DART: Dynamic Animation and Robotics Toolkit

Release 7.0.0-alpha20230101

The DART development contributors

Apr 23, 2024

HOME

1 Introduction 1

2 Updates 3

3 Social Media 5

4 Vision for the next version of DART 7

5 Citation 9

CHAPTER

ONE

INTRODUCTION

DART (Dynamic Animation and Robotics Toolkit) is a collaborative, cross-platform, open-source library developed by the Graphics Lab and Humanoid Robotics Lab at the Georgia Institute of Technology, with ongoing contributions from the Personal Robotics Lab at the University of Washington and the Open Source Robotics Foundation. It provides data structures and algorithms for kinematic and dynamic applications in robotics and computer animation. DART stands out due to its accuracy and stability, which are achieved through the use of generalized coordinates to represent articulated rigid body systems and the application of Featherstone' s Articulated Body Algorithm to compute motion dynamics.

CHAPTER TWO

UPDATES

- 2022-12-31: DART version 6.13.0 released.

CHAPTER

THREE

SOCIAL MEDIA

Stay updated with the latest news and developments about DART by following us on Twitter and subscribing to our YouTube channel.

CHAPTER

FOUR

VISION FOR THE NEXT VERSION OF DART

- Elevate the Python binding to a first-class component, ensuring full support and equivalent functionality to the C++ APIs, rather than remaining in an experimental stage.
- Modularize the library so that users can select specific components to use with minimal required dependencies, rather than having to use the entire library, including unnecessary parts.
- Utilize hardware accelerations, such as SIMD, multi-core CPUs, and GPUs, whenever available and enabled by the user, to maximize overall performance.
- Support both single and double precision, with options to compile the library for required scalar types or leave the template code uncompiled.
- Minimize dependencies to make the library usable without bringing in all transitive dependencies.
- Modernize implementation and public APIs to enable users to work with more intuitive and user-friendly APIs.
- Provide various resources, such as a quick start guide, examples, and tutorials, to lower the initial learning curve for users.

CHAPTER

FIVE

CITATION

If you use DART in an academic publication, please consider citing this JOSS Paper [BibTeX]

```
@article{Lee2018,
  doi = {10.21105/joss.00500},
  url = {https://doi.org/10.21105/joss.00500},
  year  = {2018},
  month = {Feb},
  publisher = {The Open Journal},
  volume = {3},
  number = {22},
  pages = {500},
  author = {Jeongseok Lee and Michael X. Grey and Sehoon Ha and Tobias Kunz and Sumit␣
↪Jain and Yuting Ye and Siddhartha S. Srinivasa and Mike Stilman and C. Karen Liu},
  title = {{DART}: Dynamic Animation and Robotics Toolkit},
  journal = {The Journal of Open Source Software}
}
```

5.1 Overview

DART (Dynamic Animation and Robotics Toolkit) is a collaborative, cross-platform, open-source library developed by the Graphics Lab and Humanoid Robotics Lab at the Georgia Institute of Technology, with ongoing contributions from the Personal Robotics Lab at the University of Washington and the Open Source Robotics Foundation. It provides data structures and algorithms for kinematic and dynamic applications in robotics and computer animation. DART stands out due to its accuracy and stability, which are achieved through the use of generalized coordinates to represent articulated rigid body systems and the application of Featherstone' s Articulated Body Algorithm to compute motion dynamics.

For developers, DART offers full access to internal kinematic and dynamic quantities, such as the mass matrix, Coriolis and centrifugal forces, transformation matrices, and their derivatives, unlike many popular physics engines that treat the simulator as a black box. It also provides efficient computation of Jacobian matrices for arbitrary body points and coordinate frames. The frame semantics of DART allow users to define and use arbitrary reference frames (both inertial and non-inertial) to specify or request data.

DART is suitable for real-time controllers due to its lazy evaluation, which automatically updates forward kinematics and dynamics values to ensure code safety. It also allows for the extension of the API to embed user-provided classes into DART data structures. Contacts and collisions are handled using an implicit time-stepping, velocity-based linear complementarity problem (LCP) to guarantee non-penetration, directional friction, and approximated Coulomb friction cone conditions.

In summary, DART has applications in robotics and computer animation as it features a multibody dynamic simulator and various kinematic tools for control and motion planning.

5.1.1 Features

General

- Open-source C++ library licensed under the BSD license.
- Supports multiple platforms including Ubuntu, Archlinux, FreeBSD, macOS, and Windows.
- Fully integrated with Gazebo.
- Supports models in URDF and SDF formats.
- Provides default integration methods (semi-implicit Euler and RK4) and an extensible API for other numerical integration methods.
- Supports lazy evaluation and automatic updates of kinematic and dynamic quantities.
- Allows for the extension of the API to embed user-provided classes into its data structures.
- Provides comprehensive event recording in the simulation history.
- 3D visualization API using OpenGL and OpenSceneGraph with ImGui support.
- Extensible API to interface with various optimization problems, such as nonlinear programming and multi-objective optimization.

Collision Detection

- Support for multiple collision detectors: FCL, Bullet, and ODE.
- Support for various collision shapes including primitive shapes, concave mesh, and probabilistic voxel grid.
- Support for minimum distance computation.

Kinematics

- Support for numerous types of Joints.
- Support for numerous primitive and arbitrary body shapes with customizable inertial and material properties.
- Support for flexible skeleton modeling, including cloning and reconfiguring skeletons or subsections of a skeleton.
- Comprehensive access to kinematic states (e.g. transformation, position, velocity, or acceleration) of arbitrary entities and coordinate frames.
- Comprehensive access to various Jacobian matrices and their derivatives.
- Flexible conversion of coordinate frames.
- Fully modular inverse kinematics framework.
- Plug-and-play hierarchical whole-body inverse kinematics solver.
- Analytic inverse kinematics interface with ikfast support.

Dynamics

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

5.2 Gallery

5.2.1 Built-in Examples

Atlas Simbicon

The *Atlas Simbicon* demo simulates Atlas humanoid robot controlled by Simbicon, which is a simple biped locomotion controller. This demo is a fully 3D simulation with articulated dynamics and rigid body collisions. This demo uses OpenSceneGraph for 3D rendering and ImGui for 2D on-screen buttons. The source code can be found in the examples/osg/osgAtlasSimbicon directory.

The controller has three control modes: 'No Control', 'Short-Stride Walking', and 'Normal-Stride Walking'. The Atlas robot can walk keeping its balance with disturbances. You can apply external forces to the torso using the keyboard:

- 'A' Key: push forward the torso of Atlas
- 'S' Key: push backward the torso of Atlas
- 'D' Key: push left the torso of Atlas
- 'F' Key: push right the torso of Atlas

"Reset Atlas" button is for recovering the robot to the initial location and initial pose, which is useful when it' s fallen or bungee-jumped out of the ground.

Gravity can be varied using the slider at the 2D GUI.

Tinkertoy

The *tinkertoy* demo simulates trees of *tinkertoys*, which are blocks connected to each other via various joint types. This is a fully 3D simulation with articulated dynamics and rigid body collisions. This demo uses OpenSceneGraph for 3D rendering and ImGui for 2D on-screen buttons. The source code can be found in the examples/osg/osgTinkertoy directory.

The scene starts out with two trees of *tinkertoys*. Left-click on one of the blocks to attach a target to it. During simulation, the block that is attached to the target will be pulled towards the target with a spring-like force. Move the target around by clicking on its handles to change the direction of the force.

When simulation is paused, the user can change the structure of the tinkertoy by adding and removing blocks. Click on:

- Add a Weld-Joint Block
- Add a Revolute-Joint Block
- Add a Ball-Joint Block

to add a block with the specified joint type at the current target location. The new block will be attached to whichever block the target is currently attached to. You can use the Reset Target button to detach the target from all blocks, allowing you to construct a new tinkertoy tree. The new block will run along the x-axis (red arrow) of the target. For revolute joints, the joint axis will align with the target' s z-axis (blue arrow).

Gravity can be toggled from the 2D GUI or by pressing the *G* key.

5.2.2 Examples on Gazebo

The non-profit organization, OSRF (Open Source Robotic Foundation), conducted a performance comparison (first video) on four physics engines: ODE, Bullet, DART, and SimBody (ROSCon 2014). An Atlas robot was simulated using each of the physics engines to determine the highest RTF that results in stable walking motion. RTF (real-time factor) is a metric to measure the speed of the physics engine. RTF < 1 means slower than real time and RTF > 1 means faster than real time. DART achieved 1.6-1.9 RTF, faster than ODE (1.45), Bullet (0.15), and SimBody (0.1). The second video shows that, despite the differences in performance, the motion trajectories simulated by different physics engines are qualitatively similar.

5.3 Installation

5.3.1 Python

To install the Python bindings for DART using the *dartpy* package from PyPI, you can use the following command:

```
pip install dartpy -U
```

The following operating systems are currently supported:

Operating System	Python 3.7	Python 3.8	Python 3.9	Python 3.10	Python 3.11
Linux x86_64	O	O	O	O	O
Linux arm64	X	X	X	X	O
macOS x86_64	X	O	O	O	O
macOS arm64	X	O	X	O	O
Windows x86_64	X	O	O	O	O
Windows arm64	X	X	X	X	O

Note: This table may not be up-to-date. For the latest information on the availability of the Python bindings for DART, please refer to the dartpy package on PyPI: https://pypi.org/project/dartpy/. If you would like to use dartpy on an unsupported OS or Python version, please let us know so we can consider adding support.

5.3.2 C++

Ubuntu

To install DART on Ubuntu, you can use the following commands:

1. Add the DART PPA to your system:

```
sudo apt-add-repository ppa:dartsim/ppa
```

2. Update your package list:

```
sudo apt-get update
```

3. Install the *libdart6-all-dev* package:

```
sudo apt-get install libdart7-all-dev-nightly
```

macOS

To install DART on macOS, you can use Homebrew:

1. Install Homebrew if you haven' t already:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/
↪install.sh)"
```

2. Install the *dartsim* formula:

```
brew install dartsim
```

Windows

To install DART on Windows, you can use vcpkg:

1. Install vcpkg if you haven' t already:

```
git clone https://github.com/microsoft/vcpkg.git
cd vcpkg
bootstrap-vcpkg.bat
```

2. Install the *dartsim* package:

```
vcpkg install dartsim:x64-windows
```

Arch Linux (experimental)

To install DART on Arch Linux using the *yay* package manager, you can use the following commands:

1. Update your package list:

```
yay -Syu
```

2. Install the *libdart* package:

```
yay -S libdart
```

FreeBSD (experimental)

To install DART on FreeBSD, you can use the following commands:

1. Update your package list:

```
pkg update
```

2. Install the *dartsim* package:

```
pkg install dartsim
```

5.4 Examples

Once you have installed dartpy using pip install -U dartpy, you can run the following “Hello World” example to simulate a 6-DOF robot using DART.

Note: In order to load the URDF, please clone dart repository and set the *DART_DATA_LOCAL_PATH* environment variable to where the *data* folder is in the cloned repository (e.g., *C:/ws/dart/data/* if cloned to *C:/ws/dart/*)

```python
import dartpy as dart

def main():
    world = dart.simulation.World()

    urdf_parser = dart.io.DartLoader()
    kr5 = urdf_parser.parseSkeleton("dart://sample/urdf/KR5/KR5 sixx R650.urdf")
    ground = urdf_parser.parseSkeleton("dart://sample/urdf/KR5/ground.urdf")
    world.addSkeleton(kr5)
    world.addSkeleton(ground)
    print("Robot {} is loaded".format(kr5.getName()))

    for i in range(100):
        if i % 10 == 0:
            print(
                "[{}] joint position: {}".format(
                    world.getSimFrames(), kr5.getPositions()
                )
            )
        world.step()
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    main()
```

When you run this script, it will perform a forward dynamic simulation of the 6-DOF robot for 100 steps. Joint angles are printed every 10 steps, producing the following output:

```
Robot KR5sixxR650WP_description is loaded
[0] joint position: [0. 0. 0. 0. 0. 0.]
[10] joint position: [ 0.00220342  0.00021945 -0.00040518  0.00011133  0.00074889 -0.
↪00010902]
[20] joint position: [ 0.00841056  0.0008539  -0.0015611   0.00042308  0.00284968 -0.
↪00040791]
[30] joint position: [ 0.01861372  0.00194988 -0.00350848  0.00092958  0.0062733  -0.
↪00087254]
[40] joint position: [ 0.03279843  0.00358421 -0.00631463  0.00162151  0.01097006 -0.
↪0014636 ]
[50] joint position: [ 0.05094093  0.00586373 -0.01007336  0.00248606  0.01686793 -0.
↪00212772]
[60] joint position: [ 0.07300469  0.00892482 -0.01490498  0.00350698  0.02387041 -0.
↪00279907]
[70] joint position: [ 0.098936    0.01293271 -0.02095646  0.00466479  0.03185446 -0.
↪0034017 ]
[80] joint position: [ 0.12865883  0.01808055 -0.02840175  0.00593683  0.04066865 -0.
↪00385261]
[90] joint position: [ 0.16206903  0.02458815 -0.03744247  0.00729732  0.05013223 -0.
↪00406555]
```

You can find additional example code at https://github.com/dartsim/dart/tree/main/python/examples

5.5 Tutorials

The purpose of this tutorial is to provide a quick introduction to using DART. We designed many hands-on exercises to make the learning effective and fun. To follow along with this tutorial, first locate the tutorial code in the directory: dart/tutorials. For each of the four tutorials, we provide the skeleton code as the starting point (e.g. tutorial_multi_pendulum) and the final code as the answer to the tutorial (e.g. tutorial_multi_pendulum_finished). The examples are based on the APIs of DART 7.0.

5.5.1 Biped

Overview

This tutorial demonstrates the dynamic features in DART useful for developing controllers for bipedal or wheel-based robots. The tutorial consists of seven Lessons covering the following topics:

- Joint limits and self-collision.
- Actuators types and management.
- APIs for Jacobian matrices and other kinematic quantities.
- APIs for dynamic quantities.
- Skeleton editing.

Please reference the source code in **tutorialBiped.cpp** and **tutorialBiped-Finished.cpp**.

Lesson 1: Joint limits and self-collision

Let' s start by locating the `main` function in tutorialBiped.cpp. We first create a floor and call `loadBiped` to load a bipedal figure described in SKEL format, which is an XML format representing a robot model. A SKEL file describes a `World` with one or more `Skeletons` in it. Here we load in a World from **biped.skel** and assign the bipedal figure to a `Skeleton` pointer called *biped*.

