EXOTica: An Extensible Optimization Toolset for Prototyping and Benchmarking Motion Planning and Control

Vladimir Ivan, Yiming Yang, Wolfgang Merkt,
Michael P. Camilleri, Sethu Vijayakumar

School of Informatics, The University of Edinburgh
10 Crichton Street, EH8 9AB, Edinburgh, UK
v.ivan@ed.ac.uk
yiming.yang@ed.ac.uk
wolfgang.merkt@ed.ac.uk
michael.p.camilleri@ed.ac.uk
sethu.vijayakumar@ed.ac.uk
https://www.ed.ac.uk/informatics

Abstract. In this research chapter, we will present a software toolbox called EXOTica that is aimed at rapidly prototyping and benchmarking algorithms for motion synthesis. We will first introduce the framework and describe the components that make it possible to easily define motion planning problems and implement algorithms that solve them. We will walk you through the existing problem definitions and solvers that we used in our research, and provide you with a starting point for developing your own motion planning solutions. The modular architecture of EXOTica makes it easy to extend and apply to unique problems in research and in industry. Furthermore, it allows us to run extensive benchmarks and create comparisons to support case studies and to generate results for scientific publications. We demonstrate the research done using EXOTica on benchmarking sampling-based motion planning algorithms, using alternate state representations, and integration of EXOTica into a shared autonomy system. EXOTica is an open-source project implemented within ROS and it is continuously integrated and tested with ROS Indigo and Kinetic. The source code is available at `https://github.com/ipab-slmc/exotica` and the documentation including tutorials, download and installation instructions are available at `https://ipab-slmc.github.io/exotica`.

Keywords: motion planning, algorithm prototyping, benchmarking, optimization

1 Introduction

The ROS community has developed several packages for solving motion planning problems such as pick-and-place (MoveIt! [1]), navigation [2], and reactive

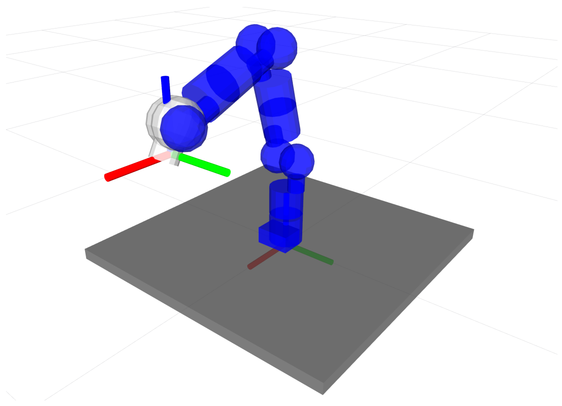


Fig. 1. Example task of computing the inverse kinematics for a robot arm.

obstacle avoidance [3]. These tools are easily accessible to the end-user via standardized interfaces but developing such algorithms takes considerable effort as these interfaces were not designed for prototyping motion planning algorithms. In this chapter, we present EXOTica, a framework of software tools designed for development and evaluation of motion synthesis algorithms within ROS. We will describe how to rapidly prototype new motion solvers that exploit a common problem definition and structure which facilitates benchmarking through modularity and encapsulation. We will refer to several core concepts in robotics and motion planning throughout this chapter. These topics are well presented in robotics textbooks such as [4] and [5]. This background material will help you to understand the area of research that motivated development of EXOTica.

Our motivation to begin this work stems from the need to either implement new tools or to rely on existing software often designed for solving a problem other than the one we intended to study. The need to implement and test new ideas rapidly led us to the specification of a library that is modular and generic while providing useful tools for motion planning. A guiding principle hereby is to remove implementation-specific bias when prototyping and comparing algorithms, and hitherto create a library of solvers and problem formulations.

In this chapter, we will use a well-known algorithm as an example in order to explain how to use and extend the core components of EXOTica to explore novel formulations. Consider a robot arm mounted to a workbench as shown in Figure 1. The arm consists of several revolute joints actuated by servo motors moving the links of the robot body. A gripper may be attached to the final link. The task is to compute a single configuration of the robot arm which will place the gripper at a desired grasping position—i.e. our example will follow the implementation of an inverse kinematics solver. Once the problem and motion solver have been implemented, we can compute the robot configuration using EXOTica with the following code:

```
#include <exotica/Exotica.h>
using namespace exotica;

int main(int argc, char **argv)
{
  MotionSolver_ptr solver = XMLLoader::loadSolver("{
      exotica_examples}/resources/configs/example.xml");
  Eigen::MatrixXd solution;
  solver->Solve(solution);
}
```

This snippet shows how little code is required to run a motion planning experiment. We load a motion solver and a problem definition from an example configuration file located in the `exotica_examples` package, allocate the output variable, and solve the problem using three lines of code.[1] What this snippet does not show is the definition of the planning problem, the implementation of the algorithm and an array of other tools available in EXOTica. This code and most of the rest of this chapter will focus on motion planning. However, we view motion planning and control as two approaches to solving the same motion synthesis problem at different scales. For example, the problem in Figure 1 could be viewed as an end-pose motion planning problem as well as operational space control, when executed in a loop. This allows us to formulate complex control problems as re-planning and vice versa. EXOTica provides the tools to implement such systems.

To motivate and explain the EXOTica software framework, we focus on how it can be used in research and prototyping. The first part of this chapter will describe how problems and solvers are defined, and it will provide an overview of the library. The second part of the chapter will demonstrate how EXOTica has been used to aid motion planning research and it will help you to understand how EXOTica may be useful for your research and development.

2 Overview

Prototyping of novel motion planning algorithms relies on defining mathematical models of the robotic system and its environment. To aid this process, EXOTica provides several abstractions and generic interfaces that are used as components for building algorithms. Figure 2 shows the three components central to algorithm design in EXOTica: (1) a *planning scene*, providing tools to describe the state of the robot and the environment, (2) a *planning problem* formally defining the task, and (3) a *motion solver*. These abstractions allow us to separate problem definitions from solvers. In particular, motion solvers implement algorithms such as *AICO* [6] and *RRTConnect* [7]. These implementations may perform

[1] This and other basic examples with detailed explanation of each line of code, as well as instruction how to download, compile and run them are available in our online documentation at `https://ipab-slmc.github.io/exotica/Installation.html`.

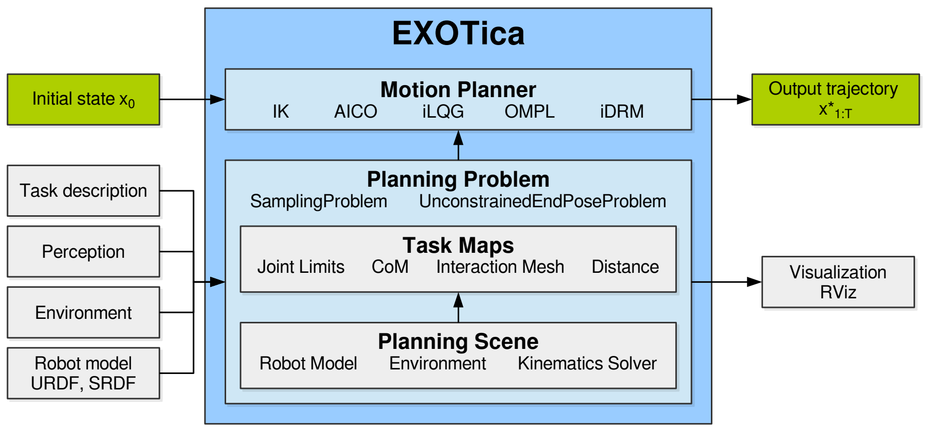


Fig. 2. The core concept of EXOTica highlighting the interplay of the planning scene, problem solver, and efficient batch kinematics solver. Problem and task definitions are generic across solvers and can be loaded easily from configuration files as dynamic plug-ins without the need to recompile. Furthermore, the same problem can be solved using a variety of baseline algorithms provided by EXOTica for benchmarking and comparison.

optimization, randomized sampling, or any other computation which requires a very specific problem formulation.

How the problem is formulated is fully contained within the definition of a *planning problem*. Each algorithm solves exactly one type of motion planning problem while one type of problem may be compatible with multiple solvers. As a result, several algorithms can be benchmarked on the exact same problem. When benchmarking two algorithms that are compatible with different types of problems, the problems have to be converted explicitly. This is a useful feature that makes it easy to track differences between problem formulations that are intended to describe the same task.

All planning problems use the *task maps* as components to build cost functions, constraints, or validity checking criteria. Task maps perform useful computations such as forward kinematics, center-of-mass position calculation, and joint limit violation error computation. To further support the extensibility of EXOTica, the motion solvers and the task maps are loaded into EXOTica as plug-ins. As such, they can be developed separately and loaded on demand. One such example is the plug-in which wraps the sampling-based algorithms implemented in the OMPL library [8].

Figure 2 also shows the *planning scene* which separates the computation of kinematics from the computation of task related quantities.

3 System model

To synthesize motion, we describe the system consisting of the robot and its environment using a mathematical model. This system model may be kinematic or it may include dynamic properties and constraints. EXOTica uses the system model to evaluate the state using tools implemented inside the *planning scene*. The diagram in Figure 2 shows the *planning scene* as a part of the planning problem where it performs several computations required for evaluating the problem.

3.1 Planning scene

The *planning scene* implements the tools for updating and managing the robot model and the environment. The robot model is represented by a kinematic tree which stores both the kinematic and dynamic properties of the robot, e.g. link masses and shapes, joint definitions, etc. The environment is a collection of additional models that are not part of the robot tree but that may interact with the robot. The environment may contain reference frames, other simplified models (geometric shapes), and real sensor data based representations such as pointclouds and OctoMaps [9]. The planning scene implements algorithms for managing the objects in the environment (e.g. adding/removing obstacles) as well as computing forward kinematics and forward dynamics.

