PSO_Notebook

November 22, 2018

```
In [1]: import numpy as np
        import random
        import matplotlib.pyplot as plt
        from mpl_toolkits import mplot3d
        import math
        import copy
        import sys
        import pickle
```

0.1 Define Hyperparameters

```
In [2]: ########### Hyper params ###########
        num_particles = 50
        num_iterations = 200
        pop_size = 10
        num_dims = 3
        w=0.8
        c1=1.0
        c2=1.0
        c3=1.0
        seed1=1
        seed2=2
        rgn1 = np.random.RandomState(seed1)
        rgn2 = np.random.RandomState(seed2)
        gbesty = +np.inf
        gbestx = []
        lbestx = []
        print_every = 20
        bounds=[(0,500),(0,500),(0,500)]

        ########PLANT Parameters###########
        L = 0.5      #Length of pendulum
        G = 9.8      #Acceleration due to gravity
        M_p = 0.1    #Mass of pendulum
        M_c = 1      #Mass of Cart
        DT = 0.01    #Discretized time
        T_END = 20   #Simulation time span
        TARGET = 0   #Setpoint
```

```
INITIAL_STATE = [0.1,0.0,0.0,0.0]
threshold = 0.0001
Kp_plots = [] #To save and plot
Ki_plots = []
Kd_plots = []
y_plots  = []
####################################
```

0.1.1 Define Particles

In [3]:
```
class Particle():
    def __init__(self,objective):
        global num_dims
        self.x = np.random.random([num_dims])*10      #current position
        self.y = np.inf                               #current output
        self.v = np.random.uniform(-1,1,[num_dims])  #current velocity
        self.pbesty = self.y                          #personal best score
        self.pbestx = copy.deepcopy(self.x)           #personal best state variables

    def update_velocity(self,gbestx,lbestx=np.zeros(num_dims)):
        #Use PSO update formula
        global w,c1,c2,c3,bounds,num_dims
        rand1 = np.random.random(num_dims)
        rand2 = np.random.random(num_dims)
        rand3 = np.random.random(num_dims)
        new_velocity = np.zeros([num_dims])
        for i in range(num_dims):
            factor1 = w*self.v[i]
            factor2 = c1*rand1[i]*(self.pbestx[i] - self.x[i])
            factor3 = c2*rand2[i]*(gbestx[i] - self.x[i])
            factor4 = c3*rand3[i]*(lbestx[i]-self.x[i])
            new_velocity[i] = factor1+ factor2 + factor3 + factor4
            self.v[i] = new_velocity[i]                              #Update velocity
            self.x[i] +=self.v[i]                            #Update position
            self.x[i] = np.clip(self.x[i],*bounds[i])  #Clip to bounds
```

0.1.2 Define Optimization Methodology and Logic

In [8]:
```
import sys
def PSO(objective):
    global gbestx,gbesty,num_iterations,num_particles,num_dims,pop_size
    particles = []
    current_error = 0
    prev_error = np.inf
    count=1
    #initialise particles
    for i in range(num_particles):
        p = Particle(objective)
```

```python
    particles.append(p)
    if p.y < gbesty:
        gbesty = p.y   #Set initial Global bests
        gbestx = copy.deepcopy(p.x)

#Run Optimization
for i in range(num_iterations):
    #calculate outputs for particles and update pbest,gbest
    output = []
    for p in range(num_particles):
        count = count%pop_size

        #Evaluate objective function for each particle

        particles[p].y = objective(particles[p].x)
        output.append(particles[p].y)

        # Set personal best for particle

        if output[p] < particles[p].pbesty:
            particles[p].pbesty = output[p]
            particles[p].pbestx = copy.deepcopy(particles[p].x)

        #Update Global best

        if output[p] < gbesty or len(gbestx) == 0:
            gbesty = output[p]
            gbestx = copy.deepcopy(particles[p].x)
                 #Update velocity

        if count == 0:
            #improved update law

            lbest = max(particles[p-pop_size+1:p+1], key = lambda p:p.y)
            lbestx = lbest.x
            for j in range(p-pop_size+1,p+1):
                particles[j].update_velocity(gbestx = gbestx,lbestx = lbestx)
            lbest = []
            lbestx=[]
        count+=1
    Kp_plots.append(gbestx[0])
    Ki_plots.append(gbestx[1])
    Kd_plots.append(gbestx[2])
    y_plots.append(gbesty)
    sys.stdout.write("\r %i" % int(i))

print("DONE")
print(gbestx)
```

```
        print(gbesty)
        return gbestx
```