```
SkeletonPtr loadBiped()
{
...
    WorldPtr world = SkelParser::readWorld(DART_DATA_LOCAL_PATH"skel/biped.skel");
    SkeletonPtr biped = world->getSkeleton("biped");
...
}
```

Running the skeleton code (hit the spacebar) without any modification, you should see a human-like character collapse on the ground and fold in on itself. Before we attempt to control the biped, let' s first make the biped a bit more realistic by enforcing more human-like joint limits.

DART allows the user to set upper and lower bounds on each degree of freedom in the SKEL file or using provided APIs. For example, you should see the description of the right knee joint in **biped.skel**:

```
<joint type="revolute" name="j_shin_right">
...
    <axis>
        <xyz>0.0 0.0 1.0</xyz>
        <limit>
            <lower>-3.14</lower>
            <upper>0.0</upper>
        </limit>
    </axis>
...
</joint>
```

The <upper> and <lower> tags make sure that the knee can only flex but not extend. Alternatively, you can directly specify the joint limits in the code using `setPositionUpperLimit` and `setPositionLowerLimit`.

In either case, the joint limits on the biped will not be activated until you call `setPositionLimited`:

```
SkeletonPtr loadBiped()
{
...
    for(size_t i = 0; i < biped->getNumJoints(); ++i)
        biped->getJoint(i)->setLimitEnforcement(true);
...
}
```

Once the joint limits are set, the next task is to enforce self-collision. By default, DART does not check self-collision within a skeleton. You can enable self-collision checking on the biped by

```
SkeletonPtr loadBiped()
{
...
    biped->enableSelfCollisionCheck();
```

(continues on next page)

(continued from previous page)

```
...
}
```

This function will enable self-collision on every pair of body nodes. If you wish to disable self-collisions on adjacent body nodes, call the following function

```
biped->disableAdjacentBodyCheck();
```

Running the program again, you should see that the character is still floppy like a ragdoll, but now the joints do not bend backward and the body nodes do not penetrate each other anymore.

Lesson 2: Proportional-derivative control

To actively control its own motion, the biped must exert internal forces using actuators. In this Lesson, we will design one of the simplest controllers to produce internal forces that make the biped hold a target pose. The proportional-derivative (PD) control computes control force by Τ = -kp (θ - θtarget) - kd θ, where θ and θ are the current position and velocity of a degree of freedom, θtarget is the target position set by the controller, and kp and kd are the stiffness and damping coefficients. The detailed description of a PD controller can be found here.

The first task is to set the biped to a particular configuration. You can use `setPosition` to set each degree of freedom individually:

```
void setInitialPose(SkeletonPtr biped)
{
...
    biped->setPosition(biped->getDof("j_thigh_left_z")->getIndexInSkeleton(), 0.15);
...
}
```

Here the degree of freedom named "j_thigh_left_z" is set to 0.15 radian. Note that each degree of freedom in a skeleton has a numerical index which can be accessed by `getIndexInSkeleton`. You can also set the entire configuration using a vector that holds the positions of all the degreed of freedoms using `setPositions`.

We continue to set more degrees of freedoms in the lower body to create a roughly stable standing pose.

```
biped->setPosition(biped->getDof("j_thigh_left_z")->getIndexInSkeleton(), 0.15);
biped->setPosition(biped->getDof("j_thigh_right_z")->getIndexInSkeleton(), 0.15);
biped->setPosition(biped->getDof("j_shin_left")->getIndexInSkeleton(), -0.4);
biped->setPosition(biped->getDof("j_shin_right")->getIndexInSkeleton(), -0.4);
biped->setPosition(biped->getDof("j_heel_left_1")->getIndexInSkeleton(), 0.25);
biped->setPosition(biped->getDof("j_heel_right_1")->getIndexInSkeleton(), 0.25);
```

Now the biped will start in this configuration, but will not maintain this configuration as soon as the simulation starts. We need a controller to make this happen. Let' s take a look at the constructor of our `Controller` in the skeleton code:

```
Controller(const SkeletonPtr& biped)
{
...
    for(size_t i = 0; i < 6; ++i)
    {
        mKp(i, i) = 0.0;
        mKd(i, i) = 0.0;
    }

    for(size_t i = 6; i < mBiped->getNumDofs(); ++i)
```

(continues on next page)

(continued from previous page)

```
    {
        mKp(i, i) = 1000;
        mKd(i, i) = 50;
    }

    setTargetPositions(mBiped->getPositions());
}
```

Here we arbitrarily define the stiffness and damping coefficients to 1000 and 50, except for the first six degrees of freedom. Because the global translation and rotation of the biped are not actuated, the first six degrees of freedom at the root do not exert any internal force. Therefore, we set the stiffness and damping coefficients to zero. At the end of the constructor, we set the target position of the PD controller to the current configuration of the biped.

With these settings, we can compute the forces generated by the PD controller and add them to the internal forces of biped using `setForces`:

```
void addPDForces()
{
    math::VectorXd q = mBiped->getPositions();
    math::VectorXd dq = mBiped->getVelocities();

    math::VectorXd p = -mKp * (q - mTargetPositions);
    math::VectorXd d = -mKd * dq;

    mForces += p + d;
    mBiped->setForces(mForces);
}
```

Note that the PD control force is *added* to the current internal force stored in mForces instead of overriding it.

Now try to run the program and see what happens. The skeleton disappears almost immediately as soon as you hit the space bar! This is because our stiffness and damping coefficients are set way too high. As soon as the biped deviates from the target position, huge internal forces are generated to cause the numerical simulation to blow up.

So let' s lower those coefficients a bit. It turns out that each of the degrees of freedom needs to be individually tuned depending on many factors, such as the inertial properties of the body nodes, the type and properties of joints, and the current configuration of the system. Figuring out an appropriate set of coefficients can be a tedious process difficult to generalize across new tasks or different skeletons. In the next Lesson, we will introduce a much more efficient way to stabilize the PD controllers without endless tuning and trial-and-errors.

Lesson 3: Stable PD control

SPD is a variation of PD control proposed by Jie Tan. The basic idea of SPD is to compute control force using the predicted state at the next time step, instead of the current state. This Lesson will only demonstrate the implementation of SPD using DART without going into details of SPD derivation.

The implementation of SPD involves accessing the current dynamic quantities in Lagrange' s equations of motion. Fortunately, these quantities are readily available via DART API, which makes the full implementation of SPD simple and concise:

```
void addSPDForces()
{
    math::VectorXd q = mBiped->getPositions();
    math::VectorXd dq = mBiped->getVelocities();

    math::MatrixXd invM = (mBiped->getMassMatrix() + mKd * mBiped->getTimeStep()).
```

(continues on next page)

(continued from previous page)

```
↪inverse();
    math::VectorXd p = -mKp * (q + dq * mBiped->getTimeStep() - mTargetPositions);
    math::VectorXd d = -mKd * dq;
    math::VectorXd qddot = invM * (-mBiped->getCoriolisAndGravityForces() + p + d +␣
↪mBiped->getConstraintForces());

    mForces += p + d - mKd * qddot * mBiped->getTimeStep();
    mBiped->setForces(mForces);
}
```

You can get mass matrix, Coriolis force, gravitational force, and constraint force projected onto generalized coordinates using function calls `getMassMatrix`, `getCoriolisForces`, `getGravityForces`, and `getConstraintForces`, respectively. Constraint forces include forces due to contacts, joint limits, and other joint constraints set by the user (e.g. the weld joint constraint in the multi-pendulum tutorial).

With SPD, a wide range of stiffness and damping coefficients will all result in stable motion. In fact, you can just leave them to our original values: 1000 and 50. By holding the target pose, now the biped can stand on the ground in balance indefinitely. However, if you apply an external push force on the biped (hit ',' or '.' key to apply a backward or forward push), the biped loses its balance quickly. We will demonstrate a more robust feedback controller in the next Lesson.

Lesson 4: Ankle strategy

Ankle (or hip) strategy is an effective way to maintain standing balance. The idea is to adjust the target position of ankles according to the deviation between the center of mass and the center of pressure projected on the ground. A simple linear feedback rule is used to update the target ankle position: θa = -kp (x - p) - kd (ẋ - ṗ), where x and p indicate the center of mass and center of pressure in the anterior-posterior axis. kp and kd are the feedback gains defined by the user.

To implement ankle strategy, let' s first compute the deviation between the center of mass and an approximated center of pressure in the anterior-posterior axis:

```
void addAnkleStrategyForces()
{
    math::Vector3d COM = mBiped->getCOM();
    math::Vector3d offset(0.05, 0, 0);
    math::Vector3d COP = mBiped->getBodyNode("h_heel_left")->getTransform() * offset;
    double diff = COM[0] - COP[0];
...
}
```

DART provides various APIs to access useful kinematic information. For example, `getCOM` returns the center of mass of the skeleton and `getTransform` returns transformation of the body node with respect to any coordinate frame specified by the parameter (world coordinate frame as default). DART APIs also come in handy when computing the derivative term, -kd (ẋ - ṗ):

```
void addAnkleStrategyForces()
{
...
    math::Vector3d dCOM = mBiped->getCOMLinearVelocity();
    math::Vector3d dCOP =  mBiped->getBodyNode("h_heel_left")->
↪getLinearVelocity(offset);
    double dDiff = dCOM[0] - dCOP[0];
...
}
```

The linear/angular velocity/acceleration of any point in any coordinate frame can be easily accessed in DART. The full list of the APIs for accessing various velocities/accelerations can be found in the API Documentation. The following table summarizes the essential APIs.

Function Name	Description
getSpatialVelocity	Return the spatial velocity of this BodyNode in the coordinates of the BodyNode.
getLinearVelocity	Return the linear portion of classical velocity of the BodyNode relative to some other BodyNode.
getAngularVelocity	Return the angular portion of classical velocity of this BodyNode relative to some other BodyNode.
getSpatialAcceleration	Return the spatial acceleration of this BodyNode in the coordinates of the BodyNode.
getLinearAcceleration	Return the linear portion of classical acceleration of the BodyNode relative to some other BodyNode.
getAngularAcceleration	Return the angular portion of classical acceleration of this BodyNode relative to some other BodyNode.

The remaining of the ankle strategy implementation is just the matter of parameters tuning. We found that using different feedback rules for falling forward and backward result in more stable controller.

Lesson 5: Skeleton editing

DART provides various functions to copy, delete, split, and merge parts of skeletons to alleviate the pain of building simulation models from scratch. In this Lesson, we will load a skateboard model from a SKEL file and merge our biped with the skateboard to create a wheel-based robot.

We first load a skateboard from **skateboard.skel**:

```
void modifyBipedWithSkateboard(SkeletonPtr biped)
{
    WorldPtr world = SkelParser::readWorld(DART_DATA_LOCAL_PATH"skel/skateboard.skel
↪");
    SkeletonPtr skateboard = world->getSkeleton(0);
...
}
```

Our goal is to make the skateboard Skeleton a subtree of the biped Skeleton connected to the left heel BodyNode via a newly created Euler joint. To do so, you need to first create an instance of `EulerJoint::Properties` for this new joint.

