libSOC
Stochastic Optimal Control Library

Marc Toussaint

July 23, 2009

When using this library, please cite [1].

Contents

1 Installation

- on Debian, install the packages liblapack-dev, freeglut3-dev, libqhull-dev, libf2c2-dev
- type make, and pray
- if that didn't work: have a look in the make-config in the base directory. If the linker complained about undefined template function references, uncomment the –DMT_IMPLEMENT_TEMPLATES entry in make-config. If linking with glut, lapack or qhull failed, comment these flags in the make-config. If it still doesn't work, contact me.
- cd test/array; ./x.exe
- briefly try all other tests in test/*/x.exe
- read this guide
- to activate collision detection, also install SWIFT++, and set the SWIFT=1 flag in make-config

2 Scope & overview

The *primary* scope of this lib is the implementation of Stochastic Optimal Control (SOC) methods (namespace `soc`) – that is, methods to compute (approximatly) optimal controllers and trajectories, typically in the context of robot motion. In particular, this includes

- *Approximate Inference Control* [1],
- iLQG (iterative Linear-Quadratic-Gaussian)
- gradient/spline trajectory optimization
- methods for 1-step control (optimal dynamic control, regularized/Bayesian motion rate control, etc)

The *secondary* scope of this lib is a robot simulator (namespace `ors`) that provides the necessary inputs to the methods above. Using this simulator is optional – it is provided only for completeness of the lib (and I use it in my work). But all the methods above can also be linked to your own simulation environment. My ors implementation tries to be minimalistic in its core, but can link to many conventient external libraries and engines: it defines basic data structures to describe robot configurations (trees/graphs of rigid bodies), implements the basic computation of kinematic/Jacobian/Hessian functions, and of course implements the SocAbstraction. 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

The interface between the SOC methods and the simulator is the `soc::SocAbstraction`: a class that defines functions that the SOC methods need access to and

that need to be provided by the simulator. This SocAbstraction tries to be as close as possible to the typical mathematical notation used for Stochastic Optimal Control problems. If you're only interested in the SOC methods and not in the ORS simulator, you should start reading from section 3.2.

3 Programmer's guide

There are three headers which, in the end, you should understand:

– `array.h`
– `ors.h` and `ors_control.h`
– `soc.h`

They implement quite a lot – the following should give some orientation.

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

  – very robust range checking, easy debugging

  – 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 SOC, 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, velicities). 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 to 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.

3.2 Stochastic Optimal Control

You should read this when you want to use your own simulator and thereby have to implement the SocAbstration. Otherwise, when you use ORS, you may skip this section (although it's interesting in itself :-)).

We consider a discrete time stochastic controlled system of the form

$$P(x_{t+1} \,|\, u_t, x_t) = \mathcal{N}(x_{t+1} \,|\, f_t(x_t, u_t), Q_t) \quad (1)$$

with time step t, state $x_t \in \mathbb{R}^n$, control $u_t \in \mathbb{R}^m$, and Gaussian noise ξ of covariance Q; where

$$\mathcal{N}(x|a, A) \propto \exp\{-\frac{1}{2}(x-a)^\top A^{-1} (x-a)\} \tag{2}$$

is a Gaussian over x with mean a and covariance A. For a given state-control sequence $x_{0:T}, u_{0:T}$ we define the cost as

$$C(x_{0:T}, u_{0:T}) = \sum_{t=0}^{T} c_t(x_t, u_t) \,. \tag{3}$$

The optimal value function $J_t(x)$ gives the expected future cost when in state x at time t for the best controls and obeys the Bellman optimality equation

$$J_t(x) = \min_u \Big[c_t(x,u) + \int_{x'} P(x' \,|\, u, x)\; J_{t+1}(x')\Big] \,. \tag{4}$$

The closed-loop (feedback) control problem is to find a control policy $\pi_t^* : x_t \mapsto u_t$ (that uses the true state observation in each time step and maps it to a feedback control signal) that minimizes the expected cost.

The linear quadratic gaussian (LQG) case plays an important role as a local approximation model. LQG is a linear control process with Gaussian noise,

$$P(x_{t+1} \,|\, x_t, u_t) = \mathcal{N}(x_{t+1} \,|\, A_t x_t + a_t + B_t u_t, Q_t) \,,$$

and quadratic costs,

$$c_t(x_t, u_t) = x_t^\top R_t x_t - 2 r_t^\top x_t + u_t^\top H_t u_t \,. \tag{5}$$

The LQG process is defined by matrices and vectors $A_{0:T}, a_{0:T}, B_{0:T}, Q_{0:T}, R_{0:T}, r_{0:T}, H_{0:T}$. In the LQG case, the optimal controller can be computed exactly using the Ricatti equation – and the optimal controller will always be a linear controller in the form

$$u_t^*(x_t) = G_t\,(x_t - g_t) \tag{6}$$

and we can also compute the most likely trajectory $x_{0:T}^*$, which is also the optimal (cost minimal) trajectory in the zero-noise case $Q = 0$.

