DART: Dynamic Animation and Robotics Toolkits

C. Karen Liu

School of Interactive Computing
Georgia Institute of Technology

Contents

1 Introduction

DART is an open source library for developing kinematic and dynamic applications in robotics and computer animation. DART features two main components: a multibody dynamic simulator developed by Georgia Tech Graphics Lab and a variety of motion planning algorithms developed by Georgia Tech Humanoid Robotics Lab. This document focuses only on the dynamic simulator.

The multibody dynamics simulator in DART is designed to aid development of motion control algorithms. DART uses generalized coordinates to represent articulated rigid body systems and computes Lagranges equations derived from DAlemberts principle to describe the dynamics of motion. In contrast to many popular physics engines which view the simulator as a black box, DART provides full access to internal kinematic and dynamic quantities, such as mass matrix, Coriolis and centrifugal force, transformation matrices and their derivatives, etc. DART also provides efficient computation of Jacobian matrices for arbitrary body points and coordinate frames.

The contact and collision are handled using an implicit time-stepping, velocity-based LCP (linear-complementarity problem) to guarantee non-penetration, directional friction, and approximated Coulombs friction cone conditions. The LCP problem is solved efficiently by Lemke's algorithm. For the collision detection, DART directly uses FCL package developed by UNC Gamma Lab.

In addition, DART supports various joint types (ball-and-socket, universal, hinge, and prismatic joints) and arbitrary meshes. DART also provides two explicit integration methods: first-order Runge-Kutta and fourth-order Runge Kutta.

The rest of tutorial will go over the examples included in the source code. These examples demonstrate how to use DART for developing forward simulation, inverse kinematics, inverse dynamics, feedback control for a manipulator, under-actuated balance control for a biped system, and hybrid control for a mobile manipulator.

2 Forward Simulation

In this section, we will build and simulate a simple chain of rigid bodies connected by ball joints. This example demonstrates how an articulated rigid body system is represented using DART data structures and how a simulation step is formulated using different numerical integration methods. Because this is the first example, we will examine the source code in more details.

2.1 Skeleton data structures

Let us begin with the main function, which only does two things: loading a skeleton file and creating a window for UI and rendering.

Listing 1: Main.cpp

```
int main(int argc, char* argv[])
{
  FileInfoSkel<SkeletonDynamics> model;
  model.loadFile(DART_DATA_PATH"/skel/Chain.skel", SKEL);

  MyWindow window((SkeletonDynamics*)model.getSkel());

  glutInit(&argc, argv);
  window.initWindow(640, 480, "Forward Simulation");
  glutMainLoop();

  return 0;
}
```

We first focus on the data structure of skeleton. A skeleton comprises a set of body nodes connected by joints. We can directly load a skeleton from a file or manually build a new skeleton in the code. DART currently supports two formats of skeleton file: .vsk and .skel. In this example, we load in a skeleton (Chain.skel) with 10 rigid links connected by ball joints.

In general, an app developer does not need to know the details of DART library. However, a basic understanding of the following classes can help one quickly grasp the data structures DART uses for building a simulation app.

- Skeleton. A basic class that contains a list of body nodes, a list of joints, and a list of degrees of freedom.
- SkeletonDynamics. A derived class from **Skeleton Class**. It provides member functions to compute dynamic equations of the skeleton.
- BodyNode. A basic class that provides member functions to evaluate transformation and first derivatives of transformation, to access physical and geometric parameters of the node, and to compute the linear and angular Jacobian matrices for the node.
- BodyNodeDynamics. A derived class from **BodyNode Class**. It provides member functions to compute dynamic variables required in the equations of motion.

2.2 A simple app with graphics and UI

To create and visualize a basic simulation app, we integrate DART into Glut framework. All the examples in this tutorial use Glut and OpenGL libraries to handle user interface and rendering. However, DART does not assume a particular UI or rendering package.

In the main function, we create a graphics application based on **MyWindow Class** derived from **GlutWindow Class**, which is a simple API to interface with Glut functions. Based on the features of the app, the app developer has to implement appropriate virtual functions of **MyWindow Class**. In most graphics apps with keyboard user interaction, three basic virtual functions must be implemented:

Listing 2: MyWindow.h

```
  virtual void draw();
  virtual void keyboard(unsigned char key, int x, int y);
  virtual void displayTimer(int _val);
```

After instantiating and initializing **MyWindow Class** in main function, **glutMainLoop** is invoked and Glut callback functions will start handling display updates, keyboard inputs, etc.