```
void modifyBipedWithSkateboard(SkeletonPtr biped)
{
...
    EulerJoint::Properties properties = EulerJoint::Properties();
    properties.mT_ChildBodyToJoint.translation() = math::Vector3d(0, 0.1, 0);
...
}
```

Here we increase the vertical distance between the child BodyNode and the joint by 0.1m to give some space between the skateboard and the left foot. Now you can merge the skateboard and the biped using this new Euler joint by

```
void modifyBipedWithSkateboard(SkeletonPtr biped)
{
...
    skateboard->getRootBodyNode()->moveTo<EulerJoint>(biped->getBodyNode("h_heel_left
```

(continues on next page)

(continued from previous page)

```
↪"), properties);
}
```

There are many other functions you can use to edit skeletons. Here is a table of some relevant functions for quick references.

Function Name	Example	Description
remove	bd1->remove()	Remove the BodyNode bd1 and its subtree from their Skeleton.
moveTo	bd1->moveTo(bd2)	Move the BodyNode bd1 and its subtree under the BodyNode bd2.
split	auto newSkel = bd1->split("new skeleton")`	Remove the BodyNode bd1 and its subtree from their current Skeleton and move them into a newly created Skeleton with "new skeleton" name.
changeParentJointType	bd1->changeParentJointType<	Change the Joint type of the BodyNode bd1' s parent joint to BallJoint
copyTo	bd1->copyTo(bd2)	Create clones of the BodyNode bd1 and its subtree and attach the clones to the specified the BodyNode bd2.
copyAs	auto newSkel = bd1->copyAs("new skeleton")	Create clones of the BodyNode bd1 and its subtree and create a new Skeleton with "new skeleton" name to attach them to.

Lesson 6: Actuator types

DART provides five types of actuator. Each joint can select its own actuator type.

Type	Description
FORCE	Take joint force and return the resulting joint acceleration.
PASSIVE	Take nothing (joint force = 0) and return the resulting joint acceleration.
ACCELERATION	Take desired joint acceleration and return the joint force to achieve the acceleration.
VELOCITY	Take desired joint velocity and return the joint force to achieve the velocity.
LOCKED	Lock the joint by setting the joint velocity and acceleration to zero and return the joint force to lock the joint.

In this Lesson, we will switch the actuator type of the wheels from the default FORCE type to VELOCITY type.

```
void setVelocityActuators(SkeletonPtr biped)
{
    Joint* wheel1 = biped->getJoint("joint_front_left");
    wheel1->setActuatorType(Joint::VELOCITY);
...
}
```

Once all four wheels are set to VELOCITY actuator type, you can command them by directly setting the desired velocity:

```
void setWheelCommands()
{
...
    int index1 = mBiped->getDof("joint_front_left_2")->getIndexInSkeleton();
    mBiped->setCommand(index1, mSpeed);
```

(continues on next page)

(continued from previous page)

```
...
}
```

Note that `setCommand` only exerts commanding force in the current time step. If you wish the wheel to continue spinning at a particular speed, `setCommand` needs to be called at every time step.

We also set the stiffness and damping coefficients for the wheels to zero.

```
void setWheelCommands()
{
    int wheelFirstIndex = mBiped->getDof("joint_front_left_1")->getIndexInSkeleton();
    for (size_t i = wheelFirstIndex; i < mBiped->getNumDofs(); ++i)
    {
        mKp(i, i) = 0.0;
        mKd(i, i) = 0.0;
    }
...
}
```

This is because we do not want the velocity-based actuators to incorrectly affect the computation of SPD. If we use simple PD control scheme, the values of these spring and damping coefficients do not affect the dynamics of the system.

Let' s simulate what we' ve got so far. The biped now is connecting to the skateboard through a Euler joint. Once the simulation starts, you can use 'a' and 's' to increase or decrease the wheel speed. However, the biped falls on the floor immediately because the current target pose is not balanced for one-foot stance. We need to find a better target pose.

Lesson 7: Inverse kinematics

Instead of manually designing a target pose, this time we will solve for a balanced pose by formulating an inverse kinematics (IK) problem and solving it using gradient descent method. In this example, a balanced pose is defined as a pose where the center of mass is well supported by the ground contact and the left foot lies flat on the ground. As such, we cast IK as an optimization problem that minimizes the horizontal deviation between the center of mass and the center of the left foot, as well as the vertical distance of the four corners of the left foot from the ground:

where c and p indicate the projected center of mass and center of pressure on the ground, and *pi* indicates the vertical height of one corner of the left foot.

To compute the gradient of the above objective function, we need to evaluate the partial derivatives of each objective term with respect to the degrees of freedom, i.e., the computation of Jacobian matrix. DART provides a comprensive set of APIs for accessing various types of Jacobian. In this example, computing the gradient of the first term of the objective function requires the Jacobian of the center of mass of the Skeleton, as well as the Jacobian of the center of mass of a BodyNode:

```
math::VectorXd solveIK(SkeletonPtr biped)
{
...
    math::Vector3d localCOM = leftHeel->getCOM(leftHeel);
    LinearJacobian jacobian = biped->getCOMLinearJacobian() - biped->
↪getLinearJacobian(leftHeel, localCOM);
...
}
```

`getCOMLinearJacobian` returns the linear Jacobian of the center of mass of the Skeleton, while `getLinearJacobian` returns the Jacobian of a point on a BodyNode. The BodyNode and the local coordinate of the point are specified as parameters to this function. Here the point of interest is the center of mass of the left foot, which local coordinates can be accessed by `getCOM` with a parameter indicating the left foot being the frame of reference. We use `getLinearJacobian` again to compute the gradient of the second term of the objective function:

```
math::VectorXd solveIK(SkeletonPtr biped)
{
...
    math::Vector3d offset(0.0, -0.04, -0.03);
    gradient = biped->getLinearJacobian(leftHeel, offset).row(1);
...
}
```

The full list of Jacobian APIs can be found in the API Documentation. The following table summarizes the essential APIs.

Function Name	Description
getJacobian	Return the generalized Jacobian targeting the origin of the BodyNode. The Jacobian is expressed in the Frame of this BodyNode.
getLinearJacobian	Return the linear Jacobian targeting the origin of the BodyNode. You can specify a coordinate Frame to express the Jacobian in.
getAngularJacobian	Return the angular Jacobian targeting the origin of the BodyNode. You can specify a coordinate Frame to express the Jacobian in.
getJacobianSpatialDeriv	Return the spatial time derivative of the generalized Jacobian targeting the origin of the BodyNode. The Jacobian is expressed in the BodyNode’ s coordinate Frame.
getJacobianClassicDeriv	Return the classical time derivative of the generalized Jacobian targeting the origin of the BodyNode. The Jacobian is expressed in the World coordinate Frame.
getLinearJacobianDeriv	Return the linear Jacobian (classical) time derivative, in terms of any coordinate Frame.
getAngularJacobianDeriv	Return the angular Jacobian (classical) time derivative, in terms of any coordinate Frame.

This Lesson concludes the entire Biped tutorial. You should see a biped standing stably on the skateboard. With moderate acceleration/deceleration on the skateboard, the biped is able to maintain balance and hold the one-foot stance pose.

5.5.2 Collisions

Overview

This tutorial will show you how to programmatically create different kinds of bodies and set initial conditions for Skeletons. It will also demonstrate some use of DART’ s Frame Semantics.

The tutorial consists of five Lessons covering the following topics:

- Creating a rigid body
- Creating a soft body
- Setting initial conditions and taking advantage of Frames
- Setting joint spring and damping properties
- Creating a closed kinematic chain

Please reference the source code in **tutorialCollisions.cpp** and **tutorialCollisions-Finished.cpp**.

Lesson 1: Creating a rigid body

Start by going opening the Skeleton code tutorialCollisions.cpp. Find the function named `addRigidBody`. You will notice that this is a templated function. If you’ re not familiar with templates, that’ s okay; we won’ t be doing anything too complicated with them. Different Joint types in DART are managed by a bunch of different classes, so we need to use templates if we want the same function to work with a variety of Joint types.

Lesson 1a: Setting joint properties

The first thing we’ ll want to do is set the Joint properties for our new body. Whenever we create a BodyNode, we must also create a parent Joint for it. A BodyNode needs a parent Joint, even if that BodyNode is the root of the Skeleton, because we need its parent Joint to describe how it’ s attached to the world. A root BodyNode could be attached to the world by any kind of Joint. Most often, it will be attached by either a FreeJoint (if the body should be completely free to move with respect to the world) or a WeldJoint (if the body should be rigidly attached to the world, unable to move at all), but *any* Joint type is permissible.

Joint properties are managed in a nested class, which means it’ s a class which is defined inside of another class. For example, `RevoluteJoint` properties are managed in a class called `RevoluteJoint::Properties` while `PrismaticJoint` properties are managed in a class called `PrismaticJoint::Properties`. However, both `RevoluteJoint` and `PrismaticJoint` inherit the `SingleDofJoint` class so the `RevoluteJoint::Properties` and `PrismaticJoint::Properties` classes both inherit the `SingleDofJoint::Properties` class. The difference is that `RevoluteJoint::Properties` also inherits `RevoluteJoint::UniqueProperties` whereas `PrismaticJoint::Properties` inherits `PrismaticJoint::UniqueProperties` instead. Many DART classes contain nested `Properties` classes like this which are compositions of their base class’ s nested `Properties` class and their own `UniqueProperties` class. As you’ ll see later, this is useful for providing a consistent API that works cleanly for fundamentally different types of classes.

To create a `Properties` class for our Joint type, we’ ll want to say

```
typename JointType::Properties joint_properties;
```

We need to include the `typename` keywords because of how the syntax works for templated functions. Leaving it out should make your compiler complain.

From here, we can set the Joint properties in any way we’ d like. There are only a few things we care about right now: First, the Joint’ s name. Every Joint in a Skeleton needs to have a non-empty unique name. Those are the only restrictions that are placed on Joint names. If you try to make a Joint’ s name empty, it will be given a default name. If you try to make a Joint’ s name non-unique, DART will append a number tag to the end of the name in order to make it unique. It will also print out a warning during run time, which can be an eyesore (because it wants you to be aware when you are being negligent about naming things). For the sake of simplicity, let’ s just give it a name based off its child BodyNode:

```
joint_properties.mName = name+"_joint";
```

Don’ t forget to uncomment the function arguments.

Next we’ ll want to deal with offsetting the new BodyNode from its parent BodyNode. We can use the following to check if there is a parent BodyNode:

```
if(parent)
{
  // TODO: offset the child from its parent
}
```

Inside the brackets, we’ ll want to create the offset between bodies:

```
math::Isometry3d tf(math::Isometry3d::Identity());
```

An `math::Isometry3d` is the Eigen library' s version of a homogeneous transformation matrix. Here we are initializing it to an Identity matrix to start out. This is almost always something you should do when creating an math::Isometry3d, because otherwise its contents will be completely arbitrary trash.

We can easily compute the center point between the origins of the two bodies using our default height value:

```
tf.translation() = math::Vector3d(0, 0, default_shape_height / 2.0);
```

We can then offset the parent and child BodyNodes of this Joint using this transform:

```
joint_properties.mT_ParentBodyToJoint = tf;
joint_properties.mT_ChildBodyToJoint = tf.inverse();
```

Remember that all of that code should go inside the `if(parent)` condition. We do not want to create this offset for root BodyNodes, because later on we will rely on the assumption that the root Joint origin is lined up with the root BodyNode origin.

Lesson 1b: Create a Joint and BodyNode pair

A single function is used to simultaneously create a new Joint and its child BodyNode. It' s important to note that a Joint cannot be created without a child BodyNode to accompany it, and a BodyNode cannot be created with parent Joint to attach it to something. A parent Joint without a child BodyNode or vice-versa would be non-physical and nonsensical, so we don' t allow it.

Use the following to create a new Joint & BodyNode, and obtain a pointer to that new BodyNode:

```
BodyNode* bn = chain->createJointAndBodyNodePair<JointType>(
      parent, joint_properties, BodyNode::AspectProperties(name)).second;
```

There' s a lot going on in this function, so let' s break it down for a moment:

```
chain->createJointAndBodyNodePair<JointType>
```

This is a Skeleton member function that takes template arguments. The first template argument specifies the type of Joint that you want to create. In our case, the type of Joint we want to create is actually a template argument of our current function, so we just pass that argument along. The second template argument of `createJointAndBodyNodePair` allows us to specify the BodyNode type that we want to create, but the default argument is a standard rigid BodyNode, so we can leave the second argument blank.

```
(parent, joint_properties, BodyNode::AspectProperties(name))
```

Now for the function arguments: The first specifies the parent BodyNode. In the event that you want to create a root BodyNode, you can simply pass in a nullptr as the parent. The second argument is a `JointType::Properties` struct, so we pass in the `joint_properties` object that we created earlier. The third argument is a `BodyNode::Properties` struct, but we' re going to set the BodyNode properties later, so we' ll just toss the name in by wrapping it up in a `BodyNode::AspectProperties` object and leave the rest as default values.

Now notice the very last thing on this line of code:

```
.second;
```

The function actually returns a `std::pair` of pointers to the new Joint and new BodyNode that were just created, but we only care about grabbing the BodyNode once the function is finished, so we can append `.second` to the end of the line so that we just grab the BodyNode pointer and ignore the Joint pointer. The joint will of course still be created; we just have no need to access it at this point.

Lesson 1c: Make a shape for the body

We' ll take advantage of the Shape::ShapeType enumeration to specify what kind of Shape we want to produce for the body. In particular, we' ll allow the user to specify three types of Shapes: `Shape::BOX`, `Shape::CYLINDER`, and `Shape::ELLIPSOID`.

