_potential, traversal_order):
    '''
    nodes: list of (i, j) nodes
    nodes: dictionary node (i, j) -> neighbors of node (i, j)
    edge_potentials: dictionary ((i, j), (k, l)) -> potential map
    traversal_order: list of [(i, j)] for message passing
    '''
    node_potentials = {}
    tree_edge_potentials = {}

    last_node = set()
    for (i, j) in traversal_order:
        node_potentials[(i, j)] = {-1: 1, 1: 1}
        for n in node_neighbors[(i, j)]:
            # set up node potential based on observation on the other comb
            # those nodes shouldn't be on the traversal path
            if n not in traversal_order:
                node_potentials[(i, j)][-1] *= edge_potential[-1, nodes[n]]
                node_potentials[(i, j)][1] *= edge_potential[1, nodes[n]]
        # set up edge potential along the tree path
        assert len(last_node) <= 2
        for n in list(last_node):
            if n in node_neighbors[(i, j)]:
                tree_edge_potentials[(n, (i, j))] = edge_potential
                last_node.remove(n)
        last_node.add((i, j))

    # get message from one pass of belief propagation
    messages = belief_propagation(
        node_potentials, tree_edge_potentials, traversal_order)

    # reverse the traversal order, perform sampling
    samples = OrderedDict()  # need O(1) query and insertion order
    for (i, j) in reversed(traversal_order):
        ep_positive = node_potentials[(i, j)][1]
        ep_negative = node_potentials[(i, j)][-1]
        for n in node_neighbors[(i, j)]:
            if n in traversal_order:
                if n in samples:
                    # already sampled
                    ep_positive *= edge_potential[1, samples[n]]
                    ep_negative *= edge_potential[-1, samples[n]]
                else:
                    # need to use message
                    ep_positive *= messages[(n, (i, j))][1]
                    ep_negative *= messages[(n, (i, j))][-1]

        # sample
        p = np.random.rand()

        if p < ep_positive / (ep_positive + ep_negative):
            samples[(i, j)] = 1
        else:
            samples[(i, j)] = -1
```

(h) Gibbs block sampling loop

```python
# gibbs block sampling
mean_vals = []
mean_vals.append(np.mean(list(nodes.values())))
for iteration in range(iterations):
    # sweep through bloack A and B
    for traversal_order in [traversal_order_a, traversal_order_b]:
        samples = sample(nodes, node_neighbors, edge_potential, traversal_order)
        for n in samples:
            nodes[n] = samples[n]

    # record the mean values
    mean_vals.append(np.mean(list(nodes.values())))

    if iteration % output_interval == 0:
        # visualization
        visualize(size, nodes, results_folder +
            file_prefix + '_iter_' + str(iteration) + '.png')
        print('iteration ', iteration)

# plot mixing behavoir
plot_mixing(mean_vals, results_folder + file_prefix + '_mixing')
```