KLAMP'T MANUAL V0.7

KRIS HAUSER

Document last updated: 3/30/2017

CONTENTS

1. What is Klamp't? 4
1.1. Features 4
1.2. Currently supported platforms 4
1.3. Comparison to related packages 4
1.4. What's new in v0.7? 5
1.5. Contributors 6
2. Downloading and building Klamp't 7
2.1. Linux-like Environments 7
2.2. Windows 8
3. Running Klamp't apps 11
3.1. Interacting with 3D worlds 13
3.2. Example files 16
3.3. Other Klamp't apps 16
4. Design philosophy 18
5. Modeling 19
5.1. Math 19
5.2. 3-D Geometry 20
5.3. Robots 22
5.4. Terrains 25

5.5. Rigid Objects 26

5.6. Worlds 26

5.7. Paths and Trajectories 26

5.8. Inverse Kinematics 28

5.9. Dynamics 30

5.10. Contacts 30

5.11. Holds, Stances, and Grasps 31

5.12. Resources and Resource Libraries 31

5.13. File Types 32

6. Simulation 39

7. Planning 41

7.1. Robot-level kinematic motion planning 41

7.2. Configuration Space kinematic motion planning 43

7.3. Time-optimal acceleration-bounded trajectories 44

7.4. Interpolation and time-optimization with closed-chain constraints (C++ only) 45

7.5. Randomized planning with closed-chain constraints 45

7.6. Time-scaling optimization (C++ only) 46

7.7. Real-time motion planning (C++ only) 46

8. Control 48

8.1. Actuators 48

8.2. Sensors 48

8.3. Controllers 49

8.4. State estimation 50

9. Controller Integration 52

9.1. Connecting an external controller to a Klamp't simulated robot 52

9.2. Controller communication protocol (CCP) 54

9.3. Connecting a Klamp't controller to a physical robot 55

9.4. Connecting a Klamp't Planner to a Controller 55
10. C++ Programming 57
11. Python Programming 57
11.1. The Klampt module 57
11.2. Sub-Modules 58
11.3. Visualization 59
11.4. Utilities and Demos 59
11.5 Experimental Controller API 60
12. Frequently Asked Questions (FAQ) 62
12.1. Should I learn the Python bindings or C++? 62
12.2. How do I set up sensors in the simulator and read them? 62
12.3. My simulator goes unstable and/or crashes. Help! 62
12.4. The simulator runs slowly. How can I make it faster? 62
12.5. How Do I implement a Behavior Script? 63
13. Recipes (How do I…?) 64
13.1. Generate a path/trajectory from keyframes 64
13.2. Animate a video of a path/trajectory 64
13.3. Simulate the execution of a keyframe path 65
13.4. Simulate the execution of a trajectory 65
13.5. Implement a custom controller 66
13.6. Process clicks on the robot or world 66
14. General recommendations 67
15. Wish list 68
16. Papers and projects using Klamp't 69

1. WHAT IS KLAMP'T?

Klamp't (Kris' Locomotion and Manipulation Planning Toolbox) is an open-source, cross-platform software package for robot modeling, simulating, planning, optimization, and visualization. It aims to provide an accessible, wide range of programming tools for learning robotics, analyzing robots, developing algorithms, and prototyping intelligent behaviors. It has particular strengths in robot manipulation and locomotion.

Historically, it began development at Indiana University since 2009 primarily as a research platform. Beginning in 2013 it has been used in education at Indiana University and Duke University. Since then, it has been adopted by other labs around the world, such as Stanford, University of Pisa, and Worcester Polytechnic Institute. It has been used in several real-world projects, including the DARPA Robotics Challenge, Amazon Picking Challenge (2015-2016), the IROS 2016 Robot Grasping and Manipulation Challenge, and Stanford's SimGrasp toolbox.

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

- Unified C++ and Python package for robot modeling, kinematics, dynamics, control, motion planning, simulation, and visualization.
- Supports legged and fixed-based robots.
- Interoperable with Robot Operating System (ROS) and Open Motion Planning Library (OMPL).
- 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. This helps simulate uncertainty and modeling errors.
- 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 cameras, depth sensors, laser range finders, gyroscopes, force/torque sensors, and accelerometers.

1.2. CURRENTLY SUPPORTED PLATFORMS

- *nux environments
- Windows
- 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 middleware system designed for distributed control of physical robots, and Klamp't is designed to be interoperable with it. Various ROS software packages can replicate many of the functions of Klamp't when used together (Gazebo, KDE, Rviz, MoveIt!), but this approach is difficult since these tools are not as tightly integrated as they are 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, has a steep learning curve especially for non-CS students, and is also 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 planning 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. OpenRAVE is no longer actively supported.
- **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 limited functionality for modeling, planning, and optimization. Klamp't also has improved mesh-mesh collision handling that makes collision handling much more stable.

1.4. WHAT'S NEW IN V0.7?

Version history:

0.7 Latest version (3/24/2017)

- Improved simulation stability, including adaptive time stepping and instability detection/recovery.
- The proprietary .tri geometry file format has been replaced with the Object File Format (OFF) for better compatibility with 3D modeling packages.
- Simulated visual, depth, and laser range sensors are now fully supported.
- ROS sensor simulation broadcasting is enabled in Klampt/IO/ROS.h.
- World XML files can now be saved to disk.
- Robot sensors and controllers can be attached directly to a robot model using the `sensors` / `controller` properties in the robot's .rob or .urdf file.
- The motion planning structure in KrisLibrary has been completely revamped in preparation for support of optimal and kinodynamic planning, but this should be a mostly transparent change to Klamp't users.
- The Python interface is now better organized. *However, the module structure is incompatible with code developed for versions 0.6.2 and earlier*. In particular, math modules (`vectorops`, `so3`, `se3`) are now in the `klampt.math` subpackage, and visualization modules (`glprogram`, `glrobotprogram`, etc) are now in the `klampt.vis` subpackage.
- Custom Python simulations of sensors, actuators, and force appliers that work on fast simulation rates are easier to integrate with slower control loops in the `klampt.sim.simulation` module.
- Revamped and enhanced Python visualization functionality in the `klampt.vis` module. Multiple windows, simultaneous viewports, trajectory visualization, custom in-visualization plotting, automatic viewport determination, and thumbnail saving are now supported.
- Cartesian trajectory generation, file loading utilities are added to Python.

0.6.2 (7/31/2016)

- New Python APIs for visualization
- Geometry caching helps load times and memory usage for large scenes
- A global IK solver has been added to the Python API
- ROS broadcasting / subscribing is enabled in the C++ API.

0.6.1 (3/21/2015)

- Added functionality in Python API to load/save/edit resources, manipulate transforms and robot configurations via widgets, change appearance of objects, and run programs through Qt.
- Removed the Python collide module. All prior functionality is now placed in the Geometry3D class in the standard klampt module.
- Real-time planning interface has been greatly simplified.
- The MilestonePathController class will be deprecated, use PolynomialPathController instead.
- Minor bug fixes

0.6. (7/4/2014)

- CMake build system makes it easier to build across multiple platforms
- Easy connections with external controllers via ROS or a serial protocol
- More user-friendly Qt application front ends
- More demos, example code, and tutorials
- Direct loading of URDF files with <klampt> XML tag
- More calibrated robots: Baxter, RobotiQ 3-finger adaptive gripper
- Unification of locomotion and manipulation via the GeneralizedRobot mechanism
- Fixed build for Cygwin
- More sophisticated logging capabilities in SimTest (contacts, commanded/actual/sensed paths)
- Miscellaneous debugging throughout

0.5. Initial release (11/17/2013)

1.5. CONTRIBUTORS

Kris Hauser has been the primary maintainer throughout the project. Other major contributors include Jordan Tritell, Jingru Luo, and Alessio Rocchi.

Adam Konnecker, Cam Allen, Steve Kuznetsov have helped with the Mac build.

As an open-source project, we welcome contributions and suggestions from the community.

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:

- CMake (version >= 2.6)
- GLUT
- GLPK, the GNU Linear Programming Kit
- Python, if you wish to use the Python bindings (tested only on Python 2.6 & 2.7).
- Boost C++ Libraries
- (recommended) Assimp, if you wish to load STL, DAE and other geometry file formats. (Only OBJ and OFF are natively supported in Klampt.)
- (recommended) Qt4, if you wish to use nicer GUIs for the core applications.
- (recommended) PyOpenGL is required for visualization (https://pypi.python.org/pypi/PyOpenGL/3.0.2). Qt4 and PyQt are optional for scripted resource editing. Python Imaging Library (PIL) is required for saving screenshots to disk.

2.1. LINUX-LIKE ENVIRONMENTS

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

To configure the dependencies, consider the following notes:

- KrisLibrary may need to be configured for your particular system. Try running `cmake-gui` and changing the Advanced variables.
- 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/Library/Makefile. *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 CMake cache.*

Enabling Assimp support (optional). To load a larger variety of 3D meshes, Klamp't can be configured to use the Asset Importer (Assimp) library. Once Assimp 3.0.1270 is installed on your system (if Klampt/Library/assimp--3.0.1270-sdk or /usr/lib/libassimp.so exists), KrisLibrary and Klampt should automatically detect it when built.

Run CMake to build Klamp't Makefiles. Run "`cmake .`" to build the Klamp't makefiles.

Building static library and apps. The static library is built using '`make Klampt`'. 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/klampt can be built using "`make python`". To install the `klampt` module into your Python package, type "`make python-install`".

IMPORTANT: You must set up Python to be able to find the shared library files for external dependencies. Otherwise, you will get errors importing the `_robotsim` module when calling `import klampt`. To do this, you may either:

1. Set the LD_LIBRARY_PATH environment variable to include the locations of the TinyXML, ODE, and (optionally) Assimp shared libraries. These will be .so (or DLL) files.
2. OR move the shared library files into your shared library path
3. OR on Linux-like systems, edit /etc/ld.so.conf as appropriate and then run ldconfig (as sudo).

Platform-specific install scripts

These commands work from a clean install of Ubuntu 12.04

```
sudo apt-get freeglut3 freeglut3-dev glpk python-opengl
```
[Optional: to enable Assimp mesh importing, before calling any of the "make" calls, call
```
sudo apt-get install libassimp-dev]
cd Klampt
cd Library
make unpack-deps
make deps
cd ..
cmake .
make all
sudo make python-install
```

Building documentation. To build the Klamp't C++ API documentation using Doxygen, type '`make docs`' in Klampt/. '`make python-docs`' will build the Python API documentation.

2.2. WINDOWS

Prebuilt binary executables and static libraries for VS2015 are available on the Klamp't website. Klamp't can also be built from source with Visual Studio 2012 (or Visual Studio 2010 SP1) and above.

Step by step instructions to install the C++ applications from binaries

From http://klampt.org, download and run the Win32 Klamp't installer. Note: If you plan to develop in the Klamp't C++ API, make sure to get the appropriate installer for your Visual Studio version.

Step by step instructions to install the Python API from binaries

1. Visit https://github.com/krishauser/Klampt and click "Clone on Desktop". Follow the on-screen instructions to clone the Klamp't Git repository.
2. Install Python 2.7.x from http://www.python.org/getit/. *Make sure to get the Win32 version even if you have a 64-bit machine.*

3. Add C:\Python27 to your PATH environment variable. (Right click My Computer -> Properties -> Advanced System Settings -> Environment Variables and append ‘`;C:\Python27`’ to the PATH variable.)
4. Install PyOpenGL from https://pypi.python.org/pypi/PyOpenGL/3.0.2 using the Win32 installer.
5. Install the glut32.dll file from http://user.xmission.com/~nate/glut.html into your SysWOW64 directory (if your machine is 64-bit, most newer machines) or System32 directory (for older 32-bit machines).
6. From http://klampt.org, download and install the Win32 Klamp't Python 2.7 bindings.
7. Done. As a test, run ‘cmd’ from the start menu, change directories to Klampt/Python/demos, and run `python gltemplate.py ../../data/athlete_fractal_1.xml`.

