The EXtensible Optimisation Toolset

EXOTica vENOM (v1.0.0)

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 ease of integration with ROS, the build system is catkin 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 sections 3 and 4: optionally 2 is useful 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 5.
- Finally **maintainers** should refer to the API section (6, TODO) 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):

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

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):

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

Of course there can be arbitrary many levels of chaining.

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

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

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, but this is up to the user) 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 ***IKTask*** class, as well as the associated ***ISPosSolver*** and ***ISVelSolver*** solvers (implementation of these is indeed provided as part of the standard library). 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 desirable, 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:

```
exotica::OptimisationFunction optim_func; //!< My optimisation
    function
  Eigen::VectorXd config_weights = Eigen::VectorXd::Ones(
    dimension_)*1e-3;//Set R
  std::string optim_name = "pos_IK"; //Set Name
  optim_func.setParams(&optim_name, &config_weights); //by
    Pointer
```

The name member is merely for convenience, while the regularisation weights are taken to be equal and quite small. Note that in the current implementation, these must be passed via pointers (allowing the user to specify NULL and thus keep the internal value unchanged), which is why we had to create the variables outside the function call.

The task can now be defined using the ***add_task(...)*** method:

```
if (optim_func.addTask("IKTask", "position", exotica::
    default_params) != True)
{
  cout << "Oops! Unable to create task of type IKTask" << endl;
  return;
}
```

This function takes in three parameters:

1. The Type of the Task, identified by its name. In EXOTica, each Task implementation has a unique name. Refer to Section 4 for a list of Tasks within the standard library.

2. The name that we will use to access this task: this can be any unique string.

3. The Optimisation Parameters. These are optimisation-wide parameters, which currently (in EXOTica v1.0.0) consist of:

   - **optimisation_window** : the size of the trajectory over which to optimise: for the IK task this is by definition unity.
   - **max_iter** : The maximum number of iterations when searching for a solution.
   - **max_step** : The maximum step size to interpolate over at every iteration (for a complete optimisation cycle). A negative value indicates there should be no scaling.

default_params is a static instantiation of the above structure with a window size of 1, 20 maximum iterations and a step-size of 0.002. Otherwise, it can be defined as necessary.

In addition each task needs values for the intra- and inter-task weights. Access to the task is through a ***boost::weak_ptr*** to enforce thread-safety.

```
{
  boost::shared_ptr<exotica::IKTask> task_ptr = boost::
    dynamic_pointer_cast<exotica::IKTask> (optim_func.access("
    position").lock());
  if (task_ptr.get() != nullptr) {
    if (!task_ptr->setGoalWeights(Eigen::MatrixXd::Identity(3,3)
    )) { return false; } //All goals equal priority
    if (!task_ptr->setTaskWeight(1.0)) { return false; } //Fail
  }
}
```

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, and should be set at this point (again refer to section 4 for specifics).

Once this is fully initialised, it can be pushed back into a vector[1]:

```
std::vector<exotica::OptimisationFunction> my_optimisation = {
    optim_func};
```

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, as this represents the priority level!

3.4 Solving the Optimisation Problem

Once the task is defined, the solution can proceed. We first need to instantiate a position solver and velocity solver:

```
boost::shared_ptr<exotica::ISPosSolver> ik_pos(new exotica::
    ISPosSolver(default_params));
boost::shared_ptr<exotica::ISVelSolver> ik_vel(new exotica::
    ISVelSolver(default_params));
```

There are a few things to note:

- Both solvers are initialised from the same value of optimisation parameters. This is important for correct functioning of the library.
- A shared pointer of each object is instantiated. This is necessary **only** for the velocity solver, for reasons which will become clear below.

Depending on the type of solvers, additional parameters may need to be set: please refer to the appropriate documentation.

The final step is to call the ***solve(...)*** function:

```
Eigen::VectorXd joint_solution;

if (!ik_pos.solve(my_optimisation, Eigen::VectorXd::Zeros(7), &
    ik_vel, joint_solution)) { return false; }
```

[1]Here we use the C++11 initialiser list format.

The function takes four arguments:

1. The vector of optimisation functions
2. The initial configuration
3. Shared Pointer to the velocity solver: this is required to allow for polymorphism of inherited velocity solver implementations[2].
4. The vector where to store the configuration-solution.

Again, once the solution is achieved, a new goal may be set and solutions queried once more. The library is optimised such that data reuse improves efficiency (most often, we do no need to change weightings/task specifications between queries).

[2]This is also the reason why the velocity solver was created through a shared pointer in the first place. It is important that this is so, because if we just pass the address of a stack-variable, it will try to delete the memory after the function exits which is illegal given that the variable would not have been dynamically allocated.

4 Library Implementations

This section lists a number of implementations which come bundled as part of the standard library, with Tasks, Velocity and Position solvers being listed independently. As much as possible, these modules are interchangeable and indicated if this is not so. Information on general usage is provided, together with dependencies (outside exotica) and initialisation details. Refer to the appropriate documentation for more details.

4.1 Task Definitions

Task Definitions are defined hereunder. The unique string identifier is given in bold blue.

