```matlab
classdef PSO_optimizer

    % OPTIMIZERS class

    properties

        % time constant
        dt = 0.1;

        % max and min joint positions (robot PandaTiago)
        jointsLimits = [
                        [-2.8973, 2.8973] ;
                        [-1.17628, 1.17628];
                        [-2.8973, 2.8973];
                        [-3.0718, -0.0698];
                        [-2.8973, 2.8973];
                        [-0.0175, 3.7525];
                        [-2.8973, 2.8973]
                       ];

        % max joint velocities from specifications
        jointVelocityLimits = [-2.1750 2.1750 ;
                               -2.1750 2.1750 ;
                               -2.1750 2.1750 ;
                               -2.1750 2.1750 ;
                               -2.6100 2.6100 ;
                               -2.6100 2.6100;
                               -2.6100 2.6100];
        baseVelocityLimits = [-1.5 1.5 ;
                              -1.5 1.5 ];

        % max joint, base velocities accounting for the time step (calc. in init
        % function)
        jointVelocityLimitsTS = [];
        baseVelocityLimitsTS = [];

        % prefered joint positions
        jointsPrefered = [0 pi/4 0 -pi/3 0 1.8675 0]';

        % x,y range for base position
        positionLimits = [
                          [-5,5];
                          [-5,5]
                         ];

        % number of parameters
        jointsNum = 9;

        % set type of particles initialization
        initializationMode = "none"; % none, smartbase

        % range of robot base search space
        rangeBase = 1;

        % optimization parameters
        w = 0.6;
        c1 = 0.8;
        c2 = 0.8;

        % particle velocity limit
        particleVelLimit = 2;

        % constant at which stop optimization
        breakConstant = 0.001 %  sub 1mm


        % error rot-pos ratio const
        errRotConst = 1;

        % cost function parts gains
        jointPositionConst = 2;
        positionBaseConst = 2;
        distConst = 15;
        paramChangeConst = 0;

        % max angle of base approach toward EE
        angleMaxApproach = pi/10;

        % param changes
        numChange1 = 0;
        numChange2 = 15;
        numChange3 = 40;
        numChange4 = 60;



        % number of parallel optimizations
        numParallel_0_15 = 5;
        numParallel_15_40 = 5;
        numParallel_40_60 = 5;
        numParallel_60_80 = 5;
```

```
        % number of particles
        numPart_0_15 = 100;
        numPart_15_40 = 500;
        numPart_40_60 = 1000;
        numPart_60_80 = 2000;

        % set values
        partNum = 10000;
        parallelNum = 5;


        % max number of iter
        iterMaxNum = 100;

        % max number of optimisation runs for one point
        maxOptim = 100;

        % imported classes
        robot = {}

    end

    methods

        function obj = PSO_optimizer(obj)

            obj.robot = robotPmb2Panda()

            % calc max velocity step based on time constant
            obj.jointVelocityLimitsTS = obj.jointVelocityLimits * obj.dt;
            obj.baseVelocityLimitsTS = obj.baseVelocityLimits * obj.dt;

        end

        function [path_parameters, convergence_success, convergence_times, convergence_runs] = PSO_optimization_diff_path(obj, path)

            % optimize joint positions for multiple points of EE
            % ------------------------------------------------------------

            path_parameters = zeros(size(path,3), obj.jointsNum);
            convergence_success = ones(1,size(path,3));
            convergence_times = zeros(1,size(path,3));
            convergence_runs = zeros(1,size(path,3));


            % calulate approx base positions

            [basePath, baseStatus] = obj.robot.baseApproxPositions(path);

            % for every point on the path
            for i_point = 1:1:size(path,3)

                t_point = tic();

                T = path(:,:,i_point);

                basePoint = basePath(i_point,:);

                success = [];
                success_points = 0;
                params = [];
                previousParams = [];


                % previous position parameters
                if (i_point > 1)
                    previousParams = path_parameters(i_point-1, :);
                end

                i_optims = 1;

                % run multiple optimizations, until reached max number of
                % tries (we dont always archieve convergence)
                while i_optims <= obj.maxOptim

                    i_optims = i_optims + 1;

                    [param, ~, history_distance, ~] = obj.PSO_optimization_diff(T, basePoint, previousParams);

                    params(i_optim,:) = param;

                    if (history_distance(end) < 0.005)
                        success(i_optim) = history_distance(end);
                        success_points = success_points + 1;
                    else
                        success(i_optim) = 2;
                    end

                    % check if any of the runs achieved convergence
                    [m,I] = min(success);
```

```
            % if converged
            if m < 1
                % save best results
                param = params(I,:);
                path_parameters(i_point,:) = param;
                convergence_success(i_point) = m;
                convergence_times(i_point) = toc(t_point);
                convergence_runs(i_point) = i_optims;

                % display transformation matrix
                [~,~,~,~,~,~,~,~,Tout] = obj.robot.GeometricRobot(param(1:7), param(8:10))

                % display status
                display("Point num.: " + i_point + " Particles num.: " + obj.partNum +  " CONVERGED.")

