CS225A: Experimental Robotics
Lecture 9: Equations vs. Code & Trajectory Generation

Samir Menon

May 7, 2015
In this class:

1. Control review & relation to code
In this class:

1. Control review & relation to code
2. Useful tools/libraries
In this class:

1. Control review & relation to code
2. Useful tools/libraries
3. Trajectory generation
1) Review: Theory vs. Code

\[ \begin{align*}
O_x &= O_T \cdot x_{ee} \\
\delta x &= J(q) \delta q \\
\bar{J}^T \ast [A(q)\ddot{q} + b(q, \dot{q}) + g(q)] &= \Gamma
\end{align*} \]
Code: Articulated body model (*static data*)

```cpp
scl::SRobotParsed rds; // Robot data structure...
scl::CParseScl p; // This time, we'll parse the tree from a file...
p.readFile("./R6Cfg.xml","/"("r6bot",rds)); // Reads robot "r6bot" from file "R6Cfg.xml"
// Now rds has all the information specified in the xml file:
```

```cpp
actuator_forces_max_ : VectorXd
actuator_forces_min_ : VectorXd
actuator_sets_ : CMapedPointerList<
                   string, SActuatorSetParsed, bool>
damping_gc_ : VectorXd
dof_ : sUInt
flag_apply_actuator_acc_limits_ : sBool
flag_apply_actuator_force_limits_ : sBool
flag_apply_actuator_pos_limits_ : sBool
flag_apply_actuator_vel_limits_ : sBool
flag_apply_gc_damping_ : sBool
flag_apply_gc_pos_limits_ : sBool
flag_controller_on_ : sBool
flag_logging_on_ : sBool
flag_wireframe_on_ : sBool
gc_pos_default_ : VectorXd
```

Press 'Ctrl+Space' to show Template Proposals
Code: Single rigid body (*static data*)

```cpp
scl::SRobotParsed rds; // Robot data structure...
sc1::CParserScl p; // This time, we'll parse the tree from a file...
p.readRobotFromXml("./R6Cf.xml", ".", "r6bot", rds); // Reads robot "r6bot" from file "R6Cf.xml"
// Now rds has all the information specified in the xml file:

// Rigid body ds: This contains the kinematic structure organized in a tree
// type = sutil::CMappedTree< std::string, SRigidBody>
sc1::SRigidBody *rb = rds.rb_tree_.at("link3"); // We get the link ids from the xml file
rb->
```

- `child_addrs_`: vector<SRigidBody*, allocator<SRigidBody*>>
- `collision_type_`: sInt
- `com_`: Vector3d
- `force_gc_lim_lower_`: sFloat
- `force_gc_lim_upper_`: sFloat
- `friction_gc_kv_`: sFloat
- `gr_child_addrs_`: vector<SRigidBody*, allocator<SRigidBody*>>
- `gr_parent_addrs_`: vector<SRigidBody*, allocator<SRigidBody*>>
- `gr_parent_names_`: vector<SRigidBody*, allocator<SRigidBody*>>
- `has been init_`: sBool
- `inertia_`: Matrix3d
- `inertia_gc_`: sFloat
- `is_root_`: sBool
- `joint_default_pos_`: sFloat

Press 'Ctrl+Space' to show Template Proposals.
Code: Single rigid body *(by numeric id)*

```cpp
csl::SRobotParsed rds;    // Robot data structure...
csl::CParserScl p;        // This time, we'll parse the tree from a file...
p.readRobotFromFile("./R6Cfg.xml","/","r6bot",rds); // Reads robot "r6bot" from file "R6Cfg.xml"
// Now rds has all the information specified in the xml file:

// Rigid body `ds`: This contains the *kinematic* structure organized in a tree
// type = sutil::CMappedTree<std::string, SRigidBody>
 rds.rb_tree;
// Access individual rigid bodies like this:
scl::SRigidBody *rb = rds.rb_tree.at("link3"); // We get the link ids from the xml file

// If you don't know the name of the link:
scl::SRigidBody *rb2 = rds.rb_tree.at(rds.robot_tree_numeric_id_to_name_[3]);
```
Review: Forward kinematic transformations

\[ O_{x_{ee}} = O_{T_{hand}}^{hand} x_{ee} \]

\[ O_{x_{ee}} = O_{T_0}^{0} T_1 \ldots T_h \] ^{h-1} \] ^{hand} x_{ee} \]

\[ i^{-1} T_i = \begin{bmatrix} i^{-1} R_i(q_i) & i^{-1} p_i(q_i) \\ 0^T & 1 \end{bmatrix} \]
scl::SRobotParsed rds; //Robot data structure....
scl::CParserScl p; //This time, we'll parse the tree from a file...
p.readRobotFromFile("./R6Cfg.xml","/","r6bot",rds); //Reads robot "r6bot" from file "R6Cfg.xml"

