RBE 550 — MOTION PLANNING

Assignment 6: Transmission

Liam Jennings

April 2026

Contents

0.1 Introduction

This project plans a collision-free removal path for the mainshaft of an SM-465 four-wheel-drive manual transmission. The mainshaft must be extracted from the assembled transmission without colliding with the countershaft or the surrounding enclosure.

The planner uses a Python implementation of Bidirectional RRT (BiRRT) [1], growing two trees simultaneously from the assembled and removed poses and connecting them when they come within range of each other. Collision checking is performed using triangular mesh geometry built with trimesh and tested with python-fcl, a Python interface for the Flexible Collision Library (FCL).

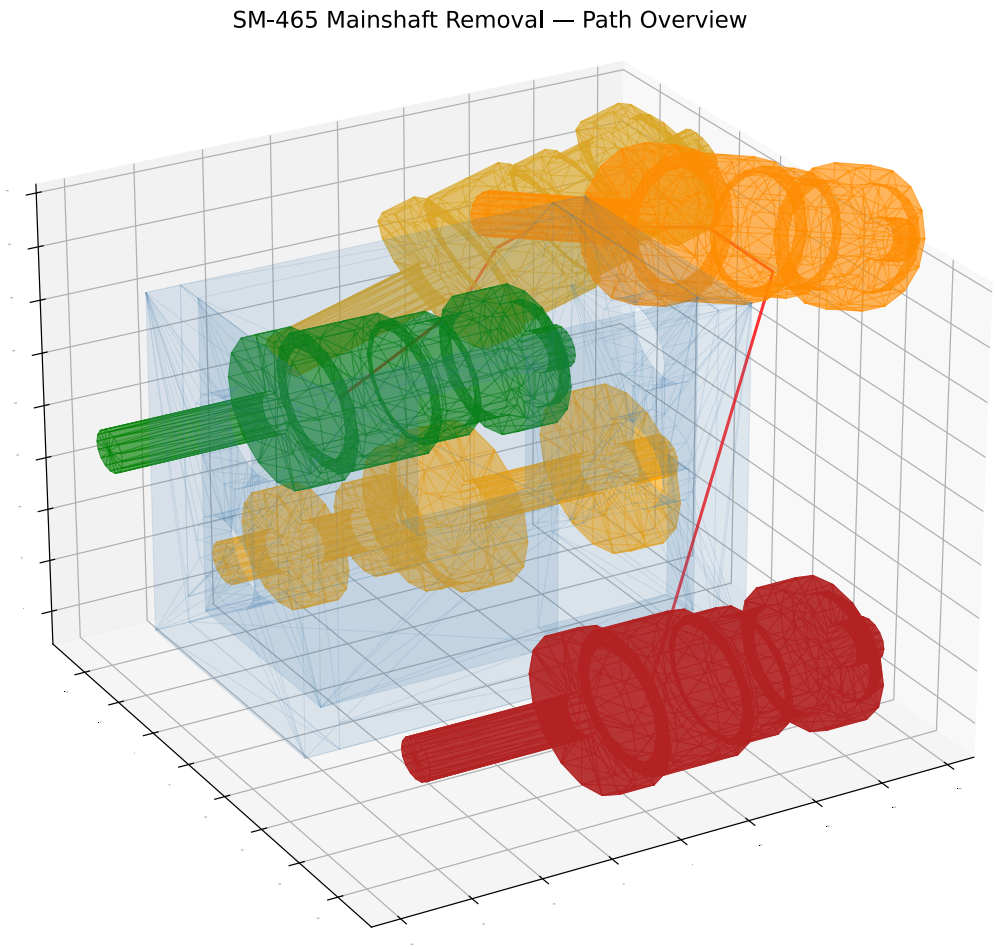

Figure 1: Overview of the planned mainshaft removal path. Green shows the assembled start pose, red the removed goal pose. Intermediate poses at 33% and 66% of the path are shown in gold and orange. The enclosure (blue) and countershaft (orange) are rendered semi-transparently.

0.2 Development Approach

The project was structured around two priorities: correctness first, then performance.

Geometry construction was ported directly from the provided OpenSCAD transmission model, using trimesh's boolean operations to replicate the same component shapes. This made cross-

checking against the known-good visual model straightforward. FCL was chosen for collision detection in favor of a personal implementation to avoid any possible user errors and get a high-performance C++ backend.