                break;

            end

            % if not converged display error
            if i_optims >= (obj.maxOptim) && m > 1
                display("Point num.: " + i_point + " Particles num.: " + obj.partNum + " ERROR.")
            end

        end
    end

end

function [path_parameters, convergence_success, convergence_times, convergence_runs] = PSO_optimization_path(obj, path)

    % optimize joint positions for multiple points of EE
    % ------------------------------------------------------------

    path_parameters = zeros(size(path,3), obj.jointsNum);
    convergence_success = ones(1,size(path,3));
    convergence_times = zeros(1,size(path,3));
    convergence_runs = zeros(1,size(path,3));


    % calulate approx base positions

    [basePath, baseStatus] = obj.robot.baseApproxPositions(path);

    % for every point on the path
    for i_point = 1:1:size(path,3)

        t_point = tic();

        T = path(:,:,i_point);

        basePoint = basePath(i_point,:);

        success = [];
        success_points = 0;
        params = [];
        previousParams = [];


        % previous position parameters
        if (i_point > 1)
            previousParams = path_parameters(i_point-1, :);
        end

        i_optims = 1;

        % run multiple optimizations and in parallel
        % (we dont always archieve convergence)
        while i_optims <= obj.maxOptim

            % set particles number and number of parallel runs
            if (i_optims > obj.numChange1)

                obj.partNum = obj.numPart_0_15;
                obj.parallelNum = obj.numParallel_0_15;

            end

            if (i_optims > obj.numChange2)

                obj.partNum = obj.numPart_15_40;
                obj.parallelNum = obj.numParallel_15_40;

            end

            if (i_optims > obj.numChange3)

                obj.partNum = obj.numPart_40_60;
                obj.parallelNum = obj.numParallel_40_60;

            end

            if (i_optims > obj.numChange4)
```

```
                obj.partNum = obj.numPart_60_80;
                obj.parallelNum = obj.numParallel_60_80;

            end


            % parallel execution
            parfor i_optim = 1:1:obj.parallelNum   % PARFOR

                i_optims = i_optims + 1;

                [param, ~, history_distance, ~] = obj.PSO_optimization(T, basePoint, previousParams);

                params(i_optim,:) = param;

                if (history_distance(end) < 0.005)
                    success(i_optim) = history_distance(end);
                    success_points = success_points + 1;
                else
                    success(i_optim) = 2;
                end

            end

            % check if any of the runs achieved convergence
            [m,I] = min(success);

            % if converged
            if m < 1
                % save best results
                param = params(I,:);
                path_parameters(i_point,:) = param;
                convergence_success(i_point) = m;
                convergence_times(i_point) = toc(t_point);
                convergence_runs(i_point) = i_optims;

                % display transformation matrix
                [~,~,~,~,~,~,~,~,Tout] = obj.robot.GeometricRobot(param(1:7), [param(8:9) 0])

                % display status
                display("Point num.: " + i_point + " Particles num.: " + obj.partNum +  " CONVERGED.")