2.3 Simulation

2.3.1 Initialization

Besides graphics and UI capabilities, **MyWindows Class** is also able to forward simulate a passive skeleton. Before the simulation begins, we need to initialize simulation state and dynamic variables by calling **initDyn()**:

Listing 3: MyWindow.cpp

```
void MyWindow::initDyn()
{
  mDofs.resize(mModel->getNumDofs());
  mDofVels.resize(mModel->getNumDofs());
  for(unsigned int i = 0; i < mModel->getNumDofs(); i++){
    mDofs[i] = random(-0.5, 0.5);
    mDofVels[i] = random(-0.1, 0.1);
  }
  mModel->initDynamics();
  mModel->setPose(mDofs, false, false);
}
```

mDofs and **mDofVels** together store the current value of the simulation state. In this example, the simulation state includes the joint position and velocity of the skeleton. In **initDyn()**, we initialize **mDofs** and **mDofVels** randomly to give the skeleton a random initial pose and joint velocity. Because this app will access dynamic variables for simulation, we also need to call the member function **initDynamics()** of the **SkeletonDynamics Class**. Finally, we set the current pose of the skeleton to **mDofs**. The two flags in **setPose()** will be explained in details later.

2.3.2 Simulation step

The simulation code is called from the callback function **displayTimer()** in MyWindow.cpp.

Listing 4: MyWindows.cpp

```
void MyWindow::displayTimer(int _val)
{
  int numIter = mDisplayTimeout / (1000 * mTimeStep);
  for (int i = 0; i < numIter; i++) {
    mIntegrator.integrate(this, mTimeStep);
    mFrame++;
  }
  glutPostRedisplay();
  if (mRunning)
    glutTimerFunc(mDisplayTimeout, refreshTimer, _val);
}
```

Each time **integrate()** is called, DART simulates one time step forward. The simulation time step is defined by **mTimeStep**. **displayTimer()** is called every **mDsiplayTimeout** millisecond. Because **mDsiplayTimeout** is typically larger than **mTimeStep**, we need to run a few iterations of simulation steps each time **displayTimer()** is called. The number of iterations is computed as **numIter**. Typically, a simulation step requires a few procedures in the following order:

- Compute internal forces
- Compute external forces
- Integrate one step forward

Since we do not have internal or external forces applied to the skeleton in this example, invoking **integrate()** is the only procedure required to simulate. We will see examples in later sections that apply internal or external forces to the simulation.

2.3.3 Integration

mIntegrator is a data member of **MyWindows** and can be declared as an explicit Euler integrator or a RK4 integrator in MyWindows.h. Both integrators are derived from **Integrator Class**. Although each integration method has its own formula to integrate the differential equation, the all explicit integration methods can be constructed using three basic operations: **Get State**, **Set State**, and **Evaluate Derivatives**. These three operations needed to be implemented by the app developer as three virtual functions of **Integrator Class**:

Listing 5: MyWindow.h

```
  virtual Eigen::VectorXd getState();
  virtual void setState(Eigen::VectorXd state);
  virtual Eigen::VectorXd evalDeriv();
```

getState() and **setState()** retrieve and assign the value of current simulation state. The most involved function to implement is **evalDeriv**, which computes the derivative of the current state via equations of motion.

Listing 6: MyWindows.cpp

```
VectorXd MyWindow::evalDeriv() {
  mModel->setPose(mDofs, false, false);
  mModel->computeDynamics(mGravity, mDofVels, true);
  VectorXd deriv(mDofs.size() + mDofVels.size());
  VectorXd qddot = mModel->getInvMassMatrix()
        * (-mModel->getCombinedVector());
  mModel->clampRotation(mDofs, mDofVels);
  deriv.head(mDofs.size()) = mDofVels + mTimeStep * qddot;
  deriv.tail(mDofVels.size()) = qddot;
  return deriv;
}
```

In **evalDeriv**, we first call **setPose()** to update the DOF values in the skeleton. **setPose()** takes in two boolean flags. The first flag indicates whether to update all the transformation matrices and the second flag indicates whether to update first derivatives of the transformation matrices. Here we set both of them to be false because the next function, **computeDynamics()**, will take care of the update of transformation matrices and we do not need the information about first derivatives in this app. As a rule of thumb, setting both flags to true is the safest option, as it will provide the most complete information at the cost of computation time.

computeDynamics() updates the mass matrix (and its inversion), Coriolis and centrifugal force, generalized gravitational force, and generalized external force based on the current DOFs and DOF velocity. This is the most important function in DART, and obviously, the most computational costly one. To speed up the computation time, DART provides an alternative way to compute Coriolis, centrifugal, and gravitational force using recursive inverse dynamics (See [**?**] for details). If the flag in **computeDynamics()** is true, DART will utilize recursive inverse dynamics formula to speed up the computation.

