1 Introduction

This is accodring to paper [1].

2 Offline Preprocessing

First we construct safe set with lower and upper bounds.

2.1 Safe Set

Major class for making safe set.

1. Making Box.
2. Making BoxCollection.
3. Making LineGraph.

```
class SafeSet:
    def __init__(self, L, U, verbose=False):
        assert L.shape == U.shape
        boxes = [Box(l, u) for l, u in zip(L, U)]
        self.B: BoxCollection = BoxCollection(boxes, verbose)
        self.G: LineGraph = self.B.line_graph(verbose)
```

Listing 1: Safe Set

2.2 Box

Class for making boxes with lower and upper bounds.

```
class Box:
    def __init__(self, l, u):
        self.d = len(l)
        self.l = np.array(l)
        self.u = np.array(u)
        assert all(self.u >= self.l)
        self.c = (self.l + self.u) / 2
```

Listing 2: Box

2.3 BoxCollection

Class that contains all boxes and compute intersections between them.

1. Get boxes.
2. Get lower and upper bounds.
3. Sort array to get the right order.
4. Compute intersections.

```
class BoxCollection:
    def __init__(self, boxes: list[Box], verbose=True, tol=0):
        self.boxes = boxes
        self.n = len(boxes)
        self.d = boxes[0].d

        # Compute lower and upper bounds.
        self.ls = np.vstack([box.l for box in boxes]).T
        self.us = np.vstack([box.u for box in boxes]).T
        self.orders = np.empty((self.d, 2, self.n), dtype=int)
        self.orders_inv = np.empty((self.d, 2, self.n), dtype=int)

        # Sort arrays to get the right order.
        def sort_coordinates_construct_forward_backwards(
            vals: np.ndarray,
            order: np.ndarray,
```

```
            order_inv: np.ndarray,
        ):
            order[:] = np.argsort(vals)
            order_inv[order] = np.arange(self.n)
            vals[:] = vals[order]


        for i in range(self.d):
            sort_coordinates_construct_forward_backwards(
                    self.ls[i], self.orders[i, 0], self.orders_inv[i, 0])
            sort_coordinates_construct_forward_backwards(
                    self.us[i], self.orders[i, 1], self.orders_inv[i, 1])

        # Compute intersections.
        self.inters = self._intersections(0, tol)
        for i in range(1, self.d):
            for k, box_indices in self._intersections(i, tol).items():
                self.inters[k] = self.inters[k] & box_indices

    def _intersections(self, i, tol=0):
        inters = {k: set() for k in range(self.n)}
        for l, k in enumerate(self.orders[i, 0]):
            xi = self.boxes[k].u[i]
            for m in self._ilcontain(i, xi, tol, first=l+1):
                inters[k].add(m)
                inters[m].add(k)

        return inters
```

Listing 3: Box Collection

2.4 Line Graph

Making line graph of a box collection.

1. Getting line graph with nx.line_graph from networkx library.
2. Making each node corresponds to each edge of original graph.
3. Making each edge corresponds to a pair of intersections edges of the original graph.
4. Add all of that to the line graph.
5. Compute intersections with two or more nodes in the graph.
6. Getting distance between nodes and saving them.
7. Define Variables.
   - pu: point in node u.
   - pv: point in node v.
   - $weight$: distance between pu and pv.
   - adj_mat: adjacency matrix for scipy's shortest-path algorithms.
   - $v2i$: vertex to index.
   - $i2v$: index to vertex.
   - box: box intersection.
   - $point$: representative point in each box intersection.
   - l: lower bound.
   - u: upper bound.
8. Constraints.
   - $constraints = [x >= l, x <= u]$: constraints.
9. Problem.
   - $cost = cp.sum(cp.norm(y, 2, axis = 1))$: cost function.