The BiRRT planner was implemented closely following the original paper [1]. This proved to be a simple but reliable approach.

Post-processing uses probabilistic shortcutting [2], which repeatedly attempts to replace a random span of the path with a direct collision-free edge, reducing path length without replanning.

0.3 AI/LLM Usage Disclosure

This project was developed with assistance from Claude. Usage included pair programming, code review, refactoring, and research. The overall structure, implementation decisions, and algorithm correctness are the author's own.

0.4 Supplementary Material

Installation instructions and CLI usage are in Appendix A. A full list of dependencies is in Appendix B.

1 Approach

1.1 Shaft Modeling

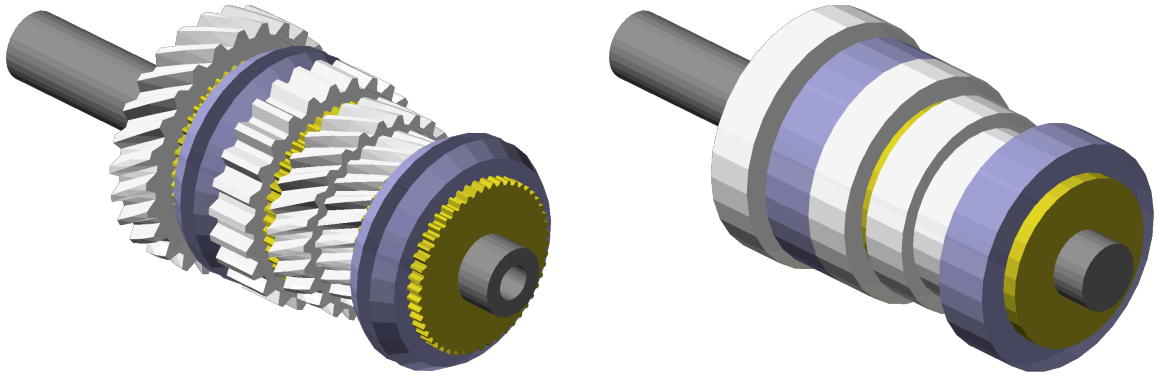

Figure 2: **Left**: the SM-465 mainshaft with full helical gear and synchroniser geometry. **Right**: the simplified planner model. Each component has been replaced by a straight cylinder with its outermost radius.

Rather than modelling the full tooth and synchroniser geometry, the planner represents each component of the mainshaft and countershaft as a straight cylinder with the outermost radius of the real part. Each cylinder is sized to the full radius of the syncro, gear, or collar, so the simplified model fully encapsulates the detailed one. Any path that is collision-free with the simplified mesh is therefore guaranteed to be collision-free with the full-detail helical geometry, giving the simplification a formal correctness guarantee.[1]

This justification enables two simplifications. First, collision checking is cheaper: cylinders are lighter to build and query than detailed gear meshes, and the tooth geometry does not need to be modelled at all[2]. Second, the cylindrical model is rotationally symmetric about the shaft axis, which makes a degree of freedom redundant: any orientation of the shaft is collision-equivalent to one rotated arbitrarily about its own axis. The full 6-DOF rigid-body space ($\mathbb{R}^3 \times S^3$) therefore reduces to $\mathbb{R}^3 \times S^2$, represented with five variables:

$$\boldsymbol{q} = \begin{pmatrix} x \\ y \\ z \\ \theta \\ \varphi \end{pmatrix} \tag{1}$$

where (x, y, z) is the position of the shaft centroid and (θ, φ) are spherical angles describing the orientation of the shaft axis [3].

[1] And one practical consequence: the meshing gear-pair cylinders are in collision in the assembled state. The start pose is therefore raised slightly in Z to clear them, equivalent to the detailed shaft performing an initial upward lift before following the planned path.