                break;

            end

            % if not converged display error
            if i_optims >= (obj.maxOptim) && m > 1
                display("Point num.: " + i_point + " Particles num.: " + obj.partNum + " ERROR.")
            end

        end



    end

end



function [param, history_cost, history_distance, tim] = PSO_optimization(obj,goalEE, goalBase, previousParams)

% combined position limits of joints and base
combinedLimits = [obj.jointsLimits ; obj.positionLimits ; -pi pi];

% history
history_distance = [];
history_cost = [];

tic()

% generate particles

if obj.initializationMode == "none"
    particles = zeros(obj.partNum, 9);
    % arm joints
    particles(:,1) = rand(obj.partNum,1)*(obj.jointsLimits(1,2)-obj.jointsLimits(1,1)) + obj.jointsLimits(1,1);
    particles(:,2) = rand(obj.partNum,1)*(obj.jointsLimits(2,2)-obj.jointsLimits(2,1)) + obj.jointsLimits(2,1);
    particles(:,3) = rand(obj.partNum,1)*(obj.jointsLimits(3,2)-obj.jointsLimits(3,1)) + obj.jointsLimits(3,1);
    particles(:,4) = rand(obj.partNum,1)*(obj.jointsLimits(4,2)-obj.jointsLimits(4,1)) + obj.jointsLimits(4,1);
    particles(:,5) = rand(obj.partNum,1)*(obj.jointsLimits(5,2)-obj.jointsLimits(5,1)) + obj.jointsLimits(5,1);
    particles(:,6) = rand(obj.partNum,1)*(obj.jointsLimits(6,2)-obj.jointsLimits(6,1)) + obj.jointsLimits(6,1);
    particles(:,7) = rand(obj.partNum,1)*(obj.jointsLimits(7,2)-obj.jointsLimits(7,1)) + obj.jointsLimits(7,1);
    % base position
    particles(:,8) = rand(obj.partNum,1)*(obj.positionLimits(1,2)-obj.positionLimits(1,1)) + obj.positionLimits(1,1);
    particles(:,9) = rand(obj.partNum,1)*(obj.positionLimits(1,2)-obj.positionLimits(1,1)) + obj.positionLimits(1,1);

end

if obj.initializationMode == "smartbase"
    particles = zeros(obj.partNum, 9);
```

```
            % arm joints
            particles(:,1) = rand(obj.partNum,1)*(obj.jointsLimits(1,2)-obj.jointsLimits(1,1)) + obj.jointsLimits(1,1);
            particles(:,2) = rand(obj.partNum,1)*(obj.jointsLimits(2,2)-obj.jointsLimits(2,1)) + obj.jointsLimits(2,1);
            particles(:,3) = rand(obj.partNum,1)*(obj.jointsLimits(3,2)-obj.jointsLimits(3,1)) + obj.jointsLimits(3,1);
            particles(:,4) = rand(obj.partNum,1)*(obj.jointsLimits(4,2)-obj.jointsLimits(4,1)) + obj.jointsLimits(4,1);
            particles(:,5) = rand(obj.partNum,1)*(obj.jointsLimits(5,2)-obj.jointsLimits(5,1)) + obj.jointsLimits(5,1);
            particles(:,6) = rand(obj.partNum,1)*(obj.jointsLimits(6,2)-obj.jointsLimits(6,1)) + obj.jointsLimits(6,1);
            particles(:,7) = rand(obj.partNum,1)*(obj.jointsLimits(7,2)-obj.jointsLimits(7,1)) + obj.jointsLimits(7,1);

            % base position behind EE
            goalAngleEE = wrapToPi(atan2(goalEE(2,3), goalEE(1,3)));
            s = sin(goalAngleEE + pi - obj.angleMaxApproach/2 + obj.angleMaxApproach * rand(obj.partNum,1));
            c = cos(goalAngleEE + pi - obj.angleMaxApproach/2 + obj.angleMaxApproach * rand(obj.partNum,1));

            % scalling for correct distance from mid point
            factorR = rand(obj.partNum,1) * obj.rangeBase ./ sqrt(s.^2 + c.^2);

            particles(:,8) = goalEE(1,4) + obj.rangeBase * factorR .* c;  % rand(obj.partNum,1)*(obj.positionLimits(1,2)-obj.positionLimits(1,1)) + obj.positio
            particles(:,9) = goalEE(2,4) + obj.rangeBase * factorR .* s; % rand(obj.partNum,1)*(obj.positionLimits(1,2)-obj.positionLimits(1,1)) + obj.position


%                 hold off
%                 scatter(particles(:,8),particles(:,9))
%                 hold on
%                 scatter(goalEE(1,4), goalEE(2,4),'x', 'LineWidth',5)
%
%                 drawnow

        end


        % initialize speeds of particles
        particles_d = zeros(size(particles));

        g_best = ones(1,size(particles,2)); % social component
        g_best_I = 1000;

        p_best = particles; % cognitive component - best particle position
        p_best_I= ones(size(particles,1),1)*1000; % cost values of personal bests

        I_all = zeros(obj.partNum,1); % particle cost

        e_all = [];


