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
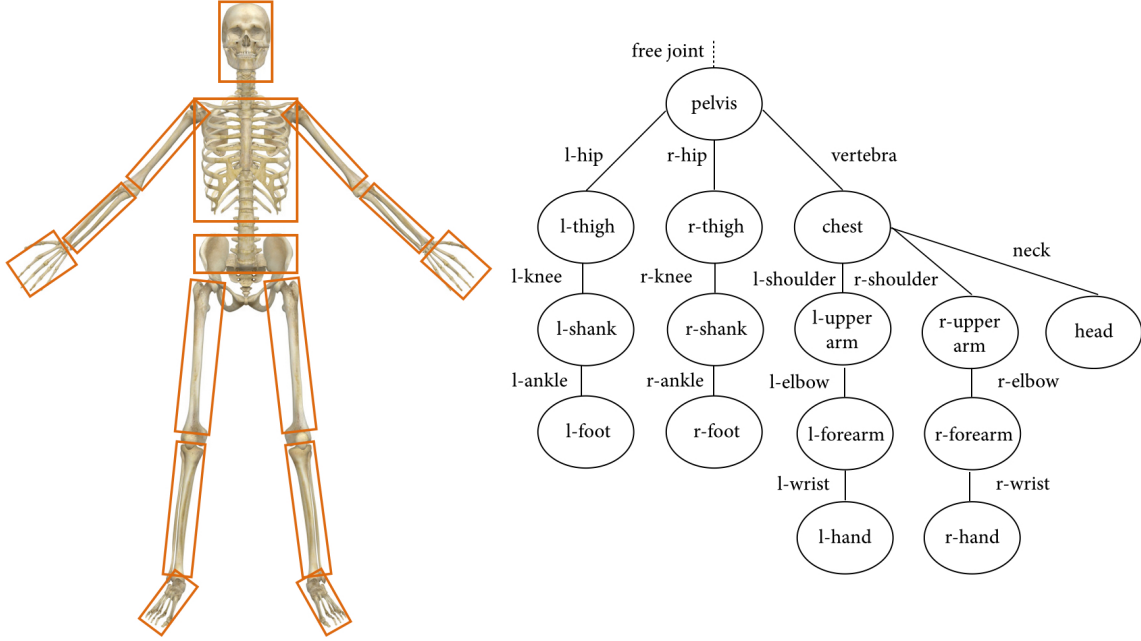
Figure 1: Tasks for each research aim.

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 modeled by an **EulerJoint** with three DOFs, while a knee joint can be modeled 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 **Joint::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
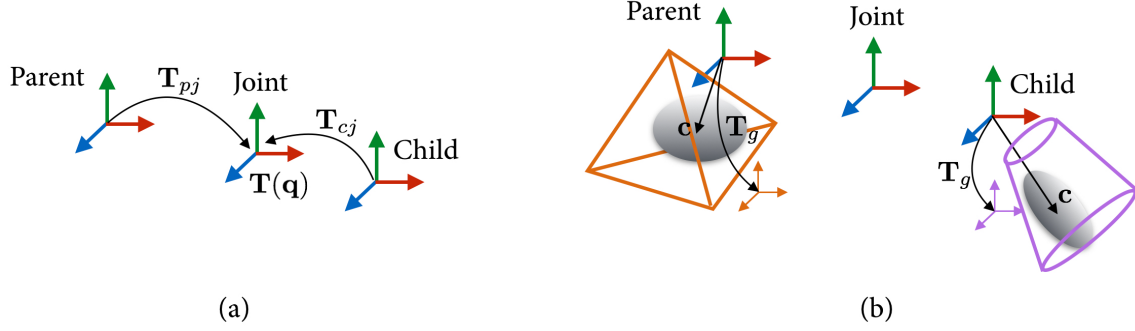
3

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 **BodyNode::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.
  TODO(MXG): We should probably mention ShapeNode when discussing geometry
- **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 **UniqueProperties**. 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. These properties can be changed at any time, and the kinematics and dynamics calculations will automatically adjust to account for those changes. The only potential negative impact would be non-physical discontinuities if property changes are made during a simulation. 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.