- $prob = cp.Problem(cp.Minimize(cost), constraints)$: problem.

```python
class LineGraph(nx.Graph):

    def __init__(self, B: BoxCollection, verbose=True, *args, **kwargs):

        # Initialize and store boxes.
        super().__init__(*args, **kwargs)
        self.B = B

        # Compute line graph using networkx.
        inters_graph = nx.Graph()
        inters_graph.add_nodes_from(B.inters.keys())
        for k, k_inters in B.inters.items():
            k_inters_unique = [l for l in k_inters if l > k]
            inters_graph.add_edges_from(product([k], k_inters_unique))
        line_graph = nx.line_graph(inters_graph)
        self.add_nodes_from(line_graph.nodes)
        self.add_edges_from(line_graph.edges)
        self.v2i = {v: i for i, v in enumerate(self.nodes)}
        self.i2v = {i: v for i, v in enumerate(self.nodes)}

        # Pair each vertex with the corresponding box intersection.
        for v in self.nodes:
            boxk = B.boxes[v[0]]
            boxl = B.boxes[v[1]]
            self.nodes[v]['box'] = boxk.intersect(boxl)

        # Place representative point in each box intersection.
        self.optimize_points()

        # Assign fixed length to each edge of the line graph.
        for e in self.edges:
            pu = self.nodes[e[0]]['point']
            pv = self.nodes[e[1]]['point']
            self.edges[e]['weight'] = np.linalg.norm(pv - pu)

        # Store adjacency matrix for scipy's shortest-path algorithms.
        self.adj_mat = nx.to_scipy_sparse_array(self)

    def optimize_points(self):

        # Making variable for each node.
        x = cp.Variable((self.number_of_nodes(), self.B.d))
        # Assigning each node to the corresponding box.
        x.value = np.array([self.nodes[v]['box'].c for v in self.nodes])

        # Making constraints, Lower and Upper Bounds.
        l = np.vstack([self.nodes[v]['box'].l for v in self.nodes])
        u = np.vstack([self.nodes[v]['box'].u for v in self.nodes])
        constraints = [x >= l, x <= u]

        A = nx.incidence_matrix(self, oriented=True)
        y = A.T.dot(x)

        # Objective function to minimize.
        cost = cp.sum(cp.norm(y, 2, axis=1))

        # Solving the problem.
        prob = cp.Problem(cp.Minimize(cost), constraints)
        prob.solve(solver='CLARABEL')

        # Saving the points.
        for i, v in enumerate(self.nodes):
            self.nodes[v]['point'] = x[i].value
```

Listing 4: Line Graph

3 Polygonal Phase

We here design a polygonal curve that contains initial and terminal points.

1. Finding shortest path to terminal point.

2. Iterative planner.

3. Plan.

```
def plan(
        S: SafeSet,
        p_init: np.ndarray,
        p_term: np.ndarray,
        T: int,
        alpha: list,
        der_init={},
        der_term={},
        verbose=True
    ):
    discrete_planner, runtime = S.G.shortest_path(p_term)
    box_seq, length, runtime = discrete_planner(p_init)
    box_seq, traj, length, solver_time = iterative_planner(S.B, p_init, p_term, box_seq, verbose)
```

Listing 5: Polygonal Phase

3.1 Finding Shortest Path to Terminal Point

Finding shortest path using dijkstra algorithm from scipy library.

```
def shortest_path(self, goal):
        tic = time()

        rows = []
        data = []
        # Add edges to goal.
        for k in self.B.contain(goal):
            for l in self.B.inters[k]:
                v = self.node(k, l)
                i = self.v2i[v]
                rows.append(i)
                pv = self.nodes[v]['point']
                data.append(np.linalg.norm(goal - pv))
        cols = [0] * len(rows)
        shape = (len(self.nodes), 1)

        # Add edges to other nodes.
        adj_col = sp.sparse.csr_matrix((data, (rows, cols)), shape)
        adj_mat = sp.sparse.bmat([[self.adj_mat, adj_col], [adj_col.T, None]])


        # Compute shortest path.
        dist, succ = sp.sparse.csgraph.dijkstra(
            csgraph=adj_mat,
            directed=False,
            return_predecessors=True,
            indices=-1
        )

        # Construct planner.
        planner = lambda start: self._planner_all_to_one(start, dist, succ)

        return planner, time() - tic
```

Listing 6: Finding shortest path to terminal point

3.2 Iterative Planner

The function uses an iterative optimization-based approach to refine the initial sequence of boxes in "box_seq" and generate a trajectory that connects them. The optimization problem seeks to minimize the distance between adjacent boxes in the sequence subject to constraints on the dynamics of the system and the bounds of the state space.

