KLAMP'T MANUAL V0.5

KRIS HAUSER

Document last updated: 11/23/2013

CONTENTS

1. WHAT IS KLAMP'T?

Klamp't (Kris' Locomotion and Manipulation Planning Toolbox) is a cross-platform software package for modeling, simulating, planning, and optimization for complex robots, particularly for manipulation and locomotion tasks. It has been developed at Indiana University since 2009 primarily as a research platform, and beginning in 2013 for it has been used in education.

This manual is meant to give **a high-level roadmap of the library's functionality** and should not be considered a replacement for the detailed API documentation.

1.1. FEATURES

- Supports legged and fixed-based robots.
- Many sampling-based motion planners implemented.
- Fast trajectory optimization routines.
- Real-time motion planning routines.
- Forward and inverse kinematics, forward and inverse dynamics
- Contact mechanics computations (force closure, support polygons, stability of rigid bodies and actuated robots)
- Planning models are fully decoupled from simulation models, which helps simulate uncertainty.
- Robust rigid body simulation with triangle mesh / triangle mesh collisions.
- Simulation of PID controlled, torque controlled, and velocity controlled motors.
- Simulation of various sensors including gyroscopes, force/torque sensors, and accelerometers. (Vision / depth sensors not yet supported)

1.2. CURRENTLY SUPPORTED PLATFORMS

- *nux environments
- Windows via Cygwin
- Windows via MS Visual Studio
- MacOS

Please let us know if you are able to compile on other platforms in order to help us support them in the future.

1.3. COMPARISON TO RELATED PACKAGES

- **ROS (Robot Operating System)** is a large operating system designed for distributed control of physical robots. Although it does come with planning tools they are not as flexible as those in Klamp't. ROS has limited support for legged robots, and is poorly suited for prototyping high-rate feedback control systems. ROS is heavy-weight and not completely cross-platform (only Ubuntu is fully supported).
- **OpenRAVE (Robotics and Animation Virtual Environment)** is similar to Klamp't and was developed concurrently by a similar group at CMU. OpenRAVE has more sophisticated manipulation functionality. Does not support planning for legged robots, but simulation is possible with some effort. Simulation models are often conflated with planning models whereas in Klamp't they are fully decoupled. Heavy-weight.

- **Gazebo, Webots, V-REP, etc** are robot simulation packages built off of the same class of rigid body simulations as Klamp't. They have more sophisticated sensor simulation capabilities, cleaner APIs, and nicer visualizations but are typically built for mobile robots and have little to no functionality for modeling, planning, and optimization. Klamp't also has improved mesh-mesh collision handling that makes collision handling much more stable.

The main drawback of Klamp't is that it is very much still in development and much of the functionality is not fully documented or wrapped in convenient APIs. ROS also has many more wrappers for integration with hardware platforms and sensors.

2. DOWNLOADING AND BUILDING KLAMP'T

Klamp't is publicly available via the git repository at https://github.com/krishauser/Klampt/. The command

```
git clone https://github.com/krishauser/Klampt
```

will download the required files.

You will also need to obtain the following dependencies, which may already be installed on your machine:

- GLUT
- GLPK, the GNU Linear Programming Kit
- Python and SWIG, if you wish to use the Python bindings (tested only on Python 2.6 & 2.7). PyGL is required for visualization.

Building dependencies. First, the dependencies must be downloaded and built. GLUT and GLPK must first be installed in your library paths. KrisLibrary, TinyXML, GLUI, and ODE can be unpacked into the KlamptLibrary folder using the command '`make unpack-deps`'. After configuring the dependencies, they can be built using the command '`make deps`'.

To configure the dependencies, consider the following notes:

- KrisLibrary must be configured for your particular system by editing Makefile.config. See the KrisLibrary readme for more details.
- On Cygwin platforms, KrisLibrary, GLUI, and Klamp't have been tested only using the W32API OpenGL implementation, NOT the X11 one. This requires some changes to some build settings, e.g., in GLUI, setting LIBGL=-lglu32 –lopengl32, LIBGLUT=-lglut32, and CPPFLAGS=-I/usr/include/w32api –DGLUT_DISABLE_ATEXIT_HACK.
- During simulation, ODE will print many warning messages of the form "ODE Message 3: LCP internal error, s <= 0". These can be safely ignored. The output can be made less verbose by commenting out the appropriate lines in ode/src/lcp.cpp (lines 1238 and 1658 in ODE 0.11.1).
- By default, we compile ODE in double floating-point precision. The reason for this is that on some Linux systems, ODE becomes unstable in single floating-point precision and may crash with assertion failures. This may be changed on other systems, if you wish, by toggling ODEDOUBLE=0 or 1 in Klampt/Makefile.config. *Note: if you have already built ODE and then later change its precision, you must do a clean build of ODE as well as the Klampt/Control and Klampt/Simulation folders.*

Building documentation. To build the documentation using Doxygen, type '`make docs`'.

Building static library and apps. The static library is built using '`make lib`'. The main apps to build are RobotTest, SimTest, and RobotPose. Typing '`make [target]`' will build the target.

Building Python bindings. Once the Klamp't static library is built, the Python bindings in Klampt/Python/robot can be built as follows:

- Set the LD_LIBRARY_PATH environment variable to include the locations of the TinyXML and ODE shared libraries, or move the .so files into your shared library path.
- Edit Klampt/Python/robot/setup.py to point to the relevant directories.

- Type '`make`' inside Klampt/robot. '`make docs`' will build the Python API documentation.

3. RUNNING KLAMP'T APPS

RobotTest helps inspect/debug robot files and is run from the command line as follows:

```
./RobotTest robot_file
```

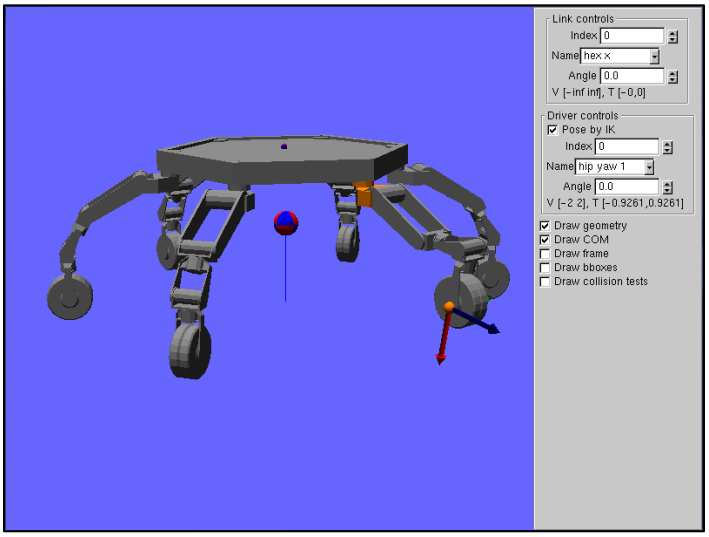


Figure 1. The RobotTest GUI.
`./RobotTest data/robots/athlete.rob`

SimTest performs physics / control simulation and is run from the command line as follows:

```
./SimTest [world, robot, environment, or object files]
```

(e.g., ./SimTest data/robots/athlete.rob data/terrains/plane.env or ./SimTest data/hubo_plane.xml)

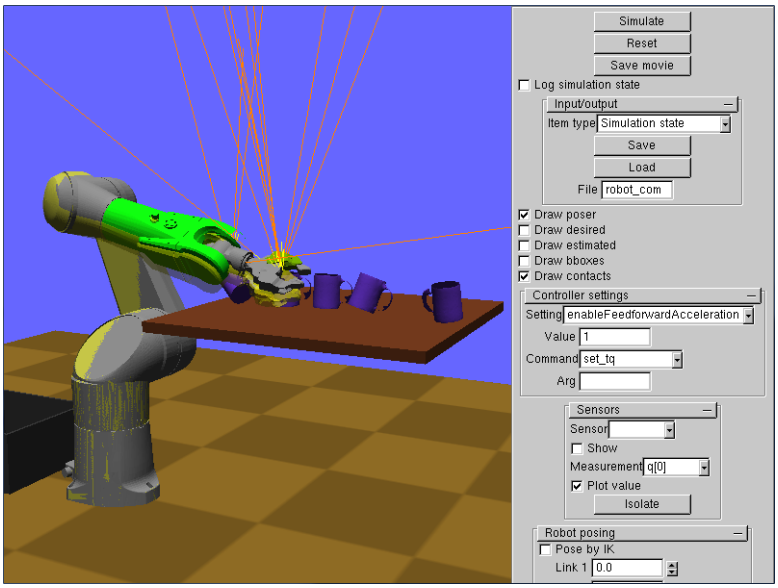


Figure 2. The SimTest GUI. The transparent yellow robot is the "poser".
`./SimTest data/tx90cups.xml`

RobotPose helps a human designer create configurations, constraints, and motions, and is run similarly to SimTest.

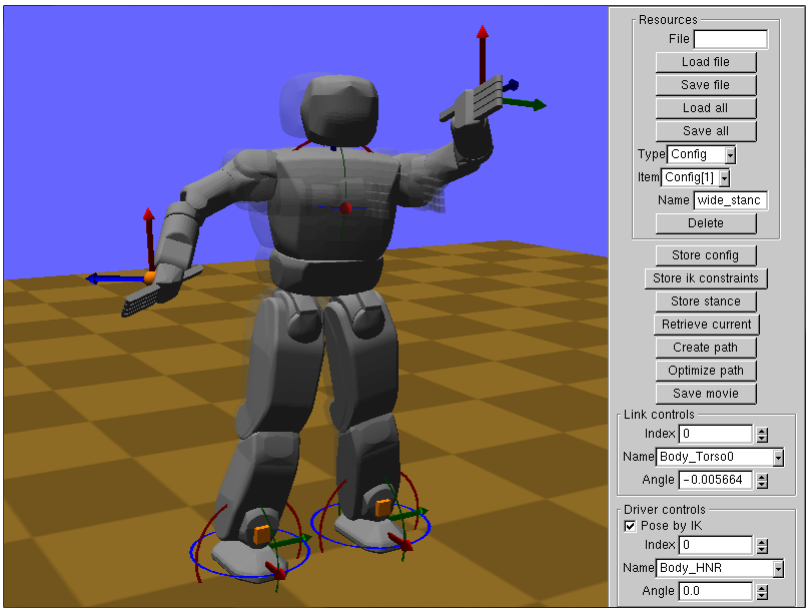


Figure 3. The RobotPose GUI. The 3D coordinate frames are "widgets" for posing links of the robot in Cartesian space.
`./RobotPose data/hubo_plane.xml`

3.1. INTERACTING WITH 3D WORLDS

Each of the above apps follows a common camera navigation and robot posing interface.

Navigating

- Dragging with the left mouse button (left-drag) rotates the camera about a focal point.
- Alt+left-drag zooms the camera.
- Ctrl+left-drag pans the camera.
- Shift+left-drag moves the camera toward and away from the focal point.

Posing robots

- Right-clicking on a robot link and dragging up and down will set its desired joint value.
- The floating base of a robot is posed by right-dragging on the widget.
- *IK posing*
    - To switch to IK-posing mode, check the "Pose by IK" button.
    - In this mode, clicking on a point on the robot will add a new IK point constraint.
    - The widget can be right-dragged to move the robot around.
    - Typing 'c' while hovering over a link will add a new fixed position and rotation constraint.

- Typing ‘d’ deletes an IK constraint.

RobotTest commands

- ‘h’ prints the full help.
- ‘p’ prints the posed configuration to the console.

SimTest commands

- *Command line options*
  - `-config [.config file]` loads a robot start configuration from disk. If more than one robot exist in the world file, multiple `-config` options may be specified to give their start configurations.
  - `-milestones [.milestone file]` loads a milestone path from disk.
  - `-path [.xml or .path file]` loads a MultiPath or piecewise linear trajectory from disk.
- ‘h’ prints the full help.
- Typing ‘ ‘ (space bar) or clicking the “Set Milestone” button will send the posed configuration to the controller.
- Typing ‘s’ or clicking the “Simulate” button toggles the simulation.
- Typing ‘a’ advances by one simulation step (1/100 s).
- Clicking “Save movie” will tell the simulator to start saving 640x480 frames to PPM files on disk at 30fps. These can be converted into a simulation-time (i.e., 1s of movie time = 1s of simulated time) movie using a utility such as ffmpeg. The movie resolution can be changed by setting the `movieWidth` and `movieHeight` parameters in simtest.settings (JSON format).
- Typing ‘f’ toggles force application mode. In force application mode, right-clicking and dragging on the robot will apply a spring-like force between the robot and the cursor position.
- Typing ‘v’ (lowercase) saves the current viewport to disk, and ‘V’ (uppercase) loads the previously saved viewport. This is useful for creating side-by-side comparisons.

Note: when simulating a path, Klamp’t will only issue a “discontinuous jump requested” warning if the path does not start from the robot’s current configuration. If you wish to initialize the robot with the start of the path, either copy the start configuration into the world file, or provide the `-config [file]` command line argument. To easily extract a start configuration from a MultiPath file, use the script “`python Python/multipath.py -s [path.xml] > temp.config`”.

