DART: Dynamic Animation and Robotics Toolkit

Contents

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.

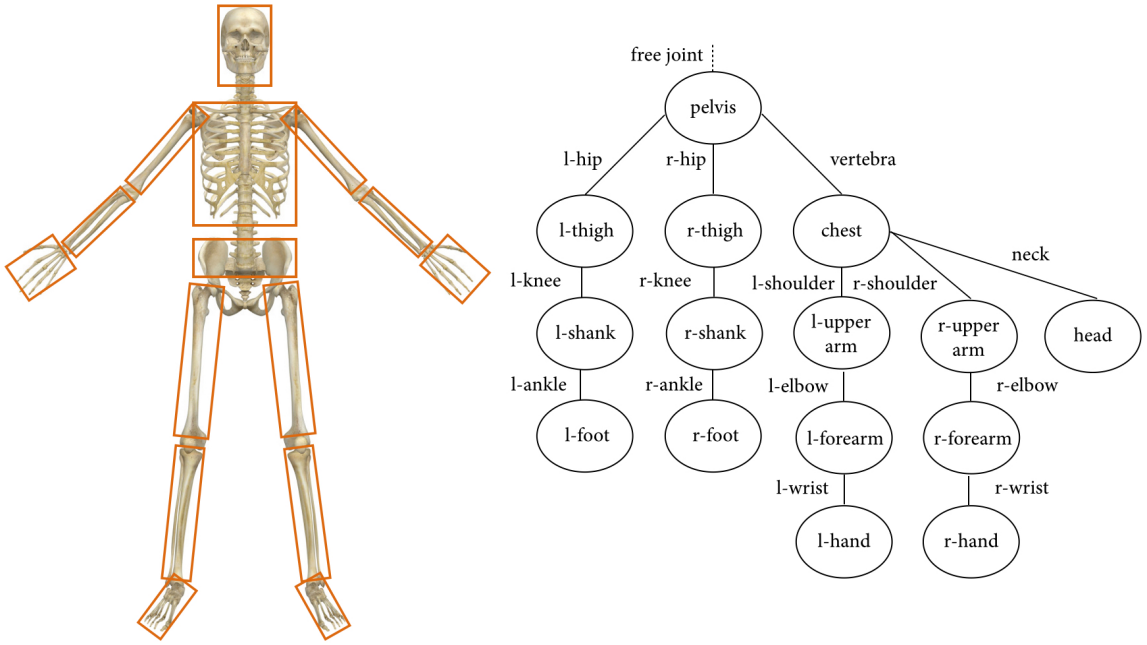


Figure 1: Tasks for each research aim.

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 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

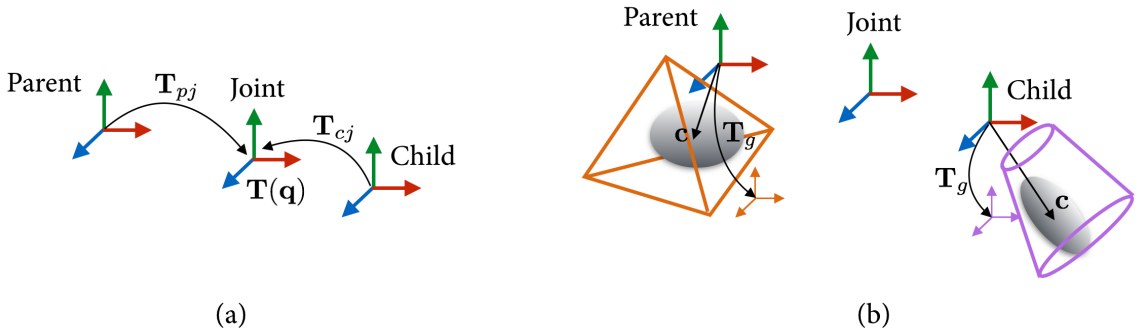


Figure 2: Tasks for each research aim.

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 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:

```
// A, B, and F are all Frame instances
B->getLinearAcceleration(A,F);
```

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

```
// 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 SE(3).

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

```
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 (although spatial vectors are expressed in the *coordinates* of the frame that they *belong to*, not the 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 Cartesian vectors, like those used in the traditional Newton-Euler equations of motion:

```
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, unexpected 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.

```
// 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

```
// 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 called **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 4.4.2. 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 Jacobian functions 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).
```

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".

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 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.

2.8.1 Gradient Method

One of the challenging aspects of Inverse Kinematics is that there are many approaches for estimating the gradient of a kinematic constraint, 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 performs gradient descent on the constraint function until convergence. 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 any 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 it. Conversely, the iterative methods mentioned above are generalizable to all kinematic structures. In KIDO, we treat an Analytical solution as a gradient by taking the difference between the solved configuration and the current configuration; that difference then represents the constraint gradient.

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.

An explanation of how to create your own custom **GradientMethod** (or **Analytical** solver) implementation is available at the end of section 2.8.5.

2.8.2 Error Method

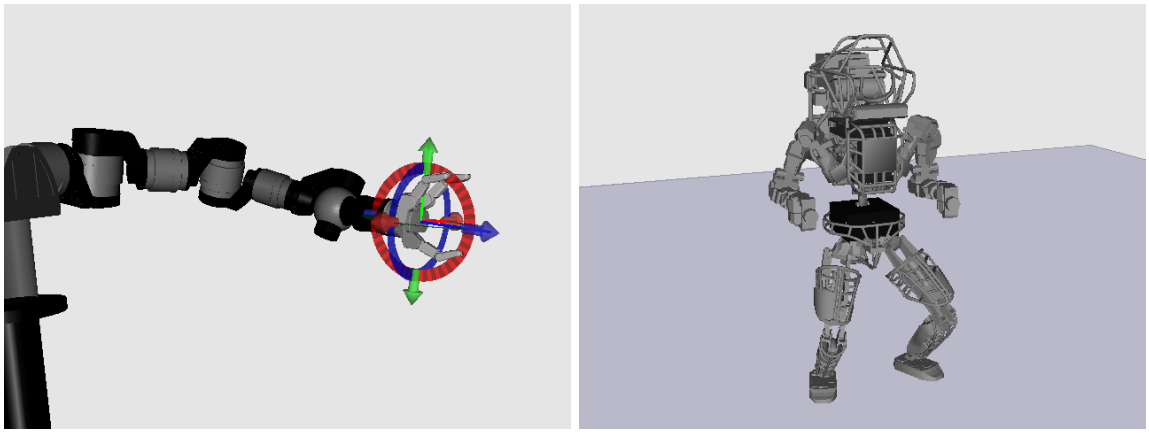
Besides variation in the methods that are used to solve inverse kinematics problems, there can also be a great deal of variation in the methods that are used to compute an error vector. In most cases, the error vector of an inverse kinematics problem is a 6×1 vector where three components refer to rotational 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. 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 yaw and to be anywhere on the surface of the floor. 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 green sphere, but it may rotate however it pleases.