[2] See Section 1.2 for implementation details

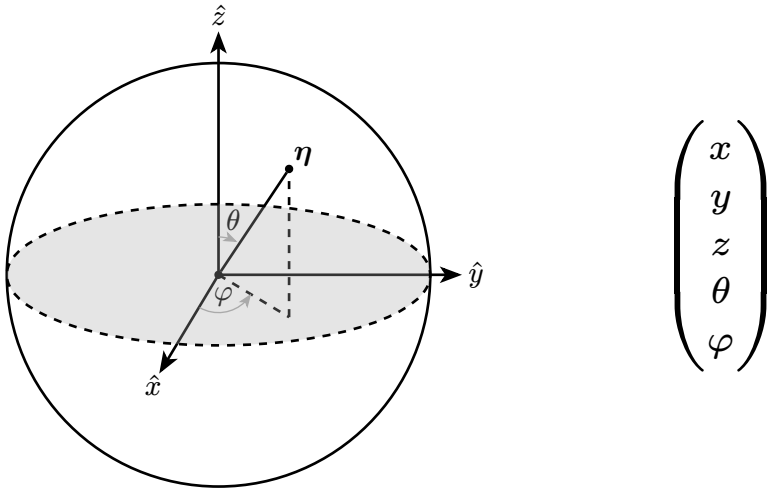

Figure 3: Spherical coordinate representation of shaft orientation.

This is implemented as a `ShaftPose` type alias, a 5-tuple of floats, throughout the codebase.

The spherical parameterisation avoids gimbal lock (which would affect Euler angles) and is simpler than quaternions, at the cost of singularities at $\theta = 0$ and $\theta = \pi$. For this problem the shaft never passes through those orientations, so this is not a practical concern.

An early consideration was whether φ could be dropped entirely, reducing the state to four dimensions (x, y, z, θ). The reasoning was problem-specific: the enclosure has a single opening, and extracting the mainshaft requires pitching it upward to clear the top of the housing, a motion with no need to yaw the shaft axis.

This would have reduced the sampling space and nearest-neighbour search cost. However, removing φ is not strictly correct: while the final path for this problem may not require azimuthal rotation, the planner has no general guarantee of this.

The simplification was deferred pending evidence that performance required it. The planner converged reliably within a few seconds without it, so the full five-dimensional representation was kept.

1.1.1 Rotation Matrix from Spherical Angles

The collision library used, python-fcl (see Appendix B.3) requires a 3×3 rotation matrix to orient the mainshaft mesh. The rotation matrix R is constructed such that the local $+Z$ axis of the shaft frame is mapped to the unit vector described by (θ, φ):

$$\vec{s} = \begin{pmatrix} \sin\theta\cos\varphi \\ \sin\theta\sin\varphi \\ \cos\theta \end{pmatrix} \tag{2}$$

The rotation is computed by composing a rotation about Z by φ, followed by a rotation about Y by θ:

$$R = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3}$$

[3]Finding the single axis-angle rotation that maps $\hat{z}$ to $\vec{s}$ directly

An alternative formulation using Rodrigues' rotation formula[3] [4] was explored but found to be less readable with no performance benefit.

1.1.2 Distance Metric

Sampling and nearest-neighbour queries in the planner require a distance metric over the state space. Position distance is standard Euclidean distance. Orientation distance is the geodesic (great-circle) distance on S^2 [5], computed as the angle between the two shaft axis vectors.

The combined metric weights the orientation component by a scalar w_{rot} so that rotational and translational distances are comparable in magnitude:

$$d(\boldsymbol{q}_1, \boldsymbol{q}_2) = \|\boldsymbol{p}_1 - \boldsymbol{p}_2\|_2 + w_{\text{rot}} \cdot \arccos(\hat{s}_1 \cdot \hat{s}_2) \tag{4}$$

where $\boldsymbol{p}_i = (x_i, y_i, z_i)$ is the position component of $\boldsymbol{q}_i$ and $\hat{s}_i \in S^2$ is its shaft axis unit vector. w_{rot} is set to half the shaft length (165 mm/rad), so a rotation $\Delta\theta$ contributes the arc length traced by the shaft tip rather than an arbitrary tuning constant.

Path segment interpolation uses SLERP[4] [6] for the orientation component, ensuring smooth motion on S^2.

1.2 Collision Checker

With the state space defined, the planner needs a way to determine whether a given pose is valid — i.e., whether the mainshaft at that pose intersects any obstacle. The collision checker provides this query, and is called on every candidate node and path segment during planning.

