DART: Dynamic Animation and Robotics Toolkit

1 Introduction

DART (Dynamic Animation and Robotics Toolkit) is a collaborative, cross-platform, open source library created by the Georgia Tech Graphics Lab and Humanoid Robotics Lab. The library provides data structures and algorithms for kinematic and dynamic applications in robotics and computer animation. DART is distinguished by its accuracy and stability due to its use of generalized coordinates to represent articulated rigid body systems and Featherstone's Articulated Body Algorithm to compute the dynamics of motion. For developers, in contrast to many popular physics engines which view the simulator as a black box, DART gives full access to internal kinematic and dynamic quantities, such as the mass matrix, Coriolis and centrifugal forces, transformation matrices and their derivatives. DART also provides efficient computation of Jacobian matrices for arbitrary body points and coordinate frames. The frame semantics of DART allows users to define arbitrary reference frames (both inertial and non-inertial) and use those frames to specify or request data. For air-tight code safety, forward kinematics and dynamics values are updated automatically through lazy evaluation, making DART suitable for real time controllers. In addition, DART gives provides flexibility to extend the API for embedding user-provided classes into DART data structures. Contacts and collisions are handled using an implicit time-stepping, velocity-based LCP (linear-complementarity problem) to guarantee non-penetration, directional friction, and approximated Coulomb friction cone conditions. DART has applications in robotics and computer animation because it features a multibody dynamic simulator and various kinematic tools for control and motion planning. Multibody dynamic simulation in DART is an extension of RTQL8, an open source software created by the Georgia Tech Graphics Lab.

Since DART launched on Github in 2011, an active group of researchers in Computer Animation and Robotics has been constantly improving the usability of DART, enhancing the efficiency and accuracy of simulation, and adding numerous practical dynamic and kinematic tools. This document highlights a set of important features of DART based on Version 5.1.

General

- Open source under BSD licence written in C++.
- Support Linux, Mac OSX, and Windows.
- Fully integrated with Gazebo.
- Support models described in URDF and SDF formats.
- Provide default integration methods, semi-implicit Euler and RK4, as well as extensible API for other numerical integration methods.

- Support multiple collision detectors: FCL and Bullet.
- Support lazy evaluation and automatic update of kinematic and dynamic quantities.
- Provide extensible API for embedding user-provided classes into DART data structures.
- Support comprehensive recording of events in simulation history.
- Support OpenGL and OpenSceneGraph.
- Provide extensible API to interface with various optimization methods

Kinematics

- Support numerous types of Joint.
- Support numerous primitive and arbitrary body shapes with customizable inertial and material properties.
- Support flexible skeleton modeling: cloning and reconfiguring skeletons or subsections of a skeleton.
- Provide comprehensive access to kinematic states (e.g. transformation, position, velocity, or acceleration) of arbitrary entity and coordinate frames
- Provide comprehensive access to various Jacobian matrices and their derivatives.
- Support flexible conversion of coordinate frames.
- A fully modular inverses kinematics framework
- A plug-and-play hierarchical whole-body inverse kinematics solver

Dynamics

- Achieve high performance for articulated dynamic systems using Lie Group representation and Featherstone hybrid algorithms.
- Enforce joints between body nodes exactly using generalized coordinates.
- Provide comprehensive API for dynamic quantities and their derivatives, such as mass matrix, Coriolis force, gravitational force, other external and internal forces.
- Support both rigid and soft body nodes.
- Model viscoelastic joint dynamics with joint friction and hard joint limits.
- Support various types of actuators.
- Handle contacts and collisions using an implicit LCP to guarantee non-penetration, directional friction, and approximated Coulomb friction cone conditions.
- Support "Island" technique to subdivide constraint handling for efficient performance.
- Support various Cartesian constraints and provide extensible API for user-defined constraints.
- Provide multiple constraint solvers: Lemke method, Dantzig method, and PSG method.
- Support dynamic systems with closed-loop structures.

