DARPA Robotics Challenge
OSRF: Gazebo / Simbody Simulator Integration
Subcontractor Stanford University
Milestone Report #21

Open Source Robotics Foundation
170 South Whisman Road
Building D, Suite A
Mountain View, CA 94041
650.450.9681
info@osrfoundation.org

December 17, 2014

Contents

1 Overview

This document details the completion of the fourth and final milestone for the DRC Simulator Stanford subcontractor project, which is Milestone #21 in the overall project. The defining goal for this milestone was:

> Demonstrate operational space control of the simulated DRC robot doing postural tasks.

"Operational space" or "task space" control requires a controller that contains an embedded model of the controlled plant (in this case the Atlas robot), and can accept control commands in Cartesian (task) space rather than joint space. The modeled plant dynamics are used to determine a set of control torques that can implement the task. The control torques are then applied to the plant, which may be a real robot or a simulated one but in any case is *not* necessarily identical to the model used in the controller. The task space controller is necessarily built using a Simbody model,

because currently only Simbody has support for the necessary task space computations. However, the simulated robot can be simulated with any of Gazebo's physics engines.

A feature of the task space control approach presented here is the ability to specify hierarchical control objectives. When the primary task does not require all of the robot's degrees of freedom (dofs), then it is possible to achieve secondary goals while performing the task. In this report, the primary task is 3-dof spatial positioning of an end-effector to follow a specified task point in Cartesian space. The task point may be moved by the user in the Gazebo GUI and the robot's hand will follow it interactively. The remaining degrees of freedom control the robot's upper-body posture to a commanded position, but only to the extent that this secondary goal does not conflict with the end-effector task. All actuated joints may be involved in both the end-effector and posture tasks, but the posture control occurs in a subspace of the full joint space in which control torques leave the end-effector task unaffected.

As for previous milestones, there were many subtasks required to achieve Milestone #21. Here is a list of the major tasks completed along the way:

1. **Researched task space theory:** Reviewed literature and discussed with experts at Stanford to determine which aspects of the theory were applicable and how to decompose those into Simbody operators for efficient computation.
2. **Created TaskSpace helper class:** Graduate student Chris Dembia developed a TaskSpace class for simplifying the use of Simbody operators to perform task space computations.
3. **Determined the best architecture for task space control in Gazebo:** We considered several options and decided that a separate process with an embedded Simbody model made the most sense.
4. **Developed UR10 robot arm example in Simbody:** We chose the UR10 robot arm as simple prototype. We developed a Simbody standalone example containing a simulated UR10 and a controller-embedded UR10 and demonstrated end-effector task control with gravity compensation in the null space.
5. **Ported the UR10 controller to Gazebo:** OSRF created an empty controller process with communication between it and Gazebo. Working together, we incorporated the Simbody example controller into the new controller process, yielding a task space controller for the UR10.
6. **Developed Atlas example in Simbody:** Extending the methodology developed for the UR10 example, we created a new Simbody example that contained a simulated Atlas humanoid with lower-body control, and an upper-body task space and posture controller.
7. **Ported the Atlas controller to Gazebo:** Extending the earlier UR10 work, we incorporated the Simbody Atlas controller into the previously-built controller process, and applied it to a full Gazebo Atlas simulation in which the pelvis was positioned explicitly while the upper body was under task space control.
8. **Demonstrated task-space and postural control of Atlas:** We were able to execute a Gazebo simulation of Atlas with an internally controlled lower body while the upper body performed an interactive reaching task and maintained a neutral posture, with optional gravity compensation.

2 Tasks performed

The following sections contain descriptions of the work completed to accomplish each of the above tasks.

2.1 Researched task space theory

Task space control is a well-developed field although its extension to whole body (posture) control is relatively recent. The primary research tasks for us here were:

- Study the literature and discuss with Stanford experts to understand whole body task space control.
- Select a subset of task space control methodology that could be applied to Atlas in Gazebo and would be feasible to implement in the available time.
- Determine how to take advantage of Simbody's unique computational capabilities to implement task space control efficiently.
- Develop an API that would permit Simbody users to take advantage of that efficiency.

Our primary references for understanding whole-body task space control were [1, 2, 3] and Luis Sentis' Ph.D. thesis [4]. Prof. Oussama Khatib at Stanford was a major contributor to operational space theory [5]; we benefited from his courses and discussions with experts from his lab, including graduate student Gerald Brantner and Khatib lab alumnus Luis Sentis, now faculty at U.T. Austin.

A "task" defines a "task space" which is a convenient coordinate system directly related to some desired behavior, such as the Cartesian coordinates of an end-effector point, a tool orientation, system center of mass location, or a task specified in joint space such as a desired neutral pose. A "motion task" specifies a desired acceleration in task space. A "force task" specifies desired reaction force values for system constraint forces. For both kinds of control tasks, task space control provides a means for calculating joint actuation forces (meaning forces or torques as appropriate to the joint type) that will best produce the desired task motions or induce the desired constraint forces. For this milestone, we are interested only in motion tasks; see [4, 6] for more information about force tasks.