RobotPose commands

- *Command line options*
  - `-l [resource_library directory or XML file]` loads a resource library from disk. Multiple libraries can be loaded in this way.
- Individual resources or resource libraries may be loaded from disk via the controls at the top.
- “Library -> Poser” sets the poser to use the currently selected configuration, stance, hold, or grasp from the resource library.
- “Poser -> Library” stores the current posed configuration, stance, or hold to the resource library. Selection is accomplished via the “Resource Type” selector.
- “Library Convert” converts the currently selected resource into a resource of the specified type in the “Resource Type” selector.

- "Create Path" generates an interpolating path and saves it to the resource library. If the currently selected resource is a Config type, it interpolates from the poser's current configuration to the resource. If a Configs resource is selected, then it interpolates amongst the configurations in the file.
- "Optimize Path" generates and optimizes a trajectory along the currently selected resource, minimizing execution time under the robot's velocity and acceleration bounds. This works when Configs, Linear Path, or MultiPath resources are selected.

Note: path editing is not particularly sophisticated at the moment due to the limitations of GLUI. The best way of generating a sophisticated path inside RobotPose is to generate keyframes into a Configs resource, and choose "Create Path" or "Optimize Path".

3.2. EXAMPLE FILES

World files for different robots are available in the Klampt/data subdirectory:

- hubo*.xml: the KAIST Hubo humanoid.
- puma*.xml: the Puma 760 industrial robot.
- tx90*.xml: the Staubli TX90L industrial robot.

Other test robots, objects, and environments are available in the Klampt/data/{robots,objects,terrains} subdirectories. Some files of interest may include:

- athlete.rob: the NASA ATHLETE hexapod (incomplete, missing wheel geometry).
- cartpole.rob: the cart-pole balancing control problem.
- footed_2d_biped.rob: a simple 2D biped mimicking a human's forward motion.
- footed_2d_monoped.rob: a simple 2D monoped.
- hrp2.rob: the AIST HRP-2 humanoid
- pr2.rob: the Willow Garage PR2 robot (requires KrisLibrary to be built with Assimp support)
- robonaut2.rob: the NASA Robonaut2 humanoid torso.
- robotiQ_3finger.rob: the RobotiQ 3-finger Adaptive Gripper.
- simple_2d_biped.rob: a simple 2D biped mimicking a human's lateral motion.
- swingup.rob: a simple pendulum swingup control problem.
- plane.env: a flat plane environment
- block.obj: a 40cm block
- block_small.obj: an 8cm block

Test motions are available in the Klampt/data/motions directory. Simulation examples can be run via:

- ./SimTest data/robots/athlete.rob data/terrains/plane.env –start data/motions/athlete_start.config –path data/motions/athlete_flex.xml
- ./SimTest data/hubo_table.xml –path data/motions/hubo_table_path_opt.xml
- ./SimTest data/hubo_stair_rail.xml –path data/motions/hubo_stair_rail_traj.xml

3.3. OTHER KLAMP'T APPS

Klamp't also comes with the following utility apps:

- URDFtoRob produces a Klamp't .rob file from a Unified Robot Description Format (URDF) file. The .rob file must be edited by hand or with the MotorCalibrate program to fix up the dynamic parameters (mostly servo gains). Settings for geometry import/export can be changed by editing urdftorob.settings.
- MotorCalibrate generates motor simulation parameters given example commanded and sensed trajectories. First, run it without arguments to generate a blank motorcalibrate.settings file. Edit the parameters to set the robot, driver indices to estimate, whether any links are rigidly fixed in space, and the commanded / sensed path files (in Linear Path format). Then run it again with the settings file as input, and it will output the optimized parameters to the console. These lines should be copied into the .rob file.
- Unpack expands a composite resource into a hierarchical directory structure containing its components. These components can be individually edited and then re-combined into the resource using Pack.
- Pack is the reverse of Unpack, taking a hierarchical directory structure and combining it into a composite resource of the appropriate type.
- UserTrials is a demonstration of Klampt's real-time planning capabilities. A similar program was used for the user studies in E. You and K. Hauser. *Assisted Teleoperation Strategies for Aggressively Controlling a Robot Arm with 2D Input*. In proceedings of Robotics: Science and Systems (RSS), Los Angeles, USA, June 2011.
- PosMeasure outputs the translations and orientations of a robot's links as it follows a trajectory.
- SimUtil is a command line interface to the simulator.

4. DESIGN PHILOSOPHY

The main philosophy behind the Klamp't design is to decouple Modeling, Planning, Control, and Simulation modules. This division provides a clear logical structure for developing large software systems for operating complex intelligent robots.

- *Modeling* refers to the underlying knowledge representation available to the robot, e.g., limb lengths, physical parameters, environment, and other objects in its vicinity. The Modeling module contains methods for representing this knowledge. It also includes the ubiquitous mathematical models, such as kinematics and dynamics, trajectory representations (e.g., splines), and contact mechanics that required for planning and control.
- *Planning* refers to the computation of paths, trajectories, feedback control strategies, configurations, or contact points for a robot. Planning may be performed either offline or online.
- *Control* refers to the high-rate processing of sensor information into low-level robot controls (e.g., motor commands). This also includes state estimation. Note that the boundary between planning and control is fuzzy, because a fast planner can be used as a controller, or a planner can compute a feedback control strategy.
- *Simulation* refers to a physical simulation of a virtual world that is meant *as a stand-in for the real world and robot*. The simulation module constructs a detailed physical rigid-body simulation and instantiates a controller and virtual sensors for a simulated robot. The controller then applies actuator commands that apply forces in the simulation.
- Auxiliary modules include *Visualization,* referring to the display of a simulated or animated robot and its environment and *I/O*, referring to the serialization and management of resources.

Planning, control, and simulation are related by the use of (largely) common models. However, the simulation model does not need to be the same as the planner or controller's model. For example, an object's position may be imperfectly sensed, or a free-floating robot like a humanoid may not know precisely where its torso lies in 3D space. Also, for computational practicality a planner might work on a simplified model of the robot (e.g., ignoring the arms during biped walking) while the controller must expand that information into the full robot representation.

Klamp't uses a concept model that is language-independent, which is implemented using language-specific APIs. Since its most complete implementation is in C++, the following sections will discuss the concept model and the C++ API together.

C++ API file structure.

- **Modeling**: Klampt/{Modeling, Contact}/, which depends heavily on KrisLibrary/robotics for basic robot kinematics and dynamics, and KrisLibrary/{math3d, geometry, meshing} for 3-D geometry
- **Planning**: Klampt/Planning/, which depends heavily on KrisLibrary/{planning, optimization}/
- **Control**: Klampt/Control/
- **Simulation**: Klampt/Simulation/
- **Visualization:** Klampt/View/
- **I/O:** native I/O is mostly embedded into models. Import/export to XML world files and external formats are found in Klampt/IO.

Python API file structure.

The Klamp't Python API is primarily given in the klampt module found in Klampt/Python. This module contains functionality in its sub-modules for modeling, simulation, planning, and visualization. Control is handled in a separate module.

- Klampt/Python/klampt: the main Klamp't module.
- Klampt/Python/control: the control module.
- Klampt/Python/demos: demonstrations about how to use and visualize the klampt module.
- Klampt/Python/exercises: exercises for implementing basic concepts in Klamp't.
- Klampt/Python/utils: utility programs.

5. MODELING

5.1. MATH

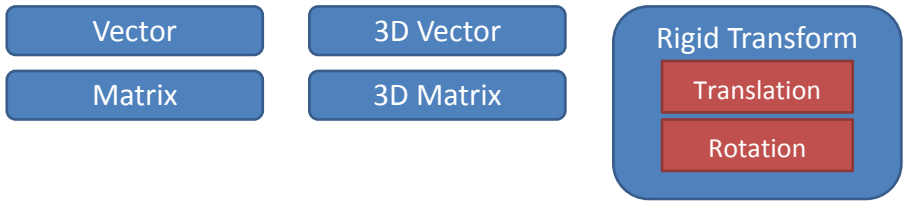


Figure 4. The Math concept model.

Klamp't assumes basic familiarity with 3D geometry and linear algebra concepts. It heavily uses structures that representing vectors, matrices, 3D points, 3D rotations, and 3D transformations. These routines are heavily tested and fast.

Most users will be satisfied with definitions in the following files:

- KrisLibrary/math/math.h contains definitions for basic mathematical routines. `Real` is typedef'ed to `double` and (probably) should not be changed.
- KrisLibrary/math/vector.h contains a `Vector` class (typedef'ed to `VectorTemplate<Real>`).
- KrisLibrary/math/matrix.h contains a `Matrix` class (typedef'ed to `MatrixTemplate<Real>`).
- KrisLibrary/math/angle.h contains functions for interpolating and measuring distances of angles on SO(2).
- KrisLibrary/math3d/primitives.h contains 2D and 3D mathematical primitives. The classes `Vector2`, `Vector3`, `Matrix2`, `Matrix3`, `Matrix4`, `RigidTransform2D` and `RigidTransform` are efficient implementations of 2D and 3D vector/matrix operations.
- KrisLibrary/math3d/rotation contains several representations of rigid 3D rotations, including euler angles, moments (aka exponential maps), angle-axis form, and quaternions. All representations can be transformed into one another. All routines are implemented to be numerically robust.

The `Vector`, `Vector3`, and `RigidTransform` classes are the most widely used math classes in Klamp't. Vectors accept all the basic arithmetic operations as well as dot products, norms, and distances. Applying a transformation (Matrix3 or RigidTransform) to a point (Vector3) is expressed using the * operator.

5.2. 3-D GEOMETRY

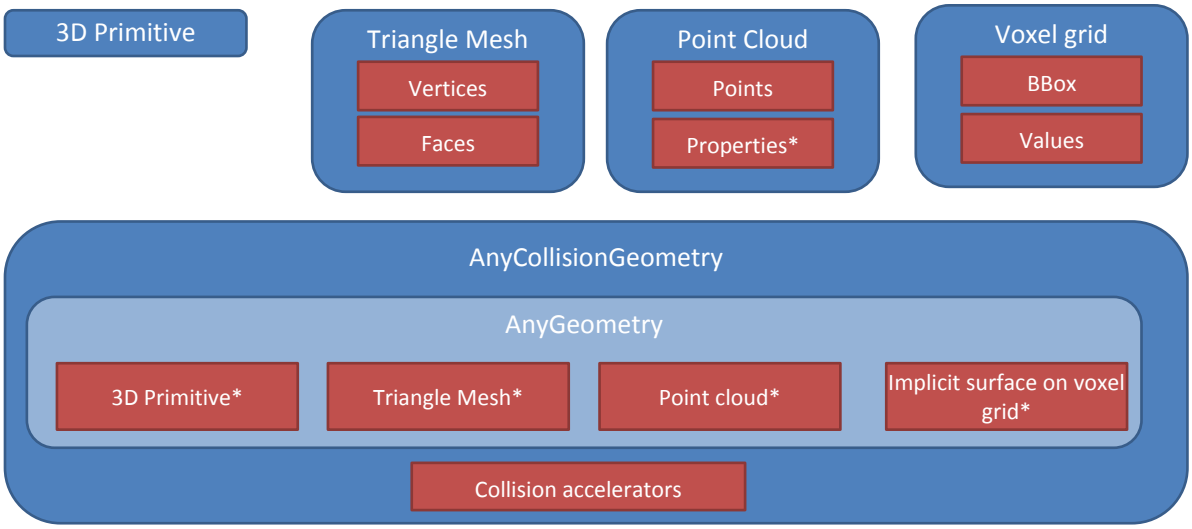


Figure 5. The 3D geometry concept model.

Klamp't uses a variety of geometry types to define geometric primitives, triangulated meshes, and point clouds.

Geometry data, collision geometries. Geometry data is stored in the `AnyGeometry3D` type, which encapsulates primitives, triangle meshes, and point clouds. (It also experimentally supports implicit surfaces defined on a voxel grid, but the implementation is highly incomplete at the moment.) This data is stored in an object's local frame.

The notion of a *collision geometry* combines some underlying geometric data with transformations and collision acceleration structures These are stored in the `AnyCollisionGeometry3D` type. Collision geometries also support an additional, nonnegative `margin` setting that "expands" the underlying geometry when performing collision testing. The margin does not actually affect the geometric data, but rather it changes the distance threshold that is used to consider colliding vs. noncolliding geometries.

Geometric operation support. Triangle mesh support is complete, optimized, and well-tested throughout Klamp't, but the other geometries types are not yet fully supported by all modules.

- *Drawing*: All types supported.
- *Collision detection in planning*. All types supported. Note: Point cloud collision detection is currently inefficient for large point clouds.
- *Tolerance verification*. All types supported. Note: Point cloud collision detection is currently inefficient for large point clouds.
- *Distance detection in planning*. Not supported at the moment, but primitive/primitive and triangle mesh/triangle mesh distance functions are available.
- *Ray casting*. Triangle meshes, point clouds.
- *Contact detection in simulation*. Triangle mesh / triangle mesh and triangle mesh / point cloud only.