Once the terms in equations of motion are updated, the acceleration of DOF (**qddot**) can be readily computed via Lagrange's equations:

$$M\ddot{\mathbf{q}} + C(\mathbf{q}, \dot{\mathbf{q}}) + G(\mathbf{q}) = \mathbf{0} \quad (1)$$

Note that **getCombinedVector()** returns the sum of Coriolis, centrifugal, and gravitational force, instead of each individual force. Please refer [**?**] for derivation of Lagrange's equations.

clampRotation() is particularly important when a joint is represented as an exponential map rotation. The system will become unstable if the exponential map rotation is outside the range of 2PI. Once the rotation magnitude is changed, the velocity needs to change accordingly to represent the same angular velocity.

2.4 Damping force

We encourage the reader to modify the example code with different time steps and integration methods. RK1 is clearly faster than RK4, but the simulation is unstable and bound to blow up after a few second. RK4 is more stable, but its stability is still conditional on the time step of the simulation. One way to create a more stable passive simulation is to introduce damping force at joints. Damping forces can be added as internal forces via member function **setInternalForces()** in **Skeleton** class.

Listing 7: MyWindow.cpp

```
void MyWindow::displayTimer(int _val)
{
  .....
  for (int i = 0; i < numIter; i++) {
    VectorXd damping = computeDamping();
    mModel->setInternalForces(damping);
    mIntegrator.integrate(this, mTimeStep);
    mFrame++;
  }
  .....
}
```

The internal forces also need to be included when evaluating the derivatives for integration:

Listing 8: MyWindows.cpp

```
VectorXd MyWindow::evalDeriv() {
  .....
  VectorXd qddot = mModel->getInvMassMatrix()
        * (-mModel->getCombinedVector()
        + mModel->getInternalForces());
  .....
}
```

Computing appropriate damping force requires some tweaking because the explicit integration methods are only conditionally stable. We set the damping coefficient to 0.001 for all the DOFs controlling twisting and 0.01 for the rest of DOFs.

Listing 9: MyWindow.cpp

```
VectorXd MyWindow::computeDamping()
{
  VectorXd damping = VectorXd::Zero(mDofVels.size());
  damping = -0.01 * mDofVels;
  for (int i = 0; i < damping.size(); i++)
  if (i % 3 == 1)
```

```
    damping[i] *= 0.1;
  return damping;
}
```

3 Contacts

This section introduces the collision detection and handling in DART. We will simulate three boxes and the ground and visualize their motion due to collisions. We follow the same Glut framework described in the previous section, but in this example we allow the user to apply forces to interact one of the boxes. We also add playback functionality so the user can replay the simulation result.

3.1 Contact dynamics

Collision detection and handling are implemented in **ContactDynamics Class**. To include collision functionality in the application, the user needs to create an instance of **ContactDynamics**. The constructor of **ContactDynamics Class** requires the user to pass in a list of skeletons potentially participating collision, the simulation time step, friction coefficient (_mu), and the number of friction cone basis vectors (_d). The default values for _mu and _d are 1.0 and 4. In general, a smaller _mu indicates a more slippery the colliding surface, and a larger _d results in more accurate the sliding contact. Please see [?] for detailed implementation of LCP contact modeling.

Applying collision detection and handling is very straightforward in DART once **ContactDynamics** is instantiated. The user can call the member function, **applyContactForces()**, every time collision and contact forces need to be updated. The resulting contact forces are represented in two different ways. **getConstraintForce()** returns the contact forces in generalized coordinates of a desired skeleton. Alternatively, the user can retrieve the contact forces in the Cartesian space by accessing the data structure of each contact point via **getCollisionChecker()**. We will see examples of both cases in the next subsection.

3.2 Simulation

3.2.1 Initialization

initDyn() for this example is very similar to the one described in Section **??**: we initialize the simulation state variables, **mDofs** and **mDofVels**, and then initialize the dynamic variables and the pose of the skeleton. The differences in this example are the following.

