Advanced Machine Learning - N-joint Arm

Angeliki-Ilektra Karaiskou (s1029746)

July 3, 2020

Introduction

Optimal control theory involves the optimization of the steps that need to be followed in order to reach a goal. One of the commonest control theory problems is to reach a specific target with an arm of N-joints [1]. The location of each i joint in the xy 2D plane is

$$x_i = \sum_{j=1}^{i} \cos\theta_j \qquad\qquad y_i = \sum_{j=1}^{i} \sin\theta_j \tag{1}$$

with $i = 1,\ldots,N$ the number of joints and θ_i the joint angles. Each joint is controlled by the controller u_i and its dynamic is

$$d\theta_i = u_i dt + d\xi_i \tag{2}$$

with $d\xi_i$ independent Gaussian noise with $\langle d\xi_i^2 \rangle = \nu dt$. From now on $\vec{\theta}$ is the vector of the joint angles and $\vec{u}$ is the controller vector.

The N-joint arm is a path integral problem and involves the minimization of the cost function $C(\vec{\theta}, t, \vec{u}_{t:T})$ Eq. (4) by finding the optimal control path $\vec{u}_{t:T}$. The expected cost is computed using the following expressions

$$C(\vec{\theta}, t, \vec{u}_{t:T}) = \langle \phi(\theta(T)) + \int_t^T \vec{u}^T(t)\vec{u}(t) \rangle \tag{3}$$

$$\phi(\theta(T)) = \frac{\alpha}{2}((x_n - x_{target})^2 + (y_n - y_{target})^2) \tag{4}$$

where x_{target}, y_{target} are the coordinates of the target to reach the end joind and T the time to reach the target.

The control for each joint at time t is computed by Eq.(5)

$$u_i = \frac{1}{T-t}(\langle \theta_i \rangle - \theta^{old}) \tag{5}$$

with $\langle \theta_i \rangle$ the expectation value of the angle θ_i given by

$$p(\vec{\theta}) = \frac{1}{\Psi(\vec{\theta}^0, t)} \exp(-\sum_{i=1}^{N} (\theta_i - \theta_i^0)^2 / 2\nu(T-t) - \phi(\vec{\theta})/\nu) \tag{6}$$

$$\Psi(\vec{\theta}, t) = \vec{\theta}'(\frac{1}{\sqrt{2\pi\nu(T-t)}})^N \exp(-\sum_{i=1}^{N} (\theta_i - \theta_i')^2 / 2\nu(T-t) - \phi(\theta')/\nu) \tag{7}$$

However, the calculation of the partition function $\Psi(\vec{\theta}, t)$ is intractable and so other ways of evaluating the $\langle\theta_i\rangle$ values should be investigated.

Problem statements

- Approximate the expectation value of each θ_i in order to compute the optimal path of a 2D N-joint arm to a pre-specified target using the Mean Field procedure and reproduce the Figure 1.7 in [1]. Create the plot using $N = 3$ and $N = 100$.
- What does influence the accuracy of the approximation?

Implementation

The use of variational inference for the approximation of the expected values of the joint angles $\Psi(\vec{\theta}, t)$ is proposed in [1]. More specifically, he is using the Mean Field algorithm and approximates the intractable distribution $p(\vec{\theta})$ with a factorized Gaussian variational distribution

$$q(\vec{\theta}) = \Pi_{i=1}^{N} \mathcal{N}(\theta_i|\mu_i, \sigma_i) \Rightarrow \tag{8}$$

$$q(\theta) = \Pi_{i=1}^{N} q_i(\theta_i) \tag{9}$$

$$q_i(\theta) = \frac{1}{\sqrt{2\pi\sigma_i}} \exp -(\theta - \mu_i)^2\sigma_i^2 \tag{10}$$

with learnable parameters $\mu_i, \sigma i$. The objective is to minimize the KL divergence between $q(\vec{\theta})$ and $p(\vec{\Theta})$. The KL divergence becomes

$$KL = -\sum_{i}^{N}(\log\sqrt{2\pi\sigma_i^2} + 0.5) + \log\Psi(\vec{\theta}^{old}, t) + \frac{1}{2\nu(T-t)}\sum_{i=1}^{N}(\sigma_i^2 + (\mu_i - \theta_i^{old})^2) + \frac{1}{\nu}\langle\phi(\vec{\theta})\rangle_q \geq 0 \Rightarrow \tag{11}$$

$$\log\Psi(\vec{\theta}^{old}, t) \geq \sum_{i}^{N}(\log\sqrt{2\pi\sigma_i^2} + 0.5) - \frac{1}{2\nu(T-t)}\sum_{i=1}^{N}(\sigma_i^2 + (\mu_i - \theta_i^{old})^2) - \frac{1}{\nu}\langle\phi(\vec{\theta})\rangle_q \Rightarrow \tag{12}$$

