Assignment 2

Assignment 2: Sampling

Goal: Get familiar with sampling methods by implementing these methods and applying them to a given probability distribution.

In this assignment we are going to learn about sampling methods. The goal is to implement *Metropolis-Hastings **(MH) algorithm and** Simulated Annealing** (SA) algorithm and analyze their behavior. Importantly, we aim at noticing differences between these two methods.

Here, we are interested in **sampling** from a mixture of two Gaussians, namely:

$$p(\mathbf{x}) = 0.25 \cdot \mathcal{N}\left(\mu = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, \Sigma = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}\right) + 0.75 \cdot \mathcal{N}\left(\mu = \begin{bmatrix} -3 \\ -3 \end{bmatrix}, \Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right)$$

In this assignemnt, you are asked to implement:

1. The Metropolis-Hastings (MH) algorithm.
2. The simulated annealing (SA) algorithm.

After implementing both methods, please run experiments and compare both methods. Please follow all instructions.

1. Understanding the probability distribution

Please run the code below and visualize the probability distribution. Please try to understand this distribution, what the modes are (you can do it by inspecting the plot). What are possible problems here?

If any code line is unclear to you, please read on that in numpy or matplotlib docs.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal

EPS = 1.e-7
```

In [2]:

```
# PLEASE DO NOT REMOVE!
# The definition of the probability distribution.
mv1 = multivariate_normal(mean=[2.,2.], cov=[[1., 0.5],[0.5, 1.]])
mv2 = multivariate_normal(mean=[-3.,-3.], cov=[[1., 0.],[0., 1.]])

def prob(x): # this is the probability distribution
  return 0.25 * mv1.pdf(x) + 0.75 * mv2.pdf(x)
```

In [3]:

```
# PLEASE DO NOT REMOVE!
# Calculating the probability distribution.
def calculate_p(x1, x2):
  p_x = []
  for i in range(len(x1)):
    for j in range(len(x2)):
      p_x.append(prob(np.asarray([[x1[i], x2[j]]])))

  return np.asarray(p_x).reshape(len(x1), len(x2))
```

In [4]:

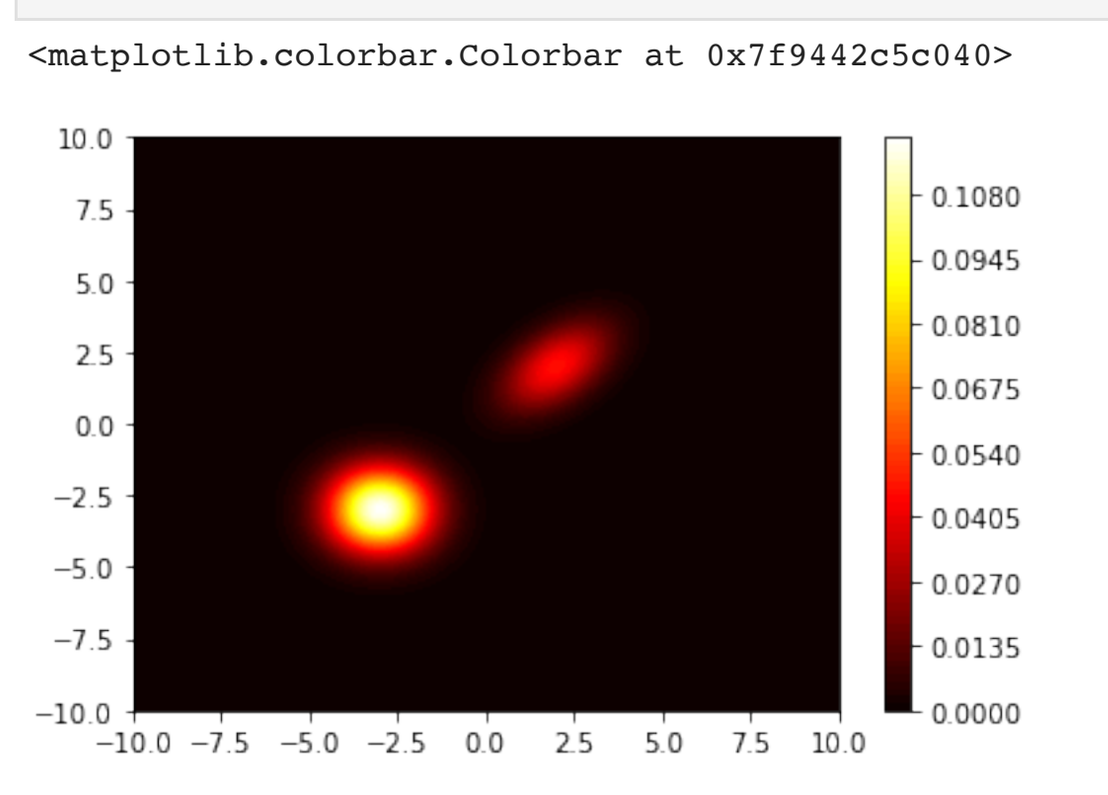
```
# PLEASE DO NOT REMOVE!
# Define coordinates
x1 = np.linspace(-10., 10., 400)
x2 = np.linspace(-10., 10., 400)