Similar to the Gradient Methods mentioned above, KIDO allows the user to create and swap in any custom Error Method that they would like to. KIDO comes prepackaged with a Task Space Region[2] error method and will use that method by default. All of the constraints seen in figure 3 can be achieved by adjusting the parameters of the **IK::TaskSpaceRegion** class in the following way:

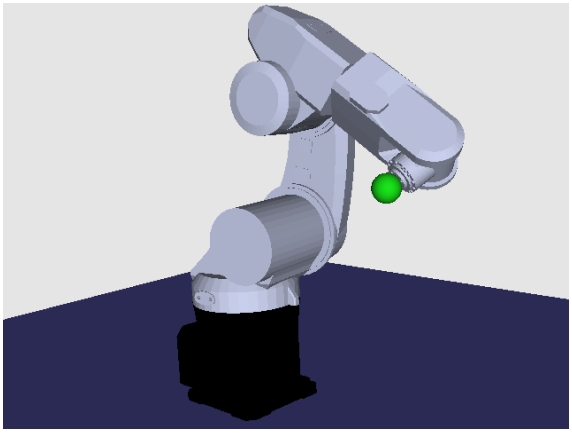
Figure 3(a)	Default Settings
Figure 3(b)	Linear Bounds: (±Inf, ±Inf, 0) Angular Bounds: (0, 0, ±Inf)
Figure 3(c)	Linear Bounds: (0, 0, 0) Angular Bounds: (±Inf, ±Inf, ±Inf) [Offset the **EndEffector** from the surface of the body]

An explanation of how to create your own custom **ErrorMethod** implementation is available at the end of section 2.8.5.



(a) Fully constrained end effector

(b) Flat feet constraint



(c) Sphere touching constraint

Figure 3: Examples of various kinematic constraints

2.8.3 Target

With KIDO, we can take advantage of the frame semantics discussed in section 2.4 to conveniently express kinematic constraints in terms of arbitrary reference frames. Furthermore, due to the automatic updating discussed later in section 4.3, the constraint will automatically adjust whenever its reference frame moves.

To achieve this, we have a target frame which acts as a reference frame for the parameters of the **ErrorMethod**. For the **TaskSpaceRegion** implementation of the **ErrorMethod**, the linear bounds will be evaluated relative to the origin of the target frame, and the angular bounds will be evaluated relative to the orientation of the target frame. In general, it is at the discretion of the implementer to decide how a custom implementation of an **ErrorMethod** utilizes the target frame, but the KIDO developers strongly urge users to implement their **ErrorMethod**s such that they use the target frame in an intuitive way. For example, if the goal of an **ErrorMethod** is to constrain the body to a parabolic surface, then the origin and orientation of the parabolic surface should be determined by the target frame.

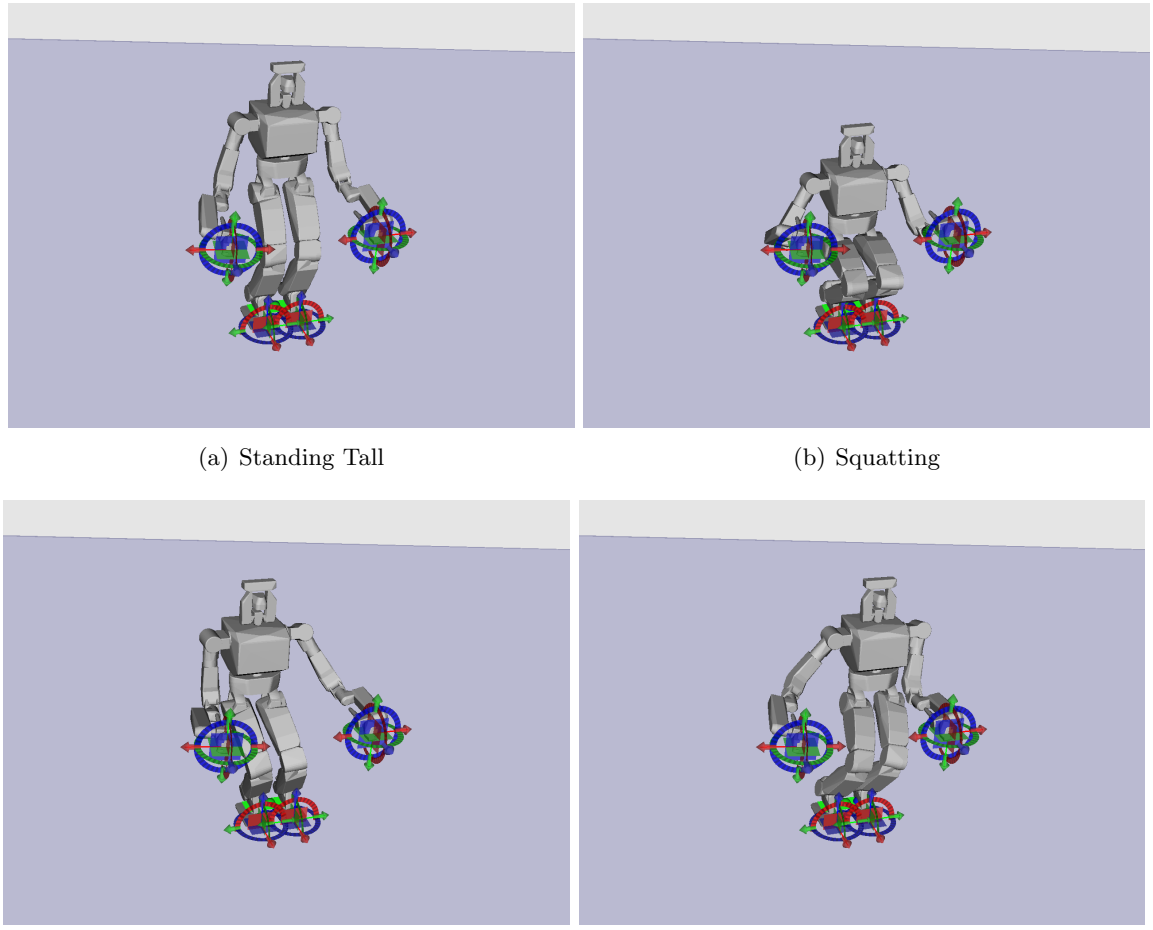
A **SimpleFrame**, described in section 2.4.2, is used as the target frame in the **InverseKinematics** framework. The target **SimpleFrame** can be switched out for a different **SimpleFrame** (or a derivative class of SimpleFrame) at any time, but it cannot be switched out for any other type of **Frame**. In order to use an arbitrary **Frame** instance as the target frame for your kinematic constraints, all you have to do is set that **Frame** to be the parent of your existing target **SimpleFrame**, and then set the relative transform of your existing target **SimpleFrame** to an identity matrix. This design makes it extremely straightforward to express your target in whatever way is most convenient for your application.

2.8.4 Objective Functions

In KIDO, we treat Inverse Kinematics as an optimization problem. The **ErrorMethod** from section 2.8.2 describes a kinematic constraint for the optimization problem. The **GradientMethod** from section 2.8.1 computes (an estimate of) the gradient of that constraint. In this section, we discuss the Objective **Function**, which describes a cost function which should be minimized while satisfying the constraints.