1. Remove repetitions.

2. Solve minimum distance problem.

3. Check if solution is feasible.

4. Check if solution is optimal.
5. Insert new box.
6. Repeat.
7. Return box sequence, trajectory, length, solver time.

```python
def iterative_planner(B, start, goal, box_seq, verbose=True, tol=1e-5, **kwargs):
    box_seq = np.array(box_seq)
    solver_time = 0
    n_iters = 0
    while True:
        n_iters += 1


        # Remove repetitions
        box_seq = jump_box_repetitions(box_seq)
        # Solve minimum distance problem.
        traj, length, solver_time_i = solve_min_distance(B, box_seq, start, goal, **kwargs)
        solver_time += solver_time_i

        # Check if solution is feasible.
        box_seq, traj = merge_overlaps(box_seq, traj, tol)

        # Check if solution is optimal.
        kinks = find_kinks(traj, tol)

        insert_k = []
        insert_i = []
        # Find the box that maximizes the norm of the force.
        for k in kinks:
            i1 = box_seq[k - 1]
            i2 = box_seq[k]
            B1 = B.boxes[i1]
            B2 = B.boxes[i2]
            cached_finf = 0

            subset = list(B.inters[i1] & B.inters[i2])
            for i in B.contain(traj[k], tol, subset):
                B3 = B.boxes[i]
                B13 = B1.intersect(B3)
                B23 = B2.intersect(B3)

                # Compute force. (force: f = (x2 - x1) - (x3 - x2))
                f = dual_box_insertion(*traj[k-1:k+2], B13, B23, tol)

                # Compute norm of force.
                f2 = np.linalg.norm(f)

                # Compute infinity norm of force.
                finf = np.linalg.norm(f, ord=np.inf)

                # Update cached values.
                if f2 > 1 + tol and finf > cached_finf + tol:
                    cached_i = i
                    cached_finf = finf

            if cached_finf > 0:
                insert_k.append(k)
                insert_i.append(cached_i)

        # Insert new box.
        if len(insert_k) > 0:
            box_seq = np.insert(box_seq, insert_k, insert_i)
        else:
            return list(box_seq), traj, length, solver_time
```

Listing 7: Iterative Planner

3.3 Solving Minimum Distance Problem

1. Define variables.

- x: control variable trajectory.
- l: lower bound.
- u: upper bound.

2. Constraints.
   - $x[0] == start$: initial condition.
   - $x[1:-1] >= l$: lower bound.
   - $x[1:-1] <= u$: upper bound.
   - $x[-1] == goal$: terminal condition.
3. Problem.
   - $cost = cp.sum(cp.norm(x[1:] - x[:-1], 2, axis = 1))$: cost function.
   - $prob = cp.Problem(cp.Minimize(cost), constr)$: problem.
4. Return trajectory, length, solver time.

```python
def solve_min_distance(B, box_seq, start, goal):

    # Define variables.
    x = cp.Variable((len(box_seq) + 1, B.d))

    boxes = [B.boxes[i] for i in box_seq]
    l = np.array([np.maximum(b.l, c.l) for b, c in zip(boxes[:-1], boxes[1:])])
    u = np.array([np.minimum(b.u, c.u) for b, c in zip(boxes[:-1], boxes[1:])])

    # Define constraints.
    cost = cp.sum(cp.norm(x[1:] - x[:-1], 2, axis=1))
    constr = [x[0] == start, x[1:-1] >= l, x[1:-1] <= u, x[-1] == goal]

    # Solve the problem.
    prob = cp.Problem(cp.Minimize(cost), constr)
    prob.solve(solver='CLARABEL')

    length = prob.value
    traj = x.value
    solver_time = prob.solver_stats.solve_time

    return traj, length, solver_time
```

Listing 8: Solving minumun distance problem

4 Smooth Phase

1. Fix box sequence.
2. Cost coefficients.
3. Boundary conditions.
4. Initialize transition times.
5. Optimize bezier with retiming.

```python
def plan(
        S: SafeSet,
        p_init: np.ndarray,
        p_term: np.ndarray,
        T: int,
        alpha: list,
        der_init={},
        der_term={},
        verbose=True
    ):
    # Fix box sequence.
    L = np.array([S.B.boxes[i].l for i in box_seq])
    U = np.array([S.B.boxes[i].u for i in box_seq])
```

```
    # Cost coefficients. (alpha: weight of the cost function)
    alpha = {i + 1: ai for i, ai in enumerate(alpha)}

    # Boundary conditions.
    initial = {0: p_init} | der_init
    final = {0: p_term} | der_term

    # Initialize transition times.
    durations = np.linalg.norm(traj[1:] - traj[:-1], axis=1)
    durations *= T / sum(durations)

    path, sol_stats = optimize_bezier_with_retiming(L, U, durations, alpha, initial, final, verbose=True)
```