// Can use parsed information to set up the dynamics data structure
scl::SGcModel rgcm; //Robot data structure with dynamic quantities...
rgcm.init(rds); //Simple way to set up dynamic tree...
// Now rgcm has placeholders for all the kinematics and dynamics information
Code: Rigid body (dynamic data)

```cpp
scl::SRobotParsed rds; // Robot data structure...
scl::CParseScl p; // This time, we'll parse the tree from a file...
p.readRobotFromfile("./R6Cfg.xml","/","r6bot",rds); // Reads robot "r6bot" from file "R6Cfg.xml"

// Can use parsed information to set up the dynamics data structure
scl::SGcmModel rgcm; // Robot data structure with dynamic quantities...
rgcm.init(rds); // Simple way to set up dynamic tree...
// Now rgcm has placeholders for all the kinematics and dynamics information

// Rigid body dyn_ds: This contains the kinematic transforms organized in a tree
// type = sutil::CMappedTree<std::string, SRigidBodyDyn>
rgcm.rbdyn_tree;
// Access individual rigid bodies like this:
scl::SRigidBodyDyn *rbd = rgcm.rbdyn_tree_.at("link3");
```

- `J_com_`: MatrixXd
- `T_Lnk_`: Affine3d
- `T_o_Lnk_`: Affine3d
- `child_addr_`: vector<SRigidBodyDyn *, allocator<SRigidBodyDyn *>>
- `gr_child_addr_`: vector<SRigidBodyDyn *, allocator<SRigidBodyDyn *>>
- `gr_parent_addr_`: vector<SRigidBodyDyn *, allocator<SRigidBodyDyn *>>
- `gr_parent_names_`: vector<SR rigidBodyDyn *, allocator<SR rigidBodyDyn *>>
- `has been init_`: sBool
- `link_ds_`: const SRigidBody *
- `name_`: string
- `parent_addr_`: SRigidBodyDyn *
- `parent_name_`: string
- `q_T_`: sFloat
- `sp_S_joint_`: MatrixXd
- `sp_Sorth_joint_`: MatrixXd

Press 'Ctrl+Space' to show Template Proposals
Review: Jacobians

\[ \delta x = J(q) \delta q \]

\[ J(q) = \begin{bmatrix}
\frac{\partial x}{\partial q_0} & \frac{\partial x}{\partial q_1} & \cdots & \frac{\partial x}{\partial q_n} \\
\frac{\partial y}{\partial q_0} & \frac{\partial y}{\partial q_1} & \cdots & \frac{\partial y}{\partial q_n} \\
\frac{\partial z}{\partial q_0} & \frac{\partial z}{\partial q_1} & \cdots & \frac{\partial z}{\partial q_n}
\end{bmatrix} \]
Code: Dynamics setup (*computations*)

```cpp
scl::SRobotParsed rds;  // Robot data structure...
scl::CParserScl p;      // This time, we'll parse the tree from a file...
p.readFile("./R6Cfg.xml","./","r6bot",rds);  // Reads robot "r6bot" from file "R6Cfg.xml"
scl::SGcModel rgcm;     // Robot data structure with dynamic quantities...
rgcm.init(rds);         // Simple way to set up dynamic tree...

// Need a dynamics computation object now. Note the C prefix instead of the S prefix
scl::CDynamicsScl dyn_scl;  // Robot *kinematics* and dynamics computation object...
dyn_scl.init(rds);
// Now dyn_scl is ready to compute dynamic quantities
```
Code: Dynamics setup (robot state)

```c
scl::SRobotParsed rds;  // Robot data structure...
scl::CParserScl p;      // This time, we'll parse the tree from a file...
p.readRobotFromFile("./R6Cfg.xml","/","r6bot",rds);  // Reads robot "r6bot" from file "R6Cfg.xml"
scl::SGcModel rgcm;     // Robot data structure with dynamic quantities...
rgcm.init(rds);         // Simple way to set up dynamic tree...

// Need a dynamics computation object now. Note the C prefix instead of the S prefix
scl::CDynamicsScl dyn_scl;  // Robot kinematics and dynamics computation object...
dyn_scl.init(rds);

// Also need some state (joint angles etc.) at which to compute the dynamic model
scl::SRobotIO rio;        // I/O data structure
rio.init(rds);
```
Code: Compute a Jacobian

```cpp
scl::SRobotParsed rds;  // Robot data structure...
scl::CParseScl p;      // This time, we'll parse the tree from a file...
p.readFile("./R6Cfg.xml", "./", "r6bot", rds); // Reads robot "r6bot" from file "R6Cfg.xml"
scl::SGcModel rgcm;   // Robot data structure with dynamic quantities...
rgcm.init(rds);       // Simple way to set up dynamic tree...

// Need a dynamics computation object now. Note the C prefix instead of the S prefix
scl::CDynamicsScl dyn_scl; // Robot kinematics and dynamics computation object...
dyn_scl.init(rds);

// Also need some state (joint angles etc.) at which to compute the dynamic model
scl::SRobotIO rio;    // I/O data structure
rio.init(rds);

// Now dyn_scl is ready to compute dynamic quantities.
// Let's begin by computing a Jacobian at a position (0,0,0.1) wrt. a rigid body
// (and we'll refresh the kinematic transformation matrices to be
// sure we don't get a stale Jacobian)...
Eigen::MatrixXd I_am_a_Jacobian_duh;
scl::SRigidBodyDyn *rbd = rgcm.rbDynTree.at("link3");
dyn_scl.computeJacobianWithTransforms(I_am_a_Jacobian_duh, *rbd, rio.sensors.q,
       Eigen::Vector3d(0,0,0.1));
```
Review: Dynamics