To build your own C++ applications that link to Klamp’t

1. Follow the instructions to install the C++ applications from binaries.
2. Clone the KrisLibrary Git repository from https://github.com/krishauser/KrisLibrary to the Klampt/Library folder as the target location.
3. From http://klampt.org, download the appropriate Win32 Klamp’t dependencies for your Visual Studio version (both Release and Debug are recommended). Unpack into Klampt/Library.
4. In your own CMake project, set KLAMPT_ROOT and BOOST_ROOT to the appropriate paths and put the following lines into your CMakeLists.txt (along with whatever other lines are needed to build your project)
```
SET (CMAKE_MODULE_PATH "${KLAMPT_ROOT}/CMakeModules")
FIND_PACKAGE(Klampt REQUIRED)
ADD_DEFINITIONS(${KLAMPT_DEFINITIONS})
INCLUDE_DIRECTORIES(${KLAMPT_INCLUDE_DIRS})
TARGET_LINK_LIBRARIES(MyApp ${KLAMPT_LIBRARIES})
```
5. Build your project in standard CMake fashion.
6. [Note: you may need to set the cmake variable BOOST_ROOT to reflect your Boost installation path using the command line option “-DBOOST_ROOT=/path/to/boost” or via adding BOOST_ROOT in cmake-gui.]

Building Klamp’t from source. After following the instructions under the heading “To build your own C++ applications that link to Klamp’t”, the standard CMake procedure should generate appropriate Visual Studio project files.

Building Python bindings from source. (tested with Python 2.7, Win32) The standard CMake procedure should generate Visual Studio project files for the project “python-install” but these are broken. Instead, download the Windows Python setup.py file from http://klampt.org and copy it to the Klampt/Python directory. Edit the paths at the top of the file to reflect your computer’s file structure. Finally, open a Visual Studio Command Prompt in *Administrative Mode*, and depending on your VS version, run:

VS 2008:

```
python setup.py install
```

VS 2010:

```
set VS90COMNTOOLS=%VS100COMNTOOLS%
python setup.py install
```

VS 2012:

```
set VS90COMNTOOLS=%VS110COMNTOOLS%
python setup.py install
```

VS 2015:

```
set VS90COMNTOOLS=%VS140COMNTOOLS%
python setup.py install
```

Building dependencies from source. If you wish to build dependencies from scratch, Visual Studio project files are available. Make sure to place all compiled library (.lib) files in the Klampt/Library folder. All libraries should be built in Win32 mode, with C++ code generation set to Multithreaded DLL / Multithreaded Debug DLL.

Note: when building KrisLibrary you may need to set the cmake variable BOOST_ROOT to reflect your Boost installation path using the command line option "-DBOOST_ROOT=/path/to/boost" or via adding BOOST_ROOT in cmake-gui.

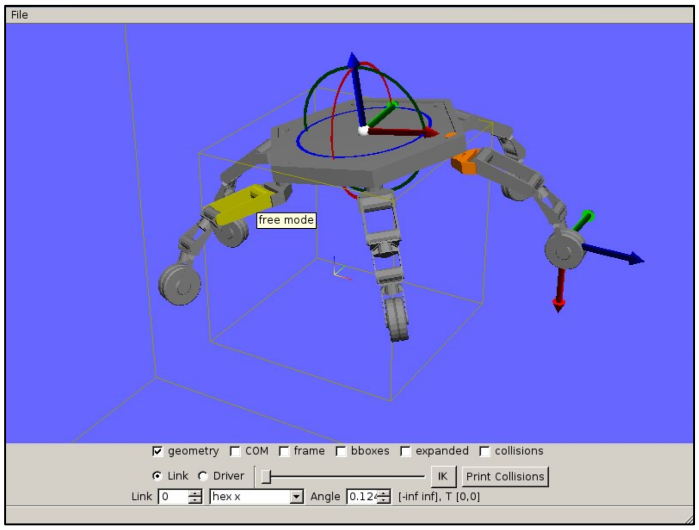
The general procedure is as follows:

1. Acquire Boost, GLUT and optionally (but recommended) WinGLPK 4.61 and/or Assimp 3.0.1270. Place the glut32.lib, glew32.lib, glpk_4_61.lib files in Klampt/Library or in your Visual Studio path. Place the Assimp folder in Klampt/Library.
2. Configure and edit dependencies as follows:
    1. GLUI: Visual Studio will complain about template instantiations inside class definitions in glui.h; simply put these in the global namespace. Also, if you are using GLUI rather than Qt4, due to Visual Studio's string range checking, GLUI will throw an assertion in Debug mode when an EditText is created. To fix this, you will have to add several checks similar to this: `if(text.empty()) return 0;` in glui_edittext.cpp.
    2. ODE: Set up build files with premake4 vs2010.
3. Compile all dependencies except for KrisLibrary. Place all generated .lib files into the Klampt/Library directory.
    1. ODE: compile in double precision, Static.
    2. GLUI: compile as usual.
    3. TinyXML: compile with STL support.
4. Compile KrisLibrary last. CMake files are available for compiling KrisLibrary with/without Assimp support and with/without GLPK support. You may need to do some editing of the BOOST directories using CMake-GUI depending on how you built Boost.
5. After compiling, all of the .dll files associated with dependency libraries should be placed in the appropriate Klamp't binary folders.

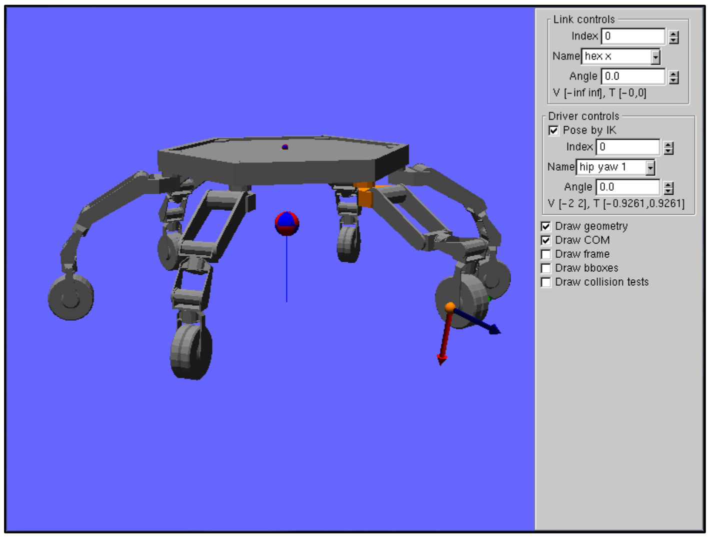
3. RUNNING KLAMP'T APPS

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

```
./RobotTest robot_file
```



(a)



(b)

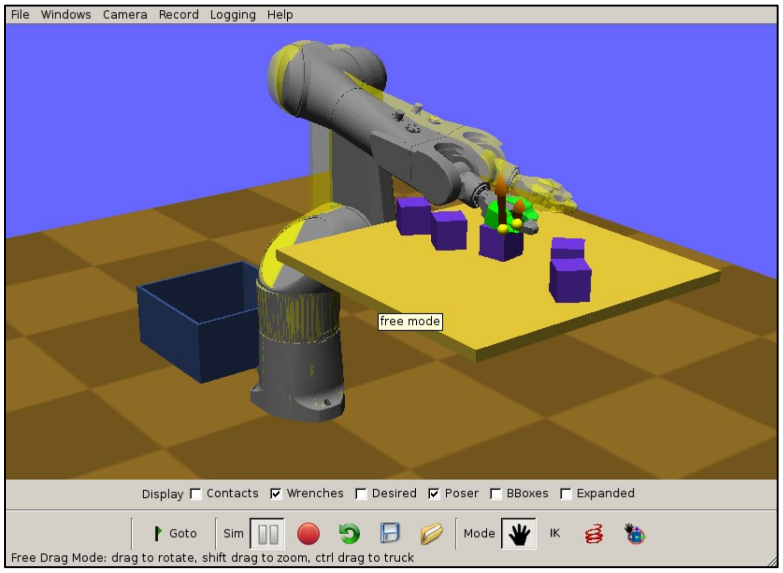
Figure 1. The RobotTest GUI ((a) Qt version, (b) GLUI version).

```
./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]
```

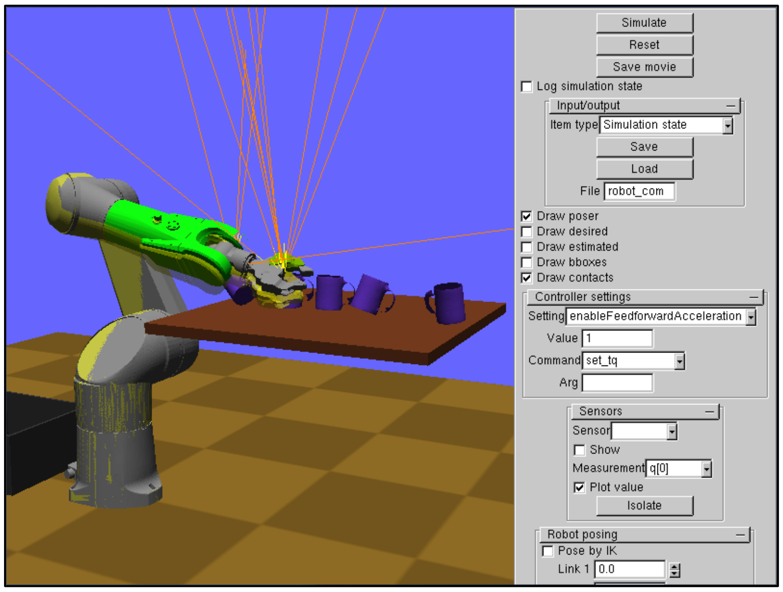
(e.g., ./SimTest data/robots/athlete.rob data/terrains/plane.env or ./SimTest data/hubo_plane.xml)



Display ☐ Contacts ☑ Wrenches ☐ Desired ☑ Poser ☐ BBoxes ☐ Expanded

Free Drag Mode: drag to rotate, shift drag to zoom, ctrl drag to truck

(a)



☐ Log simulation state

Input/output

Item type Simulation state

File robot_com

☑ Draw poser
☐ Draw desired
☐ Draw estimated
☐ Draw bboxes
☑ Draw contacts

Controller settings

Setting enableFeedforwardAcceleration

Value 1

Command set_tq

Arg

Sensors

Sensor

☐ Show

Measurement q[0]

☑ Plot value

Robot posing

☐ Pose by IK

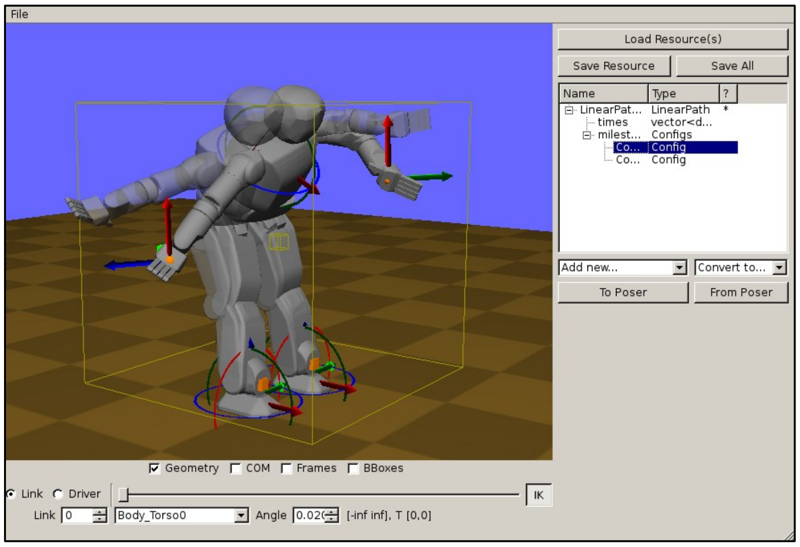
Link 1 0.0

(b)

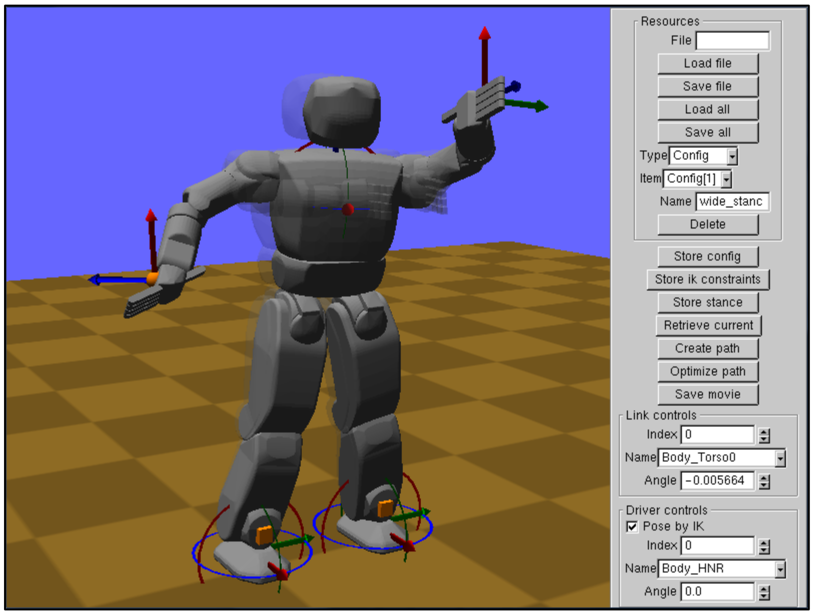
Figure 2. The SimTest GUI, (a) Qt version, (b) GLUI version. The transparent yellow robot is the "poser". Contact forces are drawn in orange.

```
./SimTest data/tx90cups.xml
```

