Supplementary Materials for In-Hand Manipulation Planning for Grippers with Active Surfaces

Supplementary Materials

- **Supplementary Notes 1.** Roadmap construction details.
- **Supplementary Figure 1.** Roadmap graph for different object geometries.
- **Supplementary Notes 2.** Primitive Definition Format
- **Supplementary Movie 1.** Introduction and overview of planning framework.
- **Supplementary Movie 2.** Physical experiments with a variable-friction gripper for the planar case executed without support. For different object geometries, the planned path is visualized, and separate simulations are matched for keyframes of physical execution to show the correspondence between planned path and real world trajectory.
- **Supplementary Movie 3.** Rubik's cube manipulation task. This task highlights a capability uniquely enabled by the combined finger–object abstraction. The goal is not only to reach a desired object orientation, but also to reposition the cube within the grasp so that a target layer protrudes from the fingers and becomes accessible.
- **Supplementary Movie 4.** Physical execution of SE(3) manipulation tasks achieved with addition of extrinsic dexterity (Gravity to slide object down, pushing object bottom to the surface to slide object up, and pushing bottom-front of the object to pivot the object). Planner reasons for available bottom protrusion of the object geometry from the grasp to propose admissible manipulation primitives.
- **Supplementary Movie 5.** Planner instantiated on Belt-Orienting Phalanges (BOP) gripper. Comparison with the baseline shows the planner recovers continuous manipulation trajectories despite discontinuities in object geometry near high-curvature regions and avoids collisions by tracking the object position within the hand.

Supplementary Notes 1: Roadmap Construction Details

This note gives the implementation details behind Algorithm 1: how contact features are found on the object mesh, how those features are linked into an adjacency graph, and how a candidate pair of features is tested for geometric validity.

1.1 Feature extraction

The object mesh is first cleaned (face normals made consistent, and inverted if the mesh volume comes out negative). Contact features are then found directly on this cleaned mesh, not its convex hull, so that a concave feature – e.g. the inward-facing wall of a pocket, such as the web of a T-shaped part – is still extracted as its own candidate feature rather than being discarded as "inside" the hull.

Triangles are grouped into flat facets whenever the angle across their shared edge is below a small threshold (default 1°). Each facet becomes a candidate *face* feature, each boundary between two facets becomes a candidate *edge* feature, and each point where three or more facets meet becomes a candidate *vertex* feature. Every candidate is given a canonical contact normal: a face's own normal, an edge's is the bisector of its two neighbouring face normals, and a vertex's is the normalized sum of all its incident face normals.

A candidate's final label is not taken from this construction directly. Instead, the real mesh vertices that would actually touch a flat finger placed on that candidate's plane are collected, and how far that contact patch spreads in-plane decides the label: wide in two directions is a face, wide in one direction is an edge, and wide in neither is a vertex. This corrects two cases the raw construction gets wrong: a flat plane resting across two edges of a star-shaped object collects both edges and reads as one face rather than two separate edges, and a facet that sits on a smoothly curved region of the object is demoted to an edge, since a finger resting on a curve only ever contacts it along a line, however wide the underlying mesh tessellation happens to be.

1.2 Adjacency construction

Features are linked into a graph that records which features are physical neighbours on the object: every edge feature is linked to the two face features it separates, and to every vertex feature that lies on it. This adjacency graph is what Algorithm 1 consults when deciding whether a finger has moved to a *neighbouring* feature rather than an unrelated one elsewhere on the object.

1.3 Geometric validity test

A candidate pair (f_L, f_R) of one feature per finger is accepted as a node only if both of the following hold:

- **Side test.** Each feature's contact normal must point away from the line joining the two contact points, i.e. neither finger would have to reach through the object to make contact.
- **Opposition test.** The two canonical normals must be anti-parallel to within $2\theta_{\max}$, plus a small fixed slack (default 1°) that accounts for the fact that "opposite" faces on a real, manufactured mesh are never exactly antipodal. $\theta_{\max}$ is split evenly between the two fingers, so each finger contributes at most $\theta_{\max}$ of tilt.

A stricter, optional variant is the antipodal grasp heuristic; this admits fewer nodes but produces grasps with no offset between the contact line and the closing direction. Every feasible pair is retained as a node of the roadmap.

1.4 Edge (transition) construction

Two nodes are connected in the roadmap when, for each finger, the contacted feature either stays the same or moves to an adjacent feature in the graph of Section 1.2, and at least one of the two fingers changes. This single rule covers rolling the object from a face-face grasp to an edge-edge grasp, even though both fingers' features change at once, because each finger's new feature is adjacent to its old one.