Table 1: Joint Types

Name	#DOF	Configuration Domain	Unique 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	$\emptyset$	

2.4   Frame semantics

Mathematical formulae are often expressed in terms of reference frames. This can make it difficult to directly program the formulae if your kinematic library is less expressive than the mathematical language. If a kinematic library only offers a frame's transformation with respect to its parent frame or the world frame, then you may need to repeatedly compute relative transformations between two arbitrary frames, which can easily introduce errors due to typos or confusion. Worse yet, the formulae for computing relative velocities and relative accelerations in non-inertial reference frames can be exceedingly complex and difficult to implement correctly. Take for example the equation to compute the relative linear acceleration of **Frame B** relative to **Frame A** in the coordinates of **Frame F**:

$$^F a_{BA} =^F R_O(^O a_B -^O a_A -^O \dot{\omega}_A \times^O p_{BA} - 2^O \omega_{BA} \times^O v_{BA} -^O \omega_A \times (^O \omega_A \times^O p_{BA}) \quad (1)$$

Any small mistake, such as mixing up a plus with a minus or switching two of the variables, will produce potentially devastatingly incorrect results. Even the individual variables in the above expression each has a non-trivial formula to compute it, which must be implemented perfectly in order for any computations to come out correctly. These issues can pose considerable barriers to writing complex controller logic which is reliable enough to safely run on expensive robots.

KIDO addresses this issue by providing frame semantics. The forward kinematics of KIDO are handled by the underlying **Frame** class. The Frame class is a pure virtual class which provides the user with an interface to compute the kinematic values of a frame with respect to any other arbitrary frame. So instead of needing to implement equation 1, KIDO allows you to simply call the function:

```
B->getLinearAcceleration(A,F);
```

Here are a few examples of what the Frame semantics API looks like:

```cpp
// Transformation Matrix
Eigen::Isometry3d Frame::getWorldTransform() const;
Eigen::Isometry3d Frame::getTransform(
    const Frame* withRespectTo = Frame::World()) const;

// Velocity
Eigen::Vector3d getLinearVelocity(
    const Frame* relativeTo = Frame::World(),
    const Frame* inCoordinatesOf = Frame::World()) const;
Eigen::Vector3d getAngularVelocity(
    const Frame* relativeTo = Frame::World(),
    const Frame* inCoordinatesOf = Frame::World()) const;

// Acceleration
Eigen::Vector3d getLinearAcceleration(
    const Frame* relativeTo = Frame::World(),
    const Frame* inCoordinatesOf = Frame::World()) const;
```

Note that an **Eigen::Isometry3d** represents a specialized type of transformation matrix which exists in SE3.

Since **Frame** is a pure virtual interface class, we also need to have concrete implementations of it in order to perform forward kinematics. Most of the functions in **Frame** are actually implemented already; there are only five functions which derived classes need to implement:

```cpp
virtual const Eigen::Isometry3d& getRelativeTransform() const = 0;
virtual const Eigen::Vector6d& getRelativeSpatialVelocity() const = 0;
virtual const Eigen::Vector6d& getRelativeSpatialAcceleration() const = 0;
virtual const Eigen::Vector6d& getPrimaryRelativeAcceleration() const = 0;
virtual const Eigen::Vector6d& getPartialAcceleration() const = 0;
```

In the first three functions, the term *Relative* means the quantity is relative to its parent frame. The first three functions are all that is needed to perform forward kinematics. The last two functions are used to reduce the number of computations when performing Featherstone's Articulated Body Algorithm for forward dynamics.