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



(a)



(b)

Figure 3. The RobotPose GUI ((a) Qt version, (b) GLUI version). 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 (GLUI version)

- *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 "Go To" (Qt) or "Set Milestone" (GLUI) button will send the posed configuration to the controller.
- Typing 's' or clicking the green arrow (Qt) or "Simulate" (GLUI) button toggles the simulation.
- Typing 'a' advances by one simulation step (1/100 s).
- Clicking the red circle (Qt) or the "Save movie" button (GLUI) 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. In Qt, the movie-making command and output file can be edited by selecting "Change Encoder…" and "Change file…" from the "Record" menu, respectively. The resolution can also be set from the "Record" menu. In GLUI, the movie-making command must be executed manually, and the movie resolution can be changed by setting the `movieWidth` and `movieHeight` parameters in simtest.settings (JSON format).
- Clicking the red spring icon (Qt) or typing 'f' (GLUI) 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.
- The "Save View" (Qt) or typing lowercase 'v' (GLUI) saves the current viewport to disk, and "Load View" (Qt) or typing uppercase 'V' (GLUI) loads the previously saved viewport. This is useful for creating side-by-side comparison videos.

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 / saved from disk via the "Load [X]" / "Save [X]" buttons at the top.
- Qt Version:
    - Resources in the resource tree can be expanded, dragged, and copied (Shift+drag) using the mouse.
    - The status indicators in the resource tree are as follows:
        - * indicates that the resource has been modified since loading.
        - @ indicates that sub-resources have been modified, and RobotPose has not yet merged the modifications into the top-level resource. (Click on the top-level resource to attempt the merge)
        - ! indicates that a prior merge was unsuccessful. For example, a Linear Path may not have the same number of times as configurations. Correct the error and try again.
    - The "Add Resource…" dropdown allows creating new resources.
    - The "Convert to…" dropdown allows resources to be converted to similar types.
    - "To poser" sends the currently selected resource to the poser. Works with "Config", "IKGoal", "Hold", and "Stance".
    - "From poser" overwrites the currently selected resource using its value in the poser. Works with "Config", "IKGoal", "Hold", and "Stance".
    - When Configs, Linear Path, or MultiPath resources are selected: "Optimize Path" generates and optimizes a trajectory along the currently selected resource, minimizing execution time under the robot's velocity and acceleration bounds.
- GLUI Version:
    - "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 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 and problem setups 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.
- baxter*.xml: the Rethink Robotics Baxter 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).
- atlas.rob: the Boston Dynamics ATLAS robot.
- cartpole.rob: a 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 –config 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. Settings for geometry import/export can be changed by editing urdftorob.settings.

  Klamp't-specific parameters (e.g., ignored self collisions, servo gains) are given default values. To change these parameters, the .rob file must be edited or the `<klampt>` element may be edited as described in

Section 5.13.

To clean up extraneous self-collision checks, the Print Self Collisions button of the RobotPose program can be used. The MotorCalibrate program may be run to fix up the servo gain and friction parameters.

- MotorCalibrate generates motor simulation parameters given example commanded and sensed trajectories. It runs a quasi-Newton optimization with random restarts to match the simulated values to the sensed parameters as closely as follows.

  To use it, 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 latter lines (beginning with servoP) should be copied into the .rob or .urdf file.

  An example optimization is given by running
  `./MotorCalibrate Examples/motorcalibrate_baxter.settings.`
  Multiple runs of this process, possibly with different initial conditions, should generate better matches to the sensed data.

- 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.

- Merge combines multiple robot and object files into a single robot file.

- SimUtil is a command line interface to the simulator.

Klamp't also contains several example applications in Klampt/Examples:

- Cartpole demonstrates generation of optimal control tables for two toy dynamic systems – a pendulum swing-up and a cart-pole balancing task.
- PlanDemo is a command line kinematic motion planner for collision-free motion between configurations.
- ContactPlan is a command line kinematic motion planner for collision-free, stable motion in contact between configurations.
- RealTimePlanning demonstrates real-time planning between randomly generated target configurations.
- 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.

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, *User interface*, 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/
- **User interface:** Klampt/Interface/
- **I/O:** native I/O is mostly embedded into models. Import/export to XML world files, ROS, and other 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, and includes robot kinematics, dynamics, simulation, and geometry representations. Also includes low-level IK solving and motion planning modules.
- Klampt/Python/klampt/math: basic 3D geometry.
- Klampt/Python/klampt/modeling: other modeling, including IK, trajectories, Cartesian interpolation, and sub-robot indexing. Setting and getting "configurations" for many objects.
- Klampt/Python/klampt/plan: motion planning for robots.
- Klampt/Python/klampt/sim: more advanced simulation functionality, such as logging and custom actuator and sensor emulation.
- Klampt/Python/klampt/io: Unified I/O for all types of Klamp't objects. Supports JSON formats for some objects as well. Resource loading, saving, and visual editing.
- Klampt/Python/klampt/vis: Visualization.
- Klampt/Python/control: custom control modules.
- Klampt/Python/demos: demonstrations about how to use various aspects of the Python klampt API.
- Klampt/Python/exercises: exercises for implementing basic concepts in Klamp't.
- Klampt/Python/utils: utility programs.

5. MODELING

5.1. MATH

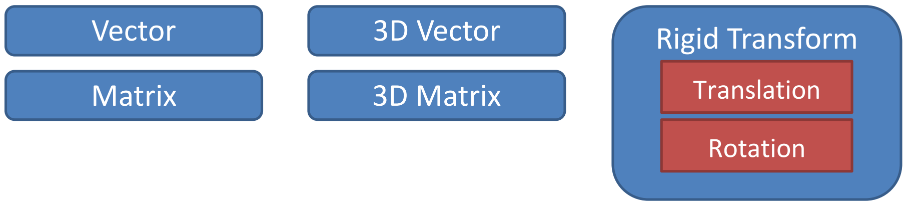


Figure 4. The Math concept model.

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.

C++ API. Users should become familiar with the 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.

Python API. 3D math operations are found in the klampt.math module under the following files.

- vectorops: basic vector operations on lists of numbers.
- so2: routines for handling 2D rotations.
- so3: routines for handling 3D rotations.
- se3: routines for handling 3D rigid transformations

The use of numpy / scipy is recommended if you are doing any significant linear algebra. More information can be found in the Klampt Math Tutorial.

5.2. 3-D GEOMETRY

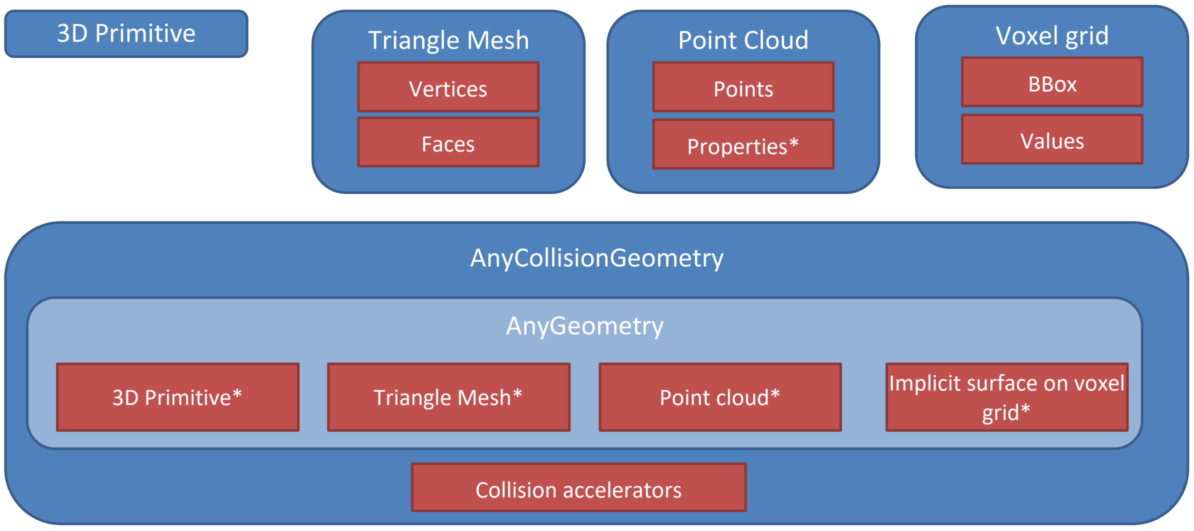


Figure 5. The 3D geometry concept model.

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

Geometry data, collision geometries. Klamp't supports a variety of geometry data including 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. Collision geometries have a *current transformation* that sets where they exist in space, and is used for collision testing. 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 Object File Format (OFF), which is a simple ASCII file format. Klamp't also natively supports OBJ file format. If Klamp't is compiled with Assimp support, it can also load a variety of other formats including STL, DAE, VMRL, etc. Point clouds can be loaded from PCD files (v0.7), as specified by the Point Cloud Library (PCL).

Geometry caching. When multiple objects load the same geometry file, Klamp't uses a caching mechanism to avoid reloading the file from disk and re-creating collision acceleration structures. This is essential for loading very large scenes with many replicated objects. However, when geometries are transformed by API calls, they are removed from the cache. So, to achieve maximum performance with many duplicated geometries, it is recommended to transform the geometry files themselves in advance rather than dynamically through the API.

*C++ API***.** Geometry data is stored in the `AnyGeometry3D` type and collision geometries are stored in the `AnyCollisionGeometry3D` type. These are essentially container types that abstract the underlying geometry and collision acceleration data structures. To operate on the data therein, users will need to inspect the geometry's type and cast to the appropriate type. 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 they are abstract and may be extended to handle more geometry representations in the future.

*Python API***.** The `Geometry3D` class in klampt module allows collision testing between geometries. All the standard Klamp't geometry types (geometric primitives, triangle meshes, point clouds) are supported. Prototypes and documentation are defined in klampt/src/geometry.h.

For convenience, the klampt.model.collide module provides utility functions for checking collision with sets of objects, as well as 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 `WorldCollider` 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.

5.3. ROBOTS

Klamp't works with arbitrary tree-structured articulated robots. Robot models provide the following functions

- Describes a list of links with their parents (an open linkage, specified in topologically sorted order)
- 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 Klamp't-specific .rob files or more widely-used URDF files. These are simple text files that are editable by hand.

Although URDF is more commonly used, there are some convenient aspects of .rob files that may be useful. For example, the `mount` command allows robot grippers and other attachments to be added automatically at load-time.

The basic URDF file format does not specify some aspects of Klamp't robots. These can be added under the `<klampt>` XML tag. See the documentation below or the Klampt import robot tutorial for more details.

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.

The URDFtoRob program converts from .urdf to .rob files. Geometric primitive link geometries will be converted to triangle meshes.