The system is parametrized by a set of variables that correspond to controllable elements, e.g. the robot joints. The full state of the system is described using these variables and we will refer to it as the *robot state*. In some cases, only a subset of the robot state is controlled. We call this subset the *joint group*. Analogous to the MoveIt! [1] definition of a move group, a joint group is a selection of controlled variables used for planning or control. From now on, whenever we refer to a joint state, we are referring to the state of the joint group.

The system model may be kinematic, kino-dynamic[2], or fully dynamic. The robot state is then described by joint positions, joint positions and velocities, or full system dynamics respectively. The system dynamics may be provided via a physics simulator but this is outside of the scope of this chapter. We will only consider the kinematic model for simplicity.

The system model is implemented as a tree structure mimicking the structure implemented in the KDL library [10]. Figure 3 illustrates the kinematic tree of a planar robot arm. Every node in the tree has one parent and possibly multiple children. The node defines a spatial transformation from the tip frame of the parent node to its own tip frame. Every node consists of a position offset of the joint, a joint transformation, and a tip frame transformation (see the KDL documentation [10]). The joint transformation is constant for fixed joints. The transformations of all joints that belong to the controlled *joint group* are updated based on the joint state. During the update, the local transformation of

[2] Here, we define a kino-dynamic model as one which captures joint positions (kinematics) and joint velocities.

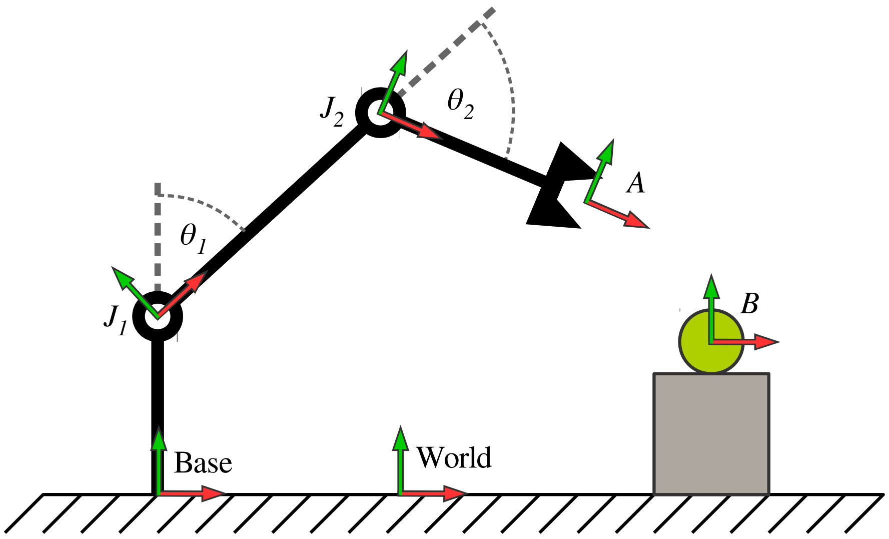


Fig. 3. The planning scene stores the kinematic tree composed of the robot model and the environment. The diagram shows a robot model which has two revolute joints J_1 and J_2 defined by joint angles θ_1 and θ_2 respectively, a base frame and an end effector frame A. A grasping target is located at frame B. The root of the tree is at the world frame. The grasping task can exploit the relative transformation M_A^B.

the node is updated and the transformation of the tip w.r.t. the world frame is accumulated. The nodes of the tree are updated in a topological order (from the root to the leafs). This ensures that the tip frame of the parent node is always updated before its children.

The EXOTica *Scene* implements a method for publishing the frames to RViz [11] using `tf` [12] for debugging purposes. These frames can be visualized using the *tf* and the *RobotModel* plug-ins.[3]

The system model provides an interface to answer kinematic queries. A query can be submitted to the *Scene*, requesting arbitrary frame transformations. Each requested frame has the following format:

– Name of the tip frame (Frame A)
– Offset of the tip frame
– Name of the base frame (Frame B)
– Offset the of base frame

Figure 3 illustrates an example scene. Any existing frame can be used to define a base or a tip frame of a relative transformation. The response to the query will then contain a transformation of the tip frame with respect to the base frame.

[3] Use the `tf` prefix `/exotica` to visualize the robot model.

If an offset is specified, each respective frame will be redefined to include the offset. If a base frame is not specified, the world frame will be used by default. Since all transformations of the tree nodes w.r.t. the world frame have been computed during the update, the query computation only adds the tip frame to the inverted base frame[4] $M_A^B = {M_B^{world}}^{-1} M_A^{world}$. The *Scene* has been designed to answer a large number of requests in batches. While some smaller problems, such as simple kinematic chains, may be more costly to update, larger kinematic trees with a large number of leaf nodes are handled more efficiently by simply iterating over the requested frames.

The system model also computes derivatives of the spatial frames w.r.t. the control variables. These are computed as geometric Jacobians (J) and Jacobian derivatives ($\dot{J}$). The Jacobian has six rows and a number of columns corresponding to the number of controlled joints. Each column represents a spatial velocity in form of a *twist*. The twist ${}^{B}t_A^i$ describes the linear and angular rate of motion of the tip frame A w.r.t. the joint frame i expressed in the base frame B. We use the notation with the *expressed in frame* in the left superscript. Using the twist representation allows us to correctly compute spatial transformations using the Lie group algebra [13].

The kinematic tree represents the robot kinematic model and the objects in the environment. The robot model can be loaded from a pair of MoveIt! compatible URDF and SRDF files. The URDF file specifies the robot kinematics, joint transformations and range of motion, frame locations, mass properties and collision shapes. The SRDF file specifies the base of the robot (fixed, mobile, or floating), joint groups, and collision pairs. The robot configuration created for MoveIt! is fully compatible with EXOTica. The *Scene* also implements an interface to populate the environment with collision objects from MoveIt! planning scene messages and from MoveIt! generated text files storing the scene objects. The *Scene* may load additional basic shape primitives, meshes, or OctoMaps.

In order to perform collision checking, a *CollisionScene* can be loaded as a plug-in into a *Scene*. This allows for different implementations of collision checking algorithms to be used as required and does not tie EXOTica to a particular collision checking library. For instance, by default, EXOTica ships with two *CollisionScene* implementations using the FCL library—one based on the stable FCL version also used in MoveIt! and one tracking the development revision of FCL. The *CollisionScene* plug-ins may hereby implement solely binary collision checking, or additional contact information such as signed distance, contact (or nearest) points, as well as contact point normals. This information is captured and exposed in a so-called *CollisionProxy*.

Referring back to the example inverse kinematics problem, the planning scene consists of the kinematics of the robot with a base link rigidly attached to the world frame. We choose to use a simplified version following the DH parameters of the KUKA LWR3 arm which we load from a pair of URDF and SRDF files. This robot has seven revolute joints. The joint group will consist of all seven

[4] Notation: the subscript and superscript denote tip and base frames respectively. M_A^B reads: transformation of frame A w.r.t. frame B.

joints as we intend to control all of them. We will not be performing collision checking in this experiment. The *planning scene* is initialized from an EXOTica XML configuration file. The XML file contains the following lines related to the setup of the *planning scene*:

```
<PlanningScene>
  <Scene>
    <JointGroup>arm</JointGroup>
    <URDF>{exotica_examples}/resources/robots/
        lwr_simplified.urdf</URDF>
    <SRDF>{exotica_examples}/resources/robots/
        lwr_simplified.srdf</SRDF>
  </Scene>
</PlanningScene>
```

where the joint group parameter selects a joint group defined in the SRDF file by name. The robot model is loaded from the URDF and SRDF files specified here. When the paths are not specified, EXOTica attempts to load the robot model from the `robot_description` ROS parameter by default. EXOTica additionally allows to set ROS parameters for the planning robot description from specified file paths if desired.

The system model provides access to some generic tools for computing kinematic and dynamic properties of the system. These tools have been designed for performing calculations for solving a wide variety of motion planning problems. The system modeling tools are generic but they can be ultimately replaced with a more specific set of kinematics and dynamics solvers in the final deployment of the algorithm. This is, however, outside of the scope of EXOTica.

4 Problem definition

EXOTica was designed for prototyping and benchmarking motion synthesis algorithms. The main objective of our framework is to provide tools for constructing problems and prototyping solvers with ease. To do so, we first separate the definition of the problem from the implementation of the solver. Each problem consists of several standardized components which we refer to as *task maps*.

4.1 Task maps

The core element of every problem defined within EXOTica is the function mapping from the configuration space (i.e. the problem state which captures the model state, a set of controlled and uncontrolled variables, and the state of the environment) to a task space. We call this function a *task map*. For example, a task map computes the center-of-mass of the robot in the world frame. A task map is a mapping from the configuration space to an arbitrary task space. The task space is, in fact, defined by the output of this function. Several commonly used task maps are implemented within EXOTica.

Joint position task map computes the difference between the current joint configuration and a reference joint configuration:

$$\Phi_{\text{Ref}}(\boldsymbol{x}) = \boldsymbol{x} - \boldsymbol{x}_{\text{ref}}, \tag{1}$$

where $\boldsymbol{x}$ is state vector of the joint configuration and $\boldsymbol{x}_{\text{ref}}$ is the reference configuration[5]. The whole state vector x may be used or a subset of joints may be selected. This feature is useful for constraining only some of the joints, e.g. constraining the back joints of a humanoid robot while performing a manipulation task. The Jacobian and Jacobian derivative are identity matrices.

Joint limits task map assigns a cost for violating joint limits. The joint limits are loaded from the robot model. The mapping is calculated as:

$$\Phi_{\text{Bound}}(x) = \begin{cases} x - x_{\text{min}} - \epsilon, & \text{if } x < x_{\text{min}} + \epsilon \\ x - x_{\text{max}} + \epsilon, & \text{if } x > x_{\text{max}} - \epsilon \\ 0, & \text{otherwise} \end{cases}, \tag{2}$$

where x_{min} and x_{max} are lower and upper joint limits respectively, and $\epsilon \geq 0$ is a safety margin. The Jacobian and Jacobian derivative are identity matrices.

