The EXtensible Optimisation Toolset

EXOTica vENOM (v0.1) pre-release

Application Programming Interface

Abstract

This document describes the Implementation of the EXOTica Library. The toolset is designed to allow simpler implementation of optimisation routines for robotic applications, particularly inverse-kinematics and path planning. The driving force behind the library is modularity (advocating a plugin architecture where users can re-implement individual components while making use of the rest) and re-useability (indeed, it is up to a point platform agnostic).

Contents

1 Introduction

This document describes the EXOTica library. It is intended to serve as a reference for all users as well as future maintainers of the tool-set.

1.1 The EXOTica Library

The EXOTica library is a generic Optimisation Toolset for Robotics platforms, written in C++. Its motivation is to provide a more streamlined process for developing algorithms for such task as Inverse-Kinematics and Trajectory Optimisation. Its design advocates:

- **Modularity** The library is developed in a modular manner making use of C++'s object-oriented features (such as polymorphism). This allows users to define their own components and 'plug them into' the existing framework. The end-effect is that an engineer need not implement a whole system whenever he needs to change a component, but rather can re-implement the specific functionality and as long as he follows certain guidelines, retain the use of the other modules.

- **Extensibility** The library is also heavily extensible, mainly thanks to the modular design. In addition, the library makes very minimal prior assumptions about the form of the problem so that it can be as generic as possible.

- **Platform Independence** The toolset is designed to operate on a variety of robotic platforms, and as such is written with platform independence in mind. It is entirely OS-agnostic and the only library dependencies are Eigen and Boost. For the pre-release version the build system is catkin (which is ROS-based) but the library itself has no dependency on ros and indeed, can be built using cmake.

- **Integration with ROS** Although it is not necessary, the library is fully integrateable with ROS. Structures for this are planned in the coming releases.

1.2 Structure of the Document

This document is aimed at three different classes of people:

- **Users** should read section 3 and optionally 2 if they are new to ROS and the catkin build environment.

- Those who wish to **extend** the library's functionality by adding new methods of solutions should also read section 4.
- Finally **maintainers** should refer to the API section (5) as well as the doxygen comments in the code.

2 Setup

EXOTica currently consists of a single (catkin) package which contains all the header files and cpp-files implementing the raw functionality.

2.1 Pre-requisites

EXOTica currently has the following dependencies:

1. **Eigen** Library for matrix manipulation, specifically Eigen 3.0. Refer to http://eigen.tuxfamily.org/ for more information.

2. **Boost** to enforce thread-safety and prevent memory leaks. Please refer to http://www.boost.org/.

2.2 Installation

The library is currently supplied as a catkin package. Hence building in ROS is trivial: ensure the correct dependencies are met and build using catkin_make. The library uses the c++11 standard, so ensure that the available cmake compiler supports this.

If you decide to build independently using your favourite compiler, you just need to link against the above two libraries and that is all.

3 Solving Optimisation Problems

This Section introduces the use of the library in as generic a way as possible. Wherever concrete examples are needed, reference is made to the task of solving Inverse Kinematics (IK).

3.1 Introductory Optimisation Theory

The Generic Optimisation problem is usually defined in terms of a cost function at each time step. Let us assume we want to achieve N tasks simultaneously. A typical cost function would have the following generic form (indexed in time t):

$$C_t(q_t, X_t^*) = ||q^t - q^{t-1}||_R^2 + \sum_{n=1}^{N} v_n ||x_n^{*t} - \phi^n(q^t)||_{W_n}^2 \quad (1)$$

Here, we employ the following notation:

- $\mathbf{q}$: The Vector of the configuration space: in IK this is the vector of joint angles for the robot.
- $\mathbf{x}^*$: Vector of task space goals for a single task (such as end-effector positions).
- $\mathbf{X}^*$: Vector of cumulative task goals. The concept of a global vector viz-a-vis the individual vectors is purely for the sake of user-friendliness.
- ϕ : Forward mapping from Configuration (*joint*) space to Task (*position*) space.