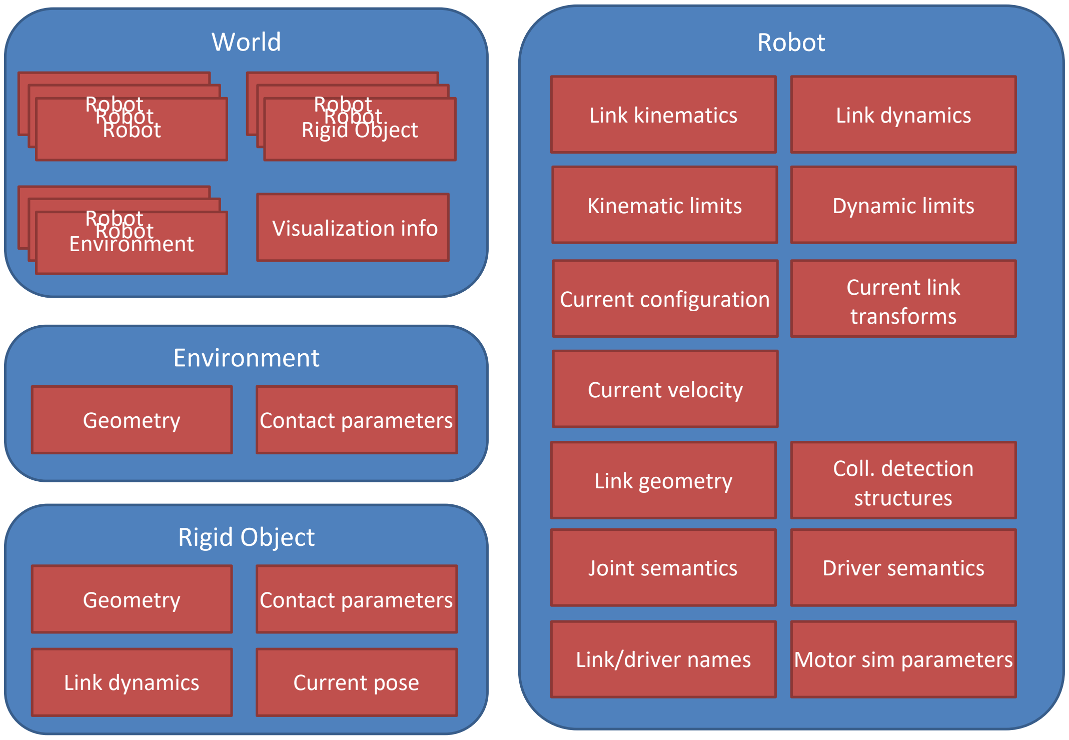


Figure 6. The Robot, Environment, Rigid Object, and World concept models.

C++ API. 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.

- `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.

Python API. The Python `RobotModel` class provides flat access to robot models.

Configurations. A robot configuration is a nonredundant description of the positions of each link of the robot, and is essentially an ordered list of numbers described by a `Config` object. Each entry in the configuration is a *degree of freedom* (DOF), which is usually movable but is sometimes fixed to a constant value.

The robot model contains a *current configuration*. It is important to *is not necessarily the current configuration of the simulated robot or an actual robot*, but is rather a temporary variable representing the controller/planner's mental state of where the robot might be posed.

C++ API. The `Config` class is simply typedef'ed as a `Vector` (see KrisLibrary/math/vector.h). The robot model's configuration is described in `Robot.q.` 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 simple assignment of the form `Robot.q=q` does not.

Python API. A `Config` object is simply a list of floating point numbers, and the robot model's configuration is retrieved / set using `RobotModel.setConfig(q)/ RobotModel.getConfig()`. Upon calling `setConfig()` the link transforms and geometries are automatically updated using forward kinematics.

Links. Links represent rigid coordinate frames that are connected to either another link or the world coordinate frame. Every degree of freedom of the robot has an associated Link. Links are named and numbered from 0 to #DOFs-1. Each link stores an index of its parent, and the parent index must be less than the link's index (topologically sorted order). A parent of -1 indicates that the link is attached to the world coordinate frame. Each link may be prismatic or revolute and moves along or around a link axis given by a 3D vector. Links also contain mass parameters, the reference transformation to its parent, and a (possibly empty) collision geometry, which is specified relative to the link's coordinate system. A link also stores a "current" world transformation which is calculated using forward kinematics.

C++ API. Links are stored in the `Robot.links` member, which is an array of `RobotLink3D`'s. The parent index of each link is stored in the `parents` member, which is a list of `int`'s. The link type is stored in `RobotLink3D.type` and its axis is stored in `RobotLink3D.w`. `RobotLink3D` also contains mass parameters (`mass`, `inertia`, `com`), the reference transformation to its parent (`T0_Parent`), and the link's current transformation `T_World`.

Link geometries are stored in the `Robot.geometry` variable (but to take advantage of the cache the `Robot.geomManagers` variable should be used for saving/loading/modifying the geometry). *The collision geometry transform is only updated to the current link transform after robot.UpdateGeometry() is called*.

Python API. References to links are retrieved using the `RobotModel.link(index or name)` method. Kinematic information can be retrieved via `get/setParent()`, `get/setAxis()`, and `get/setParentTransform()` (changing from revolute to prismatic types is not supported at the moment). The current world transformation is retrieved via `get/setTransform()`.

A reference to the link's geometry is retrieved via the `geometry()` method. Geometry current transforms are updated automatically after `RobotModel.setConfig(q)`.

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). 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.

C++ API. See `RobotKinematics3D.InitializeRigidObject` for an example of how to set up a floating base.

Python API. See klampt.model.floatingbase.py for utility functions for setting up a floating base.

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 with linear relationships between multiple DOF (such as certain cable drives or gear linkages) 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.

5.4. TERRAINS

A `Terrain` 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).

C++ API. See Klampt/Modeling/Terrain.h.

Python API. See the `TerrainModel` class.

5.5. RIGID OBJECTS

A `RigidObject` 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).

C++ API: See Klampt/Modeling/RigidObject.h.

Python API: See the `RigidObjectModel` class.

5.6. WORLDS

A World stores multiple named robots, terrain, and rigid objects, along with associated visualization information. Worlds are loaded from .xml files or created dynamically by loading individual elements. The world essentially stores three arrays containing robots, rigid objects, and terrains.

Entity names. Each entity in the world is named with a string identifier, which is ideally unique. If names are not unique, entities must be addressed by index. Furthermore, some modules like klampt.model.coordinate assume names are unique; if not, unexpected behavior may result.

Entity IDs. Each entity in the world, including each robot, robot link, rigid object, and terrain, can be addressed via a unique ID number. Note that this is not the same as an entity's index into the array containing it; the index is not unique when compared across entity types.

C++ API. See the `RobotWorld` class (Klampt/Modeling/World.h)

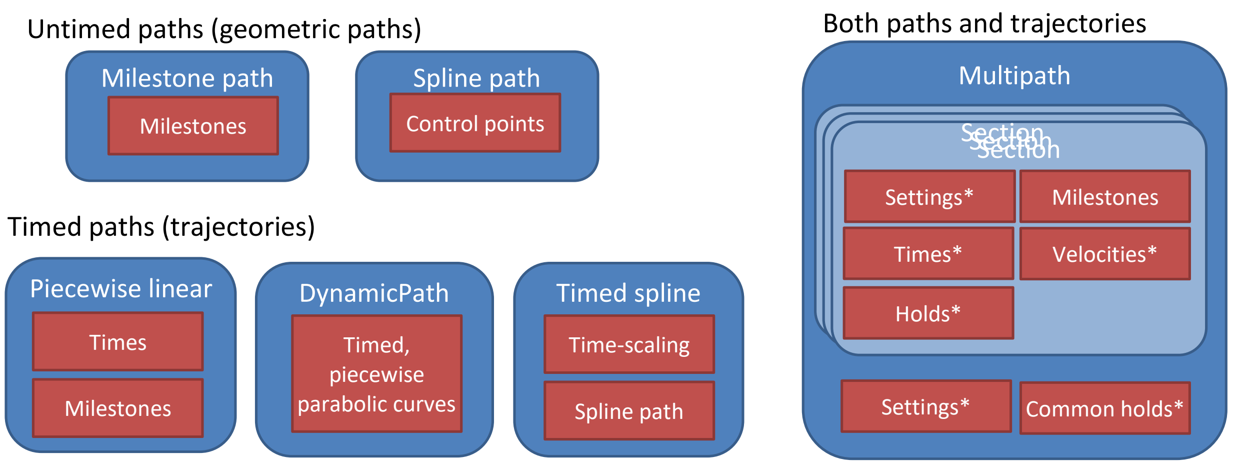


Figure 7. The Path concept models.

Python API. See the `WorldModel` class.

5.7. PATHS AND TRAJECTORIES

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 $y(s): [0,1] \to C$ while trajectories are expressed as continuous curves $y(t): [t_i, t_f] \to C$ 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 so a planner must somehow prescribe times to paths before executing them. 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 arrays of `Configs`. *Note: to properly handle a robot's rotational joints, milestones should be interpolated via robot-specific interpolation functions. Cartesian linear interpolation does not correctly handle floating and spin joints. See the functions in* Klampt/Modeling/Interpolate.h (C++) *and* `RobotModel.interpolate()` (Python) *to do so*.
- *Cubic splines (timed and untimed)*. Klamp't supports piecewise cubic curves. Routines for smooth spline interpolation of configuration lists are found in Klampt/Modeling/SplineInterpolate.h (C++) and Hermite spline interpolation in the `HermiteTrajectory` class in klampt.model.trajectory.

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 an array of `reals` listing points in time along with an array `Configs` describing the milestones reached at each of those points. [*See note above regarding interpolation*.] With a historical misnomer, these trajectories are given in the `LinearPath` class of Klampt/Modeling/Paths.h (C++). In Python they are given in the `Trajectory`, `SO3Trajectory`, `SE3Trajectory`, and `RobotTrajectory` classes of klampt.model.trajectory.
- *`DynamicPath` (piecewise parabolic curves).* These are time-optimal bounded-acceleration trajectories that include both configuration, velocity, and time. Routines in Klampt/Modeling/Paths.h or 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 (C++). Currently not implemented in Python.
- *Time-scaled cubic splines.* Found in the `TimeScaledBezierCurve` class in Klampt/Planning/TimeScaling.h (C++).

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 between most path types are supported in Klampt/Modeling/Paths.h (C++) and klampt.model.trajectory (Python).

Multipaths. A `MultiPath` 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. Details can be found in Klampt/Modeling/MultiPath.h (C++) and klampt.model.multipath (Python). The

Klampt/Python/klampt/model/multipath.py can also be run as a script to perform various simple transformations on `MultiPaths`.

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

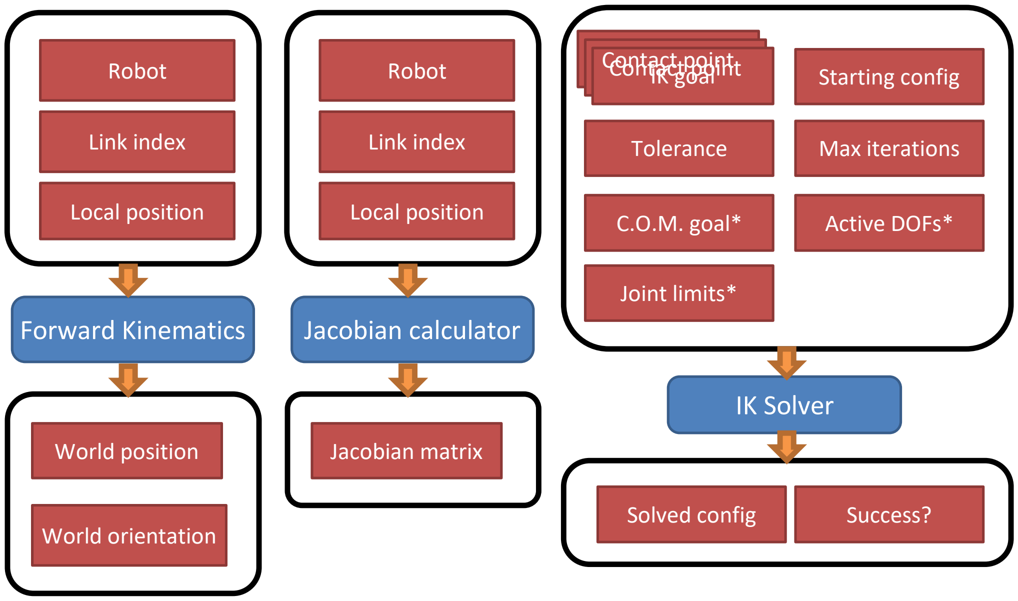


Figure 8. The Forward Kinematics and Jacobian subroutines (implemented in a Robot) and the IK solver subroutine

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

An `IKGoal` defines a constraint on a single link. The `link` member must be filled out prior to use and indicates the link's index on its robot. 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 (C++) and `setFixedPoints` method (Python) are 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 (C++) or `getRotation` `method (Python)`. To convert from a 3x3 matrix, call the `SetFixedRotation` method or `setFixedRotConstraint` method (Python).

IKGoals are implemented in KrisLibrary/robotics/IK.h (C++) and the IKObjective wrapper class in Klampt/Python/klampt/src/robotik.h (Python).

Numerical solvers. Numerical inverse kinematics solvers are extremely flexible and can solve for arbitrary combinations of IK constraints. 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.