IKTask	***I**nverse-**K**inematics **Task***
Class:	ex_ik/IKTask
Description:	Implements Positional Inverse Kinematics based on Geometric Jacobian for arbitrary robot end-effectors.
Dependencies:	Kinematica
Initialisation:	Two signatures, one taking a urdf file and another a pre-initialised KDL::Tree, together with a set of Solution parameters: refer to the Kinematica Documentation. Both return indication of Success/Failure (True/False respectively).
Notes:	By definition an IK-Task only defines a single-time solution (initialisation parameters are ignored).

```
bool initTask(const std::string &, const
    kinematica::SolutionForm_t)
bool initTask(const KDL::Tree &, const
    kinematica::SolutionForm_t)
```

4.2 Velocity Solvers

These are the velocity solvers which are currently implemented.

ISVelSolver	***I**terative **S**ingle-point **Vel**ocity **Solver***
Class:	ex_iss/ISVelSolver
Description:	Implements a velocity solver for one-shot trajectories (although may iterate to achieve at solution). Uses the Moore-Penrose pseudo-inverse.
Dependencies:	None
Initialisation:	Need to set the inverse format through: `void setPinvMode(ISVel_pinv_t)` The Parameter can take on three values: • IS_PI_LEFT : Use the Left Pseudo-Inverse $(J^TWJ + R)^{-1}J^TW)$ • IS_PI_RIGHT : Use the Right Pseudo-Inverse $(W^{-1}J^T(JR^{-1}J^T + W^{-1})^{-1})$ • IS_PI_AUTO : Let the solver decide based on numerical stability.
Notes:	Since this is a one-shot solver, indexing parameters are ignored.

4.3 Position Solvers

These are the velocity solvers which are currently implemented.

ISPosSolver	***I**terative **S**ingle-point **P**osition **S**olver*
Class:	ex_iss/ISPosSolver
Description:	Implements a position solver for one-shot trajectories, but iterates over the solution until convergence. Uses a Newton-Raphson style of search, but scales down intermediate velocity solutions to avoid instabilities.
Dependencies:	None
Initialisation:	Default
Notes:	Since this is a single-position solver, it does not use the index value.

5 Extending the Library

One of the major goals of this project is 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.

5.1 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 the single pure-virtual function.

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

The function takes as inputs the vector of current configuration and the index into the trajectory vector. Ideally the latter parameter should default to 0. The function should:

1. Compute the Forward mapping, given the current configuration and store it in the appropriate phi-matrix (use the ***setPhi(..)*** method).

2. Compute the Jacobian and store it in the appropriate Jacobian matrix (use the ***setJacobian(...)*** method).

3. Indicate whether all computations were successful (True) or not (False) through the return value.

Registering the Task

In order to allow dynamic instantiation of the inherited classes through the ***OptimisationFunction::addTask(...)*** method, the library uses object factories. These however must be provided with knowledge of the inherited classes itself. A macro is provided for this:

```
REGISTER_TASK_TYPE ("TaskName", TaskType)
```

The first parameter must be a **unique** string identifier which users will use to refer to the class. The second is the actual class name. This line must appear within the class implementation file (**.cpp*) and **not** the header file.

Tips and Tricks

- Ideally, (unless it is irrelevant such as for IK) support for multi-point trajectory optimisation should be maintained through the use of vectors for all parameters.

- 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.

- Ideally, inherited objects should use default arguments for the index variable to ease implementation of single-point trajectories.

5.2 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: the reason for this is to allow polymorphic use of different velocity solvers.
4. Configuration vector for the solution

Ideally it should indicate success or failure through the return variable: note however that this success/failure is only at the level of computation and should **not** indicate whether the goal was reached or not. 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.
- In the interest of efficiency, it was decided to put the responsibility of updating the task variables within the Position Solver rather than the Velocity Solver object. In fact, the only communication to the velocity solver is the optimisation function which must be completely defined. Specifically, the position solver should call the ***updateTask()*** on the optimisation function before calling on the Velocity Solver.

5.3 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. A default implementation is provided for the **solve** function to handle 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:

```
bool solve(std::vector<OptimisationFunction> &, Eigen::VectorXd
    &, int=0)
```

with parameters:

1. The Vector of optimisation functions (pre-initialised and updated).
2. Placeholder for the resulting **velocity vector**.
3. Integer index: defaults to 0.

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.

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.

6 Detailed API

For this version of documentation, 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.

A Limitations

- Currently, there is no object-factory concept for the position/velocity solvers
- Initialising Optimisation Functions is still pretty painful.

B Version History

Version	**Library Changes**	**Documentation Changes**
0.0.1	• First Iteration of EXOTica	• First Documentation
1.0.0 (Venom)	• First Official Release • Changed API for updating the Task state to a single function rather than individually for the Jacobian/Forward Map. • Fixed bug with calling setter functions on uninitialised vector of task parameters. • In the interest of efficiency, state updating was moved to the PositionSolver: hence the configuration variable for the VelocitySolver was removed.	• Changed Sections 3 and 5. • Added Section 4 on specific implementations.

1.0.0 (Venom)	• Moved initialisation of optimisation parameters to within an initialiser constructor and removed default constructor. However, the create function has a default argument for the window (being 1). Similarly modified the ***OptimisationFunction::addTask()*** to reflect this, however, without default parameters. There is however a default initialisation for the *OptimisationParameters_t* (a **const static** instantiation called ***default_params***) • Added function to get configuration dimension to the OptimisationFunction class. Also changed internal storage of configuration weights to be matrix rather than vector. • Fixed bug whereby when checking the dimensionality of the configuration or task space I was asking for ***size()*** rather than ***rows()*** in a 2D matrix. • Fixed bug in computation of task error within the *OptimisationFunction* class.	