Robotic systems are typically non-LQG. Nevertheless, we can approximate the system locally (i.e., around a current robot state) as LQG. This is exactly what the robot simulator has to provide and what the SocAbstraction defines. In other terms, a simulator needs to provide a mapping

$$x_t \mapsto (A_t, a_t, B_t, Q_t, R_t, r_t, H_t) \tag{7}$$

which gives the approximate system matrices for a current robot state x_t.

(NOTE: future implementations will also provide non-Gaussian messages/approximations of task constraints...)

In the following we list how to compute these matrices for typical robot motion optimization scenarios:

Kinematic motion rate control The robot state is simply the posture $x_t \equiv q_t \in \mathbb{R}^n$ (not velocities). We assume direct motion rate control. The process is simply

$$q_{t+1} = q_t + u_t + \xi \tag{8}$$

and therefore

$$A_t = 1 \,, \quad B_t = 1 \,, \quad a_t = 0 \tag{9}$$

Dynamic torque control The robot state is $x_t \equiv \bar{q}_t = \begin{pmatrix} q_t \\ \dot{q}_t \end{pmatrix}$. We assume torque control where the system process is given (approximately) in terms of the local mass matrix M and force vector F,

$$P(q_{t+1} \,|\, \dot{q}_t, q_t) = \mathcal{N}(q_{t+1} \,|\, q_t + \tau \dot{q}_{t+1}, W^{-1}) \,, \tag{10}$$

$$P(\dot{q}_{t+1} \,|\, \dot{q}_t, u_t) = \mathcal{N}(\dot{q}_{t+1} \,|\, \dot{q}_t + \tau M^{-1}(u_t + F), Q) \,, \tag{11}$$

$$\begin{pmatrix} q_{t+1} \\ \dot{q}_{t+1} \end{pmatrix} = \begin{pmatrix} 1 & \tau \\ 0 & 1 \end{pmatrix} \begin{pmatrix} q_t \\ \dot{q}_t \end{pmatrix} + \begin{pmatrix} \tau^2 \\ \tau \end{pmatrix} M^{-1}(u_t + F) + \xi$$

$$\langle d\xi\, d\xi^\top \rangle = \begin{pmatrix} W^{-1} & 0 \\ 0 & Q \end{pmatrix} \tag{12}$$

$$A = \begin{pmatrix} 1 & \tau \\ 0 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} \tau^2 M^{-1} \\ \tau M^{-1} \end{pmatrix}, \quad a = \begin{pmatrix} \tau^2 M^{-1} F \\ \tau M^{-1} F \end{pmatrix} \tag{13}$$

Pseudo-dynamic control A simplification of dynamic control which still yields nice and dynamically smooth trajectories is this: The robot state is $x_t \equiv \bar{q}_t = \begin{pmatrix} q_t \\ \dot{q}_t \end{pmatrix}$. And we assume the control directly determines accelerations,

$$P(q_{t+1} \,|\, \dot{q}_t, q_t) = \mathcal{N}(q_{t+1} \,|\, q_t + \tau \dot{q}_{t+1}, W^{-1}) \,, \tag{14}$$

$$P(\dot{q}_{t+1} \,|\, \dot{q}_t, u_t) = \mathcal{N}(\dot{q}_{t+1} \,|\, \dot{q}_t + \tau u_t, Q) \,, \tag{15}$$

$$\begin{pmatrix} q_{t+1} \\ \dot{q}_{t+1} \end{pmatrix} = \begin{pmatrix} 1 & \tau \\ 0 & 1 \end{pmatrix} \begin{pmatrix} q_t \\ \dot{q}_t \end{pmatrix} + \begin{pmatrix} \tau^2 \\ \tau \end{pmatrix} u_t + \xi$$

$$\langle d\xi\, d\xi^\top \rangle = \begin{pmatrix} W^{-1} & 0 \\ 0 & Q \end{pmatrix} \tag{16}$$

$$A = \begin{pmatrix} 1 & \tau \\ 0 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} \tau^2 \\ 1 \end{pmatrix}, \quad a = 0 \tag{17}$$