End-effector frame task map captures the relative transformation between the base frame B and the tip frame A:

$$\Phi_{\text{EffFrame}}(\boldsymbol{x}) = \boldsymbol{M}_A^B, \tag{3}$$

where $\boldsymbol{M}_A^B \in SE(3)$ is computed using the system model using the *Scene*. We use the *task space vector* data structure (described later in this section) to handle storage and operations on spatial frames. The Jacobian of this task map is the geometric Jacobian computed by the *Scene*.

End-effector position captures the translation of the relative frame transformation:

$$\Phi_{\text{EffPos}}(\boldsymbol{x}) = \boldsymbol{P}_A^B, \tag{4}$$

where $\boldsymbol{P}_A^B$ is translational part of $\boldsymbol{M}_A^B$. The Jacobian of this task consists of the rows of the geometric Jacobian corresponding to the translation of the frame.

End-effector orientation captures the rotation of the relative frame transformation:

$$\Phi_{\text{EffRot}}(\boldsymbol{x}) = \boldsymbol{R}_A^B, \tag{5}$$

where $\boldsymbol{R}_A^B \in SO(3)$ is rotational part of $\boldsymbol{M}_A^B$. Similarly to the *end-effector frame* task map, the storage and the operations on the resulting $SO(3)$ space are implemented within the *task space vector*. The Jacobian of this task consists of the rows of the geometric Jacobian corresponding to the rotation of the frame.

[5] We use notation x for scalar values, $\boldsymbol{x}$ for vectors, X for matrices, and $\boldsymbol{X}$ for vectorized matrices.

End-effector distance computes the Euclidean distance between the base and tip frames:

$$\Phi_{\text{Dist}}(\boldsymbol{x}) = \|\boldsymbol{P}_A^B\|. \tag{6}$$

The resulting task map has the same function as the *end-effector position* map. The output, however, is a scalar distance.

Center-of-mass task map computes the center-of-mass of all of the robot links defined in the system model:

$$\Phi_{\text{CoM}}(\boldsymbol{x}) = \sum_i (\boldsymbol{P}_{\text{CoM}_i}^{\text{world}} m_i), \tag{7}$$

where $\boldsymbol{P}_{\text{CoM}_i}^{\text{world}}$ is the position of the center-of-mass of the i-th link w.r.t. the world frame, and m_i is mass of the i-th body. The Jacobian is computed using the chain rule. This task map can also be initialized to compute the projection of the center-of-mass on the xy-plane. In this case, the z-component is removed.

Collision spheres task map provides a differentiable collision distance metric. The collision shapes are approximated by spheres. Each sphere is attached to the kinematic structure of the robot or to the environment. Each sphere is then assigned a collision group, e.g. $i \in \mathcal{G}$. Spheres within the same group do not collide with each other, while spheres from different groups do. The collision cost is computed as:

$$\Phi_{\text{CSphere}}(\boldsymbol{x}) = \sum_{i,j}^{G} \frac{1}{1 + e^{5\epsilon(\|\boldsymbol{P}_i^{\text{world}} - \boldsymbol{P}_j^{\text{world}}\| - r_i - r_j)}}, \tag{8}$$

where i, j are indices of spheres from different collision groups, ϵ is a precision parameter, $\boldsymbol{P}_i^{\text{world}}$ and $\boldsymbol{P}_j^{\text{world}}$ are positions of the centers of the spheres, and r_i, r_j are the radii of the spheres. The sigmoid function raises from 0 to 1, with the steepest slope at the point where the two spheres collide. Far objects contribute small amount of error while colliding objects produce relatively large amounts of error. The precision parameter can be used to adjust the fall-off of the error function, e.g. a precision factor of 10^3 will result in negligible error when the spheres are further than 10^{-3}m apart. The constant multiplier of 5 in Equation (8) was chosen to achieve this fall-off profile.

In our example, we use the **end-effector position** task map. The task space is therefore $\Phi_{\text{EffPos}}(\boldsymbol{x}) \in \mathbb{R}^3$. The task map is loaded from the XML file. The following lines of the XML configuration file correspond to the task map definition:

```
<Maps>
  <EffPosition Name="Position">
    <EndEffector>
        <Frame Link="lwr_arm_7_link" BaseOffset="0.5 0 0.5
             0 0 0 1"/>
    </EndEffector>
  </EffPosition>
</Maps>
```

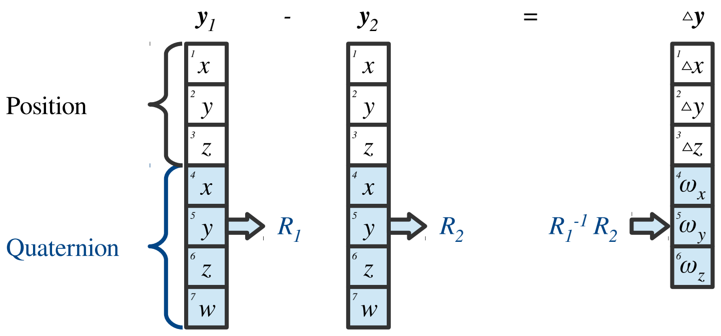


Fig. 4. Task space vector data packing combining three position coordinates $x, y, z \in \mathbb{R}$ and a sub-vector containing a $SO(3)$ rotation represented as a unit quaternion. The subtraction calculation of two task space vectors $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ first converts the quaternions into rotation matrices R_1 and R_2 and performs the rotation operation $R_2^{-1}R_1$. The result is then converted into angular velocities $\omega_x, \omega_y, \omega_z$ and packed into the output vector $\triangle y$. Notice that the dimensionality of $\triangle \boldsymbol{y} \in \mathbb{R}^6$ and $\boldsymbol{y}_1, \boldsymbol{y}_2 \in \mathbb{R}^7$ are different.

where the only parameter of the task map is a single relative spatial frame. This frame defines the translation of the seventh robot link relative to the coordinates $(0.5, 0, 0.5)$ in the world frame.[6] This example is only intended to compute inverse kinematics, we have therefore chosen to only minimize the end-effector position error. However, an arbitrary number of cost terms can be added by adding multiple task maps to this problem definition. For instance, we could easily add another task map to constrain the orientation of the end-effector.

The output of a task map is a representation of the robot's state in the task space. Most task spaces are $\mathbb{R}^n$. As such, they can be stored and handled as vectors of real numbers. However, some task maps output configurations in the $SO(3)$ or the $SE(3)$ space. In this case, Lie group algebra [13] has to be used to correctly compute the additions and subtractions in the task space. The *task space vector* implements operations on task spaces. The *task space vector* is a data structure that keeps track of $SO(3)$ sub-groups within the stored vector. The operations on this vector then implement the Lie group algebra. For example, a spatial frame may be stored as a transformation matrix $M_A^B \in \mathbb{R}^{4\times4}$. This matrix will be stored in the *task space vector*. Performing addition and subtraction on the vector will then be incorrect. The correct transformation is performed by a matrix multiplication. The *task space vector* keeps track of transformations stored within its structure and applies the correct operations on them. Furthermore, the result of subtraction is always a geometric twist, e.g. $M_A^B - M_C^B = {}^B\boldsymbol{t}_A^C$. This makes it possible to multiply the result of this operation with a geometric Jacobian, producing a geometrically correct relative

[6] If no frame is specified, world frame is assumed by default. If a relative offset is not specified, an identity transformation offset is assumed.

transformation. This feature has been used in the implementation of the inverse kinematics solver [14] and the AICO solver [15]. Additionally, a $SO(3)$ rotation can be represented and stored in different ways, e.g. as a unit quaternion $\boldsymbol{R}_{\mathcal{Q}} \in \mathbb{R}^4$ where $\|\boldsymbol{R}_{\mathcal{Q}}\| = 1$, Euler angles $\boldsymbol{R}_{\mathcal{ZYZ}}, \boldsymbol{R}_{\mathcal{ZYX}}, \boldsymbol{R}_{\mathcal{RPY}} \in \mathbb{R}^3$, angle-axis representation $\boldsymbol{R}_{\mathcal{A}} \in \mathbb{R}^3$ where $\|R_{\mathcal{A}}\| = \theta$, rotation matrix $\boldsymbol{R} \in \mathbb{R}^{3\times3}$, etc. We handle these representations implicitly. Each sub-group of the *task space vector* stores the size and type of representation that was used. The operations on the vector first convert the task space coordinates into a rotation matrix representation, then the correct spatial operation is applied and a twist is computed. As a result the input and output dimension may vary, i.e. subtraction of two rotations represented as rotation matrices is a function $f(R_1, R_2) : \mathbb{R}^9 \to \mathbb{R}^3$. The result is the angular velocity component of the twist. The *task space vector* is composed by concatenating outputs of multiple task maps. Each task map specifies if its output contains any components that have to be handled using the Lie group algebra (see Figure 4).

The output of a single task map is a segment of the *task space vector*. The input of a task map is the states of the robot model and environment as well as the arbitrary number of frame transformations required for the calculations. These are computed using the *planning scene*. The task map implements the mapping within its `update` method. This method has 3 different overloads depending on what order of derivative is requested: a) no derivative (e.g. in sampling), b) first-order derivatives (e.g. Jacobian used in gradient descent), and c) second-order derivatives. Not all overloads have to be defined, i.e. a collision checking task map may only detect collisions but it will not provide any gradients (derivatives). We exploit this for fast collision checking for sampling-based solvers [5].

The task map will update the task space vector and its derivatives when the solver requires it. These updates are normally triggered by the solver and they do not have to be called manually. This also ensures that the *task space vector* is updated correctly. The collection of task maps is therefore central to formally defining motion planning problems. How the output of the task map is used then depends on the type of the planning problem.

4.2 Planning problems

A *planning problem* within EXOTica represents a specific formulation of a motion planning problem. Since every formulation has very specific advantages for a particular type of application, the formulations may vary significantly. To provide a unified framework, we identify several categories of common features of different types of problems.