C++ API. The `SolveIK()` functions in KrisLibrary/robotics/IKFunctions.h are the easiest way to solve IK constraints. For richer functionality, consult the documentation of the `RobotIKFunction` and `RobotIKSolver` classes and `Get*Dofs()` functions.

Python API. Convenient calls to IK solvers are found in the klampt.model.ik module. Functions for global inverse kinematics (using random restarts) and local inverse kinematics (limiting the amount of joint angle deviation) solving are available.

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

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

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++ API. 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.

Python API. The `RobotModel` class can compute each of these items using the Newton-Euler method.

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.

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

- 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.

Python API. These routines can be found in klampt.model.contact which are thin wrappers around the underlying C++ functions.

- A ContactPoint is either a frictionless or frictional point contact. Consist of a position, normal, and coefficient of friction. Unlike in C++, the ContactPoint data structure also contains which objects are in contact.
- forceClosure tests whether a given set of contacts is in force closure.
- comEquilibrium tests whether the center of mass of a rigid body can be stably supported against gravity by valid contact forces at the given contact list.
- supportPolygon computes a support polygon for a given contact list. Testing the resulting boundaries of the support polygon is much faster than calling comEqulibrium multiple times.
- equilibriumTorques solves for equilibrium of an articulated robot under gravity and torque constraints. It can handle both statically balanced and dynamically moving robots.

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 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. The C++ API defines them in Klampt/Contact/Hold.h, which also defines convenience setup routines in the Setup* methods. The Python API defines them in klampt.model.contact.

The C++ API also defines a couple additional classes. 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) are more sophisticated than holds and are 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. Resources are compatible with the RobotPose app, as well as the C++ and Python APIs.

Currently supported types include:

- `Config` (.config)
- `Hold` (.hold)
- `Stance` (.stance)
- `Grasp` (.xml)
- Configuration lists (.configs)
- `TriMesh` (.off, .tri, etc.)
- `PointCloud` (.pcd)
- `Robot` (.rob)
- `RigidObject` (.obj)
- `World` (.xml)
- Linear paths (.path)
- `MultiPath` (.xml)

Klamp't also support the following additional types which do not have a dedicated file extension:

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

C++ API. The Klampt/Modeling/Resources.h file lists all available resource types. Note that 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.

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.

Python API. The klampt.io.resource module allows you to easily load, save, or edit resources. Visual editing is supported for `Config`, `Configs`, `Vector`, and `RigidTransform` types. See the Python/demos/resourcetest.py demo for more examples about how to use this module.

5.13. FILE TYPES

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

- World files (.xml)
- Robot files (.rob)
- URDF files with Klamp't-specific elements (.urdf)
- 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)

World (.xml) files

Structure: an XML v1.0 file, containing robots, rigid objects, and terrains, as well as simulation parameters. Follows the following schema.

- `world`: top level element.

  Attributes
  - `background` (Vector4, default light blue): sets the RGBA background color of the world. Each channel has the range [0,1].

  - `robot`: adds a robot to the world.

    Attributes
    - `name` (string, optional, default "Robot"): a string to be used as an identifier.
    - `file` (string): the Robot (.rob) file to be loaded. May be relative or absolute path.
    - `config` (`Config`, optional): an initial configuration. Format: `N q1 … qN` where N is the number of DOF in the robot.

  - `rigidObject`: adds a rigid object to the world. If the `file` attribute is not given, then the `geometry` child must be specified. Note: rotation attributes are applied in sequence.

    Attributes
    - `file` (string, optional): the Rigid object (.obj) file to be loaded. May be relative or absolute path.
    - `position` (`Vector3`, optional, default (0,0,0)): the position of the object center
    - `rotateRPY` (`Vector3`, optional): rotates the object about the given roll-pitch-yaw entries.
    - `rotateX` (`Real`, optional): rotates the object about the x axis.
    - `rotateY` (`Real`, optional): rotates the object about the y axis.
    - `rotateZ` (`Real`, optional): rotates the object about the z axis.
    - `rotateMoment` (`Vector3`, optional): rotates the object with a rotation matrix derived from the given exponential map representation.

    - `geometry`: sets the object's geometry.

      Attributes
      - `mesh` (string): the geometry file (.tri, .pcd). (Note: "mesh" is a misnomer, in the future it should work with any type of geometry file)
      - `scale` (`Real` or `Vector3`, optional): a scale factor for the mesh. If 3 elements are given, then this scales the mesh separately along each axis.
      - `translate` (`Vector3`, optional): a translation for the mesh.
      - `margin` (`Real`, optional, default 0): the collision boundary layer width.

- `physics`: sets the physics parameters of the object.

  Attributes
  - `mass` (`Real`, optional, default 1): the object's mass.
  - `com` (`Vector3`, optional, default (0,0,0)): the object's center of mass, relative to the origin of its coordinate frame.
  - `inertia` (`Matrix3`, optional, default 0): the object's inertia matrix.
  - `automass` (value "0" or "1", optional): the object's COM and inertia matrix will be set automatically from the geometry.
  - `kRestitution, kFriction, kStiffness, kDamping` (`Real`s, optional, defaults 0.5, 0.5, inf, inf): set the constitutive parameters of the object.

- `terrain`: adds a terrain to the world.

  Attributes
  - `file` (string): the geometry (.tri or .pcd) file to be loaded. May be relative or absolute path
  - `scale, margin`: see `world/rigidObject/geometry/scale, margin`.
  - `translation, position`: see `world/rigidObject/position`.
  - `rotate*`: see `world/rigidObject/rotate*`.
  - `kFriction`: see `world/rigidObject/physics/kFriction`.

  - `display` (optional): configures the OpenGL display of the terrain.

    Attributes
    - `color` (`Vector3` or `Vector4`, optional, default light brown): sets the RGB or RGBA color of the terrain.
    - `texture` (string, optional): sets a texture. Can be "noise", "checker", "gradient", and "colorgradient" at the moment.

- `simulation` (optional): configures the simulation model.
  - `globals` (optional): global ODE simulation parameters.

    Attributes
    - `gravity` (`Vector3`, optional, default (0,0,-9.8)): sets the gravity vector
    - `CFM`
    - `EFP`
    - `maxContacts` (int, optional, default 20): sets a maximum number of contacts per body-body contact.
    - `boundaryLayer` (bool, optional, default 1): activates boundary layer collision detection.
    - `rigidObjectCollisions` (bool, optional, default 1): activates object to object collision detection.
    - `robotSelfCollisions` (bool, optional, default 0): activates robot self-collision detection.
    - `robotRobotCollisions` (bool, optional, default 0): activates robot to robot collision detection.

  - `terrain` (optional): terrain configuration.

    Attributes
    - `index` (int): the terrain index.

    - `geometry`: sets up the geometry and constitutive parameters

      Attributes
      - `padding` (`Real`, optional, default 0 for terrains, 0.0025 for everything else): sets the boundary layer thickness.
      - `kRestitution, kFriction, kStiffness, kDamping`: see `world/rigidObject/physics/k*`

- `object` (optional): rigid object configuration

  Attributes
  - `index` (int): the rigid object index.

  - `geometry`: see `world/simulation/env/geometry`.
- `robot` (optional): robot configuration

  Attributes
  - `index` (int): the robot index.
  - `body` (int, optional, default -1): the link index. -1 applies the settings to the entire robot.

  - `geometry`: see `world/simulation/env/geometry`.
  - `controller`: configures the robot's controller. Each controller type has a certain set of optional attributes that can be set here.

    Attributes
    - `type` (string): the controller type. See Section 8.3 for more details.
    - `rate` (Real, optional, default 100Hz): rate at which the controller runs, in Hz.
    - `timeStep` (Real, optional, default 0.01): 1/rate.
  - `sensors`: configures the robot's sensors.
    - *Children:* Any of the sensor types listed in Section 8.2.
- `state`: resumes the simulator from some other initial state.

  Attributes
  - `data` (string): Base64 encoded data from a prior WorldSimulator.WriteState call. Other than simulation state, the world file must be otherwise identical to the one that produced this data.

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, 0, 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: no scaling.
- `geomtransform index m11 m12 m13 m14 m21 m22 m23 m24 m31 m32 m33 m34 m41 m42 m43 m44`: transforms the link geometry with a 4x4 transformation matrix m with entries given in row-major order.
- `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.
- `selfcollision 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]`: indicates how a group of link DOFs associated with link `index` should be interpreted. If `baseindex` is specified, this indicates that the joint operates on a group of DOFs ranging from `baseindex` to `index`. `type` indicates the type of joint, and can be `normal` (1DOF interval), `spin` (1DOF wrapping around from 0 to 2pi), `weld` (0DOF), `floating` (6DOF with 3 translational 1 rotational, `baseindex` must be specified), `floatingplanar` (3DOF with 2 translational 1 rotational, `baseindex` must be specified), `ballandsocket` (3DOF rotational, `baseindex` must be specified).

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.

Properties:

- `property sensors [file or XML string]`: defines the robot's sensors either in an XML file or string. See the World XML format above or Section 8.2 for more details on the XML format of this element.
- `property controller [file or XML string]`: defines the robot's controller either in an XML file or string. See the World XML format above or Section 8.3 for more details on the XML format of this element.

URDF files (.urdf) with Klamp't-specific elements

URDF (Unified Robot Description Format) is a widely used XML-based robot format found in ROS and other packages. Klamp't has always been able to convert URDF files to .rob files, which can be edited to introduce Klamp't-specific attributes, like motor simulation parameters and ignoring certain self-collision pairs. Starting in version 0.6, Klamp't can now read those attributes from URDF files with an extra `<klampt>` XML element. The schema for defining this element is as follows:

- `robot`: top level element. Follows URDF format as usual.
    - `klampt`: specifies Klamp't-specific parameters

      Attributes
      - `use_vis_geom` (bool, optional, default false): use visualization geometry in imported model.
      - `flip_yz` (bool, optional, default true): flip the Y-Z axes of imported link geometries.
      - `package_root` (string, optional, default "."): describe the path of the package described in any "package://" URI strings, relative to the URDF file.
      - `world_frame` (string, optional, default "world"): the name of the fixed world frame.
      - `freeze_root_link` (bool, optional, default false): if true, the root link is frozen in space (useful for debugging)
      - `default_mass` (float, optional, default 1e-8): default mass assigned to links not given mass parameters.
      - `default_inertia` (float, Vector3, or Matrix3, optional, default 1e-8): default inertia matrix assigned to links not given mass parameters.

- `link`: describes link parameters.

  Attributes
  - `name` (string): identifies the link.
  - `physical` (bool, optional, default true): if set to 0, this is a virtual link with no mass.
  - `accMax` (float, optional, default inf): sets the acceleration maximum for this link.
  - `servoP, servoI, servoD` (float, optional, defaults 10, 0, 1): sets the PID gains of this joint (note: must be a normally driven link).
  - `dryFriction, viscousFriction` (float, optional, default 0): sets the friction constants for this joint.

- `noselfcollision`: turns off self collisions.

  Attributes
  - `pairs` (string, optional): identifies one or more pairs of links for which self-collision should be turned off. Whitespace-separated. Each item can be an index or a link name.
  - `group1,group2` (string, optional): if group1 and group2 are specified, collisions between all of the links in group 1 (a whitespace separated list of link indices or names) will be turned off. Either `pairs` or both `group1` and `group2` must be present in the element.