```
ShapePtr shape;
if(Shape::BOX == type)
{
  // TODO: Make a box
}
else if(Shape::CYLINDER == type)
{
  // TODO: Make a cylinder
}
else if(SHAPE::ELLIPSOID == type)
{
  // TODO: Make an ellipsoid
}
```

`ShapePtr` is simply a typedef for `std::shared_ptr<Shape>`. DART has this typedef in order to improve space usage and readability, because this type gets used very often.

Now we want to construct each of the Shape types within their conditional statements. Each constructor is a bit different.

For box we pass in an math::Vector3d that contains the three dimensions of the box:

```
shape = std::make_shared<BoxShape>(math::Vector3d(
                                     default_shape_width,
                                     default_shape_width,
                                     default_shape_height));
```

For cylinder we pass in a radius and a height:

```
shape = std::make_shared<CylinderShape>(default_shape_width/2.0,
                                        default_shape_height);
```

For ellipsoid we pass in an math::Vector3d that contains the lengths of the three axes:

```
shape = std::make_shared<EllipsoidShape>(
      default_shape_height*math::Vector3d::Ones());
```

Since we actually want a sphere, all three axis lengths will be equal, so we can create an math::Vector3d filled with ones by using `math::Vector3d::Ones()` and then multiply it by the length that we actually want for the three components.

Finally, we want to add this shape as a visualization **and** collision shape for the BodyNode:

```
auto shapeNode =
   bn->createShapeNodeWith<VisualAspect, CollisionAspect, DynamicsAspect>(shape);
```

We want to do this no matter which type was selected, so those two lines of code should be after all the condition statements.

Lesson 1d: Set up the inertia properties for the body

For the simulations to be physically accurate, it’ s important for the inertia properties of the body to match up with the geometric properties of the shape. We can create an `Inertia` object and set its values based on the shape’ s geometry, then give that `Inertia` to the BodyNode.

```
Inertia inertia;
double mass = default_shape_density * shape->getVolume();
inertia.setMass(mass);
inertia.setMoment(shape->computeInertia(mass));
bn->setInertia(inertia);
```

Lesson 1e: Set the coefficient of restitution

This is very easily done with the following function:

```
shapeNode->getDynamicsAspect()->setRestitutionCoeff(default_restitution);
```

Lesson 1f: Set the damping coefficient

In real life, joints have friction. This pulls energy out of systems over time, and makes those systems more stable. In our simulation, we’ ll ignore air friction, but we’ ll add friction in the joints between bodies in order to have better numerical and dynamic stability:

```
if(parent)
{
  Joint* joint = bn->getParentJoint();
  for(size_t i=0; i < joint->getNumDofs(); ++i)
    joint->getDof(i)->setDampingCoefficient(default_damping_coefficient);
}
```

If this BodyNode has a parent BodyNode, then we set damping coefficients of its Joint to a default value.

Lesson 2: Creating a soft body

Find the templated function named `addSoftBody`. This function will have a role identical to the `addRigidBody` function from earlier.

Lesson 2a: Set the Joint properties

This portion is exactly the same as Lesson 1a. You can even copy the code directly from there if you’ d like to.

Lesson 2b: Set the properties of the soft body

Last time we set the BodyNode properties after creating it, but this time we' ll set them beforehand.

First, let' s create a struct for the properties that are unique to SoftBodyNodes:

```
SoftBodyNode::UniqueProperties soft_properties;
```

Later we will combine this with a standard `BodyNode::Properties` struct, but for now let' s fill it in. Up above we defined an enumeration for a couple different SoftBodyNode types. There is no official DART-native enumeration for this, we created our own to use for this function. We' ll want to fill in the `SoftBodyNode::UniqueProperties` struct based off of this enumeration:

```
if(SOFT_BOX == type)
{
  // TODO: make a soft box
}
else if(SOFT_CYLINDER == type)
{
  // TODO: make a soft cylinder
}
else if(SOFT_ELLIPSOID == type)
{
  // TODO: make a soft ellipsoid
}
```

Each of these types has a static function in the `SoftBodyNodeHelper` class that will set up your `UniqueProperties` for you. The arguments for each of the functions are a bit complicated, so here is how to call it for each type:

For the SOFT_BOX:

```
// Make a wide and short box
double width = default_shape_height, height = 2*default_shape_width;
math::Vector3d dims(width, width, height);

double mass = 2*dims[0]*dims[1] + 2*dims[0]*dims[2] + 2*dims[1]*dims[2];
mass *= default_shape_density * default_skin_thickness;
soft_properties = SoftBodyNodeHelper::makeBoxProperties(
      dims, math::Isometry3d::Identity(), math::Vector3i(4,4,4), mass);
```

For the SOFT_CYLINDER:

```
// Make a wide and short cylinder
double radius = default_shape_height/2.0, height = 2*default_shape_width;

// Mass of center
double mass = default_shape_density * height * 2*M_PI*radius
              * default_skin_thickness;
// Mass of top and bottom
mass += 2 * default_shape_density * M_PI*pow(radius,2)
            * default_skin_thickness;
soft_properties = SoftBodyNodeHelper::makeCylinderProperties(
      radius, height, 8, 3, 2, mass);
```

And for the SOFT_ELLIPSOID:

```
double radius = default_shape_height/2.0;
math::Vector3d dims = 2*radius*math::Vector3d::Ones();
```

(continues on next page)

(continued from previous page)

```
double mass = default_shape_density * 4.0*M_PI*pow(radius, 2)
              * default_skin_thickness;
soft_properties = SoftBodyNodeHelper::makeEllipsoidProperties(
      dims, 6, 6, mass);
```

Feel free to play around with the different parameters, like number of slices and number of stacks. However, be aware that some of those parameters have a minimum value, usually of 2 or 3. During runtime, you should be warned if you try to create one with a parameter that' s too small.

Lastly, we' ll want to fill in the softness coefficients:

```
soft_properties.mKv = default_vertex_stiffness;
soft_properties.mKe = default_edge_stiffness;
soft_properties.mDampCoeff = default_soft_damping;
```

Lesson 2c: Create the Joint and Soft Body pair

This step is very similar to Lesson 1b, except now we' ll want to specify that we' re creating a soft BodyNode. First, let' s create a full `SoftBodyNode::Properties`:

```
SoftBodyNode::Properties body_properties(BodyNode::AspectProperties(name),
                                          soft_properties);
```

This will combine the `UniqueProperties` of the SoftBodyNode with the standard properties of a BodyNode. Now we can pass the whole thing into the creation function:

```
SoftBodyNode* bn = chain->createJointAndBodyNodePair<JointType, SoftBodyNode>(
      parent, joint_properties, body_properties).second;
```

Notice that this time it will return a `SoftBodyNode` pointer rather than a normal `BodyNode` pointer. This is one of the advantages of templates!

Lesson 2d: Zero out the BodyNode inertia

A SoftBodyNode has two sources of inertia: the underlying inertia of the standard BodyNode class, and the point mass inertias of its soft skin. In our case, we only want the point mass inertias, so we should zero out the standard BodyNode inertia. However, zeroing out inertia values can be very dangerous, because it can easily result in singularities. So instead of completely zeroing them out, we will just make them small enough that they don' t impact the simulation:

```
Inertia inertia;
inertia.setMoment(1e-8*math::Matrix3d::Identity());
inertia.setMass(1e-8);
bn->setInertia(inertia);
```

Lesson 2e: Make the shape transparent

To help us visually distinguish between the soft and rigid portions of a body, we can make the soft part of the shape transparent. Upon creation, a SoftBodyNode will have exactly one visualization shape: the soft shape visualizer. We can grab that shape and reduce the value of its alpha channel:

```
auto shape = bn->getShapeNodesWith<VisualAspect>()[0];
math::Vector4d color = shape->getVisualAspect()->getRGBA();
color[3] = 0.4;
shape->getVisualAspect()->setRGBA(color);
```

Lesson 2f: Give a hard bone to the SoftBodyNode

SoftBodyNodes are intended to be used as soft skins that are attached to rigid bones. We can create a rigid shape, place it in the SoftBodyNode, and give some inertia to the SoftBodyNode' s base BodyNode class, to act as the inertia of the bone.

Find the function `createSoftBody()`. Underneath the call to `addSoftBody`, we can create a box shape that matches the dimensions of the soft box, but scaled down:

```
double width = default_shape_height, height = 2*default_shape_width;
math::Vector3d dims(width, width, height);
dims *= 0.6;
std::shared_ptr<BoxShape> box = std::make_shared<BoxShape>(dims);
```

And then we can add that shape to the visualization and collision shapes of the SoftBodyNode, just like normal:

```
bn->createShapeNodeWith<VisualAspect, CollisionAspect, DynamicsAspect>(box);
```

And we' ll want to make sure that we set the inertia of the underlying BodyNode, or else the behavior will not be realistic:

```
Inertia inertia;
inertia.setMass(default_shape_density * box->getVolume());
inertia.setMoment(box->computeInertia(inertia.getMass()));
bn->setInertia(inertia);
```

Note that the inertia of the inherited BodyNode is independent of the inertia of the SoftBodyNode' s skin.

Lesson 2g: Add a rigid body attached by a WeldJoint

To make a more interesting hybrid shape, we can attach a protruding rigid body to a SoftBodyNode using a WeldJoint. Find the `createHybridBody()` function and see where we call the `addSoftBody` function. Just below this, we' ll create a new rigid body with a WeldJoint attachment:

```
bn = hybrid->createJointAndBodyNodePair<WeldJoint>(bn).second;
bn->setName("rigid box");
```

Now we can give the new rigid BodyNode a regular box shape:

```
double box_shape_height = default_shape_height;
std::shared_ptr<BoxShape> box = std::make_shared<BoxShape>(
      box_shape_height*math::Vector3d::Ones());

bn->createShapeNodeWith<VisualAspect, CollisionAspect, DynamicsAspect>(box);
```

To make the box protrude, we' ll shift it away from the center of its parent:

```
math::Isometry3d tf(math::Isometry3d::Identity());
tf.translation() = math::Vector3d(box_shape_height/2.0, 0, 0);
bn->getParentJoint()->setTransformFromParentBodyNode(tf);
```

And be sure to set its inertia, or else the simulation will not be realistic:

```
Inertia inertia;
inertia.setMass(default_shape_density * box->getVolume());
inertia.setMoment(box->computeInertia(inertia.getMass()));
bn->setInertia(inertia);
```

Lesson 3: Setting initial conditions and taking advantage of Frames

Find the `addObject` function in the `MyWorld` class. This function will be called whenever the user requests for an object to be added to the world. In this function, we want to set up the initial conditions for the object so that it gets thrown at the wall. We also want to make sure that it' s not in collision with anything at the time that it' s added, because that would result in problems for the simulation.

Lesson 3a: Set the starting position for the object

We want to position the object in a reasonable place for us to throw it at the wall. We also want to have the ability to randomize its location along the y-axis.

First, let' s create a zero vector for the position:

```
math::Vector6d positions(math::Vector6d::Zero());
```

You' ll notice that this is an math::Vector**6**d rather than the usual math::Vector**3**d. This vector has six components because the root BodyNode has 6 degrees of freedom: three for orientation and three for translation. Because we follow Roy Featherstone' s Spatial Vector convention, the **first** three components are for **orientation** using a logmap (also known as angle-axis) and the **last** three components are for **translation**.

First, if randomness is turned on, we' ll set the y-translation to a randomized value:

```
if(mRandomize)
  positions[4] = default_spawn_range * mDistribution(mMT);
```

`mDistribution(mMT)` will generate a random value in the range [-1, 1] inclusive because of how we initialized the classes in the constructor of `MyWindow`.

Then we always set the height to the default value:

```
positions[5] = default_start_height;
```

Finally, we use this vector to set the positions of the root Joint:

```
object->getJoint(0)->setPositions(positions);
```

We trust that the root Joint is a FreeJoint with 6 degrees of freedom because of how we constructed all the objects that are going to be thrown at the wall: They were all given a FreeJoint between the world and the root BodyNode.

Lesson 3b: Set the object’ s name

Every object in the world is required to have a non-empty unique name. Just like Joint names in a Skeleton, if we pass a Skeleton into a world with a non-unique name, the world will print out a complaint to us and then rename it. So avoid the ugly printout, we’ ll make sure the new object has a unique name ahead of time:

```
object->setName(object->getName()+std::to_string(mSkelCount++));
```

Lesson 3c: Add the object to the world without collisions

Before we add the Skeleton to the world, we want to make sure that it isn’ t actually placed inside of something accidentally. If an object in a simulation starts off inside of another object, it can result in extremely non-physical simulations, perhaps even breaking the simulation entirely. We can access the world’ s collision detector directly to check make sure the new object is collision-free:

```
auto collisionEngine
  = mWorld->getConstraintSolver()->getCollisionDetector();
```

Now we shouldn’ t be surprised if the *other* objects are in collision with each other, so we’ ll need to check whether our new object overlaps with any existing objects. First, we use the collision engine to create a group which contains our object. Then, we get a group containing the existing objects in the world and use it to check for collisions.