$$ELBO = \sum_{i}^{N}(\log\sqrt{2\pi\sigma_i^2} + 0.5) - \frac{1}{2\nu(T-t)}\sum_{i=1}^{N}(\sigma_i^2 + (\mu_i - \theta_i^{old})^2) - \frac{1}{\nu}\langle\phi(\vec{\theta})\rangle_q \tag{13}$$

with

$$\langle\phi(\theta)\rangle_q = \frac{\alpha}{2}\sum_i(1 - \exp\sigma_i^2) + \frac{\alpha}{2}(\langle x_N\rangle - x_{target})^2 + \frac{\alpha}{2}(y_N\rangle - y_{target})^2 \tag{14}$$

$$\langle x_N\rangle = \sum_i \cos\mu_i e^{-\sigma_i/2} \tag{15}$$

$$\langle y_N\rangle = \sum_i \sin\mu_i e^{-\sigma_i/2} \tag{16}$$

The update equations are computed by $\nabla KL(\mu, \sigma) = 0 \Rightarrow \frac{\partial KL}{\partial\mu_i} = 0$ and $\frac{\partial KL}{\partial\sigma_i^2} = 0$ and are given by the following expressions

$$\mu_i^{new} \longleftarrow \theta_i^{old} + \alpha(T-t)(\sin\mu_i e^{-\sigma_i^2/2}(\langle x_N\rangle - x_{target}) - \cos\mu_i e^{-\sigma_i^2/2}(\langle y_N\rangle - y_{target})) \tag{17}$$

$$\frac{1}{\sigma_i^{new2}} \longleftarrow \frac{1}{\nu}(\frac{1}{T-t} + \alpha e^{-\sigma_i^2} - \alpha\cos\mu_i e^{-\sigma_i^2/2}(\langle x_N\rangle - x_{target}) - \alpha\sin\mu_i e^{-\sigma_i^2/2}(\langle y_N\rangle - y_{target})) \tag{18}$$

Algorithm

For our implementation we used the following algorithm

Algorithm 1: Mean Field Approximation for the N-joint Control Problem

Result: $x_N = \sum_{i=1}^{N} \cos\theta_i^{new} \cong x_{target}, y_N = \sum_{i=1}^{N} \sin\theta_i^{new} \cong y_{target}$
initialization of the parameters;
for *time* $< T_{target}$ **do**
  while *Not converged* **do**
    Update μ;
    Update σ;
  end
  $\langle\theta\rangle = \mu$;
  Update u using (5);
  Update θ using (2);
  Update position x_i, y_i using (1);
  Compute $ELBO$ using (13);
end

More specifically, the parameters have to be updated slowly and for that reason the update rule we followed was

$$\mu_i^{finale\ new} = \mu_i^{old} + \eta d\mu \qquad with\ \ dm = \mu_{new} - \mu_{old} \tag{19}$$

$$\sigma_i^{finale\ new} = \sigma_i^{old} + \eta d\sigma \qquad with\ \ d\sigma = \sigma_{new} - \sigma_{old} \tag{20}$$

The updates are occurring when $d\mu$ and $d\sigma$ are relatively small. After the convergence of $d\mu$ and $d\sigma$ the estimate for $\langle\theta_i\rangle = \mu_i^{finale\ new}$.

Results

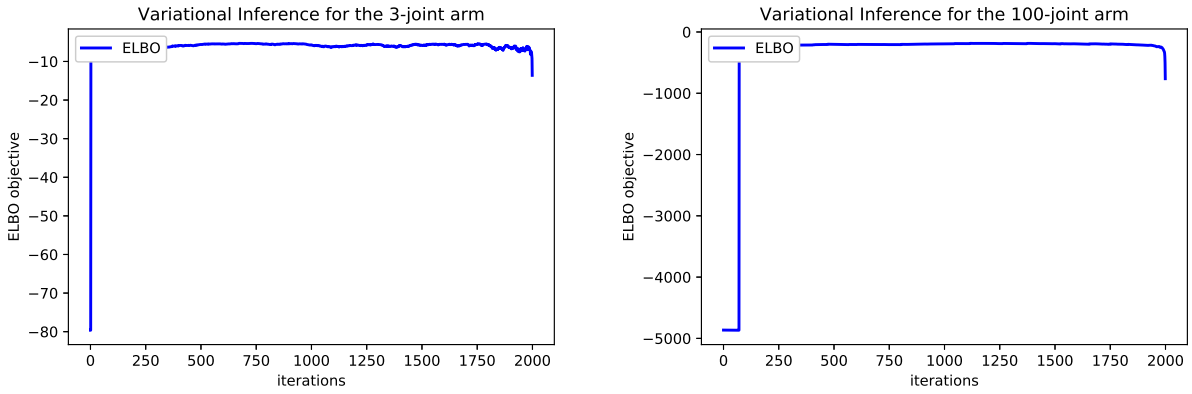
Initially we tried to solve the path integral control of a $N = 3$ joint arm. We noticed that the parameters that affected the success of the approximation where