File formats. Geometries can be loaded from a variety of file formats. The native triangle mesh format is .tri, which is a simple list of vertices followed by a list of indexed triangles. If Klamp't is compiled with Assimp support, it can also load a variety of other formats including STL, OBJ, VMRL, etc. Point clouds can be loaded from PCD files (v0.7), as specified by the Point Cloud Library (PCL).

C++ Implementation. Detailed documentation can be found in the following files:

- KrisLibrary/math3d/geometry3d.h defines 3D geometric primitives, including `Point3D`, `Segment3D`, `Triangle3D`, `AABB3D`, `Box3D`, `Sphere3D`, and `Ellipsoid3D`. There is also a `GeometricPrimitive3D` class that abstracts common operations on any geometric primitive.
- KrisLibrary/meshing/TriMesh.h defines 3D triangle meshes.
- KrisLibrary/meshing/PointCloud.h defines a 3D point cloud. Each point may contain a variety of other named properties, including color, normal, id, etc.
- KrisLibrary/geometry/CollisionMesh.h contains the `CollisionMesh` and `CollisionMeshQuery` data structures. `CollisionMesh` overloads the `Meshing::TriMeshWithTopology` class and represents a preprocessed triangle mesh for collision detection. It can be rigidly transformed arbitrarily in space for making fast collision queries via the `CollisionMeshQuery` class and the `Collide`/`Distances`/`WithinDistance` functions. Mesh-mesh proximity testing (collision and distance computation) are handled by the open source PQP library developed by UNC Chapel Hill. These routines are heavily tested and fast.
- KrisLibrary/geometry/AnyGeometry.h defines the `AnyGeometry3D`, `AnyCollisionGeometry3D`, and `AnyCollisionQuery` classes. It is recommended to use these classes for geometric operations because new geometry types will be supported through them.

5.3. ROBOTS

Klamp't works with arbitrary tree-structured articulated robots. Robots provide the following functionality

- Describes a topologically sorted open linkage as a list of links with their parents.
- Stores kinematic characteristics: link lengths, joint axis types, joint stops, inertial characteristics, and link geometry.
- Stores actuation limits
- Stores a "current" robot configuration and velocity. *Note: these should be thought of as temporary variables, see notes below.*
- Computes and stores the robot's "current" link frames via forward kinematics.
- Computes the robot's Lagrangian dynamics terms.
- Stores link collision geometries and performs collision detection.
- Stores information about which links can self-collide.
- Names each link and contains semantics of the how the degrees of freedom of the robot map to "joints" and actuators.
- Loads and saves robot descriptions from disk.

File formats. Robots are loaded from .rob files. These are simple text files that are editable by hand.

The URDFtoRob program converts from the widely-used URDF file format to .rob. Geometric primitive link geometries will be converted to triangle meshes. For simulation purposes, Klamp't will need some motor parameters to be tweaked (`servoP`, `servoI`, `servoD`, `dryFriction`, `viscousFriction`). This can be done by hand by tuning and "exercising" the robot in simulation. An automatic solution is given by the MotorCalibrate program, which will optimize the constants to match a dataset of sensed and commanded joint angles that you record while exercising the physical robot. See Section 3.3 for more details about this program.

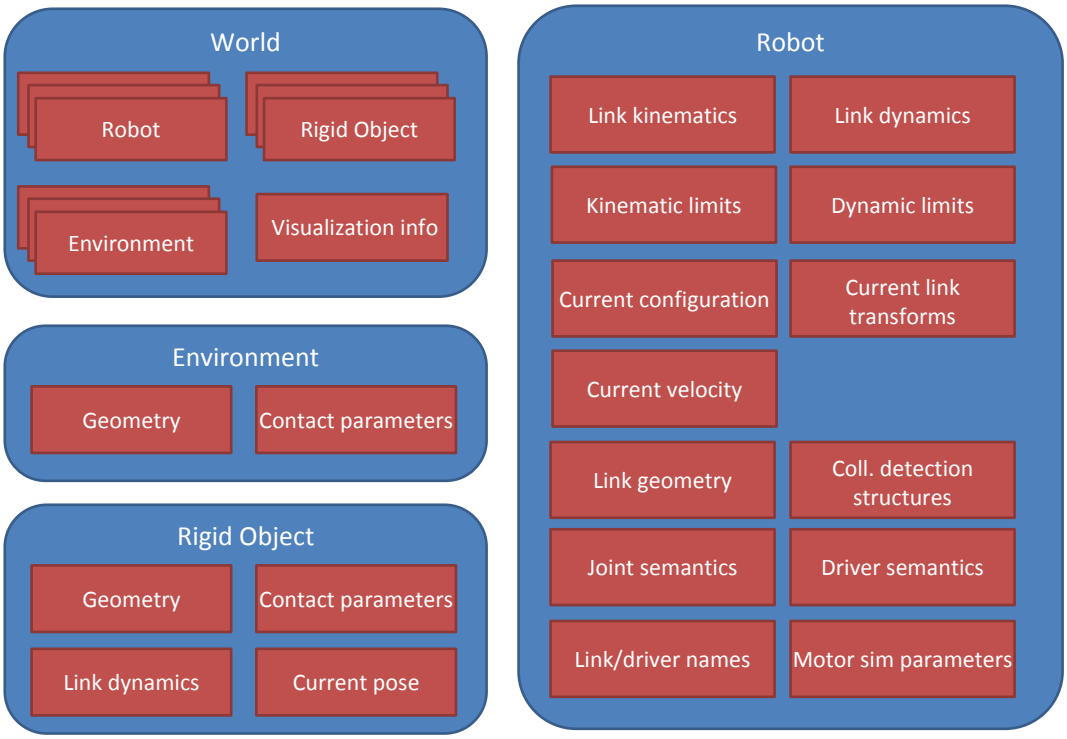


Figure 6. The Robot, Environment, Rigid Object, and World concept models.

Configurations. A robot configuration is a nonredundant description of the positions of each link of the robot, and is described by a `Config` object. The `Robot.q` member represents a "current" configuration. Note that `Robot.q` *is not the currently simulated robot configuration*, but is rather a temporary variable.

Important: to ensure consistency between the configuration and the link frames, the `Robot.UpdateConfig(q)` method should be called to change the robot's configuration. `UpdateConfig` performs forward kinematics to compute the link frames, while `Robot.q=q` does not.

Links. Links represent rigid coordinate frames that are connected to either another link or the world coordinate frame via a movable *degree of freedom* (DOF). The data for each link in the robot is stored in the `links` member, which is a list of `RobotLink3D`'s. The parent index of each link is stored in the `parents` member, which is a list of `integers`. A parent of -1 indicates that the link is attached to the world coordinate frame. Links may be prismatic or revolute and moves along or around the axis `w`. They also contain mass parameters (`mass`, `inertia`, `com`), the reference transformation to its parent (`T0_Parent`), and the link's "current" world transformation `T_World`.

Virtual links. To represent free-floating bases, one should use a set of 5 massless *virtual links* and 1 physical link that represent the x, y, and z translations and rotations around the z, y, and x axes (roll-pitch-yaw convention). See `RobotKinematics3D.InitializeRigidObject` for an example of how to set up such a base. Likewise, a mobile robot may be represented by 2 virtual links + 1 physical link: two for x, y translations connected by prismatic joints, and the last for θ, connected to its parent by a revolute joint. A ball-and-socket joint may be represented by 2 virtual links + 1 physical link.

Geometry. Each link's collision geometry is stored in the `geometry` list. A link's geometry may also be empty. *Note: in the C++ API the collision geometry is updated to the current link transforms after robot.UpdateGeometry() is called*.

Joints. The DOFs of a robot are considered as generic variables that define the extents of the articulations between links. At the `Robot` level, Klamp't introduces the notion of *Joints*, which introduce a notion of *semantics* to groups of DOFs. Most Joints will be of the `Normal` type, which map directly to a single DOF in the normal way. However, free-floating bases and other special types of Joints designate groups of DOFs that should be interpreted in special ways. These special Joints include:

- `Weld` joints, which indicate that a DOF should not move.
- `Spin` joints, which are able to rotate freely and infinitely.
- `Floating` joints, which translate and rotate freely in 3D (e.g., free-floating bases)
- `FloatingPlanar` joints, which translate and rotate freely in 2D (e.g., mobile wheeled bases)
- `BallAndSocket` joints, which rotate freely in 3D.
- `Closed` joints, which indicate a closed kinematic loop. *Note: this is simply a placeholder for potential future capabilities; these are not yet handled in Klamp't.*

Drivers. Although many robots are driven by motors that transmit torques directly to single DOFs, the Robot class can represent other drive systems that apply forces to multiple DOFs. For example, a cable-driven finger may have a single cable actuating three links, a mobile base may only be able to move forward and turn, and a satellite may have thrusters. Free-floating bases may have no drive systems whatsoever.

A robot is set up with a list of Drivers available to produce its torques. `Normal` drivers act as one would expect a motor that drives a single DOF to behave. Connected transmissions (such as cable drives) are supported through the `Affine` driver type. The other driver types are not fully tested and/or supported, although we hope to add some of this functionality in the future.

C++ implementation. Klamp't is based heavily on the KrisLibrary/robotics package for defining articulated robot kinematics and dynamics. The `Robot` class in Klampt/Modeling/Robot.h has the following class hierarchy:

`Robot` -> `RobotWithGeometry` -> `RobotDynamics3D` -> `RobotKinematics3D` -> `Chain`

The reasons for the class hierarchy are largely historical, but meaningful. For example, a protein backbone might be modeled as a RobotKinematics3D but not a RobotDynamics3D. For more transparent, flat access to the main Robot functionality, see the Python API.

- `Chain` stores the topological sorting of the articulation (the `parents` member).
- `RobotKinematics3D` stores the kinematic and dynamic information of links, joint limits, the current configuration and the current link frames. It also provides methods for computing forward kinematics, jacobians, and the center of mass.
- `RobotDynamics3D` stores the actuator limits and the current velocity. It provides methods for computing information related to the robot's dynamics.
- `RobotGeometry3D` stores link collision geometries and information about which links can self collide. It performs self-collision testing and collision testing with other geometries.
- `Robot` defines link names and semantics of Joints and Drivers.

The `Config` class is simply typedef'ed as a `Vector` (see KrisLibrary/math/vector.h).

5.4. ENVIRONMENTS

An Environment (Klampt/Modeling/Environment.h) is defined very simply as a Collision Geometry annotated with friction coefficients. They may be loaded from .env files or raw geometry files. In the latter case, some default friction value is assigned (set to 0.5).

5.5. RIGID OBJECTS

A `RigidObject` (Klampt/Modeling/RigidObject.h) is a collision mesh associated with a `RigidTransform` and other dynamic parameters. `RigidObjects` may be loaded from .obj files or raw geometry files. In the latter case, the dynamic parameters are set to default values (e.g., mass = 1).

5.6. WORLDS

The `RobotWorld` class (Klampt/Modeling/World.h) stores multiple named robots, environments, and rigid objects, along with associated visualization information. Each entity in the world, including each robot link and each geometry, can be addressed via a common ID number. Worlds are loaded from .xml files.

5.7. PATHS AND TRAJECTORIES

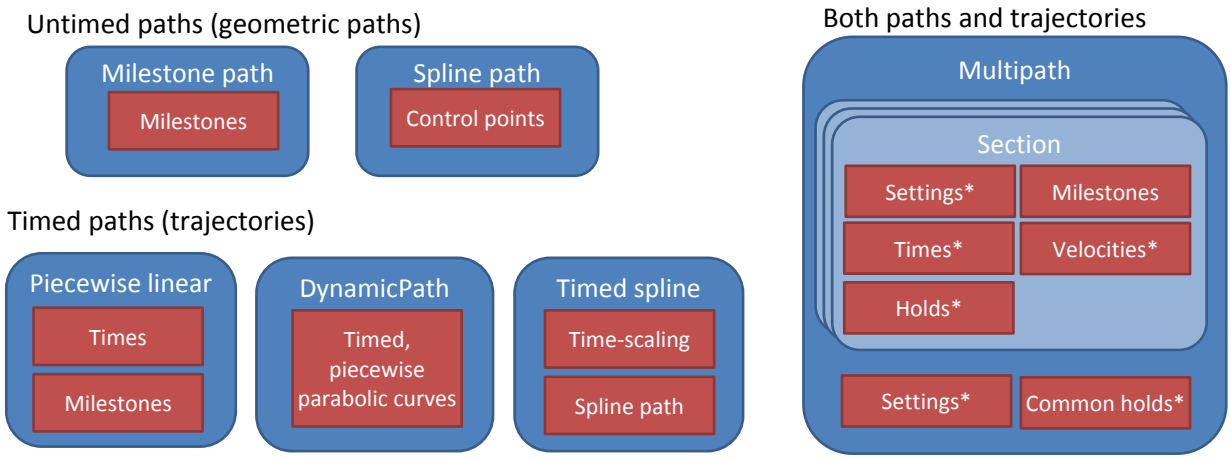


Figure 7. The Path concept models.