# Calculate the objective
p_x = calculate_p(x1, x2).reshape(len(x1), len(x2))
```

In [5]:

```
# PLEASE DO NOT REMOVE!
# Plot the objective
plt.contourf(x1, x2, p_x, 100, cmap='hot')
plt.colorbar()
```

Out[5]:

```
<matplotlib.colorbar.Colorbar at 0x7f9442c5c040>
```



Question 1 (0-0.5pt): How many modes do we have and what are they? What is the most probable point?

Answer: Mode is the most frequent value in a probability distribution. In these two distributions, there are TWO modes, we one mode at (-3,-3) and another at (2,2). Overall the most probable point is at (-3,-3) where the probability is the maximum in the whole distribution.

2. The Metropolis-Hastings algorithm

First, you are asked to implement the Metropolis-Hastings (MH) algorithm. Please take a look at the class below and fill in the missing parts.

NOTE: Please pay attention to the inputs and outputs of each function.

NOTE: To implement the MH algorithm, you need to specify the proposal distribution.

Question 2 (0-0.5pt): What is the proposal distribution, $q(\mathbf{x}_{new}|\mathbf{x}_{old})$, in your case?

Answer: using gaussian distribution where the data is normally distributed, illustrating the mean to be $\mathbf{x}_{old}$ and in this case, $q(\mathbf{x}_{new}|\mathbf{x}_{old})$ = 1/σ√2π ∗ e∗∗−(($\mathbf{x}_{new}$− $\mathbf{x}_{old}$)∗∗2)/2σ∗∗2 Since Normal distribution is symmetric the value of the $q(\mathbf{x}_{new}|\mathbf{x}_{old})$ is same as $q(\mathbf{x}_{old}|\mathbf{x}_{new})$

Question 3 (0-0.5pt): Is your proposal a proper proposal distribution? (That is, it must fulfill irreducibility and aperiodicity, see Andrieu et al., "An Introduction to MCMC for Machine Learning".)

Answer: The Gaussian distribution is positive in the range of −∞ to +∞, allowing the Probability to reach any point on a real line from any other point. The rea line is always positive, even if the values become infinitly small, thus making this proposl distribution irreducible.

In [6]:

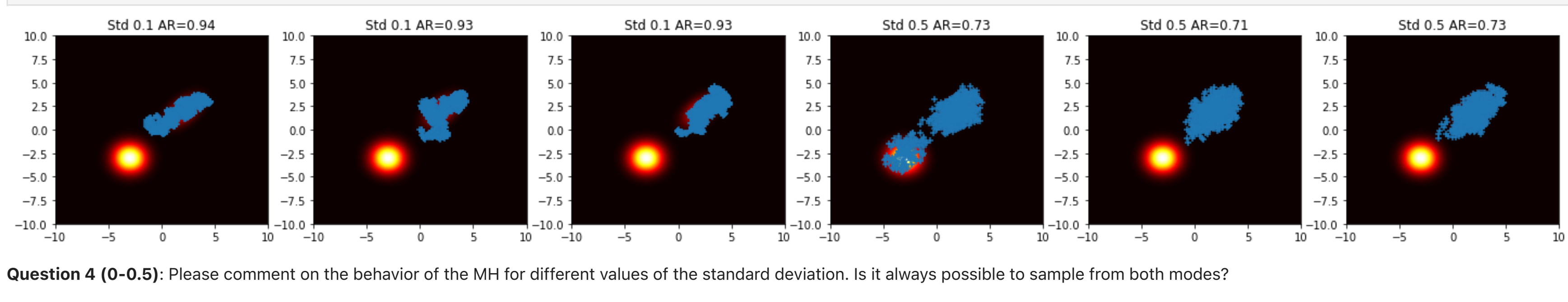
```
#=========
# GRADING:
# 0
# 0.5pt if properly implemented but badly commented
# 1.0 pt if properly implemented and clear
#=========
# Implement the Metropolis Hastings (MH) algorithm.
# It might be you use another arguments beside standard deviation (std).
# Please add them then.

