Dupli Checker

PLAGIARISM SCAN REPORT

0% Plagiarised	100% Unique	**Date**	2022-06-24
		Words	382
		Characters	5251

Content Checked For Plagiarism

```
from __future__ import division
import random
import math
import pybamm
#--- Simulation function ----------------------------------------------------------+
def parameter_init(b):
    parameter_values = pybamm.ParameterValues('Chen2020')

    parameter_values["Lower voltage cut-off [V]"] = 3
    parameter_values['Negative electrode thickness [m]']=b[0]
    parameter_values['Positive electrode thickness [m]']=b[1]

    parameter_values['Typical current [A]'] = 0.68
    parameter_values['Negative electrode porosity'] = b[2]
    parameter_values['Positive electrode porosity'] = b[3]
    parameter_values['Negative particle radius [m]'] = b[4]
    parameter_values['Positive particle radius [m]'] = b[5]
    return parameter_values

def solve_model(b):
    parameter_values = parameter_init(b)
    safe_solver = pybamm.CasadiSolver(atol=1e-3, rtol=1e-3, mode="safe")
    model = pybamm.lithium_ion.DFN()
    safe_sim = pybamm.Simulation(model, parameter_values=parameter_values, solver=safe_solver)
    #fast_sim = pybamm.Simulation(model, parameter_values=param, solver=fast_solver)
    safe_sim.solve([0, 3600])
    print("Safe mode solve time: {}".format(safe_sim.solution.solve_time))
    #fast_sim.solve([0, 3600])
    #print("Fast mode solve time: {}".format(fast_sim.solution.solve_time))
    solution = safe_sim.solution
    t = solution["Time [s]"]
    V = solution["Terminal voltage [V]"]
    x1= solution["Discharge capacity [A.h]"]
    xx = x1.entries*V.entries/0.0562
    y1= solution["Power [W]"]

    return xx[-2]
```

```
#First function to optimize
def func1(x):
    m = solve_model(x)
    return m



class particle_gen:
    #initialization of the parameters
    def __init__(self,i0):
        self.p_i=[]
        self.v_i=[]
        self.p_b_i=[]
        self.e_b_i=-1
        self.e_i=-1
     #generation of swarms
        for i in range(0,dim):
            self.v_i.append(random.uniform(-1,1))
            self.p_i.append(i0[i])

    # determining current fitness
    def determine(self,func):
        self.e_i=func(self.p_i)

        # if the current position is the best
        if self.e_i < self.e_b_i or self.e_b_i==-1:
            self.p_b_i_i=self.p_i
            self.e_b_i=self.e_i

    # update new particle velocity
    def new_velocity(self,p_b_g):
        #defining the hyperparameter values
        w=0.5
        c1=2
        c2=2

        for i in range(0,dim):
            r1=random.random()
            r2=random.random()

            v_p=c1*r1*(self.p_b_i[i]-self.p_i[i])
            v_g=c2*r2*(p_b_g[i]-self.p_i[i])
            self.v_i[i]=w*self.v_i[i]+v_p+v_g

    # new particle postion
    def new_position(self,bounds):
        for i in range(0,dim):
            self.p_i[i]=self.p_i[i]+self.v_i[i]

            # keeping within the bounds
            if self.p_i[i]>bounds[i][1]:
                self.p_i[i]=bounds[i][1]
```

```
            if self.p_i[i] < bounds[i][0]:
                self.p_i[i]=bounds[i][0]

class PSO():
    def __init__(self,costFunc,i0,bounds,num_particles,iter):
        global dim

        dim=len(i0)
        e_b_g=-1                    # best error for group
        p_b_g=[]                    # best position for group

        # generate the swarm
        swarm=[]
        for i in range(0,num_particles):
            swarm.append(particle_gen(i0))

        # initialization of iteration
        i=0

        while i < iter:
            try:
                #print i,e_b_g
                # determine the swarm fitness
                for j in range(0,num_particles):
                    swarm[j].determine(costFunc)

                    # determine the current position if it's the global best position
                    if swarm[j].e_i < e_b_g or e_b_g == -1:
                        p_b_g=list(swarm[j].p_i)
                        e_b_g=float(swarm[j].e_i)
                        print (p_b_g)
                        print (e_b_g)

                # iterate over swarm and particle for position and velocity
                for j in range(0,num_particles):
                    swarm[j].new_velocity(p_b_g)
                    swarm[j].new_position(bounds)
                i+=1
                print (p_b_g)
                print (e_b_g)
            except:
                continue


        print ('Optimized:')
        print (p_b_g)
        print (e_b_g)

if __name__ == "__PSO__":
    main()

#--- RUN ----------------------------------------------------------------------+

initial=[8.52e-05,7.56e-05,0.25,0.335,5.86e-06,5.22e-06]
```

```
N_t_min = 0.00005
N_t_max = 0.00015
P_t_min = 0.00005
P_t_max = 0.00015
N_p_min = 0.1
N_p_max = 0.4
P_p_min = 0.1
P_p_max = 0.4
N_r_max = 20.0e-06
N_r_min = 1.0e-06
P_r_min = 1.0e-06
P_r_max = 20.0e-06
bounds=[(0.00005,0.00015),(0.00005,0.00015),(0.1,0.4),(0.1,0.4),(1.0e-06,20.0e-06),(1.0e-06,20.0e-06)]  # input bounds
[(x1_min,x1_max),(x2_min,x2_max)...]
PSO(func1,initial,bounds,num_particles=5,iter=10)
```

Matched Source

No plagiarism found