Case Study: Gravity Compensation with the Sarcos Dexterous Master Arm

+ A Gravity Compensation Control Circuit
  ♦ Primary goals and subgoals
  ♦ Math and Algorithms
  ♦ Automatic C-code generation with mathematica

+ How to embed the controller in the VxWorks environment
  ♦ Spinal-Cord: the low level I/O and negative feedback processor
  ♦ Interprocessor communication (semaphore, shared semaphores, shared objects)
  ♦ Parietal-Cortex: the task level control processor
  ♦ Creating a task program

Reading Assignment for Next Class
  ♦ See http://www-slab.usc.edu/courses/CS545
Goals of Gravity Compensation

- Use the robot arm as a force reflecting manipulandum
  - Eliminate the weight due to gravity by supplying the appropriate feedforward commands at every moment of time
  - Afterwards, impose (program!) a virtual environment:
    + E.g., a “honey sphere” (in Cartesian Space!)
      - Inside of the sphere, impose viscous friction opposing the movement
      - Outside of the sphere, no viscous friction

- How dangerous is it to program this task?
- How would you do it?
Theory Part I: Gravity Compensation

• At every time step:
  – Read current positions from sensors
  – Calculate inverse dynamics feedforward torque
Control Loop on VxWorks
Gravity Compensation (cont’d)

- The Gravity Compensation Control Law

\[ B(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) = \tau \]

\[ B(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) = G(q) + K_P(q_d - q) + K_D(q_d - \dot{q}) \]

\[ \tau = G(q) + K_P(q_d - q) + K_D(q_d - \dot{q}) \]

- What is the desired position and velocity for the PD controller?
- What are the PD gains?
Gravity Compensation (cont’d)

• How to obtain $G(q)$?
  – Lagrange
  – Newton-Euler

• How to get the open parameters in $G(q)$?
  – Need mass and center of mass
  – Measure
  – Estimate
  – Estimate from data with regression methods
Automatic Generation of Inverse Dynamics

• Use Mathematica
  – Most important: Mathematica uses shift-return to execute commands
  – The relevant files: RigidBodyDynamics.m and arm2D.dyn will be made available on the web in HW IV.

Set the current working directory to the directory where the file RigidBodyDynamics.m is:

```
```

Load the Rigid Body Dynamics Package:

```
SetDirectory["ControlTheory"];
<< RigidBodyDynamics.m
```

Reset the path to the current directory

```
ResetDirectory[];
```
Automatic Generation of $G(q)$ (cont'd)

ResetDirectory[];

Get some help information about this package:

?InvDyn

InvDyn[infile, outfile, gravity] derives the inverse dynamics equations from the specification
in infile and dumps C-code output to outfile. The gravity vector in world coordinates is given (note that the gravity is supposed to be given WITH the appropriate sign!). The following rules apply:

- input files are in Mathematica notation and can use Mathematica symbolic math
- joints in the input file are numbered by integer numbers. DO NOT use the number 0 as it is used internally to refer to the base coordinate system. The numbers provided will be used as indices for arrays in the C-Code.
- branches are permitted, but no loops.
- each joint must rotate about one defined axis in its local coordinate system
- each local coordinate system has origin at the joint
- the inertia tensor is in the center of mass coordinate system
- rotation angles for coordinate transformation are alpha (rotate about x-axis),
  beta (rotate about y-axis), gamma (rotate about z-axis) in this sequence, and in Euler angle notation
- do NEVER use underscores and dashes in variable names in the input file (Mathematica syntax)
- the rotation angles to get to the next local coordinate systems should be numerical (otherwise too much code, although this could be made more efficient)

Here comes a quick example how to use these functions. "arm2D.dyn" is a special input file that the user needs to generate manually. "arm2D" is the prefix that all generated C-code files will have. \{0,0,G\} is the direction of the gravity vector.