The collision checker is implemented as a `TransmissionCollisionChecker` class that holds FCL collision objects for the three bodies (mainshaft, countershaft, and enclosure meshes) and tests the transformed mainshaft against both static obstacles for each queried pose.

1.2.1 Geometry Construction

Both shafts and the enclosure are constructed as triangular meshes using trimesh's primitive operations, closely mirroring the OpenSCAD source model. Each component is built as a trimesh primitive and unioned into a single mesh per body.

The full model includes helical gears, but the collision meshes represent each gear as a straight cylinder at the gear's tip radius. Because the cylinder encapsulates the real gear (no part of a helical gear exceeds its tip circle), any path that is collision-free with the cylindrical mesh is guaranteed to be collision-free with the full-detail geometry. The simplification therefore has a formal correctness guarantee rather than relying on empirical tuning.

One practical consequence: the assembled start pose is defined with the shaft raised slightly in Z to ensure the simplified meshes are not immediately in collision. The planner treats this as its initial condition, as it is equivalent to simply raising the detailed model before following the simplified model's path.

The resulting meshes are uploaded to FCL as BVH[5] models once at startup and reused for all subsequent queries.

1.2.2 Pose Queries

For each queried pose, the mainshaft FCL object is transformed to the pose position and orientation, then tested against the fixed countershaft and enclosure objects:

[4] Spherical Linear intERPolation
[5] Bounding Volume Hierarchy

```
function CHECK_COLLISION(pose):
    x, y, z, θ, φ ← pose
    apply rotation_from_spherical(θ, φ) and translation (x,y,z) to mainshaft
    if mainshaft collides with countershaft: return True
    if mainshaft collides with enclosure:    return True
    return False
```

Listing 1: Collision check pseudocode.

Path segment collision checking samples n interpolated poses along the straight-line segment from $\boldsymbol{q}_1$ to $\boldsymbol{q}_2$, returning `False` as soon as any intermediate pose is in collision. The sample count scales with the Euclidean position distance (1 check per 5 mm), floored at a configurable minimum.

1.3 Planner

With a state representation and collision checker in place, the planner searches the configuration space for a sequence of collision-free poses connecting the start and goal.

The planner implements BiRRT as described by Kuffner and LaValle [1]. Two trees grow simultaneously from the start pose and goal pose. On each iteration one tree is extended toward a random sample; the other then attempts to bridge the gap.

1.3.1 Algorithm Structure

```
function BIRRT(start, goal):
    Tree_start.init(start),  Tree_goal.init(goal)
    for k = 1 to K:
        active, target ← T_start, T_goal if k is even else  T_goal, T_start
        q_rand   ← RANDOM_SAMPLE()
        new_node ← EXTEND(active, q_rand)
        if new_node is not None:
            conn_node ← CONNECT(target, new_node.pose)
            if conn_node is not None:
                return ASSEMBLE_PATH(active, target, new_node, conn_node)
    return Failure
```

Listing 2: BiRRT pseudocode, following [1].

K is the maximum iteration limit set by the caller. The two trees alternate on odd/even iterations, avoiding the need to swap tree references and keeping `tree_start` and `tree_goal` stable names throughout.

1.3.2 Extend

`EXTEND` grows the active tree one step toward a target pose. It finds the nearest node in the tree by the combined distance metric, steps toward the target by at most `step_size` (mm) and `step_size_rot` (rad), and adds the resulting pose to the tree if both the pose and the connecting edge are collision-free:

```
function EXTEND(tree, q_target):
    q_near   ← NEAREST(tree, q_target)
    new_pose ← STEP_TOWARD(q_near, q_target)
    if CHECK_COLLISION(new_pose):                return None
    if not CHECK_PATH_SEGMENT(q_near, new_pose): return None
    return tree.add(new_pose, parent=q_near)
```

Listing 3: Extend pseudocode.

1.3.3 Connect

`CONNECT` attempts to bridge the target tree to a given pose. If the nearest target node is already within `connect_radius`, a direct edge is attempted. Otherwise the target tree is extended greedily toward the pose until an obstacle is hit or a connection is made:

```
function CONNECT(tree, from_pose):
    nearest ← NEAREST(tree, from_pose)
    if CAN_CONNECT(nearest.pose, from_pose):
        return nearest
    loop:
        new_node ← EXTEND(tree, from_pose)
        if new_node is None: return None
        if CAN_CONNECT(new_node.pose, from_pose):
            return new_node
```

Listing 4: Connect pseudocode.