Tasks may be prioritized, with higher-priority tasks being satisfied first and lower-priority tasks being satisfied only to the extent that can be done without interfering with higher-priority tasks. A group of tasks at the same priority is called a "task set", and all the tasks in a set are solved simultaneously. Tasks in a set may conflict, in which case they are resolved using a weighted least squares computation. The system constraints are a set of motion tasks at the highest priority and must be satisfied. All other tasks may be achieved imperfectly.

Here we will give a concise technical summary of task space control, expressed in notation that is compatible with Simbody's formulation [7]. Each task t has dimensionality d_t (e.g., $d_t = 3$ for the Atlas positioning task) and is associated with a *task Jacobian matrix* $J_t(q) \in \mathbb{R}^{d_t \times n}$, where n is the number of joint-space velocities (generalized speeds) u. For a fixed joint-space configuration q, J_t is a linear mapping from joint velocities u to task velocity $\dot{x}_t$, with $\dot{x}_t = J_t u$. In turn, the transpose of the task Jacobian maps from a force $F_t \in \mathbb{R}^{d_t}$ applied in task space to the joint space generalized forces $f \in \mathbb{R}^n$ it would produce, with $f = J_t^{\mathrm{T}} F_t$. Note that if a task is given in joint space, the corresponding Jacobian is just the $n \times n$ identity matrix.

We assign each task t a priority $p_t \in \{1, ..., p_{\max}\}$, with lower values indicating higher priority. Let the number of tasks sharing a given priority level p be n_p. When $n_p > 1$, multiple tasks are to be performed at the same priority p so they may potentially conflict and must be solved simultaneously for joint space forces f_p. All tasks at priority p are combined into a single *priority Jacobian* $G_p \in \mathbb{R}^{m_p \times n}$, where

$$G_p = \begin{bmatrix} J_{t_1} \\ J_{t_2} \\ \vdots \\ J_{t_{n_p}} \end{bmatrix} \tag{1}$$

and $m_p = \sum_{t:p_t=p} d_t$ is the total dimensionality of the n_p tasks at priority p. The highest priority $p = 1$ is reserved for system constraints, so $G_1 \equiv G \in \mathbb{R}^{m \times n}$ is the conventional Simbody constraint Jacobian [7], and task priorities begin at $p = 2$.

In these terms, Simbody's equations of motion can be written:

$$M\,\dot{u} + G^{\mathrm{T}}\lambda = f_0 + f_{\mathrm{task}} \tag{2a}$$
$$G\,\dot{u} = b \tag{2b}$$
$$\text{where} \qquad f_0 = J^{\mathrm{T}}(F_{\mathrm{applied}} + F_{\mathrm{gravity}} + F_{\mathrm{inertial}}) + f_{\mathrm{applied}} \tag{2c}$$
$$f_{\mathrm{task}} = \sum_{p=2}^{p_{\max}} f_p \tag{2d}$$
$$(f_1 \equiv -G^{\mathrm{T}}\lambda) \tag{2e}$$

where $M \in \mathbb{R}^{n \times n}$ is the symmetric, positive definite joint space mass matrix; $u \in \mathbb{R}^n$ is the joint space velocity vector; $G \in \mathbb{R}^{m \times n}$ is the constraint Jacobian; $\lambda \in \mathbb{R}^m$ is the vector of constraint-space forces; $b \in \mathbb{R}^m$ is a bias term; and $J \in \mathbb{R}^{6n_b \times n}$ is the system Jacobian mapping joint space velocities to body (link) frame velocities, for each of the n_b bodies in the system. $f_{\mathrm{task}} \in \mathbb{R}^n$ is the sum of all the joint forces produced by the task space controller over the priority levels. (f_1 is the constraint forces mapped to joint space.) $F_{\mathrm{applied}}, F_{\mathrm{gravity}} \in \mathbb{R}^{6n_b}$ are spatial forces acting on bodies from gravity and other sources, and $f_{\mathrm{applied}} \in \mathbb{R}^n$ is the non-task forces in joint space, such as damping. $F_{\mathrm{inertial}} \in \mathbb{R}^{6n_b}$ is the velocity-dependent Coriolis and gyroscopic forces due to rotation, as felt by each body at its body frame.

Although Eqns. 2 are expressed in matrix terms, Simbody actually processes them with a set of O(n) operators. Available operators are $M\odot, M^{-1}\odot, J\odot, J^{\mathrm{T}}\odot, G\odot, G^{\mathrm{T}}\odot$. Each of these may be applied to a vector of the appropriate length to yield a vector result in O(n) time or O($n + n_p$) time for operations involving a priority Jacobian containing n_p tasks. In addition, Simbody can calculate Coriolis terms $\dot{J}u$ for any Jacobian J in O($n + n_p$) time.