2 Kinematic Features

Kinematic data structures in DART are designed to be comprehensive, extensible, and efficient for dynamic computation. Our design principles are heavily influenced by practical use cases

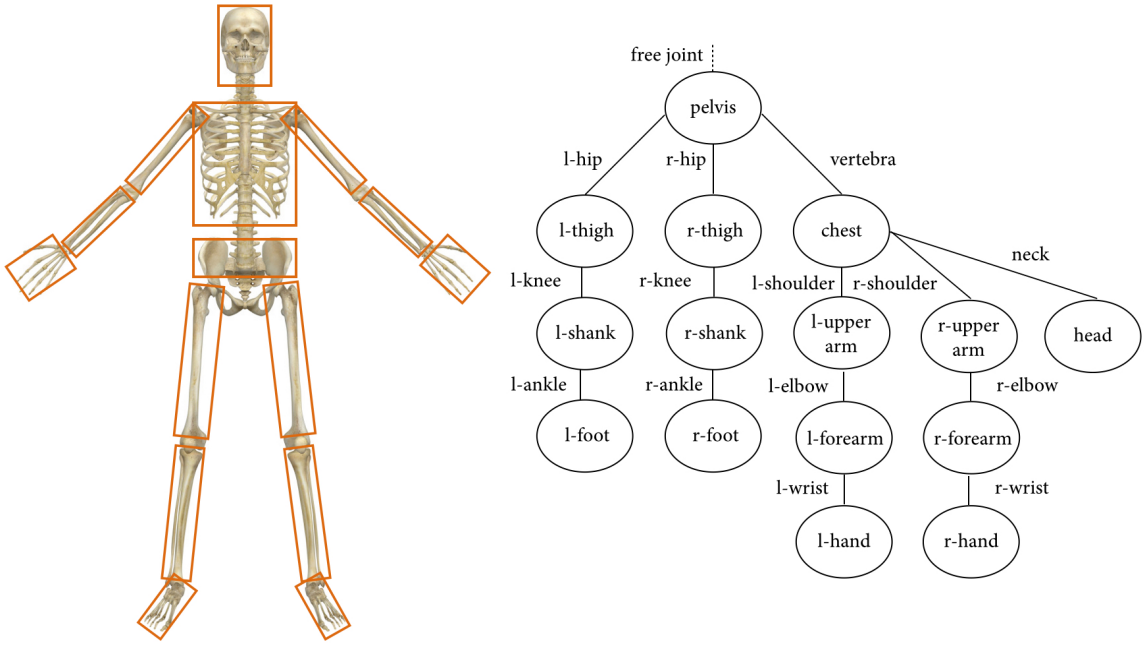

Figure 1: Tasks for each research aim.

suggested by researchers and practitioners in Robotics. We also follow the guidelines proposed by OSRF to make DART compatible with Gazebo standard.

2.1 Skeleton, BodyNode, and Joint

In DART, an articulated dynamics model is represented by a **Skeleton**. A Skeleton is a tree structure that consists of **BodyNode**s which are connected by **Joint**s. Every Joint has a child BodyNode, and every BodyNode has a parent Joint. Even the root BodyNode has a Joint that attaches it to the World. For example, the human skeleton can be organized into a tree structure (Figure 1) with nodes representing BodyNodes and edges representing Joints. Depending on the joint type (Section 2.3), each joint has a specific number of degrees of freedom (DOFs). For example, the hip joint can be modelled by a **EulerJoint** with three DOFs, while a knee joint can be modelled by a **RevoluteJoint** with one DOF. In this model, we select the pelvis to be the root BodyNode, which connects to the World via a **FreeJoint** with six DOFs.

A kinematic chain is a sequence of transformations from one BodyNode to another in the Skeleton. Figure 2(a) illustrates the transformations between two intermediate BodyNodes denoted as Parent and Child. The coordinate frames of Parent, Child, and the joint between them are shown as the RGB arrows. Let $\mathbf{T}_{pj}$ be the transformation from the Parent frame to the joint frame and $\mathbf{T}_{cj}$ be the transformation from the Child frame to the joint frame. These two transformations are fixed and can be customized as part of the properties of the joint. The transformation of the joint $\mathbf{T}(\mathbf{q})$ is a function of the DOFs, $\mathbf{q}$, associated with the joint. In the case of the hip joint, $\mathbf{q}$ is a 3x1 vector consisting of hip rotation angles in three dimensions. Thus, the transformation from the Parent frame to the Child frame is denoted

