libORS
Open Robot Simulation Toolkit

Marc Toussaint

September 7, 2011

Contents

1 Installation & testing

The README file has more detailed installation instructions. The super quick way: on Ubuntu/Debian copy this to your console:

```
sudo apt-get install liblapack-dev freeglut3-dev \
     libqhull-dev libf2c2-dev
wget http://user.cs.tu-berlin.de/~mtoussai/source-code/libORS.10.1.tgz
tar xvzf libORS.10.1.tgz
cd libORS
make
cd test/ors
./x.exe
```

Then test all executables `x.exe` in the directories `test/*`.

2 Scope & overview

Basic tools for robot simulation. The lib defines basic data structures to describe robot configurations (trees/-graphs of rigid bodies), implements the basic computation of kinematic/Jacobian/Hessian functions, and links to many external libraries and engines for more sophisticated things. It uses:

- SWIFT++ to compute shape distances/collisions
- Featherstone's Articulated Body Dynamics as an implementation of exact dynamics on articulated tree structures (much more precise than IBDS or ODE)
- IBDS (a rather robust impuls-based physical simulator)
- ODE (I don't like it)
- OpenGL for display
- read/write of file formats for robot configurations, shape/mesh files (e.g., obj files), etc

3 Source code guide

Read the header

– `ors.h`

3.1 ORS data structures

- Check the Array class in `array.h` - it's yet another generic container class. There are many reasons why I decided reimplementing such a generic container (instead of using std::vector, blast, or whatever):

  – it's fully transparent, easy debugging

  – very robust range checking

  – direct linkage to LAPACK

  – tensor (multi-dimensional array) functions which are beyond most existing matrix implementations

  – etc

  Anyway, the Array class is central in all my code. To get a first impression of its usage, check the test/array. In the context of ORS, we mainly use double arrays to represent vectors, matrices and do linear algebra, note the typedef

  ```
  typedef MT::Array<double> arr;
  ```

- *Lists, Graphs, etc* In my convention a *List* is simply an array of pointers. Since arrays allow memmove operations, insertion, deletion etc are all O(1). I also represent graph structures via lists: e.g. a list of nodes and a list of edges, a node may maintain a list of adjoint edges, etc.

  For Lists (Arrays of pointers) it makes sense to have additional methods, like calling delete for all pointers, or writing the referenced objects to some output – at the bottom of `array.h` there are a number of template functions for lists and graphs.

- See the ors.h file. It defines a number of trivial data structures and methods that should be self-explanatory:

  – Vector

  – Matrix

  – Quaternion

  – Frame (a coordinate system)

  – Mesh (a triangulated surface)

  – Spline

- Given these types, a dynamic physical configuration is defined by lists of the following objects

  – Body: describes the physical (inertial) properties of a rigid body. This is mainly simply a Frame (position, orientation, velocities). Optionally (for dynamic physical simulation) this also includes inertial properties (mass etc) and forces.

  – Joint: desribes how two bodies are geometrically linked and what/where its degree of freedom is. The geometry of a Joint is given by a rigid transformation A (from body1 into the joint frame), a free transformation Q (the transformation of the degrees of freedom), and a rigid transformation B (from the joint frame to body2). Overall, the transformation from body1 to body2 is the concatenation $B \circ Q \circ A$.

  – Shape: describes the collision and shape properties of a rigid body. To each rigid body we may associate multiple Shapes, like primitive shapes (box, sphere, etc) or Meshes; each shape has a relative transformation from its body.

  – Proxy: describes a proximity between two shapes, i.e., when two shapes are close to each other. This includes information like the closest points on the two shapes and the normal. This information is computed from external libraries like SWIFT.

- The Graph data structure contains the lists of these objects and thereby describes the configuration of the whole physical system. It includes a number of low level routines, in particular for computing kinematics, Jacobians, dynamics etc. We don't describe these routines here – the SOC abstraction will provide a higher-level interface to such quantities which is closer to the mathematical notation of stochastic optimal control.

  Use the ors_editor application to define your own physical configuration (described later in the user's guide). Learning to define a configuration should also give you sufficient understanding of the Body, Joint, and Shape data structures.

4 Tools

4.1 ors_editor and the ors-file format

- ors_editor is a very simple program that helps editing ors-files. ors-files contain the definition of a physical configuration. See the directory test/ors_editor, the binary program is test/ors_editor/x.exe, a symbolic link bin/ors_editor exists. It works like this:

```
emacs test.ors &
./ors_editor test.ors &
```

  Then you edit the test.ors file in your standard text editor (here, emacs). Whenever you like, you press enter within the OpenGL window to update the display – when you made mistakes in the syntax, error messages will be output to the console.