Important quantities in task space control are presented next. Let $N_0* = \mathbf{1}_{n \times n}$. Then for each priority $p > 0$ we define these quantities so that they operate only in the composite null space $N_{(p-1)*}$ of all higher-priority tasks, and thus cannot interfere with those tasks. We indicate such quantities with subscript $p*$, where the $*$ indicates that the quantities are cumulative over higher priority levels.

$$G_{p*} = G_p N_{(p-1)*} \qquad \textit{prioritized task set Jacobian} \tag{3a}$$
$$A_{p*} = G_{p*} M^{-1} G_{p*}^{\mathrm{T}} \qquad \textit{task set p compliance matrix} \tag{3b}$$
$$= G_p M^{-1} G_{p*}^{\mathrm{T}} \qquad \textit{see [4]} \tag{3c}$$
$$L_{p*} = A_{p*}^{+} \qquad \textit{task set p inertia matrix} \tag{3d}$$
$$\overline{G}_{p*} = M^{-1} G_{p*}^{\mathrm{T}} L_{p*} \qquad \textit{generalized inverse of } G_{p*} \tag{3e}$$
$$N_{p*} = N_{(p-1)*}(\mathbf{1} - \overline{G}_{p*} G_{p*}) \qquad \textit{null space for task set p} \tag{3f}$$

In case there are no tasks at priority p, we will have have $N_{p*} = N_{(p-1)*}$. The rank-adjusted pseudoinverse in Eqn. 3d as used in Khatib et al. [1] is one way of dealing with the potential singularity of A_{p*}; see Dariush et al. [2] for others.

We also define the following quantities to reflect the cumulative joint accelerations produced by all forces including gravity, inertial, applied, constraint, and higher priority task forces:

$$f_{p*} = \sum_{i=0}^{p} f_i \tag{4a}$$

$$\dot{u}_{p*} = M^{-1} f_{p*} \tag{4b}$$

Now for each task set p we have a desired acceleration $\ddot{x}_p^{\text{desired}}$ that we would like to achieve, as a result of some control law. Forces accumulated up to priority $p-1$ will already have produced a task acceleration

$$\ddot{x}_p^- = G_p \dot{u}_{(p-1)*} + \dot{G}_p u \tag{5}$$

We would like to generate joint actuation forces f_p that will cause a *change* $\Delta\ddot{x}_p = \ddot{x}_p^{\text{desired}} - \ddot{x}_p^-$, while not disturbing any of the higher-priority tasks. That is, we are looking for forces f_p such that

$$G_p M^{-1} f_p = \Delta\ddot{x}_p \tag{6}$$

$$\text{and} \quad N_{(p-1)*}^{\mathrm{T}} f_p = f_p \tag{7}$$

where Eqn. 7 indicates that f_p is in the null space of the higher-priority tasks because the null space matrix $N_{(p-1)*}$ is a projection matrix (thus so is its transpose).

There may be no solution or many solutions for f_p. When there are many solutions (common for humanoid robots), the particular solution can be chosen to optimize a variety of metrics. For example, one may efficiently solve the following system to obtain a least squares solution for dummy force variable f:

$$G_p M^{-1} N_{(p-1)*}^{\mathrm{T}} f = \Delta\ddot{x}_p \tag{8a}$$

$$\text{and then} \quad f_p = N_{(p-1)*}^{\mathrm{T}} f \tag{8b}$$

However, this may not guarantee that the actual applied joint force f_p is also minimal.

Khatib et al. [1] solve this as follows:

$$f_p = G_{p*}^{\mathrm{T}} F_p \tag{9a}$$

$$\text{where} \quad F_p = L_{p*} \Delta\ddot{x}_p \tag{9b}$$

which minimizes $\dot{u}_p^{\mathrm{T}} M \dot{u}_p$ which they call the "acceleration energy", where $\dot{u}_p = M^{-1} f_p$. (Khatib uses Λ for the operational space inertia matrix we are calling L.)

Neither of the above accounts for joint torque limits; those are typically dealt with by truncation, which is suboptimal. A better general approach would be to set this up as a constrained convex optimization problem such as this [8]:

$$\text{minimize} \quad \textit{objective}(f_p) \tag{10a}$$

$$\text{subject to} \quad G_p M^{-1} f_p = \Delta\ddot{x}_p \tag{10b}$$

$$\text{and} \quad f_p = N_{(p-1)*}^{\mathrm{T}} f_p \tag{10c}$$

$$\text{and} \quad \textit{lower} \preceq f_p + f_{(p-1)*} \preceq \textit{upper} \tag{10d}$$

This should be solvable in real time using an embedded convex solver like ECOS [9] or CVXGEN [10].

We have not discussed underactuated systems here; see [4] for an extensive treatment. Underactuation is awkward to incorporate efficiently into the conventional operational space framework; a convex optimization approach would make that easier. For this milestone, we followed Khatib's approach from [1], using truncation where necessary to satisfy joint torque limits and ensuring that the modeled system is fully actuated. Future work should include investigation of a convex optimization approach which would provide a great deal more flexibility, allowing minimum power or minimum time solutions while respecting joint velocity and torque limits.

2.2 Created TaskSpace helper class

Graduate student Christopher Dembia in Scott Delp's lab has training from Prof. Khatib and experience as a Simbody developer. Working with Simbody's primary author Michael Sherman, Chris developed the Simbody task space API that we employed for this milestone. This API is encapsulated in a new `SimTK::TaskSpace` class, and a family of related subclasses.