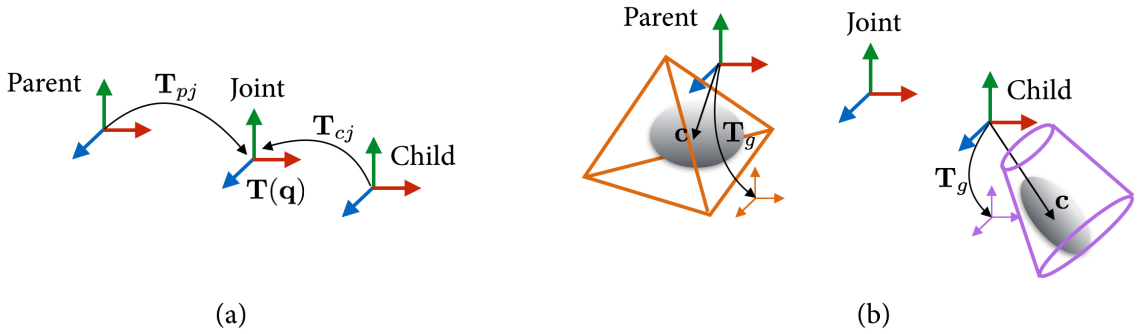

Figure 2: Tasks for each research aim.

by $\mathbf{T}_{pj}\mathbf{T}(\mathbf{q})\mathbf{T}_{cj}^{T}$.

2.2 BodyNode properties

A BodyNode contains a set of customizable properties to define its kinematic and dynamic behaviors.

- **Inertial properties:** The user can define the mass, the position of the center of mass in the BodyNode frame, and the moment of inertia around the center of mass in the BodyNode frame. The ellipsoids in Figure 2(b) illustrate the center of mass ($\mathbf{c}$) and the moment of inertia defined in the BodyNode frame.
- **Geometry properties:** The user can associate a set of **Shape**s with a BodyNode. Each shape has its own geometry information used by rendering and collision detection routines. DART supports a few primitive shapes, box, ellipsoid, cylinder, plane, line segment, in addition to arbitrary 3D polygons. Each shape can be defined in its own coordinate frame. The spatial relation between the Shape and its associated BodyNode is defined by a fixed transformation $\mathbf{T}_g$ (Figure 2(b)).
- **Collision properties:** The user can set the friction coefficient and the coefficient of restitution of a BodyNode, as well as the flag that determines whether the BodyNode is collidable.

2.3 Joint types

DART supports 10 types of joints. Each joint type has a fixed number of DOFs and a specific configuration domain. Some joint types require the user to define their unique properties. For example, the rotation axes of **RevoluteJoint**, **UniversalJoint** and **PlanarJoint** need to be specified when the joint is constructed. Likewise, the order of three axes ('xyz', 'zyx', etc) in an **EulerJoint** needs to be predefined. Note that both **BallJoint** and **EulerJoint** are used to represent rotational motion in 3D space. However, a **BallJoint** is represented by an exponential map while an **EulerJoint** is represented by three consecutive 1D rotational matrices. All joint types provided by DART are listed in Table 1.

2.4 Frame semantics

Grey

Table 1: Joint Types

Name	#DOF	Configuration Domain	Properties
BallJoint	3	$SO(3)$	
EulerJoint	3	$SO(2) \times SO(2) \times SO(2)$	3-axis order
FreeJoint	6	$SE(3)$	
PlanarJoint	3	$SE(2)$	axis1 and axis2
PrismaticJoint	1	R	axis
RevoluteJoint	1	$SO(2)$	axis
ScrewJoint	1	$SO(2)$	axis and pitch
TranslationalJoint	3	R^3	
UniversalJoint	2	$SO(2) \times SO(2)$	axis1 and axis2
WeldJoint	0	$\emptyset$	