Klamp't distinguishes between *paths* and *trajectories*: paths are geometric, time-free curves, while trajectories are paths with an explicit time parameterization. Mathematically, paths are expressed as a continuous curve $y(s)$: $[0,1] \rightarrow C$ while trajectories are expressed as continuous curves $y(t)$: $[t_i, t_f] \rightarrow C$ where C is the configuration space and t_i, t_f are the initial and final times of the trajectory, respectively.

Classical motion planners compute paths because time is essentially irrelevant for fully actuated robots in static environments. However, a robot must ultimately execute trajectories. Various methods are available in Klamp't to convert paths into trajectories.

Klamp't handles two path types.

- *Milestone lists*. The simplest path type is simply a list of *milestones* that should be piecewise linearly interpolated. These are typically simply given as `vector<Config>`. *Note: to properly handle rotational joints, milestones should be interpolated via the functions in* Klampt/Modeling/Interpolate.h*. Cartesian linear interpolation does not correctly handle floating and spin joints.*
- *Cubic splines*. Klamp't also has partial support for cubic Bezier curves. Routines for interpolating configuration lists are found in Klampt/Modeling/SplineInterpolation.h.

Klamp't handles three trajectory types.

- *Piecewise linear*. These trajectories are given by a list of times and milestones that should be piecewise linearly interpolated. These are typically simply given as two arrays: `vector<Real>` and `vector<Config>`. [*See note above regarding interpolation*.]
- *`DynamicPath` (piecewise parabolic curves).* These are time-optimal bounded-acceleration trajectories that include both configuration, velocity, and time. Routines in Klampt/Modeling/DynamicPath.h are available to quickly compute `DynamicPaths` from milestone lists, milestone+velocity lists, and milestone+time lists given velocity and acceleration bounds.
- *Timed cubic splines.* Found in the `TimeScaledBezierCurve` class in Klampt/Planning/TimeScaling.h.

Especially for legged robots, the preferred path type is `MultiPath`, which allows storing both untimed paths and timed trajectories. It can also store multiple path sections with inverse kinematics constraints on each section. Conversions to/from piecewise linear paths, `DynamicPath`'s, and cubic splines are supported.

Multipaths. A `MultiPath` (Klampt/Modeling/MultiPath.h) is a rich path representation for legged robot motion. They contain one or more path (or trajectory) *sections* along with a set of IK constraints and holds that should be satisfied during each of the sections. This information can be used to interpolate between milestones more intelligently, or for controllers to compute feedforward torques more intelligently than a raw path. They are loaded and saved to XML files.

Each `MultiPath` section maintains a list of IK constraints in the `ikObjectives` member, and a list of Holds in the `holds` member. There is also support for storing common holds in the `MultiPath`'s `holdSet` member, and referencing them through a section's `holdNames` or `holdIndices` lists (keyed via string or integer index, respectively). This functionality helps determine which constraints are shared between sections, and also saves a bit of storage space.

`MultiPaths` also contain arbitrary application-specific settings, which are stored in a string-keyed dictionary member `settings`. Common settings include:

- `robot`, which indicates the name of the robot for which the path was generated.
- `resolution`, which indicates the resolution to which a path has been discretized. If `resolution` has not been set or is too large for the given application, a program should use IK to interpolate the path.
- `program`, the name of the procedure used to generate the path.
- `command_line`, the shell command used to invoke the program that generated the path.

Sections may also have settings. No common settings have yet been defined for sections.

5.8. INVERSE KINEMATICS

Inverse kinematics (IK) constraints state that some variables in a link's coordinate system should meet fixed values

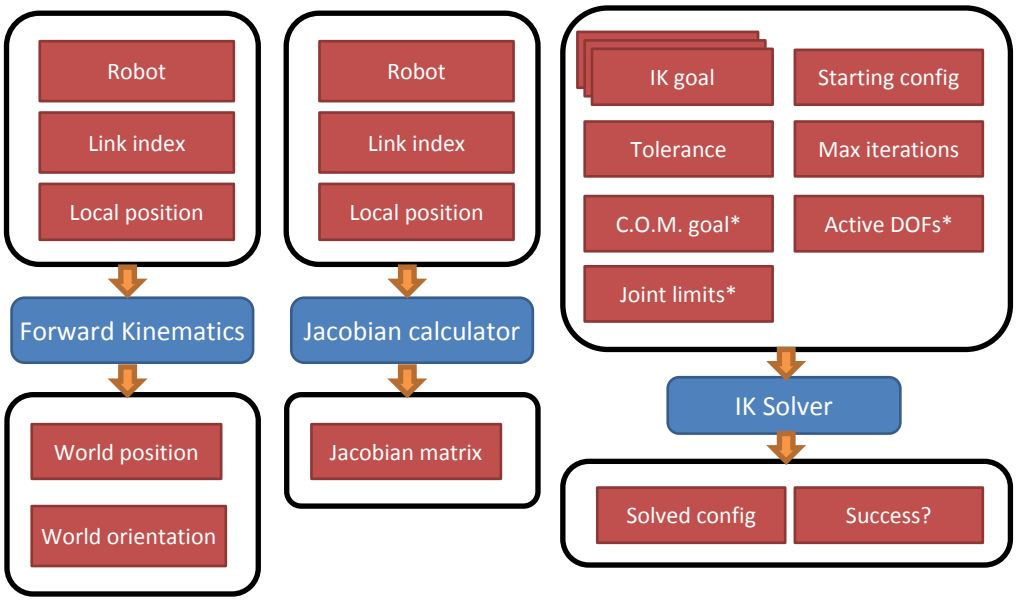


Figure 8. The Forward Kinematics and Jacobian subroutines (implemented in a Robot) and the IK solver subroutine

relative to the world coordinate system, or fixed values in the coordinate system of any other link. These can be position constraints, orientation constraints, or also linear constraints on either position or orientation. To achieve such constraints, Klamp't contains a Newton-Raphson numerical solver for general sets of IK constraints, possibly also including joint limits and center-of-mass constraints.

The `IKGoal` (implemented in KrisLibrary/robotics/IK.h) class defines a constraint on a single link. The `link` member must be filled out prior to use. If the constraint is meant to constrain the link to a target link on the robot (rather than the world), then the `destLink` member should be filled out. By default, `destLink` is -1, indicating that the target is in world coordinates.

Easy setup. For convenience, the `SetFromPoints` method is provided to map a list of local points to a list of target space points. This function covers most typical IK constraints. If there is a single point, the constraint is a fixed point constraint. If the points are collinear, the constraint is an edge constraint. If the points span a plane, the constraint is a fixed constraint.

Detailed setup. Position constraints are defined by the `localPosition`, `endPosition`, and optionally the direction members. There are four types of position constraint available.

- `Free`: no constraint
- `Planar`: the point is constrained in one dimension, i.e., to lie on a plane. Here `endPosition` refers to a point on the plane and `direction` refers to the plane normal.
- `Linear`: the point is constrained in two dimensions, i.e., to lie on a line. Here `endPosition` refers to a point on the line and `direction` refers to the line direction.
- `Fixed`: the point is constrained to a fixed point. Here `endPosition` refers to that point and direction is ignored.

Rotation constraints are defined by the `endRotation` and optionally the `localAxis` members. There are three types of rotation constraint available.

- `Free`: no constraint
- `Axis`: rotation is constrained about an axis. The direction `localAxis` maps to the `endRotation` direction. These must be unit vectors.
- `Fixed`: rotation is fixed. The `endRotation` member is a `MomentRotation` that represents the fixed orientation. To convert to a 3x3 matrix, call the `GetFixedGoalRotation` method. To convert from a 3x3 matrix, call the `SetFixedRotation` method.

Numerical solvers. Numerical inverse kinematics solvers are extremely flexible and can solve for arbitrary combinations of IK constraints. The `SolveIK()` functions in KrisLibrary/robotics/IKFunctions.h are the easiest way to do so. They take the robot's current configuration as a starting point and run a Newton-Raphson technique to (hopefully) solve all constraints simultaneously. These routines automatically try to optimize only over the relevant variables, e.g., if the only constraint is on the robot's right foot, then the arms, head, and left leg will not be included as optimization variables.

For richer functionality, use the `RobotIKFunction` and `RobotIKSolver` classes and `Get*Dofs()` functions directly.

Analytical solvers. There are hooks for analytical solvers in KrisLibrary/robotics/AnalyticIK.h but these are not used yet in Klamp't. Future versions may support them.

5.9. DYNAMICS

The fundamental Langrangian mechanics equation is

$$B(q)\ddot{q} + C(q,\dot{q}) + G(q) = \tau + \sum_i J_i(q)^T f_i \quad (1)$$

Where q is configuration $\dot{q}$ is velocity, $\ddot{q}$is acceleration, $B(q)$ is the positive semidefinite *mass matrix*, $C(q,\dot{q})$ is the *Coriolis force*, $G(q)$ is the *generalized gravity*, τ is the link torque, f_i are *external forces*, and $J_i(q)$ are the Jacobians of the points at which the points are applied. A robot's motion under given torques and external forces can be computed by multiplying both sides by B^{-1} and integrating the equation forward in time.

C++ implementation. Klamp't has several methods for calculating and manipulating these terms. The first set of methods is found in `RobotKinematics3D` and `RobotDynamics3D`. These use the "classic" method that expands the terms mathematically in terms of Jacobians and Jacobian derivatives, and runs in $O(n^3)$. The `CalcAcceleration` method is used to convert the RHS to accelerations (*forward dynamics*). `CalcTorques` is used to convert from accelerations to the RHS (*inverse dynamics*).

The second set of methods uses the Newton-Euler rigid body equations and the Featherstone algorithm (KrisLibrary/robotics/NewtonEuler.h). These equations are O(n) for sparsely branched chains and are typically faster than the classic methods for modestly sized robots (e.g., n>6). Although `NewtonEuler` is designed particularly for the `CalcAccel` and `CalcTorques` methods for forward and inverse dynamics, it is also possible to use it to calculate the C+G term in O(n) time, and it can calculate the B or B^{-1} matrices in $O(n^2)$ time.

5.10. CONTACTS

Klamp't supports several operations for working with contacts. Currently these support legged locomotion more conveniently than object manipulation, because the manipulated object must be defined as part of the robot, and robot-object contact is considered self-contact.

- A `ContactPoint` is either a frictionless or frictional point contact. Consist of a position, normal, and coefficient of friction.
- A `ContactFormation` defines a set of contacts on multiple links of a robot. Consists of a list of links and a list of lists of contacts. For all indices `i`, `contacts[i]` is the set of contacts that affect `links[i]`. Optionally, self-contacts may be defined by providing the list of target links `targets[i]`, with -1 denoting the world coordinate frame. Contact quantities may be given target space or in link-local coordinates is application-defined.
- The `TestCOMEquilibrium` functions test whether the center of mass of a rigid body can be stably supported against gravity by valid contact forces at the given contact list.
- The `EquilibriumTester` class provides richer functionality than `TestCOMEquilibrium`, such as force limiting and adding robustness factors. It may also save some memory allocations when testing multiple centers of mass with the same contact list.
- The `SupportPolygon` class explicitly computes a support polygon for a given contact list, and provides even faster testing than `EquilibriumTester` for testing large numbers of centers of mass (typically around 10-20).
- The `TorqueSolver` class solves for equilibrium of an articulated robot under gravity and torque constraints. It can handle both statically balanced and dynamically moving robots.

C++ implementation. These routines can be found in KrisLibrary/robotics, in particular Contact.h, Stability.h, and TorqueSolver.h.

5.11. HOLDS, STANCES, AND GRASPS

The contact state of a single link, or a related set of links, is modeled with three higher-level concepts. `Holds` are a set of contacts of a link against the environment and are used for locomotion planning. `Stances` are a set of Holds. `Grasps` are generally used for manipulation planning but could also be part of locomotion as well (grasping a rail for stability, for example).

`Holds` (Klampt/Contact/Hold.h) are defined as a set of contacts (the `contacts` member) and the associated IK constraint (the `ikConstraint` member) that keeps a link on the robot placed at those contacts. These contacts are considered fixed in the world frame. Holds may be saved and loaded from disk. They also contain convenience setup routines in the `Setup*` methods.

`Stances` (Klampt/Contact/Stance.h) define all contact constraints of a robot. They are defined simply as a map from links to Holds.

`Grasps` (Klampt/Contact/Grasp.h) is more sophisticated than a hold and is most appropriate for modeling hands that make contact with fingers. A Grasp defines an IK constraint of some link (e.g., a palm) relative to some movable object or the environment, as well as the values of related link DOFs (e.g., the fingers) and possibly the contact state. *Note: support for planning with Grasps is limited in the current version.*

5.12. RESOURCES AND RESOURCE LIBRARIES

Most of the types mentioned in this section can be saved and loaded from disk conveniently through the Klamp't resource management mechanism. When working on a large project, it is recommended that configurations, paths, holds, etc. be stored in dedicated sub-project folders to avoid polluting the main Klamp't folder.

