The EXtensible Optimisation Toolset

EXOTica Kinky Slinky (v 2.0.0)

Application Programming Interface

Michael Camilleri

Abstract

This document describes the Implementation of the EXOTica Library. The toolset aims to simplify the 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-usability.

Contents

1 Introduction **3**
- 1.1 Document Structure . . . 3
- 1.2 The EXOTica Library . . . 3

2 Setup **5**
- 2.1 Package Organisation . . . 5
- 2.2 Pre-requisites . . . 5
- 2.3 Installation . . . 5

3 Introductory Optimisation Theory **6**
- 3.1 Notation . . . 6
- 3.2 Cost Functions . . . 7
- 3.3 Optimisation . . . 8

4 Solving Optimisation Problems **10**
- 4.1 Architectural Overview . . . 10
- 4.2 Boiler-Plate . . . 11
- 4.3 Initialisation . . . 11
- 4.4 Solving the Optimisation Problem . . . 14

5 Library Implementations **15**
- 5.1 Task Definitions . . . 15
- 5.2 Velocity Solvers . . . 16
- 5.3 Position Solvers . . . 17

6 Extending the Library **18**
- 6.1 Creating Task-Definition Classes . . . 18
- 6.2 Creating Position Solvers . . . 19
- 6.3 Creating Velocity Solvers . . . 20

7 Detailed API **23**

A Limitations **24**

B Version History **24**

C Listings **27**

1 Introduction

This document describes the EXOTica library, a toolset for Kinematics and Dynamics optimisation problems for Robotic Systems.

1.1 Document Structure

- General **Users** should read sections 4 and 5: 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 functionality should also read section 6.
- Finally **maintainers** should refer to the API section (7, TODO) as well as the doxygen comments in the code.

1.2 The EXOTica Library

The EXOTica library is a generic Optimisation Tooolset for Robotics platforms, written in C++. It provides a framework for developing algorithms for Inverse-Kinematics and Trajectory Optimisation (and potentially Dynamic optimisation as well). Its design advocates:

- **Modularity** The library is developed in a modular manner making use of C++'s object-oriented features (e.g. polymorphism). This allows users to *plug in* (mix'n match) different modules depending on their needs.
- **Extensibility** The library is 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. The end result is that users can work around the existing framework by *extending* the library with new functionality to suit their needs, rather than *rewriting* everything from scratch.
- **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 the Robot Operating System (ROS), the build system is catkin but the library itself has no dependency on ROS and indeed, can be built using native compilers.

- **Integration with ROS** Although it is not necessary, the library is fully integrate-able with ROS.

2 Setup

2.1 Package Organisation

EXOTica is currently organised in two (catkin) packages, plus an optional testing package.

- **exotica** contains the core library components and the abstract interface.
- **exotations** (short for **exot**ica implement**ations**) holds the concrete implementations (currently Inverse-Kinematics) for the library.
- **testing_pkg** implements a series of automatic tests following the Google Test conventions.

2.2 Pre-requisites

The library currently has the following dependencies:

1. **Eigen** Library for matrix manipulation, specifically Eigen 3.0. Refer to http://eigen.tuxfamily.org/ for more information. Used throughout the library.
2. **Boost** to enforce thread-safety and prevent memory leaks. Please refer to http://www.boost.org/. Used throughout the library.
3. **Kinematica** for Kinematics problems (Forward and Inverse kinematics). A separate project by our group (http://wcms.inf.ed.ac.uk/ipab/slmc/research/EXOTi Used by the *IKTask* implementation.
4. **googletest** used within the testing package. Refer to https://code.google.com/p/googletest/

In addition, the testing package currently relies heavily on ROS (Hydro).