Depending on how the system is modeled, we distinguish: a) kinematic, b) kino-dynamic, and c) dynamic systems. We then categorize the problem based on the *state representation* required by these types of systems: position ($\boldsymbol{x}$), position and velocity ($\boldsymbol{x}, \dot{\boldsymbol{x}}$), and the full dynamic state ($\boldsymbol{x}, \dot{\boldsymbol{x}}, \ddot{\boldsymbol{x}}, \boldsymbol{\tau}, \boldsymbol{F}$) where the variables denote positions, velocities, accelerations, joint torques, and external forces respectively. We then distinguish between *planning spaces*: a) configuration space, and b) task space (e.g. end-effector position and orientation). These categories define how the state of the the system is stored. This affects both memory layout and the format of the input to the solver (e.g. the start state has to include joint positions and velocities).

Furthermore, we categorize the problem based on the type of the output. The *output type* may include: a) single configuration (e.g. output of a inverse kinematics solver), b) a time-indexed trajectory (e.g. output of trajectory optimization), or c) non-time indexed trajectory (e.g. output of a sampling-based solver). Other types and subtypes of problems do exist. A time-indexed trajectory may have fixed or variable number of time steps or a variable timing between steps. The second category is related to the output type with respect to the controls. Here we consider control paradigms such as position control, velocity control, and torque control. The output of a problem may therefore consist of various combinations of position, velocity, acceleration, and torque trajectories. We refer to this as the *control type*.

Finally, we consider types of common problem formulations, for instance:

An **unconstrained quadratic cost minimization problem** w.r.t. metric Q as presented in [14]:

$$\underset{\boldsymbol{x}}{\operatorname{argmin}}(f(\boldsymbol{x})^\top Q f(\boldsymbol{x})). \tag{9}$$

A **linear programming problem**:

$$\begin{aligned} &\underset{\boldsymbol{x}}{\operatorname{argmin}}(Q\boldsymbol{x} + \boldsymbol{c}) && (10)\\ &\text{s.t. } A\boldsymbol{x} \le \boldsymbol{b}, && (11)\\ &\qquad B\boldsymbol{x} = \boldsymbol{b}. && (12) \end{aligned}$$

A **quadratic programming problem with linear constraints**, e.g. used by the authors of [16]:

$$\begin{aligned} &\underset{\boldsymbol{x}}{\operatorname{argmin}}(\boldsymbol{x}^\top Q\boldsymbol{x} + \boldsymbol{c}^\top \boldsymbol{x})\\ &\text{s.t. } A\boldsymbol{x} \le \boldsymbol{b}, && (13)\\ &\qquad B\boldsymbol{x} = \boldsymbol{b}. && (14) \end{aligned}$$

A **generic non-linear programming problem**, e.g. as used in [17]:

$$\begin{aligned} &\underset{\boldsymbol{x}}{\operatorname{argmin}}\|f(\boldsymbol{x})\|^2\\ &\text{s.t. } g(\boldsymbol{x}) \le 0, && (15)\\ &\qquad h(\boldsymbol{x}) = 0. && (16) \end{aligned}$$

A **mixed-integer non-linear programming problem with constraints**, such as MIQCQP presented in [18]:

$$\begin{aligned} &\underset{\boldsymbol{x}}{\operatorname{argmin}}\|f(\boldsymbol{x}, \boldsymbol{i})\|^2\\ &\text{s.t. } g(\boldsymbol{x}, \boldsymbol{i}) \le 0, && (17)\\ &\qquad h(\boldsymbol{x}, \boldsymbol{i}) = 0, && (18)\\ &\qquad \boldsymbol{i} \in \mathbb{N}. && (19) \end{aligned}$$

Planning space	Problem type	State representation	Output type type
[CSpace]	[Unconstrained]	[Kinematic]	[EndPose]
[CSpace]	[Unconstrained]	[Kinematic]	[TimeIndexed]
[CSpace]	[Sampling]	[Kinematic]	[NonIndexed]
[CSpace]	[NLP]	[Dynamic]	[TimeIndexed]

Table 1. Naming convention of motion planning problems based on problem type categories. Gray name components are omitted for brevity.

A **sampling problem**:[7] which is common across several algorithms presented in [5].

$$\arg_{\boldsymbol{x}} f(\boldsymbol{x}) = \text{True}. \tag{20}$$

A **mixed sampling and optimization problem**, e.g. as presented in [19]:

$$\begin{aligned} &\underset{\boldsymbol{x}}{\text{argmin}} \|f(\boldsymbol{x})\|^2 \\ &\quad \text{s.t. } g(\boldsymbol{x}) = \text{True}. \end{aligned} \tag{21}$$

These are just some of the commonly used formulations of motion planning problems used across literature. EXOTica provides the tools and a generic structure for implementing these types of problem formulations.

EXOTica uses a problem naming system based on this categorization. The names are constructed based on the four main categories: *planning space*, *problem type*, *state representation* and the *output type*. Table 1 shows how the name is constructed. To achieve brevity, each category has a default type. The default types are *configuration space*, *sampling*, *kinematic*, *non-time indexed trajectory* respectively for each category in Table 1. When the problem falls within the default type for a category, this type is omitted from the name. For example, a problem of type `SamplingProblem` is referring to a configuration space sampling problem using a kinematic robot model and returning a non-time indexed trajectory. Similarly, `NLPDynamicTimeIndexedTorqueControlledProblem` is an example of a fully specified problem corresponding to the one defined in [20]. Three sample problem types which are implemented within EXOTica are highlighted here.

Unconstrained End-Pose Problem defines a problem minimizing the quadratic cost using a kinematic system. The state is represented by joint configurations and the output is a single configuration that minimizes the cost defined by

[7] We refer to a problem with binary variables as a sampling problem because randomized or another type of sampling is required to solve them. These types of problems often cannot be solved by numerical optimization because their constraints are not differentiable.

Equation (9). The cost function is composed of weighted components:

$$f(\boldsymbol{x}) = \sum_i \rho_i \|\Phi_i(\boldsymbol{x}) - \boldsymbol{y}_i^*\|, \tag{22}$$

where $\Phi_i(\boldsymbol{x})$ is the mapping function of the i-th task map as defined in Section 4.1, $\boldsymbol{y}_i^*$ is the reference or goal in the task space, and ρ_i is the relative weighting of the task. By definition, This problem provides the first derivative of the quadratic cost function. The derivatives of the task terms are provided by the *task map*. Additionally, configuration space weighting W is specified. This allows us to scale the cost of moving each joint (or control variable) individually. Optionally, a nominal pose $\boldsymbol{x}_{\text{nominal}}$ is provided as a reference often used to minimize secondary cost in case of redundancies. This type of problem can be used for solving inverse kinematics problems as proposed in [14].

The example inverse kinematics problem is using this formulation. The full problem definition contains the definition of the *planning scene* and the *task map* as discussed previously. We also set the configuration space weighting W. The problem is then fully defined using the following XML string:

```
<UnconstrainedEndPoseProblem Name="MyProblem">
  <PlanningScene>
    <Scene>
      <JointGroup>arm</JointGroup>
      <URDF>{exotica_examples}/resources/robots/
          lwr_simplified.urdf</URDF>
      <SRDF>{exotica_examples}/resources/robots/
          lwr_simplified.srdf</SRDF>
    </Scene>
  </PlanningScene>
  <Maps>
    <EffPosition Name="Position">
      <EndEffector>
          <Frame Link="lwr_arm_7_link" BaseOffset="0.5 0
              0.5 0 0 0 1"/>
      </EndEffector>
    </EffPosition>
  </Maps>
  <W> 7 6 5 4 3 2 1 </W>
  <StartState>0 0 0 0 0 0 0</StartState>
  <NominalState>0 0 0 0 0 0 0</NominalState>
</UnconstrainedEndPoseProblem>
```

The weighting W is set to reduce movement of the joints closer to the root (root joint weight 7 to tip joint weight 1). We set the start and the nominal configuration, or state, to a zero vector. This problem is now complete and we can use it to compute the robot configuration which moves the top of the robot to coordinates $(0.5, 0, 0.5)$. EXOTica provides several useful tools that make defining problems using XML more versatile. All strings are automatically parsed as the required data types. Furthermore, file paths containing curly bracket macros will be replaced with catkin package paths, e.g. `{exotica_examples}` will get replaced with the absolute path to the EXOTica package.[8]

Unconstrained Time-Indexed Problem defines a problem minimizing the quadratic cost over a trajectory using a kinematic model of the system. The trajectory is uniformly discretized in time. The time step duration $\triangle t$ and number of time steps T are specified. The system transition from one time step to another is parametrized by the covariance of the state transition error W (analogous to the weighting W used in the Unconstrained End-Pose Problem) and the covariance to the control error H. These parameters define the evolution of a stochastic system approximated by a linear model over time. See [21] for more details about this model. The cost function is then defined as:

$$f(\boldsymbol{x}) = \sum_t \sum_i \rho_{i,t} \|\varPhi_{i,t}(\boldsymbol{x}) - \boldsymbol{y}^*_{i,t}\|, \tag{23}$$

where $t \in (1, ..., T)$ is the time index. This type of problem is suitable for use with iLQG-like (iterative linear-quadratic Gaussian [22]) solvers and with the approximate inference control (AICO) algorithm [15].

Sampling Problem defines a class of kinematic problems that do not require any cost function. Each state is evaluated for validity but not for quality as described in Equation (20). This type of problem also requires a goal configuration $\boldsymbol{x}^*$. The objective of the solvers is to compute a valid trajectory from the start state to the goal state. The validity of each state is checked by applying a threshold ϵ on the output of the task map: $\rho_i(\varPhi_i(\boldsymbol{x}) - \boldsymbol{y}^*_i) < \epsilon$. The output trajectory is not indexed on time and the number of configurations may vary between solutions. This type of planning problem is used with sampling-based motion solvers, such as RRT and PRM [5].