```
auto newGroup = collisionEngine->createCollisionGroup(object.get());
auto collisionGroup = mWorld->getConstraintSolver()->getCollisionGroup();

dart::collision::CollisionOption option;
dart::collision::CollisionResult result;
bool collision = collisionGroup->collide(newGroup.get(), option, &result);
```

If the new skeleton doesn’ t overlap an existing object, we can add it to the world without any complaints:

```
if (!collision)
{
  mWorld->addSkeleton(object);
}
else
{
  std::cout << "The new object spawned in a collision. "
    << "It will not be added to the world." << std::endl;
  return false;
}
```

Of course we should also print out a message so that user understands why we didn’ t throw a new object.

Lesson 3d: Creating reference frames

DART has a unique feature that we call Frame Semantics. The Frame Semantics of DART allow you to create reference frames and use them to get and set data relative to arbitrary frames. There are two crucial Frame types currently used in DART: `BodyNode`s and `SimpleFrame`s.

The BodyNode class does not allow you to explicitly set its transform, velocity, or acceleration properties, because those are all strictly functions of the degrees of freedom that the BodyNode depends on. Because of this, the BodyNode is not a very convenient class if you want to create an arbitrary frame of reference. Instead, DART offers the `SimpleFrame` class which gives you the freedom of arbitarily attaching it to any parent Frame and setting its transform, velocity, and acceleration to whatever you' d like. This makes SimpleFrame useful for specifying arbitrary reference frames.

We' re going to set up a couple SimpleFrames and use them to easily specify the velocity properties that we want the Skeleton to have. First, we' ll place a SimpleFrame at the Skeleton' s center of mass:

```
math::Isometry3d centerTf(math::Isometry3d::Identity());
centerTf.translation() = object->getCOM();
SimpleFrame center(Frame::World(), "center", centerTf);
```

Calling `object->getCOM()` will tell us the center of mass location with respect to the World Frame. We use that to set the translation of the SimpleFrame' s relative transform so that the origin of the SimpleFrame will be located at the object' s center of mass.

Now we' ll set what we want the object' s angular and linear speeds to be:

```
double angle = default_launch_angle;
double speed = default_start_v;
double angular_speed = default_start_w;
if(mRandomize)
{
  angle = (mDistribution(mMT) + 1.0)/2.0 *
      (maximum_launch_angle - minimum_launch_angle) + minimum_launch_angle;

  speed = (mDistribution(mMT) + 1.0)/2.0 *
      (maximum_start_v - minimum_start_v) + minimum_start_v;

  angular_speed = mDistribution(mMT) * maximum_start_w;
}
```

We just use the default values unless randomization is turned on.

Now we' ll convert those speeds into directional velocities:

```
math::Vector3d v = speed * math::Vector3d(cos(angle), 0.0, sin(angle));
math::Vector3d w = angular_speed * math::Vector3d::UnitY();
```

And now we' ll use those vectors to set the velocity properties of the SimpleFrame:

```
center.setClassicDerivatives(v, w);
```

The `SimpleFrame::setClassicDerivatives()` allows you to set the classic linear and angular velocities and accelerations of a SimpleFrame with respect to its parent Frame, which in this case is the World Frame. In DART, classic velocity and acceleration vectors are explicitly differentiated from spatial velocity and acceleration vectors. If you are unfamiliar with the term "spatial vector", then you' ll most likely want to work in terms of classic velocity and acceleration.

Now we want to create a new SimpleFrame that will be a child of the previous one:

```
SimpleFrame ref(&center, "root_reference");
```

And we want the origin of this new Frame to line up with the root BodyNode of our object:

```
ref.setRelativeTransform(object->getBodyNode(0)->getTransform(&center));
```

Now we' ll use this reference frame to set the velocity of the root BodyNode. By setting the velocity of the root BodyNode equal to the velocity of this reference frame, we will ensure that the overall velocity of Skeleton' s center of mass is equal to the velocity of the `center` Frame from earlier.

```
object->getJoint(0)->setVelocities(ref.getSpatialVelocity());
```

Note that the FreeJoint uses spatial velocity and spatial acceleration for its degrees of freedom.

Now we' re ready to toss around objects!

Lesson 4: Setting joint spring and damping properties

Find the `setupRing` function. This is where we' ll setup a chain of BodyNodes so that it behaves more like a closed ring.

Lesson 4a: Set the spring and damping coefficients

We' ll want to set the stiffness and damping coefficients of only the DegreesOfFreedom that are **between** two consecutive BodyNodes. The first six degrees of freedom are between the root BodyNode and the World, so we don' t want to change the stiffness of them, or else the object will hover unnaturally in the air. But all the rest of the degrees of freedom should be set:

```
for(size_t i=6; i < ring->getNumDofs(); ++i)
{
  DegreeOfFreedom* dof = ring->getDof(i);
  dof->setSpringStiffness(ring_spring_stiffness);
  dof->setDampingCoefficient(ring_damping_coefficient);
}
```

Lesson 4b: Set the rest positions of the joints

We want to make sure that the ring' s rest position works well for the structure it has. Using basic geometry, we know we can compute the exterior angle on each edge of a polygon like so:

```
size_t numEdges = ring->getNumBodyNodes();
double angle = 2 * dart::math::pi() / numEdges;
```

Now it' s important to remember that the joints we have between the BodyNodes are BallJoints, which use logmaps (a.k.a. angle-axis) to represent their positions. The BallJoint class provides a convenience function for converting rotations into a position vector for a BallJoint. A similar function also exists for EulerJoint and FreeJoint.

```
for(size_t i=1; i < ring->getNumJoints(); ++i)
{
  Joint* joint = ring->getJoint(i);
  math::AngleAxisd rotation(angle, math::Vector3d(0, 1, 0));
  math::Vector3d restPos = BallJoint::convertToPositions(
        math::Matrix3d(rotation));

  // TODO: Set the rest position
}
```

Now we can set the rest positions component-wise:

```
  for(size_t j=0; j<3; ++j)
    joint->setRestPosition(j, restPos[j]);
```

Lesson 4c: Set the Joints to be in their rest positions

Finally, we should set the ring so that all the degrees of freedom (past the root BodyNode) start out in their rest positions:

```
for(size_t i=6; i < ring->getNumDofs(); ++i)
{
  DegreeOfFreedom* dof = ring->getDof(i);
  dof->setPosition(dof->getRestPosition());
}
```

Lesson 5: Create a closed kinematic chain

Find the `addRing` function in `MyWindow`. In here, we' ll want to create a dynamic constraint that attaches the first and last BodyNodes of the chain together by a BallJoint-style constraint.

First we' ll grab the BodyNodes that we care about:

```
BodyNode* head = ring->getBodyNode(0);
BodyNode* tail = ring->getBodyNode(ring->getNumBodyNodes()-1);
```

Now we want to compute the offset where the BallJoint constraint should be located:

```
math::Vector3d offset = math::Vector3d(0, 0, default_shape_height / 2.0);
offset = tail->getWorldTransform() * offset;
```

The offset will be located half the default height up from the center of the tail BodyNode.

Now we have everything we need to construct the constraint:

```
auto constraint = std::make_shared<dart::dynamics::BallJointConstraint>(
      head, tail, offset);
```

In order for the constraint to work, we' ll need to add it to the world' s constraint solver:

```
mWorld->getConstraintSolver()->addConstraint(constraint);
```

And in order to properly clean up the constraint when removing BodyNodes, we' ll want to add it to our list of constraints:

```
mJointConstraints.push_back(constraint);
```

And that' s it! You' re ready to run the full tutorialCollisions application!

When running the application, keep in mind that the dynamics of collisions are finnicky, so you may see some unstable and even completely non-physical behavior. If the application freezes, you may need to force quit out of it.

5.5.3 Dominoes

Overview

This tutorial will demonstrate some of the more advanced features of DART' s dynamics API which allow you to write robust controllers that work for real dynamic systems, such as robotic manipulators. We will show you how to:

- Clone Skeletons
- Load a URDF

- Write a stable PD controller w/ gravity and coriolis compensation
- Write an operational space controller

Please reference the source code in **tutorialDominoes.cpp** and **tutorialDominoes-Finished.cpp**.

Lesson 1: Cloning Skeletons

There are often times where you might want to create an exact replica of an existing Skeleton. DART offers cloning functionality that allows you to do this very easily.

Lesson 1a: Create a new domino

Creating a new domino is straightforward. Find the function `attemptToCreateDomino` in the `MyWindow` class. The class has a member called `mFirstDomino` which is the original domino created when the program starts up. To make a new one, we can just clone it:

```
SkeletonPtr newDomino = mFirstDomino->clone();
```

But keep in mind that every Skeleton that gets added to a world requires its own unique name. Creating a clone will keep the original name, so we should we give the new copy its own name:

```
newDomino->setName("domino #" + std::to_string(mDominoes.size() + 1));
```

So the easy part is finished, but now we need to get the domino to the correct position. First, let’ s grab the last domino that was placed in the environment:

```
const SkeletonPtr& lastDomino = mDominoes.size() > 0 ?
      mDominoes.back() : mFirstDomino;
```

Now we should compute what we want its position to be. The `MyWindow` class keeps a member called `mTotalAngle` which tracks how much the line of dominoes has turned so far. We’ ll use that to figure out what translational offset the new domino should have from the last domino:

```
math::Vector3d dx = default_distance * math::Vector3d(
      cos(mTotalAngle), sin(mTotalAngle), 0.0);
```

And now we can compute the total position of the new domino. First, we’ ll copy the positions of the last domino:

```
math::Vector6d x = lastDomino->getPositions();
```

And then we’ ll add the translational offset to it:

```
x.tail<3>() += dx;
```

Remember that the domino’ s root joint is a FreeJoint which has six degrees of freedom: the first three are for orientation and last three are for translation.

Finally, we should add on the change in angle for the new domino:

```
x[2] = mTotalAngle + angle;
```

Be sure to uncomment the `angle` argument of the function.

Now we can use `x` to set the positions of the domino:

```
newDomino->setPositions(x);
```

The root FreeJoint is the only joint in the domino’ s Skeleton, so we can just use the `Skeleton::setPositions` function to set it.

Now we’ ll add the Skeleton to the world:

```
mWorld->addSkeleton(newDomino);
```

Lesson 1b: Make sure no dominoes are in collision

Similar to **Lesson 3** of the **Collisions** tutorial, we’ ll want to make sure that the newly inserted Skeleton is not starting out in collision with anything, because this could make for a very ugly (perhaps even broken) simulation.

First, we’ ll tell the world to compute collisions:

```
dart::collision::CollisionDetector* detector =
    mWorld->getConstraintSolver()->getCollisionDetector();
detector->detectCollision(true, true);
```

Now we’ ll look through and see if any dominoes are in collision with anything besides the floor. We ignore collisions with the floor because, mathematically speaking, if they are in contact with the floor then they register as being in collision. But we want the dominoes to be in contact with the floor, so this is okay.