- The parameter α which affects the cost ϕ (Eq.14)
- the parameter dt which represents the time step of each update.
- the parameter η which is responsible for the update of μ and σ
- the starting points (x_{start}, y_{start}) and the target (x_{target}, y_{target})

Initially, we computed the path integral control using $\alpha = 100, \eta = 0.0001, dt = 0.001, \nu = 0.5$. We chose a high value α because the parameter is proportional to the cost (see Eq.(4)). Namely, we wanted a wrong movement to be much more expensive than it actually is. Regarding the learning rate η we also chose a very small value. The learning rate is used for the update of μ_i, σ_i which are then used for the update of the controller u and eventually the new θ. The reason we chose a small learning rate was because we wanted to have small updates so that the joints want pass the equilibrium. Finally, we chose a really small timing step because we wanted the update to happen gradually so that the system would converge to the right target. In Fig.(2) we can see the gradual configuration of the N-joint arm plot in the moments

$t = 0.05, 0.55, 1.8, \sim 2$. Both $N = 3, 100$-joints arm have reached the target by $t = T$. In Fig.(1), we can see how the ELBO evolves. Interest presents the fact that even thought both ELBOs are converging after a few iterations, namely after a few time steps, the configuration does not. ELBO Eq.(13) is computed by the expectation values and not the actual. Furthermore, we can see that the joint is following the configuration of the expected joint position as it was anticipated. The expected joint position is already in the optimal configuration early (for $t = 1.8$ in the 3-joint arm). On the contrary, for a low $\alpha = 0.1$ Fig.(4), for the 3-joint arm, we notice that the target is not reached. This confirmed our speculation about the dependence of cost to the parameter α, with low cost resulting to mistaken result.

In a second trial, we computed the path integral control using the same $\alpha = 100, \eta = 0.0001, dt = 0.005, \nu = 0.5$, but we chose the starting points and the targets randomly Fig.(3). We this initial points, we observed a behaviour that we did not expect. Both the configurations did not reach the final target. For that reason, we say that the convergence of the inference also depends on the initial points given to the system. The system reaches another equilibrium which is closer to the initial points and for that reason it cannot update.



a *ELBO for a $N = 3$ joint arm.: $\alpha = 100, \eta = 0.0001, dt = 0.001, \nu = 0.5$* **b** *ELBO for a $N = 100$ joint arm: : $\alpha = 100, \eta = 0.0001, dt = 0.001, \nu = 0.5$*

Figure 1 | ***Convergence of ELBO.*** *For both $N = 3$ and $N = 100$ the ELBO converges after a few iterations. Parameters used: : $\alpha = 100, \eta = 0.0001, dt = 0.001, \nu = 0.5$*