Task space control is conveniently described in terms of matrix operations. However, direct computation with joint-space matrices can lead to O(n^3) execution times for matrix multiplications and factoring, where n is the number of joint degrees of freedom (dofs). In many cases what appear to be matrix operations are actually matrix-vector products that can be computed in O(n) time using Simbody's built-in operators for computing Jacobian-vector, Jacobian Transpose-vector, Mass Matrix-vector, and Inverse Mass Matrix-vector products. However, use of those operators makes the code look considerably different than the presentation of task space theory in the literature, or in section 2.1, which can lead to difficulty in translating desired task space control to practical code. The end result is that users are likely simply to form the explicit matrices and then work with those, resulting in clean but unnecessarily slow code. The goal of the API for this project was to allow the code to be written in terms of familiar matrices while the underlying execution would be done transparently in terms of O(n) operators.

To illustrate the utility of this API, we will walk through how one could use `TaskSpace` to implement a typical operational space controller: positioning an end-effector while compensating for gravity. The intent here is to convey the strategy we employed; the details and terminology have been simplified but can be seen in the code on GitHub. for the UR10 and Atlas examples described in the following sections. First, we create an object `ts` of type `TaskSpace`. Then we provide `ts` with a description of a task set containing one or more tasks at the same priority; this description is necessary for computing the priority Jacobian G. Currently, the `TaskSpace` class supports a task set composed of Cartesian positioning tasks (e.g. end-effector placement), and simplifies the gravity compensation task. Point positioning tasks are defined by a link (likely the end-effector) and a fixed point on that link (also known as a *station*) whose location we would like to control. Here is how we create a `TaskSpace` object for a single task of positioning a hand of the Atlas robot:

```
Vec3 endEffectorOffset(1, 0, 0);
TaskSpace ts;
ts.addStationTask(atlas.getLink("r_hand"), endEffectorOffset);
```

The `TaskSpace` class defines a collection of local classes, one for each of the task space matrices, and overloads the multiplication operator so that objects of these types may be used as though they were matrices. But these overloaded operators actually invoke compositions of Simbody's O(n) operators for efficiency. (The explicit matrices can also be formed, which is helpful for debugging.) Here is a sampling of the available subclasses and how they are used.

```
TaskSpace::Jacobian&            G    = ts.getJacobian();
TaskSpace::JacobianTranspose&   GT   = G.T();
```

```
TaskSpace::Inertia&             L    = ts.getInertia();
TaskSpace::NullSpaceTranspose&  NT   = ts.getNullSpace().T();
TaskSpace::Gravity&             Fg   = ts.getGravity();         // Khatib: p
TaskSpace::InertialForces&      Fi   = ts.getInertialForces(); // Khatib: mu
```

We can also provide access to the joint space gravity force:

```
Vector&                         fg   = Fg.g();
```

These objects represent the corresponding matrices but do not themselves contain any explicit matrices; instead, they allow one to perform operations with these matrices. For example, the objects `GT` and `L` can be considered to be operators $G^{\mathrm{T}}\odot$ and $L\odot$.

We use the following control law to drive the right hand to a point P:

```
Vector xdd_desired = -kp * (x - P) - kd * xd;
```

where `x` and `xd` are the current position and velocity of the end-effector point, `P` is the target location, and `kp`, `kd` > 0 are control gains. Now, we perform the task space calculations that will achieve this task. We assume there are no other forces but gravity and inertial forces, and no constraints. Using the notation described earlier, the end-effector task is at priority $p = 2$ and the gravity compensation is at $p = 3$.

```
Vector F2        = L*xdd_desired - (Fg + Fi); // Eqn. 9b
Vector f2        = GT * F2;                  // Eqn. 9a
Vector f3        = NT * (-fg);
Vector ftask     = f2 + f3;
```

To achieve the task space control, we apply the joint forces `ftask` to the robot. (In practice we would adjust that to obey joint torque limits.) The code above shows how the `TaskSpace` class allows us to easily translate our equations from paper to code without sacrificing performance.

We consider the current implementation of `TaskSpace` a proof of concept; we learned a lot while implementing it and intend to refine it as outlined in the Future Work section below.

2.3 Determined the best architecture for task space control in Gazebo

The controller architecture used in this exercise is a network service based requester-responder pair over TCP/IP. On every simulation time step update, the robot makes a blocking request to the Task Space Controller, sending out its states (RobotState) and expects a joint torque command (RobotCommand) in return. The strict real-time fashion control is achieved by blocking simulation advancement on a request while waiting for the controller to return a command for the next time step. The communication was implemented based on ignition transport protocol[11], which has a local network latency of less than 10 micro-seconds, capable of closing the robot control loop at 1kHz with minimal added overhead.