2.3 Installation

The current supported build system is Catkin from ROS: refer to the catkin tutorial at http://wiki.ros.org/catkin. One thing to note is that the library should be built using the C++11 compiler: this can be enforced by adding to the CMakeLists.txt:

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++0x")
```

Alternatively, one may build the library (excluding the testing package) using another compiler: in this case, ensure that you link against the above libraries.

3 Introductory Optimisation Theory

This section is a refresher of Optimisation problems and conventions. It aims to be as generic as possible, but wherever concrete examples are needed, reference is made to the task of solving Inverse Kinematics (IK).

3.1 Notation

Throughout this document we will use the following notation:

- Scalars will be represented by lower-case italic symbols (x).
- Vectors are denoted by **bold** lower-case symbols ($\mathbf{x}$) indexed via a super-script ($\mathbf{x}^i$).
- Matrices are denoted by **bold** UPPER-CASE symbols ($\mathbf{X}$) indexed by a double super-script ($\mathbf{X}^{i,j}$).
- Time-domain indexing will be indicated by a subscript, as a function of t (s.a. x_{t-1}): a discrete trajectory of T steps is defined as a set of states from $t = 0$ to $t = T - 1$.
- In contrast, a super-script ^{T} denotes a transpose.
- A desired value for a state (such as a goal-state) is indicated by an asterisk (x^*).

In our setting, we describe a problem by Fig. 1:

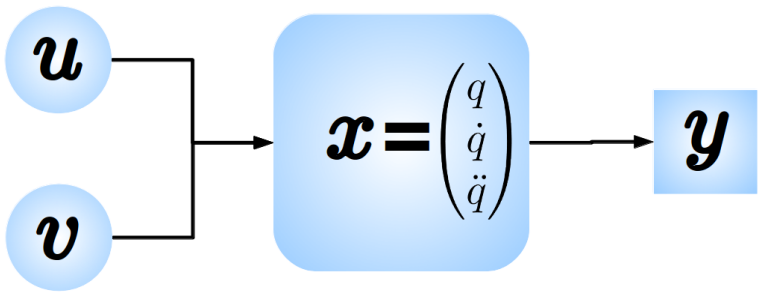

Figure 1: Problem Representation within EXOTica

where we define:

- $\mathbf{u}$: The Controllable inputs to the system (*s.a. the joint commands in IK*).

- $\mathbf{v}$: The External inputs (*such as the positions of obstacles*).
- $\mathbf{q}$: The **Configuration** of the system (*for IK, this translates to exactly the joint values or* u).
- $\mathbf{x}$: The State of the system, within the **Configuration** space: this could consist of q and higher derivatives (typically with respect to time).
- $\mathbf{y}$: The Output of the system, in **Task** space co-ordinates (*s.a. position of end-effectors*).
- ϕ : The Forward Mapping from inputs and possibly current configuration to the task space: i.e.

$$y_t = \phi(\mathbf{x_{t-1}}, \mathbf{u_t}, \mathbf{v_t}) \tag{1}$$

3.2 Cost Functions

The Optimisation problem is usually defined in terms of a cost function at each time step t. Generally, we wish to achieve N simultaneous goals in the task space: we represent this by a set of N systems/tasks of the form of Fig. 1, which have the same inputs (u and v) but different outputs (y) and hence different ϕ functions[1], as illustrated in Fig. 2

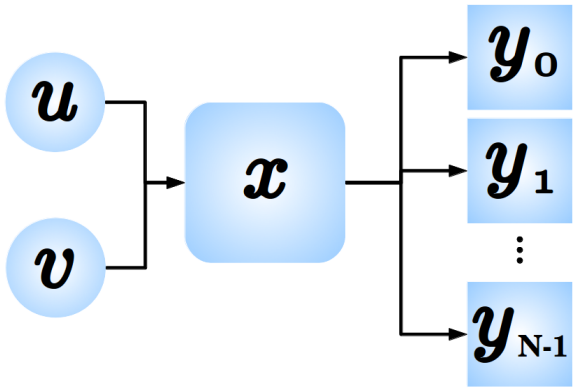