Path integral control: $\alpha = 100, \eta = 0.0001, dt = 0.001, \nu = 0.5$

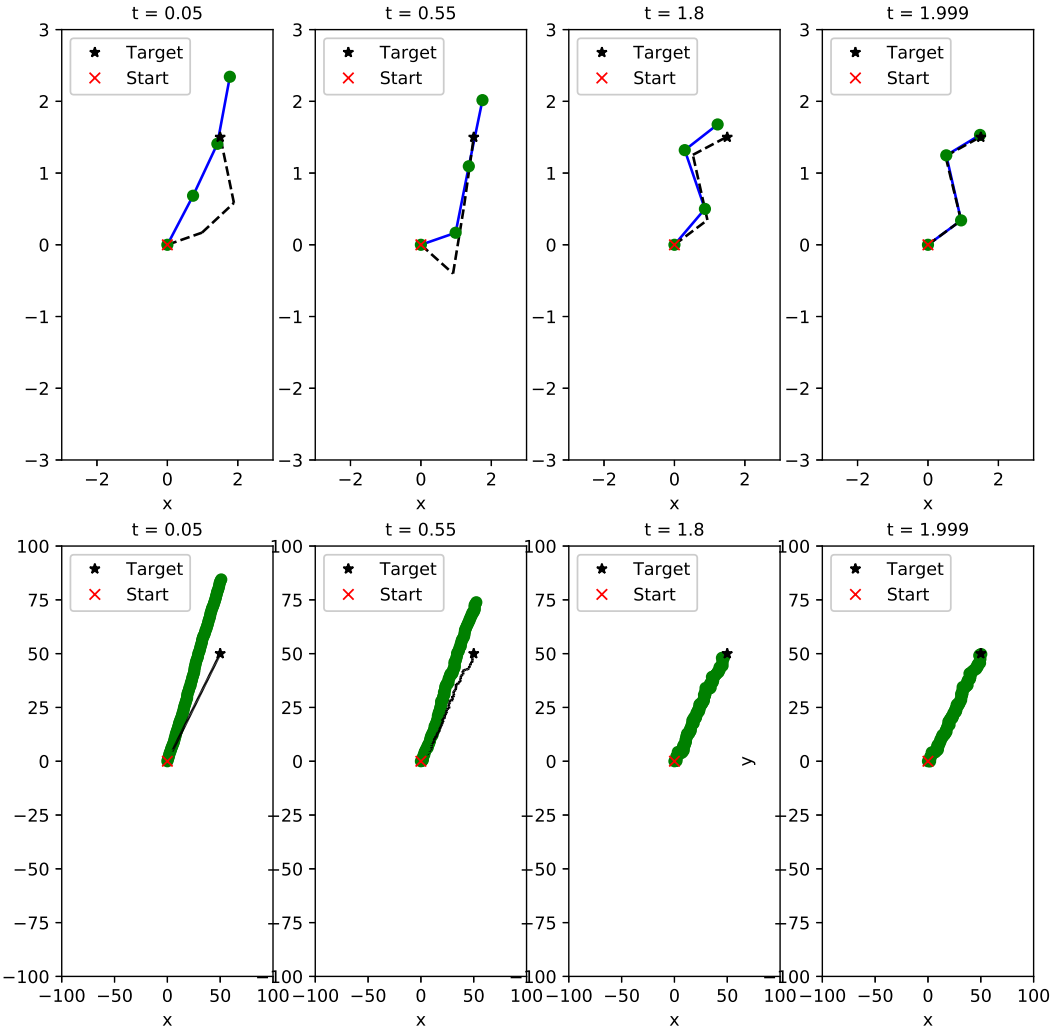


Figure 2 | ***First Row: Path integral control for a*** $N = 3$ ***joint arm. Second row: Path integral control for a*** $N = 100$ ***joint arm.*** *Solid line: Current joint position in the* x, y *plane corresponding to the angle state* $\vec{\theta}$ *at time* t*. Dashed line: Expected joint position computed at the horizon time* $T = 2$ *corresponding to the expected angle state* $\langle\vec{\theta}\rangle$*.In both* $N = 3, 100$ *we can see that the final target is approached.As time progresses, in both cases, the solid and dashed line are converging. Initial Parameters: timing step:* $dt = 0.001$*, joint length:* $l = 1$ *every joint has the same length, time to reach the target:* $T = 2$*, Learning rate for Eq.(20):* $\eta = 0.0001$*,* α *parameter:* $\alpha = 100$ *Eq.(18), noise variance:* $\nu = 0.5$*.*

Path integral control, with random start and target points: $\alpha = 100, \eta = 0.0001, dt = 0.005, \nu = 0.5$