Listing 10: MyWindow.cpp

```
void MyWindow::initDyn()
```

```
{
  .....
  mSkels[0]->setImmobileState(true);
  mCollisionHandle = new dynamics::ContactDynamics(mSkels,
      mTimeStep);
}
```

First, we set the "ground" skeleton to be immobile. If a skeleton is immobile, it is not included in the dynamic simulation, namely, it cannot be moved by forces. However, it still participates in collision detection and will induce contact forces to other skeletons. Second, we create an instance of **ContactDynamics Class**, **mCollisionHandle**, to handle collisions between all the skeletons in the scene, including a ground and three boxes.

3.2.2 Simulation step

Again, simulation code is invoked from the callback function **displayTimer()**. In this example, we add a new feature of playback, which bakes and replays the simulation results.

Listing 11: MyWindow.cpp

```
void MyWindow::displayTimer(int _val)
{
  int numIter = mDisplayTimeout / (mTimeStep * 1000);
  if (mPlay) {
    mPlayFrame += 16;
    if (mPlayFrame >= mBakedStates.size())
      mPlayFrame = 0;
 }else if (mSim) {
    for (int i = 0; i < numIter; i++) {
      static_cast<BodyNodeDynamics*>(mSkels[1]->getNode(0))
          ->addExtForce(Vector3d(0.0, 0.0, 0.0), mForce);
      mIntegrator.integrate(this, mTimeStep);
      bake();
      mSimFrame++;
    }
    mForce.setZero();
  }
  glutPostRedisplay();
  glutTimerFunc(mDisplayTimeout, refreshTimer, _val);
}
```

When the flag **mPlay** is true, playback mode is active and the frame index **mPlayFrame** is incrementing based on the desired playback speed (16 is arbitrarily chosen). If **mSim** is true, DART enters the simulation mode and runs **numIter** steps of simulation as described in Section **??**. The difference in this example is that the user can push the largest box on the ground and DART will take into account this external force in the simulation. **addExtForce()** is a member function of **BodyNodeDynamics Class**. It takes input arguments as the coordinates of the point of application in the body frame, as well as the force vector in the world frame. **addExtForce()** also provides the options to express the point of application in the world frame or the force vector in the body frame.

Listing 12: BodyNodeDynamics.h

```
void addExtForce( const Eigen::Vector3d& _offset,
                             const Eigen::Vector3d& _force,
                             bool _isOffsetLocal=true,
                             bool _isForceLocal=false );
```

After each integration step, we call **bake()** to store the new states of the skeletons and the collision results. The information of collision is stored in the data structure **ContactPoint**.

Listing 13: CollisionSkeleton.h

```
struct ContactPoint {
  Eigen::Vector3d point;
  Eigen::Vector3d normal;
  Eigen::Vector3d force;
  .....
}
```

The developer can access the position, normal, and Cartesian force of any contact point via the member data, **mCollisionChecker** in **ContactDynamics Class**.

Listing 14: MyWindow.cpp

```
void MyWindow::bake()
{
  .....
  for (int i = 0; i < nContact; i++) {
    int begin = mIndices.back() + i * 6;
    state.segment(begin, 3) = mCollisionHandle
      ->getCollisionChecker()->getContact(i).point;
    state.segment(begin + 3, 3) = mCollisionHandle
      ->getCollisionChecker()->getContact(i).force;
  }
  .....
}
```

3.2.3 Integration

The integration process is very similar to the previous example where **evalDeriv()** updates the degrees of freedom in skeletons according to the current simulation state, evaluates the dynamic variables of equations of motion, and finally computes the derivatives for integration.

Listing 15: MyWindow.cpp

```
VectorXd MyWindow::evalDeriv() {
  for (unsigned int i = 0; i < mSkels.size(); i++) {
    if (mSkels[i]->getImmobileState()) {
      mSkels[i]->setPose(mDofs[i], true, false);
    } else {
      mSkels[i]->setPose(mDofs[i], false, true);
      mSkels[i]->computeDynamics(mGravity, mDofVels[i], true);
    }
  }
  mCollisionHandle->applyContactForces();

  VectorXd deriv = VectorXd::Zero(mIndices.back() * 2);
  for (unsigned int i = 0; i < mSkels.size(); i++) {
    if (mSkels[i]->getImmobileState())
      continue;
    int start = mIndices[i] * 2;
    int size = mDofs[i].size();
    VectorXd qddot = mSkels[i]->getInvMassMatrix()
        * (-mSkels[i]->getCombinedVector()
        + mSkels[i]->getExternalForces()
        + mCollisionHandle->getConstraintForce(i));
    mSkels[i]->clampRotation(mDofs[i], mDofVels[i]);
    deriv.segment(start, size) = mDofVels[i] + (qddot * mTimeStep);
    deriv.segment(start + size, size) = qddot;
  }
  return deriv;
}
```