Kinematic task costs The robot state $x_t \equiv q_t \in \mathbb{R}^n$ is kinematic. We have m task variables $y_i \in \mathbb{R}^{\dim(y_i)}$. For isntance, these could be the 3D endeffector position, the 2D horizontal balance, a 1D collision cost variable, a 1D joint limit cost variable, etc. For each we have a kinematic function $\phi_i(q) = y_i$ and a Jacobian $J_i(q) = \partial_q \phi_i(q)$. We are given task targets $y_{i,0:T}^*$ and want to follow them with (time-dependent) precisions $\varrho_{i,0:T}$. We have

$$\begin{aligned} c_t(q_t, u_t) &= \sum_{i=1}^m \varrho_{i,t}[y_{i,t}^* - \phi_i(q_t)]^2 + u_t^\top H_t u_t \\ &\approx \sum_{i=1}^m \varrho_{i,t}[y_{i,t}^* - \phi_i(\hat{q}_t) + J_i \hat{q}_t - J_i q_t]^2 + u_t^\top H_t u_t \,, \quad J_i = J_i(\hat{q}_t) \\ &= \sum_{i=1}^m \varrho_{i,t}[q_t^\top J_i^\top J_i q_t - 2(y_{i,t}^* - \phi_i(\hat{q}_t) + J_i \hat{q}_t)^\top J_i q_t + \text{const}] \\ &\quad + u_t^\top H_t u_t \end{aligned} \tag{18}$$

$$R_t = \sum_{i=1}^{m} \varrho_{i,t} J_i^\top J_i \tag{19}$$

$$r_t = -2 \sum_{i=1}^{m} \varrho_{i,t} J_i^\top (y^*_{i,t} - \phi_i(\hat{q}_t) + J_i \hat{q}_t) \tag{20}$$

Dynamic task costs The robot state $x_t \equiv \bar{q}_t = \begin{pmatrix} q_t \\ \dot{q}_t \end{pmatrix}$ is dynamic. We have m task variables $y_i \in \mathbb{R}^{\dim(y_i)}$ with kinematic function $\phi_i(q) = y_i$ and Jacobian $J_i(q) = \partial_q \phi_i(q)$. We are given task targets $y^*_{i,0:T}$ and $\dot{y}^*_{i,0:T}$ and want to follow them with (time-dependent) precisions $\varrho_{i,0:T}$ and $\nu_{i,0:T}$. We have

$$c(q_t, \dot{q}_t, u_t) = \sum_{i=1}^{m} \varrho_{i,t} [y^*_{i,t} - \phi_i(q_t)]^2 + \nu_{i,t} [\dot{y}^*_{i,t} - J_i \dot{q}_t]^2 + u$$

$$\approx \sum_{i=1}^{m} \varrho_{i,t} [q_t^\top J_i^\top J_i q_t - 2(y^*_{i,t} - \phi_i(\hat{q}_t) + J_i \hat{q}_t)^\top J_i q_t + const]$$

$$+ \nu_{i,t} [\dot{q}_t^\top J_i^\top J_i \dot{q}_t - 2(\dot{y}^*_{i,t})^\top J_i \dot{q}_t + const] + u_t^\top H_t u_t \tag{21}$$

$$R_t = \sum_{i=1}^{m} \begin{pmatrix} \varrho_{i,t} J_i^\top J_i & 0 \\ 0 & \nu_{i,t} J_i^\top J_i \end{pmatrix} \tag{22}$$

$$r_t = -2 \sum_{i=1}^{m} \begin{pmatrix} \varrho_{i,t} J_i^\top (y^*_{i,t} - \phi_i(\hat{q}_t) + J_i \hat{q}_t) \\ \nu_{i,t} J_i^\top \dot{y}^*_{i,t} \end{pmatrix} \tag{23}$$

The SocAbstraction should implement exactly these computations of the system matrices.

3.3 Control (task) variables

IT'S ALL ABOUT COUPLED VARIABLES!

The whole philosophy of my approaches is that we are faced with a problem of coupled (random) variables, which refer to goals, constraints, observations, states, etc, and the problem is to find values for these variables consistent with all given information (a posterior distribution over undetermined variables conditioned on the determined variables).

So, the central aspect of using this code is to define such variables, and define whether/how they should be constrained to desired target values and by which precision these constraints should be fulfilled.

The ORS simulator includes a number of ways to declare task variables – which in the code are called ControlVariable (sorry for this overload of names...). Defining such ControlVariables means to specify the actual motion problem and objectives. Let's start with an example.