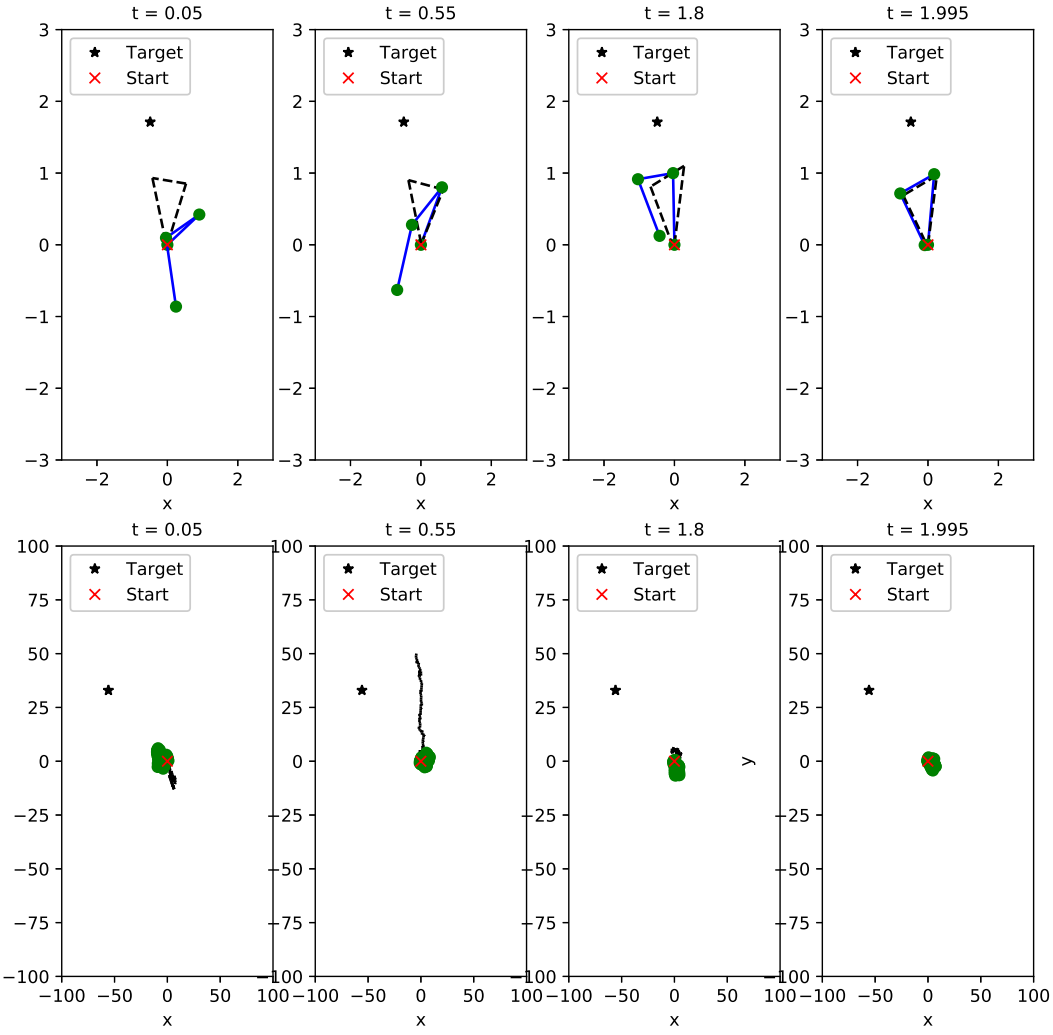


Figure 3 | ***First Row: Path integral control for a*** $N = 3$ ***joint arm. Second row: Path integral control for a*** $N = 100$ ***joint arm.*** *Solid line: Current joint position in the* x, y *plane corresponding to the angle state* $\vec{\theta}$ *at time* t*. Dashed line: Expected joint position computed at the horizon time* $T = 2$ *corresponding to the expected angle state* $\langle\vec{\theta}\rangle$*. The starting points () and the targets were chosen randomly. This time, both of the configurations do not achieve to reach the target. Initial Parameters: timing step:* $dt = 0.005$*, joint length:* $l = 1$ *every joint has the same length, time to reach the target:* $T = 2$*, Learning rate for Eq.(20):* $\eta = 0.0001$*,* α *parameter:* $\alpha = 100$ *Eq.(18), noise variance:* $\nu = 0.5$*.*