InvDyn["arm2D.dyn", "arm2D", \{0, 0, G\}];
The Structure of the Input File (*.dyn)

```plaintext
{
    {jointID, {ID=1}},
    {jointAxis, {0,0,1}},
    {translation, {0,0,0}},
    {rotationMatrix, {0,0,0}},
    {successors, {2}},
    {inertia, {{j111, j112, j113}, {j112, j122, j123}, {j113, j123, j133}}},
    {centerMass, {xcm1, ycm1, zcm1}},
    {mass, {m1}},
    {jointVariables, {th1, th1d, th1dd, torque1, tex1}},
    {extForce, {0,0,0,0,0}}
}
{
    {jointID, {ID=2}},
    {jointAxis, {0,0,1}},
    {translation, {0,-l1,0}},
    {rotationMatrix, {0,0,0}},
    {successors, {}},
    {inertia, {{j211, j212, j213}, {j212, j222, j223}, {j213, j223, j233}}},
    {centerMass, {xcm2, ycm2, zcm2}},
    {mass, {m2}},
    {jointVariables, {th2, th2d, th2dd, torque2, tex2}},
    {extForce, {0,0,0,0,0}}
}
```
For Gravity Compensation:
\[ \text{thd}^* = \text{thdd}^* = 0! \]

```plaintext
{
    {jointID, {ID=1}},
    {jointAxis, {0, 0, 1}},
    {translation, {0, 0, 0}},
    {rotationMatrix, {0, 0, 0}},
    {successors, {2}},
    {inertia, {{j111, j112, j113}, {j112, j122, j123}, {j113, j123, j133}}},
    {centerMass, {x_cm1, y_cm1, z_cm1}},
    {mass, {m1}},
    {jointVariables, {th1, 0, 0, torque1, 0}},
    {extForce, {0, 0, 0, 0, 0}}
}

{
    {jointID, {ID=2}},
    {jointAxis, {0, 0, 1}},
    {translation, {0, -l1, 0}},
    {rotationMatrix, {0, 0, 0}},
    {successors, {}},
    {inertia, {{j211, j212, j213}, {j212, j222, j223}, {j213, j223, j233}}},
    {centerMass, {x_cm2, y_cm2, z_cm2}},
    {mass, {m2}},
    {jointVariables, {th2, 0, 0, torque2, 0}},
    {extForce, {0, 0, 0, 0, 0}}
}
```
The Output Files of InvDyn:

- See file arm2D_InvDyn_math.h
- See file arm2D_gcomp_InvDyn_math.h
- See file arm2D_InvDyn_declare.h
- See file arm2D_gcomp_InvDyn_declare.h
What to do with these files?

```c
void compute_gcomp(double *th, double *mass, double *xcm, double *ycm, double *zcm, double *torque)
{
    #include "arm2D_gcomp_InvDyn_declare.h"
    double th1,th2;
    double xcm1,xcm2,ycm1,ycm2,zcm1,zcm2;
    double m1,m2;
    double l1=1.0;

    th1=th[1];
    th2=th[2];
    xcm1=xcm[1];
    xcm2=xcm[2];
    ycm1=ycm[1];
    ycm2=ycm[2];
    zcm1=zcm[1];
    zcm2=zcm[2];
    m1 = mass[1];
    m2 = mass[2];

    #include "arm2D_gcomp_InvDyn_math.h"

    torque[1] = torque1;
    torque[2] = torque2;
}
```
Some Shortcuts to Make Things Easier

```{jointID, {ID=1}},
{jointAxis, {0, 0, 1}},
{translation, {0, 0, 0}},
{rotationMatrix, {0, 0, 0}},
{successors, {2}},
{inertia, GenInertiaMatrixA["Inertia", ID]},
{centerMass, GenCMVectorA["cm", ID]},
{mass, GenMassA["m", ID]},
{jointVariables, {th[[1]], 0, 0, torque[[1]], 0}},
{extForce, {0, 0, 0, 0, 0}}
}
{jointID, {ID=2}},
{jointAxis, {0, 0, 1}},
{translation, {0, -l1, 0}},
{rotationMatrix, {0, 0, 0}},
{successors, {}},
{inertia, GenInertiaMatrixA["Inertia", ID]},
{centerMass, GenCMVectorA["cm", ID]},
{mass, GenMassA["m", ID]},
{jointVariables, {th[[2]], 0, 0, torque[[2]], 0}},
{extForce, {0, 0, 0, 0, 0}}
}`
The C-Program becomes

```c
void compute_gcomp(double *th, double *m, double **cm, double *torque)
{
#include "arm2D_gcomp_InvDyn_declare.h"

#include "arm2D_gcomp_InvDyn_math.h"

}
```
How To Program The “Honey Sphere”? 

• In Joint Coordinates:
  – Within a certain joint angle range of each DOF, add a negative component to the feedforward command proportional to the current DOF velocity

• In Cartesian Coordinates:
  – Check whether the endeffector is in the sphere
  – If yes, calculate viscous friction force according to endeffector velocity
  – Convert viscous force into joint torques with Jacobian Transpose
  – A “cheap version”: turn on viscous force in joint space if the endeffector is in the Cartesian sphere