        % optimization loop

        iternum = 0;

        while true

            % calculate cost of each particle
            for i = 1:1:obj.partNum

                [I_all(i), e_all(i,1), e_all(i,2), e_all(i,3), e_all(i,4)] = obj.costFunction(goalEE, goalBase, particles(i,:), previousParams);

            end

            % check which particle is global best
            [I_best,I_index] = min(I_all);

            % check if g_best is worse than new best and update:
            g_best_I_old = g_best_I;
            g_best_I = (g_best_I <= I_best) * g_best_I + (g_best_I > I_best) * I_best;
            g_best = (g_best_I_old <= I_best)* g_best + (g_best_I_old > I_best)* particles(I_index, :);

            [g_best_dist, ~, ~] = obj.costDistanceNorm(goalEE, particles(I_index, :));

%             if ~(mod(iternum, 20) )
%                 display("Best particles cost:  " + g_best_I)
%                 display("Best particles dist:  " + g_best_dist)
%             end
%             display((particles(I_index, :)))

            history_cost = [history_cost g_best_I];
            history_distance = [history_distance g_best_dist];

            % update personal best, if this generation is better
            p_best = (I_all <= p_best_I)*ones(1,size(particles,2)) .* particles + (I_all > p_best_I)*ones(1,size(particles,2)) .* p_best;
            p_best_I = (I_all <= p_best_I) .* I_all + (I_all > p_best_I) .* p_best_I;


            % calculate rp  and rg random values
            rp = rand(size(particles));
            rg = rand(size(particles));

            % calculate speed updates
            particles_d = obj.w*particles_d + obj.c1*rp .*(p_best-particles) + obj.c2*rg .* (repmat(g_best,obj.partNum, 1) - particles);

            % calculate velocity limits
            particles_d = (particles_d <= -obj.particleVelLimit) * (-obj.particleVelLimit) + (particles_d > -obj.particleVelLimit) .* (particles_d < obj.partic

            % calculate new positions
```

```matlab
            particles_old_xy = particles(:,8:9);
            particles = particles + particles_d;

            % calculate position limits

            if (obj.initializationMode == "none")
                particles = (particles <= (combinedLimits(:,1))') .* (combinedLimits(:,1))'  + (particles >= (combinedLimits(:,2))') .* (combinedLimits(:,2))'
            end

            if (obj.initializationMode == "smartbase")
                % joints
                particles(:,1:7) = (particles(:,1:7) <= (obj.jointsLimits(:,1))') .* (obj.jointsLimits(:,1))'  + (particles(:,1:7) >= (obj.jointsLimits(:,2))')

                % base
                x = particles(:,8);
                y = particles(:,9);

                % scalling for max distance from mid point
                r = sqrt((x-goalEE(1,4)).^2 + (y-goalEE(2,4)).^2);
                particles(:,8:9) = (r > obj.rangeBase) .* particles_old_xy(:,:) + (r <= obj.rangeBase) .* particles(:,8:9);

                % no points in front of the EE
                angleParticle = wrapToPi(atan2(particles(:,9)-goalEE(2,4), particles(:,8)-goalEE(1,4)));
                particles(:,8:9) = (abs(angleParticle - wrapToPi(goalAngleEE + pi)) > obj.angleMaxApproach) .* particles_old_xy(:,:) + (abs(angleParticle - wra

%                 hold off
%                 scatter(particles(:,8),particles(:,9))
%                 hold on
%                 scatter(goalEE(1,4), goalEE(2,4),'x', 'LineWidth',5)
%
%                 drawnow


            end

            % increase iter count
            iternum = iternum + 1;

%             toc();


            % break conditions
            if iternum  > obj.iterMaxNum || g_best_dist < obj.breakConstant || (iternum > 50 && g_best_dist > 0.01) || (iternum > 25 && g_best_dist > 0.1)
%                 display("Duration: " + toc())
                param = particles(I_index, :);

                tim = toc();

                % goal reached
                if (g_best_dist < obj.breakConstant)
                    display("GOAL REACHED, distance: " + g_best_dist +  " Particles num.: " + obj.partNum)