The above cost function achieves two things: the second term (the summation) penalizes deviation from the task specification and hence enforces motion towards the goals. Individual tasks may be weighted internally (between the dimensions of the task) through the W_n matrix and between tasks themselves using the v_n scalar weighting. The first term, often known as the regularisation term, penalizes huge deviations from the current configuration space and is weighted internally through R.

The above setup, while allowing relative weighting between tasks/dimensions, is still restrictive. If a priority scheme is required (where a set of tasks is solved for first and the remaining ones are solved subject to solution of the first), the null-space of the first solution is used to solve additional tasks.

Denoting the cost of these additional tasks (similar form to the second term) by h, we modify the above equation to achieve Eq. (2):

$$C_t(q_t, X_t^*) = ||q^t - q^{t-1}||_R^2 + \sum_{n=1}^{N} v_n ||x_n^{*t} - \phi^n(q^t)||_{W_n}^2 - 2h_t^T R q_t \qquad (2)$$

Of course there can be arbitrary many levels of chaining.

Optimisation of (2) results in the following deterministic solution:

$$q^t = q^{t-1} + J_{W,R,q_t}^{\sharp}(X^* - X^{t-1}) + (\mathbf{I} - J^{\sharp}J)h \qquad (3)$$

Here, the key computation occurs in the pseudo-inverse of the Jacobian or $J^{\sharp}$. The Jacobian maps changes in the config-space to velocities in task-space: thus the pseudo-inverse is used to obtain the optimal changes in config-space to achieve the desired change in task-space. Note three things:

- The method of computing a Jacobian (commonly the moore-penrose pseudo-inverse) incorporates the relative weightings W and R and governs the form of the solution, which is usually only a local solution, requiring some form of iteration.
- We use the concept of a '**Big Jacobian**' which incorporates all task-variables at the same priority level.
- h, referred to as null-space motion, can contain a further optimisation using the respective pseudo-inverse of the Jacobian.

3.2 Architectural Overview

In view of the above requirements, the library is implemented as a hierarchical structure:

1. At the lowest level sits the TaskDefinition object. This defines the properties of the task, namely ϕ, J, x^*, W_n and v_n.
2. A set of tasks at the same priority level are encapsulated into an OptimisationFunction object, which also implements the regularisation term.
3. An *std::vector* holds prioritized OptimisationFunction objects, with the first one having the highest priority and so on.

4. The solver is implemented in two stages:

    - A VelocitySolver which computes the pseudo-inverse optimisation and null-space resolution (the last two terms in (3)
    - A PositionSolver which iterates and integrates velocities until convergence.

Thus using the library amounts to creating and defining vectors of OptimisationFunction objects and passing these to the appropriate solver. For the sake of user-friendliness and ease of use, object factories are used behind the scenes to dynamically create objects as required.

3.3 Creating an Optimisation Function

Using the library involves first defining the Optimisation problem. The example of Inverse Kinematics will be used. Let us assume that there is an implemented ***IK_Task*** class, as well as the associated ***IK_Pos*** and ***IK_Vel*** solvers (implementation of these is indeed underway as part of the standard library implementations). As an example task to solve, assume there is the need to control the position of the gripper of a 7-DOF robotic arm. In addition, a specific orientation of the arm is also desireable, given that the position is achieved.

