New IK Plugin API for Constraint-Based Solvers

- Design flaws
- Improve support for:
  - Kinematic Trees
  - Redundancy Resolution
  - Tolerances
- Hierarchy of Tasks
Applications of IK

• Find *all* solutions for given eef pose(s) to serve as goal configs

• Compute *closest* solution to seed to get a smooth trajectory obeying Cartesian constraints
Applications of IK

- Find all solutions for given eef pose(s) to serve as goal configs

- Compute closest solution to seed to get a smooth trajectory obeying Cartesian constraints

Both are not well supported by current API
IK approaches

IK solvers

- global
  - deterministic
    - algebraic
      - ikfast
    - geometric
  - randomized

- local
  - gradient-based
    - QP
      - TracIK
  - Jacobian
    - KDL
IK approaches

IK solvers

- global
  - deterministic
  - randomized

- local
  - gradient-based
  - Jacobian

+ Enumerating all solutions
  - Hard to pick a good one

+ Converging to close by solution
  - Get stuck in saddles / singularities
Current IK-plugin API

- `getPositionIK(pose, seed_state, solution)`
  - Find (single) solution, *closest to seed state* for (single) eef
  - only used in `ompl_interface::PoseModelStateSpace`
Current IK-plugin API

- `getPositionIK(pose, seed_state, solution)`
  - Find (single) solution, *closest to seed state* for (single) eef
  - only used in `ompl_interface::PoseModelStateSpace`

- `searchPositionIK(pose, seed_state, timeout, solution)`
  - Same as before, but allow random re-seeding
  - Can return essentially any solution
Current IK-plugin API

- `getPositionIK(pose, seed_state, solution)`
  - Find (single) solution, closest to seed state for (single) eef
  - only used in `ompl_interface::PoseModelStateSpace`

- `searchPositionIK(pose, seed_state, timeout, solution)`
  - Same as before, but allow random re-seeding
  - Can return essentially any solution
  - Variants:
    - `consistency_limits`: allowed per-joint deviations from seed
    - `solution_callback`: validate solutions
    - most generic variant wrapped in `RobotState::setFromIK()`
Current IK-plugin API

- **getPositionIK(pose, seed_state, solution)**
  - Find (single) solution, *closest to seed state* for (single) eef
  - only used in `ompl_interface::PoseModelStateSpace`

- **searchPositionIK(pose, seed_state, timeout, solution)**
  - Same as before, but allow random re-seeding
  - Can return essentially any solution
  - Variants:
    - `consistency_limits`: allowed per-joint deviations from seed
    - `solution_callback`: validate solutions
    - most generic variant wrapped in `RobotState::setFromIK()`

**Simplify: Only keep most generic variant!**
Consistency Limits

- Choosing a proper consistency limit is impossible!
- Moving through singularities results in *strong* changes in joint space
How can we do better?

•Validate *interpolation pose* between joint configs
How can we do better?

- Validate *interpolation pose* between joint configs
  - eef shouldn’t move much
How can we do better?

- Validate *interpolation pose* between joint configs
  - eef shouldn’t move much
  - **Provide utility function in base class to measure „distance“ between configurations**
Current API: Support for Kinematic Trees?

- Compute common solution to place multiple tips
  - `searchPositionIK(poses vector, seed_state, solution)`
  - `getPositionIK(poses vector, seed_state, solutions)`
Current API: Support for Kinematic Trees?

• Compute common solution to place multiple tips
• `searchPositionIK(poses vector, seed_state, solution)`
• `getPositionIK(poses vector, seed_state, solutions)`
  – ambiguity mentioned in src comments:
    • return (multiple?) common solution(s) for given eef poses
    • return a solution for each pose (of a single eef)
    • introduced in Feb 2015 by ROS-I to get *multiple* solutions, but:
Current API: Support for Kinematic Trees?

- Compute common solution to place multiple tips
- `searchPositionIK(poses vector, seed_state, solution)`
- `getPositionIK(poses vector, seed_state, solution)`
  - ambiguity mentioned in src comments:
    - return (multiple?) common solution(s) for given eef poses
    - return a solution for each pose (of a single eef)
    - introduced in Feb 2015 by ROS-I to get multiple solutions, but:
      - Not required! Call IK repeatedly.
Current API: Support for Kinematic Trees?

- Compute common solution to place multiple tips
- `searchPositionIK(poses vector, seed_state, solution)`
- `getPositionIK(poses vector, seed_state, solutions)`
  - ambiguity mentioned in src comments:
    - return (multiple?) common solution(s) for given eef poses
    - return a solution for each pose (of a single eef)
  - introduced in Feb 2015 by ROS-I to get `multiple` solutions, but:
    - method not used in MoveIt code base!

Not required! Call IK repeatedly.
Current API: Support for Kinematic Trees?

- Compute common solution to place multiple tips
- `searchPositionIK(poses vector, seed_state, solution)`
- `getPositionIK(poses vector, seed_state, solutions)`
  - ambiguity mentioned in src comments:
    - return (multiple?) common solution(s) for given eef poses
    - return a solution for each pose (of a single eef)
- introduced in Feb 2015 by ROS-I to get *multiple* solutions, but:
  - method not used in MoveIt code base!