```
bool dominoCollision = false;
size_t collisionCount = detector->getNumContacts();
for(size_t i = 0; i < collisionCount; ++i)
{
  // If neither of the colliding BodyNodes belongs to the floor, then we
  // know the new domino is in contact with something it shouldn't be
  const dart::collision::Contact& contact = detector->getContact(i);
  if(contact.bodyNode1.lock()->getSkeleton() != mFloor
     && contact.bodyNode2.lock()->getSkeleton() != mFloor)
  {
    dominoCollision = true;
    break;
  }
}
```

The only object that could possibly have collided with something else is the new domino, because we don’ t allow the application to create new things except for the dominoes. So if this registered as true, then we should take the new domino out of the world:

```
if(dominoCollision)
{
  // Remove the new domino, because it is penetrating an existing one
  mWorld->removeSkeleton(newDomino);
}
```

Otherwise, if the new domino is in an okay position, we should add it to the history:

```
else
{
  // Record the latest domino addition
  mAngles.push_back(angle);
```

(continues on next page)

(continued from previous page)

```
  mDominoes.push_back(newDomino);
  mTotalAngle += angle;
}
```

Lesson 1c: Delete the last domino added

Ordinarily, removing a Skeleton from a scene is just a matter of calling the `World::removeSkeleton` function, but we have a little bit of bookkeeping to take care of for our particular application. First, we should check whether there are any dominoes to actually remove:

```
if(mDominoes.size() > 0)
{
  // TODO: Remove Skeleton
}
```

Then we should grab the last domino in the history, remove it from the history, and then take it out of the world:

```
SkeletonPtr lastDomino = mDominoes.back();
mDominoes.pop_back();
mWorld->removeSkeleton(lastDomino);
```

The `SkeletonPtr` class is really a `std::shared_ptr<Skeleton>` so we don’ t need to worry about ever calling `delete` on it. Instead, its resources will be freed when `lastDomino` goes out of scope.

We should also make sure to do the bookkeeping for the angles:

```
mTotalAngle -= mAngles.back();
mAngles.pop_back();
```

Now we can add and remove dominoes from the scene. Feel free to give it a try.

Lesson 1d: Apply a force to the first domino

But just setting up dominoes isn’ t much fun without being able to knock them down. We can quickly and easily knock down the dominoes by magically applying a force to the first one. In the `timeStepping` function of `MyWindow` there is a label for **Lesson 1d**. This spot will get visited whenever the user presses ‘f’ , so we’ ll apply an external force to the first domino here:

```
math::Vector3d force = default_push_force * math::Vector3d::UnitX();
math::Vector3d location =
    default_domino_height / 2.0 * math::Vector3d::UnitZ();
mFirstDomino->getBodyNode(0)->addExtForce(force, location);
```

Lesson 2: Loading and controlling a robotic manipulator

Striking something with a magical force is convenient, but not very believable. Instead, let’ s load a robotic manipulator and have it push over the first domino.

Lesson 2a: Load a URDF file

Our manipulator is going to be loaded from a URDF file. URDF files are loaded by the `dart::io::DartLoader` class (pending upcoming changes to DART’ s loading system). First, create a loader:

```
dart::io::DartLoader loader;
```

Note that many URDF files use ROS’ s `package:` scheme to specify the locations of the resources that need to be loaded. We won’ t be using this in our example, but in general you should use the function `DartLoader::addPackageDirectory` to specify the locations of these packages, because DART does not have the same package resolving abilities of ROS.

Now we’ ll have `loader` parse the file into a Skeleton:

```
SkeletonPtr manipulator =
    loader.parseSkeleton(DART_DATA_LOCAL_PATH"urdf/KR5/KR5 sixx R650.urdf");
```

And we should give the Skeleton a convenient name:

```
manipulator->setName("manipulator");
```

Now we’ ll want to initialize the location and configuration of the manipulator. Experimentation has demonstrated that the following setup is good for our purposes:

```
// Position its base in a reasonable way
math::Isometry3d tf = math::Isometry3d::Identity();
tf.translation() = math::Vector3d(-0.65, 0.0, 0.0);
manipulator->getJoint(0)->setTransformFromParentBodyNode(tf);

// Get it into a useful configuration
manipulator->getDof(1)->setPosition(140.0 * M_PI / 180.0);
manipulator->getDof(2)->setPosition(-140.0 * M_PI / 180.0);
```

And lastly, be sure to return the Skeleton that we loaded rather than the dummy Skeleton that was originally there:

```
return manipulator;
```

Feel free to load up the application to see the manipulator in the scene, although all it will be able to do is collapse pitifully onto the floor.

Lesson 2b: Grab the desired joint angles

To make the manipulator actually useful, we’ ll want to have the `Controller` control its joint forces. For it to do that, the `Controller` class will need to be informed of what we want the manipulator’ s joint angles to be. This is easily done in the constructor of the `Controller` class:

```
mQDesired = mManipulator->getPositions();
```

The function `Skeleton::getPositions` will get all the generalized coordinate positions of all the joints in the Skeleton, stacked in a single vector. These Skeleton API functions are useful when commanding or controlling an entire Skeleton with a single mathematical expression.

Lesson 2c: Write a stable PD controller for the manipulator

Now that we know what configuration we want the manipulator to hold, we can write a PD controller that keeps them in place. Find the function `setPDForces` in the `Controller` class.

First, we' ll grab the current positions and velocities:

```
math::VectorXd q = mManipulator->getPositions();
math::VectorXd dq = mManipulator->getVelocities();
```

Additionally, we' ll integrate the position forward by one timestep:

```
q += dq * mManipulator->getTimeStep();
```

This is not necessary for writing a regular PD controller, but instead this is to write a "stable PD" controller which has some better numerical stability properties than an ordinary discrete PD controller. You can try running with and without this line to see what effect it has on the stability.

Now we' ll compute our joint position error:

```
math::VectorXd q_err = mQDesired - q;
```

And our joint velocity error, assuming our desired joint velocity is zero:

```
math::VectorXd dq_err = -dq;
```

Now we can grab our mass matrix, which we will use to scale our force terms:

```
const math::MatrixXd& M = mManipulator->getMassMatrix();
```

And then combine all this into a PD controller that computes forces to minimize our error:

```
mForces = M * (mKpPD * q_err + mKdPD * dq_err);
```

Now we' re ready to set these forces on the manipulator:

```
mManipulator->setForces(mForces);
```

Feel free to give this PD controller a try to see how effective it is.

Lesson 2d: Compensate for gravity and Coriolis forces

One of the key features of DART is the ability to easily compute the gravity and Coriolis forces, allowing you to write much higher quality controllers than you would be able to otherwise. This is easily done like so:

```
const math::VectorXd& Cg = mManipulator->getCoriolisAndGravityForces();
```

And now we can update our control law by just slapping this term onto the end of the equation:

```
mForces = M * (mKpPD * q_err + mKdPD * dq_err) + Cg;
```

Give this new PD controller a try to see how its performance compares to the one without compensation

Lesson 3: Writing an operational space controller

While PD controllers are simply and handy, operational space controllers can be much more elegant and useful for performing tasks. Operational space controllers allow us to unify geometric tasks (like getting the end effector to a particular spot) and dynamics tasks (like applying a certain force with the end effector) all while remaining stable and smooth.

Lesson 3a: Set up the information needed for an OS controller

Unlike PD controllers, an operational space controller needs more information than just desired joint angles.

First, we' ll grab the last BodyNode on the manipulator and treat it as an end effector:

```
mEndEffector = mManipulator->getBodyNode(mManipulator->getNumBodyNodes() - 1);
```

But we don' t want to use the origin of the BodyNode frame as the origin of our Operational Space controller; instead we want to use a slight offset, to get to the tool area of the last BodyNode:

```
mOffset = default_endeffector_offset * math::Vector3d::UnitX();
```

Also, our target will be the spot on top of the first domino, so we' ll create a reference frame and place it there. First, create the SimpleFrame:

```
mTarget = std::make_shared<SimpleFrame>(Frame::World(), "target");
```

Then compute the transform needed to get from the center of the domino to the top of the domino:

```
math::Isometry3d target_offset(math::Isometry3d::Identity());
target_offset.translation() =
    default_domino_height / 2.0 * math::Vector3d::UnitZ();
```

And then we should rotate the target' s coordinate frame to make sure that lines up with the end effector' s reference frame, otherwise the manipulator might try to push on the domino from a very strange angle:

```
target_offset.linear() =
    mEndEffector->getTransform(domino->getBodyNode(0)).linear();
```

Now we' ll set the target so that it has a transform of `target_offset` with respect to the frame of the domino:

```
mTarget->setTransform(target_offset, domino->getBodyNode(0));
```

And this gives us all the information we need to write an Operational Space controller.

Lesson 3b: Computing forces for OS Controller

Find the function `setOperationalSpaceForces()`. This is where we' ll compute the forces for our operational space controller.

One of the key ingredients in an operational space controller is the mass matrix. We can get this easily, just like we did for the PD controller:

```
const math::MatrixXd& M = mManipulator->getMassMatrix();
```

Next we' ll want the Jacobian of the tool offset in the end effector. We can get it easily with this function:

```
Jacobian J = mEndEffector->getWorldJacobian(mOffset);
```

But operational space controllers typically use the Moore-Penrose pseudoinverse of the Jacobian rather than the Jacobian itself. There are many ways to compute the pseudoinverse of the Jacobian, but a simple way is like this:

```
math::MatrixXd pinv_J = J.transpose() * (J * J.transpose()
                       + 0.0025 * math::Matrix6d::Identity()).inverse();
```

Note that this pseudoinverse is also damped so that it behaves better around singularities. This is method for computing the pseudoinverse is not very efficient in terms of the number of mathematical operations it performs, but it is plenty fast for our application. Consider using methods based on Singular Value Decomposition if you need to compute the pseudoinverse as fast as possible.

Next we' ll want the time derivative of the Jacobian, as well as its pseudoinverse:

```
// Compute the Jacobian time derivative
Jacobian dJ = mEndEffector->getJacobianClassicDeriv(mOffset);

// Comptue the pseudo-inverse of the Jacobian time derivative
math::MatrixXd pinv_dJ = dJ.transpose() * (dJ * dJ.transpose()
                        + 0.0025 * math::Matrix6d::Identity()).inverse();
```

Notice that here we' re compute the **classic** derivative, which means the derivative of the Jacobian with respect to time in classical coordinates rather than spatial coordinates. If you use spatial vector arithmetic, then you' ll want to use `BodyNode::getJacobianSpatialDeriv` instead.

Now we can compute the linear components of error:

```
math::Vector6d e;
e.tail<3>() = mTarget->getWorldTransform().translation()
            - mEndEffector->getWorldTransform() * mOffset;
```

And then the angular components of error:

```
math::AngleAxisd aa(mTarget->getTransform(mEndEffector).linear());
e.head<3>() = aa.angle() * aa.axis();
```

Then the time derivative of error, assuming our desired velocity is zero:

```
math::Vector6d de = -mEndEffector->getSpatialVelocity(
      mOffset, mTarget.get(), Frame::World());
```

Like with the PD controller, we can mix in terms to compensate for gravity and Coriolis forces:

```
const math::VectorXd& Cg = mManipulator->getCoriolisAndGravityForces();
```

The gains for the operational space controller need to be in matrix form, but we' re storing the gains as scalars, so we' ll need to conver them:

```
math::Matrix6d Kp = mKpOS * math::Matrix6d::Identity();

size_t dofs = mManipulator->getNumDofs();
math::MatrixXd Kd = mKdOS * math::MatrixXd::Identity(dofs, dofs);
```

And we' ll need to compute the joint forces needed to achieve our desired end effector force. This is easily done using the Jacobian transpose:

```
math::Vector6d fDesired = math::Vector6d::Zero();
fDesired[3] = default_push_force;
math::VectorXd f = J.transpose() * fDesired;
```

And now we can mix everything together into the single control law:

```
math::VectorXd dq = mManipulator->getVelocities();
mForces = M * (pinv_J * Kp * de + pinv_dJ * Kp * e)
          - Kd * dq + Kd * pinv_J * Kp * e + Cg + f;
```

Then don’ t forget to pass the forces into the manipulator:

```
mManipulator->setForces(mForces);
```

Now you’ re ready to try out the full dominoes app!

5.5.4 Multi Pendulum

Overview

This tutorial will demonstrate some basic interaction with DART’ s dynamics API during simulation. This will show you how to:

- Create a basic program to simulate a dynamic system
- Change the colors of shapes
- Add/remove shapes from visualization
- Apply internal forces in the joints
- Apply external forces to the bodies
- Alter the implicit spring and damping properties of joints
- Add/remove dynamic constraints

Please reference the source code in **tutorialMultiPendulum.cpp** and **tutorialMultiPendulum-Finished.cpp**.

Lesson 0: Simulate a passive multi-pendulum

This is a warmup lesson that demonstrates how to set up a simulation program in DART. The example we will use throughout this tutorial is a pendulum with five rigid bodies swinging under gravity. DART allows the user to build various articulated rigid/soft body systems from scratch. It also loads models in URDF, SDF, and SKEL formats as demonstrated in the later tutorials.

In DART, an articulated dynamics model is represented by a `Skeleton`. In the `main` function, we first create an empty skeleton named *pendulum*.

```
SkeletonPtr pendulum = Skeleton::create("pendulum");
```

A Skeleton is a structure that consists of `BodyNodes` (bodies) which are connected by `Joints`. 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. In the function `makeRootBody`, we create a pair of a `BallJoint` and a BodyNode, and attach this pair to the currently empty pendulum skeleton.

```
BodyNodePtr bn = pendulum->createJointAndBodyNodePair<BallJoint>(
      nullptr, properties, BodyNode::AspectProperties(name)).second;
```

Note that the first parameters is a nullptr, which indicates that this new BodyNode is the root of the pendulum. If we wish to append the new BodyNode to an existing BodyNode in the pendulum, we can do so by passing the pointer of the existing BodyNode as the first parameter. In fact, this is how we add more BodyNodes to the pendulum in the function `addBody`:

```
BodyNodePtr bn = pendulum->createJointAndBodyNodePair<RevoluteJoint>(
      parent, properties, BodyNode::AspectProperties(name)).second;
```

The simplest way to set up a simulation program in DART is to use `SimWindow` class. A SimWindow owns an instance of `World` and simulates all the Skeletons in the World. In this example, we create a World with the pendulum skeleton in it, and assign the World to an instance of `MyWindow`, a subclass derived from SimWindow.