KIDO currently provides four concrete implementations of the **Frame** class:

2.4.1 BodyNode Implementation

The **BodyNode** class is used to define the kinematic structure of a **Skeleton**. Because of this, it is very helpful for **BodyNode** to inherit the **Frame** class and be compatible with all of its forward kinematics and frame semantics features. The transformation, velocity, and acceleration of a **BodyNode** relative to its parent **Frame** is fully determined by its parent **Joint**. In the general case, it is not possible to explicitly move a **BodyNode** to an arbitrary transformation or give it an arbitrary velocity or acceleration. These physical quantities are all constrained by the parent **Joint** of the **BodyNode** as well as every **Joint** that it descends from. To achieve an arbitrary transformation for a **BodyNode**, KIDO offers an **InverseKinematics** module discussed in section 2.8. KIDO does not currently offer an inverse *differential* kinematics module, but implementing one is straightforward with the tools

that KIDO does provide.

The BodyNode class and how to use it will be discussed in far greater detail throughout the rest of this report. In many ways, it is the cornerstone class of KIDO.

2.4.2 SimpleFrame Implementation

In some cases, it might be too cumbersome to be constrained by a **Joint** the way **BodyNode** is. If you want to have a reference **Frame** with a completely arbitrary transform, velocity, and acceleration, then being constrained by **Joint** can make things more difficult. Also, if you want your frame to represent a reference frame which is not a physical body, then the **BodyNode** class would be overkill.

For these reasons, we offer the **SimpleFrame** class. The **SimpleFrame** class is simply a Frame; it does not have any collision or inertial properties and therefore cannot be simulated. However, what **SimpleFrame** does offer is the ability to arbitrarily set its transformations, velocities, and accelerations. This can be done with the following functions:

```
// Set transformation relative to parent Frame
void SimpleFrame::setRelativeTransform(const Eigen::Isometry3d& newRelTransform);

// Set transformation relative to any other Frame
void SimpleFrame::setTransform(
   const Eigen::Isometry3d& newTransform,
   const Frame* withRespectTo = Frame::World());

// Set the spatial velocity relative to the parent Frame, in the coordinates of
    this Frame
void SimpleFrame::setRelativeSpatialVelocity(
   const Eigen::Vector6d& newSpatialVelocity);

// Set the spatial velocity relative to the parent Frame, in the coordinates of
    any Frame
void SimpleFrame::setRelativeSpatialVelocity(
   const Eigen::Vector6d& newSpatialVelocity,
   const Frame* inCoordinatesOf);

// Set the spatial acceleration relative to the parent Frame, in the coordinates
    of this Frame
void SimpleFrame::setRelativeSpatialAcceleration(
   const Eigen::Vector6d& newSpatialAcceleration);

// Set the spatial acceleration relative to the parent Frame, in the coordinates
    of any Frame
void SimpleFrame::setRelativeSpatialAcceleration(
   const Eigen::Vector6d& newSpatialAcceleration,
   const Frame* inCoordinatesOf);
```

It is important to note that, by default, spatial quantities like spatial velocity and spatial acceleration should be given in the coordinates of the frame that they belong to (**not** in the coordinates of the parent frame). But KIDO does provide convenience functions that allow you to represent the spatial quantities in whichever coordinates you would like to. However,

they still must represent the quantities **relative to** the parent, even if they're expressed in the coordinates of a different frame (i.e. the values are rotated).

For those who are not familiar or not comfortable with spatial quantities, KIDO also offers an API for classic 3D vectors, like those used in the traditional Newton-Euler equations of motion:

```cpp
void SimpleFrame::setClassicDerivatives(
    const Eigen::Vector3d& linearVelocity = Eigen::Vector3d::Zero(),
    const Eigen::Vector3d& angularVelocity = Eigen::Vector3d::Zero(),
    const Eigen::Vector3d& linearAcceleration = Eigen::Vector3d::Zero(),
    const Eigen::Vector3d& angularAcceleration = Eigen::Vector3d::Zero());
```

Even though KIDO provides an API that supports both spatial and classic vectors, under the hood it only utilizes spatial vectors, because spatial vectors require fewer operations for forward differential kinematics. This also means that only spatial vectors get stored under the hood, and therefore if a user wants to set the velocity and/or acceleration using classic vectors, then all classic vectors must be provided simultaneously, or else the final result would depend on the order in which the classic vectors get set, which would be confusing and undesirable behavior.