If a skeleton is immobile, we do not need to evaluate its dynamic equations, but we still need to call **setPose()** to update its degrees of freedom for collision detection. Note that the first flag is set to true because the updated transformation matrices are required in collision detection routine. For a mobile skeleton, we call both **setPose()** and **computeDynamics()**. Similar to the previos example, the first flag in **setPose()** is false. However, the second flag must be true for this example because the updated derivative information is needed in collision handling routine.

Because this app involves collision, we need to invoke **applyContactForces()** in **evalDeriv()**. The collision detection and handling routine compute appropriate contact forces and store them in generalized coordinates. When computing the derivatives **qddot**, we need to include the contact forces by calling **getConstraintForce()**. In this example, we also allow for external push forces as user input. Therefore, **getExternalForces()** is called when computing **qddot**.

4 Proportional-Derivative Control

So far we have only applied DART to simulate passive skeletons. In this section, we will develop a simple proportional-derivative (PD) controller for a hand skeleton. The goal of this controller is to compute the torques required to track an input trajectory of the hand against gravity and external push forces. Later in this section, we will modify the standard PD controller to a different formulation, called stable proportional-derivative (SPD) control.

4.1 Controller

We create a **Controller Class** to compute joint torques at each simulation time step. The constructor takes as input a reference motion, a skeleton, and the simulation time step.

Listing 16: Controller.cpp

```
Controller::Controller(kinematics::FileInfoDof *_motion,
  dynamics::SkeletonDynamics *_skel, double _t) {
  mMotion = _motion;
  mSkel = _skel;
  mTimestep = _t;
  int nDof = mSkel->getNumDofs();
  mKp = MatrixXd::Identity(nDof, nDof);
  mKd = MatrixXd::Identity(nDof, nDof);;

  mTorques.resize(nDof);
  mDesiredDofs.resize(nDof);
  for (int i = 0; i < nDof; i++){
    mTorques[i] = 0.0;
    mDesiredDofs[i] = mSkel->getDof(i)->getValue();
  }

  for (int i = 0; i < nDof; i++) {
    mKp(i, i) = 800.0;
    mKd(i, i) = 15;
  }

  mSimFrame = 0;
  mInterval = (1.0 / mMotion->getFPS()) / mTimestep;
}
```

For each DOF, we need to determine the stiffness **mKp** and the damping **mKd** parameters. The values of these parameters directly affect the stability of the simulation, but tweaking them by hand is not a trivial task. Many factors, such as mass distribution, skeleton configuration, and joint types, need to be considered in the parameter tuning process. In this example, we are able to select one single value for the entire set of DOFs because we can take advantage of the mass matrix information computed by DART (detailed later). Finally, we need to account for the difference in a simulation time step and input motion frame rate. **mInterval** stores the number of simulation time steps for each frame of the input motion.

Given the current DOF and DOF velocity, **computeTorques()** updates the desired DOFs (**mDesiredDofs** and computes the generalized torques (**mTorques**). We scale **mTorques** by the current mass matrix so that the final torques accounts for the accumulated mass and inertial controlled by each DOF.

Listing 17: Controller.cpp

```
void Controller::computeTorques(const VectorXd& _dof,
  const VectorXd& _dofVel) {
  int motionFrame = mSimFrame / mInterval;
  if (motionFrame >= mMotion->getNumFrames())
    motionFrame = mMotion->getNumFrames() - 1;

  mDesiredDofs = mMotion->getPoseAtFrame(motionFrame);
  mTorques = -mKp * (_dof - mDesiredDofs) - mKd * _dofVel;
  mTorques = mSkel->getMassMatrix() * mTorques;
  mSimFrame++;
}
```

4.2 Simulation

4.2.1 Initialization

The initialization code is very similar to previous examples. One thing different in this example is that **initDyn()** needs to call **computeDynamics()** from the skeleton. This is because the computation of control force at first simulation step needs to access mass matrix (recall that we scale **mTorques** by the mass matrix). Another difference here is that we need to instantiate **Controller Class** and assign the current pose to **mDesiredDofs** for the controller.