`CAN_CONNECT` checks that the two poses are within `connect_radius` and that the direct edge between them is collision-free, scaling the number of edge samples with distance.

1.3.4 Path Assembly and Smoothing

When a connection is found, the path is assembled by tracing each tree back to its root and concatenating: `path_start → ... → new_node → conn_node → ... → path_goal`. The direction of each sub-path depends on which tree was active at connection time.

The raw path is then smoothed by probabilistic shortcutting [2]: two random waypoints are selected, and if the direct edge between them is collision-free it replaces the intermediate segment. After 200 iterations, the waypoint count is typically reduced significantly while preserving collision-freedom.

2 Results

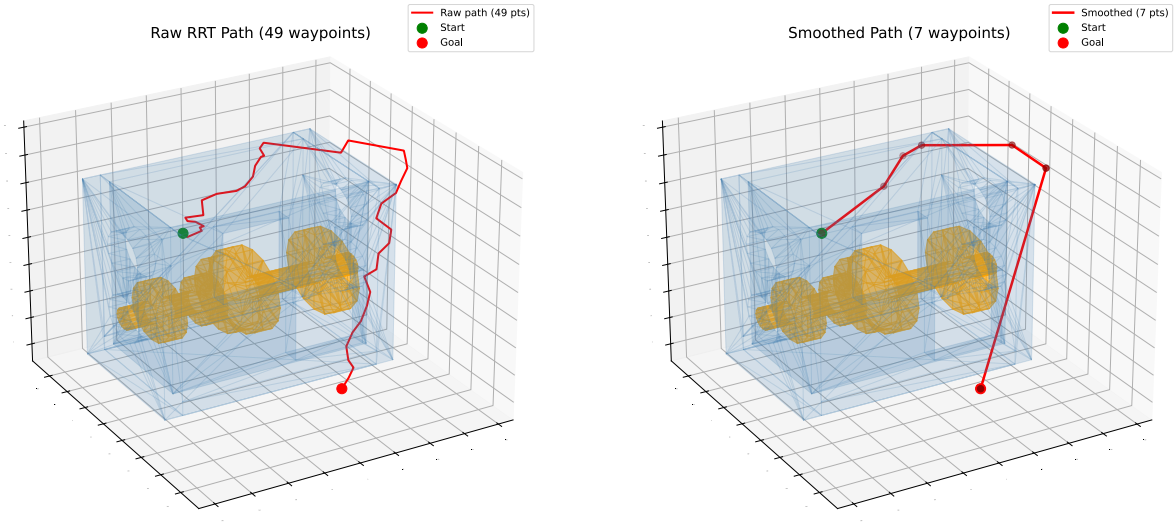

Figure 4: **Left:** Raw BiRRT path. **Right:** Smoothed path after probabilistic shortcutting. The start pose (green) has the mainshaft assembled horizontally inside the enclosure; the goal pose (red) has it extracted and resting upright.