By default, there is no objective function. As soon as a solution within the constraints is found, the result is returned. For kinematic structures with many redundant degrees of freedom, there may be infinite solutions to a given set of kinematic constraints, and some of those solutions might be more "favorable" than others. For example, in figure 4 all four panels have the same exact kinematic constraints, but all four configurations are noticeably different. Depending on the context, this might or might not matter, but in some cases, there may be some valid configurations which are considered preferable to other valid configurations. In those cases, an Objective **Function** can be provided which will direct the solver towards a preferable configuration. The solver will seek to **minimize** the Objective Function, therefore it should be thought of as a **cost** function.

When performing inverse kinematics on a live robot, it might be crucial for all the robot's motions to satisfy the kinematic constraints, or else the task may fail. A common method for ensuring this is to project the objective through the null space of the constraint Jacobian. KIDO allows you to specify both an Objective function and a Null Space Objective function.



(a) Standing Tall

(b) Squatting

(c) Leaning to the Right

(d) Leaning to the Left

Figure 4: Examples of kinematic redundancy. The kinematic constraints for all four panels are identical.

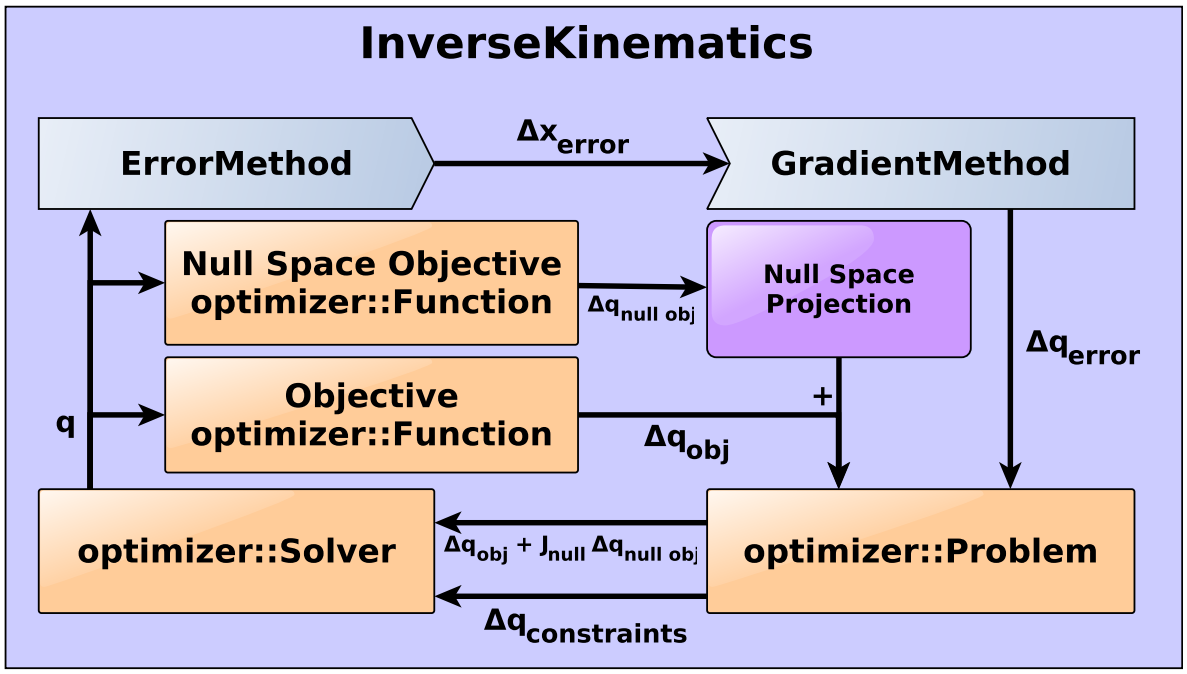


Figure 5: Overview of **InverseKinematics** Framework

The output of the Null Space Objective function will automatically be projected through the null space of the body's Jacobian, and then added to the overall Objective. If keeping the kinematic constraint satisfied is far more important than minimizing some objective, then that objective should be set as the null space objective and not set as the normal objective.

2.8.5 Basic Framework

The components that are mentioned in the previous sections are ultimately tied together by the framework of the **dynamics::InverseKinematics** class. The information flow within the framework can be seen in figure 5. **InverseKinematics** takes advantage of KIDO's **optimizer** framework, which is discussed in further detail in section 4.6.

This entire framework is automatically constructed when an **InverseKinematics** object is created. Each piece has a default instantiation:

- **ErrorMethod** is defaulted to a **TaskSpaceRegion** where each component of the linear and angular bounds are set to ±**DefaultIKTolerance**, which is a small number (currently 1×10^{-6}). The target frame is a **SimpleFrame** whose parent frame is **Frame::World()**, and whose world transform matches the current transformation of the body.
- **GradientMethod** is defaulted to the Damped Least Squares Pseudoinverse Jacobian method with a damping factor of **DefaultIKDLSCoefficient** (currently 0.05).

- **Objective** is a plain **optimizer::Function** which always returns zero. This is the same as not having an objective.
- **Null Space Objective** is also a zero function by default.
- **Problem** is automatically set up to contain the kinematic constraint and an objective which merges together the normal objective function and the projected null space objective function. As the user, you have the freedom to overwrite any of this by manually replacing the objective function or adding/remove constraints of your own. If standard problem setup is insufficient for your needs, this is where you have the freedom to completely reconfigure the problem. If the Problem is ever altered by the user, it can always be returned to its original setup by calling InverseKinematics::resetProblem().
- **Solver** defaults to KIDO's built-in **optimizer::GradientDescentSolver**. This is a simple gradient descent solver which has no third-party dependencies. This can be swapped out for a more sophisticated solver at any time. For an explanation of the different solvers that are available, see section 4.6.

Any of the default components can be swapped out with alternatives at any time. Users can also create their own custom implementations of each component by inheriting the base class for each and overriding a few pure virtual functions, listed below:

kido::dynamics::IK::ErrorMethod

```
// Override this by creating a new instance of your class type and returning it in
// a unique_ptr. This allows IK setups to be easily replicated which could be
// useful for spawning threads to parallelize your computations.
//
// Note that the constructor of your class is required to take in an
// InverseKinematics pointer as its first argument.
std::unique_ptr<ErrorMethod> clone(InverseKinematics* newIK) const;

// Override this by computing a 6D error vector. The first three components of the
// vector should represent orientation error (usually as Euler Angles or a logmap)
// and the last three components should represent translational error. NOTE THAT
// THIS IS THE REVERSE OF SOME CONVENTIONS. We do this in order to match
// Featherstone's spatial vector convention, which is used in KIDO for spatial
// velocity and spatial acceleration vectors.
Eigen::Vector6d computeError();

// We allow some flexibility in the exact meaning of the error vector that gets
// computed by computeError(), because iterative methods are robust to various
// representations of rotational error. However, Analytical methods typically
// require an exact desired transform. This function should construct an exact
// desired transform based on the current transform and the error (as computed
// by this error method) of the current transform.
Eigen::Isometry3d computeDesiredTransform(
   const Eigen::Isometry3d& currentTransform,
   const Eigen::Vector6d& error);
```