This construction only checks feature adjacency and geometric feasibility; it makes no assumption about which manipulation primitive will later realize a given transition. That association is made separately, once primitives are tested against the roadmap in a later stage.

Supplementary Figure 1. Roadmap graph for different object geometries.

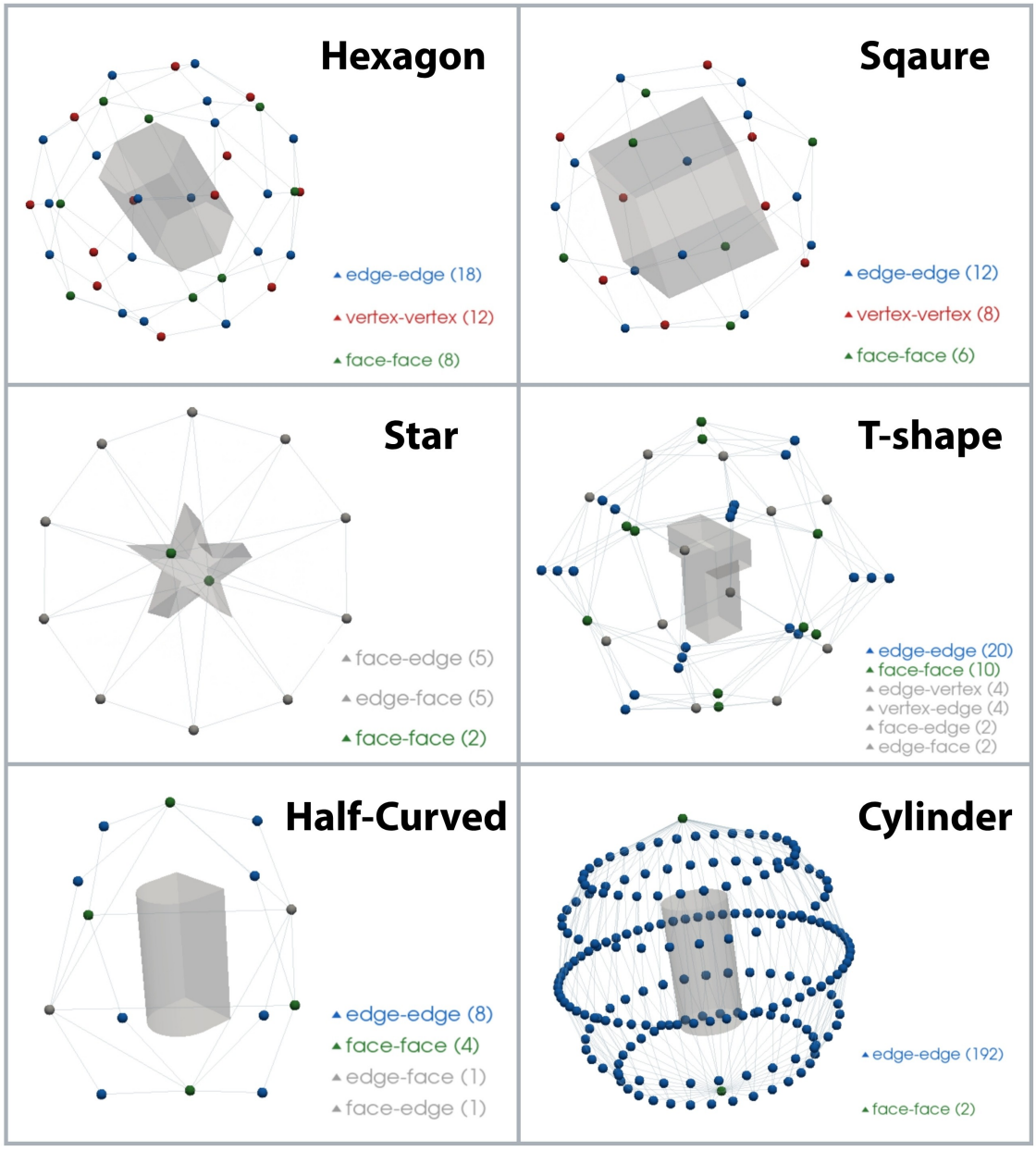

Figure 1: **Roadmap nodes are the contact topologies tuple (τ_1, τ_2) derived from only object geometry, here for a two-fingered gripper.**

Supplementary Notes 2: Primitive Definition Format

Each entry in the primitive library is a single YAML file describing one manipulation primitive as a PDDL-style operator: a name, the parameters it is instantiated over, the preconditions that must hold for it to be admissible, and the effects it has on the planning state. A primitive's YAML file is the only place its behaviour is declared; nothing about the planner, the predicate checks, or the low-level execution routine is hand-written per primitive; adding a new primitive means writing a new YAML file.