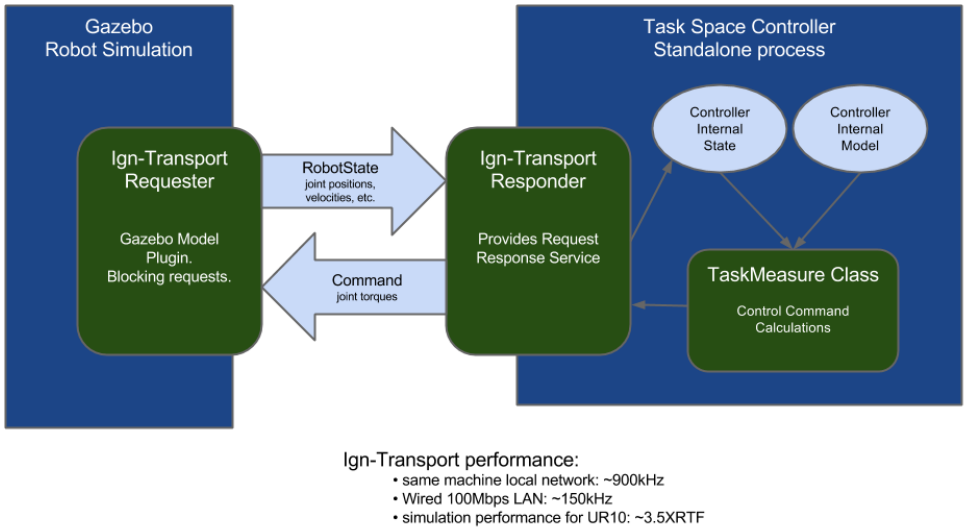

Figure 1: Controller architecture.

2.4 Developed UR10 robot arm example in Simbody

As a first test of the new `SimTK::TaskSpace` class, we implemented an example program controlling a model of the Universal Robotics UR10 6-dof robot arm. The idea was to develop working code in which a task space controller containing a Simbody-based model would control a separate Simbody-based simulation of the UR10. Then the controller alone would be incorporated in a Gazebo controller, which would then control a Gazebo-based simulation of the UR10. For this example, we hand-built an explicit Simbody model of the UR10 in C++ code using data from an existing Gazebo model.

A model-based controller for a physical robot must make do with imperfections in the model and noisy, discretely sampled sensor data. To ensure that our controller did not take advantage of the availability of a perfect model, we sampled the simulated robot's sensors at a specified interval and permitted only those samples to be used by the controller to set the state of its internal model. Also, we added the ability to inject noise into the sensor reading for both pose and rate. See Figure 2 for a screenshot, where the noise-injecting slider controls are visible. With the noise at zero as shown in the screenshot, the controller is able to track the task target extremely closely. With noise injection the tracking can be made very poor. We added optional end-effector sensing, which substantially improves tracking in the presence of noise. When that is enabled, the simulated robot arm samples its end effector location at the same time it samples its joints, and the model applies its control law to the sampled location. Otherwise, the controller queries its internal model (after positioning it with the sampled joint poses), and applies the control law instead to *its* end effector position, which is noisy due to imperfect joint sensing.

See section 2.1 for the notation used below to describe the controller.

There were no constraints in this model (joints are not constraints in Simbody), so the highest priority task is the end-effector positioning task t (at priority $p = 2$). We used the TaskSpace class to calculate the joint torques needed to accomplish that task, and to obtain the null space of that task. In this case task t has dimensionality $d_t = 3$ (a point location), and the robot has 6 dofs, so the null space still has rank 3 and can be used to achieve other goals. In this example, we determined gravity compensation and joint damping torques to achieve stable positioning, and projected those into the task null space before applying them.

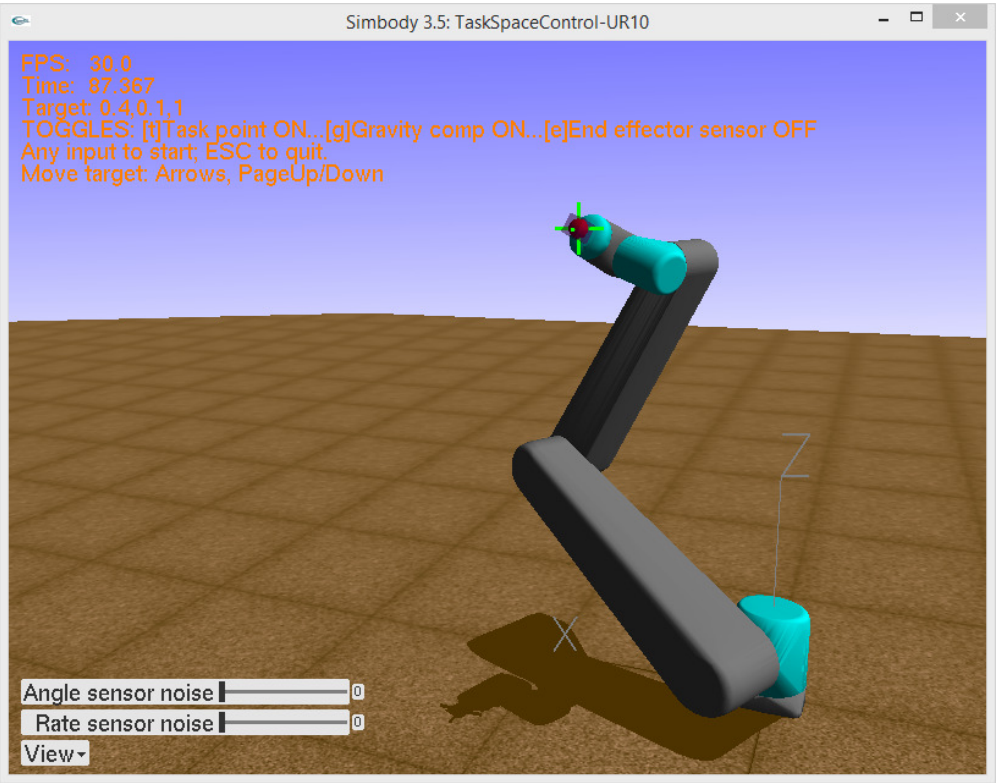

Figure 2: Screenshot of Simbody UR10 task space control example `TaskSpaceControl-UR10`. The green cross marks the end effector location; the red sphere marks the target location for the commanded task and can be moved interactively. Gravity compensation is active in the task null space.