Listing 18: MyWindow.cpp

```
void MyWindow::initDyn()
{
  .....

  for (unsigned int i = 0; i < mSkels.size(); i++) {
    mSkels[i]->initDynamics();
    mSkels[i]->setPose(mDofs[i], false, false);
    mSkels[i]->computeDynamics(mGravity, mDofVels[i], false);
  }
  int nDof = mSkels[0]->getNumDofs();
  mController = new Controller(mMotion, mSkels[0], mTimeStep);
  for (int i = 0; i < nDof; i++)
    mController->setDesiredDof(i, mController->getSkel()
      ->getDof(i)->getValue());
}
```

4.2.2 Simulation Step

At each simulation step, we need to collect the external push force, compute internal control force, and integrate to the next step. In this example, we allow the user to use keyboard to push the distal segment of index finger.

Listing 19: MyWindow.cpp

```
void MyWindow::displayTimer(int _val)
{
  .....
  static_cast<BodyNodeDynamics*>(mSkels[0]
    ->getNode("fixedHand_indexDIP"))
    ->addExtForce(Vector3d(0.02, 0.0, 0), mForce);
  mController->computeTorques(mDofs[0], mDofVels[0]);
  mSkels[0]->setInternalForces(mController->getTorques());
  mIntegrator.integrate(this, mTimeStep);
  .....
}
```

4.3 Stable PD Control

Using the current mass matrix to scale the control forces greatly reduces the need to tuen stiffness and damping parameters for the PD controllers. However, we implement another control method that produces even more robust tracking results. This controller, termed SPD, was introduced by Tan *et al.* []. Please refer to the paper for formulation details.

The range of stiffness and damping parameters that produce stable results is a lot wider than standard PD controllers. Here we arbitrary pick two values and assign them for all DOFs.

Listing 20: Controller.cpp

```
Controller::Controller(kinematics::FileInfoDof *_motion,
  dynamics::SkeletonDynamics *_skel, double _t) {
  .....
#ifdef SPD
```

```
  for (int i = 0; i < nDof; i++) {
    mKp(i, i) = 15.0;
    mKd(i, i) = 2.0;
  }
#else
  for (int i = 0; i < nDof; i++) {
    mKp(i, i) = 800.0;
    mKd(i, i) = 15;
  }
#endif
  .....
}
```

We simply follow the formulation in [] to compute control force at each simulation step.

$$\ddot{\mathbf{q}}^n = (M + K_d\Delta t)^{-1}(-C - K_p(\mathbf{q}^n + \dot{\mathbf{q}}^n\Delta t - \bar{\mathbf{q}}^{n+1}) - K_d\dot{\mathbf{q}}^n + \tau_{ext} + \tau_{int}) \tag{2}$$

where $\bar{\mathbf{q}}^{n+1}$ is the reference pose in the next time step.

Listing 21: Controller.cpp

```
void Controller::computeTorques(const VectorXd& _dof,
  const VectorXd& _dofVel) {
  .....
#ifdef SPD
  MatrixXd invM = (mSkel->getMassMatrix() + mKd * mTimestep)
    .inverse();
  VectorXd p = -mKp * (_dof + _dofVel * mTimestep - mDesiredDofs);
  VectorXd d = -mKd * _dofVel;
  VectorXd qddot = invM * (-mSkel->getCombinedVector() + p + d);
  mTorques = p + d - mKd * qddot * mTimestep;
#else
  mTorques = -mKp * (_dof - mDesiredDofs) - mKd * _dofVel;
  mTorques = mSkel->getMassMatrix() * mTorques;
#endif
  .....
}
```

Note that the computation of SPD is more costly due to the inversion of augmented mass matrix. The developer should weigh in stability, performance, and development effort when choose between using PD or SPD.

5 Motion Analysis

This app provides some basic motion analysis functionality which computes the torques, linear momentum, and angular momentum for a given skeleton and motion sequence. In this app, we use the same hand skeleton and motion from the previous section. The analysis results will be output into a file, “output.txt”.

5.1 Analyzer

The **Analyzer Class** has three private member functions: **computeTorques()**, **evalLinMomentum()**, and **evalAngMomentum()**. With the dynamic computation provided by DART library, computing torque, linear momentum, and angular momentum become very simple and can be done in a few lines of code.