Figure 2: Multi-Task formulation for EXOTica

[1]Actually, the difference will **only** be in the mapping from $\mathbf{x}_t$ to $\mathbf{y}_t$, with the mapping from ($\mathbf{x}_{t-1}$, $\mathbf{u}_t$, $\mathbf{v}_t$) to $\mathbf{x}_t$ being common.

Weighted Optimisation

In order to optimise, we require (at minimum) a cost for each time-step. We define our error function $\mathbf{E}$ at time t as:

$$\mathbf{E}_t(\mathbf{u_t}, \mathbf{v_t}, \mathbf{x_t}, \mathbf{y}^*) = \mathbf{f}(\mathbf{u}_t, \mathbf{u}_{t-1}, \mathbf{R}) + \sum_{n=0}^{N-1} \omega_n \mathbf{g}_n \left(\mathbf{y}^*, \mathbf{y_t}, \mathbf{W_n}\right) \tag{2}$$

where:

- $\mathbf{f}$ is the regularisation cost, which with weighting matrix $\mathbf{R}$ penalizes excessive deviations from the previous inputs (and thus *improves* stability).
- $\mathbf{g}_n$ is the individual task-error function, which evaluates the cost of deviations from the goal for task n, with internal weighting (between the dimensions of the task) governed by $\mathbf{W}_n$.
- ω_n serves to weight the tasks relative to each other.

Prioritised Optimisation

While allowing relative weighting between tasks/dimensions, Eq.2 is still somewhat restrictive. In certain cases we may want a prioritised solution, meaning that a subset of goals are achieved, subject to first satisfying other goals. This is usually achieved through null-space resolution: we therefore get a function of the form of Eq

$$\mathbf{C}_t = \mathbf{E}_t^0 - 2(\mathbf{E}_t^1)^T \mathbf{R}\mathbf{u}_t \tag{3}$$

Of course there can be arbitrary many levels of chaining.

3.3 Optimisation

Optimisation of Eq.3 can proceed in a number of ways ranging from random Monte-Carlo like techniques to deterministic solutions. We present here one of the most common cases used, namely the Linear-Quadratic-Regulator (LQR) solver, for which the library is most suited: however, with some added effort, other techniques can also be used.

Restrictions

In LQR, we restrict (or assume) that:

1. The forward mapping of the system is linear. In actual case we can usually approximate this for a small change in control inputs $\delta\mathbf{u}$:

$$\lim_{\delta u \to 0} \phi(\mathbf{u_t} + \delta\mathbf{u}) = \phi(\mathbf{u_t}) + \mathbf{J}(\mathbf{u_t})\delta\mathbf{u} \tag{4}$$

   where we have dropped the implicit dependence of the function on the previous state ($\mathbf{x_t}$) and uncontrollable inputs ($\mathbf{v_t}$). In this case, $\mathbf{J}$, referred to as the Jacobian, is the first derivative of the forward mapping ϕ, i.e.:

$$\mathbf{J}\left(\mathbf{u_t}\right) = \frac{\partial \phi}{\partial \mathbf{u_t}} \tag{5}$$

2. The cost functions, $\mathbf{f}$ and $\mathbf{g}$ are quadratic in nature, of the form:

$$h = ||\mathbf{z}^* - \mathbf{z}||_{\mathbf{M}} \tag{6}$$

   where $\mathbf{M}$ is the weighting matrix and $\mathbf{z}$ the task vector.

The Solution

Optimisation of Eq.3 under the constraints imposed by 4 and 6 results in the following deterministic solution:

$$\mathbf{u_t} = \mathbf{u_{t-1}} + \mathbf{J}^\sharp\left(\mathbf{y}^* - \mathbf{y}_{t-1}\right) + \left(\mathbf{I} - \mathbf{J}^\sharp\mathbf{J}\right)\mathbf{h} \tag{7}$$

where

- with a slight abuse of notation, the $\mathbf{y}$ here signifies the '**Big**' task-vector, which incorporates all the tasks at the same priority level.
- $\mathbf{J}^\sharp$ is the Jacobian pseudo-inverse, which depends on the task weighting matrices and regularisation matrices, besides the current configuration (for approximation of non-linear tasks). In effect, this is again the '**Big**' Jacobian, which incorporates all tasks at the same level.
- $\mathbf{h}$, which is the null-space motion, is the solution (in input-space $\mathbf{u}$) provided by tasks at a lower priority level. This is cast into the null-space of the current solution.

4 Solving Optimisation Problems

This Section introduces the use of the library in as generic a way as possible, following the theory laid down in 3

4.1 Architectural Overview

EXOTica is implemented as a hierarchical structure, as displayed in Fig.3.

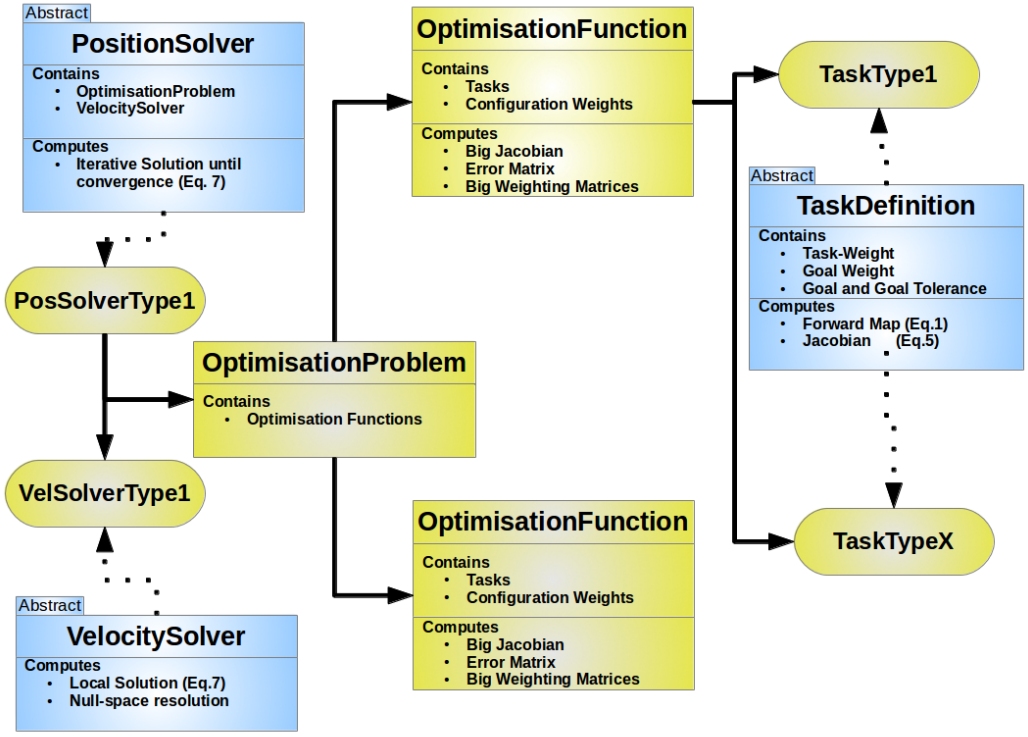

Figure 3: Architecture of the EXOTica library

At the lowest level sits the abstract **TaskDefinition** object. This serves as a placeholder for the goal-vector ($\mathbf{y}^*$), the goal weights ($\mathbf{W_n}$) and the inter-task weight (ω_n). All concrete implementations inherit from it and implement at minimum specific computations for ϕ (Eq.1) and the Jacobian (Eq.5).

A set of tasks at the same priority level are encapsulated into an **OptimisationFunction** object. This provides a wrapper around a set of tasks at the same priority level (representing Eq.(2)). In turn, a hierarchy of Optimisation Functions is contained within an **OptimisationProblem**, by storing

them in a user-defined order in an *std::vector*: in this case, the lower indices always have higher priority.

Finally, the solver is (usually) implemented in two stages:

- A **VelocitySolver** computes the incremental steps in converging towards the goal. It is also the responsibility of the velocity solver to resolve the priority levels and weightings of the problem. This amounts to computing the last two terms of Eq.(7).
- The task of the **PositionSolver** is then to (possibly) iterate and integrate velocities until the solution converges.

4.2 Boiler-Plate

Generally, including the main header file is enough:

```
#include <exotica/EXOTica.hpp>
```

Additionally, you may need to include implementation-specific headers for functionality which you implemented.

Using the library involves defining the Optimisation problem (basically Eq.(3)) and then solving. As an example, Inverse Kinematics will be used: we assume an implemented ***IKTask*** together with ***ISPosSolver*** and ***ISVelSolver*** solvers[2].

4.3 Initialisation

The library currently provides two modes of initialisation, either through XML initialisation scripts or manually through setters and getters.

Initialisation through XML

This is by far the easiest and preferred method for initialising the problem, since it takes care of everything, and also provides a great deal of flexibility. A symplified XML-file format appears below as Listing 1, while a more complete example for the IK-scenario is in the Appendix (Listing 4).

The format is identified by the **Exotica** keyword, which must appear as the root element in every file (line 2). Its **only** children must be:

[2]Implementation of these is indeed provided as part of the standard library.

- The **OptimisationParamaters** : currently specifies only the window size of the trajectory (lines 3-5).
- The **PositionSolver** definition (lines 7-35).

Note that these can however appear in any order.

```
1  <?xml version="1.0" ?>
2  <Exotica>                        <!--Compulsary-->
3    <OptimisationParameters>  <!--Compulsary-->
4      <Window>1</Window>
5    </OptimisationParameters>
6
7    <PositionSolver type="ISPosSolver">   <!--PositionSolver-->
8      <VelocitySolver type="ISVelSolver"> <!--VelocitySolver-->
9        <!-- Any parameters -->
10     </VelocitySolver>
11
12     <OptimisationFunction name="Level_0"> <!--Function
       Definition -->
13       <ConfigWeights dim="2">0.001 0.0 0.0 0.001</ConfigWeights>
14
15       <Task type="IKTask" name="left_hand"> <!--TaskDefinition
       Type -->
16         <TaskParameters>
17           <TimeElement>
18             <GoalWeights dim="2">1.0 0.0 0.0 1.0</GoalWeights>
19             <TaskWeight>0.5</TaskWeight>
20             <Goal tolerance="0.2">0.2 0.4 0.6</Goal>
21           </TimeElement>
22         </TaskParameters>
23         <!--Implementation-Specific parameters-->
24       </Task>
25
26       <OptimisationFunction> <!--Nested function cast into null-
       space -->
27         <ConfigWeights dim="2">0.001 0.0 0.0 0.001</
       ConfigWeights>
28
29         <Task type="ExoticaTask_1" name="right_hand">
30           <!-- :: -->
31         </Task>
32
33       </OptimisationFunction>
34     </OptimisationFunction>
35   </PositionSolver>
36 </Exotica>
```

Listing 1: Example XML-Specification

The **PositionSolver** element encapsulates the complete problem. It may *optionally* indicate use of a **VelocitySolver** (lines 8-10), which may in turn require specific parameters. In both cases the *type* of solver(s) must be specified as an attribute. The **PositionSolver** element should also contain reference to a *single* **OptimisationFunction**, which is optionally named. This element defines the actual problem (lines 12-34). It contains the matrix of configuration weights (line 13)[3] and *at least* one **Task** (lines 15-24).

The **Task** contains reference to the universal task-parameters (goal-weights, goal, tolerance, inter-task weighting) organised as an *ordered* list of **TimeElement**s, one for each time-step of the trajectory (whose size must equal the Window parameter). In additional specific implementations may require additional parameters which must also be specified, but outside the **TaskParameters** tag.

If a priority scheme is required, this is indicated by a nested **OptimisationFunction** element, which may in turn contain other such elements for a complex nesting problem.

One final thing to note is that in general the order in which elements appear is irrelevant except for:

- The Order of nesting of **OptimisationFunction**s defines the null-space priority resolution
- The Order of **TimeElement**s within the **Task** is associated with the parameters at each time-step of the trajectory.

Once the XML-specification is defined, EXOTica provides an initialiser function which does all the work for you.