                else
                    display("NOT REACHED, distance: " + g_best_dist + " Particles num.: " + obj.partNum)
                end

               break;
            end
        end
        end


    function [param, history_cost, history_distance, tim] = PSO_optimization_diff(obj,goalEE, goalBase, previousParams)

        % combined position limits of joints and base
        combinedLimits = [obj.jointVelocityLimitsTS ; obj.baseVelocityLimitsTS];

        % history
        history_distance = [];
        history_cost = [];

        tic()

        % generate particles
        % -----------------------------------------------------

        particles = zeros(obj.partNum, 9);

        % arm joints (dq1 ... dq7)
        particles(:,1) = rand(obj.partNum,1)*(obj.jointVelocityLimitsTS(1,2)-obj.jointVelocityLimitsTS(1,1)) + obj.jointVelocityLimitsTS(1,1);
        particles(:,2) = rand(obj.partNum,1)*(obj.jointVelocityLimitsTS(2,2)-obj.jointVelocityLimitsTS(2,1)) + obj.jointVelocityLimitsTS(2,1);
        particles(:,3) = rand(obj.partNum,1)*(obj.jointVelocityLimitsTS(3,2)-obj.jointVelocityLimitsTS(3,1)) + obj.jointVelocityLimitsTS(3,1);
        particles(:,4) = rand(obj.partNum,1)*(obj.jointVelocityLimitsTS(4,2)-obj.jointVelocityLimitsTS(4,1)) + obj.jointVelocityLimitsTS(4,1);
        particles(:,5) = rand(obj.partNum,1)*(obj.jointVelocityLimitsTS(5,2)-obj.jointVelocityLimitsTS(5,1)) + obj.jointVelocityLimitsTS(5,1);
        particles(:,6) = rand(obj.partNum,1)*(obj.jointVelocityLimitsTS(6,2)-obj.jointVelocityLimitsTS(6,1)) + obj.jointVelocityLimitsTS(6,1);
        particles(:,7) = rand(obj.partNum,1)*(obj.jointVelocityLimitsTS(7,2)-obj.jointVelocityLimitsTS(7,1)) + obj.jointVelocityLimitsTS(7,1);
        % base (v,w)
        particles(:,8) = rand(obj.partNum,1)*(obj.baseVelocityLimitsTS(1,2)-obj.baseVelocityLimitsTS(1,1)) + obj.baseVelocityLimitsTS(1,1);
        particles(:,9) = rand(obj.partNum,1)*(obj.baseVelocityLimitsTS(1,2)-obj.baseVelocityLimitsTS(1,1)) + obj.baseVelocityLimitsTS(1,1);


        % generate pso variables
        % -----------------------------------------------------
```

```
        % initialize speeds of particles
        particles_d = zeros(size(particles));

        g_best = ones(1,size(particles,2)); % social component
        g_best_I = 1000;

        p_best = particles; % cognitive component - best particle position
        p_best_I= ones(size(particles,1),1)*1000; % cost values of personal bests

        I_all = zeros(obj.partNum,1); % particle cost

        e_all = [];


        % optimization loop
        % -----------------------------------------------------


        iternum = 0;

        while true

            % calculate cost of each particle
            for i = 1:1:obj.partNum

                [I_all(i)]  = obj.costFunctionDiff(goalEE, goalBase, particles(i,:), previousParams);

            end

            % check which particle is global best
            [I_best,I_index] = min(I_all);

            % check if g_best is worse than new best and update:
            g_best_I_old = g_best_I;
            g_best_I = (g_best_I <= I_best) * g_best_I + (g_best_I > I_best) * I_best;
            g_best = (g_best_I_old <= I_best)* g_best + (g_best_I_old > I_best)* particles(I_index, :);

            [g_best_dist, ~, ~] = obj.costDistanceNorm(goalEE, particles(I_index, :));

%             if ~(mod(iternum, 20) )
%                 display("Best particles cost:  " + g_best_I)
%                 display("Best particles dist:  " + g_best_dist)
%             end
%             display((particles(I_index, :)))

            history_cost = [history_cost g_best_I];
            history_distance = [history_distance g_best_dist];

            % update personal best, if this generation is better
            p_best = (I_all <= p_best_I)*ones(1,size(particles,2)) .* particles + (I_all > p_best_I)*ones(1,size(particles,2)) .* p_best;
            p_best_I = (I_all <= p_best_I) .* I_all + (I_all > p_best_I) .* p_best_I;


            % calculate rp  and rg random values
            rp = rand(size(particles));