A sub-project folder can be loaded all at once through the `ResourceLibrary` class (KrisLibrary/utils/ResourceLibrary.h). After initializing a `ResourceLibrary` instance with the `MakeRobotResourceLibrary` function in (Klampt/Modeling/Resources.h) to make it Klamp't-aware, the `LoadAll/SaveAll()` methods can load an entire folder of resources. These resources can be accessed by name or type using the `Get*()` methods.

Currently supported types include:

- `Config` (.config)
- `Hold` (.hold)
- `Stance` (.stance)
- `Grasp` (.xml)
- Configuration lists (.configs)
- `TriMesh` (.tri)
- `PointCloud` (.pcd)
- `Robot` (.rob)
- `World` (.xml)
- Linear paths (.path) (Note that the data must be extracted from the `LinearPathResource`)
- `MultiPath` (.xml)

Alternatively, resource libraries can be saved to XML files via the `LoadXml/SaveXml()` methods. This mechanism may be useful in the future, for example to send complex robot data across a network. These also support the following additional types which do not have a dedicated file extension:

- `Vector3`
- `Matrix3`
- `RigidTransform`
- `Matrix`
- `IKGoal`

5.13. FILE TYPES

The following standard file types are used in Klamp't.

- World files (.xml)
- Robot files (.rob)
- Triangle mesh files (.tri)
- Rigid object files (.obj)
- Configuration files (.config)
- Configuration set files (.configs)
- Simple linear path files (.path)
- Multipath files (.xml)
- Hold files (.hold)

- Stance files (.stance)
- Grasp files (.xml)

Robot (.rob) files

Structure: a series of lines, separated by newlines. Comments start with `#`, may appear anywhere on a line, and comments continue until the end of the line. Lines can be continued to the next line using the backslash `\`.

A robot has N links, and D drivers. Elements of each line are whitespace-separated. Indices are zero-based. `inf` indicates infinity. Some items are optional, indicated by default values.

Kinematic items:

- `links LinkName[0] … LinkName[N-1]`: link names, names with spaces can be enclosed in quotes.
- `parents parent[0] … parent[N-1]`: link parent indices. -1 indicates a link's parent is the world frame.
- `jointtype v[0] … v[N-1]`: DOF motion type, can be `r` for revolute or `p` for prismatic.
- `tparent T[0] … T[N-1]`: relative rigid transforms between each link and its parent. Each T[i] is a list of column vectors of the rotation matrix, followed by the translation (12 values for each T).
- `{alpha, a, d, theta} v[0] … v[N-1]`: Denavit-Hartenberg parameters. Either tparent or D-H parameters must be specified. alphadeg is equivalent to alpha and thetadeg is equivalent to theta, but in degrees.
- `axis a[0] … a[N-1]`: DOF axes, in the local frame of the link (`3` values for each a). Default: z axis (0,0,1).
- `qmin v[0] … v[N-1]`: configuration lower limits, in radians. `qmindeg` is equivalent, but in degrees. Default: -inf.
- `qmax v[0] … v[N-1]`: configuration upper limits, in radians. `qmaxdeg` is equivalent, but in degrees. Default: inf.
- `q v[0] … v[N-1]`: initial configuration values, in radians. `qdeg` is equivalent, but in degrees. Default: 0.
- `translation`: a shift of link 0. Default: 0.
- `rotation`: a rotation of link 0, given by columns of a 3x3 rotation matrix. Default: identity.
- `scale`: scales the entire robot model.
- `mount link fn [optional transform T]`: mounts the sub-robot file in `fn` as a child of link `link`. If T is provided, this is the relative transform of the sub-robot given by columns of a 3x3 rotation matrix followed by the translation (12 values in T).

Dynamic Items:

- `mass v[0] … v[N-1]`: link masses.
- `automass`: set the link centers of mass and inertia matrices automatically from the link geometry.
- `com v[0] … v[N-1]`: link centers of mass, given in local (x,y,z) coordinates (3 values for each v). May be omitted if `automass` is included.
- `inertiadiag v[0] … v[N-1]`: link inertia matrix diagonals (Ixx, Iyy, Izz), assuming off-diagonal elements are all zero (3 values for each v). May be omitted if `inertia` or `automass` is included.
- `inertia v[0]` … v[N-1]: link 3x3 inertia matrices (9 items for each v). May be omitted if `inertiadiag` or `automass` is included.
- `velmin v[0] … v[N-1]`: configuration velocity lower limits, in radians. `velmindeg` is equivalent, but in degrees. Default: -inf.

- `velmax v[0] … v[N-1]`: configuration velocity upper limits, in radians. `velmaxdeg` is equivalent, but in degrees. Default: inf.
- `accmax v[0] … v[N-1]`: configuration acceleration absolute value limits, in radians. `accmaxdeg` is equivalent, but in degrees. Default: inf.
- `torquemax v[0] … v[N-1]`: DOF torque absolute value limits, in Nm (revolute) or N (prismatic). Default: inf.
- `powermax v[0] … v[N-1]`: DOF power (torque*velocity) absolute value limits. Default: inf.
- `autotorque`: set the torquemax values according to an approximation: acceleration maxima * masses * radii of descendent links.
-

Geometric items:

- `geometry fn[0] … fn[N-1]`: geometry files for each link. File names can be either absolute paths or relative paths. Files with spaces can be enclosed in quotes. Empty geometries can be specified using “”.
- `geomscale v[0] … v[N-1]`: scales the link geometry. Default: 1.
- `geomtransform v[0] … v[N-1]`: transforms the link geometry with a 4x4 transformation matrix in row-major order. Default: identity.
- `geommargin v[0] … v[N-1]`: sets the collision geometry to have this virtual margin around each geometric mesh. Default: 0.
- `noselfcollision i[0] j[0] … i[k] j[k]`: turn off self-collisions between the indicated link pairs. Each item may be a link index in the range 0,…,N-1 or a link name.
- `noselfcollision i[0] j[0] … i[k] j[k]`: turn on self-collisions between the indicated link pairs. Each item may be a link index in the range 0,…,N-1 or a link name. Default: all self-collisions enabled, except for link vs parent.

Joint items:

- `joint type index [optional baseindex]`: TODO: describe joint types normal, spin, weld, floating, floatingplanar, ballandsocket

Driver items:

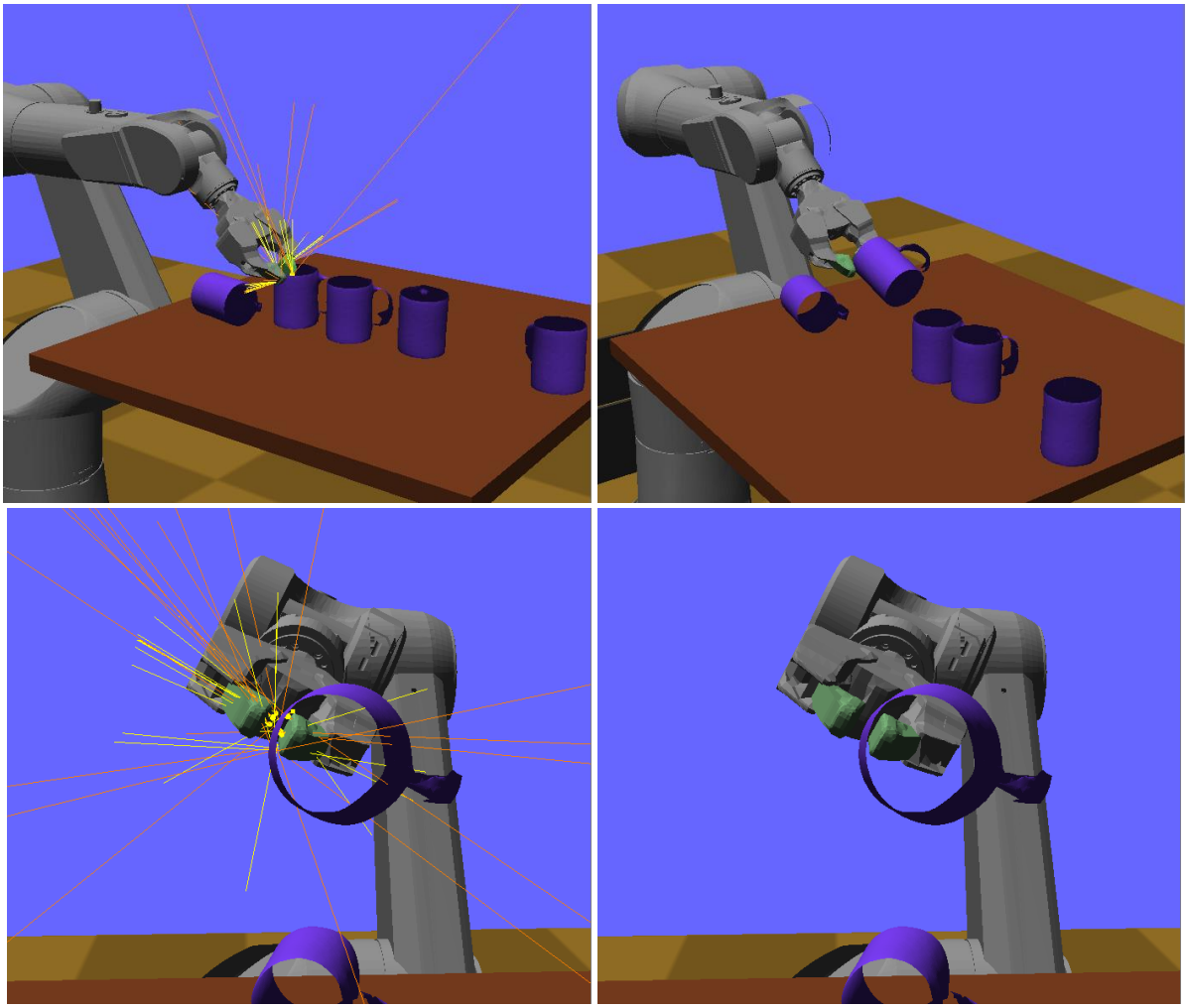
- `driver type [params]`: TODO: describe driver types normal, affine, translation, rotation.
- `servoP`: driver position gains.
- `servoI`: driver integral gains.
- `servoD`: driver derivative gains.
- `dryFriction`: driver dry friction coefficients.
- `viscousFriction`: driver viscous friction coefficients.

6. SIMULATION

Simulation functionality in Klamp't is built on top of the Open Dynamics Engine (ODE) rigid body simulation package, but adds emulators for robot sensors and actuators, and features a robust contact handling mechanism. When designing new robots and scenarios, it is important to understand a few details about how Klamp't works in order to achieve realistic simulations.

Boundary-layer contact detection. Other rigid body simulators tend to suffer from significant collision handling artifacts during mesh-mesh collision: objects will jitter rapidly, interpenetrate, or react to "phantom" collisions. The primary cause is that contact points, normals, and penetration depths are estimated incorrectly or inconsistently from step-to-step. Klamp't uses a new *boundary layer contact detection* procedure that leads to accurate and consistent estimation of contact regions. Moreover, the boundary layer can simulate some limited compliance in the contact interface, such as soft rubber coatings or soft ground.

In Klamp't, contact is detected along the boundary layers rather than the underlying mesh. The thickness of the boundary layer is a simulation parameter called *padding*. Padding for each body can be set via the `padding` attribute in the `<simulation>{<robot>,<object>,<terrain>}<geometry>` XML element, with all bodies padded with 2.5mm by default. This allows it to handle thin-shell meshes as illustrated in the following figure.



The first step of Klamp't's collision handling routine is to compute all contacts between all pairs of geometric primitives within the padding range. This is somewhat slow when fine meshes are in contact. In order to reduce the number of contacts that must be handled by ODE, Klamp't then performs a clustering step to reduce the number of contacts to a manageable number. The maximum number of contacts between two pairs of bodies is given by the *maxContacts* global parameter, which can be set as an attribute in the XML `<simulation>` tag.

For more details, please see: *K. Hauser. Robust Contact Generation for Robot Simulation with Unstructured Meshes. In proceedings of International Symposium of Robotics Research, 2013.*

Collision response. In addition to padding, each body also has coefficients of restitution, friction, stiffness, and damping (`kRestitution`, `kFriction`, `kStiffness`, and kD`amping` attributes in `<simulation>{<robot>,<object>,<terrain>}<geometry>` XML elements). The stiffness and damping coefficients can be set to non-infinite values to simulate softness in the boundary layer. When two bodies come into contact, these four coefficients are blended using arithmetic, harmonic, harmonic, and harmonic means, respectively.

The blending mechanism is convenient because only one set of parameters needs to be set for each body, rather than each pair of bodies, and is a reasonable approximation of most material types. Currently there is no functionality to specify custom properties between pairs of bodies.

Actuator simulation. Klamp't handles actuators in one of two modes: PID control and torque control modes. It also simulates dry friction (stiction) and viscous friction (velocity-dependent friction) in joints using the `dryFriction` and `viscousFriction` parameters in .rob files. Actuator commands are converted to torques (if in PID mode), capped to torque limits, and then applied directly to the links. ODE then handles the friction terms.