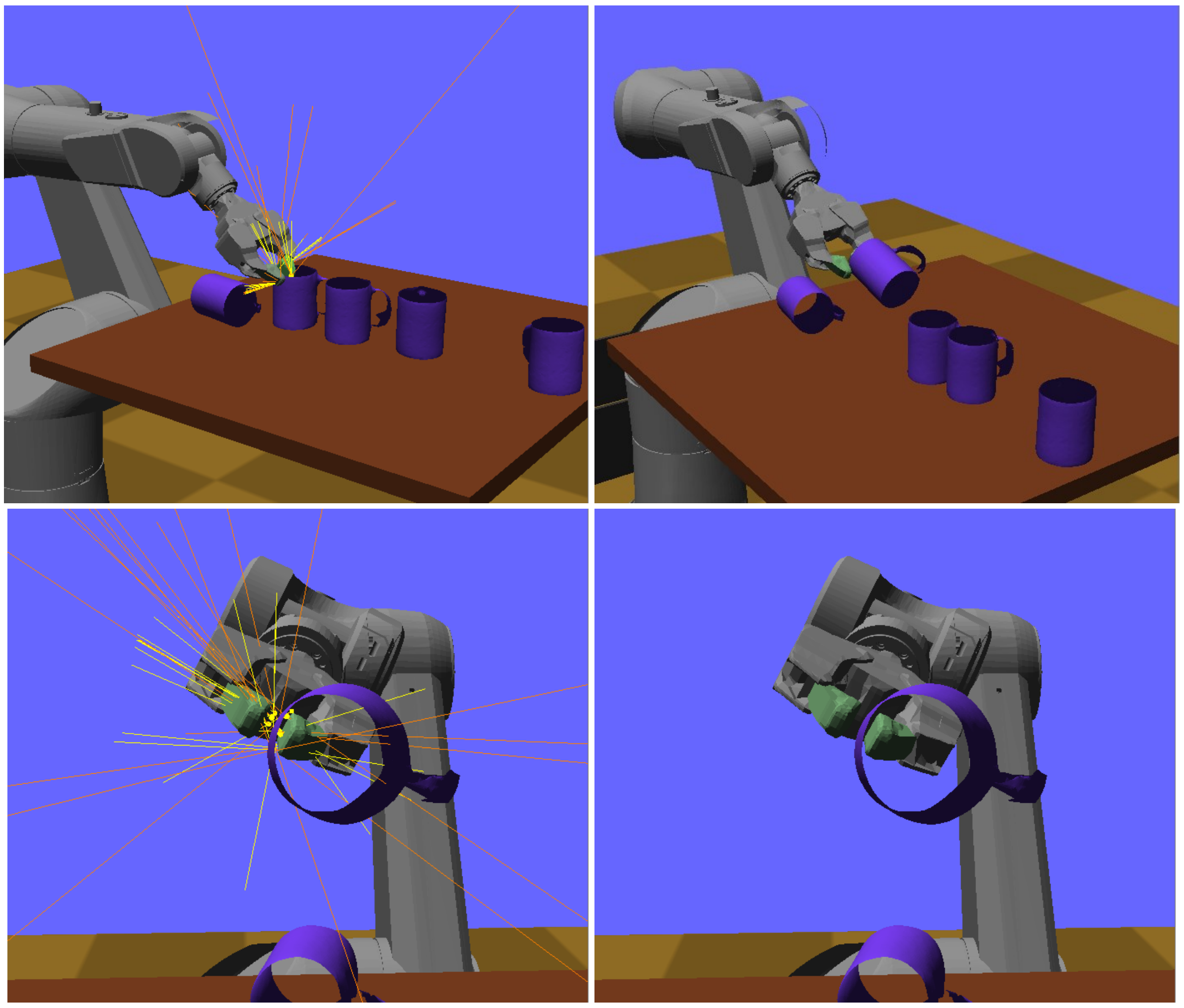
- `selfcollision`: turns on certain self collisions. Note: if this item is present, default self collisions are not used. *Same attributes as noselfcollisions.*
- `sensors:` specifies sensors to be attached to the robot. See the World XML format above or Section 8.2 for more details on the XML format of this element.

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, their coefficients are blended using arithmetic mean for kRestitution, and harmonic means for kFriction, kStiffness, and kDamping.

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.

A variety of kinematic planning algorithms are supported, including PRM, RRT, RRT*, PRM*, Lazy-RRT*, Lazy-PRM*, LBT-RRT, SBL, and PRT.

There are two levels of planning interface. The *robot-level* interface is a higher-level interface automatically defines notions of sampling, collision checking, etc. (similar to the functionality of MoveIt!) The *configuration space* interface is much lower level and more abstract, and requires the user to define feasibility tests and sampling routines (similar to the functionality of OMPL). The lower level approach is more tedious, but provides greater power.

Regardless of which interface you use, the general pipeline is as follows:

1. Construct a **planning problem**. Define the configuration space (C-space) and terminal conditions (start and goal configurations, or in general, sets)
2. Instantiate a **planning algorithm**. *Take care: some algorithms work with some problems and not others*.
3. **Call the planner**. Sampling-based planners are set up for use in any-time fashion:
   a) Plan as long as you want in a while loop, OR
   b) Set up a termination criterion

   Any-time planning means that the likelihood of success increases as more time spent. For optimizing planners, the quality of path improves too.

4. **Retrieve** the path (sequence of milestones)

The resulting paths is then ready for execution or for some postprocessing smoothing.

7.1. ROBOT-LEVEL KINEMATIC MOTION PLANNING

High-level kinematic motion planning generates collision-free paths for robots. The most basic form of planning considers fixed-base robots in free space (i.e., not in contact with the environment or objects).

C++ API. Example code is given in Examples/plandemo.cpp (the application can be created via the command `make PlanDemo`).

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 "any" setting will choose an algorithm automatically.
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/RobotCSpace.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 = "any";  //options are "prm", "rrt", "sbl", "prm*", etc
//do more planner setup here if desired, e.g., change perturbation size
MotionPlannerInterface* planner = factory.Create(&cspace);
int istart=planner->AddMilestone(qstart); //should be 0
int igoal=planner->AddMilestone(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, distance metric weights and contact tolerances may be tuned. Collision margins can be tuned by editing the margins of robot/object/terrain geometries.

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.

Python API. At the highest level, the klampt.robotplanning module offers convenience functions (`planToX`) to set up plans to generate collision-free plans for a robot to different types of targets. Planning options can be configured and extra constraints fed into the planner using these functions. `SubRobotModels` are also supported to plan for selected parts of a robot.

For even greater control, you should create an appropriate C-space for your problem and then call a planner manually. Several robot-level C-spaces are available for you in klampt.plan.robotcspace.

- `RobotCSpace`: avoids collisions with other objects in the world.
- `ContactCSpace`: avoids collisions, maintains IK constraints.
- `StanceCSpace`: same as `ContactCSpace`, but also enforces balance under gravity given known points of contact.

The planToX functions generate an instance of a `MotionPlan` class, defined in klampt.plan.cspace. For manual CSpace creation, you will need to create a MotionPlan instance and set up your C-space and start and goal conditions via `MotionPlan.setEndpoints`.

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.
- For a complete description of the accepted options, see the `setPlanSetting` documentation in the Python/klampt/src/motionplanning.h file.
- The constructor selects between different planner types via the `type` argument. Examples may include 'prm', 'rrt', 'sbl', 'rrt*', etc.

To run the planning algorithm, call `MotionPlan.planMore` with the desired number of iterations. Continue calling it until `MotionPlan.getPathEndpoints` returns non-`None`.

To debug or inspect the results of a planner, the, `MotionPlan.getRoadmap()` or `MotionPlan.planner.getStats()` methods can be used.

7.2. CONFIGURATION SPACE KINEMATIC MOTION PLANNING

For even more control, the base C-space interfaces can be overridden with custom behavior. A wide variety of systems can be defined in the configuration space framework, including vehicles and other non-robotic mechanisms.

C++ API. Each C-space is a subclass of the configuration space interface class `CSpace` defined in KrisLibrary/planning/CSpace.h. Please see the documentation

Python API. Each C-space is a subclass of the configuration space interface `CSpace` defined in klampt.plan.cspace. At a minimum, the subclass should set up the following:

- `bound`: a list of pairs [(a_1,b_1),…,(a_n,b_n)] giving an n-dimensional bounding box containing the free space
- `eps`: a visibility collision checking tolerance, which defines the resolution to which motions are checked for collision.
- `feasible(x)`: returns true if the vector x is in the feasible space. (an alternative to overriding `feasible` is to call `addFeasibilityTest(func,name)` for each constraint test.)

The feasibility test is an *authoritative* representation of C-space obstacles, and will be called thousands of times during planning. For sampling-based planners to work well, this must be fast (ideally, microseconds).

To implement non-Euclidean spaces, users may optionally override:

- `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

Setting up and invoking motion planners is the same as in the robot-level interface.

7.3. 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 is currently only supported in the C++ API.

C++ API. The example program in Examples/dynamicplandemo.cpp demonstrates how to do this (the program can be built using the command `make DynamicPlanDemo`).

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, will satisfy velocity and acceleration bounds, and will be feasible.

Warning: free-rotational joints (robots with free-floating bases) will not be interpolated correctly because this method assumes a Cartesian configuration space. Spin joints are also not handled correctly at step 3 but they can be handled by replacing step 5 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.*

7.4. INTERPOLATION AND TIME-OPTIMIZATION WITH CLOSED-CHAIN CONSTRAINTS (C++ ONLY)

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 TrajOpt program will do this from the command line.

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.

7.5. RANDOMIZED PLANNING WITH CLOSED-CHAIN CONSTRAINTS

Klamp't has utilities to plan for collision-free motions that satisfy closed chain constraints (e.g., that a robot's hands and feet touch a support surface).

C++ API. 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. Example code is given in Examples/contactplan.cpp (the application can be created via the command `make ContactPlan`).

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.

Python API. The `planToX` functions in klampt.plan.robotplanning accept arbitrary inverse kinematics constraints using the `equalityConstraints` keyword argument. Internally, these functions use the `ContactCSpace` class defined in klampt.plan.robotcspace. As in the C++ API, the plans are milestone lists, which should not be interpolated linearly in joint space. Rather, the `space.discretizePath(path,epsilon=1e-2)` convenience function is provided to calculate an approximate piecewise-linear joint space path from the milestone path.

7.6. TIME-SCALING OPTIMIZATION (C++ ONLY)

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.

7.7. REAL-TIME MOTION PLANNING (C++ ONLY)

Real-time motion planning allows a robot to plan while executing a previously planned path. This allows the robot to avoid moving obstacles, improve path quality without large delays, and change its goals in real-time. It is critical to use a system architecture that tightly controls the synchronization between planning and execution; the planner must not spend more than a predetermined amount of time in computation before delivering the updated result, or else the path could change in an uncontrolled manner with catastrophic consequences. Moreover, such a method must be robust to unpredictable communication delays.

Klampt's real-time motion planning routines are built to handle these issues gracefully, and furthermore have the following theoretical guarantees

- The executed path is guaranteed to be continuous and within joint, velocity, and acceleration limits
- In a static environment the path is guaranteed to be collision free
- Any goal will eventually be reached given sufficient time (in wall clock time)

Real-time planning is only supported in the C++ API. The main files containing this functionality are the `RealTimePlannerBase` base class in Klampt/Planning/RealTimePlanner.h and the subclass `RealTimeTreePlanner` in Klampt/Planning/RealTimeRRTPlanner.h. A complete implementation including communication with the User Interface Thread/Execution Thread is given in the `MTPlannerCommandInterface` class in Klampt/Interface/UserInterface.h.

Conceptually, the main requirement is that the Execution Thread and Planning Thread must be synchronized via a *motion queue*. The motion queue is a modifiable trajectory $y(t)$ that is steadily executed by the Execution Thread. The Planning Thread is allowed to edit the motion queue asynchronously by *splicing* in a changed path, which modifies the motion queue after at a given time. Right now, the motion queue must be a DynamicPath (in future implementations this requirement may be relaxed). Splices are specified on an absolute clock, because when a splice is made at time t_s, the planner must ensure that the old motion queue and the new suffix match at the same configuration $y(t_s)$ and velocity at $y'(t_s)$.

The Planning Thread should be initialized with the robot's initial configuration (or path) and its inner loop should proceed as follows:

1. Globally, the planner's objective is set using Reset and a planning cycle is begun at time t_p by calling PlanUpdate.
2. The planner determines a split time t_s and planning duration δt. It is required that $t_s > t_p + \delta t$.

3. The planner tries to compute a path starting from $y(t_s)$ and $y'(t_s)$. If unsuccessful, the planning cycle terminates with failure.
4. Otherwise, the planner requests that the path gets spliced to the motion queue via the `SendPathCallbackBase` mechanism. The queue has an opportunity to reject the request, such as if it arrives after the current execution time or has incorrect configuration or velocity. A rejected splice is signaled by returning false to the callback.
5. Return to step 1.

The generic `SendPathCallbackBase` callback must be subclassed and implemented to make splice requests. In practice, properly implementing this callback requires locking and synchronization between threads. Either 1) the motion queue must be synchronized, or 2) splice requests are written to the Execution Thread, and a reply is written to the Planning Thread (as done in `MTPlannerCommandInterface, the` result to `SendPathCallbackBase` is queried via a polling mechanism).

A planning cycle can be interrupted with the `StopPlanning` method. This is useful to maintain responsiveness to changing user input.

There are two policies for determining the planning duration: constant and adaptive. When using sampling-based planners we recommend using the adaptive time stepping mechanism because it adapts to planning problem difficulty. For deterministic planners, a well-chosen constant time step may be more appropriate.