2.4.3 FixedFrame Implementation

The **BodyNode** frame implementation is constrained by Joints while the **SimpleFrame** frame implementation is completely unconstrained. The **FixedFrame** implementation allows the user complete freedom in setting the initial relative transform, but after that it may not move. Its relative velocity and relative acceleration vectors are always exactly zero. **FixedFrame** instances do not even offer a way to change the relative transform after construction.

However, it is still possible for a class to inherit the **FixedFrame** class, and the inheriting class does have the authority to alter the relative transform. For example, the **EndEffector** class is a derivative of **FixedFrame** and allows the user to change the relative transform freely. But no matter how the relative transform gets changed, the relative velocity and relative acceleration will always be reported as zero, so altering a relative transform during a simulation could result in non-physical discontinuities.

2.4.4 WorldFrame Implementation

The **WorldFrame** is a very special case of the **Frame** class. It always returns identity for its relative transformation as well as its world transformation. Its velocities and accelerations are always zero. Its parent frame is itself. It is a singleton class which only gets created once per program, and every kinematic chain ultimately descends from it. To access the **WorldFrame**, simply call the function **Frame::World()**.

Conventional wisdom and good software engineering practices dictate that singletons should be avoided almost always. In the case of the **WorldFrame**, it is designed to be a read-only object whose values are never changed. In some sense, it is used as a symbol rather than an object instance. Therefore it is not at risk of issues related to race conditions, un-

expected interdependencies within the code, or any of the other pitfalls that are commonly associated with singletons.

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.

```cpp
// 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 relative to the left upper arm expressed in the coordinate frame of the pelvis, it can be achieved by

```cpp
// 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 the Jacobian matrix is a common requirement 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, where m is 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;
```

JacobianNode. A generalization of the **BodyNode** API is available via the interface class **JacobianNode**. Arbitrary **Node**s in KIDO are able to inherit the **JacobianNode** class and utilize the exact same API that **BodyNode** provides. **Node**s are further explained in section **TODO(MXG): CITE NODE SECTION HERE**. An example of a **JacobianNode** class (besides BodyNode) is the **EndEffector** class, which allows users to specify a **FixedFrame** that is rigidly attached to a **BodyNode**. With that, Jacobians can be computed for arbitrary child transforms of a **BodyNode**.

Joint. Local 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. A **Joint**-level Jacobian is expressed in the coordinates of the **Joint**'s *child* **BodyNode**. They do not offer the same **Frame** semantics API that the more sophisticated Jacobians do, because the **Joint**-level Jacobians are primarily intended for low-level use.

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.

Some functions have optional parameters for more advanced modeling operations. For example, **moveTo** is templated by **JoinType** and can take in **JointType::Properties** as an argument, such as:

```
EulerJoint::Properties properties;
// ... modify properties here ... //
bd1->moveTo<EulerJoint>(bd2, properties).
```

2.8 Inverse Kinematics

Robotics and graphics applications often need fast and reliable inverse kinematics for controllers, planners, and animation. Inverse kinematics is any procedure which finds a set of

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".

generalized positions that allow an articulated body to satisfy some set of kinematic constraints. To put this in simpler terms, a common example of inverse kinematics would be to find a set joint angles that can place an end effector (a.k.a. robot hand) at a desired transformation. This can be thought of as the inverse process of forward kinematics: Forward Kinematics computes the transformation of a body given a set of joint angles, so Inverse Kinematics computes a set of joint angles given a transformation for a body.

One of the challenging aspects of Inverse Kinematics is that there are many approaches for performing it, and each approach has strengths and weaknesses which may depend on the context in which they are being used. Here are a few popular examples:

Jacobian Transpose Method   simply applies the transpose of the Jacobian matrix to an error vector and iterates until convergence, for example using gradient descent. This method is computationally inexpensive which allows it to be fast, and it is also very numerically stable. However, it tends to be imprecise, which might make it unsuitable for high-performance applications.

Inverse Jacobian Method   simply applies the inverse of the Jacobian matrix to an error vector and iterates until convergence. Computing the inverse of the Jacobian can be costly if the Jacobian is high-dimensional, and this method is very numerically unstable around singular configurations. However, it tends to be precise when it is not close to singularities. Another restriction of this method is that the error vector needs to match the dimensionality of the joint space, which restricts when this method can be applied

Pseudoinverse Jacobian Method is similar to the Inverse Jacobian Method except that it uses a pseudoinverse of the Jacobian instead of a normal inverse. This allows it to be applied in cases where the dimensionality of the error vector does not match the dimensionality of the joint space.

Damped Least Squares Pseudoinverse Jacobian Method is similar to the Pseudoinverse Jacobian Method, except it adds damping terms when computing the pseudoinverse. This allows the method to be robust and numerically stable around singularities with a very small penalty to the method's overall precision compared to the normal Pseudoinverse Jacobian Method.

Selectively Damped Least Squares Method[1] is similar to Damped Least Squares method except that instead of always damping with a fixed value, it adjusts the magnitude of the damping based on how close the generalized positions are to a singular configuration. This improves the precision compared to the standard Damped Least Squares method, but with the penalty of a slightly higher computational cost.

Analytical (a.k.a. Closed-Form) Solutions are symbolic expressions which directly compute the solutions to an inverse kinematics problem. Unlike the previously mentioned methods, analytical solutions do not rely on iterating. They have perfect numerical precision, are typically much faster than iterative methods, and cannot get caught in local minima. Analytical solutions can even recognize when a goal is unreachable by the kinematic structure and provide the "closest" solution to what was asked for. The key disadvantage of analytical solutions is that they do not exist for all kinematic structures, and even when one does exist, it may still be prohibitively difficult to find. Conversely, the iterative methods mentioned above are generalizable to all kinematic structures.