kido::dynamics::IK::GradientMethod

```
// Override this by creating a new instance of your class type and returning it in
// a unique_ptr. This allows IK setups to be easily replicated which could be
// useful for spawning threads to parallelize your computations.
//
// Note that the constructor of your class is required to take in an
// InverseKinematics pointer as its first argument.
std::unique_ptr<GradientMethod> clone(InverseKinematics* newIK) const;

// This function should take in a 6D error vector (which will have been computed
// by the current ErrorMethod) and then modify the contents of the gradient
// variable, which is passed in as an output parameter.
void computeGradient(
   const Eigen::Vector6d& error,
   Eigen::VectorXd& gradient);
```

kido::dynamics::IK::Analytical is a specialized version of **GradientMethod** which allows users to plug in Analytical (a.k.a. Closed-Form) inverse kinematics solvers such that they are compatible with KIDO's inverse kinematics interface and hierarchical inverse kinematics solver (discussed in section 2.8.6). Note that computeGradient should not be implemented for an Analytical solver; that gets done automatically.

```
// Override this by creating a new instance of your class type and returning it in
// a unique_ptr. This allows IK setups to be easily replicated which could be
// useful for spawning threads to parallelize your computations.
//
// Note that the constructor of your class is required to take in an
// InverseKinematics pointer as its first argument.
std::unique_ptr<GradientMethod> clone(InverseKinematics* newIK) const;

// This function should fill in the mSolutions protected variable and then return
// it. It would also be a good idea to call checkSolutionJointLimits() prior to
// returning, unless your function implementation checks the joint limits on its
// own.
//
// The Solution struct is a container which holds an Eigen::VectorXd for the
// configuration that's been generated, and a flag called mValidity which uses a
// bitwise map to indicate whether the configuration violates joint limits or is
// out of reach from the target transform. The expectation is that your
// implementation will set mValidity to OUT_OF_REACH if the configuration does
// not reach the goal, and then you should call checkSolutionJointLimits() just
// before returning, which will inject LIMIT_VIOLATED into any Solutions whose
// joint limits are invalid.
const std::vector<Solution>& computeSolutions(
   const Eigen::Isometry3d& desiredTransform);

// An Analyical solver might only be able to solved for a subset of the degrees
// of freedom that are being utilized by an inverse kinematics solver. Because
// of this, you need to provide a function that indicates which degrees of
// freedom your Analytical method can provide solutions for.
//
```

```
// To determine how the remaining degrees of freedom get used, the Analytical
// class provides a setExtraDofUtilization(ExtraDofUtilization_t) function.
const std::vector<size_t>& getDofs() const;
```

It is also worth noting that the **Analytical** class has a **QualityComparison** function which is used to compare the qualities of various solutions. The default **QualityComparison** function will simply pick the configuration whose largest component of difference from the current configuration is the smallest. This is essentially the same as picking the configuration whose highest joint velocity would be the smallest out of all the options if the robot were to move to it from the current configuration. The chart below provides an example:

Rank	**Largest Component**	Norm	Δq0	Δq1	Δq2	Δq3	Δq4
1	3.1	4.58	1.3	1.2	2.8	0.6	**3.1**
2	4.0	4.01	0.1	0.0	0.2	**4.0**	0.0
3	5.3	7.59	3.8	**5.3**	0.5	2.3	3.1

The right side of the table shows the component-wise differences between the current configuration and each solution that has been generated. The left side shows the rank of each solution, along with the largest component of difference and the norm of its difference from the current configuration. Notice that the solutions are not sorted based on the norm; they are sorted in ascending order of the largest component in the difference vector. This sorting approach tends to reduce the likelihood of an "elbow flip" solution being selected.

kido::dynamics::IK::Function & kido::optimizer::Function are classes which should both be inherited when creating a class that represents an Objective Function. Technically, any class which inherits **optimizer::Function** can be used as an Objective Function, but it can only be cloned if it also inherits **dynamics::IK::Function**. Without that, the function will be replaced by a default zero-function in any clones that get produced.

kido::optimizer::Problem & kido::optimizer::Solver are standard tools from the kido-optimizer library, which is discussed in section 4.6.

2.8.6 Hierarchical Framework

The basic **InverseKinematics** framework is only designed to solve an inverse kinematics problem for a single body or end effector. This is typically sufficient for a simple robot arm, such as an industrial manipulator. More advanced robots or characters, such as humanoids, might require a more sophisticated type of inverse kinematics solver. In particular, they might need a solver that can solve for kinematic constraints on multiple bodies or end effectors simultaneously. To facilitate this, KIDO offers the **HierarchicalIK** class.

Figure 6 shows an example of how a **HierarchicalIK** object might be set up. Suppose we have a humanoid robot with two arms, two legs, and a camera on its head. We want the robot to grasp an object with both hands while standing in a balanced configuration, and we want it to look along a specific ray with its camera. The stance constraints on the feet are easily solved within the null space of the grasping constraints, so we place the IK solvers for the grasping constraints in level 0 (the highest priority level), and the stance constraints for