To compute torques, we can simply call the function **computeInverseDynamicsLinear()** to perform inverse dynamics on the current state. We need to provide the current velocity and acceleration of the state, **qdot** and **qddot**, which are computed by finite differencing the input motion. Note that we set both flags to false when calling **computeInverseDynamicsLinear()**. The first flag indicates whether Jacobian matrices need to be recomputed and the second flag indicates whether these is external force involved. They are both negative in our case here.

Listing 22: Analyzer.cpp

```
void Analyzer::computeTorques() {
  int nFrame = mMotion->getNumFrames();
  double timestep = 1.0 / mMotion->getFPS();
  for (int i = 0; i < nFrame - 2; i++) {
    VectorXd qdot = (mMotion->getPoseAtFrame(i + 1)
      - mMotion->getPoseAtFrame(i)) / timestep;

    VectorXd qddot = (mMotion->getPoseAtFrame(i + 2)
      - 2 * mMotion->getPoseAtFrame(i + 1)
      + mMotion->getPoseAtFrame(i)) / (timestep * timestep);
    mSkel->setPose(mMotion->getPoseAtFrame(i), true, false);
    mTorques[i] = mSkel->computeInverseDynamicsLinear(
      mGravity, &qdot, &qddot, false, false);
  }
}
```

evalLinMomentum() provides the functionality to compute linear momentum of the current state from the current state velocity. Because DART allows the developer to access Jacobian matrix for each body node, we can easily project joint velocity to the center of mass velocity of the body node by $\dot{\mathbf{c}} = \frac{\partial \mathbf{c}}{\partial \mathbf{q}}\dot{\mathbf{q}} = \mathbf{J}\dot{\mathbf{q}}$.

Listing 23: Analyzer.cpp

```
Vector3d Analyzer::evalLinMomentum(const VectorXd& _dofVel) {
  MatrixXd J(MatrixXd::Zero(3, mSkel->getNumDofs()));
  for (int i = 0; i < mSkel->getNumNodes(); i++) {
    BodyNodeDynamics *node = (BodyNodeDynamics*)mSkel->getNode(i);
    MatrixXd localJ = node->getJacobianLinear() * node->getMass();
    for (int j = 0; j < node->getNumDependentDofs(); j++) {
      int index = node->getDependentDof(j);
      J.col(index) += localJ.col(j);
    }
  }
  Vector3d cDot = J * _dofVel;
  return cDot / mSkel->getMass();
}
```

Similarly, we can compute the anular momentum from the current state velocity. We use the following formulation. The details can be found in another tutorial on angular momentum [].

$$\mathbf{L} = \sum_{i=1}^{n} (m_i(\mathbf{c}_i - \mathbf{c}) \times (\dot{\mathbf{c}}_i - \dot{\mathbf{c}}) + R_i \bar{I}_i R_i^T \omega_i) \tag{3}$$

where $\bar{I}_i$ is the inertia matrix of i-th rigid body in the local frame, which can be precomputed based on the shape of the rigid body. ω_i is the angular momentum of i-th rigid body in the world frame. R_i is the rotation matrix that transforms a vector from the local frame to the world frame.

Listing 24: Analyzer.cpp

```
Vector3d Analyzer::evalAngMomentum(const VectorXd& _dofVel) {
  Vector3d c = mSkel->getWorldCOM();
  Vector3d sum = Vector3d::Zero();
  Vector3d temp = Vector3d::Zero();
  for (int i = 0; i < mSkel->getNumNodes(); i++) {
    BodyNodeDynamics *node = (BodyNodeDynamics*)mSkel
      ->getNode(i);
    node->evalVelocity(_dofVel);
    node->evalOmega(_dofVel);
    sum += node->getInertia() * node->mOmega;
    sum += node->getMass() * (node->getWorldCOM() - c)
      .cross(node->mVel);
  }
  return sum;
}
```

6 Inverse Kinematics

Inverse kinematics (IK) is one of the most frequently used techniques in computer animation and robotics. With the capability of computing arbitrary Jacobian matrices, DART makes IK computation extremely straightforward. In this section, we will build a simple IK app that enforces the foot positions of a human skeleton (Figure **??**).

6.1 Optimization

DART provides a namespace called "optimizer". Many data structures and functionalities in this namespace are designed for solving constrained optimization problems. In this example, we will use our own steepest gradient descent solver instead of other commercial optimization packages, such as MOSEK, or SNOPT. To formulate an optimization problem in DART, the developer first needs to define three components: constraints, objective function, and variables.