And these are only the most popular methods; others approaches may also exist. Despite any conceptual similarities between these methods, the implementation of each one tends to be drastically different. Each one may need to be tuned to each application that it gets used for. It is for this reason that finding a completely general inverse kinematics library is difficult. This is also why so many people resort to creating their own from scratch. KIDO seeks to break this pattern. Instead of simply providing an intransigent inverse kinematics solver, we offer an entire framework for performing inverse kinematics. This framework allows the user to freely swap out its solution finding method, and it even allows the user to create and plug in their own solution finding method. There is also a more advanced interface which allows users to plug in analytical solutions and access the features which are unique to analytical methods. KIDO comes pre-packaged with Jacobian Transpose and Damped Least Squares Pseudoinverse methods (underlined above). We do not currently offer any analytical solutions out of the box.

Besides variation in the methods that are used to solve inverse kinematics problems, there can also be a great deal of variation in the methods that are used to compute an error vector. In most cases, the error vector of an inverse kinematics problem is a 6×1 vector where three components refer to rotational error and the other three refer to translational

error. For lower dimensional spaces, such as a planar robot, the vector might have fewer dimensions. When there are multiple bodies that have kinematic constraints, the vector might have more dimensions. In KIDO's Inverse Kinematics framework, we expect each error method to produce a 6×1 vector. If fewer components are needed, then the method should fill in zero for each extra component. If multiple bodies have kinematic constraints, then you can plug in an error method for each body, and the hierarchical inverse kinematics solver can coordinate them.

The constraint functions that compute error vectors can also take on many forms. For example, there might be a zero-dimensional constraint manifold such as 3(a) where the translation and rotation of the robot's left hand is fully constrained to the target transformation **TODO(MXG): technically this is not a zero-dimensional constraint manifold since it can move its body around, so I should make a better example**. Or there may be a three-dimensional manifold like in 3(b) where the feet are constrained in height, pitch, and roll, but they are free to move across the surface of the floor and yaw. Or there could be a four-dimensional manifold like in 3(c) where the robot's end effector needs to touch the surface of the sphere, but it may rotate however it pleases.

3 Dynamic Features

Physics simulation is the core mission 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:

$$M(\mathbf{q})\ddot{\mathbf{q}} + C(\mathbf{q}, \dot{\mathbf{q}}) = \mathbf{Q}, \tag{2}$$

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
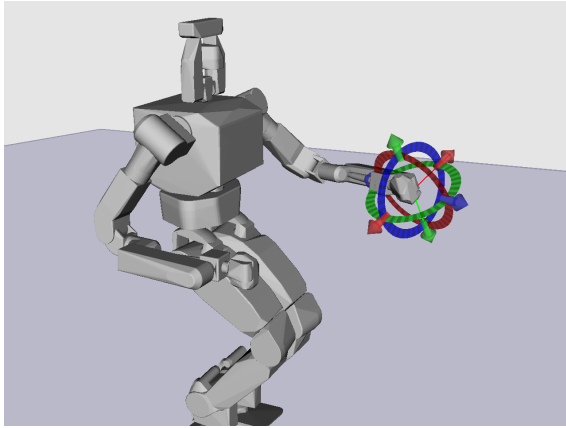
(a) Fully constrained end effector
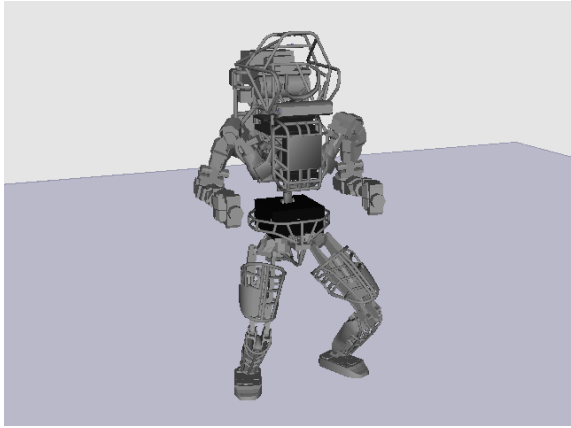
(b) Flat feet constraint
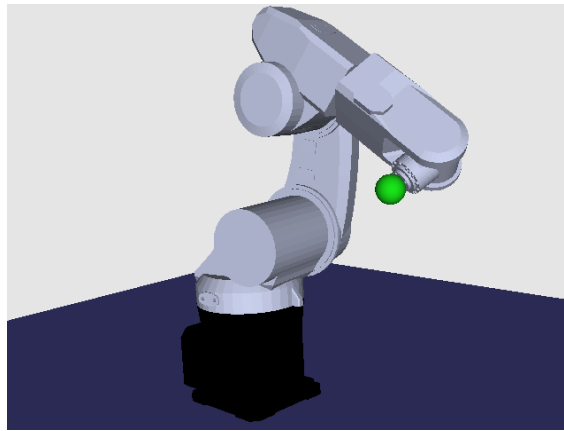
(c) Sphere touching constraint

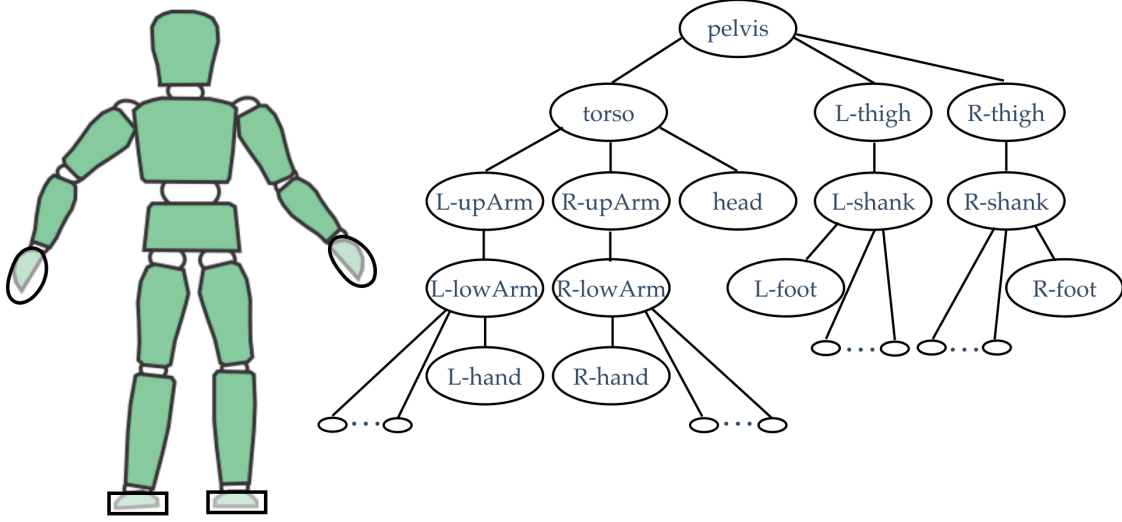Figure 3: Examples of various inverse kinematics constraints

Figure 4: The hands and the feet of the humanoid are composed of a rigid body with a layer of deformable surface. Both rigid and soft bodies are represented in generalized coordinates.

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

DART is able to simulate soft bodies dynamically coupling with rigid bodies. For example, one can simulate a robotic gripper with deformable surface or a humanoid with rubber soles on the feet. The soft body simulation in DART is based on spring-mass systems described in generalized coordinates. We use a unified representation for both rigid and soft bodies so that two-way dynamic coupling can be handled by the same set of equations of motion (Figure 4.

Each SoftBodyNode consists of a set of point masses representing the surface of the deformable part of the body. Each point mass is attached to the frame of BodyNode as a child link with three translational degrees of freedom. This compact representation allows us to soften or harden any part of deformable body at any time, by simply adding or removing the

translational degrees of freedom of involved point masses, without switching dynamic regimes or introducing any instability. Based on this flexible representation, we develop an efficient system where only the site of collision needs to be simulated as deformable body while the rest of the character remains rigid.

To make a BodyNode soft, one can simply add a <soft_shape> element in the <body> element in the SKEL file. For example, if we want to make the left foot of the skeleton "humanoid" soft, we will modify the SKEL file as follows:

Listing 5: Joint.h