In `test/soc` there is an example program. The test.ors file defines a really simple configuration with a 7DoF arm, a green target ball, and a red obstacle ball. The interesting parts of the code are:

```
// ...
// [setup the ORS simulator, swift, opengl, and the SocAbstraction]

//– setup the control variables (problem definition)
ControlVariable *pos = new ControlVariable(
        "position",ors, posCVT,"arm7","<t(0 0 .2)>",0,0,ARR());
pos->x_target = arr(ors.getName("ball")->X.p.v,3);
pos->setInterpolatedTargetTrajectory(T);
pos->setPrecisionTrajectoryFinal(T,1e-2,1e4);

ControlVariable *col = new ControlVariable(
        " collision ",ors, collCVT,0,0,0,0,ARR(.1));
col->x_target = ARR(0.);
col->setInterpolatedTargetTrajectory(T);
col->setPrecisionTrajectoryConstant(T,1e6);

soc.setControlVariables(TUPLE(pos,col));

// [use inverse kinematics or planning to compute the motion]
// ...
```

This code defines two control variables. See the constructor of the first variable, pos: it is named "position", it is associated to the simulator ors, its type is a kinematic position variable (enum posCVT), it refers to the body named "arm7", and it assumes an additional relative transformation "<t(0 0 .2)>" of the actual reference point relative to the body coordinate system. This is a 3D variable and conditioning this variable corresponds controlling this point of reference during the motion (corresponds to standard inverse endeffector kinematics of the 7th arm body).

The second control variable, col, is named " collision ", is computed from ors, has the type collCVT, and gets as last parameter an array [0.1] which specifies the distance threshold (margin) for collision costs. This is a 1D variable that measures the sum of cost of collisions (or shape-shape distances below the threshold) summed over all shape pairs that are below the threshold. Conditioning this variable to zero means that we'll avoid collisions.

For both variables we first define a (far future) target x_target and then specify a target trajectory (including precisions) over a time interval of $T = 200$ time steps. For pos, the future target is the position of the green ball (the body called "ball"), the target (endeffector) trajectory interpolates linearly from the initial position to the target – but the precision along the target trajectory is such that we only require for the last time step high precision (1e4 $\sim$ 1 centimeter standard deviation) whereas time steps $0..T-1$ low precision (1e-2 $\sim$ 10 meters standard deviation). For the collision variabl we require high precision (1e-6) throughout the time interval $0..T$.

Specifying such control variables and their target trajectories/precisions is the core of defining the motion problem. Once they are specified, the algorithms (Bayesian IK, AICO approximate inference control, or gradient methods) should do the rest of the job.

3.4 The OrsSocImplementation

The `soc::OrsSocImplementation` is the connecting interface between the ORS simulator and the control variables on the one hand, and the SocAbstraction on the other hand. It is very instructive to have a look at the implementation of the routines – in particular when you want to implement another SocAbstraction based on your own simulator. For instance, consider `soc::OrsSocImplementation::setq`: the q array contains all joint angles, we first set them in the `ors::Graph` data structure and recompute all body positions according to these joint angles. Then we update all ControlVariables by recomputing their state and their Jacobian w.r.t. the current state. After we've set the state using setq, we can easily access all necessary information from the SocAbstraction. For instance, `soc::OrsSocImplementation::getJtJ` simply accesses the Jacobian of a particular ControlVariable – the algorithm behind the SocAbstraction doesn't need to know any semantics or meaning of that ControlVariable, it only needs to know its current state, target/precicion, and the Jacobian. For instance, an easy algorithm for motion computation is the `soc::bayesianIKControl` – have a look at its code: it simply loops through all existing ControlVariables, queries their state, target/precision, and Jacobian, adds things up (following equations (19,20), which implicitly computes a task constraints message), and returns the maximum aposteriori step dq in joint space.

3.5 Motion algorithms

See the documentation of `soc.h` for a list of all motion algorithms. All of them are implemented on the basis of the SocAbstraction.

4 User's guide

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) { attribute 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 torso (){
  X=<t(0 0 1)>                  #coordinate system of this body
}
body arm {}

shape some_shape_name (torso) {
  rel=<d(90 0 1 0)>             #rel. transf. torso -> shape
  type=3
  meshfile='filename.tri'
}

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

References

[1] M. Toussaint. Robot trajectory optimization using approximate inference. In *Proc. of the 26rd Int. Conf. on Machine Learning (ICML 2009)*, 2009.