class MetropolisHastings(object):
  def __init__(self, x, prob, std=0.1):
    self.prob = prob
    self.std = std
    self.x_old = x

  def proposal(self, x):
    # PLEASE FILL IN
    # we are taking the randome distribution of x as x_new and passing it as our mean and self.std to be SD.
    x_new = np.random.normal(x, self.std) # our proposal distribution

    return x_new #.clip([-10],[10]) # limiting the values in the array from -10 to 10

  def evaluate(self, x_new, x_old):
    # PLEASE FILL IN
    A = ((prob(x_new))/(prob(x_old))) # the acceptance probability where x_new and x_old pass as arguments

    return np.minimum(1., A)

  def select(self, x_new, A):
    # PLEASE FILL IN
    # ...
    # NOTE: If a new candidate is not selected, we must return self.x_old

    # we select if A is larger than the uniform distribution, then accept x_old = x_new
    if np.random.uniform(0,1)<A:
        self.x_old = x_new
        # return x_new
    # otherwise return x_old
    return self.x_old

  def step(self):
    # 1. Proposal
    x_prop = self.proposal(self.x_old)
    # 2. Evaluate
    A = self.evaluate(x_prop, self.x_old)
    # 3. Select
    x_new = self.select(x_prop, A)

    return x_new
```

In [7]:

```
# PLEASE DO NOT REMOVE!
# An auxiliary function for plotting.
def plot_sampling_process(ax, sampler, title):
  # Plot the objective function
  ax.contourf(x1, x2, p_x/p_x.sum(), 100, cmap='hot')

  # Init the solution
  x_samp = sampler.x_old
  # Run the optimization algorithm
  count = 0 # we calculate how many of new candidates were accepted
  for i in range(num_epochs):
    x = sampler.step()

    if (x == x_samp[-1]).all():
      pass
    else:
      count = count + 1
      x_samp = np.concatenate((x_samp, x),0)

  ax.scatter(x_samp[:,0], x_samp[:,1], marker='+')
  # We add AR (Acceptance Rate) to the title.
  ax.set_title(title + ' AR={:.2f}'.format(count/num_epochs))
```

In [8]:

```
# PLEASE DO NOT REMOVE!
# Running the MH algorithm with different step sizes
num_epochs = 1500 # the number of epochs
stds = [0.1, 0.1, 0.1, 0.5, 0.5, 0.5] # we repeat experiments 3 times per std
## If you add any other hyperparameters, please include them during init!
## However, please do not include them in the for-loop. It should stay for stds only.

# plotting the convergence of the MH
fig_mh, axs = plt.subplots(1,len(stds),figsize=(20, 3))
fig_mh.tight_layout()

x_init = np.asarray([[0., 0.]])