The library requires the "***exotica/EXOTica.hpp***" include. The first step is to instantiate and initialise an OptimisationFunction object:

```
OptimisationFunction position_ik; //Create the Object

position_ik.setParams("position_ik", Eigen::VectorXd::Ones(7)
    *0.001); //Set the Name and Regularisation Weights R
```

The name member is merely for convenience, while the regularisation weights are taken to be equal. Within the function, the task can now be defined using the ***add_task(...)*** method:

```
if (position_ik.addTask("IK_Task", "position") != True)
{
  cout << "Oops! Unable to create task of type IK_Task" << endl;
  return;
}
```

The definition of the tasks needs values for the intra- and inter-task weights as well as the goal. Access to the task is through a ***boost::weak_ptr*** to enforce thread-safety.

```
{
  boost::shared_ptr<exotica::IK_Task>    task_ptr; //Create a
    shared pointer
  task_ptr = position_ik.access("position").lock();//Assign weak
    pointer to temporary shared pointer
  task_ptr->setTaskWeight(1.0); //One task so unity weight
  task_ptr->setGoalWeight(Eigen::VectorXd::Ones(3));   //All
    cartesian co-ordinates equally important
  task_ptr->setGoal(Eigen::VectorXd::Ones(3)*0.3, 0.001); //Go
    to (0.3, 0.3, 0.3) with 1mm tolerance
}
```

Note how the operations are carried out within a new block to enforce locality of the task_ptr. Additionally, different tasks may require additional parameters, specific to the implementation. Please refer to the appropriate readme. Once this is fully initialised, it can be pushed back into a vector:

```
std::vector<exotica::OptimisationFunction> my_optimisation;
my_optimisation.push_back(position_ik);
```

Similarly, an orientation task definition can be defined and pushed back to the vector. Note that although the ordering of tasks within a single optimisation function does not matter (i.e. it does not depend on the order in which ***addTask(...)*** is called), the order of optimisation functions within the optimisation vector does!

3.4 Solving the Optimisation Problem

Once the task is defined, the solution can proceed. We first set some global parameters. Within the current version (v0.1) this consists solely of the size of the trajectory which for IK is by definition unity.

```
exotica::OptimisationParameters_t init_params;
init_params.optimisation_window = 1;
```

A position solver and velocity solver must be instantiated:

```
exotica::IK_Pos ik_pos(init_params);
exotica::IK_Vel ik_vel(init_params);
```

The final step is to solve:

```
Eigen::VectorXd joint_solution;
bool success;

success = ik_pos.solve(my_optimisation, Eigen::VectorXd::Zeros
    (7), &ik_vel, joint_solution);
```

The key operation is calling the ***solve(...)*** function which takes 4 arguments:

1. The vector of optimisation functions
2. The initial configuration
3. Pointer to the velocity solver (allows one to arbitrarily mix and match position/velocity solvers).
4. The vector where to store the configuration-solution.

4 Extending the Library

One of the major goals of this project were to provide an extendible platform. This applies both at the level of task-types, velocity solvers and position solvers. Hence, implementing any of these abstract classes allows the library to grow as required.

4.1 Creating Task-Definition Classes

The TaskDefinition class **must** contain everything that defines the task. This is currently an abstract interface and must be implemented to model the required problem (for example an IK-type task).

The Abstract Interface

The abstract class itself handles storage of the weighting matrix for intra-task weights, the inter-task weights as well as the goal vector (with tolerance). In addition, it provides support for trajectory optimisation and hence deals with vectors of the above data structures, one for each time-step (by default the vector is of size 1). Finally, the base-class as it is is guaranteed to be thread-safe.

Inheriting from the TaskDefinition Class

All inherited objects must implement two functions.

```
bool computeJacobian(const Eigen::VectorXd &, Eigen::MatrixXd &,
    int)
```

The function takes as inputs the vector of current configuration, the Matrix for storing the resulting jacobian and the index (if applicable) into the time-indexed trajectory (should default to 0). It should indicate success or failure through its return value. In addition, the function should internally set the respective jacobian (again time-indexed) using the provided ***setJacobian(...)*** function.

```
bool computeForwardMap(const Eigen::VectorXd &, Eigen::VectorXd
   &, int)
```

The function takes as input again the current configuration, a vector for storing the corresponding task-space co-ordinates and any indexing into the

trajectory vector. Again, success or failure should be indicated as well as storing the computed ϕ using the ***setPhi(...)*** method.