In PID mode, the torque applied by the actuator is $\tau = k_P(\theta_D - \theta_A) + k_D(\dot{\theta}_D - \dot{\theta}_A) + k_I I$ where k_P, k_I, and k_D are the PID constants, θ_D and $\dot{\theta}_D$ are the desired position and velocity, θ_A and θ_A are the actual position and velocity, and I is an integral error term.

The friction forces resist the motion of the joint, and Klamp't uses a simple stick-slip friction model where the sticking mode breaking force is equal to μ_D and the sliding mode friction force is $-sgn(\dot{\theta}_A)(\mu_D + \mu_V|\dot{\theta}_A|)$. *Note: passive damping should be handled via the friction terms*.

Like all simulators, Klamp't does not perfectly simulate all of the physical phenomena affecting real robots. Some common phenomena include:

- Backlash in the gears.
- Back EMF.
- Angle-dependent torques in cable drives.
- Motor-induced inertial effects, which are significant particularly for highly geared motors. Can be approximated by adding a new motor link connected by an affine driver to its respective link.
- Velocity-dependent torque limits (e.g. power limits). Can be approximated in a controller by editing the robot's driver torque limits depending on velocity. Can be correctly implemented by adding a `WorldSimulationHook` or editing the `ControlledRobotSimulator` class.
- Motor overheating. Can be implemented manually by simulating heat production/dissipation as a differential equation dependent on actuator torques. May be implemented in a `WorldSimulationHook`.

7. PLANNING

Motion planning is the problem of connecting two configurations with a feasible kinematic path or dynamic trajectory under certain constraints. The output may also be required to satisfy some optimality criteria. Klamp't has the ability to plan:

- Collision-free kinematic paths in free space,
- Collision-free, stable kinematic paths on constraint manifolds,
- Minimum-time executions of a fixed trajectory under velocity and acceleration constraint,
- Minimum-time executions of a fixed trajectory under torque and frictional force constraints,
- Replanning under hard real-time constraints.

7.1. BASIC KINEMATIC MOTION PLANNING

Basic kinematic motion planning generates collision-free paths for fixed-base robots in free space (i.e., not in contact with the environment or objects). The general way to plan a path connecting configurations `qstart` and `qgoal` is as follows:

1. Initialize a `WorldPlannerSettings` object for a `RobotWorld` with the `InitializeDefault` method.
2. Create a `SingleRobotCSpace` (Klampt/Planning/RobotCSpace.h) with the `RobotWorld`, the index of the robot (typically 0), and the initialized `WorldPlannerSettings` object.
3. Then, a `MotionPlannerFactory` (KrisLibrary/planning/AnyMotionPlanner.h) should be initialized with your desired planning algorithm. The `SBL` type is recommended as a good first choice.
4. Construct a `MotionPlanningInterface*` with the `MotionPlannerFactory.Create()` method. Call `MotionPlanningInterface.AddConfig(qstart)` and `MotionPlanningInterface.AddConfig(qgoal)`
5. Call `MotionPlanningInterface.PlanMore(N)` to plan for `N` iterations, or call `PlanMore`() until a time limit is reached. Terminate when `IsConnected(0,1)` returns true, and call `GetPath(0,1,path)` to retrieve the path.
6. Delete the `MotionPlanningInterface*`.

Example code is as follows.

```
#include "Planning/SingleRobotCSpace.h"
#include <planning/AnyMotionPlanner.h>

//TODO: setup world
WorldPlannerSettings settings;
settings.InitializeDefault(world);
//do more constraint setup here if desired, e.g., set edge collision checking resolution
SingleRobotCSpace cspace(world,0,&settings); //plan for robot 0
MotionPlannerFactory factory;
factory.type = MotionPlannerFactory::SBL;
//do more planner setup here if desired, e.g., change perturbation size
MotionPlanningInterface* planner = factory.Create(&cspace);
int istart=planner->AddConfig(qstart); //should be 0
int igoal=planner->AddConfig(qgoal); //should be 1
int maxIters=1000;
bool solved=false;
MilestonePath path;
for(int i=0;i<maxIters;i++) {
 planner->PlanMore();
 if(planner->IsConnected(0,1)) {
  planner->GetPath(0,1,path);
```

```
      solved=true;
      break;
     }
    }
    delete planner;
```

The default settings in `WorldPlannerSettings` (Klampt/Planning/PlannerSettings.h) and `MotionPlannerFactory` should be sufficient for basic testing purposes, but many users will want to tune them for better performance. For example, collision avoidance margins, distance metric weights, and contact tolerances may be tuned.

To plan for part of a robot (e.g., the arm of a legged robot), the `SingleRobotCSpace2` class can be used instead. Be sure to configure the `fixedDofs` and `fixedValues` members before using it.

Note: although RobotCSpace.h contains multi-robot planning classes, they are not yet well-tested. Use at your own risk.

7.2. TIME-OPTIMAL ACCELERATION-BOUNDED TRAJECTORIES

The result of kinematic planning is a sequence of milestones, which ought to be converted to a time-parameterized trajectory to be executed. The standard path controllers (see Section 8.3) do accept milestone lists and will do this internally. Occasionally you may want to do this manually, for example, to perform path smoothing before execution.

This functionality is contained within the `DynamicPath` class in the Klampt/Modeling/DynamicPath.h file, which builds on the classes in Klampt/Modeling/ParabolicRamp.h. To shortcut a path, the following procedure is used:

1. Set the velocity and acceleration constraints, and optionally, the joint limits in the `DynamicPath`.
2. Call `DynamicPath.SetMilestones()`. The trajectory will now interpolate linearly and start and stop at each milestone.
3. Subclass the `FeasibilityCheckerBase` class with the appropriate kinematic constraint checkers overriding `ConfigFeasible` and `SegmentFeasible`. Construct an instance of this checker.
4. Construct a `RampFeasibilityChecker` with a pointer to the `FeasibilityCheckerBase` instance and an appropriate checking resolution.
5. Call `DynamicPath.Shortcut(N,checker)` where `N` is the desired number of shortcuts.

The resulting trajectory will be smoothed, satisfy velocity and acceleration bounds, and feasible.

Warning: free-rotational joints will not be interpolated correctly. Spin joints are not automatically handled correctly at step 3 and must be "unwrapped" manually; step 5 must be replaced with the `WrappedShortcut` method.

For more details, please see: *K. Hauser and V. Ng-Thow-Hing. Fast Smoothing of Manipulator Trajectories using Optimal Bounded-Acceleration Shortcuts. In proceedings of IEEE Int'l Conference on Robotics and Automation (ICRA), 2010.*

7.3. INTERPOLATION AND TIME-OPTIMIZATION WITH CLOSED-CHAIN CONSTRAINTS

Several routines in Klampt/Planning/RobotTimeScaling.h are used to interpolate paths under closed chain constraints. There is also functionality for converting paths to minimum-time, dynamically-feasible trajectories using a time-scaling method.

The suggested method for doing so is to use a `MultiPath` with the desired constraints in each section, and to input the control points as milestones. `DiscretizeConstrainedMultiPath` can be used to produce a new path that interpolates the milestones, but with a finer-grained set of constraint-satisfying configurations. `EvaluateMultiPath` interpolates a configuration along the path that satisfies the constraints. `GenerateAndTimeOptimizeMultiPath` does the same as `DiscretizeConstrainedMultiPath` except that the timing of the configurations is optimized as well.

Each method takes a resolution parameter that describes how finely the path should be discretized. In general, interpolation is slower with finer discretizations.

See the following reference for more details: K. Hauser. *Fast Interpolation and Time-Optimization on Implicit Contact Submanifolds*. Robotics: Science and Systems, 2013.

7.4. RANDOMIZED PLANNING WITH CLOSED-CHAIN CONSTRAINTS

To plan for collision-free motions that satisfy closed chain constraints (e.g., that a robot's hands and feet touch a support surface), the `ContactCSpace` class (Klampt/Planning/ContactCSpace.h) should be used in the place of `SingleRobotCSpace`. Fill out the `contactIK` member, optionally using the `Add*()` convenience routines. The kinematic planning approach can then be used as usual.

Note that the milestones outputted by the planner should NOT be interpolated linearly because the motion lies on a lower-dimensional, nonlinear constraint manifold in configuration space. Rather, the path should be discretized finely on the constraint manifold before sending it to any function that assumes a configuration-space path. There are two methods for doing so: 1) using `MilestonePath.Eval()` with a fine discretization, which uses the internal `ContactCSpace::Interpolate` method, or 2) construct an interpolating path via the classes in Klampt/Planning/RobotConstrainedInterpolator.h. This latter approach guarantees that the resulting path is sufficiently close to the constraint manifold when interpolated linearly.

To use `RobotConstrainedInterpolator`, construct an instance with the robot and its IK constraints. Then, calling `RobotConstrainedInterpolator.Make()` with two consecutive configurations will produce a list of finely-discretized milestones up to the tolerance `RobotConstrainedInterpolator.xtol`. Alternatively, the `RobotSmoothConstrainedInterpolator` class and the `MultiSmoothInterpolate` function can be used to construct a smoothed cubic path.

7.5. TIME-SCALING OPTIMIZATION

The `TimeOptimizePath` and `GenerateAndTimeOptimizeMultiPath` functions in Klampt/Planning/RobotTimeScaling.h perform time optimization with respect to a robot's velocity and acceleration bounds. `TimeOptimizePath` takes a piecewise linear trajectory as input, interpolates it via a cubic spline, and then generates keyframes of time-optimized trajectory. `GenerateAndTimeOptimizeMultiPath` does the same except that it takes `MultiPath`s as input and output, and the constraints of the multipath may be first interpolated at a finer resolution before time-optimization is performed.

7.6. REAL-TIME MOTION PLANNING

TODO

8. CONTROL

Controllers provide the "glue" between the physical robot's actuators, sensors, and planners. They are very similar to planners in that they generate controls for the robot, but the main difference is that a controller is expected to work online and synchronously within a fixed, small time budget. As a result, they can only perform relatively light computations.

8.1. ACTUATORS

At the lowest level, a robot is controlled by *actuators*. These receive instructions from the controller and produce link torques that are used by the simulator. Klamp't supports three types of actuator:

- *Torque control* accepts torques and feeds them directly to links.
- *PID control* accepts a desired joint value and velocity and uses a PID control loop to compute link torques servo to the desired position. Gain constants kP, kI, and kD should be tuned for behavior similar to those of the physical robot. PID controllers may also accept feedforward torques.
- *Locked velocity control* drives a link at a fixed velocity. *Experimental*. (Note: this is different from "soft" velocity control which feeds a piecewise linear path to a PID controller)

Note that the PID control and locked velocity control loops are performed as fast as possible with the simulation time step. This rate is typically faster than that of the robot controller. Hence a PID controlled actuator typically performs better (rejects disturbances faster, is less prone to instability) than a torque controlled actuator with a simulated PID loop at the controller level.

Important: When using Klamp't to prototype behaviors for a physical robot, the simulated actuators should be calibrated to mimic the robot's true low-level motor behavior as closely as possible. It is also the responsibility of the user to ensure that the controller uses the simulated actuators in the same fashion as it would use the robot's physical actuators. For example, for a PID controlled robot with no feedforward torque capabilities, it would not be appropriate to use torque control in Klamp't. If a robot does not allow changing the PID gains, then it would not be appropriate to do so in Klamp't. Klamp't will not automatically configure your controller for compatibility with the physical actuators, nor will it complain if such errors are made.

C++ implementation. The `RobotMotorCommand` (Klampt/Control/Command.h) structure contains a list of `ActuatorCommands` that are then processed by the simulator.

8.2. SENSORS

Klamp't can emulate a handful of sensors typically found on robots. At the user's level of abstraction, they generically provide streaming numerical-valued measurements. It is up to the user to process these raw measurements into meaningful information.

The following sensors are natively supported:

- `JointPositionSensor`: Standard joint encoders.
- `JointVelocitySensor`: Velocity sensors. Here velocities are treated raw measurements, not differenced from a position encoder, and hence they are rarely found in real life. However, these will be good approximations of differenced velocity estimates from high-rate encoders.

- `DriverTorqueSensor`: Torques fed back from a robot's motors.
- `ContactSensor`: A contact switch/sensor defined over a rectangular patch.
- `ForceTorqueSensor`: A force/torque sensor at a robot's joint. Can be configured to report values from 1 to 6DOF.
- `Accelerometer`: An accelerometer. Can be configured to report values from 1 to 3 channels.
- `TiltSensor`: A tilt sensor. Can be configured to report values from 1 to 2 axes, and optionally tilt rates.
- `GyroSensor`: A gyroscope. Can be configured to report accelerations, velocities, or absolute rotations.
- `IMUSensor`: An inertial measurement unit that uses an accelerometer and/or gyroscope to provide estimates of a link's transformation and its derivatives. It will fill in the gaps that are not provided by the accelerometer / gyro using either integration or differencing.
- `FilteredSensor`: A "virtual sensor" that simply filters the measurements provided by another sensor.