```
WorldPtr world(new World);
world->addSkeleton(pendulum);
MyWindow window(world);
```

Every single time step, the `MyWindow::timeStepping` function will be called and the state of the World will be simulated. The user can override the default timeStepping function to customize the simulation routine. For example, one can incorporate sensors, actuators, or user interaction in the forward simulation.

Lesson 1: Change shapes and applying forces

We have a pendulum with five bodies, and we want to be able to apply forces to them during simulation. Additionally, we want to visualize these forces so we can more easily interpret what is happening in the simulation. For this reason, we' ll discuss visualizing and forces at the same time.

Lesson 1a: Reset everything to default appearance

At each step, we' ll want to make sure that everything starts out with its default appearance. The default is for everything to be blue and there not to be any arrow attached to any body.

Find the function named `timeStepping` in the `MyWindow` class. The top of this function is where we will want to reset everything to the default appearance.

Each BodyNode contains visualization `Shapes` that will be rendered during simulation. In our case, each BodyNode has two shapes:

- One shape to visualize the parent joint
- One shape to visualize the body

The default appearance for everything is to be colored blue, so we' ll want to iterate through these two Shapes in each BodyNode, setting their colors to blue.

```
for(size_t i = 0; i < mPendulum->getNumBodyNodes(); ++i)
{
  BodyNode* bn = mPendulum->getBodyNode(i);
  auto visualShapeNodes = bn->getShapeNodesWith<VisualAspect>();
  for(std::size_t j = 0; j < 2; ++j)
  {
    visualShapeNodes[j]->getVisualAspect()->setColor(dart::Color::Blue());
  }
```

(continues on next page)

(continued from previous page)

```
  // TODO: Remove any arrows
}
```

Additionally, there is the possibility that some BodyNodes will have an arrow shape attached if the user had been applying an external body force to it. By default, this arrow should not be attached, so in the outer for-loop, we should check for arrows and remove them:

```
if(visualShapeNodes.size() == 3)
{
  visualShapeNodes[2]->remove();
}
```

Now everything will be reset to the default appearance.

Lesson 1b: Apply joint torques based on user input

The `MyWindow` class in this tutorial has a variable called `mForceCountDown` which is a `std::vector<int>` whose entries get set to a value of `default_countdown` each time the user presses a number key. If an entry in `mForceCountDown` is greater than zero, then that implies that the user wants a force to be applied for that entry.

There are two ways that forces can be applied:

- As an internal joint force
- As an external body force

First we' ll consider applying a Joint force. Inside the for-loop that goes through each `DegreeOfFreedom` using `getNumDofs()`, there is an if-statement for `mForceCountDown`. In that if-statement, we' ll grab the relevant DegreeOfFreedom and set its generalized (joint) force:

```
DegreeOfFreedom* dof = mPendulum->getDof(i);
dof->setForce( mPositiveSign? default_torque : -default_torque );
```

The `mPositiveSign` boolean gets toggled when the user presses the minus sign '-' key. We use this boolean to decide whether the applied force should be positive or negative.

Now we' ll want to visualize the fact that a Joint force is being applied. We' ll do this by highlighting the joint with the color red. First we' ll grab the Shape that corresponds to this Joint:

```
BodyNode* bn = dof->getChildBodyNode();
auto shapeNodes = bn->getShapeNodesWith<VisualAspect>();
```

Because of the way the pendulum bodies were constructed, we trust that the zeroth indexed visualization shape will be the shape that depicts the joint. So now we will color it red:

```
shapeNodes[0]->getVisualAspect()->setColor(dart::Color::Red());
```

Lesson 1c: Apply body forces based on user input

If mBodyForce is true, we' ll want to apply an external force to the body instead of an internal force in the joint. First, inside the for-loop that iterates through each `BodyNode` using `getNumBodyNodes()`, there is an if-statement for `mForceCountDown`. In that if-statement, we' ll grab the relevant BodyNode:

```
BodyNode* bn = mPendulum->getBodyNode(i);
```

Now we' ll create an `math::Vector3d` that describes the force and another one that describes the location for that force. An `math::Vector3d` is the Eigen C++ library' s version of a three-dimensional mathematical vector. Note that the `d` at the end of the name stands for `double`, not for "dimension". An math::Vector3f would be a three-dimensional vector of floats, and an math::Vector3i would be a three-dimensional vector of integers.

```
math::Vector3d force = default_force * math::Vector3d::UnitX();
math::Vector3d location(-default_width / 2.0, 0.0, default_height / 2.0);
```

The force will have a magnitude of `default_force` and it will point in the positive x-direction. The location of the force will be in the center of the negative x side of the body, as if a finger on the negative side is pushing the body in the positive direction. However, we need to account for sign changes:

```
if(!mPositiveSign)
{
  force = -force;
  location[0] = -location[0];
}
```

That will flip the signs whenever the user is requesting a negative force.

Now we can add the external force:

```
bn->addExtForce(force, location, true, true);
```

The two `true` booleans at the end are indicating to DART that both the force and the location vectors are being expressed with respect to the body frame.

Now we' ll want to visualize the force being applied to the body. First, we' ll grab the Shape for the body and color it red:

```
auto shapeNodes = bn->getShapeNodesWith<VisualAspect>();
shapeNodes[1]->getVisualAspect()->setColor(dart::Color::Red());
```

Last time we grabbed the 0-index visualization shape, because we trusted that it was the shape that represented the parent Joint. This time we' re grabbing the 1-index visualization shape, because we trust that it is the shape for the body.

Now we' ll want to add an arrow to the visualization shapes of the body to represent the applied force. The `MyWindow` class already provides the arrow shape; we just need to add it:

```
bn->createShapeNodeWith<VisualAspect>(mArrow);
```

Lesson 2: Set spring and damping properties for joints

DART allows Joints to have implicit spring and damping properties. By default, these properties are zeroed out, so a joint will only exhibit the forces that are given to it by the `Joint::setForces` function. However, you can give a non-zero spring coefficient to a joint so that it behaves according to Hooke' s Law, and you can give a non-zero damping coefficient to a joint which will result in linear damping. These forces are computed using implicit methods in order to improve numerical stability.

Lesson 2a: Set joint spring rest position

First let' s see how to get and set the rest positions.

Find the function named `changeRestPosition` in the `MyWindow` class. This function will be called whenever the user presses the 'q' or 'a' button. We want those buttons to curl and uncurl the rest positions for the pendulum. To start, we' ll go through all the generalized coordinates and change their rest positions by `delta`:

```
for(size_t i = 0; i < mPendulum->getNumDofs(); ++i)
{
  DegreeOfFreedom* dof = mPendulum->getDof(i);
  double q0 = dof->getRestPosition() + delta;

  dof->setRestPosition(q0);
}
```

However, it' s important to note that the system can become somewhat unstable if we allow it to curl up too much, so let' s put a limit on the magnitude of the rest angle. Right before `dof->setRestPosition(q0);` we can put:

```
if(std::abs(q0) > 90.0 * M_PI / 180.0)
  q0 = (q0 > 0)? (90.0 * M_PI / 180.0) : -(90.0 * M_PI / 180.0);
```

And there' s one last thing to consider: the first joint of the pendulum is a BallJoint. BallJoints have three degrees of freedom, which means if we alter the rest positions of *all* of the pendulum' s degrees of freedom, then the pendulum will end up curling out of the x-z plane. You can allow this to happen if you want, or you can prevent it from happening by zeroing out the rest positions of the BallJoint' s other two degrees of freedom:

```
mPendulum->getDof(0)->setRestPosition(0.0);
mPendulum->getDof(2)->setRestPosition(0.0);
```

Lesson 2b: Set joint spring stiffness

Changing the rest position does not accomplish anything without having any spring stiffness. We can change the spring stiffness as follows:

```
for(size_t i = 0; i < mPendulum->getNumDofs(); ++i)
{
  DegreeOfFreedom* dof = mPendulum->getDof(i);
  double stiffness = dof->getSpringStiffness() + delta;
  dof->setSpringStiffness(stiffness);
}
```

However, it' s important to realize that if the spring stiffness were ever to become negative, we would get some very nasty explosive behavior. It' s also a bad idea to just trust the user to avoid decrementing it into being negative. So before the line `dof->setSpringStiffness(stiffness);` you' ll want to put:

```
if(stiffness < 0.0)
  stiffness = 0.0;
```

Lesson 2c: Set joint damping

Joint damping can be thought of as friction inside the joint actuator. It applies a resistive force to the joint which is proportional to the generalized velocities of the joint. This draws energy out of the system and generally results in more stable behavior.

The API for getting and setting the damping is just like the API for stiffness:

```
for(size_t i = 0; i < mPendulum->getNumDofs(); ++i)
{
  DegreeOfFreedom* dof = mPendulum->getDof(i);
  double damping = dof->getDampingCoefficient() + delta;
  if(damping < 0.0)
    damping = 0.0;
  dof->setDampingCoefficient(damping);
}
```

Again, we want to make sure that the damping coefficient is never negative. In fact, a negative damping coefficient would be far more harmful than a negative stiffness coefficient.

Lesson 3: Add and remove dynamic constraints

Dynamic constraints in DART allow you to attach two BodyNodes together according to a selection of a few different Joint-style constraints. This allows you to create closed loop constraints, which is not possible using standard Joints. You can also create a dynamic constraint that attaches a BodyNode to the World instead of to another BodyNode.

In our case, we want to attach the last BodyNode to the World with a BallJoint style constraint whenever the function `addConstraint()` gets called. First, let' s grab the last BodyNode in the pendulum:

```
BodyNode* tip  = mPendulum->getBodyNode(mPendulum->getNumBodyNodes() - 1);
```

Now we' ll want to compute the location that the constraint should have. We want to connect the very end of the tip to the world, so the location would be:

```
math::Vector3d location =
    tip->getTransform() * math::Vector3d(0.0, 0.0, default_height);
```

Now we can create the BallJointConstraint:

```
mBallConstraint =
    std::make_shared<dart::dynamics::BallJointConstraint>(tip, location);
```

And then add it to the world:

```
mWorld->getConstraintSolver()->addConstraint(mBallConstraint);
```

Now we also want to be able to remove this constraint. In the function `removeConstraint()`, we can put the following code:

```
mWorld->getConstraintSolver()->removeConstraint(mBallConstraint);
mBallConstraint = nullptr;
```

Setting mBallConstraint to a nullptr will allow its smart pointer to delete it.

Now you are ready to run the demo!

5.6 Build

5.6.1 Building DART

This guide describes how to build DART, a C++ library for robotics and motion planning, using CMake. DART also has Python bindings, called dartpy, which will be covered in a separate section.

Supported Environments

DART is supported on the following operating systems and compilers:

Operating System	Compiler
Ubuntu 22.04 or later	GCC 11.2 or later
Windows 2022 or later	Visual Studio 2022
macOS 13 or later	Clang 13 or later

Prerequisites

Before you can build DART, you' ll need to install the required and optional dependencies. The required dependencies are the minimum set of dependencies needed to build DART, while the optional dependencies enable additional features in DART.

The steps for installing dependencies may vary depending on your operating system and package manager. Below, we provide instructions for installing the required and optional dependencies on Ubuntu, macOS, and Windows, as well as some experimental guidance for other platforms.

Note: Please note that the dependencies and installation steps are subject to change, so we encourage you to report any issues you encounter and contribute to keeping the instructions up-to-date for the community. By working together, we can help ensure that the DART documentation is accurate and helpful for everyone who uses it.

Ubuntu

The dependencies for Ubuntu can be installed using the `apt` package manager. The following command will install the required dependencies:

```
$ sudo apt install \
  build-essential cmake pkg-config git libassimp-dev \
  libeigen3-dev libfcl-dev libfmt-dev
```

The following command will install the optional dependencies:

```
$ sudo apt install \
   coinor-libipopt-dev freeglut3-dev libxi-dev libxmu-dev libbullet-dev \
   libtinyxml2-dev liburdfdom-dev liburdfdom-headers-dev \
   libopenscenegraph-dev libnlopt-cxx-dev liboctomap-dev libode-dev \
   libspdlog-dev libyaml-cpp-dev ocl-icd-opencl-dev opencl-headers \
   opencl-clhpp-headers
```

macOS

The dependencies for macOS can be installed using the `brew` package manager. The following command will install the required dependencies:

```
$ brew install assimp cmake eigen fmt fcl
```

The following command will install the optional dependencies:

```
$ brew install bullet freeglut ipopt nlopt octomap ode \
   open-scene-graph --HEAD \
   spdlog tinyxml2 urdfdom yaml-cpp
```

Windows

The dependencies for Windows can be installed using the `vcpkg` package manager. The following command will install the required dependencies:

```
$ vcpkg install --triplet x64-windows assimp eigen3 fcl fmt spdlog
```

The following command will install the optional dependencies:

```
$ vcpkg install --triplet x64-windows \
   assimp eigen3 fcl fmt spdlog bullet3 freeglut glfw3 nlopt ode \
   opencl opengl osg pagmo2 pybind11 tinyxml2 urdfdom yaml-cpp
```

Arch Linux (experimental)

The dependencies for Arch Linux can be installed using the `yay` package manager. The following command will install the required dependencies:

```
$ yay -S assimp cmake eigen fcl fmt
```

The following command will install the optional dependencies:

```
$ yay -S \
   bullet coin-or-ipopt freeglut nlopt octomap ode opencl-clhpp \
   opencl-headers opencl-icd-loader openscenegraph pagmo spdlog tinyxml2 \
   urdfdom pybind11
```

FreeBSD (experimental)

TODO

Dependency Info

Here' s a summary of the dependencies required to build DART (WIP):

Dependency	Required	Type	Min. Version	Notes
CMake	Yes	Build	3.22.1	
Assimp	Yes	Runtime	5.2.2	
Eigen	Yes	Runtime	3.4.0	
FCL	Yes	Runtime	0.7.0	
fmt	Yes	Runtime	8.1.1	
Bullet	No	Runtime	3.06	
Ipopt	No	Runtime	3.11.9	
Octomap	No	Runtime	1.9.7	
ODE	No	Runtime	0.16.2	
Pagmo	No	Runtime	2.18.0	
spdlog	No	Runtime	1.9.2	
tinyxml2	No	Runtime	9.0.0	
urdfdom	No	Runtime	3.0.1	
OpenSceneGraph	No	Runtime	3.6.5	

Clone the DART Repository

To get started with building DART, you' ll need to clone the DART repository. Here' s how to do it:

1. Clone the DART repository by running the following command in your terminal:

```
$ git clone https://github.com/dartsim/dart.git
```

2. (Optional) If you want to build a specific version of DART, you can checkout a specific branch, tag, or commit.