2.5 Kinematic state of the system

DART provides a variety of ways to query the position and the velocity of the system in generalized, Cartesian, and spatial coordinates.

```
// Generalized coordinates
Eigen::VectorXd MetaSkeleton::getPositions() const;
Eigen::VectorXd MetaSkeleton::getVelocities() const;

// Cartesian coordinates
Eigen::Isometry3d Frame::getTransform(const Frame* _withRespectTo =
    Frame::World(), const Frame* _inCoordinatesOf = Frame::World()) const;
Eigen::Vector3d Frame::getLinearVelocity(const Eigen::Vector3d& _offset, const
    Frame* _relativeTo = Frame::World(), const Frame* _inCoordinatesOf =
    Frame::World()) const;
Eigen::Vector3d Frame::getAngularVelocity(const Frame* _relativeTo =
    Frame::World(), const Frame* _inCoordinatesOf = Frame::World()) const;
Eigen::Vector3d Skeleton::getCOM(const Frame* _withRespectTo = Frame::World())
    const override;
Eigen::Vector3d Skeleton::getCOMLinearVelocity(const Frame* _relativeTo =
    Frame::World(), const Frame* _inCoordinatesOf = Frame::World()) const
    override;
Eigen::Vector3d BodyNode::getCOM(const Frame* _withRespectTo = Frame::World())
    const;
Eigen::Vector3d BodyNode::getCOMLinearVelocity(const Frame* _relativeTo =
    Frame::World(), const Frame* _inCoordinatesOf = Frame::World()) const;

// Spatial coordinates
Eigen::Vector6d Frame::getSpatialVelocity(const Eigen::Vector3d& _offset, const
    Frame* _relativeTo, const Frame* _inCoordinatesOf) const;
```

For example, if one wishes to know the transformation of the left hand from the left upper arm expressed in the coordinate frame of the pelvis, it can be achieved by

```
// Assume l-hand, l-upper-arm, and pelvis are pointers of BodyNode
 Isometry3d T = l-hand->getTransform(l-upper-arm, pelvis);
```

The velocity of any point in any BodyNode frame relative to any other frame can be queried. For example, the relative linear velocity of a local point, *offset*, in the *l-hand* frame from the origin of the *l-upper-arm* frame expressed in the *pelvis* frame can be accessed by

```
Vector3d v = l-hand->getLinearVelocity(offset, l-upper-arm, pelvis);
```

2.6 Jacobian matrices

Calculating Jacobian matrix is a common theme in forward simulation, inverse kinematics, and many other robotics or graphics applications. DART computes Jacobian matrices and its derivatives efficiently and provides API to access the results in various forms.

Skeleton. The full Jacobian matrix with respect to all DOFs in the system can be accessed via the member functions of Skeleton.

Listing 1: Skeleton.h

```
math::LinearJacobian getLinearJacobian(const BodyNode* _bodyNode, const
    Eigen::Vector3d& _localOffset, const Frame* _inCoordinatesOf =
    Frame::World()) const override;

math::AngularJacobian getAngularJacobian(const BodyNode* _bodyNode, const Frame*
    _inCoordinatesOf = Frame::World()) const override;
```

The first function returns J_v that maps the generalized velocity of the system to the velocity of the point attached to *bodyNode* with an offset, *localOffset*, from the origin of *bodyNode*, $\mathbf{v} = J_v\dot{\mathbf{q}}$. The second function returns J_ω that maps the generalized velocity to the angular velocity of *bodyNode*, $\omega = J_\omega\dot{\mathbf{q}}$. The last optional parameter, *inCoordinatesOf*, determines in which coordinate frame the Jacobian matrix is expressed. The full Jacobian matrix that combines both J_v and J_ω can be obtained by the following function.

Listing 2: Skeleton.h

```
math::Jacobian getJacobian(const BodyNode* bodyNode, const Eigen::Vector3d&
    localOffset, const Frame* inCoordinatesOf) const override;
```

BodyNode. More compact Jacobian matrices can be acquired via the member functions of a BodyNode. Unlike the Jacobain matrices for the entire Skeleton, these Jacobian matrices have m columns, the number of DOFs on the kinematic chain from the root node to the BodyNode of interest.

Listing 3: BodyNode.h

```
math::LinearJacobian getLinearJacobian(const Eigen::Vector3d& _offset, const
    Frame* _inCoordinatesOf = Frame::World()) const;

math::AngularJacobian getAngularJacobian(const Frame* _inCoordinatesOf =
    Frame::World()) const;
```

```
math::Jacobian getJacobian(const Eigen::Vector3d& _offset, const Frame*
    _inCoordinatesOf) const;
```

Joint. Similar Jacobian functions are also available for the Joint class. The number of columns of the Jacobian depends on the number of DOFs of the joint. For example, a BallJoint returns a 6×3 Jacobian while a RevoluteJoint returns a 6×1 Jacobian.