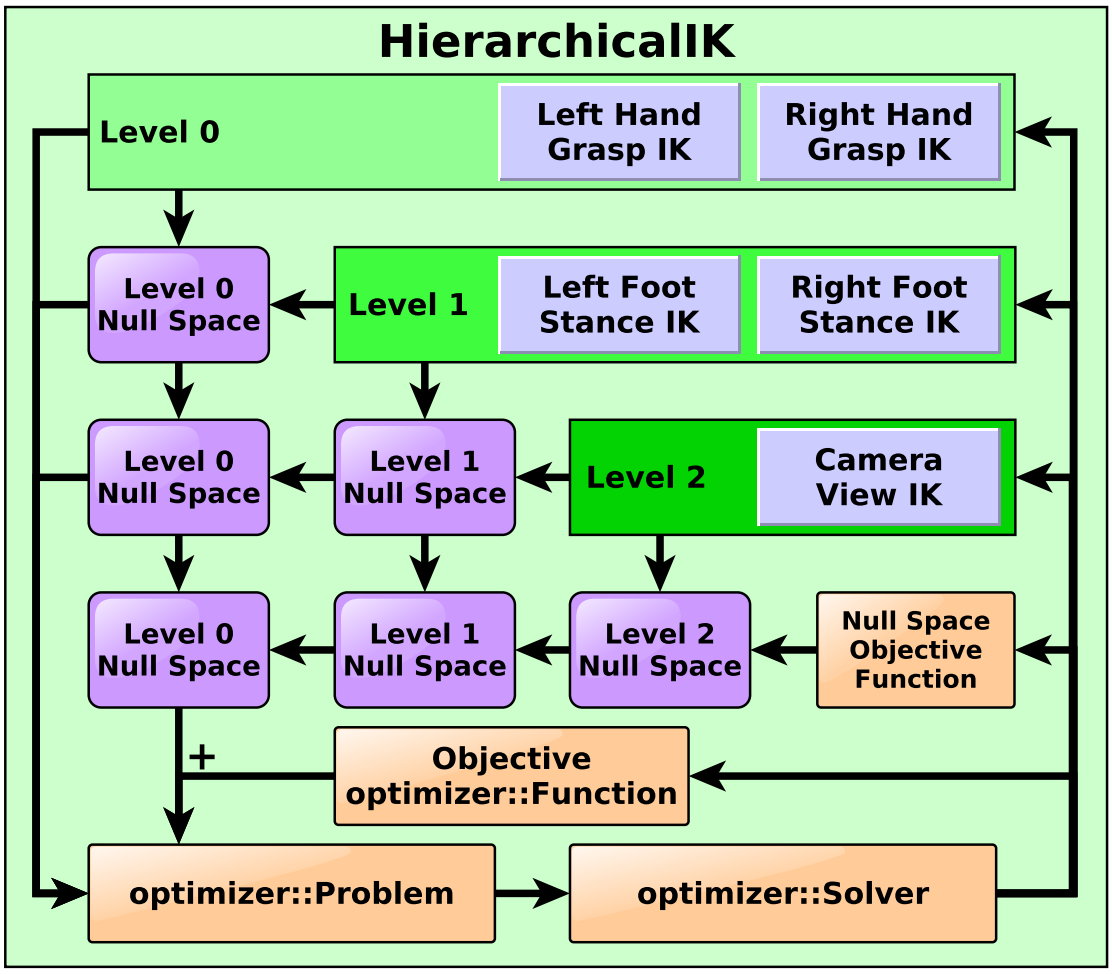


Figure 6: Overview of **HierarchicalIK** Framework

the feet go below them in level 1. The camera view constraint is less important than any of the other four constraints, so we put that below all of them, in level 2.

The **HierarchicalIK** class can handle arbitrarily many hierarchy levels. All the inverse kinematics modules that are placed on the same level will be given the same amount of priority; their constraint functions will essentially be superimposed together. This superposition of constraint functions will then be projected through all the null spaces of any higher priority (lower level value) inverse kinematics modules. The level of an **InverseKinematics** object can be set using `InverseKinematics::setHierarchyLevel(size_t)`.

Even though a Hierarchical object uses references to **InverseKinematics** objects, only the **ErrorMethod** and **GradientMethod** components of each **InverseKinematics** object gets used by a **HierarchicalIK**. This means that passing in an Objective or a Null Space Objective to the individual **InverseKinematics** objects will have no effect on a **HierarchicalIK**. Instead, the **HierarchicalIK** object has its own Objective and Null Space Objective. It also has its own **Problem** and **Solver**, so modifications to the **Problem**s and **Solver**s of the individual **InverseKinematics** objects also has no effect on a **HierarchicalIK** object. However, configuring an **InverseKinematics** module to use a specific **Analytical** (or any other **GradientMethod**) type *will* have an impact on any **HierarchicalIK** object that it gets passed to. This allows **HierarchicalIK**s to string together multiple **Analytical** IK solutions simultaneously. For example, if a humanoid has an analytical IK solution for each limb, then the **HierarchicalIK** may be able to solve an arbitrary whole body inverse kinematics problem with a single iteration. Analytical IK solutions can also be seamlessly integrated with iterative solvers within a single **HierarchicalIK**. The **HierarchicalIK** will also respect whatever **ErrorMethod** is given to each **InverseKinematics** module.

The **HierarchicalIK** class is a pure virtual class, because it does not specify any way to decide which **InverseKinematics** modules it will use. Instead, there are two derived classes which offer different behavior:

CompositeIK is a derivative of **HierarchicalIK** which allows the user to freely select which InverseKinematics modules it should utilize.

WholeBodyIK is a derivative of **HierarchicalIK** which will automatically detect all **BodyNode** and **EndEffector** instances in a given **Skeleton** and use any **InverseKinematics** modules attached to them.

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 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 7.

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

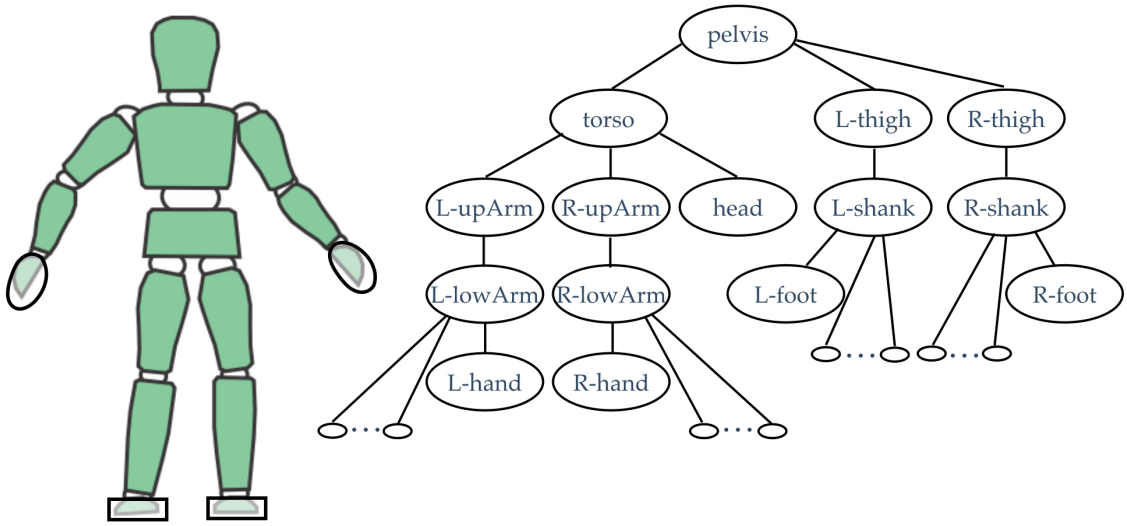


Figure 7: The hands and the feet of the humanoid are composed of a rigid body with a layer of deformable surface. Both rigid and soft bodies are represented in generalized coordinates.