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++ API. 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.

A robot's sensors are dynamically configured via an XML tag of the form `<sensors> <TheSensorType name="some_name" attr1="value" … /> </sensors>`. 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.

These XML strings can be inserted into .rob files under a line `property sensors [file]`, URDF files under the `<klampt>` element, or world XML files under the `<simulation>` and `<robot>` elements

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, perform some processing, and write motor commands. The implementation of the internal processing is open to the user.

Default motion queue controller. The default controller for each robot is a `FeedforwardPolynomialPathController`, which simulates typical controllers for industrial robots. This is a motion-queued controller with optional feedforward torques. It supports piecewise linear and piecewise cubic interpolation, as well as time-optimal acceleration-bounded trajectories.

C++ API: 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. The basic control loop repeatedly executes:

1. The `RobotSensors* sensors` structure is filled in by the Klamp't simulation (or physical robot).
2. The `RobotController.Update` method is called. Here, the controller should fill in the `RobotMotorCommands* command` structure as necessary.
3. The `command` structure is read off by the Klamp't simulation (or physical robot).

Python API. By default, the `SimRobotController` class implements a `FeedforwardPolynomialPathController`. The `setMilestone` and `addMilestone` methods set and append a new destination milestone to the motion queue, respectively. The `klampt.model.trajectory.execute_trajectory` function helps execute trajectories or arising from planners. However, this behavior can be overridden using the low-level `setPIDCommand` and `setTorqueCommand` functions.

To define a custom controller, the user should implement a custom control loop. At every time step, read the robot's sensors, compute the control, and then send the control to the robot via the `setPIDCommand` or `setTorqueCommand` methods.

(Note: One limitation of the API is that it is impossible to have certain joints controlled by a motion queue, while others are controlled by PID commands.)

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.
- *SerialController* (Klampt/Control/SerialController.h): a thin communication layer that serves sensor data and accepts commands to/from a client controller through a serial interface. It listens on the port given by the setting `servAddr` and sends sensor data at the rate `writeRate` (in Hz). Sensor data and commands are converted to/from JSON format, in a form that is compatible with the Python API dictionaries used by the `klampt.control.BaseController` class (see also Klampt/Python/control/controller.py).

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. If state estimation is performed, 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.

9. CONTROLLER INTEGRATION

Klamp't supports a number of mechanisms for connecting controllers and simulated robots with external modules. An external controller can be connected to a Klamp't simulated robot, or a Klamp't controller can be connected to a physical robot.

9.1. CONNECTING AN EXTERNAL CONTROLLER TO A KLAMP'T SIMULATED ROBOT

There are three options for doing so (plus a fourth variant):

1. **Direct instantiation in C++**. You must define a subclass of `RobotController` and perform all necessary processing in its `Update` method. (To integrate with SimTest you must edit and recompile Klamp't).
2. **Direct instantiation in Python**. This can be done manually in the simulation loop by sending PID/ torque commands to the simulator, or it can be done by subclassing the controller.py interface used by simtest.py as described in Section **Error! Reference source not found.**.
3. **Socket communication with controller**. This interface allows you to communicate directly with SimTest. To do so, a robot is given a `SerialController` controller, which acts as a relay to an external client by sending sensor data and receiving motor commands over a socket. Data is serialized in JSON format.
4. **(another instance of method 3) ROS joint_trajectory and joint_state messages.** These are supported in SimTest via a SerialController and the rosserialrelay.py script, or in simtest.py using the roscontroller.py script.

More details for each of the methods are given below.

Direct instantiation in C++. Once you have created your new controller, a new controller object of your class should be sent to the `WorldSimulation.SetController()` method when launching your own simulation. Or, the controller can be registered using `RobotControllerFactory::Register` as described in Section 8.3, and its type can be specified in the world XML file.

Direct instantiation in Python. See Section **Error! Reference source not found.** for details.

Socket communication with controller. This procedure consists of first setting a robot to use a SerialController controller, and writing a binding for your external controller to connect to the server socket, and process messages using the controller communication protocol (CCP) covered in Section 9.2.

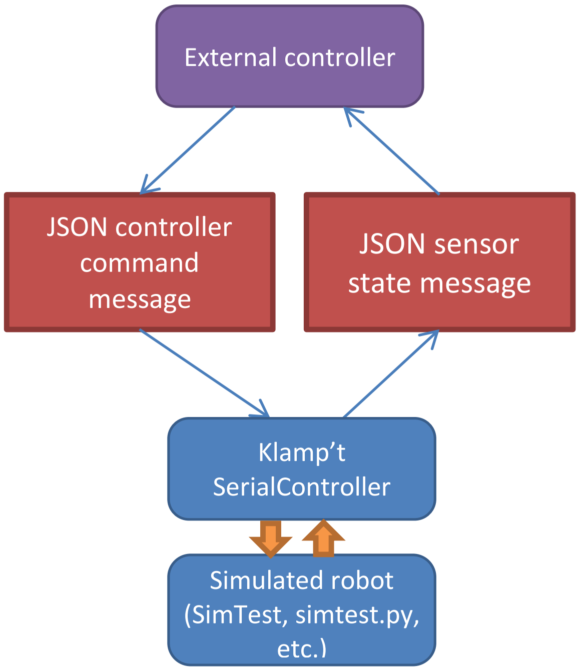

External controller
JSON controller command message
JSON sensor state message
Klamp't SerialController
Simulated robot (SimTest, simtest.py, etc.)

Figure 10. Connecting an external controller to a simulated robot via a socket communication protocol with a SerialController controller.

As an example, consider an external Python controller.

1. Run `./SimTest data/tx90serialinput.xml` in one window. The SerialController controller in SimTest will listen for clients to connect to localhost:3456 (the port is specified in the world XML file). Once a client connects, it will write *sensor messages* to the socket at a fixed rate and then receive *command messages* from the socket as they are generated.
2. Run `python Python/control/serialcontroller.py data/motions/tx90sway.txt` in another window. This script connects as a client and begins receiving *sensor messages* over the socket, processes them (in this case using a trajectory controller), and sends the resulting *command messages* back over the socket.

ROS communication with controller. The rosserialrelay.py script runs a daemon to relay ROS messages to a SerialController. It reads position, velocity, and/or feedforward torque commands from the `/[robot_name]/joint_trajectory` ROS topic and writes sensed joint states to the `/[robot_name]/joint_states` ROS topic. It directly translates these items to a SerialController on localhost:3456 by default. As usual, you may start up the SerialController through SimTest's Controller window, or by specifying a SerialController as a robot's controller via the world XML file.

A more direct method for use in the simtest.py controller interface is provided by the roscontroller.py script. It functions nearly identically to rosserialrelay.py, but without the need to communicate over a socket or to edit XML files to set up the SerialController instance.

Note that in both cases, you must build the `klampt` ROS package (a Catkin workspace has already been provided for you in the Python/Control/klampt_catkin folder), and use `rosrun` to start the scripts. Please refer to the ROS documentation for details.

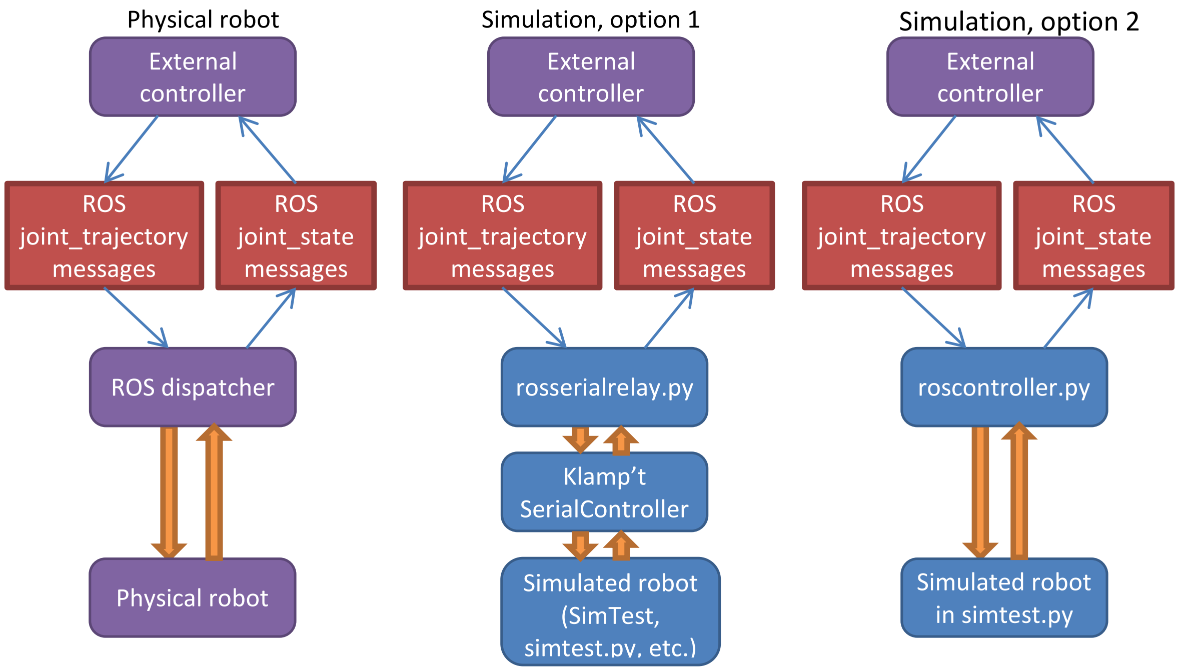


Figure 11. Connecting an external controller to a simulated robot via ROS.

9.2. CONTROLLER COMMUNICATION PROTOCOL (CCP)

A *sensor message* is a structure with the following elements:

- `t`: the current simulation time.
- `dt`: the controller time step.
- `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.

A *command message* is a structure which contains one of the following combinations of keys, signifying which type of joint control should be used:

- `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 with the given actuator velocities, executed for time `tcmd`.
- `torquecmd`: use torque control.

Each command key (except `tcmd`) must be associated with a list of driver values. Note that these are driver values rather than configuration values; as a result the controller must be aware of which drivers are present in the .rob file (as well as their ordering).

CCP messages are serialized in JSON format for socket communication with a SerialController, or as Python dictionaries as used in simtest.py.

9.3. CONNECTING A KLAMP'T CONTROLLER TO A PHYSICAL ROBOT

To connect a Klamp't controller to a physical robot, a wrapper around the control loop should repeatedly fill in the controller's sensor data from the physical data, and write the actuator commands to the physical motors.

The standard interface is given in the `ControlledRobot` base class in Klampt/Control/ControlledRobot.h. Your subclass should override the `Init`, `ReadSensorData`, and `WriteCommandData` methods to provide whatever code is necessary to communicate with your robot. See Examples/cartpole.cpp for an example.

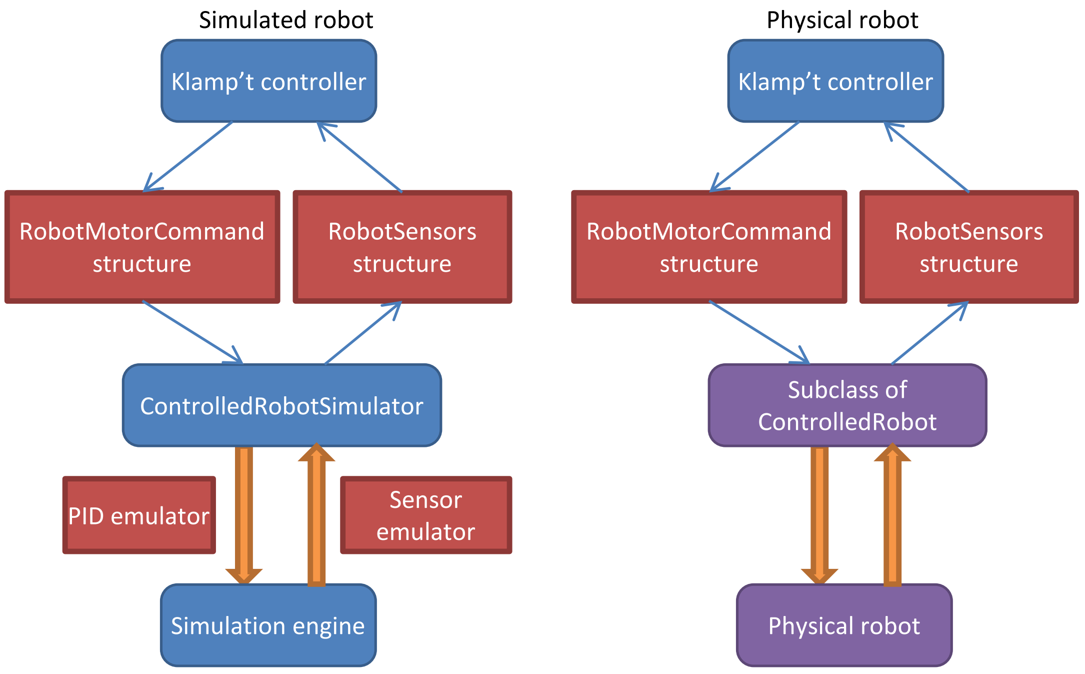


Figure 12. Connecting a Klamp't controller to a physical robot using a ControlledRobot subclass.