For the end-effector positioning task, we have task Jacobian J_t, and since this is the only task at priority $p = 2$ we have $G_2 = J_t$. The task is given by $x_2^{\text{desired}} = P$ where P is the ground frame location of the target point as positioned interactively by the user. We would like the end-effector acceleration and velocity to settle to zero when the task is achieved. We use the following PD control law to specify the desired acceleration:

$$\ddot{x}_2^{\text{desired}} = -k_p(x_2^{\text{sampled}} - P) - k_d\dot{x}_2 \tag{11}$$

where x_2^{sampled} is either the sensed end-effector location if enabled, or the estimated location based on the joint angle samples applied to the controller's internal model. $k_p, k_d > 0$ are control gains which we chose to provide critical damping.

Since the end effector will already have an acceleration due to gravity and inertial forces, we have

$$\ddot{x}_2^- = G_2\dot{u}_0 + \dot{G}_2 u \tag{12a}$$
$$\Delta\ddot{x}_2 = \ddot{x}_2^{\text{desired}} - \ddot{x}_2^- \tag{12b}$$

The `SimTK::TaskSpace` class provides us with task space operators, so we are able to solve for the joint torques f_2 using the method presented in Eqns. 9:

$$F_2 = L_2\Delta\ddot{x}_2 \tag{13a}$$
$$f_2 = G_2^{\mathrm{T}} F_2 \tag{13b}$$

The force of gravity on each body is given by F_{gravity} in Eqn. 2c. This is mapped into joint space torques f_{gravity} using the system Jacobian:

$$f_{\text{gravity}} = J^{\text{T}} F_{\text{gravity}} \tag{14}$$

Then we can cancel gravity by applying joint torques of $-f_{\text{gravity}}$. Joint damping torques f_{damping} are calculated to oppose joint velocities, here using a single gain $c > 0$ for all the joints:

$$f_{\text{damping}} = -cu \tag{15}$$

These are in joint space, so need only be projected into the null space of the end-effector task:

$$f_3 = N_2^{\text{T}}(f_{\text{damping}} - f_{\text{gravity}}) \tag{16}$$

giving $f = f_2 + f_3$ as the total task joint torques. These may exceed the allowed limits on the joint actuators, so we finally set $f_{\text{task}} = \text{clamp}(f)$ where each joint torque is moved to its nearest limit if it is out of range. f_{task} is the set of joint actuator torques returned by the controller to be applied to the simulated robot arm.

The example program is called `TaskSpaceControl-UR10` and has been checked into the Simbody master branch on GitHub. It will be included with the Simbody 3.5 release.

2.5 Ported the UR10 controller to Gazebo

As described in Section 2.3 and illustrated in Fig. 1, the Task Space Controller is implemented as a standalone process with a message-passing interface over the ignition transport middleware. The standalone process maintains an internal Simbody model for the Task Space Controller and listens for request messages that provide a state update. With the provided state update, the controller returns joint torques computed by the controller. The controller can be simulated in Gazebo with a plugin that sends the current robot state and then applies the received joint torques.

With the framework in place, transferring the controller code from the Simbody UR10 examples was a relatively straightforward cut-and-paste operation. OSRF and Stanford participants worked side-by-side to do this together and it went very smoothly as a result. Errors included an off-by-one in the communication protocol for sending the sampled joint information to the controller and the control forces back. We also discovered minor errors in the hand translation that had been done to produce the Simbody C++ model of the UR10; several joint angle offsets were at $+90^\circ$ rather than -90°. We fixed those and then the controller worked very well, resulting in behavior in Gazebo indistinguishable from the behavior in Simbody. We were able to use ODE, Bullet, or Simbody in Gazebo to execute the arm simulation, while the controller always used its embedded Simbody model.

Here is a link to an example of the UR10 task space controller in action:

https://www.youtube.com/watch?v=1zwisgod_uM.

2.6 Developed Atlas example in Simbody

Since we already had trouble hand-crafting a Simbody model for the simple UR10, it was clear that approach would not work for the much larger Atlas model. So the first step here was to create a Simbody reader for the ROS urdf-format Atlas model that was being used in Gazebo. We were able to adapt much of the code from the sdf-format reader