Listing 4: Joint.h

```
virtual math::Jacobian getLocalJacobian(const Eigen::VectorXd&
_positions) const = 0;
```

2.7 Skeleton modeling

Building a complex skeleton from scratch can be very difficult and tedious. Often time, the developer may prefer to modify an existing skeleton described by a URDF or SDF file. DART provides many functions to facilitate skeleton editing summarized in the following table.

Table 2: Functions for Skeleton Modeling

Function Example	**Description**
bd1->**remove**()	Remove BodyNode bd1 and its subtree from their Skeleton.
bd1->**moveTo**(bd2)	Move BodyNode bd1 and its subtree under BodyNode bd2.
auto newSkel = bd1->**split**("new skeleton")	Remove BodyNode bd1 and its subtree from their current Skeleton and move them into a newly created Skeleton named "new skeleton".
bd1->**changeParentJointType**<BallJoint>()	Change the joint type of BodyNode bd1's parent joint to BallJoint.
bd1->**copyTo**(bd2)	Create a clone of BodyNode bd1 and its subtree and attach the clone to an existing BodyNode bd2.
auto newSkel = bd1->**copyAs**("new skeleton")	Create a clone of BodyNode bd1 and its subtree as a new skeleton named "new skeleton".

Some functions have optional parameters for more advanced modeling operations. For example, **moveTo** is templated by **JoinType** and can take in **Joint::Properties** as an argument, such as: bd1-¿moveTo¡EulerJoint¿(bd2, properties).

2.8 Inverse kinematics

Grey

3 Dynamic Features

Physics simulation is the core of DART. The earlier versions of DART adopted the physics engine, RTQL8 [], which features forward simulation based on Lagrangian dynamics and generalized coordinates. Later, we reimplemented the Featherstone algorithms using Lie group representation to largely improve the computation efficiency. The constraint solver, which handles contact, joint limits, and other Cartesian constraints, has also been optimized in the later versions of DART.

3.1 Lagrangian dynamics and generalized coordinates

DART is distinguished by its accuracy and stability due to its use of generalized coordinates to represent articulated rigid body systems, Featherstones algorithm to compute the dynamics of motion, and Lie group for compact representation and derivation of dynamic equations.

Articulated human motions can be described by a set of dynamic equations of motion of multibody systems. Since the direct application of Newtons second law becomes difficult when a complex articulated rigid body system is considered, we use Lagranges equations derived from D'Alemberts principle to describe the dynamics of motion:

$$M(\mathbf{q})\ddot{\mathbf{q}} + C(\mathbf{q}, \dot{\mathbf{q}}) = \mathbf{Q}, \tag{1}$$

where $\mathbf{q}$ is the generalized coordinates that indicate the configuration of the skeleton. $M(\mathbf{q})$ is the mass matrix, $C(\mathbf{q}, \dot{\mathbf{q}})$ is the Coriolis and centrifugal term of the equation of motion, and $\mathbf{Q}$ is the vector of generalized forces for all the degrees of freedom in the system.

Once we know how to compute the mass matrix, Coriolis and centrifugal terms, and generalized forces, we can compute the acceleration in generalized coordinates, $\ddot{\mathbf{q}}$ for forward dynamics. Conversely, if we are given $\ddot{\mathbf{q}}$ from a motion sequence, we can use these equations of motion to derive generalized forces for inverse dynamics. However, directly evaluating the mass matrix and Coriolis term is computationally expensive, especially for applications that require real-time simulation. DART implements Featherstone's articulated rigid body algorithms to compute forward and inverse dynamics. In addition, DART uses Lie group representation to further improve efficiency.

Featherstone's algorithms. In particular, we implement the recursive Newton-Euler algoritm (RNEA) for inverse dynamics and the articulated-body algorithm (ABA) for forward dynamics. These two algorithms are known for their $O(n)$ time with n being the number of body nodes in the skeleton.

Lie group representation JS (Describe the high-level ideas of Lie group representation and its advantages).

3.2 Soft body simulation

Karen

3.3 Joint dynamics

Karen

3.4 Actuators

Karen

3.5 Constraints

JS

Contacts

Cartesian constraints

LCP solvers

3.6 Performance

JS

4 Other Features

4.1 Collision detectors

JS

4.2 Integration methods

Karen

4.3 Lazy evaluation and automatic update

Grey

4.4 Extensible data structures

Grey

4.5 History recorder

4.6 Optimization interface

Grey

4.7 Extensible GUI

JS