for i in range(len(stds)):
  # take the std
  std = stds[i]
  # init the MH
  mh = MetropolisHastings(x=x_init, prob=p_x, std=std)
  # plot the convergence
  plot_sampling_process(axs[i], sampler=mh, title='Std ' + str(mh.std))
```



Question 4 (0-0.5): Please comment on the behavior of the MH for different values of the standard deviation. Is it always possible to sample from both modes?

Answer:

MH at standard deviation of 0.1 samples points too close to the mean, we can see this at the second peak, where the objective function seems to be outside the vision or sample space of the Normal distribution. So, it mostly moves in the direction of one of the peaks.

For SD = 0.5, we have a greater chance to sample the points from areas around both the peaks of the objective function. therefore, the samples are more likely from around both the peaks. In theory, given infinite iterations, there will be sampling from both modes. But as the experiments show, especially for smaller standard deviations, there will be sampling from just one mode.

Question 5 (0-0.5): Please comment on the acceptance ratio (AR) for std=0.1 and std=0.5. How can you explain the difference?

Answer: The suggested samples from proposal distribution with std=0.1 are closer to the mean, so the value of f(xnew) is similar to f(xold). Therefore, the acceptance probability f(xnew)/f(xold) is closer to 1, making most of the samples suggested have a high probability of getting accepted.

However, the suggested samples from proposal distribution with std=0.5, illustrates samples which are further from the mean point. Depcting that ther are high chances that the acceptance probability is not always closer to 1.Thus, more samples from this distribution are rejected.

Consequently, proposal distribution with std=0.1 has an acceptance rate of around 0.93 in this case, while distribution with std=0.5 has around 0.73 acceptance rate.

3. The simulated annealing (SA) sampling

In the second part of this assignment, you are asked to implement the Simulated Annealing (SA) algorithm with cooling scheme as dicussed during the lecture.

Question 6 (0-0.5pt): Please explain what is the difference between MH and SA?

Answer: Both algorithms are stochastic, generating new points to move to at random. Where they differ is in their acceptance/rejection criterion. Both algorithms move to a new random point with a certain probability, which is based on the difference (or the ratio) of the current and new proposed point in the search space.

Simulated annealing is a meta-heuristic algorithm used for optimization, that is finding the minimum/maximum of a function with the application of temperature. It moves to a new point depending if the temperature is high or low. If it is high then it will move at random to a new point, but if the temperature is low the algorithm will be more deterministic and select the next point to move to.

Metropolis-Hastings is an algorithm used for exploring a function (finding possible values/samples). the algorithm calculates the ration between x_new and x_old. If the ratio is > 1 then the algorithms determines to move to the respective point. If the ratio is less than 1, then it will determine a point between 0-1 and evaluate if the new point is greater than or less than the ratio. If it is greater the new point will be accepted, otherwise it will be rejected.

Question 7 (0-0.5pt): Why is SA sometimes more preferrable than MH? Which of these two methods would you use for optimization (not sampling)?

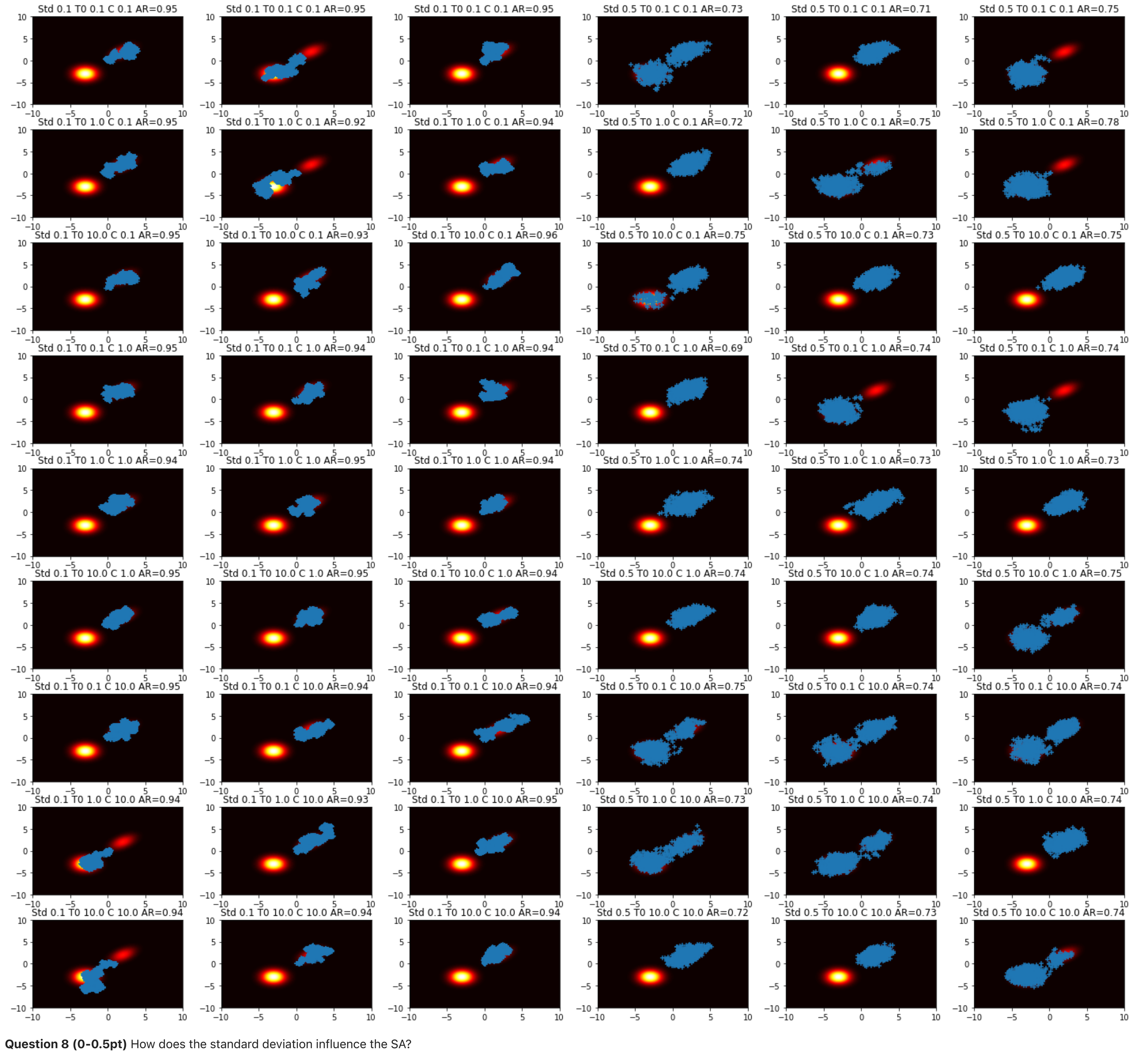
Answer: Simulted annealing is more preferably because the algorithm is a meta-herusitc algorithm which is used for optimization. It is similar to metropolis hastings but with the application of temperature. SA is better, because when it determines which point to go to, the algorithm becomes more determinisitc, choosing the next best point, in contrast to MH it chooses a random point and determines if it is the best point to go to.

In [9]:

```
#=========
# GRADING:
# 0
# 0.5pt if properly implemented but badly commented
# 1.0 pt if properly implemented and clear
# NOTE: The cooling scheme must be properly implemented!
#=========
# Implement the Simulated Annealing (SA) algorithm.
# It is equivalent to implementing the step function.
class SimulatedAnnealing(object):
  def __init__(self, x, prob, std=0.1, T0=1., C=1.):
    self.prob = prob
    self.std = std
    self.x_old = x
    self.T0 = T0
    self.C = C
    self.t = 0 # we init the step number to 0; we need it for cooling scheme

  def proposal(self, x):
    # PLEASE FILL IN
    #...
    x_new = np.random.normal(x, self.std) # our proposal distribut

    return x_new # .clip([-10],[10]) #limiting the values in the array from -10 to 10

  def evaluate(sef, x_new, x_old, T):
    # PLEASE FILL IN
    # ...
    A = (prob(x_new)**1/T)/(prob(x_old)**1/T) # the acceptance probability where x_new and x_old pass as arguments

    return np.minimum(1., A)

  def select(self, x_new, A):
    # PLEASE FILL IN
        # we select if A is larger than the uniform distribution, then accept x_old = x_new
    if np.random.uniform(0,1)<A:
        self.x_old = x_new
        # return x_new
    # otherwise return x_old
    return self.x_old

  def step(self):
    # The temperature scheme (the cooling scheme)
    # PLEASE FILL IN
    T = 1/(self.C * np.log(self.t + self.T0 + 1))
    self.t = self.t + 1
    # 1. Proposal
    x_prop = self.proposal(self.x_old)
    # 2. Evaluate
    A = self.evaluate(x_prop, self.x_old, T)
    # 3. Select
    x_new = self.select(x_prop, A)
    return x_new
```

In [10]:

```
# PLEASE DO NOT REMOVE!
# Running the SA algorithm with different step sizes
num_epochs = 1500 # the number of epochs
stds_sa = [0.1, 0.1, 0.1, 0.5, 0.5, 0.5] # values of std; notice we repeat them due to randomness
T0s = [0.1, 1., 10.] # values of T0
Cs = [.1, 1., 10.] # values of C

# plotting the convergence of the MH
fig_sa, axs = plt.subplots(len(Cs) * len(T0s),len(stds_sa),figsize=(20, 18))
fig_sa.tight_layout()

# Always start with the same point
x_init = np.asarray([[0., 0.]])

# START SA
for i in range(len(Cs)):
  # Take a value for C
  C = Cs[i]
  for k in range(len(T0s)):
    # Take a value for the init temperature T0
    T0 = T0s[k]
    for j in range(len(stds_sa)):
      # take the std
      std = stds_sa[j]
      # init the SA
      sa = SimulatedAnnealing(x=x_init, prob=p_x, std=std, T0=T0, C=C)
      # plot the sampling process
      plot_sampling_process(axs[len(T0s)*i+k,j], sampler=sa, title='Std ' + str(sa.std) + ' T0 ' + str(T0) + ' C ' + str(C))
```



Question 8 (0-0.5pt) How does the standard deviation influence the SA?

Answer:

By correlating a rate of cooling in terms of a distance from equilibrium state—the higher the standard deviation (the system is far away from the equilibrium), the higher the rate of cooling. We can see that as the standard deviation increases to 0.5 the points cool and sccumulate around the mean point, illustrtaing that as the temperature keeps cooling down, the results slowly become worse as SA has to take decivsive steps instead of randomly selecting point.

Standard Deviation at 0.1 limited the search space for proposing next points. On one hand it is slow to move, but on the other it allowed the SA to sample more points from the given probability distribution. It has a consistently higher acceptance rate as compared to the SA with SD as 0.5, and it sampled the objective function more accurately.

Question 9 (0-0.5pt) How does the initial temperature, T_0, influence the SA?

Answer: PLEASE FILL IN

Question 10 (0-0.5pt) How does the constant C influence the SA?

Answer: PLEASE FILL IN

Question 11 (0-0.5pt) Which setting (i.e., std, T_0, C) did perform the best in terms of sampling?

Answer:

Looking at the consistency of sampling, we can see the best sampling to be one one of the best sampling was obtained by the following parameters: std=0.1, T0=0.1, C=10

Question 12 (0-0.5pt) How do different values of the hyperparameters (i.e., std, T_0, C) influence the acceptance ratio (AR)? Why?

Answer: PLEASE FILL IN

4. Final remarks: MH vs. SA

Eventually, please answer the following last questions that will allow you to conclude the assignment and draw conclusions.

Question 13 (0-0.5-1pt): Which of the two algorithms did perform better? Why?

Answer: Both algorithms performed well, but MH performed better. We can see in MH that it often circled around the mean point and sampled close to the target pointsm but with the right parameters SA was more likely to find the global maxima.

MH focussed around the objective function, thus the proposals were nearby relevant values. Thus, the acceptance rate was high and more accurate sampling was present. MH didn't explore much and at lower SD=0.1, it didn't move from one of the modes.

SA "explored" much more points. It was more successful in reaching global maxima but needed the right hyperparameters for that. It was bad at sampling initially but with lowered temperatures, the sampling improved.

Question 14 (0-0.5-1pt): Which of the two algorithms is easier to use? Why?

Answer: MH with Gaussian distribution was easy to apply, because in contrast to SA we needed hyperparametr tuning, setting up and calcualting T. Although SA is better for real-life optimization problems, it is trickier to set-up.
MH is easy for accurate sampling.