\[ A(q)\ddot{q} + b(q, \dot{q}) + g(q) = \Gamma \]
Code : Dynamics

```cpp
scl::SRobotParsed rds; //Robot data structure....
scl::CParserScl p; //This time, we'll parse the tree from a file...
p.readRobotFromFile("./R6Cfg.xml", "/", "r6bot", rds); //Reads robot "r6bot" from file "R6Cfg.xml"
scl::SGcModel rgcm; //Robot data structure with dynamic quantities...
rgcm.init(rds); //Simple way to set up dynamic tree...

// Need a dynamics computation object now. Note the C prefix instead of the S prefix
scl::CDynamicsScl dyn_scl; //Robot kinematics and dynamics computation object...
dyn_scl.init(rds);

// Also need some state (joint angles etc.) at which to compute the dynamic model
scl::SRobotIO rio; //I/O data structure
rio.init(rds);

// Let's now compute the entire gc model
dyn_scl.computeGCModel(&rio, &rgcm); // (that's it folks)

// You may now use anything inside rgc
```

- `M_g_c_` : MatrixXd
- `M_g_c_inv_` : MatrixXd
- `computed_spatial_transformation_and_inertia_` : bool
- `dq_` : VectorXd
- `force_g_c_cc_` : VectorXd
- `force_g_c_grav_` : VectorXd
- `has been init_` : sBool
- `mass_` : sFloat
- `name_` : string
- `name_robot_` : string
- `pos_com_` : Vector3d
- `processing_order_` : vector<string, allocator<string>>
- `q_` : VectorXd
- `rbdyn_tree_` : CMappedTree<string, SRigidBodyDyn>
- `type_` : string

Press 'Ctrl+Space' to show Template Proposals
Code: An important omission

• The code we wrote was correct
  • Drawn from tutorial 4

• But we omitted all the error checks (*for brevity*)
  • Bad idea!

• Most of that code won't work
  • Simple error: 'link3' is actually 'link_3'
  • When things don't work, break them down
  • Always check for errors!
Review: Task space dynamics

\[ \bar{J}^T \star [A(q)\ddot{q} + b(q, \dot{q}) + g(q) = \Gamma] \]

\[ \Lambda(q)\dddot{x} + \mu(q, \dot{q}) + p(q) = F_x \]
Set up new control task (computation)

Advanced tutorial:

scl.git/tutorial/scl_advanced_creating_new_tasks
Set up new control task (data structures)