9.4. CONNECTING A KLAMP'T PLANNER TO A CONTROLLER

A planner can communicate asynchronously with a controller in real-time using several methods. The general technique is to instantiate a planning thread that sends / receives information with the controller whenever planning is completed.

As an example, consider the real-time planning classes in Planning/RealTimePlanner.h and their interfaces in Interface/UserInterface.h. The real time planners send trajectory information to the controller via a `MotionQueueInterface`, which just relays information to the `PolynomialPathController` in the simulation.

[The reason why the interface is used rather than communicating directly with a `PolynomialPathController` is that it is possible to implement a `MotionQueueInterface` to send trajectory data to the robot directly. The real-time planning demos produced by the IML on the physical TX90L robot use a `MotionQueueInterface` that communicates with the real controller over Ethernet via a simple serial API. This approach often saves bandwidth over implementing a `ControlledRobot` subclass.]

10. 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.

11. 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. For beginners or for rapid prototyping, this is the best API to use. However, it does not contain all of the functionality of the C++ API.

Missing features include:

- Advanced IK constraint types
- Trajectory optimization
- Some contact processing algorithms
- Robot reachability bound determination
- Advanced force/torque balance solvers
- Advanced motion planners (optimal planning with custom objective functions, kinodynamic planning, etc)
- Direct access to a robot's trajectory queue.

11.1. THE KLAMPT MODULE

The core modeling and simulation Klamp't functionality is found in the klampt module, which automatically imports several classes that wrap C++ functionality via SWIG. 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 forward kinematics and dynamics.

It should be noted that the documentation of these basic classes are found under the klampt.robotsim submodule. Their online documentation may also look somewhat strange for Python users, having been converted from C++ comments via SWIG.

Other native Python modules exist for a whole host of other functions, such as computing IK solutions via the klampt.model.ik module, or do other kinds of planning tasks via klampt.plan.robotplanning module.

11.2. SUB-MODULES

Sub-modules of klampt include math, model, io, plan, sim, and vis. Sub-modules not discussed elsewhere are as follows:

- model.cartesian_trajectory: reliable Cartesian interpolation functions between arbitrary task space points. Also defines a convenient "bump" function that modify joint-space paths to achieve a task-space displacement.
- model.collide: defines a `WorldCollider` class that enables querying the collision status of the world and subsets of bodies in the world.
- model.config: a uniform interface for determining a flattened list of floats describing the configuration of a world entity, a mathematical object, or an IK goal.
- model.contact: allows querying contact maps from a simulator and computing wrench matrices, and equilibrium testing.
- model.coordinates: a coordinate transform manager, similar to the tf module in ROS, that lets you attach points / vectors to frames and determine relative or world coordinates.
- model.hold.py: defines a `Hold` class and writes / reads holds to / from disk.
- model.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.*
- model.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().
  ```

  Most notably used in the sim.batch module.

- model.sensing: functions for processing simulated sensor data.
- model.subrobot: a class that is `RobotModel`-like but only modifies selected degrees of freedom of the robot (e.g., an arm, a leg). Many klampt module functions accept `SubRobotModels` in the place of `RobotModels`.
- model.types: retrieving the resource manager type string for various Klamp't objects.
- io.loading: functions for loading/saving Klamp't objects to strings and/or disk in both native format and JSON formats.
- io.resource: functions for loading/saving/editing Klamp't resources.
- plan.cspaceutils contains helpers for constructing composite `CSpaces` and slices of `CSpaces`.
- sim.batch: functions for batch Monte-Carlo simulation of many simulation initial conditions.
- sim.settle: convenience functions to let objects fall under gravity and extract their equilibrium configurations.
- sim.simlog: simulation logging classes (used in SimpleSimulator)

- sim.simulation: a more full-featured simulation class than standard Simulation. Defines sensor and actuator emulators, sub-step force appliers, etc.

The klampt module does not (yet) contain interfaces to trajectory optimization and state estimation.

11.3. VISUALIZATION

Klamp't supports two frameworks for producing interactive visualizations:

- Method 1: use the klampt.vis scene manager.
- Method 2: overload GLPluginInterface and customize the event handling and drawing routines.

Method 1 is much easier to set up, and is more intuitive for users who may be unfamiliar with the event-driven paradigm used in GUI programming. Using the scene manager, GUI windows pop up in a separate visualization thread, and the main thread can add and remove items to the scene manager. Simple functions are available to build multi-viewport GUIs, to customize appearances, control animations, and other visualization functions. For more information see the documentation of klampt.vis.visualization, and the example code in Klampt/Python/demos/vistemplate.py.

In Method 2, users will need to override the event handling functions to draw, process mouse and keyboard input, etc. Users are also welcome to use Klamp't object OpenGL calls in their own frameworks. For more information, see the documentation of klampt.vis.glinterface, and the simple example file Klampt/Python/demos/gltest.py.

A hybrid of Method 1 and Method 2 is also available in Klampt/Python/demos/visplugin.py. This hybrid approach is primarily used to customize how the scene manager responds to user input.

11.4. 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/kbdrive.py: drive a simulated robot around using the keyboard. The first 10 joints can be driven via a positive velocity with the top row of keys 1,2,…,0 and a negative velocity with the second row of keys q,w,…,p.
- demos/robotiq.py: modeling and simulating the RobotiQ 3-finger Adaptive Gripper. This code emulates the underactuated transmission mechanism of each finger.
- demos/robotiqtest.py: performs a simulation of the RobotiQ gripper closing and opening on an object.
- demos/simtest.py: an imitation of SimTest program programmed entirely in Python, and an entry point to fast prototyping of controllers using the Python API.
- demos/sphero.py: simulates the Sphero 2.0 robot driving around.
- demos/vistemplate.py: demonstrates how to use the basic interface to the visualization module.
- demos/visplugin.py: demonstrates how to develop plugins for the visualization module.
- utils/config_to_driver_trajectory.py: converts a linear path from configuration space (# of DOF) to driver space (# of actuators).

- utils/driver_to_config_trajectory.py: converts a linear path from driver space (# of actuators) to configuration space (# of DOF).
- utils/discretize_path.py: splits a linear path into a fixed time-domain discretization.
- utils/make_thumbnails.py: generates thumbnails of a folder full of world, robot, object files, etc.
- 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.
- utils/tri2off.py: converts old-style .tri files to .off files.

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.

11.5 EXPERIMENTAL CONTROLLER API

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 important methods:

- `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.
- `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!*
- `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 a sensor message as described in the controller communication protocol (CCP) in Section 9.2. The arguments are Python dictionaries. simtest.py expects output to return a dictionary that represents a command message as described in the CCP.

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 trajectory tracking controller is given in control/trajectory_controller.py. Its `make` function accepts a robot model (optionally None) and a linear path file name.

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

12. FREQUENTLY ASKED QUESTIONS (FAQ)

12.1. 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 provide a strict subset of the C++ functionality.

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

Sensors are set up in the `property sensors` line of the robot file or world XML file. See Sections 5.13 and 8.2 for more details, and see data/robots/huboplus/huboplus_col.rob and data/simulation_test_worlds.xml for some examples. Sensors can be debugged and drawn in RobotTest.

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

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

12.3. 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". The simulator status will also return "unreliable."

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.

12.4. 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.

12.5. HOW DO I IMPLEMENT A BEHAVIOR SCRIPT?

Many engineers and students tend to approach robotics 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*. However, it is true that scripts / state machines are sometimes the easiest way to accomplish a given behavior with the current generation of robot AI tools.

To implement a behavior script in Klamp’t, the script should be launched in a separate thread from the execution thread. It can then monitor the state of the execution thread (e.g., waiting for a movement to finish) and react accordingly. For those new to threading, please see the C++ classes in KrisLibrary/utils/threadutils.h or the Python `threading` module for more information.

To implement a state machine, a controller should manually maintain and simulate its behavior in its feedback loop. A framework for such controllers the `StateMachineController` class in Python/control/controller.py.

13. RECIPES (HOW DO I…?)

13.1. GENERATE A PATH/TRAJECTORY FROM KEYFRAMES

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

The Qt GUI makes it easy:

- Create a Configs resource (call it "milestones").
- Create a Config resource (call it "milestone1") and drag and drop it under the "milestones" resource
- Pose the robot as desired, and while "milestone1" is selected click "From Poser"
- Repeat steps 2 and 3 for as many milestones as desired. Use drag and drop to order the milestones as necessary.
- (untimed path) While selecting the "milestones" resource, click the "Convert To…" dropdown menu. Select MultiPath or Linear Path as desired.
- (timed path) While selecting the "milestones" resource, click the "Optimize" button.
- Save the resulting new resource.

It Qt is not available, this can also be done in the GLUI GUI, but with more work. 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. (for an untimed path) Click "Create Path" to generate a new interpolating path. This will create a new `Multipath` resource in the Resource Library.
6. (for a timed path) Click "Optimize Path" to generate a new interpolating trajectory.
7. Name the Multipath and save it to disk.
8. [optional] If you prefer a linear path, you may select the Multipath, click "Convert" and type in "LinearPath" when prompted in the command line.

13.2. ANIMATE A VIDEO OF A PATH/TRAJECTORY

Qt GUI. In RobotPose, paths/trajectories can be played when selected in the Resource Library using the media controls in the lower right,. Run "`./RobotPose [world file] [path file]`" and select the path. Uncheck the "Draw geometry" button or move the poser robot away, then click the Record button (red dot) to begin saving PPM screenshots to disk. These files will be processed into a video file using a utility like ffmpeg once recording is stopped.

Note: for best results with your video encoder, you may have to set the frame size manually to a standard size using the Camera menu.

GLUT GUI. 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. The Python/demos/simtest.py program uses the `GLSimulationPlugin` visualization plugin, which start saving frames for a movie by pressing 'm'. For kinematic simulations, you must manually interpolate and save image files to disk. The `GLProgram` class in the klampt.vis.glprogram module has a `save_screen` method that uses the Python Imaging Library to save the current OpenGL view to disk. See Python/demos/gltemplate.py for an example.

13.3. 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.getController(0).addMilestone(q)
```

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

```
for (q,v) in path:
        sim.getController(0).addMilestone(q,v)
```

13.4. 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 Python/demos/simtest.py, run "`./simtest.py [world file] [trajectory file]`".

Manual operation: Load a `Trajectory` object (see klampt.model.trajectory). 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 call `execute_trajectory` in klampt.model.trajectory, or use the `TrajectoryController` class in control/trajectory_controller.py.

13.5. IMPLEMENT A CUSTOM CONTROLLER

C++ API.

1. Create a new subclass of `RobotController` and override, at a minimum, the `Type` method, which provides a name for the controller, and the `Update` method, which reads from the `sensors` member and writes to the `command` member.
2. Add your controller to the default controller factory by adding the line RobotControllerFactory::Register(new MyController(robot)) in the `RobotControllerFactory::RegisterDefault` method in Control/Controller.cpp.
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="[string returned by MyController::Type()]" />`.

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 11.5.

13.6. 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. The `GLPluginInterface` interface class allows users to call the method `click_world,` which returns a depth-sorted list of objects clicked at the mouse's x-y position. See Python/demos/gltemplate.py for an example of how to use it.

14. 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 or `klampt.io.resource.edit()`.
- *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.

15. 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):

- Monte-Carlo simulation generation and browsing for mechanism design, behavior evaluation, and machine learning
- More convenient manipulation planning support
- Convenience routines for easier dynamic motion planning
- Unification of locomotion and manipulation planning
- Specifying and solving optimization and optimal control problems
- State estimators for free-floating robots
- Planning with sliding and rolling contacts
- Rolling friction simulation
- Expansion of the Python API (e.g., trajectory optimization, real-time planning)

16. 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.