            rg = rand(size(particles));

            % calculate speed updates
            particles_d = obj.w*particles_d + obj.c1*rp .*(p_best-particles) + obj.c2*rg .* (repmat(g_best,obj.partNum, 1) - particles);

            % calculate velocity limits
            particles_d = (particles_d <= -obj.particleVelLimit) * (-obj.particleVelLimit) + (particles_d > -obj.particleVelLimit) .* (particles_d < obj.partic

            % calculate new positions
            particles_old_xy = particles(:,8:9);
            particles = particles + particles_d;

            % calculate position limits
            particles = (particles <= (combinedLimits(:,1))') .* (combinedLimits(:,1))'  + (particles >= (combinedLimits(:,2))') .* (combinedLimits(:,2))' + (



%                 hold off
%                 scatter(particles(:,8),particles(:,9))
%                 hold on
%                 scatter(goalEE(1,4), goalEE(2,4),'x', 'LineWidth',5)
%
%                 drawnow



            % increase iter count
            iternum = iternum + 1;

%             toc();

            % break conditions
            if iternum  > obj.iterMaxNum || g_best_dist < obj.breakConstant || (iternum > 50 && g_best_dist > 0.01) || (iternum > 25 && g_best_dist > 0.1)
%                 display("Duration: " + toc())
                param = particles(I_index, :);

                tim = toc();
```

```
                % goal reached
                if (g_best_dist < obj.breakConstant)
                    display("GOAL REACHED, distance: " + g_best_dist +  " Particles num.: " + obj.partNum)
                else
                    display("NOT REACHED, distance: " + g_best_dist + " Particles num.: " + obj.partNum)
                end

               break;
            end
        end
    end



    function [costs, errPos, errRot] = costDistanceNorm(obj, goalEE, params)

        global rG
        global rEE

        % COST DISTANCE NORM
        % simplest of cost functions, just second norm of error between EE distance and goal

        % Approach described in article: Particle swarm optimization for inverse kinematics solution and trajectory planning of 7-DOF and 8-DOF robot manipulat

        % calc direct kinematics
        [~,~,~,~,~,~,~,~,T] = obj.robot.GeometricRobot(params(1:7), params(8:10));

        % calculated pos & rot
        pEE = T(1:3,4)';
        rEE = rotm2quat(T(1:3,1:3));
%         rEEe = rotm2eul(T(1:3,1:3));

        % get goal pos & rot
        pG = goalEE(1:3,4)';
        rG = rotm2quat(goalEE(1:3,1:3));
%         rGe = rotm2eul(goalEE(1:3,1:3));

        % position error
        errPos = norm((pG - pEE));

        % rotation error

        % simple way
        errRotS = abs(rG(1)-rEE(1));
        errRotV = norm(rG(2:end) - rEE(2:end));
        errRot = (errRotS + 2 * errRotV) * obj.errRotConst;


        % total error
        costs = (errPos + obj.errRotConst * errRot);

    end

    function costs = calcJointsPositionError(obj, params)

        % COST ERROR FROM PREFERED JOINT POSITIONS

        jointsError = obj.jointsPrefered - params(1:7)';
        costs = norm(jointsError,2);

    end

    function costs = calcBasePositionError(obj, goal, params)

        % COST ERROR OF BASE POSITION DEVIATION FROM GOAL

        x = params(8);
        y = params(9);

        costs = norm([x-goal(1), y-goal(2)],2);

    end


    function [costs]  = costFunctionDiff(obj, goalEE, goalBase, params, previousStates)

        % COST FUNCTION
        % cost functions, second norm of error between EE distance and goaL
        %
        % -----------------------------------
        % goalEE - T mat [4x4]
        % goalBase - x,y,fi
        % previousStates - q1 ... q7, x, y, fi

        % calculate new joint angles (changes are already accounting
        % for time-step duration)
        states(1:7) = previousStates(1:7) + params(1:7);

        % wheel speeds to vw
        base_wv = obj.robot.convertWheelsToVW(obj, params(8:9));

        % vw to xy and phi
```

```
        states(8:10) = previousStates(8:10) + [base_wv(1) * cos(previousStates(10)+base_wv/2);