The updating of the task maps is always handled by the planning problem. This ensures that all the storage of the task-related properties is consistent and timely. Each problem class is also the storage container for all of the task related data and parameters. For example, the `UnconstrainedEndPoseProblem` stores the task weights ρ, the task map outputs $\varPhi(\boldsymbol{x})$ and goals $\boldsymbol{y}^*$ in form of a *task space vector*, and the Jacobians of each task map $J = \frac{\partial \varPhi(\boldsymbol{x})}{\partial \boldsymbol{x}}$. Since each problem has a different formulation and structure, how the data is stored may vary. However, each problem has to fully contain all the data and parameters required by the solvers. This ensures modularity and makes it possible to benchmark different solvers on the same set of problems.

5 Motion solvers

The structure of a planning problem within EXOTica allows us to formally define an interface for solving specific types of problems. The motion solver then takes

[8] This feature combines regular expressions with the rospack library to parse catkin package paths.

the problem instance as input and computes the solution. How this computation is performed depends entirely on the implementation of the solver. EXOTica offers several built-in solvers.

In our example, we created a system model and an unconstrained end-pose problem that uses this system model. We will now use our implementation of the inverse kinematics solver to compute the solution.

The *inverse kinematics solver* implements the regularized, dampened Jacobian pseudo-inverse iterative algorithm described in [14]. This algorithm minimizes the cost function defined in Equation (22) by iteratively improving the current solution until convergence as described in Algorithm 1.

Algorithm 1 IK solver

```
Require: $C, \alpha$
 1: $(\boldsymbol{x}_0, \boldsymbol{x}_{\text{nominal}}, W, max\_iter, \epsilon) \leftarrow$ GetProblem
 2: $\boldsymbol{x} \leftarrow \boldsymbol{x}_0$
 3: $iter \leftarrow 0$
 4: repeat
 5:     $(\Phi, J) \leftarrow$ UpdateProblem($\boldsymbol{x}$)
 6:     $J^\dagger \leftarrow W^{-1}J^\top(JW^{-1}J^\top + C)^{-1}$
 7:     $\triangle\boldsymbol{x} \leftarrow \alpha\left[J^\dagger(f(\boldsymbol{x}) - \boldsymbol{y}^*) + (I - J^\dagger J)(\boldsymbol{x}_{\text{nominal}} - \boldsymbol{x})\right]$
 8:     $\boldsymbol{x} \leftarrow \boldsymbol{x} + \triangle\boldsymbol{x}$
 9:     $iter \leftarrow iter + 1$
10: until $\|\triangle\boldsymbol{x}\| < \epsilon$ and $iter < max\_iter$
11: return $\boldsymbol{x}$
```

In Algorithm 1, $\boldsymbol{x}$ is the state of the robot, α is the convergence rate, $J = SJ(\boldsymbol{x})$ is the concatenated Jacobian matrix multiplied by a diagonal matrix $S = \text{diag}((\rho_1, \rho_2, ...))$ constructed from the task map weights ρ, Φ is the concatenated task space vector, I is the identity matrix, $\boldsymbol{x}_{\text{nominal}}$ is the nominal configuration that is achieved as a secondary goal in case of redundancies, W is the configuration space cost metric or weighting, and C is the regularization. The solver iterates until convergence ($\|\triangle\boldsymbol{x}\| < \epsilon$) or until the maximum number of iterations is reached. This algorithm implements a local cost minimization method that requires access to the cost function $f(\boldsymbol{x})$ and its first derivative $J(\boldsymbol{x}) = \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$. The cost function and its derivative are provided by the unconstrained end-pose problem in line 5. Parameters C and α are properties of the motion solver. Parameters $\boldsymbol{x}_0, \boldsymbol{x}_{\text{nominal}}, W, max\_iter$ and ϵ are properties of the planning problem and they are extracted using the GetProblem method. The output of this solver is a single robot configuration solving the inverse kinematics problem.

EXOTica was designed for development of new motion planning algorithms. We provide two more solvers that can be used for benchmarks. The AICO solver is the implementation of the **A**pproximate **I**nference **CO**ntrol algorithm presented in [6]. This algorithm performs trajectory optimization on problems with a quadratic approximation of the cost, linear approximation of the system model, uniformly discretized time axis, and no hard constraints. The complete algorithm, including the pseudo code, is described in [6]. The implementation of AICO within EXOTica uses the unconstrained time-indexed problem to calculate and store all task related properties. The OMPL solver is a wrapper exposing several sampling-based algorithms implemented within the Open Motion Planning Library [8].

6 Python wrapper

The utility of the EXOTica framework is in fast prototyping of motion synthesis algorithms. The modular structure of EXOTica provides tools for creating new problem and solver types which can now be evaluated and compared with competing algorithms. Setting up tests and benchmarks is straightforward as all parameters of the algorithm are clearly exposed. To make this process even more versatile, we provide a Python wrapper for EXOTica. The following code shows how to initialize and solve our example inverse kinematics problem using Python:

```
import pyexotica as exo
solver = exo.Setup.loadSolver('{exotica_examples}/
    resources/configs/example.xml')
print(solver.solve())
```

The EXOTica Python wrapper is intended for instantiating problems and solvers to provide a high level interface to the EXOTica tools. This interface is suitable for creating planning services, benchmarks, and unit tests. All core classes and methods are exposed.

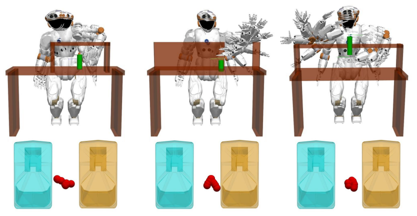
7 Applications

The rest of this chapter will provide examples of how the different elements of EXOTica are leveraged for prototyping and evaluating new algorithms.

7.1 Algorithm Benchmarking

The modularity of EXOTica enables us to create benchmarks comparing a variety of problems and solvers. In [23], we construct such benchmark to evaluate several sampling-based algorithms implemented inside the Open Motion Planning Library [8] (OMPL) on a set of reaching problems on a humanoid robot. These algorithms were primarily designed for solving navigation problems and motion planning for fixed base robot arms. However, we have applied these methods to planning whole-body trajectories for manipulation in cluttered environments using humanoid robots.

Valid trajectories for humanoid robots can only contain states that are collision-free while they also have to satisfy additional constraints such as center-of-mass


(a) Reaching motion on the NASA Valkyrie robot.

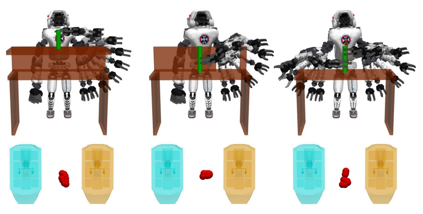

(b) Reaching motion on the Boston Dynamics Atlas robot.

Fig. 5. Collision-free whole-body motion generated in different environments with different robot models. The corresponding CoM trajectories are illustrated in the second row (red dots). The benchmark is designed so that one can easily switch to new robot platforms without extensive integration effort.

position, foot placement, and sometimes torso orientation. Generating collision-free samples is straightforward by using random sample generators and standard collision checking libraries. However, the additional constraints create a manifold in the unconstrained configuration space. Generating samples which lie on this manifold without having to discard a majority of them in the process is non-trivial. A sampling bias has to be introduced to increase the probability of generating correctly constrained samples. In our approach, a whole-body inverse kinematic solver is employed to produce the constrained samples. We formulate the inverse kinematics problems as a non-linear program (NLP):

$$\begin{aligned} &\underset{\boldsymbol{x}}{\operatorname{argmin}} \|\boldsymbol{x} - \boldsymbol{x}_{\text{nominal}}\|_Q^2, \\ &\text{s.t. } \boldsymbol{b}_l \leq \boldsymbol{x} \leq \boldsymbol{b}_u, \\ &c_i(\boldsymbol{x}) \leq 0, i \in \mathcal{C} \end{aligned} \tag{24}$$

where $\|\boldsymbol{x} - \boldsymbol{x}_{\text{nominal}}\|_Q^2$ is the squared deviation from the nominal pose $\boldsymbol{x}_{\text{nominal}}$ with respect to a configuration space metric Q. The system is subject to lower and upper bound constraints $\boldsymbol{b}_l$ and $\boldsymbol{b}_u$, and a set of non-linear constraints $\mathcal{C}$. The solver is described in [16]. We will call this solver using the following routine $IK(\boldsymbol{x}_0, \boldsymbol{x}_{\text{nominal}}, \mathcal{C})$, where $\boldsymbol{x}_0$ is start state, $\boldsymbol{x}_{\text{nominal}}$ is the nominal state and $\mathcal{C}$ is the set of constraints.