- The general syntax of the ors-file is very simple: it lists elements in the syntax

```
elem_type elem_name (list of parents) { key-value list }
```

  (This is a general hypergraph syntax, which I also use in other contexts (factor graphs), where elements may connect an arbitrary number of parent elements; nodes are special case in that they connect no parents, edges are special case in that they connect exactly two parents, etc)

  In our case we have three possible types: body, joint, shape. This is a simple example:

```
#any comment after a # sign

body base (){
  X=<t(0 0 1)>              #coordinate system of this bo
}

body arm {}

shape some_shape_name (arm) {
  rel =<d(10 0 1 0)>        #rel . transf . torso -> shap
  type=2
  size=[0 0 1 .1]
#  mesh='filename.tri'   #if you had a mesh file: set typ
}

joint some_joint_name (base arm){
  A=<t(0 0 .5) d(90 0 1 0)> #rel . transf . torso -> join
  B=<t(0 0 .5)>             #rel . transf . joint -> arm
}
```

  The attribute list is simply a list of tag=something declarations. The 'something' can be a single double number, an array [1 2 3 4] of numbers, a string in quotes, a transformation $< \cdots >$, or a list of strings in parenthesis (string1 string2 etc). Generally, you can set any attributes you like. But only some special tags have effects right now – the most important ones are explained in the example. See the routines ors::Body::read, ors::Joint::read, ors::Shape::read for details on which attributes have actually effects. The routine ors::Graph::read

parses a whole ors-file and creates the respective data structures.

- We need to explain coordinate systems and how to specify transformations. A transformation is given as a sequence of primitive transformations enclosed in brackets <...>. The most important primitive transformations are a translation t(x y z), a rotation d(degrees axis_x axis_y axis_z). Concatenating them you can generate any transformation. See the ors::Frame::read routine to learn about all primitive transformations.

  Every body has its own coordinate system (position and rotation in world coordinates), which you can specify with X=<...>. Also every joint has its own coordinate system – we assume that the x-axis is always the rotation axis of the joint. One can specify the coordinate system of a joint directly with X=<...> (in world coordinates), or the relative transformations from parent→joint→child with A=<...> and B=<...>, respectively. Specifying all these transformations at the same time is redundant, of course. Whatever transformations you do not specify (including body coordinates), the parser tries to compute from the given absolute or relative transformations and the tree structure of the kinematics. [[This doesn't work fully automatically in the current version!]]

4.2 ors_fileConverter

To view, convert, resize, and cleanup meshfiles, there is a little application test/ors_fileConverter/x.exe (and a symbolic link bin/ors_fileConverter). It simply provides an application interface to the functionalities of the ors::Mesh data structure. Please see the test/ors_fileConverter/main.cpp to learn about all functionalities. Test something like

```
./ors_fileConverter filename.obj -view -box
./ors_fileConverter filename.stl -view -box -center -qhull -save
```

A Thinking in sequences of transformations

An intuitive way to describe transformations is to specify a sequence of translations and rotations. The turtle way to do this is to assume that *each transformation is interpreted relative to the "current" carried-along turtle frame*. For instance, if A and B are two rotations. Then the turtle notation $\langle A \cdot B \rangle$ describes the transformation that first rotates A and then rotates B which is interpreted relative to the coordinate frame that is the outcome of the A rotation. Let us compare this to standard math notation: The matrix expression $(AB)x = A(Bx)$ means to first apply B on the vector x (both represented in "world coordinates") and then apply A on Bx (again both represented in "world coordinates"). The turtle notation $\langle A \cdot B \rangle$ means to first apply A on x and then ABA^T (i.e., B interpreted relative to the outcome frame of the A rotation) on Ax. But that is the same as $(ABA^T)(Ax) = ABx$. In conclusion, the turtle notation $\langle A \cdot B \rangle$ describes exactly the same transformation as the matrix expression AB – but the turtle notation interprets this as first applying A and then applying B (interpreted relative), whereas the matrix expression means first applying B (intepreted in global coordinates) and then A. The same also works for translations: When T is a translation $\langle T \cdot A \rangle$ means a translation by a vector T followd by a rotation where A is intepreted as a rotation *around the current frame origin* T and not around the world coordinate origin. Thus, applying $\langle T \cdot A \rangle$ on a vector x we get $T + Ax$, which in affine matrix notation is the same as TAx. Again, the turtle interpretation is first applying T then A, whereas the matrix intepretation is first applying A and then the affine translation T.

The turtle view on concatenating transformations is rather intuitive, especially for mechalically linked system where we think of one body being attached to another. It also corresponds to the usual OpenGL thinking of stacking transformations, where (in GL_PROJECTION mode) we typically first load the identity matrix (glLoadIdentity), and then "add" transformations on top (using glTranslate or glRotate). This procedure directly corresponds to the turtle notation.

ORS uses the following ascii notation for transformations

t(x y z)	translation by (x, y, z)
q(q0 q1 q2 q3)	rotation by a quaternion (q_0, q_1, q_2, q_3)
r(r x y z)	rotation by r radians around the axis (x, y, z)
d(d x y z)	rotation by d degrees around the axis (x, y, z)
v(x y z)	addition of linear velocity (x, y, z)
w(x y z)	addition of angular velocity (x, y, z)
s(x z y)	a scaling (diagonal matrix) with factors (x, y, z)

For instance the notation <t(0 0 1) d(90 1 0 0)> means a translation along the z-axis followed by a rotation (90 degrees) around the x-axis.