Advanced tutorial:
scl.git/tutorial/scl_advanced_creating_new_tasks
Code: Task dynamics \((\text{computeModel}())\)

\[
\Lambda(q)\ddot{x} + \mu(q, \dot{q}) + p(q) = F_x
\]

```c
const SGcModel* gcm = data_->gc_model_;  
flag = flag && dynamics_->computeJacobian(data_->J_,*(data_->rbd_),  
arg_sensors->q_,data_->pos_in_parent_);  
//Use the position jacobian only. This is an op-point task.  
data_->J_ = data_->J_.block(0,0,3,data_->robot_->dof_);  
//Operational space mass/KE matrix:  
//Lambda = (J * Ainv * J')^-1  
data_->M_task_inv_ = data_->J_ * gcm->M_gc_inv_ * data_->J_.transpose();
```
Code: Inverting $\Lambda$ is complicated

$$\Lambda(q) \ddot{x} + \mu(q, \dot{q}) + p(q) = F_x$$

```cpp
const SGcModel* gcm = data_->gc_model_

flag = flag && dynamics_->computeJacobian(data_->J_, *(data_->rbd_),
arg_sensors->q_, data_->pos_in_parent);

// Use the position jacobian only. This is an op-point task.
data_->J_ = data_->J_.block(0,0,3,data_->robot->dof_);

// Operational space mass/KE matrix:
// Lambda = (J * Ainv * J')^-1
data_->M_task_inv_ = data_->J_ * gcm->M_gc_inv_ * data_->J_.transpose();
```
Code: Inverting $\Lambda$ (if invertible)

\[ \Lambda(q) \ddot{x} + \mu(q, \dot{q}) + p(q) = F_x \]

//The general inverse function works very well for op-point controllers. //3x3 matrix inversion behaves quite well. Even near singularities where //singular values go down to ~0.001. If the model is coarse, use a n-k rank //approximation with the SVD for a k rank loss in a singularity.
qr_.compute(data_→M_task_inv_);
if(qr_.isInvertible())
    { data_→M_task_ = qr_.inverse(); }
else
    { use_svd_for_lambda_inv_ = true; }
Code: Inverting Λ (if not invertible)

// Use a Jacobi svd. No preconditioner is required coz lambda inv is square.
// NOTE: This is slower and generally performs worse than the simple inversion
// for small (3x3) matrices that are usually used in op-space controllers.
svd_.compute(data_->M task inv, 
   Eigen::ComputeFullU | Eigen::ComputeFullV | Eigen::ColPivHouseholderQRPreconditioner);

int rank_loss=0;

// NOTE: A threshold of .005 works quite well for most robots.
// Experimentally determined: Take the robot to a singularity
// and observe the response as you allow the min singular values
// to decrease. Stop when the robot starts to go unstable.
// NOTE: This also strongly depends on how good your model is
// and how fast you update it. A bad model will require higher
// thresholds and will result in coarse motions. A better model
// will allow much lower thresholds and will result in smooth
// motions.
if(svd_.singularValues()(0) > 0.005)
    { singular_values_(0,0) = 1.0/svd_.singularValues()(0); } 
else { singular_values_(0,0) = 0.0; rank_loss++; }
if(svd_.singularValues()(1) > 0.005)
    { singular_values_(1,1) = 1.0/svd_.singularValues()(1); } 
else { singular_values_(1,1) = 0.0; rank_loss++; }
if(svd_.singularValues()(2) > 0.005)
    { singular_values_(2,2) = 1.0/svd_.singularValues()(2); } 
else { singular_values_(2,2) = 0.0; rank_loss++; }
if(0 < rank_loss)
    { std::cout<<"\nCTaskOpPos::computeModel() : Warning. Lambda inv is ill conditioned. SVD rank loss (@.005) = "<<rank_loss; }

data_->M task_ = svd_.matrixV() * singular_values_ * svd_.matrixU().transpose();
Code: Inverting J (& null space)

\[ \bar{J}^T \ast [A(q)\ddot{q} + b(q, \dot{q}) + g(q) = \Gamma] \]

```c++
// Compute the Jacobian dynamically consistent generalized inverse:
// J\_dyn\_inv = Ainv * J' (J * Ainv * J')^-1
data\_->J\_dyn\_inv\_ = gcm\_->M\_gc\_inv\_ * data\_->J\_.transpose() * data\_->M\_task\_;

// J' * J\_dyn\_inv'
// sUInt dof = data\_->robot\_->dof;
data\_->null\_space\_ = Eigen::MatrixXd::Identity(dof, dof) -
\bar{data\_->J\_.transpose()} * data\_->J\_dyn\_inv\_.transpose();
```
Code: All this is computed in a “task”

\[ \bar{J}^T \ast [A(q)\ddot{q} + b(q, \dot{q}) + g(q) = \Gamma] \]

\[ \Lambda(q)\ddot{x} + \mu(q, \dot{q}) + p(q) = F_x \]
Code: All this is computed in a “task”

\[
J^T \ast [A(q)\ddot{q} + b(q, \dot{q}) + g(q) = \Gamma] \\
\Lambda(q)\ddot{x} + \mu(q, \dot{q}) + p(q) = F_x
\]