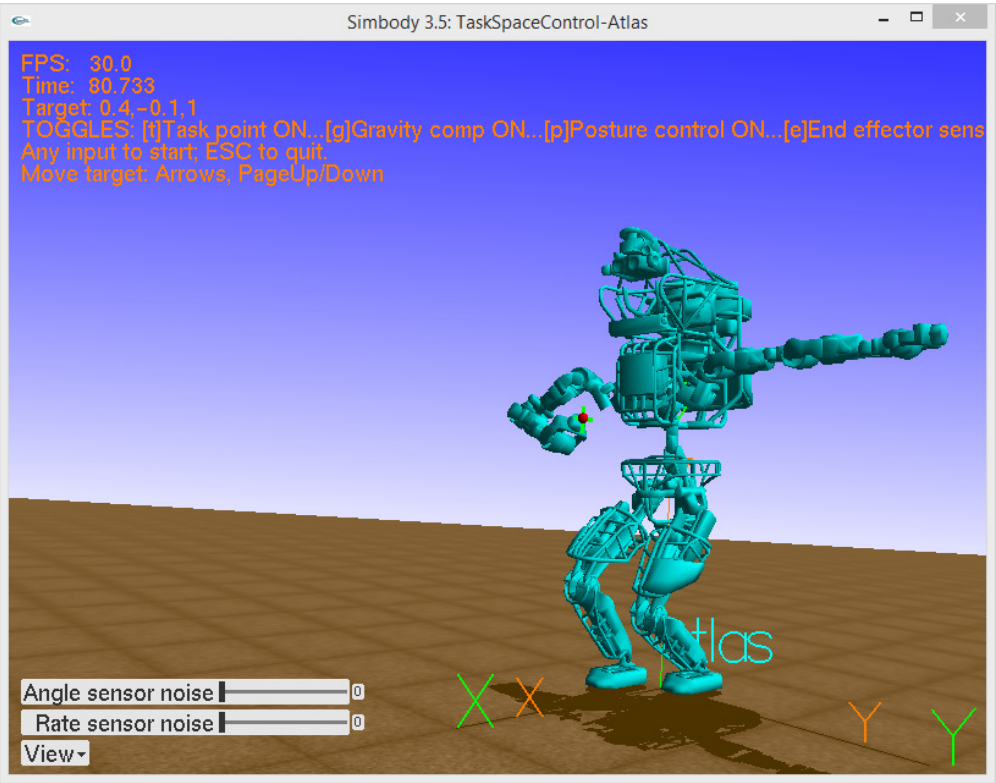

Figure 3: Screenshot of the Simbody Atlas task space control example `TaskSpaceControl-Atlas`. The green cross on the right hand marks the end effector location; the red sphere marks the target location for the commanded task and can be moved interactively. Upper body posture control and gravity compensation are active in the task null space. The commanded neutral posture is cruciform and is achieved for the head, torso, and left arm. The lower body is separately controlled with the feet held in ground contact and the pelvis moving in a sinusoidal pattern.

developed for a previous milestone under this subcontract. The urdf reader permitted us to build the Simbody model automatically and avoid human error in the process.

Our intention was to develop a reaching and posture controller for the Atlas upper body, while employing separate control for the lower body from pelvis down. So in this case the controller model was built from only the upper-body half of the Atlas urdf, while the simulated Atlas used the whole model. The upper body model has its pelvis welded to the ground, but expects to receive the actual pelvis pose as a sampled output of the simulated model. That pose is then used to adjust the target position and gravity direction relative to the pelvis so that the controller's model correctly reflects the lower body pose of the simulated robot. We did not adjust for the pelvis acceleration, which we assumed would be small.

To emulate a controlled lower body in the simulated robot, we welded the feet to the floor and applied sinusoidal prescribed motion to the pelvis, in all translational and rotation directions. This enabled us to test that the controller's model was being properly configured from the sampled pose and was able to track the effectively moving target and varying gravity vector. Figure 3 shows a screenshot. Without noise, end effector tracking was very good, although the lack of compensation for pelvis acceleration caused noticeable errors if the pelvis acceleration was set to be very high. This could be compensated by sampling or estimating pelvis acceleration, although we did not do that.

The control law here is similar to the UR10 control described above, however it introduced a new joint-space task for

posture control. In this case we chose to prefer a neutral pose, which we defined as one in which all joint angles are zero. For the Atlas model definition, that corresponds to a cruciform posture with torso and head erect and arms held horizontal. The posture control, gravity compensation, and/or joint damping were combined and projected into the end-effector null space.

Because of the widely varying mass properties in the links of the humanoid robot, we discovered that uniform feedback gains for the posture and damping terms were unstable. To address that, we used Simbody's O(n) $M\odot$ multiply-by-mass-matrix operator to mass-weight the gains, allowing us to choose only a single position gain k and velocity gain c while effectively applying joint-specific gains. That proved to be very stable. So the control law for the end-effector task is identical to that presented in Eqns. 12 and 13 to yield the end-effector task joint torques f_2. The lower-priority joint space task was modified as follows:

$$f_{\text{damping}} = -c(M \odot u) \tag{17}$$

$$f_{\text{posture}} = -k(M \odot q) \tag{18}$$

$$f_3 = N_2^{\text{T}}(f_{\text{posture}} + f_{\text{damping}} - f_{\text{gravity}}) \tag{19}$$

where f_{posture} is the vector of joint torques for enforcing the neutral posture, and f_{gravity} is the joint torque due to gravity as defined in Eqn. 14. The scaling of the joint angles q in this case is identical to scaling of u because we have $\dot{q} = u$ in this model; Simbody provides an appropriate O(n) mapping that could be used here in the general case where $\dot{q} \neq u$. Finally, as before we clamp the joint torques to limit the maximum values, and set $f_{\text{task}} = \text{clamp}(f_2 + f_3)$, and return f_{task} as the set of control torques to apply to the simulator robot.