Path integral control: $\alpha = 0.1, \eta = 0.0001, dt = 0.001, \nu = 0.5$

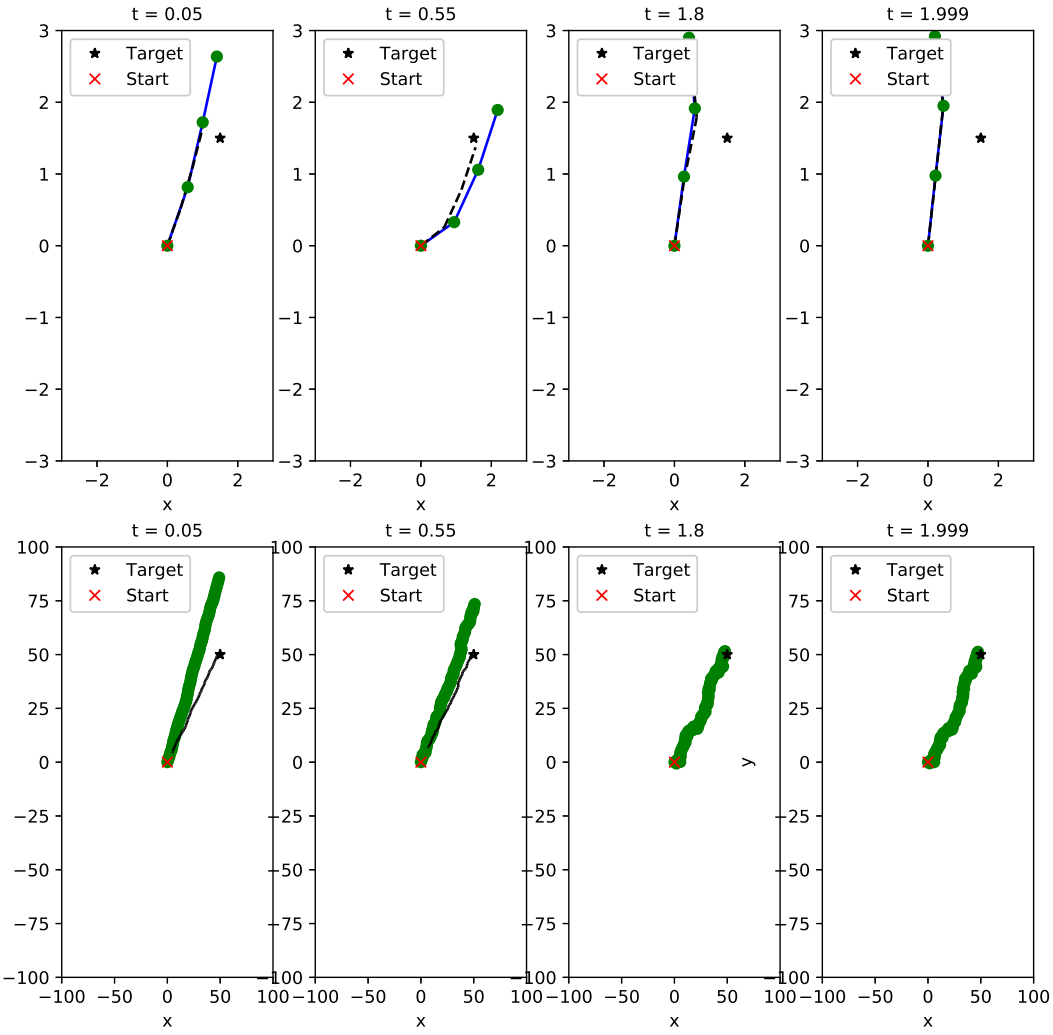


Figure 4 | ***First Row: Path integral control for a*** $N = 3$ ***joint arm. Second row: Path integral control for a*** $N = 100$ ***joint arm.*** *Solid line: Current joint position in the* x, y *plane corresponding to the angle state* $\vec{\theta}$ *at time* t*. Dashed line: Expected joint position computed at the horizon time* $T = 2$ *corresponding to the expected angle state* $\langle\vec{\theta}\rangle$*.In* $N = 100$ *we can see that the final target is approached. On the other hand, for* $N = 3$ *the target is not approached. As time progresses, in both cases, the solid and dashed line are converging. Initial Parameters: timing step:* $dt = 0.005$*, joint length:* $l = 1$ *every joint has the same length, time to reach the target:* $T = 2$*, Learning rate for Eq.(20):* $\eta = 0.0001$*,* α *parameter:* $\alpha = 0.1$ *Eq.(18), noise variance:* $\nu = 100$*.*

Conclusions and Discussion

In this report, we have solved the 2D path integral problem of a N-joint arm using variational inference. More specifically, we used the mean field approximation to compute the distribution of the expected angles $\langle\vec{\theta}\rangle$ so that we can find the optimal controller u to reach a pre-specified target. Initially, we simulated the path integral for $N = 3,100$ and in both cases we reached the target with quite a big accuracy. We investigated which parameters affect the accuracy of the final configuration and we found out that the variance of noise ν, the time step dt, the parameter α as well as the learning rate η influence the final result. It is straight forward that the variance of noise affects the accuracy of the approximation, the highest the variance less accurate the approximation. Furthermore, the smallest the learning rate η the best the approximation as the reach of the equilibrium is happening with smallest steps and it , but since the control problems require high accuracy the trade off time-accuracy is smaller. Parameter α is inverse proportional to the cost of ending up in a state different than the target (Eq.(4)). Therefore, α is logical to take large values, since ending in a different state than the target, has to be quite expensive so that the action wont be considered. In addition, the time step dt also influenced our results with higher dt $\sim (0.01 - 0.1)$ leading to less accurate outcome. This is also explainable by considering the N-joint arm is a system that needs coordination. If small updates are occurring more frequent, the movements will be more elegant and stable. Finally, we conclude that the behaviour of the system also depends on the initial values given (starting and target points). The argumentation behind the previous statement rises from the stability of the system. If the system reaches another equilibrium, it will stop the update.

References

[1] H. J. Kappen. An introduction to stochastic control theory, path integrals and reinforcement learning. In *AIP conference proceedings*, volume 887, pages 149–181. American Institute of Physics, 2007.