0.1.3 Plant Physics = Objective Function

```
In [13]: def CartPendulumModel(q,t,u):
             dqdt = np.zeros_like(q)

             dqdt[0] = q[1]
             dqdt[2] = q[3]
             dqdt[1] = (M_c+M_p)*G*math.sin(q[0]) - \
             math.cos(q[0])*(-u+M_p*L*math.sin(q[0])*q[1]**2)
             dqdt[1] = dqdt[1] / (4.0/3*(M_p+M_c)*L - M_p*L*math.cos(q[0])**2)
             dqdt[3] = -(-u + M_p*L*(math.sin(q[0])*q[1]**2-math.cos(q[0])\
                                     *dqdt[1]))/(M_p+M_c)

             return dqdt

         def ObjectiveFunction(X):
             Kp=X[0]
             Ki=X[1]
             Kd=X[2]

             total_error = 0
             integrated_sq_error = 0
             time = 0.0

             states = np.array(INITIAL_STATE)

             # ~ TARGET = 0
             TARGET = math.pi * math.sin(time)/30
             prev_error = 0

             T_END = 2

             while (time<T_END):
                 TARGET = math.pi * math.sin(time)/30
                 error = states[0]-TARGET
                 P = Kp * (error)
                 I = Ki * total_error
                 D = Kd * (error-prev_error)/DT
                 F = - (P + I + D)

                 t = np.array([0,DT,2*DT])

                 prev_error = error
                 integrated_sq_error = integrated_sq_error \
                 + (((prev_error+error)/2)**2)*DT
```

```
                total_error = total_error + error*DT

                states = states + CartPendulumModel(states,t,F)*DT
                time=time+DT

                if (abs(states[0])>1.0):
                    # ~ print(abs(states[0]))
                    return 1000000.0/time
            return integrated_sq_error

        def evaluate_rosenbrack(x):
            return (100*(x[0]**2 - x[1])**2 + (1-x[0])**2)

        def evaluate_ratrigin(x):
            return (20 + x[0]**2 - 10*math.cos(2*np.pi*x[0]) + x[1]**2 - \
                    10*math.cos(2*np.pi*x[1]))
```

0.1.4 Run Optimization

```
In [14]: PSO(ObjectiveFunction)
```

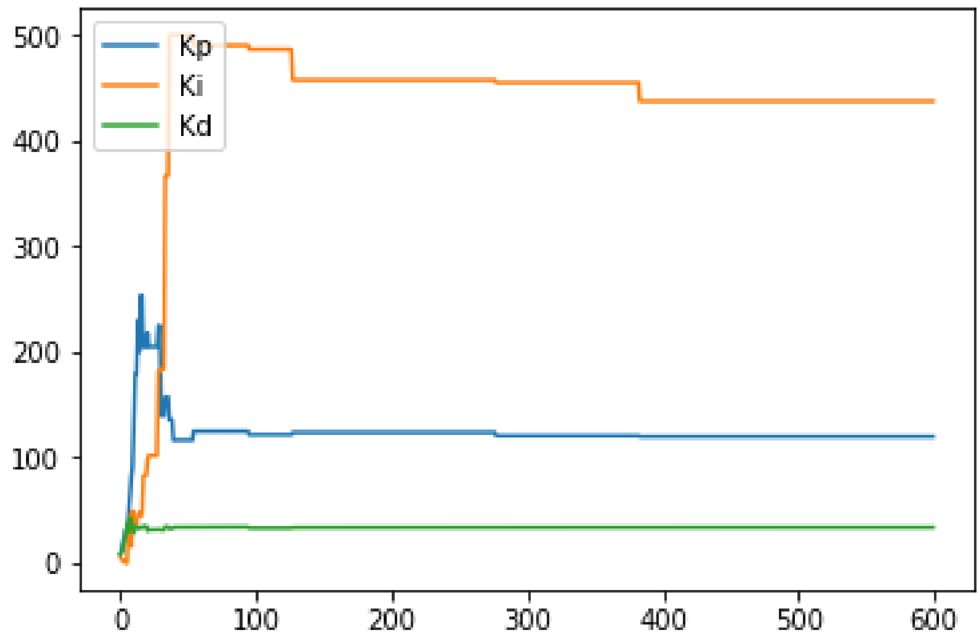
```
 199DONE
[119.74987611 437.43176844  33.77386315]
0.0002747392623207183
```