2.1 Schema

Every definition file has the following fields.

- **name.** The primitive's identifier, e.g. `Move`, `Roll`, `Slide`.
- **kind.** `preserving` if the primitive keeps the same finger-object contact topology (roadmap node) throughout, or `switching` if it moves the object to an adjacent node. This determines which roadmap edges the primitive can realize: `preserving` primitives move within one manifold, and `switching` primitives traverse an edge of the roadmap built in Supplementary Notes 1.
- **description.** A short, free-text description of what the primitive does.
- **capability.** The gripper mechanism(s) the primitive requires (e.g. `friction_modulation`, `tangential_thrust`). This is checked once per gripper, not per state: a primitive whose required capability is not present in the gripper's active surfaces is dropped from the library entirely when it is instantiated for that gripper, and is never even considered during planning.
- **parameters.** A mapping from parameter name to the list of discrete values it may take, e.g. `direction: [1, -1]`. The primitive is instantiated once per combination of parameter values (the Cartesian product across all listed parameters), so a single definition file with k parameters, each with n_k values, expands into $\prod_k n_k$ concrete primitives – e.g. `Slide`'s `finger` $\times$ `direction` expands into left/$+1$, left/-1, right/$+1$, right/-1.
- **preconditions.** A predicate tree that must evaluate to true for the primitive to be admissible in a given state (Section 2.2).
- **effects.** A declaration of which planning-state fields this primitive is allowed to change, and which it must leave untouched (Section 2.3).

`name` and `kind` are required; a definition file missing any of them, or using an unrecognized `kind`, fails to load with an explicit error rather than being silently accepted.

2.2 Preconditions

A precondition tree is built from a small, closed vocabulary of named predicates combined with three composition rules:

- a bare list, or a dictionary keyed `all`, is a conjunction (every subtree must hold);
- a dictionary keyed `any` is a disjunction (at least one subtree must hold);
- a dictionary keyed by parameter-value, e.g. `direction +1` / `direction -1`, selects one subtree depending on which concrete parameter value this instance of the primitive was given, rather than requiring every branch at once. This is how a single definition file gives its two directions of motion two different admissibility conditions, as in `Move` (Section 2.4).

Predicate leaves fall into two families, checked at two different points in the pipeline:

- *Capability predicates* discussed before.
- *Geometric predicates* ask whether the *current* contact state satisfies some condition – e.g. `has_plane_contact`, `has_line_contact`, `line_contact_aligned` (is a line contact running along a requested axis), `bottom_protrusion` (is the object positioned so an external surface, such as a table, is actually available under it). These are re-evaluated at every candidate state during planning.

A primitive is only expanded into candidate transitions once both gates pass: capability first (gripper-level, checked once), then the geometric precondition tree (state-level, checked per candidate node).

2.3 Effects

The `effects` block declares, for the six fields that make up a planning state (`node`, `d_left`, `d_right`, `d_z`, `orientation`, `hand`), which the primitive is allowed to change and which it must leave exactly as they were. Every one of the six fields must appear in exactly one of the two lists.

`changes` is an upper bound, not a promise that every listed field moves on every call: `Slide`'s `finger` parameter means only one of `d_left`/`d_right` actually changes on a given call, but both appear in `changes` since either can, depending on which finger was selected. `unchanged` is the strict guarantee: after executing the primitive, none of those fields may differ from their value before, under any parameter binding.

2.4 Primitive definitions

```yaml
name: Slide
kind: preserving

description: >
capability: friction_modulation

parameters:
  finger: [left, right]
  direction: [1, -1]
  step_size: [0.5]

preconditions:
  any:
    - has_plane_contact
    - has_line_contact

effects:
  changes: [d_left, d_right, orientation]
  unchanged: [node, d_z, hand]
```

```yaml
name: Rotate
kind: switching

description: >
capability: tangential_thrust, friction_modulation

parameters:
  direction: [1, -1]
  max_hops: [6]
  contact_tol: [0.5]

preconditions:
  any:
    - has_plane_contact
    - has_line_contact

effects:
  changes: [node, d_left, d_right, orientation]
  unchanged: [d_z, hand]
```

```
name: Convey
kind: preserving

description: >
capability: tangential_thrust

parameters:
  direction: [1, -1]
  step_size: [1.0]

preconditions: []

effects:
  changes: [d_left, d_right]
  unchanged: [node, d_z, orientation, hand]
```

```
name: Roll
kind: switching

description: >
capability: tangential_thrust

parameters:
  axis: [world_z]
  direction: [1, -1]
  step_deg: [10.0]

preconditions: []

effects:
  changes: [orientation, node]
  unchanged: [d_left, d_right, d_z, hand]
```