1. Mathematically, the smaller the Δq is, the more accurate the gradient is calculated. However, since we calculate the forward kinematics using the physical simulator, too small Δq will not result in expected changes, or any changes in the forward kinematics due to physical limitations. However, if the Δq is too large, the gradient is not accurate. When Δq is around 0.01, the performance is optimal.

Δq	Position Error (m)	Orientation Error (rad)
0.005	0.000387	0.008021
0.01	0.000674	0.007756
0.015	0.001030	0.008015

2. The first checker, is_state_valid, will make the robot EEF stay within the boundary without collisions between the robot and the ground, as well as making sure that the EEF has a high probability of pushing the cubes (EEF coordinates kept within the workspace). The second checker, is_state_high_quality will ensure the robot arm to have enough manipulability, and avoid weird poses that will limit the further movements of the arm and ensure the EEF and move in all direction.

3. My method of optimization is inherited from the paper provided. The code snippet is attached below.

```
def path_optimization(self, plan, time_budget = 200.0, K = 10):
    start_time = time.time()
    original_plan_dist = 0
    for i in range (1, len(plan)):
        original_plan_dist += np.linalg.norm(plan[i].state["stateVec"][:7] - plan[i - 1].state["stateVec"][:7])
    print("Original Distance = ", original_plan_dist)
    optimized_plan_dist = original_plan_dist
    while (time.time() - start_time < time_budget):
        #Generate k candidate plans and choose the best from it
        best_candidate = None
        for k in range (K):
            candidate_plan = self.add_noise(plan)
            candidate_plan_dist = 0
            #Calculate the distance of this candidate plan
            for i in range (1, len(candidate_plan)):
                state, valid = self.pdef.propagate(candidate_plan[i - 1].state, candidate_plan[i].control)
                if valid and self.pdef.is_state_valid(state):
                    candidate_plan[i].state = state
                else:
                    break
                candidate_plan_dist += np.linalg.norm(state["stateVec"][:7] - candidate_plan[i - 1].state["stateVec"][:7])
            if (self.pdef.goal.is_satisfied(state) and candidate_plan_dist < optimized_plan_dist):
                best_candidate = candidate_plan
                optimized_plan_dist = candidate_plan_dist
                print("Optimized Distance", optimized_plan_dist)
        if best_candidate is not None:
            plan = best_candidate
            print("Best plan Updated!", optimized_plan_dist)
    return plan
```

It basically generates K noisy candidates to find one best candidate route that has less cost and also achieves the goal. The cost is defined as the total difference between the state vectors, which is a measure of the total rotations of all the joints. If the candidate is found, the plan is updated, and we will use the best candidate so far to continue adding on noises to optimize.

The below is an example of an optimized plan. The original plan is shown as the red route and the optimized plan is shown as the blue route. The original cost is 5.16 and the optimized cost is 3.76.