Sensors can be dynamically configured from world XML files under the `<simulation>` and `<robot>` elements via a statement of the form `<sensor type="TheSensorType" name="some_name" attr1="value" … />`. Each of the attribute/value pairs is fed to the sensor's `SetSetting` method, and details on sensor-specific settings are found in the documentation in Control/Sensor.h.

8.3. CONTROLLERS

The number of ways in which a robot may be controlled is infinite, and can range from extremely simple methods, e.g., a linear gain, to extremely complex ones, e.g. an operational space controller or a learned policy. Yet, all controllers are structured as a simple callback loop: repeatedly read off sensor data from a `RobotSensors` structure, perform some processing, and write commands to a `RobotMotorCommands` structure. The implementation of the internal processing is open to the user.

Any controller must subclass the `RobotController` class (Klampt/Control/Controller.h) and overload the `Update` method. The members `sensors` and `command` are available for the subclass to use.

Dynamically loadable controllers. Controllers can be dynamically and automatically loaded from world XML files via a statement of the form `<controller type="TheControllerType" attr1="value" … />` under the `<simulation>` and `<robot>` elements. The following controllers are supported:

- *JointTrackingController* (Klampt/Control/JointTrackingController.h): a simple open-loop controller that accepts a desired setpoint.
- *MilestonePathController* (Klampt/Control/PathController.h): an open-loop controller based on a `DynamicPath` trajectory queue.
- *PolynomialPathController* (Klampt/Control/PathController.h): an open-loop controller based on a `PiecewisePolynomialSpline` trajectory queue. Somewhat more flexible than `MilestonePathController`.
- *FeedforwardJointTrackingController* (Klampt/Control/FeedforwardController.h): a controller that additionally computes feedforward torques for gravity compensation and acceleration compensation. Works properly only with fixed-based robots. Otherwise works exactly like `JointTrackingController`.
- *FeedforwardMilestonePathController*: see above.
- *FeedforwardPolynomialPathController*: see above.

New controller types can also be defined for dynamic loading in world XML files using the `RobotControllerFactory::Register(name,ptr)` function. This hook must be called before the world file is

loaded. Afterward, the specified controller type will be instantiated whenever the registered type appears in the world file.

Generic external interfaces. Optionally, controllers may expose various configuration settings to be loaded from XML files by implementing the `*Settings` methods. (These may also be manipulated by GUI programs and higher-level controllers/planners). They may also accept arbitrary external commands by overloading the `*Command*` methods.

8.4. STATE ESTIMATION

Controllers may or may not perform state estimation internally. If so, it is good practice to define the state estimator as independent of the controller, such as via a subclass of `RobotStateEstimator`. The `RobotStateEstimator` interface is fairly sparse, but the calling convention helps standardize their use in controllers.

Using state estimators. Controllers should instantiate a state estimator explicitly on construction. Inside the `Update` callback, the controller should:

1. Call `RobotStateEstimator.ReadSensors(*sensors)`, then `UpdateModel()` to update the robot's model.
2. Read off the estimated state of the robot model (and potentially other information computed by the state estimator, such as uncertainty levels) and compute its command as usual.
3. Just before returning, call the `ReadCommand(*command)` and `Advance(dt)` methods on the `RobotStateEstimator` object.

A few experimental state estimators are available. `OmniscientStateEstimator` gives the entire actual robot state to the controller, regardless of the sensors available to the robot. `IntegratedStateEstimator` augments accelerometers and gyros with an integrator that tries to track true position. These integrators are then merged (in a rather simple-minded way) to produce the final model.

8.5. CONTROLLER INTEGRATION

To **connect a controller to a simulated robot**, simply construct the controller and call the `WorldSimulation.SetController()` method. Or, the controller type can be specified in the world XML file as described in Section 8.3.

To **connect a controller to a physical robot**, slightly more work is needed to write a wrapper loop that repeatedly fills in the controller's sensor data from the physical data, and write's the controller's actuator commands to the physical motors.

To **connect a planner to a controller**, there are two options. The first is to *externally instantiate* a planning thread that communicates periodically with the controller through some well-defined interface (for example, the `SendCommand()` API). The second is to *internally instantiate* a planning thread inside the controller, and the controller can read data from the planner whenever it is available. Both methods are suitable, so the choice is simply a matter of taste.

9. C++ PROGRAMMING

Klamp't is written in C++, and using C++ will give you full access to its functionality. But, it does require comfort with large code bases and moderate-to-advanced C++ programming abilities. Here are some conventions and suggestions for programming C++ apps that use Klamp't.

- Use a debugger (e.g., GDB) to debug crashes.
- Use STL and smart pointers (KrisLibrary/utils/SmartPointer.h) rather than managing memory yourself.
- KrisLibrary contains a lot of functionality, including linear algebra routines, 3D math, optimization, geometric routines, OpenGL drawing, statistics, and graph structures. Browse KrisLibrary before you reinvent the wheel.
- Avoid hard-coding. A much better practice is to place all settings into a class (e.g., with a `robotLeftHandXOffsetAmount` member) that gets initialized to a default value in the class' constructor. If you need to hard-code values, define them as `const static` variables or `#defines` at the top of your file. Name them descriptively, e.g., `gRobotLeftHandXOffsetAmount` is much better than `shift` or (God forbid) `thatStupidVariable`, when you come back to the file a month from now.
- The main() function in Klampt/Main/simtest.cpp is a good reference for setting up a world and a simulation from command-line arguments.

10. PYTHON PROGRAMMING

The Klampt/Python folder contains a Python API for Klamp't that is much cleaner and easier to work with than the C++ API. It is, however, not as fully functional.

10.1. THE KLAMPT MODULE

The core modeling and simulation Klamp't functionality is found in the klampt module. Users will typically load a `WorldModel`, construct a `Simulator`, and implement a robot controller by interacting with the `SimRobotController`. They may also wish to use the `RobotModel` to compute IK solutions (via the `IKObjective` and `IKSolver` classes), or do other kinds of planning tasks. A simple example file is found in Klampt/Python/demos/gltest.py.

Motion queue control. By default, the `SimRobotController` class implements a `FeedforwardMilestonePathController`, which is a motion-queued controller with optional feedforward torques. The `setMilestone` and `addMilestone` methods set and append a new destination milestone, respectively.

Custom control. It is possible to completely override the controller's behavior to implement a custom control loop by reading the robot's sensors, computing the control, and sending the control to the robot via the `setPIDCommand` or `setTorqueCommand` methods at every simulation time step.

Sub-modules of klampt:

- vectorops: basic vector operations.
- so3: routines for handling rotations.
- se3: routines for handling rigid transformations
- trajectory: a basic piecewise linear trajectory class.
- hold.py: defines a `Hold` class and writes / reads holds to / from disk.

- multipath.py: defines a `MultiPath` class and writes / reads `MultiPath`'s to / from disk. Can also be run as a script to perform various simple transformations on paths.
- collide: bindings for C++ primitive-primitive, primitive-mesh, and mesh-mesh collision detection (see below). Flexible, but the robotcollide module is much more convenient to use when checking collisions in a world.
- robotcollide: defines a `WorldCollider` class that enables querying the collision status of the world and subsets of bodies in the world.
- contact: allows querying contact maps from a simulator and computing wrench matrices. *Stability testing not supported yet.*
- ik: convenience routines for setting up and solving IK constraints. *We do not yet allow solving across multiple robots and objects but this functionality may be supported in the future.*
- loading: methods for loading/saving Klamp't objects to strings, which can be loaded/saved to disk.
- map: convenient object-oriented interface for accessing worlds, robots, objects, links, etc. For example, you can write

```
wm = map.map(world)
wm.robots[0].links[4].transform
```

  instead of

```
world.robot(0).getLink(4).getTransform().
```

- glprogram: a 3D navigation and basic user interface class based on GLUT.
- gldraw: OpenGL drawing routines for primitive objects
- motionplanning: bindings to C++ motion planners. The interface in the cspace module is much more convenient.
- cspace: configuration space base classes and a motion plan class.
- robotcspace: defines a configuration space for a robot in a world to be used in kinematic motion planning.

The klampt module does not (yet) contain interfaces to contact analysis, trajectory optimization, state estimation, and resource libraries. Instead these must be implemented in the user's own Python code.

10.2. COLLISION TESTING

The klampt.collide module allows collision testing between geometric primitives and triangle meshes. Prototypes and documentation are defined in klampt/src/collide.h.

For convenience, the klampt.robotcollide module provides a `WorldCollider` class that by checks collision between any set of objects and any other set of objects. These methods return an iterator over collision pairs, which allows the user to either stop at the first collision or enumerate all collisions. The following methods are used most often:

- `collisions()`: checks for all collisions.
- `collisions(filter)`: checks for all collisions between objects for which `filter(obj)` returns `True`
- `collisions(filter1,filter2)`: checks for all collisions between pairs of objects for which `filter1(objA)` and `filter2(objB)` both return True

- `robotSelfCollisions`, `robotObjectCollisions`, `robotTerrainCollisions`, `objectObjectCollisions`, and `objectTerrainCollisions` check collisions only between the indicated robots/objects/terrains.
- `rayCast(s,d)` performs ray casting against objects in the world and returns the nearest collision found.

10.3. MOTION PLANNING

The klampt.motionplanning module is used with the configuration space prototype base classes (`CSpace` in cspace.py) to invoke a motion planner (`MotionPlan` class in klampt.motionplanning).

To define a custom `CSpace`, subclasses will need to override (* indicates that the method is optional):

- `feasible(x)`: returns true if the vector x is in the feasible space
- *`sample()`: returns a new vector x from a superset of the feasible space. If this is not overridden, then subclasses should set `CSpace.bound` to be a list of pairs defining an axis-aligned bounding box.
- *`sampleneighborhood(c,r)`: returns a new vector x from a neighborhood of c with radius r
- *`visible(a,b)`: returns true if the path between a and b is feasible. If this is not overridden, then paths are checked by subdivision, with the collision tolerance `CSpace.eps`.
- *`distance(a,b)`: return a distance between a and b
- *`interpolate(a,b,u)`: interpolate between a, b with parameter u

`cspaceutils.py` contains helpers for constructing composite `CSpaces` and slices of `CSpaces`.

The `MotionPlan` class supports various options that must be set before construction of the planner.

- `setOptions` takes a variety of arguments including:
  - 'knn': k-nearest neighbors parameter.
  - 'connectionThreshold': maximum distance over which a connection between two configurations is attempted.
  - 'perturbationRadius': maximum expansion radius for RRT and SBL.

  SBL takes other various settings, as described in the `setPlanSetting` documentation in the Python/klampt/src/motionplanning.h file.
- The constructor selects between the PRM, RRT, and SBL planners via the type strings 'prm', 'rrt', and 'sbl'.

To initialize the planner, call `MotionPlan.setEndpoints` with the start and goal configurations. To plan, call `MotionPlan.planMore` with the desired number of iterations. Continue calling it until `MotionPlan.getPathEndpoints` returns non-`None`.

10.4. SIMULATION AND CONTROL

The simtest.py program is an entry point to fast prototyping of controllers using the Python API. The control folder contains several simple controllers that may be built upon in more sophisticated applications.

Like the compiled SimTest, simtest.py simulates a world file and possibly robot trajectories. The user interface is a simplified SimTest, with 's' beginning simulation and 'm' saving frames to disk. Right-dragging applies spring forces to the robot.

simtest.py also accepts arbitrary feedback controllers given as input. To do so, give it a .py file with a single `make(robot)` function that returns a controller object. This object should be an instance of a subclass

`BaseController` in control.controller. For example, to see a controller that interfaces with ROS, see control/roscontroller.py.

A Python controller is a very simple object with three major methods:

1. `output(self,**inputs)`: given a set of named inputs, produce a dictionary of named outputs. The semantics of the inputs and outputs are defined by the caller.
2. `advance(self,**inputs)`: advance by a single time step, performing any necessary changes to the controller's state. *Note: `output` should NOT change internal state!*
3. `signal(self,type,**inputs)`: sends some asynchronous signal to the controller. The usage is caller dependent. (This method is never called directly by simtest.py.)

For simtest.py, the inputs to `output` and `advance` will be

- `t`: the current simulation time.
- `q`: the robot's current sensed configuration
- `dq`: the robot's current sensed velocity
- `qcmd`: the robot's current commanded configuration
- `dqcmd`: the robot's current commanded configuration
- The names of each sensors in the simulated robot controller, mapped to a list of its measurements.

simtest.py expects the dictionary returned by output to contain either:

- `qcmd`: use PI control.
- `qcmd` and `dqcmd`: use PID control.
- `qcmd`, `dqcmd`, and torquecmd: use PID control with feedforward torques.
- `dqcmd` and `tcmd`: perform velocity control for time tcmd.
- `torquecmd`: use torque control.