Appendix: Implementations

nJointArm.py

```
import numpy as np
import matplotlib.pyplot as plt
import yaml
from math import exp
import random
import math
import sys
import tqdm

class nJointArm:

    def __init__(self,  thetas, ax, n_joints = 3, T =2, v =0.5, alpha =100, lr = 0.0001,  joint_length =1, target_f
        self.n_joints = n_joints
        self.axis = ax
        self.x = [0]*(n_joints+1) #x position of each joint
        self.y = [0]*(n_joints+1) #y position of each joint
        self.joint_angles = thetas
        self.joint_length = joint_length
        self.m = np.zeros(self.n_joints) # initialization of mu_i
        self.s = np.ones(self.n_joints) # initialization of sigma_i
        self.target_flag = target_flag
        self.x_target, self.y_target = self.get_target_location()
        self.x_exp, self.y_exp = self.expected_Values()
        self.total_length = self.joint_length*self.n_joints #total length of the arm
        self.variance = v #variance noise
        self.T = T # Time to reach the target
        self.alpha = alpha #alpha parameter
        self.lr = lr # learning rate
        self.update_position()

    def update_position(self):
        # update the position of the joints
        for i in range(1,self.n_joints+1):
            self.x[i]=self.x[i-1]+np.cos(self.joint_angles[i-1]);
            self.y[i]=self.y[i-1]+np.sin(self.joint_angles[i-1]);

    def expected_Values(self):
        # <x_i>, <y_i>
        # mu, sigma : (3x1) , (3x1)
        # x_expected(0) = 0
        # y_expected(0) = 0
        # x_expected.shape = (4x1) points
        # y_expected.shape = (4x1) points

        x_expected = np.zeros(self.n_joints+1)
        y_expected = np.zeros(self.n_joints+1)
        x_expected[1] = self.joint_length*np.cos(self.m[0])*np.exp(-self.s[0]/2);
```

```
    y_expected[1] = self.joint_length*np.sin(self.m[0])*np.exp(-self.s[0]/2);
    for i in range(2,self.n_joints+1):
        x_expected[i] = x_expected[i-1]+self.joint_length*np.cos(self.m[i-1])*exp(-self.s[i-1]/2);
        y_expected[i] = y_expected[i-1]+self.joint_length*np.sin(self.m[i-1])*exp(-self.s[i-1]/2);

    return x_expected, y_expected

def get_target_location(self):
    # get the coordinates (x,y) of the target location
    if self.target_flag:
        x_target = self.n_joints/2
        y_target = self.n_joints/2
    else:
        # randomly chosen targets from a uniform distribution
        x_target = np.random.uniform(-self.n_joints/1.5,self.n_joints/1.5,1)
        y_target = np.random.uniform(-self.n_joints/1.5,self.n_joints/1.5,1)
    return x_target, y_target

def update_angles(self,new_angles,t):
    # Update of the angles
    self.joint_angles = new_angles
    self.update_position()

def plot_position(self,ax,t):
    # creates the plots of the position of the n-joint arm
    self.x_exp, self.y_exp = self.expected_Values()

    for i in range(self.n_joints+1):
        if i is not self.n_joints:
            ax.plot([self.x[i], self.x[i + 1]],
                     [self.y[i], self.y[i + 1]], 'b-')
            ax.plot([self.x_exp[i], self.x_exp[i+1]], [
             self.y_exp[i], self.y_exp[i+1]], 'k--')
        ax.plot(self.x[i], self.y[i], 'go')

    ax.plot(self.x_target, self.y_target, 'k*', label = 'Target')
    ax.plot(self.x[0], self.y[0], 'rx', label = 'Start')

    ax.set_xlim([-1*self.total_length, 1*self.total_length])
    ax.set_ylim([-1*self.total_length, 1*self.total_length])
    ax.set_title('t = '+str(t), fontsize=10)
    ax.set_xlabel('x'); plt.ylabel('y')
    ax.legend(loc="upper left", fontsize=10)
    plt.draw()



############################## FUNCTIONS ########################################
#################################################################################


def update_parameters( self, t,  mu_old, sigma_old):

    # vdt = var(noise)+mu(noise)
    # a : parameter
    # mu_old and sigma_old
    # returns : mu, sigma and 0 for convergence 1 otherwise

    x_n = np.cos(mu_old).dot(np.exp(-sigma_old.T/2)); # mean end point
```

```
        y_n = np.sin(mu_old).dot(np.exp(-sigma_old.T/2)); # mean end point
        x_target, y_target = self.get_target_location()
        mu = self.joint_angles + self.alpha*(self.T-t)*(np.sin(mu_old)*np.exp(-sigma_old/2)*(x_n - x_target) - \
                              (np.cos(mu_old)*np.exp(-sigma_old/2)*(y_n - y_target)))
        sigma_inv = (1/self.variance)*((1/(self.T-t)) + self.alpha*np.exp(-sigma_old) - \
                                       self.alpha*(np.sin(mu_old)*np.exp(-sigma_old/2.)*(y_n - y_target)) - \
                                           self.alpha*(np.cos(mu_old)*np.exp(-sigma_old/2.)*(x_n - x_target)))

        dm = mu - mu_old
        mu = mu_old + self.lr * dm
        if sigma_inv.all():
            sigma_inv += sys.float_info.epsilon
        dsigma = 1/(sigma_inv) - sigma_old
        sigma = sigma_old + self.lr *dsigma
        if sigma.all():
            sigma += sys.float_info.epsilon
        diff = np.maximum(np.absolute(dm),np.absolute(dsigma))
        d = np.where(diff > 0.01, diff,False) # if dm or dsigma <0.01 then the update has converge