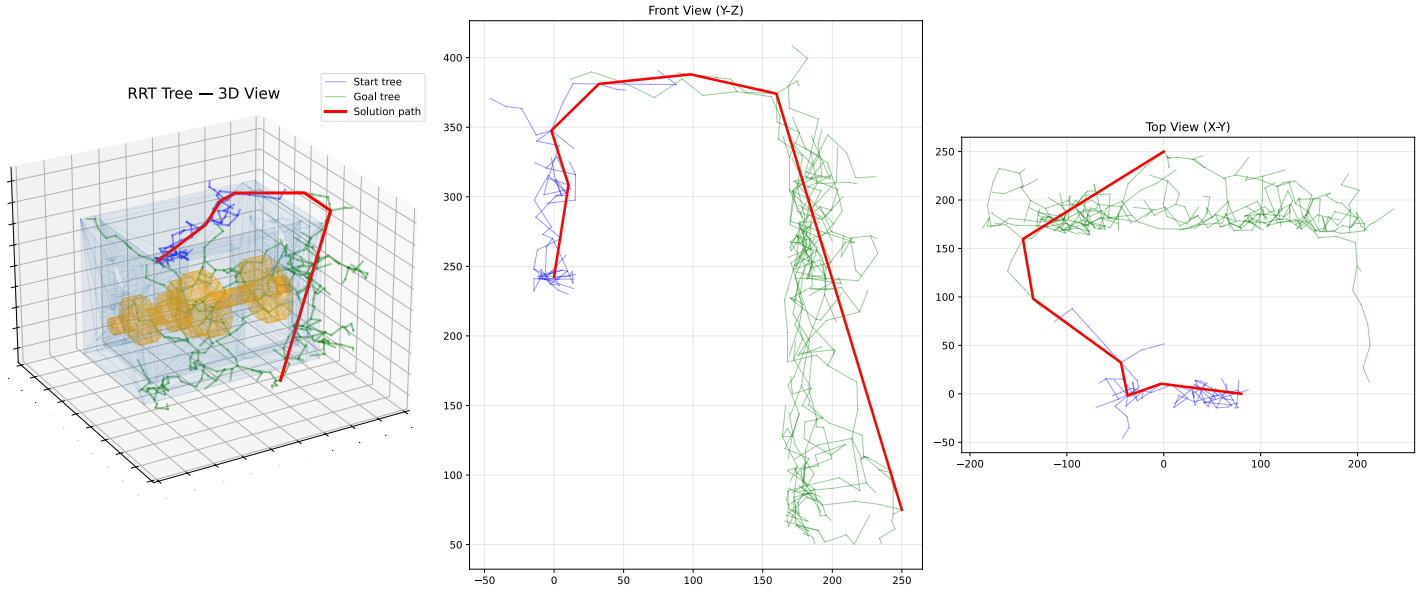

Figure 5: RRT tree structure. **Left:** 3D view of both trees (blue: start tree, green: goal tree) with the solution path in red. **Centre:** front view (Y-Z plane). **Right:** top view (X-Y plane).

The planner reliably finds a solution within a few thousand iterations. The tree structure in Figure 5 shows the geometric constraints: the start tree is much smaller than the goal tree, as its potential to connect to new states is constrained by the enclosure. In the same time, the goal tree grows much larger, enveloping the enclosure until the trees find a path over the top of the enclosure.

The raw path is often indirect, as the search prioritises finding any valid path over path quality. Probabilistic shortcutting consistently reduces the waypoint count and removes unnecessary detours, as visible in the path figure.