Internally the controller can produce arbitrarily complex behavior. Several common design patterns are implemented in control/controller.py:

- `TimedControllerSequence`: runs a sequence of sub-controllers, switching at predefined times.
- `MultiController`: runs several sub-controllers in parallel, with the output of one sub-controller cascading into the input of another. For example, a state estimator could produce a better state estimate `q` for another controller.
- `ComposeController`: composes several sub-vectors in the input into a single vector in the output. Most often used as the last stage of a `MultiController` when several parts of the body are controlled with different sub-controllers.
- `LinearController`: outputs a linear function of some number of inputs.
- `LambdaController`: outputs `f(arg1,…,argk)` for any arbitrary Python function `f`.
- `StateMachineController`: a base class for a finite state machine controller. The subclass must determine when to transition between sub-controllers.
- `TransitionStateMachineController`: a finite state machine controller with an explicit matrix of transition conditions.

A preliminary velocity-based operational space controller is implemented in control/OperationalSpaceController.py, but its use is highly experimental at the moment.

10.5. UTILITIES AND DEMOS

The Python/utils and Python/demos folders contain a few example utilities and programs that can be built upon to start getting a flavor of programming Klamp't applications in Python.

- demos/gltest.py: a simple simulation with force sensor output.
- demos/gltemplate.py: a simulation with GUI hooks and mouse-clicking capabilities.
- demos/test.py: assorted, disorganized tests.
- utils/multipath_to_path.py: simple script to convert a `MultiPath` to a timed milestone trajectory. Parameters at the top of the script govern the speed of the trajectory.
- utils/multipath_to_timed_multipath.py: simple script to convert a `MultiPath` to a timed `MultiPath`. Parameters at the top of the script govern the speed of the trajectory.

11. FREQUENTLY ASKED QUESTIONS (FAQ)

11.1. SHOULD I LEARN THE PYTHON BINDINGS OR C++?

This is mostly a matter of preference. Python tends to be cleaner, easier to use, and faster for prototyping. However, the Python bindings providing a strict subset of the C++ functionality.

11.2. HOW DO I SET UP SENSORS IN THE SIMULATOR AND READ THEM?

Sensors are set up in the world XML file, not the robot file. See Section 8.2 for more details, and see data/hubo_plane.xml for an example.

To read sensors in C++, declare a variable `vector<double> measurements` and call `WorldSimulation.controlSimulators[robotIndex].sensors.GetNamedSensor(sensorName)->GetMeasurements(measurements);`

To read sensors in Python, call `Simulator.getController(robotIndex).getNamedSensor(sensorName).getMeasurements()`.

11.3. MY SIMULATOR GOES UNSTABLE AND/OR CRASHES. HELP!

There are two reasons that the simulator may go unstable: 1) the simulated robot is controlled in an inherently unstable manner, or 2) rigid body simulation artifacts due to poor collision handling or numerical errors. The second reason may also cause ODE to crash, typically on Linux systems. In testing we have found that configuring ODE with double precision fixes such crashes.

Unstable robot: an unstably controlled robot will oscillate and jitter, and if these oscillations become violent enough they may also cause rigid body simulation instability/crashing. If the robot goes unstable, then its PID constants and dryFriction/viscousFriction terms need to be tuned. These values must be set carefully in order to avoid oscillation and, ideally should be calibrated against the physical motors' behavior. This is currently an entirely manual process that must be done for every new robot. As a rule of thumb, large PID damping terms are usually problematic, and should be emulated as viscous friction.

Collision handling errors: Klamp't uses a contact handling method wherein each mesh is wrapped within a thin *boundary layer* that is used for collision detection. When objects make contact only along their boundary layers, the simulation is robust, but if their underlying meshes penetrate one another, then the simulator must fall back to less robust contact detection methods. This occurs if objects are moving too quickly or light objects in contact are subject to high compressive forces. If this happens, Klamp't will print a warning of the form "ODECustomMesh: Triangles penetrate margin X, cannot trust contact detector".

To avoid penetration, there are two remedies: 1) increase the thickness of the boundary layer, or 2) make the boundary layer stiffer. See Section 8 for more details on how to implement these fixes.

11.4. THE SIMULATOR RUNS SLOWLY. HOW CAN I MAKE IT FASTER?

Unless you are simulating a huge number of joints, the limiting steps in simulation are usually contact detection and calculating the contact response.

The speed of contact detection is governed by the resolution of the meshes in contact. Simpler meshes will lead to faster contact detection. Most 3D modeling packages will provide mesh simplification operators

The speed of contact response is governed by the number of contact points retained in the contact handling procedure after clustering. The maxContacts simulation parameter governs the number of clusters and can be reduced to achieve a faster simulation. However, setting this value too low will lead to a loss of physical realism.

11.5. HOW DO I IMPLEMENT A BEHAVIOR SCRIPT?

Many engineers and students tend to approach robots from a "scripting" approach, whereby a complex behavior is broken down into a script or state machine of painstakingly hand-tuned, heuristic behaviors. Unlike some other packages, Klamp't does not try to make scripting convenient. This choice was made deliberately in order to discourage the use of heuristic behaviors. The philosophy is that *hand-tuned behaviors should be rare in intelligent robots*.

To implement a behavior script in Klamp't, a controller should manually maintain and simulate the behavior of a state machine in its feedback loop. A framework for such controllers the `StateMachineController` class in Python/control/controller.py.

12. RECIPES (HOW DO I…?)

12.1. GENERATE A PATH/TRAJECTORY FROM KEYFRAMES

The easiest way to generate a path by hand is to define keyframes in the RobotPose program. To do so:

1. Use the poser to pose keyframes, and save these to the Resource Library using the "Poser -> Library" button. The keyframes will appear as `Config`'s. Name them appropriately (e.g., keyframe1,…, keyframeN) and save them to disk via the "Save File" button.
2. Concatenate all the .config files into one .configs file, e.g. using `cat keyframe1.config … keyframeN.config > keyframes.configs`.
3. Load the .configs file from disk, which gives a new `Config Set` resource in the Resource Library.
4. [optional] Set up any IK constraints in the poser that you wish the path to obey.
5. Click "Create Path" to generate a new interpolating path (untimed), or click "Optimize Path" to generate a new trajectory (timed). These will create a new `Multipath` resource in the Resource Library.
6. Name the Multipath and save it to disk.
7. [optional] If you prefer a linear path, you may select the Multipath, click "Convert" and type in "LinearPath" when prompted in the command line.

12.2. ANIMATE A VIDEO OF A PATH/TRAJECTORY

In RobotPose, paths/trajectories will be automatically animated when selected in the Resource Library. Run "`./RobotPose [world file] [path file]`" and select the path. Uncheck the "Draw geometry" button or move the poser robot away, then click the "Save Movie" button to begin saving PPM screenshots to disk. These files can then be processed into a video file using a utility like ffmpeg.

Note: to change the default movie size in RobotPose/SimTest, edit the `movieWidth` and `movieHeight` elements of robotpose.settings / simtest.settings.

Python API. You must manually interpolate and save image files to disk. The `GLProgram` class in the klampt.glprogram module has a `save_screen` method that uses the Python Imaging Library to save the current OpenGL view to disk. See demos/gltemplate.py for an example.

12.3. SIMULATE THE EXECUTION OF A KEYFRAME PATH

In SimTest, run "`./SimTest [world file] –config [start config file] –milestones [milestone path file]`". A milestone path file consists of a list of T configuration / velocity pairs:

N q1[0] … qN[0] N v1[0] … vN[0]
…
N q1[T] … qN[T] N v1[T] … vN[T]

To start and stop at each keyframe, set the velocities to zero.

Python API. Set up a simulator, then run:

```
for q in path:
```

```
        sim.robotControllers(0).appendMilestone(q)
```

This will start and stop at each keyframe. If keyframe velocities are given, run:

```
for (q,v) in path:
        sim.robotControllers(0).appendMilestone(q,v)
```

12.4. SIMULATE THE EXECUTION OF A TRAJECTORY

In SimTest, run "`./SimTest [world file] -config [start config file] -path [trajectory file]`".

Tips:

- For the most precise control over the trajectory, use a Linear Path file or a timed MultiPath. Otherwise, SimTest will do some processing to assign times and this may not generate the desired results. The Python/utils/multipath_to_timed_multipath.py script can be used to generate timing using a speedup/slowdown heuristic.
- To easily extract the start configuration from a MultiPath, run "`python Python/multipath.py -s [trajectory file] > start.config`".

Python API. In simtest.py, run "`./simtest.py [world file] [trajectory file]`".

Manual operation: Load a `Trajectory` object (see klampt/trajectory.py). During the control loop, read the simulation time (`sim.getTime()`), look up the configuration/velocity q/dq of the trajectory at that time using `(q,dq)=(traj.eval(t),traj.deriv(t))`, and then call `sim.getController(0).setPIDCommand(q,dq)`.

You may also use the `TrajectoryController` class in control/trajectory_controller.py.

12.5. IMPLEMENT A CUSTOM CONTROLLER

C++ API.

1. Create a new subclass of `RobotController` and override, at a minimum, the `Type` method to provide a name to the controller, and the `Update` method to read from the `sensors` member and write to the `command` member.
2. Add your controller to the factory by editing the `RobotControllerFactory::RegisterDefault` method in Control/Controller.cpp by calling RobotControllerFactory::Register(new MyController(robot))
3. Recompile SimTest.
4. Now you can set the robot's controller in the world XML file by setting the tag `<simulation><robot><controller type="MyControllerTypeString" />`.

Python API. See the Python/demos/gltemplate.py file for an empty method `control_loop` that provides a hook that gets called every `dt` seconds and should be used for interacting with the controller.

Alternatively, if you wish to follow the standardized control API in the Python/control module, please see Section 10.4.

12.6. PROCESS CLICKS ON THE ROBOT OR WORLD

C++ API. The `WorldViewWidget` class in Main/WorldViewProgram.h provides the `Hover` method to determine the closest object and robot when clicked via the mouse's x-y position. This must be provided the current OpenGL viewport (i.e., the `viewport` member of the `GLUTNavigationProgram` or `GLUINavigationProgram` classes).

Python API. See the Python/demos/gltemplate.py file for a routine `click_world` that will return a depth-sorted list of objects clicked at the mouse's x-y position.

13. GENERAL RECOMMENDATIONS

- Ask questions and report issues/bugs. This will help us make improvements to Klamp't. If you write a piece of code that you think will be useful to others, consider making it a contribution to the library.
- Practice self-documenting code. Name files, functions, classes, and variables descriptively. Comment as you go.
- Use *visual debugging* to debug your algorithms. For example, output intermediate configurations or paths to disk and inspect them with the RobotPose program.
- *Think statefully*. Decompose your programs into algorithms, state, parameters, and data. State is what the algorithm changes during its running. Parameters are values that are given as input to the algorithm when it begins (arguments and settings), and they do not change during execution. Data is the knowledge available to the algorithm and the information logged as a side effect of its execution.
- When prototyping long action sequences, build in functionality to save and restore the state of your system at intermediate points.

14. WISH LIST

Klamp't is an evolving project and we hope to grow and refine it in the future with the help of others. Future development of Klamp't will focus on the following items (in no particular order):

- Comprehensive GUI redesign with a better GUI package, e.g., Qt
- More convenient manipulation support in contact mechanics routines
- Convenience routines for easier motion planning
- Unification of locomotion and manipulation planning
- Specifying and solving optimization and optimal control problems
- Refinement of sensors and state estimators
- Expansion of the Python API (e.g., contact modeling, trajectory optimization)
- Binding with Open Motion Planning Library, Move It, and/or the Python Task and Motion Library (PyTAMP).

15. PAPERS AND PROJECTS USING KLAMP'T

- TeamHubo in the DARPA Robotics Challenge: http://dasl.mem.drexel.edu/DRC/
- K. Hauser. *Robust Contact Generation for Robot Simulation with Unstructured Meshes.* International Symposium of Robotics Research, 2013.
- K. Hauser. *Fast Interpolation and Time-Optimization on Implicit Contact Submanifolds*. Robotics: Science and Systems, 2013.
- K. Hauser. *On Responsiveness, Safety, and Completeness in Real-Time Motion Planning*. Autonomous Robots, 32(1):35-48, 2012.
- Y. Zhang, J. Luo, and K. Hauser. *Sampling-based Motion Planning with Dynamic Intermediate State Objectives: Application to Throwing*. In proceedings of IEEE Int'l Conference on Robotics and Automation (ICRA), May 2012.
- E. You and K. Hauser. *Assisted Teleoperation Strategies for Aggressively Controlling a Robot Arm with 2D Input*. In proceedings of Robotics: Science and Systems (RSS), Los Angeles, USA, June 2011.
- K. Hauser. *Adaptive Time Stepping in Real-Time Motion Planning*. In Algorithmic Foundations of Robotics IX, Springer Tracts in Advanced Robotics (STAR), Springer Berlin / Heidelberg, vol 68, p215-230, 2010.
- K. Hauser. *Recognition, Prediction, and Planning for Assisted Teleoperation with Freeform Tasks*. In proceedings of Robotics: Science and Systems, July 2012.
- K. Hauser. *The Minimum Constraint Removal Problem with Three Robotics Applications*. In proceedings of Workshop on the Algorithmic Foundations of Robotics, June 2012.