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

```
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 updating

Most kinematics libraries have a strict workflow that must be followed in order to produce correct results in a timely manner. Typically this workflow requires that all joint positions must be set and then all transforms must be computed before the transformation of a specific frame may be queried. This is a wasteful and inefficient approach for many applications.

For example, a dexterous robot manipulator may have seven links from the base to the end effector, but then it may have three fingers which each have three links. While iteratively performing inverse kinematics, it will be necessary to repeatedly compute the seven matrix transformations that go from the base to the end effector, but the transforms of the nine finger links are unnecessary. A naive approach to updating kinematics will compute all 16 transforms, even though only 7 are necessary. This means that over 56% of the computational effort of the forward kinematics is wasted. When iteratively solving an inverse kinematics problem, the bulk of time is spent computing forward kinematics. This means that intelligently updating the forward kinematics could yield a nearly 50% reduction in computation time for this example. Similar (and more extreme) examples could be imagined for robots with multiple limbs.

Strict workflow requirements can also lead to bugs in code. If a human programmer is not familiar with the correct workflow, does not fully understand the necessary order of operations, or simply makes a programming error, then there may be bugs in a complex controller which would produce incorrect results. Incorrect results on a dexterous manipulator could cause a task to fail and damage the robot. Incorrect results on a high-powered robot could severely damage the robot and even endanger human lives.

4.3.1 Design

KIDO resolves those concerns using lazy evaluation and automatic updating. Most kinematic and dynamic quantities in KIDO are automatically updated and lazily evaluated (see section 4.3.2 for the exceptions). *Automatic updating* means that you never need to explicitly call any kind of "update" function in order to get the correct output based on your most recent input. *Lazy evaluation* means that expensive computations are held off until they are absolutely required by the user. Combined, these two add a decisive level of code safety to applications that use KIDO, and they cut out a considerable degree of potential waste, both while making the library easier to use. This is accomplished with the following setup:

- Mutator and accessor methods (a.k.a. setters and getters) intercept all interactions with the internal states and properties of kinematic objects.

- Dirty flags are used to keep track of which quantities are out of date, and the dirtiness gets propagated to all dependencies.
- Accessor methods will check any relevant dirty flags and perform computations as necessary.
- Mutator methods will trigger dirty flags.

A visual example of this scheme applied to forward kinematics can be seen in figure 8. Note that the propagation of dirty flags throughout the kinematic structure will short-circuit any time a kinematic object already knows that it is dirty. This prevents the bookkeeping from having $O(N^2)$ complexity when all N joints in the kinematic structure undergo changes.

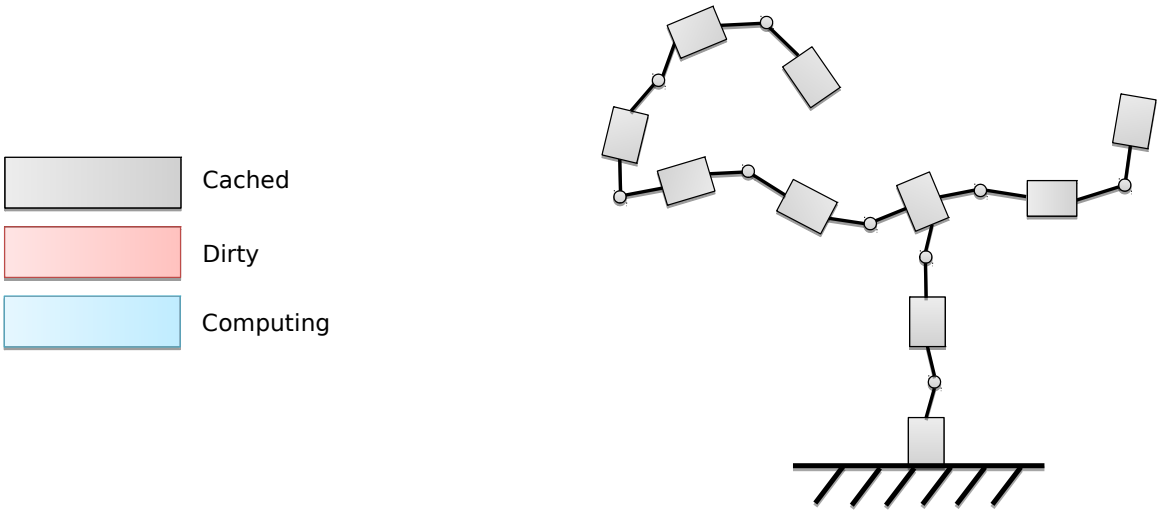
To facilitate this lazy evaluation, KIDO abides by logical const-correctness (rather than physical const-correctness). The values that are lazily evaluated are stored as mutable members of their class, allowing the `getX()` functions to be const member functions. However, because of this mutability, the const-qualifier on member functions does **not** guarantee thread-safety. Currently, the only guaranteed way to ensure thread-safety with KIDO is to spawn identical copies of the kinematic structures for each thread. This is done easily with the various `clone()` functions throughout KIDO. To spawn an identical copy of a single Skeleton, simply call `Skeleton::clone()` on it. To spawn an identical copy of a whole environment, you can use the `simulation::World::clone()` function.

In some cases, it might be undesirable to delay the computations until the time that they are requested. For example, if a real-time controller has strict scheduling requirements, there might be a specific time window which is best suited for performing heavy computations. In that case, KIDO provides a `Skeleton::computeForwardKinematics()` function which will compute all the forward kinematics within a Skeleton instead of waiting for it to be automatically updated. There are also `Skeleton::computeForwardDynamics()` and `Skeleton::computeInverseDynamics()` functions which will compute all the various dynamics parameters instead of waiting to evaluate them as needed.

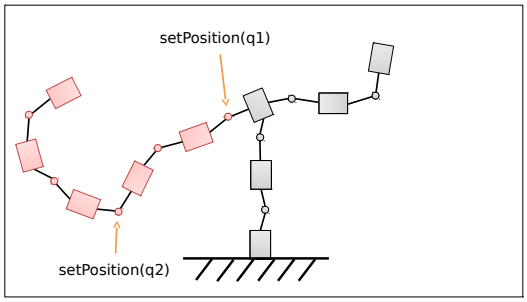
4.3.2 Exceptions: Forward and Inverse Dynamics Updating

Even though forward kinematics is updated automatically, neither forward nor inverse dynamics have this quality. Individual dynamics parameters—such as the mass matrix, Coriolis forces, and articulated inertia—are updated automatically, but KIDO will **not** automatically compute the body accelerations due to the current joint torques or vice versa. This is due to the ambiguous nature of how the user might want to utilize the library: Do they want to simulate forward to see a result or compute the inverse dynamics to inform a controller? In order to compute one or the other automatically, the library would need to make an assumption about which one the user is interested in, but KIDO is meant to be suitable for both. Instead, KIDO requires the user to specifically request one or the other each time they want the results:

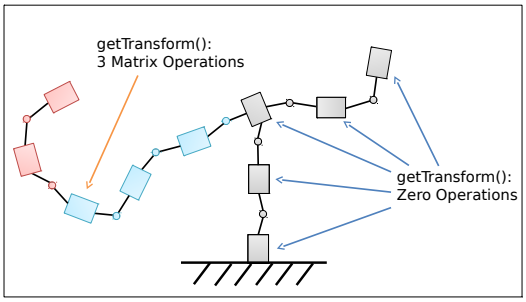
- **`Skeleton::computeForwardDynamics()`** should be used to compute the accelerations of all the BodyNodes given the current generalized forces (or more generally, the current commands) of their joints.



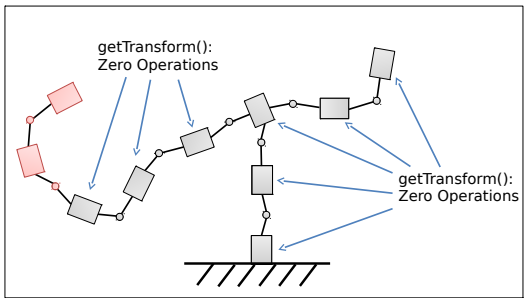
(a) Everything is cached



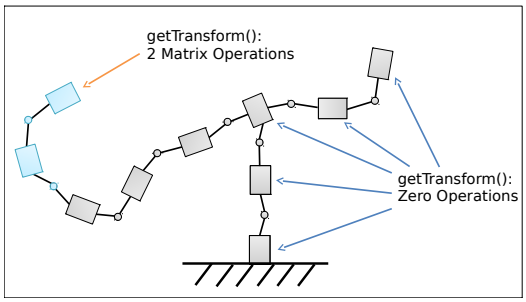
(b) Some joint values have changed



(c) A dirty transform is queried



(d) Cached transforms can be used freely



(e) Another dirty transform is queried

Figure 8: Illustration of lazy evaluation for forward kinematics

- **`Skeleton::computeInverseDynamics()`** should be used to compute the generalized forces needed by the joints in order to achieve the current accelerations of the BodyNodes.

The results of each call will show up in the joint states of the Skeleton. The results of `computeForwardDynamics()` will show up in `Joint::getAccelerations()` and then by extension `BodyNode::getSpatialAcceleration()`, `getLinearAcceleration()`, and `getAngularAcceleration()`. The results of `computeInverseDynamics()` will show up in `Joint::getForces()` and by extension `Skeleton::getForces()`.

In both cases, the functions make use of kinematics and dynamics parameters which are cached and lazily evaluated, so making redundant calls to these functions is not prohibitively expensive, although it is still not completely free of charge.

4.4 Extensible data structures

There are countless applications for kinematics and dynamics software, and the developers of KIDO do not anticipate that we could possibly account for all use cases. Instead of limiting the potential applications of the library with a closed-off design, we have made the kinematics and dynamics structures extensible by introducing the concepts of **Addon**s and **Node**s.

Addons and **Node**s have some practical and semantic differences, but their overarching motivation is the same: They provide a way to embed arbitrary data and functionality into kinematic and dynamic structures. For example, if you want to have a custom sensor type that can be attached to a body on your robot, you could create a custom **Node** and attach instances of your *SensorNode* on whichever bodies need it.

Any concept can be encapsulated into either an **Addon** or a **Node** as long as its contents can be decomposed into a **State** structure and/or a **Properties** structure (or neither if the concept contains no information at all). To fully understand whether an Addon or a Node should be used for your purposes, the following subsections will explain the difference, but the differences can be boiled down to these:

- A **BodyNode** can only contain a single instance of any particular type of **Addon**, but it can contain arbitrarily many of any particular type of **Node**.
- **Node**s can only be attached to **BodyNode**s, but **Addon**s can be attached to any class that inherits **AddonManager** (e.g. **Joint**, **Skeleton**, **EndEffector**).

4.4.1 Addons

There may be times where you have developed a novel algorithm that requires extra contextual information that might not be natively available in KIDO. For example, you might have computed an occupancy grid for some geometry, but KIDO does not natively support occupancy grids. Nevertheless, you want the information to be embedded in the geometric object; you want it to be saved as part of the object's **Properties**, and you want it to get copied over whenever the object is cloned.

You could achieve this by defining an *OccupancyGrid* class that inherits the **Addon** class. Then you could embed an instance of an *OccupancyGrid* class into the **ShapeNode** instance that holds the geometric data for which the occupancy grid was calculated:

```
shapeNode->create<OccupancyGrid>(gridInfo);
```

You can then retrieve it with:

```
OccupancyGrid* grid = shapeNode->get<OccupancyGrid>();
```

If the object named `shapeNode` has never created or been given an `OccupancyGrid` object, this function will return a `nullptr` by default. It is always a good idea to check whether the **Addon** pointer that gets returned is a `nullptr`, because KIDO does not throw any exceptions or assertions when you request an **Addon** that is not present in the object.

There are many native Addons in KIDO:

- **Support** which is an Addon for **EndEffector**
- **RevoluteJoint::Addon** which contains unique properties for **RevoluteJoint**
- **PrismaticJoint::Addon** which contains unique properties for **PrismaticJoint**
- (All the other six Joint-type Addons)
- **TODO(MXG): Mention the Addons for ShapeFrame once they are finished**

In some cases, such as the Joint-type Addons, the Addon is crucial for the object to function correctly. In other cases, such as the **Support** class, the Addon simply provides extra features. It might seem counter-intuitive to put critical property information into an Addon if the class cannot function without it, but there is a strong motivation for this: When the **Skeleton::ExtendedProperties** is retrieved, it pulls in all the **Properties** of all the **BodyNode**s, **Node**s, and **Joint**s in the **Skeleton**, as well as the **Properties** of all the **Addon**s attached to those **BodyNode**s, **Node**s, and **Joint**s. This means that there is no special handling required in order to save all the unique property information for each different Joint-type; all their unique properties are stored in **Addon**s, and the Properties of those **Addon**s will get sucked into the **Skeleton::ExtendedProperties** automatically.

By creating your own custom Addons and attaching them to objects in Skeletons, you can extend the scope of a Skeleton's **State** and **ExtendedProperties**. The State of your Addon will be embedded into the **Skeleton::State**, and the Properties of your Addon will be embedded into the **Skeleton::ExtendedProperties**. Creating your own custom Addon involves—at a minimum—inheriting the class **kido::common::Addon** and then overriding the **Addon::cloneAddon** function. Additionally, if your Addon has a **State**, then you should also override the functions **Addon::setAddonState(const State&)** and **Addon::getAddonState()**. Neglecting to override those functions will cause **Skeleton::State** to treat your Addon as though it has no State. Similarly, if your Addon has **Properties**, then you should override the functions **Addon::setAddonProperties(const Properties&)** and **Addon::getAddonProperties()**. Neglecting to override these will cause

Skeleton::ExtendedProperties to treat your Addon as though it has no Properties. Also note that the constructor of your Addon is expected to take an **AddonManager*** as its first argument; it can have any number of arbitrary argument types after that.

The most common use case of an Addon is to embed plain old data (POD) into the Properties or State of a Skeleton. To accommodate this, we offer two templated classes which take care of constructing an Addon with **State** and/or **Properties** that can live happily within a Skeleton:

- **AddonWithProtectedPropertiesInSkeleton** is suitable for Addons which only has a POD **Properties** and no **State**.
- **AddonWithProtectedStateAndPropertiesInSkeleton** is suitable for Addons which have both a POD **State** and a POD **Properties**.

The template arguments of these classes work as follows:

- The class type that inherits this class. These are CRTP classes.
- (Only for AddonWithProtectedStateAndPropertiesInSkeleton) The POD **State** type
- The POD **Properties** type
- The type of AddonManager that this class is intended for. Pass in `AddonManager` if the type of AddonManager is irrelevant.
- (Only for AddonWithProtectedStateAndPropertiesInSkeleton) The function that should be called whenever the State of the Addon is updated.
- The function that should be called whenever the Properties of the Addon are updated

The update functions that are used for the final template argument(s) must accept a single argument, which will be a pointer to the full Addon class. CRTP (Curiously Recurring Template Pattern) is used to make this possible.

The **State** and **Properties** are protected for this type of Addon to ensure that all changes to the **Properties** are registered by the **Skeleton**. Otherwise, **Skeleton**s would not be able to correctly keep track of their "version" number **TODO(MXG): We should probably have a section about Skeleton version and cite it here**. This can be inconvenient if the POD **State** or **Properties** structures have many fields that need to be accessed or changed, because then accessor and mutator functions are needed for each one. This can entail a troublesome amount of typing, and worse yet: When writing mutator functions, you must be sure to call the correct update functions, and increment the Skeleton's version number if a property is being changed.

To avoid the hassle and dangers of writing many accessor and mutator functions, KIDO provides a few macros which can take of this:

KIDO_DYNAMICS_SET_GET_ADDON_PROPERTY will create a setter and getter function for a variable in the Properties POD. The first argument for this macro is the variable type, and the second argument is the variable name. There is also:

- **KIDO_DYNAMICS_SET_ADDON_PROPERTY** which will only create the set
- **KIDO_DYNAMICS_GET_ADDON_PROPERTY** which will only create the get

KIDO_DYNAMICS_SET_GET_ADDON_PROPERTY_ARRAY can be used to access and mutate the individual components of an array or vector field within the POD **Properties** structure. For variable names whose plural form is "irregular" (i.e. is not the same as adding 's' to the end), there is also

- **KIDO_DYNAMICS_IRREGULAR_SET_GET_ADDON_PROPERTY_ARRAY** which allows you to specify the singular and plural forms of the name.

There are more macros, similar to these, which address a variety of use cases. Although C-macros can be strange and clumsy, they can save a significant amount of typing and prevent errors that may result from forgetfulness.

4.4.2 Nodes

Nodes are conceptually similar to **Addon**s, so this section will focus on what makes **Node**s different from **Addon**s. Reading the previous section (4.4.1) on **Addon**s before this section is recommended.

Only one instance of any particular type of **Addon** can be embedded into a single AddonManager. **Node**s differ from this in that any number of a **Node** type can be attached to a **BodyNode**. This brings up another difference: A **Node** can only be attached to a **BodyNode**, whereas an **Addon** can be attached to anything that inherits the AddonManager class. Each **Node** also requires a name, which is not necessary for an **Addon**. Moreover, each **Node** of any given **Node** type within a **Skeleton** must have a unique name. **Node**s can be accessed by name and type through their **Skeleton**, for example:

```
// Assume skeleton is a SkeletonPtr type:
EndEffector* hand = skeleton->getNode<EndEffector>("left_hand");
```

Even though **Node**s are accessible through the **Skeleton**, they are not owned by the Skeleton; they are owned by the **BodyNode** to which they are attached. If the **BodyNode** is moved to a new **Skeleton**, all **Node**s that are attached to it will follow. However, the indexing of the **Node**s may change after being moved into a new Skeleton, and the names of the **Node**s may change (by appending a number to the string) if there is a name collision with any **Node**s of the same type that were already in the **Skeleton**. If a name change occurs, a message will be printed to `stdout`, and a **NameChangedSignal** will emanate from the **Node** whose name needed to be changed. When a whole **Skeleton** is cloned into a new **Skeleton**, the names of its **Node**s are guaranteed to remain the same, and the indexing of the **Node**s is guaranteed to remain the same.

The defining feature of a **Node** is that it is something which belongs to a **BodyNode** in some sense. **BodyNode** is a special case of the **Node** class; it is a **Node** which belongs to itself. Because of this, it has its own specialized management semantics (described in section 2.7). Most other **Node** types will simply inherit the **AccessoryNode** class which allows them to be detached from their **BodyNode** by calling the **remove()** function on them. When a **Node** is removed from its **BodyNode**, it can no longer be accessed by calling any **getNode** function from the **BodyNode** or **Skeleton** that it had been attached to. If no strong references (such a **NodePtr**) remain for a removed **Node**, then the **Node** instance will be deleted. If a strong reference does exist, then the **Node** will continue to function as normal, except any features which would normally access it through a **getNode** call will no longer work. The Node will also have **INVALID_INDEX** as its index within the **BodyNode** and within the **Skeleton** (note that this does not mean you can access the Node using **INVALID_INDEX**). A removed **AccessoryNode** can be reattached—if and only if a strong reference to it remains—by calling the **reattach()** function on it.

When creating a custom **Node** type, you will most likely want to inherit **AccessoryNode**, because it will provide the interface for removing and reattaching that was described in the previous paragraph. If your **Node** inherits the **Frame** class in some way, you might also want to have it inherit the **TemplatedJacobianNode** class in order to provide it with an interface for computing Jacobians. Note that the Node class does not allow multiple-inheritance (just like the Addon class), so if you have an implementation of some features which you would like to share among a number of different **Node** types, we advise you to use CRTP-style mix-in classes, like the **AccessoryNode** class.

Just like the **Addon** class, the **Node** class has a concept of **State**, **Properties**, and cloning. It also has a concept of naming. The following functions **need** to be overridden by inheriting classes:

```
// Create a new Node which is identical to this one, except it will be attached to
// the BodyNode named bn.
Node* cloneNode(BodyNode* bn) const;

// Change the name of this Node. When implementing this function, you must be sure
// to call the protected function Node::registerNameChange(const std::string&),
// otherwise the Skeleton would be left unaware of the name change. Note that if
// the new name collides with an already existing name, then
// Node::registerNameChange(~) will let you know this by returning the modified
// version of the name. Otherwise, it will send back the name that you passed in.
//
// You should return the final name, as given back by Node::registerNameChange(~),
// to convey to the user what the final name will be.
const std::string& setName(const std::string& newName);

// Different Nodes might store their names in different ways, so this function is
// pure virtual in order to give the user the freedom to decide how their Node's
// name is stored.
const std::string& getName() const;
```

Additionally, if the Node has State data, the following functions should be overridden:

```
// Set the State of the Node given a reference to a const Node::State object. You
```

```
// will be required to downcast the reference to the correct type for your Node.
// Even though KIDO guarantees that the incoming reference will always be
// appropriate type for your Node type, this guarantee does not apply to any time
// that a user decides to invoke this function. Therefore, a static_cast is
// recommended for downcasting if performance is the most important factor, but
// only a dynamic_cast can ensure that the function is safe from malicious or
// careless users.
void setNodeState(const Node::State& otherState);

// Return a pointer containing a newly instantiated version of the Node::State
// for your Node type.
std::unique_ptr<Node::State> getNodeState() const;
```

Similarly, if the Node has Properties data, the following functions should be overridden:

```
// Similar to setNodeState (see above), except for Properties data
void setNodeProperties(const Node::Properties& otherProperties);

// Similar to getNodeState (see above), except for Properties data
std::unique_ptr<Node::Properties> getNodeProperties() const;
```

In some cases, the **State** or **Properties** of a particular **Node** type might contain a significant amount of data which would be slow or wasteful to frequently allocate and deallocate. For cases like this, the following functions will pass a **Node::State** or **Node::Properties** pointer which can be modified directly to match the current **State** or **Properties** of the Node. This avoids the need to allocate and deallocate the **State** or **Properties** from the heap:

```
// You can copy the state information directly instead of allocating a copy in
// new memory.
void copyNodeStateTo(std::unique_ptr<Node::State>& outputState) const;

// You can copy the property information directly instead of allocating a copy in
// new memory.
void copyNodePropertiesTo(
    std::unique_ptr<Node::Properties>& outputProperties) const;
```

This is all that is needed to create a fully functional Node that is compatible with KIDO and can be attached to a BodyNode.

4.5 History recorder

TODO(MXG): It doesn't look like we'll have this in time for the next major release, although it will probably be available for the next minor release. Will it be possible to update this tech report as we add features and make changes?

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.

[2] Dmitry Berenson, Siddhartha Srinivasa, and James Kuffner. Task space regions: A framework for pose-constrained manipulation planning. *International Journal of Robotics Research (IJRR)*, 30(12):1435 – 1460, October 2011.