        diff = all(d)

        return mu,sigma,diff


    def new_angles(self, t, dt_temp ,  theta, theta_expected):
        # u: controller
        # t: time point
        # T: total time
        # dt: time_new-time_old (time step)
        # dksi: noise
        # theta_expected = mu
        # formula : dtheta = udt + dxi
        dksi = np.sqrt(self.variance*dt_temp)*np.random.randn(1,self.n_joints) #
        u = self.update_controller( theta, theta_expected, t)
        theta_new = theta + u*dt_temp + dksi
        return theta_new[0]

    def update_controller(self, theta, theta_expected, t):
        # u: controller
        # t: time point
        # T: total time
        return (1/(self.T-t))*(theta_expected - theta)


    def compute_ELBO(self,t):
        # compute the ELBO
        theta_old = self.joint_angles
        x_target,y_target = self.x_target, self.y_target

        x_exp = np.cos(self.m).dot(np.exp(-self.s.T/2)); # mean end

        y_exp = np.sin(self.m).dot(np.exp(-self.s.T/2)); # mean end

        Phi_expected = (self.alpha/2) *( (1-np.exp(-self.s)).sum() + (x_exp - x_target)**2+ (y_exp - y_target)**2 )
        elbo1 = (1/(2*self.variance*(self.T-t)))*(self.s +np.square(self.m - theta_old)).sum()
        elbo2 = (1/self.variance)*Phi_expected
        entropy = -(np.log(np.sqrt(2*np.pi*self.s)) + 0.5).sum()
        ELBO = - entropy  - elbo1 - elbo2
        return ELBO.sum()
```

```
    def plot_diagr(self, ELBO, ax_elbo):#,Hx_mean,F):
        # Creates the figure of the ELBO
        ax_elbo.plot(ELBO, color='b', lw=2.0, label='ELBO')
        ax_elbo.set_title('Variational Inference for the '+str(self.n_joints)+'-joint arm')
        ax_elbo.set_xlabel('iterations'); plt.ylabel('ELBO objective')
        ax_elbo.legend(loc='upper left')
        plt.draw()
        plt.savefig('ELBO_random_'+str(self.n_joints)+'.eps', format='eps')

    def learning(self, times, elbo_flag = True):
        # Learn the parameters
        ELBO = []
        fig_num = 0
        t0 = 0
        for j in tqdm.tqdm(range(len(times))):
                t = times[j]
                dt_temp = t-t0
                diff = 1
                mu_new, sigma_new = self.m,self.s

                while diff:
                    mu_new, sigma_new,diff= self.update_parameters(t, mu_new, sigma_new)
                    theta_new = self.new_angles(t, dt_temp, self.joint_angles, mu_new)

                self.m = mu_new
                self.s = sigma_new
                ELBO.append(self.compute_ELBO(t))
                self.update_angles(theta_new,t)
                t0 = t
                if ((t == 0.05) or (t == 0.55) or (t ==1.8) or (t==times[-1])):
                    self.plot_position(self.axis[fig_num],t)
                    fig_num+=1

        fig_elbo, ax_elbo = plt.subplots(1)
        if elbo_flag:
            self.plot_diagr(ELBO, ax_elbo)

def main():


    with open('config.yml', 'r') as f:
        cfg = yaml.load(f)

    times = np.arange(0, 2, cfg['dt'])
    if cfg['Example1']:
        fig, ax = plt.subplots(2,4,figsize=(10,10))

        thetas = [np.pi/3 for i in range(cfg['n_joints_3'])]
        n_arm3 = nJointArm(thetas, ax[0,:], cfg['n_joints_3'], cfg['T'], cfg['v'], cfg['alpha'] , cfg['lr'], cfg['j

        thetas = [np.pi/3 for i in range(cfg['n_joints_100'])]
        n_arm100 = nJointArm(thetas, ax[1,:], cfg['n_joints_100'], cfg['T'], cfg['v'], cfg['alpha'] , cfg['lr'], cf


        n_arm3.learning(times)
        n_arm100.learning(times)

        plt.show()
        #fig.savefig('PathIntegral.eps', format='eps')
```

```
    elif cfg['Example2']:

        fig, ax = plt.subplots(2,4,figsize=(10,10))

        thetas = [random.uniform(math.radians(-180),math.radians(+180)) for i in range(cfg['n_joints_3'])]
        n_arm3 = nJointArm(thetas, ax[0,:], cfg['n_joints_3'], cfg['T'], cfg['v'], cfg['alpha'] , cfg['lr'], cfg['j

        thetas = [random.uniform(math.radians(-180),math.radians(+180)) for i in range(cfg['n_joints_100'])]
        n_arm100 = nJointArm(thetas, ax[1,:], cfg['n_joints_100'], cfg['T'], cfg['v'], cfg['alpha'] , cfg['lr'], cf


        n_arm3.learning(times)
        n_arm100.learning(times)

        plt.show()
        fig.savefig('PathIntegral_example2.eps', format='eps')


    elif not cfg['target_flag']:
        # targets randomly chosen

        fig, ax = plt.subplots(1,4,figsize=(10,10))
        thetas = [random.uniform(math.radians(-180),math.radians(+180)) for i in range(cfg['n_joints'])]
        n_arm = nJointArm(thetas, ax, cfg['n_joints'], cfg['T'], cfg['v'], cfg['alpha'] , cfg['lr'], cfg['joint_len

        n_arm.learning(times)
        plt.show()
        fig.savefig('PathIntegral_randomTarget.eps', format='eps')


    else:

        fig, ax = plt.subplots(1,4,figsize=(10,10))
        thetas = [random.uniform(math.radians(-180),math.radians(+180)) for i in range(cfg['n_joints'])]
        n_arm = nJointArm(thetas, ax, cfg['n_joints'], cfg['T'], cfg['v'], cfg['alpha'] , cfg['lr'], cfg['joint_len

        n_arm.learning(times)
        plt.show()



if __name__ == '__main__':
    main()
```