The majority of algorithms implemented in OMPL perform three basic steps: (1) sample a random state, (2) perform steering, (to compute a *near state* that is close to the random state according to some metric), (3) append the *near state* to the solution if it satisfies all constraints. To preserve compatibility with these algorithms, we augment steps 1 and 2. In step 1, we sample a random unconstrained configuration and return the constrained sample computed using the inverse kinematics (see routine `sampleUniform` in Algorithm 2). In the second step, we then compute the constrained near state using the `sampleUniformNear` routine. The steering function then uses an interpolation routine to check if a path from the current state to the near state is viable. We have augmented the

Algorithm 2 Humanoid Configuration Space Sampling-based Planning

```
sampleUniform()
 1: $succeed$ = False
 2: while not $succeed$ do
 3:     $\bar{\mathbf{x}}_{rand} = RandomConfiguration()$
 4:     $\mathbf{x}_{rand}, succeed = IK(\bar{\mathbf{x}}_{rand}, \bar{\mathbf{x}}_{rand}, \mathcal{C})$
    return $\mathbf{x}_{rand}$
```

```
sampleUniformNear($\mathbf{x}_{near}, d$)
 1: $succeed$ = False
 2: while not $succeed$ do
 3:     $\boldsymbol{a} = 0$
 4:     while not $succeed$ do
 5:         $\bar{\mathbf{x}}_{rand} = RandomNear(\mathbf{x}_{near}, d)$
 6:         $\mathcal{C}_{extended} = \mathcal{C}$
 7:         $\mathcal{C}_{extended} \leftarrow \|\mathbf{x}_{rand} - \bar{\mathbf{x}}_{rand}\|_Q < \boldsymbol{a}$
 8:         $(\mathbf{x}_{rand}, succeed) = IK(\bar{\mathbf{x}}_{rand}, \mathbf{x}_{near}, \mathcal{C}_{extended})$
 9:         Increase $\boldsymbol{a}$
10:     if $distance(\mathbf{x}_{rand}, \mathbf{x}_{near}) > d$ then
11:         $succeed$ = False
    return $\mathbf{x}_{rand}$
```

```
interpolate($\mathbf{x}_a, \mathbf{x}_b, d$)
 1: $\bar{\mathbf{x}}_{int} = InterpolateConfigurationSpace(\mathbf{x}_a, \mathbf{x}_b, d)$
 2: $succeed$ = False
 3: $\boldsymbol{a} = 0$
 4: while not $succeed$ do
 5:     $\mathcal{C}_{extended} = \mathcal{C}$
 6:     $\mathcal{C}_{extended} \leftarrow \|\mathbf{x}_{int} - \bar{\mathbf{x}}_{int}\|_Q < \boldsymbol{a}$
 7:     $(\mathbf{x}_{int}, succeed) = IK(\bar{\mathbf{x}}_{int}, \mathbf{x}_a, \mathcal{C}_{extended})$
 8:     Increase $\boldsymbol{a}$
    return $\mathbf{x}_{int}$
```

interpolation routine as shown in Algorithm 2. In EXOTica, a new problem type is defined called the `ConstrainedSamplingProblem`. The constraints defined within this problem are passed to the *IK* solver when a constrained solution is required. The motion solver then instantiates the OMPL algorithm overriding the default sampling and and interpolation methods with the ones defined in Algorithm 2.

The second category of problems perform the exploration in the task space. Specifically, the space of positions and orientations of the gripper is used as the state space. We call this space the *end-effector space*. The pose of the gripper does not uniquely describe the configuration of the whole robot. Each gripper position is therefore associated with a full robot configuration to avoid redundancies in the representation. This type of planning problem is using a low dimensional state representation but the connectivity of states depends on the configuration of the whole robot. Therefore, each valid state lies on a manifold in the end-effector

Algorithm 3 Humanoid End-Effector Space Sampling-based Planning

```
sampleUniform()
1: $succeed$ = False
2: while not $succeed$ do
3:     $\bar{\mathbf{y}}_{rand} = RandomSE3()$
4:     $\mathcal{C}_{extended} = \mathcal{C}$
5:     $\mathcal{C}_{extended} \leftarrow \|\bar{\mathbf{y}}_{rand} - \Phi(\mathbf{x}_{rand})\| \leq 0$
6:     $\mathbf{x}_{rand}, succeed = IK(\bar{\mathbf{x}}_{rand}, \bar{\mathbf{x}}_{rand}, \mathcal{C}_{extended})$
7: $\mathbf{y}_{rand} = \Phi(\mathbf{x}_{rand})$
   return $\mathbf{y}_{rand}, \mathbf{x}_{rand}$
```

```
sampleUniformNear($\mathbf{y}_{near}, d$)
1: $succeed$ = False
2: while not $succeed$ do
3:     $\bar{\mathbf{y}}_{rand} = RandomNearSE3(\mathbf{y}_{near}, d)$
4:     $\mathcal{C}_{extended} = \mathcal{C}$
5:     $\mathcal{C}_{extended} \leftarrow \|\bar{\mathbf{y}}_{rand} - \Phi(\mathbf{x}_{rand})\| \leq 0$
6:     $\mathbf{x}_{rand}, succeed = IK(\mathbf{x}_{rand}, \mathbf{x}_{near}, \mathcal{C}_{extended})$
7: $\mathbf{y}_{rand} = \bar{\mathbf{y}}_{rand}$
   return $\mathbf{y}_{rand}, \mathbf{x}_{rand}$
```

```
interpolate($\mathbf{y}_a, \mathbf{y}_b, d$)
1: $\bar{\mathbf{y}}_{int} = InterpolateSE3(\mathbf{y}_a, \mathbf{y}_b, d)$
2: $succeed$ = False
3: $\boldsymbol{b} = 0$
4: while not $succeed$ do
5:     $\mathcal{C}_{extended} = \mathcal{C}$
6:     $\mathcal{C}_{extended} \leftarrow \|\bar{\mathbf{y}}_{int} - \Phi(\mathbf{x}_{int})\| < \boldsymbol{b}$
7:     $\mathbf{x}_{int}, succeed = IK(\mathbf{x}_a, \mathbf{x}_a, \mathcal{C}_{extended})$
8:     Increase $\boldsymbol{b}$
9: $\mathbf{y}_{int} = \Phi(\mathbf{x}_{int})$
   return $\mathbf{y}_{int}, \mathbf{x}_{int}$
```

space. This is a very similar problem to the constrained sampling problem in the configuration space. To implement this, we augmented the solver by replacing the sampling and steering steps of the algorithm with the inverse kinematics solver as described in Equation (24). Additionally, we use the forward kinematics $\Phi(\boldsymbol{x})$ to compute the gripper pose corresponding to a whole robot configuration $\boldsymbol{x}$. Algorithm 3 show the modifications to Algorithm 2 required to perform planning in the end-effector space. We have implemented this type of problem in EXOTica and we call it the `ConstrainedEndEffectorSamplingProblem`.

With the problem formulation in place, we created several environments containing obstacles such as desks and shelves. We have also defined grasping targets placed on top of the horizontal surfaces. Each environment was then tested with two humanoid robot models: NASA Valkyrie and Boston Dynamics Atlas. Figure 5 shows several scenarios used in the benchmark. We have performed 100 trials in each environment with each robot. We have collected data on computation time, task accuracy, and constraint violation. The results ultimately show that existing (off the shelf) sampling based motion solvers can be used with humanoid robots. The RRT-Connect algorithm outperforms all other methods when planning in configuration space, with planning times averaging at 3.45s in the most challenging test scenario. We present the complete benchmark and the analysis in [23].

The benchmark described in this section was aimed at a specific aspect of sampling-based motion solvers. EXOTica enabled us to perform this analysis while maximizing the reuse of existing code. The problem and solver formulations described in this chapter are suitable for creating benchmarks evaluating various aspects of motion planning. In the next section, we explore and compare different problem formulations and how EXOTica can be used to validate that a particular formulation is suitable for solving a very specific task.

7.2 Dynamic Motion Adaptation and Replanning in Alternate Spaces

Adapting motion to dynamic changes is an active area of research. Dynamic environments cause existing plans to become invalid, and it is often necessary to adapt an exiting motion or to re-plan the motion entirely. This process can be computationally expensive when the problem is non-convex and contains local minima. However, a highly non-linear and non-convex problem can sometimes be simplified if we choose an *alternate task space.* A task space introduces a new metric or even a new topology. As a result, motion that is very complex in the configuration may be as simple as a linear interpolation in the appropriate task space. Figure 6 illustrates the effect of using an alternate mapping. EXOTica allows us to define such new task spaces.

In [24], we implemented a novel task map we call the *Interaction Mesh.* To construct the interaction mesh, we define vertices, points attached to the kinematic structure of the robot or the scene. We then define edges between the vertices to create a mesh. Each edge represents a spatial relationship that has to be preserved. In [24], the mesh is fully connecting all of the vertices. We then compute the Laplace coordinate L_G of each vertex $\boldsymbol{p}$ as

$$\begin{aligned} L_G(\boldsymbol{p}) &= \boldsymbol{p} - \sum_{r\in\partial_G \boldsymbol{p}} \frac{\boldsymbol{r} w_{pr}}{\sum_{s\in\partial_G \boldsymbol{p}} w_{ps}}, \\ w_{pr} &= \frac{W_{pr}}{|\boldsymbol{r}-\boldsymbol{p}|}, w_{ps} = \frac{W_{ps}}{|\boldsymbol{s}-\boldsymbol{p}|}, \end{aligned} \tag{25}$$

where $\partial_G p$ is the neighbourhood of $\boldsymbol{p}$ in the mesh G, and w_{pr} is the weight inversely proportional to the distance of vertices $\boldsymbol{p}, \boldsymbol{r}$ and multiplied by the

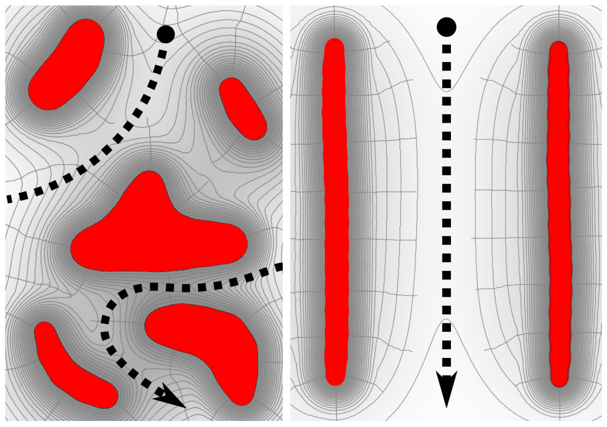


Fig. 6. A complex trajectory in the configuration space (left) becomes a linear trajectory in an alternate space (right).

edge importance weighting[9] W_{pr}. The weights are then normalized over the neighbouring nodes $s \in \partial_G p$. The position of the vertices $\boldsymbol{p}, \boldsymbol{r}$ and $\boldsymbol{s}$ is computed using kinematics (e.g. $\boldsymbol{p} = \Phi(\boldsymbol{x})$). The Jacobian of the interaction mesh can be derived using the chain rule.