Listing 9: Smooth Phase

4.1 Optimize Bezier with Retiming

```
def optimize_bezier_with_retiming(L, U, durations, alpha, initial, final,
    omega=3, kappa_min=1e-2, verbose=False, **kwargs):

    # Solve initial Bezier problem.
    path, sol_stats = optimize_bezier(L, U, durations, alpha, initial, final, **kwargs)
    cost = sol_stats['cost'] # Cost of the trajectory.
    cost_breakdown = sol_stats['cost_breakdown'] # Cost of each Bezier curve.
    retiming_weights = sol_stats['retiming_weights'] # Retiming weights.

    # Lists to populate.
    costs = [cost]
    paths = [path]
    durations_iter = [durations]
    bez_runtimes = [sol_stats['runtime']]
    retiming_runtimes = []

    # Iterate retiming and Bezier.
    kappa = 1 # Trust region.
    n_iters = 0 # Number of iterations.
    i = 1
    while True:
        n_iters += 1

        # Retiming.
        new_durations, runtime, kappa_max = retiming(kappa, cost_breakdown,
            durations, retiming_weights, **kwargs)
        durations_iter.append(new_durations)
        retiming_runtimes.append(runtime)

        # Improve Bezier curves.
        path_new, sol_stats = optimize_bezier(L, U, new_durations,
            alpha, initial, final, **kwargs)
        cost_new = sol_stats['cost']
        costs.append(cost_new)
        paths.append(path_new)
        bez_runtimes.append(sol_stats['runtime'])

        decr = cost_new - cost
        accept = decr < 0

        # If retiming improved the trajectory.
        if accept:
            durations = new_durations
            path = path_new
            cost = cost_new
            cost_breakdown = sol_stats['cost_breakdown']
            retiming_weights = sol_stats['retiming_weights']

        # If retiming did not improve the trajectory then break.
        if kappa < kappa_min:
            break
        kappa = kappa_max / omega
        i += 1

    # Total runtime.
    runtime = sum(bez_runtimes) + sum(retiming_runtimes)
```

```
    # Solution statistics.
    sol_stats = {}
    sol_stats['cost'] = cost
    sol_stats['n_iters'] = n_iters
    sol_stats['costs'] = costs
    sol_stats['paths'] = paths
    sol_stats['durations_iter'] = durations_iter
    sol_stats['bez_runtimes'] = bez_runtimes
    sol_stats['retiming_runtimes'] = retiming_runtimes
    sol_stats['runtime'] = runtime

    return path, sol_stats
```

Listing 10: Optimize Bezier with Retiming

4.2 Optimize Bezier

The functions uses bezier curves to represent trajectory. The problem is formulated as a second-order cone program and solved using the CLARABEL solver.