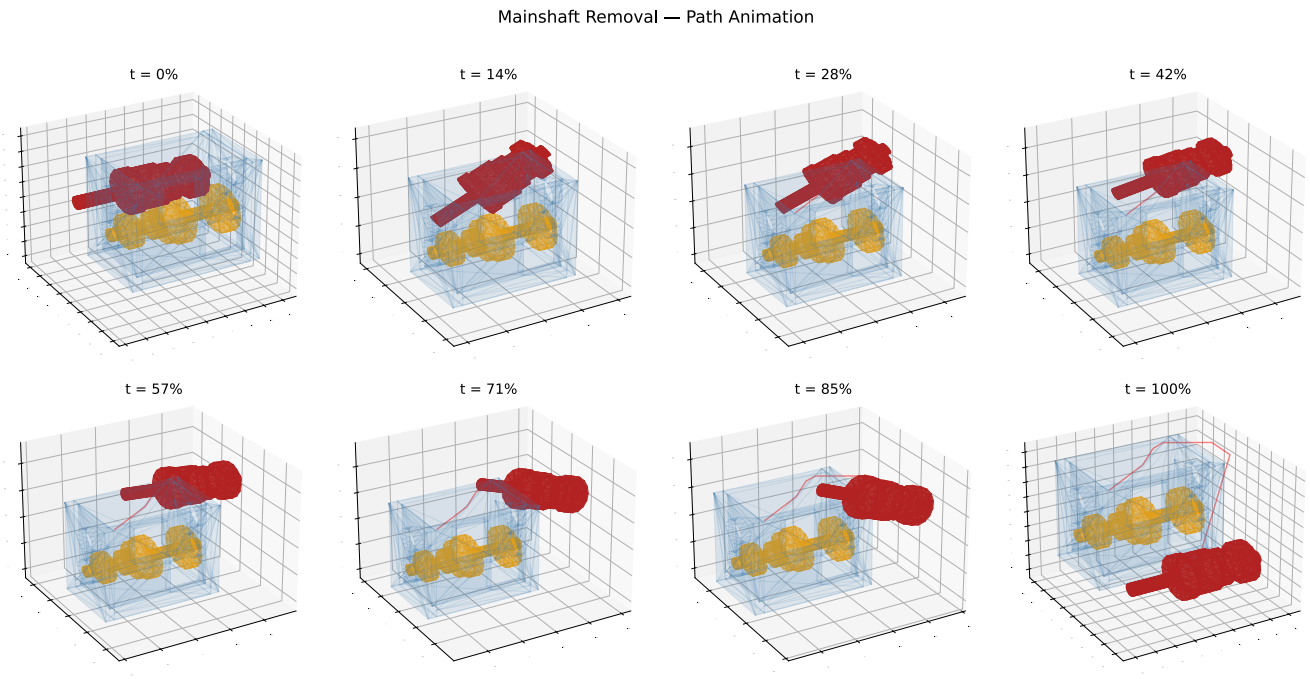

Figure 6: Multi-frame removal sequence at eight evenly-spaced points along the smoothed path. The mainshaft (red) begins horizontal inside the enclosure, tilts and translates to clear the countershaft and enclosure walls, and ends upright outside. The enclosure (blue) and countershaft (orange) are rendered semi-transparently.

The animation frames in Figure 6 show the shaft tilting upward to clear the enclosure opening while avoiding the countershaft, then translating out once clear.

Bibliography

[1] J. J. Kuffner and S. M. LaValle, "RRT-Connect: An Efficient Approach to Single-Query Path Planning," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2000, pp. 995–1001. doi: 10.1109/ROBOT.2000.844730.

[2] R. Geraerts and M. H. Overmars, "Creating High-Quality Paths for Motion Planning," *The International Journal of Robotics Research*, vol. 26, no. 8, pp. 845–863, 2007, doi: 10.1177/0278364907079280.

[3] Wikipedia contributors, "Spherical coordinate system — Conventions." [Online]. Available: https://en.wikipedia.org/wiki/Spherical_coordinate_system#Conventions

[4] O. Rodrigues, "Des lois géométriques qui régissent les déplacements d'un système solide dans l'espace, et de la variation des coordonnées provenant de ces déplacements considérés indépendants des causes qui peuvent les produire," *Journal de Mathématiques Pures et Appliquées*, vol. 5, pp. 380–440, 1840, [Online]. Available: https://www.numdam.org/item/JMPA_1840_1_5__380_0.pdf

[5] Wikipedia contributors, "Great-circle distance." [Online]. Available: https://en.wikipedia.org/wiki/Great-circle_distance

[6] Wikipedia contributors, "Slerp." [Online]. Available: https://en.wikipedia.org/wiki/Spherical_linear_interpolation

Appendices

Appendix A: Setup

A.1 Installation

The project requires Python 3.12 and uses uv for dependency management. See `README.md` for additional installatin instructions.

```
git clone <repo-url>
cd transmission
uv sync
```

A.2 Usage

Run the planner with:

```
uv run treemission [options]
```

Flag	Effect
`--seed N`	Fix the RNG seed for reproducibility (default: random).
`--iterations N`	Maximum BiRRT iterations (default: 30 000).
`--no-viz`	Skip figure generation after planning.
`--out-dir PATH`	Directory for output figures (default: `./out/`).

Appendix B: Dependencies

B.1 trimesh

trimesh provides triangular mesh construction and boolean operations, used to build the transmission geometry.

B.2 manifold3d

manifold3d is the fast mesh boolean backend used by trimesh for union operations during geometry construction. The python package provides bindings for the underlying C++ manifold library.

B.3 python-fcl

python-fcl provides Python bindings for the C++ Flexible Collision Library, giving access to BVH-accelerated mesh-vs-mesh collision queries.

B.4 matplotlib

matplotlib is used for all figure generation: path, tree, animation, and overview plots.

An interactive 3D viewer using pyvista was experimented with during development, but is not included in the final submission.