Unify: Provide similar APIs for both functions!
Supporting Redundancies

- 6-DoF robots have discrete set of redundant solutions

- Enumerate them all?
Supporting Redundancies

- 6-DoF robots have discrete set of redundant solutions

- Enumerate them all?
- Interpolating between solutions of different branches results in large joint-space motions
- Usually we want to stay within a single solution branch during planning (to avoid these large-scale motions)
Supporting Redundancies

- 6-DoF robots have discrete set of redundant solutions

- Enumerate them all?
- Interpolating between solutions of different branches results in large joint-space motions
- Usually we want to stay within a single solution branch during planning (to avoid these large-scale motions)

Do we really need to find all solutions?
Supporting Redundancies

- 6-DoF robots have discrete set of redundant solutions

- Redundant robots (#joints > 6) additionally exhibit continuous solution manifolds
  - Finding all solutions not possible
  - Requires discretization
Supporting Redundancies

- 6-DoF robots have discrete set of redundant solutions

- Redundant robots (#joints > 6) additionally exhibit continuous solution manifolds
  - Finding all solutions not possible
  - Requires discretization

- `getPositionIK()` introduced by ROS-I, but not actually used
Redundancy Resolution

• No mechanism provided to resolve redundancies

• Possible Criteria:
  – Keep joints close to a „preferred“ pose / avoid limits
  – Minimize joint velocities = Jacobian Pseudoinverse
  – Minimize kinetic energy
  – Maximize manipulability
  – Minimize joint torques / effort
  – Avoid obstacles, reaching around obstacles
  – Avoid singularities
Redundancy Resolution: Joint Weighting

- Criteria usually compute a scalar cost function that is minimized
- Weighting joint contributions can yield different behaviour
- Example: Joint Velocities

- Provide YAML params for generic distance measure?
KinematicsQueryOptions: further arguments

- `discretization_method, setRedundantJoints()`
  - only relevant for specific IK solvers (e.g. ikfast)
    ➔ **move to kinematics.yaml**

- `search_resolution` (per redundant joint)
KinematicsQueryOptions: further arguments

- **discretization_method**, `setRedundantJoints()`
  - only relevant for specific IK solvers (e.g. ikfast)
  - move to `kinematics.yaml`

- **search_resolution** (per redundant joint)

- **lock_redundant_joints**
  - **discretization_method** = NO_DISCRETIZATION?
  - not used (anymore)
KinematicsQueryOptions: further arguments

- `discretization_method, setRedundantJoints()`
  - only relevant for specific IK solvers (e.g. ikfast)
  - move to kinematics.yaml

- `search_resolution` (per redundant joint)

- `lock_redundant_joints`
  - `discretization_method = NO_DISCRETIZATION`?
  - not used (anymore)

- `return_approximate_solution`
  - used in KDL-based plugins
  - returns any not-converged solution
  - not useful: better introduce explicit tolerances
Explicit Tolerances

- Allow tolerances for all Cartesian directions individually
- Creates additional DoFs in tolerance region
- Facilitates / enables IK for underactuated robots
- Example Grasping
  - position tolerance
  - orientation tolerance
  - infinite tolerance range disables Cartesian axis
  - w.r.t. a specific frame

- Use Constraint messages?
moveit_msgs/Constraints

string name
JointConstraint[] joint_constraints
PositionConstraint[] position_constraints
    std_msgs/Header header
    string link_name
    geometry_msgs/Vector3 target_point_offset
moveit_msgs/BoundingVolume constraint_region
    float64 weight
OrientationConstraint[] orientation_constraints
VisibilityConstraint[] visibility_constraints
Prioritizing Goal Constraints

- Constraint messages allow weighting of tasks
  \[ E(\theta) = w_1 E_1(\theta) + w_2 E_2(\theta) + \cdots + w_n E_n(\theta) \]
- When tasks are conflicting, all fail reaching their goals
Prioritizing Goal Constraints

- Constraint messages allow weighting of tasks
  \[ E(\theta) = w_1 E_1(\theta) + w_2 E_2(\theta) + \cdots + w_n E_n(\theta) \]
- When tasks are conflicting, all fail reaching their goals

- Better: Stack-of-Tasks Approach
  - order tasks by priority
  - optimize subordinate tasks in nullspace of more important ones
  - can be mixed with task weighting to merge tasks on same priority level
Prioritizing Goal Constraints

• Constraint messages allow weighting of tasks
  \[ E(\theta) = w_1 E_1(\theta) + w_2 E_2(\theta) + \cdots w_n E_n(\theta) \]

• When tasks are conflicting, all fail reaching their goals

• Better: Stack-of-Tasks Approach
  - order tasks by priority
  - optimize subordinate tasks in nullspace of more important ones
  - can be mixed with task weighting to merge tasks on same priority level

• How could we extend Constraints messages?
Relative Position Control

• Control left w.r.t. right hand
• Realized by simple Jacobian arithmetics

\[ J = \begin{pmatrix} \text{left} \\ \text{right} \end{pmatrix} - \begin{pmatrix} \text{right} \\ \text{left} \end{pmatrix} \]

• Nullspace control: preferred pose
Summary

• Simplify, unify and clarify IK plugin API
  – `getClosestIK` (const `std::map<string, Constraints>& goals`,
    `const std::vector<double>& seed_state`,
    `std::vector<double>& solution`,
    `KinematicsQueryOptions& options`)
  – `getMultipleIK` (const `std::map<string, Constraints>& goals`,
    `const std::vector<double>& seed_state`,
    `std::list<std::vector<double>>& solutions`,
    `KinematicsQueryOptions& options`)

• Provide corresponding wrappers in RobotState

• Provide generic distance measures
  – Interpolate joint-space configs, measure Cartesian distance
  – Weighted distance from preferred joint-space config
Handling the Migration Process

- New, independent base class
- Provide generic, thin wrapper for existing IK plugins
- Failure on new constraint-based tasks that do not map to old IK API