```
$ git checkout -b <branch_or_tag_or_commit>
```

Note: Please note that the DART repository is actively maintained, so there may be changes and updates to the repository over time. To get the latest information, we recommend referring to the DART GitHub repository.

Build Configuration

DART uses CMake as its build system. CMake is a powerful tool that generates build files for a variety of build systems, including Makefiles, Visual Studio projects, and Xcode projects. For more information about available generators, we recommend referring to the CMake documentation.

To configure the build, you' ll need to create a build directory and run CMake from that directory. Here' s how to do it:

1. Create a build directory by running the following command in your terminal:

```
$ mkdir build
```

2. Change into the build directory by running the following command:

```
$ cd build
```

3. Run CMake from the build directory by running the following command:

```
$ cmake ..
```

If you want to configure the build, you can pass additional options to CMake. For example, you can specify the build type by passing the `-DCMAKE_BUILD_TYPE` option. DART provides a number of CMake options that allow you to customize the build process. Here are some of the most important options:

Option	Default Value	Description
CMAKE_BUILD_TYPE	Release	Specifies the build type.
DART_ENABLE_SIMD	ON	Enables use of SIMD instructions.
TODO		

Note: This list of options may not be exhaustive or up-to-date. Please refer to the main CMakeLists.txt file in the DART repository to confirm the list of available options. If you find any discrepancies or errors, please consider submitting a pull request to update this document.

Here are some example commands that you can use to configure the build on different platforms with different generators:

```
$ cmake .. -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release
$ cmake .. -G "Visual Studio 15 2017" -A x64 -DCMAKE_BUILD_TYPE=Release
$ cmake .. -G "Xcode" -DCMAKE_BUILD_TYPE=Release
```

Building DART from Command Line

Whether or not you configured the build for IDEs, you can still build DART from the command line using CMake' s unified build commands.

To build DART from the command line, you' ll need to run the build command from the build directory. Here' s how to do it:

1. Change into the build directory by running the following command:

```
$ cd build
```

2. Run the build command by running the following command:

```
$ cmake --build . [--target <target> [, <target2>, ...]] [-j<num_core>]
```

DART provides a number of CMake targets that you can use to build different parts of the project. Here are some of the most important targets:

- `ALL`: Builds all the targets in the project, including building tests, examples, tutorials, and running tests.
- `all`: Builds core targets without tests, examples, and tutorials.
- `tests`: Builds all the tests.

- `test`: Runs tests (need to build tests first).
- `tests_and_run`: Builds and runs tests.
- `examples`: Builds all the examples.
- `tutorials`: Builds all the tutorials.
- `benchmarks`: Builds all the benchmarks.
- `view_docs`: Builds the documentation and opens it in a web browser.
- `install`: Installs the project.
- `dartpy`: Builds the Python bindings (it’ s encouraged to build using pip instead).
- `pytest`: Runs Python tests (building tests if necessary).
- `coverage`: Runs tests and generates a coverage report.
- `coverage_html`: Runs tests and generates an HTML coverage report.
- `coverage_view`: Runs tests, generates an HTML coverage report, and opens it in a web browser.

Note: Please note that this list of targets may not be exhaustive or up-to-date. To confirm the full list of available targets, we recommend referring to the main CMakeLists.txt file in the DART repository. If you find any discrepancies or errors, we encourage you to submit a pull request to update this document and help keep the documentation up-to-date for the community.

Building DART from IDEs

If you configured the build for IDEs, you can build DART from the IDEs. This section doesn’ t cover how to build DART from IDEs. Please refer to the IDEs documentation for more information. However, it’ s always to welcome to submit a pull request to update this document with instructions for your favorite IDE!

5.6.2 Building dartpy

In general, building dartpy from source is not necessary. The easiest way to install dartpy is to use pip:

```
$ pip install dartpy -U
```

TODO

5.7 Contributing to DART

DART is a collaborative project, and we welcome contributions from anyone who is interested in helping make the project better. Whether you’ re interested in fixing bugs, adding new features, improving documentation, or something else entirely, we appreciate your contributions.

5.7.1 Getting Help

If you' re looking to contribute to DART but need help getting started, there are many resources available to you. Here are some places you can turn to for help:

- DART documentation
- Issue tracker on GitHub
- Feature requests on GitHub
- Community showcases on GitHub

Of course, you are welcome to reach out to us directly if you need further assistance. We are always happy to help!

5.7.2 Credits

DART was initially created by C. Karen Liu and Mike Stilman in 2011 at Georgia Tech, and has since evolved with the contributions of various institutions and individuals. We would like to extend our thanks and appreciation to the following institutions and individuals who have contributed to DART:

- Humanoid Lab, Georgia Tech Research Corporation
- Personal Robotics Lab, Carnegie Mellon University
- Graphics Lab, Georgia Tech Research Corporation
- Personal Robotics Lab, University of Washington
- Open Source Robotics Foundation
- The Movement Lab, Stanford University

and:

- C. Karen Liu: project creator, multibody dynamics, constraint resolution, tutorials
- Mike Stilman: project creator
- Siddhartha S. Srinivasa: project advisor
- Jeongseok Lee: project director, multibody dynamics, constraint resolution, collision detection, tutorials
- Michael X. Grey: project director, extensive API improvements, inverse kinematics, gui::osg, tutorials
- Tobias Kunz: former project director, motion planner
- Sumit Jain: multibody dynamics
- Yuting Ye: multibody dynamics, GUI
- Michael Koval: uri, resource retriever, bug fixes
- Ana C. Huamán Quispe: urdf parser
- Chen Tang: collision detection
- Matthew Dutton: build and bug fixes
- Eric Huang: build and bug fixes
- Pushkar Kolhe: early DART build system design
- Saul Reynolds-Haertle: examples, bug fixes
- Arash Rouhani: build fixes

- Kristin Siu: integrators, bug fixes
- Steven Peters: build improvements and fixes
- Can Erdogan: planning, examples
- Jie Tan: lcp solver, renderer
- Yunfei Bai: build and bug fixes
- Konstantinos Chatzilygeroudis: mimic joint, OSG shadows, shape deep copy, build and bug fixes
- Sehoon Ha: early DART data structure design, pydart
- Donny Ward: build fix
- Andrew Price: build fix
- Eric Tobis: build fix
- Jonathan Martin: build fix

and many others have contributed bug fixes, documentation, and other improvements to DART, which can be found in the DART GitHub repository.

Note: If you have contributed to DART and your name is missing from the list above, or if your contributions are not accurately reflected, or any issue is found, please let us know by opening an issue on GitHub or contacting us directly. We apologize for any oversights and will make every effort to update the list in a timely manner. Thank you for your contributions to DART!

5.8 Code Style Guide

This section describes the code style used in DART project.

5.8.1 C++ Style Guide

Macro Definitions

In DART, we use macros to define compile-time constants and control code flow. All macros in our codebase are prefixed with `DART_` to distinguish them from other identifiers. Macros that control optional dependencies and features follow a consistent naming convention:

- `DART_HAS_<optional_dep>`: A boolean value that is set to true when an optional dependency is detected in the system.
- `DART_ENABLE_<optional_feature>`: A boolean value that is set to true if the optional feature should be enabled when the requirements are met.
- `DART_ENABLED_<optional_feature>`: A boolean value that is set to true if the optional feature is enabled.

We use all-caps for all macro names to ensure consistency and to visually distinguish macros from other types of variables.

5.8.2 Python Style Guide

Naming Conventions

This project uses different naming conventions for the C++ code and the Python bindings. In the C++ code, function names are in camelCase and variables and member variables use snake_case, whereas in the Python bindings, both function names and variables use snake_case.

Here are the naming conventions used in the Python bindings:

- Function names are in snake_case, with words separated by underscores (e.g. *calculate_average*).
- Class names are in CamelCase, with the first letter of each word in uppercase (e.g. *MyClass*).
- Variables and member variables are in snake_case, with words separated by underscores (e.g. *my_variable*).
- Constants are in ALL_CAPS, with words separated by underscores (e.g. *MY_CONSTANT*).
- Namespaces are represented by modules, and are in lowercase, with words separated by underscores (e.g. *my_module.my_namespace*).

For example, the identity member function is called isIdentity in C++:

```cpp
auto so3 = SO3();
bool is_identity = so3.isIdentity();
```

while it is called is_identity in Python:

```python
so3 = SO3()
is_identity = so3.is_identity()
```

Motivations for Different Naming Conventions

The reason for using different naming conventions in the C++ code and the Python bindings is to follow the conventions that are most commonly used in each language. The camelCase convention is more common for function names in the C++ community, while the snake_case convention is more common for function names in the Python community.

By using the standard naming conventions in each language, we can make the code more readable and easier to understand for developers who are familiar with each language. Consistency within each language is important, but it’ s also crucial to document the conventions clearly so that other developers can understand how to use the code and what the naming conventions mean in each context. Additionally, following the naming conventions of each language can help with integration with other Python modules or projects.

5.8.3 CMake Style Guide

TODO

5.9 Migration Guide

5.9.1 From DART 6 to DART 7

TODO

5.10 License

DART is licensed under the 2-Clause BSD License. See the LICENSE file for details.

5.11 Who Uses DART?

5.11.1 Software

DART serves as the backend physics engine for several software projects, including:

- Gazebo: Gazebo simulates multiple robots in a 3D environment, with extensive dynamic integration between objects. Gazebo supports multiple physics engines: ODE, Bullet, DART, and Simbody. website, source on bitbucket
- Aikido: a C++ library, complete with Python bindings, for solving robotic motion planning and decision making problems. This library is tightly integrated with DART for kinematic/dynamics calculations and OMPL for motion planning.
- robot_dart: A generic and lightweight wrapper over DART simulator for fast and flexible robot simulations.

5.11.2 Research

DART has been used in various research domains, including robotics, biomechanics, computer graphics, animation, and physics-based simulation. Notable institutions, universities, or companies that have used DART in their research include Georgia Tech, Oxford University, MIT, Disney Research, and Toyota Research Institute.

DART has been utilized in research areas such as:

- Development of black-box priors for model-based policy search for robotics
- Bayesian optimization with automatic prior selection
- Alternating optimization and quadrature for robust control
- Reset-free trial-and-error learning for robot damage recovery
- Data-driven approach to simulating realistic human joint constraints
- Multi-task learning with gradient-guided policy specialization
- Learning human behaviors for robot-assisted dressing
- Expanding motor skills through relay neural networks
- Learning to navigate cloth using haptics
- Simulation-based design of dynamic controllers for humanoid balancing
- Humanoid manipulation planning using backward-forward search
- Evolutionary optimization for parameterized whole-body dynamic motor skills

- Dexterous manipulation of cloth
- Multiple contact planning for minimizing damage of humanoid falls
- Animating human dressing
- Coupling cloth and rigid bodies for dexterous manipulation
- Orienting in mid-air through configuration changes to achieve a rolling landing for reducing impact after a fall
- Dexterous manipulation using both palm and fingers
- Several conferences and journals where DART has been prominently featured include the International Conference on Robotics and Automation (ICRA), the AAAI Conference on Artificial Intelligence, IEEE Transactions on Evolutionary Computation, Computer Graphics Forum (Eurographics), and ACM Transactions on Graphics (presented at SIGGRAPH Asia).

As of February 2023, DART had over 777 stars on GitHub, and DART has been cited over 236 times, indicating its widespread adoption and use in the research community.

More research papers cited DART can be found at Google Scholar.

Note: If you are using DART in your project and would like to be listed here, please send a pull request to the GitHub repository.