```
boost::shared_ptr<exotica::PositionSolver> pos_solv_ptr;
tinyxml2::XMLDocument document;

if (document.LoadFile("path_to_spec") == tinyxml2::XML_NO_ERROR)
{
  tinyxml2::XMLHandle xml_handle(document);
  if (exotica::initialiseSolver(xml_handle, pos_solv_ptr_))
  {
    //!< Execute
  }
}
```

[3]Note that matrix definitions require a **dim**ension attribute and should be listed in row-major order.

The above code creates a Boost smart pointer to a **PositionSolver** object and then attempts to load the XML specification (checking that it is correctly formatted in the process). The ***initialiseSolver(...)*** function in turn, takes an XMLHandle and the shared pointer in which to store the solver. It return true if everything is successful and false otherwise.

Manual Initialisation

Coming Soon

4.4 Solving the Optimisation Problem

Coming Soon

5 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.

5.1 Task Definitions

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

IKTask	***Inverse-Kinematics 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).

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

Notes:	By definition an IK-Task only defines a single-time solution (initialisation parameters are ignored).

5.2 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 $(J^T W J + R)^{-1} J^T W)$
- IS_PI_RIGHT : Use the Right Pseudo-Inverse $(W^{-1} J^T (J R^{-1} J^T + W^{-1})^{-1})$
- IS_PI_AUTO : Let the solver decide based on numerical stability.

Notes: Since this is a one-shot solver, indexing parameters are ignored.

5.3 Position Solvers

These are the velocity solvers which are currently implemented.

ISPosSolver	***I**terative **S**ingle-point **Pos**ition **Solver***
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.

6 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.

6.1 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 pure-virtual functions:

```
bool updateState(const Eigen::VectorXd &, int)
bool initDerived(tinyxml2::XMLHandle &)
```

The first 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.

The second function is used to initialise any implementation-specific parameters for the task. If it is not required, it should only return true.

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.

6.2 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.

6.3 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.

7 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 4 and 6. • Added Section 5 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.	

2.0.0 (Kinky Slinky)	• Completely re-organised the architecture of the library with respect to containment. • Created the OptimisationProblem wrapper around the vector of OptimisationFunctions. • Implemented XML-based initialisation for all classes. • Added factories for the Position and Velocity Solvers. • Added JointLimits Task (not documented yet). • Extensively tested.	• Partially modified documentation (wip).

C Listings

```
<?xml version="1.0" ?>
<Exotica>
  <OptimisationParameters>
    <Window>1</Window>
  </OptimisationParameters>

  <PositionSolver type="ISPosSolver">
    <VelocitySolver type="ISVelSolver">
      <PinvMode>IS_PI_AUTO</PinvMode>
    </VelocitySolver>

    <OptimisationFunction name="Level_0">
      <ConfigWeights dim="2">0.001 0.0 0.0 0.001</ConfigWeights>

      <Task type="IKTask" name="left_hand">
        <TaskParameters>
          <TimeElement>
            <GoalWeights dim="3">1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
     1.0</GoalWeights>
            <TaskWeight>0.5</TaskWeight>
            <Goal tolerance="0.2">0.2 0.4 0.6</Goal>
          </TimeElement>
        </TaskParameters>

        <Urdf>robot.urdf</Urdf>

        <Root segment="root">
          <vector>0.0 0.0 0.0</vector>
          <quaternion>1.0 0.0 0.0 0.0</quaternion>
        </Root>

        <Update zero_unnamed="true">
          <joint name="joint_1"/>
          <joint name="joint_2"/>
        </Update>

        <EndEffector ignore_unused="true">
          <limb segment="left_hand">
            <vector>0.0 0.0 0.0</vector>
            <quaternion>1.0 0.0 0.0 0.0</quaternion>
          </limb>
        </EndEffector>
      </Task>

      <OptimisationFunction>
```

```
45          <ConfigWeights dim="2">0.001 0.0 0.0 0.001</
    ConfigWeights>

47          <Task type="ExoticaTask_1" name="right_hand">
              <TaskParameters>
49              <TimeElement>
                  <GoalWeights dim="3">1.0 0.0 0.0 0.0 0.5 0.0 0.0
    0.0 0.8</GoalWeights>
51                <TaskWeight>1.0</TaskWeight>
              </TimeElement>
53          </TaskParameters>
          </Task>
55      </OptimisationFunction>
    </OptimisationFunction>
57  </PositionSolver>
</Exotica>
```

Listing 4: Example XML-Specification