The example program is called `TaskSpaceControl-Atlas` and has been checked into the Simbody master branch on GitHub. It will be included with the Simbody 3.5 release.

2.7 Ported the Atlas controller to Gazebo

OSRF and Stanford participants again performed this port working side-by-side, and our earlier experience with the UR10 made this smoother. There were no problems with the Simbody vs. Gazebo models since the Simbody model was constructed directly from the same urdf that was used in Gazebo. We revised the selection of joint position and velocity samples in the communication protocol to select only the upper-body joints, and added sensing and communication of the pelvis pose.

The controller was unstable at first, which turned out to be due to other control forces being applied to the Gazebo model that were not known to the controller. Once these were found and disabled the controller worked as it did in the Simbody example. This does emphasize that a model-based controller is very sensitive to the quality of its embedded model, and needs to be aware of all the forces acting on the controlled robot so that it can compensate for them. In the above terms, it needs a reasonably accurate estimate of f_0 so that its estimate of the desired change to task accelerations $\Delta\ddot{x}$ is accurate enough.

With this port complete we were able to achieve the goals of this milestone: demonstrate task-space control of Atlas performing postural tasks.

2.8 Demonstrated task-space and postural control of Atlas

Here is a link to an example of the Atlas upper body task space controller in action:

https://www.youtube.com/watch?v=RbAxK2l5qls.

3 Future work

The `SimTK::TaskSpace` class as implemented works well for one or more potentially-competing, equal priority point-tracking tasks, plus one level of lower-priority joint space tasks such as posture control or gravity compensation, operating in the null space of the point-tracking tasks. While this covers many application areas, for general applicability this class should be extended as follows:

- Bring names and terminology closer in line with the theory presentation.
- Treat model constraints as priority 1 (highest priority) tasks, as described in section 2.1.
- Allow nesting of task priorities to any depth.
- Allow for control of underactuated systems.
- Support force tasks in addition to motion tasks.
- Use fast convex optimization to provide a more general capability to choose the best joint forces, incorporating both performance goals and constraints.

The Task Space Controller process can read a ROS urdf file to specify the model, but still requires hand-programming to implement the details of the controller. This could be extended to include more features from the urdf specification, and the specification could be extended (perhaps using a separate file) to provide the additional information needed by the controller. Provision for automatic generation of the necessary communication protocol could be included. This would lower the barrier to using task space controllers with Gazebo models.

4 Conclusions

We have shown that a model-based task space controller can be used to efficiently control end effector and posture tasks for the UR10 robot arm and Atlas humanoid robot. The same underlying `SimTK::TaskSpace` class was used in both cases, confirming the broad applicability of task space control methods. We successfully used embedded Simbody models to control robots simulated in Gazebo with a variety of physics engines, showing that the controller is independent of the simulation and could thus have been applied to a physical robot just as well. In fact the controller is implemented as a separately-running executable that shares nothing with the simulated robot except a communication protocol.

In addition to its use with Gazebo, several examples were added to the Simbody package demonstrating standalone task space control (that is, using Simbody for the simulated robot also). This will be used from the OpenSim biomechanics application for research into neural control of gait, providing additional testing and validation and continued motivation to enhance Simbody's task space capabilities.

References

[1] O. Khatib, L. Sentis, J. Park, and J. Warren, "Whole-body dynamic behavior and control of human-like robots," *International Journal of Humanoid Robotics*, vol. 1, no. 01, pp. 29–43, 2004.

[2] B. Dariush, M. Gienger, B. Jian, C. Goerick, and K. Fujimura, "Whole body humanoid control from human motion descriptors," in *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pp. 2677–2684, IEEE, 2008.

[3] O. Khatib, E. Demircan, V. De Sapio, L. Sentis, T. Besier, and S. Delp, "Robotics-based synthesis of human motion," *Journal of Physiology-Paris*, vol. 103, no. 3, pp. 211–219, 2009.

[4] L. Sentis, *Synthesis and control of whole-body behaviors in humanoid systems*. PhD thesis, Citeseer, 2007.

[5] O. Khatib, "A unified approach for motion and force control of robot manipulators: The operational space formulation," *Robotics and Automation, IEEE Journal of*, vol. 3, no. 1, pp. 43–53, 1987.

[6] V. De Sapio, O. Khatib, and S. Delp, "Task-level approaches for the control of constrained multibody systems," *Multibody System Dynamics*, vol. 16, no. 1, pp. 73–102, 2006.

[7] M. A. Sherman, A. Seth, and S. L. Delp, "Simbody: multibody dynamics for biomedical research," *Procedia IUTAM*, vol. 2, pp. 241–261, 2011.

[8] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, U.K.: Cambridge University Press, 2004.

[9] A. Domahidi, E. Chu, and S. Boyd, "Ecos: An socp solver for embedded systems," in *Control Conference (ECC), 2013 European*, pp. 3071–3076, IEEE, 2013.

[10] J. Mattingley and S. Boyd, "Cvxgen: a code generator for embedded convex optimization," *Optimization and Engineering*, vol. 13, no. 1, pp. 1–27, 2012.

[11] O. S. R. Foundation, "Ignition Trasnport Software Library."