```
<skeleton name="humanoid">
...
  <body name="Left foot">
    <gravity>1</gravity>
    <transformation>0 -0.10 0 0 0 0</transformation>
    <inertia>
      <mass>0.5</mass>
      <offset>0 0 0</offset>
    </inertia>
    <soft_shape>
      <total_mass>0.5</total_mass>
      <geometry>
        <box>
          <size>0.5 0.25 0.5</size>
          <frags>3 3 3</frags>
        </box>
      </geometry>
      <kv>500.0</kv>
      <ke>0.0</ke>
      <damp>5.0</damp>
    </soft_shape>
  </body>
...
```

The material of the soft body is crucial to the behavior of the soft body. DART uses four parameters to describe the characteristics of the soft body:

1. Number of point masses (n): The resolution of the soft body surface.

2. Shape deformation coefficient (K_e): This parameter affects the deformation of the soft body in its material space.

3. Surface displacement coefficient (K_v): This parameter affects the displacement of the soft body in the frame of the BodyNode. The displacement can be caused by the deformation of the shape and the rigid transformation relative to the BodyNode frame.

4. Damping coefficient (K_d): The damping coefficient of the spring.

3.3   Joint dynamics

By default, a joint is completely passive, but DART also provides a few parameters for the user to model a joint as an angular damped spring with tunable rest pose, stiffness coefficient,

and damping coefficient.

Listing 6: Joint.h

```cpp
virtual void setRestPosition(size_t _index, double _q0);
virtual void setSpringStiffness(size_t _index, double _k);
virtual void setDampingCoefficient(size_t _index, double _coeff);
```

It is important to note that DART simulates joint dynamics using implicit integration method which uses the future state to evaluate the spring and damper forces. The implicit formulation results in solving a linear system, which is more computational costly, but it makes the system more stable and allows for larger time step in simulation.

3.4 Actuators

Karen

3.5 Constraints

JS

3.6 Accuracy and 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

References

[1] Samuel R. Buss and Jin-Su Kim. Selectively damped least squares for inverse kinematics. *Journal of Graphics Tools*, 10(3), 2005.