Tips and Tricks

- Remember to maintain support for trajectory optimisation by always making use of indexing and storing relevant task-parameters within vectors.
- Enforce thread-safety through mutexes. This is a major requirement of this object. One efficient way of dealing with the fact that the jacobian computation typically calls the forward map function is to have the forward map call a **private** unlocked compute function which is also the function used by the jacobian computer. All public interfaces will be blocking but this internal one will not so there won't be a problem due to blocking within the same thread.
- Ideally, since the classes can be instantiated dynamically, it is preferable to check for correct initialisation within the *compute* functions.
- The abstract class uses default arguments for the index-variable. This must be maintained.

4.2 Creating Position Solvers

Position Solvers define the interpolation and approximation process in solving optimisation problems by interpolating velocity changes. The class is currently an Abstract Interface.

The Abstract Interface

The PositionSolver abstract class only holds the optimisation parameters and that is it.

Inheriting from the PositionSolver class

All inherited objects must implement the solve function:

```
bool solve(std::vector<OptimisationFunction> &, const Eigen::
    VectorXd &, boost::shared_ptr<VelocitySolver>, Eigen::
    VectorXd &)
```

The inputs are in order:

1. Vector of Optimisation Functions
2. Initial Configuration of the robot
3. Shared Pointer to the Velocity Solver
4. Velocity Solution

Ideally it should indicate success or failure through the return variable. The function is intended to call the velocity solver's ***solve()*** function for actually computing incremental updates to the current solution: it is thus the task of the position solver to integrate these local solutions to achieve a final result.

Tips and Tricks

- Again, ensure that thread safety is maintained as much as possible within the ***solve(...)*** function.
- The position and velocity solvers are tightly coupled although they are still independent. Take care of this in implementation.

4.3 Creating Velocity Solvers

The final piece of the puzzle is the velocity solver which computes "locally approximate velocities" to reduce the error towards the goal. The class is also an abstract interface.

The Abstract Interface

The Abstract version of the class serves as a place-holder for the optimisation parameters (which must be initialised through the constructor). It exposes two functions, one to solve for the optimal velocity which in turn calls the inverse-calculator. In the interest of modularity, both may be overloaded, but the inverse-function is pure-virtual and so **must** be implemented.

Inheriting from the VelocitySolver class

All classes inheriting from the velocity solver **must** implement the inverse computation:

```
bool getInverse(const Eigen::MatrixXd &, const Eigen::MatrixXd
    &, const Eigen::MatrixXd &, Eigen::MatrixXd &)
```

The function takes the following arguments (in order):

1. The '**big**' **Jacobian** to invert.
2. The **Configuration-Weight Matrix** R.
3. The '**big**' **Task-Weight Matrix** W.
4. The place-holder for the resulting inverted **Jacobian**.

In keeping with the convention of the library, it should indicate success or failure of the solution. Typically, the pseudo-inverse can take the form of the moore-penrose decomposition or singular value decomposition, but this is entirely up to the implementer.

Additionally, one may choose to overload the solve function, although this is not necessary. The function currently handles the construction of the weight matrices for passing on to the pseudo-inverse calculation, as well as the casting of temporary solutions into the null-spaces required. The function has the following signature:

```
bool solve(std::vector<OptimisationFunction> &, const Eigen::
    VectorXd &, Eigen::VectorXd &)
```

with the arguments being, in order:

1. The vector of optimisation functions
2. The initial configuration of the system
3. Placeholder for the velocity vector (solution)

Tips and Tricks

- It is important that the ***getInverse(...)*** function indicates success through its return value, as this is checked by the ***solve(...)*** function. At the very least it should always return true.
- While it might be a bit tricky, thread-safety should be encouraged throughout the implementation.

5 Detailed API

Since this is only a pre-release version, the reader is asked to refer to the in-code documentation (particularly comments in the header files) for detailed information. Future releases will contain a separate section with the full API.