In EXOTica, we define a task map which implements Equation (25). This task maps the vertex positions into the space of their Laplace coordinates. We call this the interaction mesh space. In the experiments presented in [24], we recorded a reference trajectory in the interaction mesh space. We have then formulated an unconstrained end-pose problem as defined in Section 4.2 to minimize the error in this space. We employed the inverse kinematics solver described in Section 5 in a control loop to track the dynamically moving targets in real-time. We have closed the control loop with motion tracking data streamed in real-time. Figure 7 shows the timelapse of this experiment.

Applying local optimisation methods in alternate spaces is a powerful tool. We have achieved fast and robust motion transfer from a human demonstrator to a robot using the interaction mesh and a motion capture suit (see Figure 8a). This principle has been further studied in [25], where we introduce a more flexible variant of the interaction mesh. We call this alternate representation the *distance mesh*. Distance mesh captures the spatial relationships over edges, rather than the vertices of the mesh. This allows us to encode obstacle avoidance and target reaching behavior more directly. We have demonstrated this technique by performing a welding task, maintaining the contact of the welding tool with the work piece, while reactively avoiding moving obstacles (see Figure 8b). This

[9] The edge importance weight matrix allows us to further parametrize the representation. For example, high weighting between the target and the end-effector allows us to perform accurate reaching and grasping.

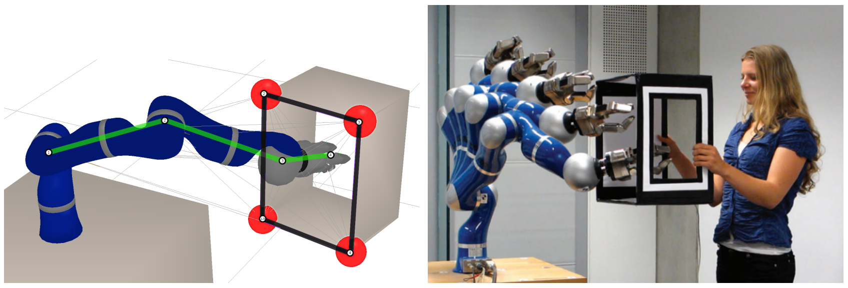


Fig. 7. The vertices and edges of the interaction mesh attached to the kinematic structure of the robot and parts of the dynamically moving obstacle (left), and the timelapse of the optimised motion (right).

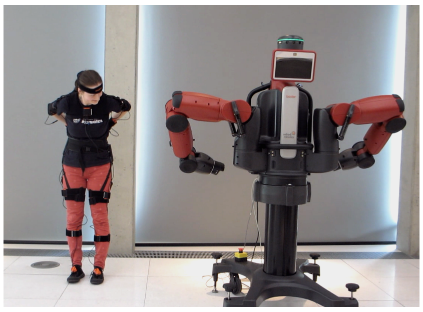
experiment also uses the unconstrained end-pose problem formulation and the inverse kinematics solver described in this chapter.

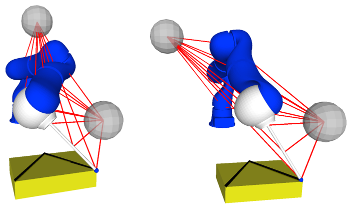
In [24], we also explore a class of representations based on topology of the work space. These alternate spaces abstract away the geometry of the space and capture topological metrics such as winding (see Figure 8d), wrapping, and enclosing volumes of space. We have applied these metrics to reaching and grasping problems. Each of these representations introduce a space with a topology very different to the topology of the configuration space. However, each of these spaces is still implemented as a task map, and as such, it can be used with any solver within EXOTica. This work was then extended in [26] to optimize area coverage when moving robot links around an object of interest, such as 3D scanning and painting (see Figure 8c).

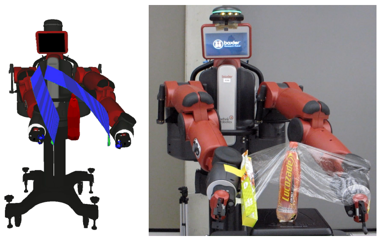
The concept of encapsulation of the task map makes it straight forward to define new task spaces. Constructing experiments to evaluate the utility of each space therefore requires only very minimal implementation. This makes EXOTica ideal for rapid prototyping of new task spaces and defining very specialized problems. Once a suitable planning problem is identified, EXOTica can be used to implement a motion planning service that can be integrated with sensing and user interface modules to solve more complex tasks. We will describe this in the next section.

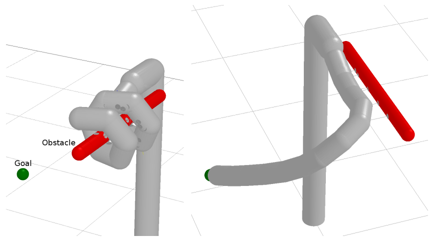
7.3 Robust Shared Autonomy with Continuous Scene Monitoring

The level of autonomy of a robot is directly correlated with the predictability of the task and environment. Robots executing a predictable task in a predictable environment such as in a factory setting can work fully autonomously, while for field robots where the environment changes unpredictably, teleoperation is still the accepted gold standard with the human operators as the decision makers and the robot acting as their extension. We focus on a hybrid approach called *shared autonomy*. It is often perceived as a middle ground, combining autonomous sequences requested by the operator and executed by the robot. The operator


(a) Motion transfer using interaction mesh.


(b) Reactive obstacle avoidance using distance mesh.


(c) Wrapping an item using spatio-temporal flux representation.



(d) Untangling a multi-link robot using the writhe representation.

Fig. 8. Examples of implementations of different task spaces using the EXOTica framework.

provides high-level decision making reducing cognitive load and improving the reliability of the systems.

The work in [27] illustrates a complete framework and system application for robust shared autonomy which builds extensively on EXOTica. This systems is composed of four core components:
(1) A *mapping* module acquiring, filtering, and fusing several sources of real sensor data and using it to continuously update the EXOTica planning and collision scene.
(2) An implementation of the *Inverse Dynamic Reachability Map* (iDRM) presented in [28] in EXOTica and integration of the iDRM-based inverse kinematics solver with sampling based motion planning.
(3) A *user interface* for synthesizing EXOTica planning problems, where the user provides constraint sets for both inverse kinematics and motion planning.
(4) A *scene monitoring* service for detecting and assessing dynamic changes in the environment in real-time and updating the map and the EXOTica planning scene.
Figure 9 shows an overview of the system.

The details of the mapping and user interface components provide updates for the EXOTica planning scene. The sensor data is being processed and filtered

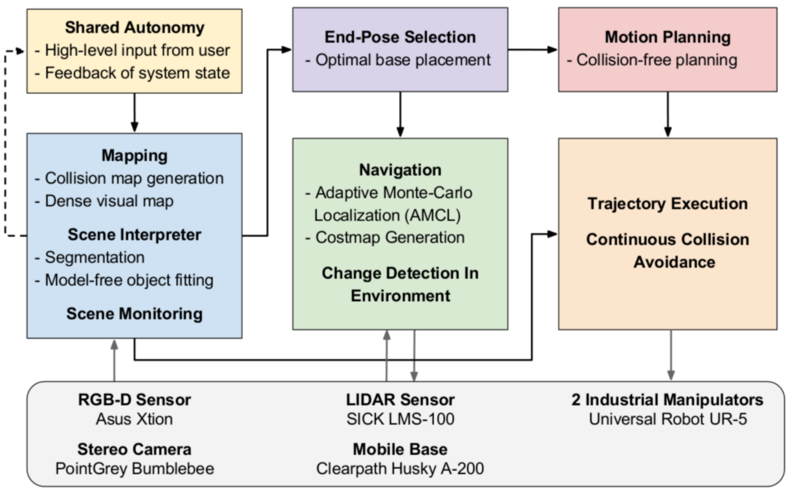


Fig. 9. Overview of the mapping, motion planning, scene monitoring and user interface components with existing ROS-based navigation, control, sensor driver packages.

to produce an OctoMap that is used for collision avoidance. The details of this process are described in [27]. Other collision objects such as CAD models of known structures can be inserted into the EXOTica scene directly. The most up-to-date snapshot of the scene is then used for motion planning.

The user interface also uses the scene monitoring and mapping components to provide the user with a virtual view of the robot environment. The user then specifies high-level goals, such as an object to grasp. The user interface component will process this input and create a set of goals that may contain a grasping target, a navigation target, and motion plan to pick-up or place an item. We compute these goals using EXOTica.

Given a grasping target location from the user interface, we construct an end-pose planning problem. Since the target location is known but the base of the robot is not fixed, we invert the problem. We compute the location of the base of the robot using the inverse kinematics solver. Here we use the reaching target as a virtual fixed base and we compute an optimal location of the robot base. The problem is formulated as `iDRMEndPoseProblem`. We exploit the iDRM structure to compute collision-free inverse kinematics for the robot base as proposed in [28,29]. We have implemented a dedicated solver for this type of problems. The output of the solver is a collision-free robot configuration that we use as a goal for sampling-based motion planning. Additionally, we use the computed robot base location as a target for navigation. The motion planning is then performed using the OMPL interface we have described in Sections 5 and 7.1.

The motion planning component is implemented as a ROS action server. This creates a clean interface with the rest of the system. We have performed

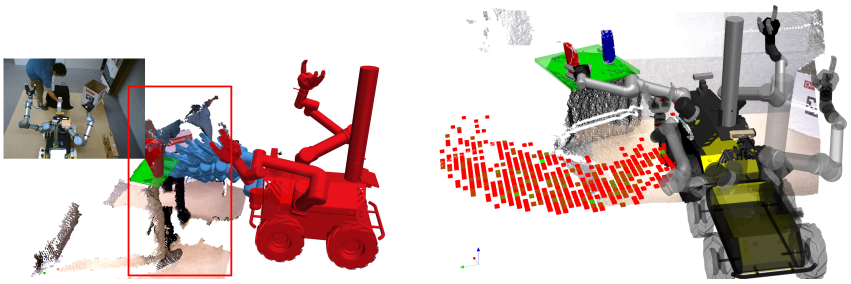


Fig. 10. (Left) The continuous scene monitoring continuously integrates fused and filtered sensor data in an OctoMap and reasons about changes: Here, a human reaches into the robot working space crossing the planned trajectory (key samples in blue), and the robot halts execution (robot state in red). (Right) Candidate robot base locations computed using iDRM (red and green cubes colored by reachability) and a the optimal robot pose for grasping the red box selected by the user.

several experiments with the integrated system. Figure 10 shows snapshots of the system performing a pick-and-place task on a bi-manual robot with a mobile base.