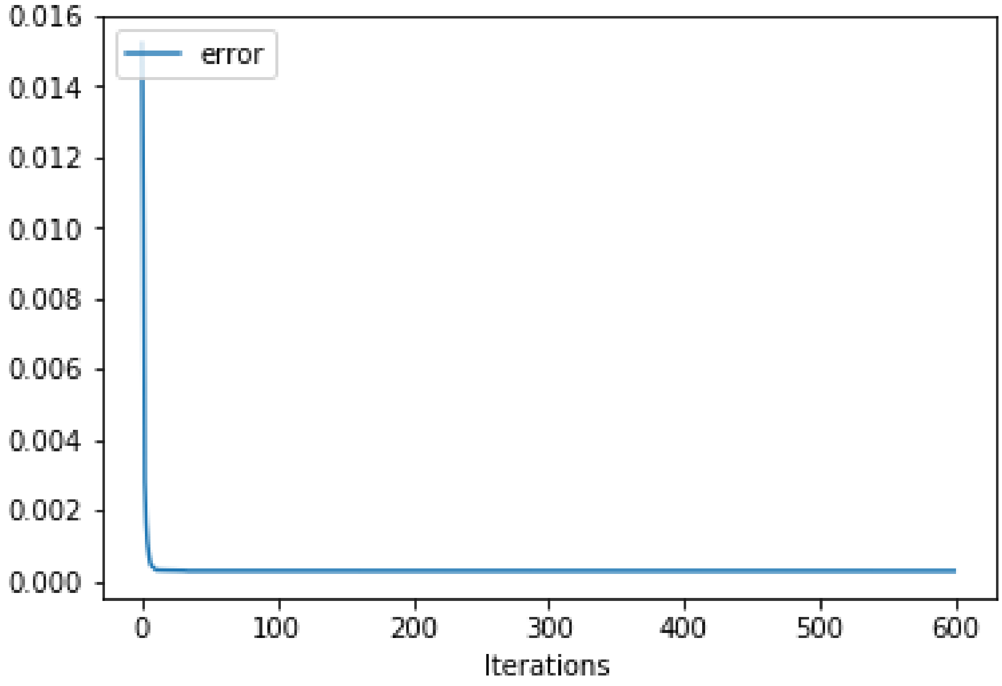
```
Out[14]: array([119.74987611, 437.43176844,  33.77386315])
```

0.1.5 Visualization

```
In [15]: x = np.arange(0,len(Kp_plots),1)
         plt.plot(x,Kp_plots,label = "Kp")
         plt.plot(x,Ki_plots,label = "Ki")
         plt.plot(x,Kd_plots,label = "Kd")
         plt.legend(loc='upper left')
         plt.show()

         plt.plot(x,y_plots,label = "error")
         plt.legend(loc='upper left')
         plt.xlabel("Iterations")
         plt.show()
```


Kp
Ki
Kd
500
400
300
200
100
0
0
100
200
300
400
500
600


error
0.016
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0
100
200
300
400
500
600
Iterations

0.1.6 Saving And Loading Parameters

```
In [ ]: with open("saved_data.txt","wb") as fp:
            pickle.dump(Kp_plots,fp)
            pickle.dump(Ki_plots,fp)
            pickle.dump(Kd_plots,fp)


        with open("saved_data.txt","rb") as fp:
            Kp=pickle.load(fp)
            Ki=pickle.load(fp)
            Kd=pickle.load(fp)


        fp.close()
```