1. Define variables.
   - $points$: control points of the curves and their derivatives.
   - L: lower bound.
   - U: upper bound.
2. Constraints.
   - $points[0] == initial$: initial condition.
   - $points[-1] == final$: terminal condition.
   - $points[0] >= L$: lower bound.
   - $points[0] <= U$: upper bound.
   - $points[i][1:] - points[i][:-1] == ci * points[i+1]$: Bezier dynamics.
   - $points[k][i][-1] == points[k+1][i][0]$: continuity.
   - $points[k][i][-1] == points[k+1][i][0]$: differentiability.
3. Problem.
   - $cost = cp.sum(cp.norm(p, 2, axis = 1))$: cost function.
   - $prob = cp.Problem(cp.Minimize(cost), constr)$: problem.
4. Reconstruct trajectory.
5. Reconstruct costs.
6. Solution statistics.
7. Return path, solution statistics.

```
def optimize_bezier(L, U, durations, alpha, initial, final,
    n_points=None, **kwargs):

    # Problem size.
    n_boxes, d = L.shape
    D = max(alpha)
    assert max(initial) <= D
    assert max(final) <= D
    if n_points is None:
        n_points = (D + 1) * 2

    # Control points of the curves and their derivatives.
    points = {}
    for k in range(n_boxes):
        points[k] = {}
        for i in range(D + 1):
            size = (n_points - i, d)
```

```
            points[k][i] = cp.Variable(size)

    # Boundary conditions.
    constraints = []
    for i, value in initial.items():
        constraints.append(points[0][i][0] == value) # Initial condition.
    for i, value in final.items():
        constraints.append(points[n_boxes - 1][i][-1] == value) # Terminal condition.

    # Loop through boxes.
    cost = 0
    continuity = {}
    for k in range(n_boxes):
        continuity[k] = {}

        # Box containment.
        Lk = np.array([L[k]] * n_points)
        Uk = np.array([U[k]] * n_points)
        constraints.append(points[k][0] >= Lk) # Lower bound.
        constraints.append(points[k][0] <= Uk) # Upper bound.

        # Bezier dynamics.
        for i in range(D):
            h = n_points - i - 1
            ci = durations[k] / h
            constraints.append(points[k][i][1:] - points[k][i][:-1] == ci * points[k][i + 1]) # Bezier
    dynamics.

        # Continuity and differentiability.
        if k < n_boxes - 1:
            for i in range(D + 1):
                constraints.append(points[k][i][-1] == points[k + 1][i][0]) # Continuity.
                if i > 0:
                    continuity[k][i] = constraints[-1]

        # Cost function.
        for i, ai in alpha.items():
            h = n_points - 1 - i
            A = np.zeros((h + 1, h + 1))
            for m in range(h + 1):
                for n in range(h + 1):
                    A[m, n] = binom(h, m) * binom(h, n) / binom(2 * h, m + n) # Bernstein basis.
            A *= durations[k] / (2 * h + 1)
            A = np.kron(A, np.eye(d))
            p = cp.vec(points[k][i], order='C') # Flatten array.
            cost += ai * cp.quad_form(p, A) # Cost function.

    # Solve problem.
    prob = cp.Problem(cp.Minimize(cost), constraints)
    prob.solve(solver='CLARABEL')

    # Reconstruct trajectory.
    beziers = []
    a = 0
    for k in range(n_boxes):
        b = a + durations[k]
        beziers.append(BezierCurve(points[k][0].value, a, b)) # Control points.
        a = b
    path = CompositeBezierCurve(beziers) # Trajectory.

    retiming_weights = {}
    for k in range(n_boxes - 1):
        retiming_weights[k] = {}
        for i in range(1, D + 1):
            primal = points[k][i][-1].value
            dual = continuity[k][i].dual_value
            retiming_weights[k][i] = primal.dot(dual)

    # Reconstruct costs.
    cost_breakdown = {}
    for k in range(n_boxes):
        cost_breakdown[k] = {}
        bez = beziers[k]
        for i in range(1, D + 1):
            bez = bez.derivative()
            if i in alpha:
```

```
                cost_breakdown[k][i] = alpha[i] * bez.l2_squared() # Cost of each Bezier curve.

    # Solution statistics.
    sol_stats = {}
    sol_stats['cost'] = prob.value
    sol_stats['runtime'] = prob.solver_stats.solve_time
    sol_stats['cost_breakdown'] = cost_breakdown
    sol_stats['retiming_weights'] = retiming_weights

    return path, sol_stats
```

Listing 11: Optimize Bezier

4.3 Retiming

Rescaling the durations to generate new, retimed trajectory.

1. Decision variables.
   - *eta*: scaling factors.
   - *durations*: durations of the original trajectory.
2. Constraints.
   - $durations@eta == sum(durations)$: sum of durations.
   - $eta[1:] - eta[:-1] <= kappa$: trust region.
   - $eta[:-1] - eta[1:] <= kappa$: trust region.
3. Problem.
   - $cost = c * cp.power(eta)$: cost function.
   - $prob = cp.Problem(cp.Minimize(cost), constr)$: problem.
4. Scale costs from previous trajectory.
5. Retiming weights.
6. Trust region.
7. Solve SOCP and get new durations.
8. New candidate for kappa.
9. Return new durations, solver time, kappa max.

```
def retiming(kappa, costs, durations, retiming_weights, **kwargs):

    # Decision variables.
    n_boxes = max(costs) + 1
    eta = cp.Variable(n_boxes)
    eta.value = np.ones(n_boxes)
    constr = [durations @ eta == sum(durations)]

    # Scale costs from previous trajectory.
    cost = 0
    for i, ci in costs.items():
        for j, cij in ci.items():
            cost += cij * cp.power(eta[i], 1 - 2 * j) # Cost function.

    # Retiming weights.
    for k in range(n_boxes - 1):
        for i, w in retiming_weights[k].items():
            cost += i * retiming_weights[k][i] * (eta[k + 1] - eta[k])

    # Trust region.
    if not np.isinf(kappa):
        constr.append(eta[1:] - eta[:-1] <= kappa)
        constr.append(eta[:-1] - eta[1:] <= kappa)

    # Solve SOCP and get new durations.
```

```
    prob = cp.Problem(cp.Minimize(cost), constr)
    prob.solve(solver='CLARABEL')
    new_durations = np.multiply(eta.value, durations) # New durations.

    # New candidate for kappa.
    kappa_max = max(np.abs(eta.value[1:] - eta.value[:-1]))

    return new_durations, prob.solver_stats.solve_time, kappa_max
```

Listing 12: Retiming

5 Experiments

You can find experiments we did in HERE

References

[1] Fast Path Planning Through Large Collections of Safe Boxes