- Drawn from scl_tutorial_advanced Creating_new_tasks
- You'll have to compute this for your task
  - Else, use standard tasks
  - $ls scl.git/src/scl/control/task/tasks/C*hpp
Code: All this is computed in a “task”

\[ \bar{J}^T \ast [A(q)\ddot{q} + b(q, \dot{q}) + g(q) = \Gamma] \]

\[ \Lambda(q)\ddot{x} + \mu(q, \dot{q}) + p(q) = F_x \]

- Drawn from scl_tutorial_advanced_creating_new_tasks

- You'll have to compute this for your task
  - Else, use standard tasks
    - $\texttt{ls scl.git/src/scl/control/task/tasks/C*hpp}$

```
samir@iconoclast:~/Code/control/scl-manips-v2.git$ ls src/scl/control/task/tasks/C*hpp
src/scl/control/task/tasks/CTaskComPos.hpp
src/scl/control/task/tasks/CTaskGc.hpp
src/scl/control/task/tasks/CTaskGcLimitCentering.hpp
src/scl/control/task/tasks/CTaskGcSet.hpp
src/scl/control/task/tasks/CTaskKNULL.hpp
src/scl/control/task/tasks/CTaskNullSpaceDamping.hpp
src/scl/control/task/tasks/CTaskOpPos.hpp
src/scl/control/task/tasks/CTaskOpPosNoGravity.hpp
src/scl/control/task/tasks/CTaskOpPosPIDA1OrderInfTime.hpp
```
Any questions...

\[ O_x = O_T x \]

\[ \delta x = J(q) \delta q \]

\[ J^T \ast [A(q)\ddot{q} + b(q, \dot{q}) + g(q) = \Gamma] \]
2) Useful tools & libraries

Enter the heading, attitude and bank, in degrees, below. Then click calculate

<table>
<thead>
<tr>
<th>heading</th>
<th>0</th>
<th>degrees</th>
</tr>
</thead>
<tbody>
<tr>
<td>attitude</td>
<td>0</td>
<td>degrees</td>
</tr>
<tr>
<td>bank</td>
<td>90</td>
<td>degrees</td>
</tr>
</tbody>
</table>

Y
Z
X

<table>
<thead>
<tr>
<th>real</th>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7071067811865476</td>
<td>0.7071067811865475</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Note: In the code, the convention is (i, j, k, real)
Mesh conversion

Introduction

meshconv converts to and from several popular 3D file formats. Currently only geometry conversion is supported.

Features

- input and output formats:
  - nearly 100% VRML 2.0 support (input 2.0, output 1.0 and 2.0)
  - simple geometry only support for Wavefront OBJ, Geomview OFF, Autocad DXF, PLY, 3DS, three.js JSON, and STL

Download

version 1.16, added 27 Feb 2015

- Linux 64 bit executable
  - compiled under CentOS 6.4, kernel 2.6.32, let me know if it doesn’t work for you
- Windows executable
  - 64 bit, compiled using a cross-compiler
- Mac OS X (Mavericks) executable
  - compiled under OS X 10.9.2

(the binaries have been compressed using the UPX executable compressor)(they are self-decompressing)
(after download on Linux/Mac, you may have to do chmod 755 meshconv first)

Usage

Run meshconv without parameters for a usage summary.

Example: to convert a VRML 2.0 model to PLY, triangulated:
meshconv -c ply -tr1 mymodel.wrl

Note to Windows users: meshconv is a commandline tool, which means it must be run from a command shell. In Windows, start a command shell with Start -> Run -> cmd (enter), then use the cd command to change to the folder where you have meshconv.exe and the model you want to convert.

FAQ

Q: can you publish the source code for meshconv?
A: sorry, the program is part of a large codebase, it would take too much of my time to make it open source.

Credit

If you use meshconv for your (published) work, please add a reference to me and to this web page. I’d love to hear what you use meshconv for as well.
3) Trajectory generation *(time-optimal)*
3) Trajectory generation (*time-optimal*)

Type II: Already integrated in scl (3rdparty/otgtypeii)

Type IV: Download from piazza link
3) Trajectory generation (velocity fields)

For details, contact Mohi Khansari
<khansari@stanford.edu>
That's all folks!