                 base_wv(1) * sin(previousStates(10)+base_wv/2);
                 previousStates(10) + base_wv(2) ]';

        % calculate <EE - goal EE> distance
        [costEE, ~, ~] = obj.costDistanceNorm(goalEE, states);

        % calculate <base - goal base> distance
        costBase = sqrt((states(8)-goalBase(1))^2+(states(9)-goalBase(2))^2);

        % total cost
        costs = costEE + costBase;


    end


    function [costs, eDist, eJoint, eBase, ePrevious]  = costFunction(obj,goalEE, goalBase, params, previousParams)

        % COST FUNCTION
        % cost functions, second norm of error between EE distance and goaL
        % with added preference for certain joints poses

        [distErr, errPos, errRot] = obj.costDistanceNorm(goalEE, params);

        % total error

        eDist = obj.distConst * distErr;
        eJoint = obj.jointPositionConst * obj.calcJointsPositionError(params);
        eBase = obj.positionBaseConst * obj.calcBasePositionError(goalBase, params);
        ePrevious = 0;

        if previousParams
            ePrevious = obj.paramChangeConst * obj.paramChangeError(params, previousParams);
        end


        costs = eDist + eJoint + eBase + ePrevious;

        % if previous joint positions are aveliable calculate close
        % solution


    end

    function cost = paramChangeError(obj, params, previousParams)

        cost = norm([previousParams-params],2); % !!!!

    end


end
end
```

```
obj =

  PSO_optimizer with properties:

                     dt: 0.1000
           jointsLimits: [7×2 double]
     jointVelocityLimits: [7×2 double]
      baseVelocityLimits: [2×2 double]
   jointVelocityLimitsTS: []
    baseVelocityLimitsTS: []
          jointsPrefered: [7×1 double]
          positionLimits: [2×2 double]
               jointsNum: 9
      initializationMode: "none"
               rangeBase: 1
                       w: 0.6000
                      c1: 0.8000
                      c2: 0.8000
        particleVelLimit: 2
           breakConstant: 1.0000e-03
             errRotConst: 1
      jointPositionConst: 2
       positionBaseConst: 2
               distConst: 15
         paramChangeConst: 0
        angleMaxApproach: 0.3142
              numChange1: 0
              numChange2: 15
              numChange3: 40
              numChange4: 60
         numParallel_0_15: 5
        numParallel_15_40: 5
        numParallel_40_60: 5
        numParallel_60_80: 5
             numPart_0_15: 100
            numPart_15_40: 500
```

```
            numPart_40_60: 1000
            numPart_60_80: 2000
                  partNum: 10000
              parallelNum: 5
               iterMaxNum: 100
                 maxOptim: 100
                    robot: [1×1 robotPmb2Panda]


ans = 

  PSO_optimizer with properties:

                       dt: 0.1000
             jointsLimits: [7×2 double]
      jointVelocityLimits: [7×2 double]
       baseVelocityLimits: [2×2 double]
    jointVelocityLimitsTS: [7×2 double]
     baseVelocityLimitsTS: [2×2 double]
           jointsPrefered: [7×1 double]
           positionLimits: [2×2 double]
                jointsNum: 9
       initializationMode: "none"
                rangeBase: 1
                        w: 0.6000
                       c1: 0.8000
                       c2: 0.8000
         particleVelLimit: 2
            breakConstant: 1.0000e-03
              errRotConst: 1
       jointPositionConst: 2
        positionBaseConst: 2
                distConst: 15
         paramChangeConst: 0
         angleMaxApproach: 0.3142
               numChange1: 0
               numChange2: 15
               numChange3: 40
               numChange4: 60
         numParallel_0_15: 5
        numParallel_15_40: 5
        numParallel_40_60: 5
        numParallel_60_80: 5
             numPart_0_15: 100
            numPart_15_40: 500
            numPart_40_60: 1000
            numPart_60_80: 2000
                  partNum: 10000
              parallelNum: 5
               iterMaxNum: 100
                 maxOptim: 100
                    robot: [1×1 robotPmb2Panda]
```

Published with MATLAB® R2022b