Constraints and objective function are managed using data structure **ConstraintBox Class** and **ObjectiveBox Class**. A constraint box maintains a list of functions, each of which "knows" how to evaluate the current function value and gradients. Similarly, an objective box also maintains a list of functions, each of which represents a term in the objective function. DART defines a unified data type for functions in the constraint box and the objective box, called **Constraint Class**. Different functions can be derived from **Constraint Class** as subclasses. We will show one example of subclass, **Position Constraint Class** in the next subsection. Once a constraint is instantiated, we can choose to put it into constraint box or objective box, based on whether it should be a hard constraint or a soft constraint.

Before solving the optimization problem, the developer also needs to define variables using **Var Class**. The constructor requires the initial value, the upper bound, and the lower bound of the variable. In this example, we instantiate **Var Class** for each degree of freedom of the human skeleton:

Listing 25: MyWindow.cpp

```
void MyWindow::initIK()
{
  for (int i = 0; i < mSkel->getNumDofs(); i++) {
    Var *v = new Var(mSkel->getDof(i)->getValue(),
      mSkel->getDof(i)->getMin(), mSkel->getDof(i)->getMax());
    mVariables.push_back(v);
  }
  .....
}
```

6.2 Position Constraint

The most common constraints used in IK problem is position constraint, which enforces a point on the skeleton to a point in space. **PositionConstraint Class** is a subclass derived from **Constraint Class**. Its main functionality is to evaluate the constraint function and gradients by implementing two virtual functions: **evalCon()** and **fillJac()**.

Evaluating the current function value is very straightforward for a position constraints: computing the difference between the body point in the world frame and its target location. Note that the returned vector can have various sizes depending on the constraint function. In the case of position constraint, it returns a 3 by 1 vector.

Listing 26: PositionConstraint.cpp

```
VectorXd PositionConstraint::evalCon() {
  Vector3d wp = mNode->evalWorldPos(mOffset);
  Vector3d c = wp - mTarget;
  VectorXd ret(c);
  return ret;
}
```

Evaluating the constraint gradient is also simple because the gradients of transformation chains are taken care of by DART. The developer only needs to access the gradients of transformation chain relevant to the DOF of interest and store the results in a global Jacobian matrix.

Listing 27: PositionConstraint.cpp

```
void PositionConstraint::fillJac(VVD jEntry, VVB jMap,
  int index) {
  for(int i = 0; i < mNode->getNumDependentDofs(); i++) {
    int dofindex = mNode->getDependentDof(i);
    const Var* v = mVariables[dofindex];
    VectorXd J = xformHom(mNode->getDerivWorldTransform(i),
      mOffset);
    for (int k = 0; k < 3; k++) {
      (*jEntry)[index + k]->at(dofindex) = J[k];
      (*jMap)[index + k]->at(dofindex) = true;
    }
  }
}
```

The data type **VVD** and **VVB** are defined in **OptimizerArrayTypes.h**. We use them as the data structures to store the content of global Jacobian matrixes (**jEntry**) and its non-zero pattern (**jMap**). The input variable **index** indicates the beginning row of this constraint in the global Jacobian matrix.

Another virtual function we often need to implement is **fillObjGrad()**, which is used when this constraint is put into the objective box. **fillObjGrad()** computes the gradient of the objective function due to the contribution of this term. For example, assuming the objective function has the form of $g(\mathbf{x}) = \sum_i \|f_i(\mathbf{x})\|^2$, where f_i is evaluated by **evalCon** of the i-th constraint, **fillObjGrad()** will then compute $2f_i(\mathbf{x})\frac{\partial f_i}{\partial \mathbf{x}}$

Listing 28: PositionConstraint.cpp

```
void PositionConstraint::fillObjGrad(std::vector<double>& dG) {
  VectorXd dP = evalCon();
  for(int i = 0; i < mNode->getNumDependentDofs(); i++) {
    int dofindex = mNode->getDependentDof(i);
    VectorXd J = xformHom(mNode->getDerivWorldTransform(i),
      mOffset);
    dG[dofindex] += 2 * dP.dot(J);
  }
}
```

7 Other Examples

DART also includes a balance controller for an under-actuated human skeleton and a hybrid controller for a mobile manipulator. These two controllers follow exactly the same framework described in the previous sections and only vary in their control force computation. Please refer to the source code for implementation details.