8 EXOTica installation instructions

The up to date installation instructions are available at:
`https://ipab-slmc.github.io/exotica/Installation.html`
Example code is available within the `exotica_examples` package. We provide several other examples solving different types of problems as well as applications and tests within this package.

9 Conclusion

The utility of ROS lies in the modularity of its packages. With this in mind, EXOTica was developed as a tool to prototype, evaluate, and rapidly deploy novel motion synthesis algorithms. Existing motion planning tools such as MoveIt! provide mature algorithms and well defined but fairly restrictive interfaces. EXOTica complements this with a more generic and open architecture that allows for rapid prototyping and benchmarking of algorithms at a much earlier stage of development.

Authors

Vladimir Ivan received his Ph.D. on the topic of Motion synthesis in topology-based representations at the University of Edinburgh where he is currently working as a Research Associate in the School of Informatics. He has previously received a M.Sc. in Artificial Intelligence specializing in Intelligent Robotics at the University of Edinburgh and a B.Sc. in AI and Robotics from the University of Bedfordshire. Vladimir has published over 20 peer-reviewed papers in top level conferences and journals. He contributed to several UK and EU funded academic research projects as well as industry-led projects with partners within EU and Japan and a collaboration with NASA-JSC.

Yiming Yang completed his Ph.D. in Robotics from the University of Edinburgh, in December 2017. His main areas of research interest are robot planning and control algorithms, human-robot interaction, collaborative robots, shared autonomy and machine learning for robotics. He is currently working as a Research Associate at the School of Informatics, University of Edinburgh, where he has been involved in several research projects in robotics, such as the NASA Valkyrie humanoid project, Hitachi Logistics Robotics, collaborative mobile robot manipulation is complex scenes, etc. Yiming has published over ten peer-reviewed articles in international journals, conferences and workshops.

Wolfgang Merkt is a Ph.D. student in Robotics at the University of Edinburgh, with a focus on robust online replanning in high-dimensional and real-world applications through efficient precomputation, machine learning, and storage. Wolfgang obtained an M.Sc. in Robotics from the University of Edinburgh focusing on motion planning, perception, and shared autonomy, and a B.Eng. in Mechanical Engineering with Management with a thesis on human-to-humanoid motion transfer using topological representations and machine learning. He has been involved in several research projects in robotics, such as the NASA Valkyrie humanoid project, collaborative mobile robot manipulation in complex scenes, and previously worked in service robotics and on industrial robotics applications for small and medium enterprises (co-bots).

Michael P. Camilleri is a Doctoral Student in the CDT in Data Science at the University of Edinburgh, investigating Time-Series Modelling for mouse behavior. His interests are in probabilistic machine learning, Bayesian statistics and Deep Learning. Michael obtained an M.Sc. in Artificial Intelligence at the University of Edinburgh, focusing on the motion planning for the ATLAS robot for the DARPA Robotics Challenge 2013. Following this, he was employed with the same University, and was the lead developer of the original iterations of EXOTica. He subsequently spent time working in industry and academia, in various software engineering, management and lecturing positions.

Professor Sethu Vijayakumar FRSE holds a Personal Chair in Robotics within the School of Informatics at the University of Edinburgh and is the Director of the Edinburgh Centre for Robotics. Since 2007, he holds the Senior Research Fellowship of the Royal Academy of Engineering, co-funded by Microsoft Research and is also an Adjunct Faculty of the University of Southern California (USC), Los Angeles and a Visiting Research Scientist at the ATR Computational Neuroscience Labs, Kyoto-Japan. He has a Ph.D.(1998) in Computer Science and Engineering from the Tokyo Institute of Technology. His research interests include planning and control of anthropomorphic robotic systems, statistical machine learning, human motor control and rehabilitation robotics and he has published over 170 refereed papers in journals and conferences in these areas.

References

1. S. Chitta, I. Sucan, and S. Cousins, "Moveit! [ros topics]," *IEEE Robotics Automation Magazine (RAM)*, vol. 19, no. 1, pp. 18–19, 2012.
2. D. V. Lu, M. Ferguson, and E. Marder-Eppstein, "ROS Navigation Stack." [Online]. Available: `https://github.com/ros-planning/navigation`.
3. J.-L. Blanco-Claraco, "Reactive navigation for 2D robots using MRPT navigation algorithms." [Online]. Available: `http://wiki.ros.org/mrpt_reactivenav2d`.
4. B. Siciliano and O. Khatib, *Springer Handbook of Robotics*. Springer Handbook of Robotics, Springer Berlin Heidelberg, 2008.
5. S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006. [Online]. Available: `http://planning.cs.uiuc.edu/`.
6. M. Toussaint, "Robot trajectory optimization using approximate inference," in *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, ICML '09, (New York, NY, USA), pp. 1049–1056, ACM, 2009.
7. J. J. Kuffner and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, vol. 2, pp. 995–1001, 2000.
8. I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine (RAM)*, vol. 19, no. 4, pp. 72–82, 2012. [Online]. Available: `http://ompl.kavrakilab.org`.
9. A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, 2013. [Online]. Available: `http://octomap.github.com`.
10. R. Smits, "KDL: Kinematics and Dynamics Library." [Online]. Available: `http://www.orocos.org/kdl`.
11. D. Hershberger, D. Gossow, and J. Faust, "RViz: 3D visualization tool for ROS." [Online]. Available: `http://wiki.ros.org/rviz`.
12. T. Foote, E. Marder-Eppstein, and W. Meeussen, "TF2: transform library for ROS." [Online]. Available: `http://wiki.ros.org/tf2`.
13. J.-L. Blanco, "A tutorial on se(3) transformation parameterizations and on-manifold optimization," tech. rep., University of Malaga, 2010.
14. Y. Nakamura and H. Hanafusa, "Inverse kinematic solutions with singularity robustness for robot manipulator control," *ASME, Transactions, Journal of Dynamic Systems, Measurement, and Control*, vol. 108, pp. 163–171, 1986.
15. K. Rawlik, M. Toussaint, and S. Vijayakumar, "On stochastic optimal control and reinforcement learning by approximate inference (extended abstract)," in *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 3052–3056, AAAI Press, 2013.
16. H. Dai, A. Valenzuela, and R. Tedrake, "Whole-body motion planning with centroidal dynamics and full kinematics," in *Proceedings of IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, pp. 295–302, 2014.
17. S. Bradley, A. Hax, and T. Magnanti, *Applied Mathematical Programming*. Addison-Wesley Publishing Company, 1977.
18. R. Deits and R. Tedrake, "Footstep planning on uneven terrain with mixed-integer convex optimization," in *Proceedings of IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, pp. 279–286, 2014.
19. B. D. Luders, S. Karaman, and J. P. How, "Robust sampling-based motion planning with asymptotic optimality guarantees," in *Proceedings of the AIAA Guidance, Navigation, and Control (GNC) Conference*, American Institute of Aeronautics and Astronautics, 2013.

20. M. Hutter, H. Sommer, C. Gehring, M. Hoepflinger, M. Bloesch, and R. Siegwart, "Quadrupedal locomotion using hierarchical operational space control," *The International Journal of Robotics Research (IJRR)*, vol. 33, no. 8, pp. 1047–1062, 2014.
21. M. Toussaint, "A tutorial on Newton methods for constrained trajectory optimization and relations to SLAM, Gaussian Process smoothing, optimal control, and probabilistic inference," in *Geometric and Numerical Foundations of Movements* (J.-P. Laumond, ed.), Springer, 2017.
22. E. Todorov and W. Li, "A generalized iterative lqg method for locally-optimal feedback control of constrained nonlinear stochastic systems," in *Proceedings of the American Control Conference*, pp. 300–306, 2005.
23. Y. Yang, V. Ivan, W. Merkt, and S. Vijayakumar, "Scaling Sampling-based Motion Planning to Humanoid Robots," in *Proceedings of IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pp. 1448–1454, 2016.
24. V. Ivan, D. Zarubin, M. Toussaint, T. Komura, and S. Vijayakumar, "Topology-based representations for motion planning and generalization in dynamic environments with interactions," *The International Journal of Robotics Research (IJRR)*, vol. 32, pp. 1151–1163, 2013.
25. Y. Yang, V. Ivan, and S. Vijayakumar, "Real-time motion adaptation using relative distance space representation," in *Proceedings of IEEE International Conference on Advanced Robotics (ICAR)*, pp. 21–27, 2015.
26. V. Ivan and S. Vijayakumar, "Space-time area coverage control for robot motion synthesis," in *Proceedings of IEEE International Conference on Advanced Robotics (ICAR)*, pp. 207–212, 2015.
27. W. Merkt, Y. Yang, T. Stouraitis, C. E. Mower, M. Fallon, and S. Vijayakumar, "Robust shared autonomy for mobile manipulation with continuous scene monitoring," in *Proceedings of IEEE International Conference on Automation Science and Engineering (CASE)*, pp. 130–137, 2017.
28. Y. Yang, V. Ivan, Z. Li, M. Fallon, and S. Vijayakumar, "iDRM: Humanoid motion planning with realtime end-pose selection in complex environments," in *Proceedings of IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)*, pp. 271–278, 2016.
29. Y. Yang, W. Merkt, H. Ferrolho, V. Ivan, and S. Vijayakumar, "Efficient humanoid motion planning on uneven terrain using paired forward-inverse dynamic reachability maps," *IEEE Robotics and Automation Letters (RA-L)*, vol. 2, no. 4, pp. 2279–2286, 2017.