

(Spring 2007-08)

PUMA Simulation & Control Software

Most of the CS225A lab assignments involve writing controllers for a PUMA. These controllers will run in a software package that can be compiled for either simulation or PUMA control.

The simulator can be compiled on a Windows machine with Visual Studio 2003 .NET. All lab assignments should be prepared and tested using the simulator before each group's lab time.

Getting Started

Before you start lab #2, you should spend 15-30 minutes getting familiar with the simulator:

1. If no VS 2003 .NET is installed on your computer, download the installation executable by visiting "<http://www.stanford.edu/class/cs/compilers/>".
2. Download Puma Simulator from "<http://www.stanford.edu/class/cs225a/download/cs225aSimulator.zip>" and extract it on a directory of your computer.
3. Open the VS project by running "`your_directory/controlDLL/controlDLL.sln`".
4. Build up the project by clicking the menu "Build > Build Solution". Make sure "control.dll" is create on "`your_directory/Debug/`".
5. Run the Simulator with "`your_directory/Debug/CS22502.exe`". (If your computer is fast enough, the rendering rate can become faster by editing "`your_directory/bin/sim/Configuration.dat`" to set the value of "REFRESH_RATE_GRAPHIC" with smaller number.)
6. Play around with the simulator. Some of the settings won't be useful until you start writing controllers, but here are a few things you can try:
 - (a) Apply external forces to the PUMA, by dragging the manipulator's arm with the mouse. This feature is only enabled in simulation mode.
 - (b) To view the PUMA from other angles, click anywhere else in the image, and drag the mouse left or right. (You can also enable vertical rotation via the pop-up menu.)
 - (c) Go to the the Settings tab, and disable the simulated friction.
 - (d) Hover the mouse over the labels, especially on the Settings tab, to see tooltips.
 - (e) Go to the Control tab, and click the combo box labeled "float" in the upper-left corner. You will see a list of controllers your group will implement this quarter.

- (f) Select the “goto” controller (or almost any other controller in the bottom half of the list), and press the “Alternate Parameters...” button. You will get a pop-up dialog that allows you to enter operational-space coordinates (“configuration parameters”) in several different notations. Playing with this dialog can help clarify how different notations relate to each other — for example, you can change the Euler angles and see how this changes the rotation matrix. All angles are in degrees.
7. Try modifying the open loop controller `initOpenControl()` and `openControl()` in `control.cpp` by applying torques to the joints. The global variable for the torques is `gv.tau`.
For example, you can make a joint spin by assigning a large torque to one of `gv.tau[0]...gv.tau[5]` in `openControl()`. (Do not try this with real PUMA! The joints can’t turn that far. But you can try whatever you want in simulation.)
You can use conditions based on joint positions `gv.q` or joint velocities `gv.dq` to create more interesting motions.
8. To run your open loop controller:
- Compile and run the simulator.
 - Select the Control tab.
 - Change the combo box near the upper-left corner from **float** to **open**.
 - Press the Start button.
 - Note: You can make the simulation more interesting by going to the Settings tab, and disabling the torque limits and/or simulated friction.
9. To debug `control.cpp` in VS .NET 2003 environment, set break points in `control.cpp` and run the simulator by “Debug > Start” in the menu or pressing F5 button. Then the program will stop at every break point and values of variables at the moment can be seen in debugger windows. The debugger loads “your_directory/Configuration.dat” to start the simulation program.

Controlling a PUMA in the lab

Each group will be assigned two lab computers:

1. A computer called **perse** or **garuda**, which is connected to a PUMA, and uses a real-time operating system called “QNX”. This computer will run the servo loop, including the controllers you write.
2. The QNX machines aren’t fast enough to control a PUMA and display a GUI at the same time, so the GUI will run on a remote Windows client computer.

To control a PUMA:

1. Copy the “control.cpp” to your directory on the QNX machine (perse or garuda) from your Windows machine.

2. Copy one of the following environment variables into “your_directory/bin/sim/Configuration.dat” on your Windows computer:
 - If your server is perse: **CS225A_SERV_HOST_ADDR=172.24.68.108**
 - If your server is garuda: **CS225A_SERV_HOST_ADDR=172.24.68.162**
3. Type “make rt” on perse/garuda to compile the real-time server. (If you need to force a full rebuild, type “make clean-all” first.)
4. Turn on the PUMA controller and hold the red button.
5. Execute “./run” on perse/garuda.
6. Execute “your_directory/bin/RemoteClient/CS22502.exe” on your Windows computer.
7. Check if the joint positions and torques are correct.
8. Put the PUMA in Float mode.
9. Release the red button (this requires a clockwise twist on perse/garuda’s controller) and press the green button. You can now control the PUMA from the GUI on panthro/neruda.
10. Warning: you might not find all of your bugs at simulation time, so one group member should always watch the PUMA and keep a thumb poised over the red button. If the PUMA starts acting strangely, press the red button immediately.

Resetting the PUMA

Every so often, the PUMA sensors might report incorrect joint positions. To reset the PUMA:

1. Run the software, as described above.
2. Put the PUMA in Float mode.
3. Manually adjust the joints into the positions where the sensors should read 0° .
4. Type “reset” at the “CS225A:>” prompt. This changes the zero reference point to the current configuration.

Notes and Restrictions on Coding

- No dynamic memory allocation. You may not use malloc/free, new/delete. Please allocate space needed statically. Not following this rule may result in not completing the servo loop in time.
- The library has classes for matrices and vectors (`PrMatrix`, `PrVector`, etc). You can refer to the header files in the *include* directory.

A `PrVector` can be indexed like an array (`gv.tau[0]`), and a `PrMatrix` can be indexed like a two-dimensional array (`gv.Lambda[0][5]`) or a function (`gv.Lambda(0, 5)`). You can add,

subtract, and multiply them using the usual rules of linear algebra, so multiplying a matrix by a vector does exactly what you'd expect. (There's one exception: If you multiply two vectors together, the result is a vector produced by an element-wise multiplication. This is useful for multiplying by k_p or k_v vectors. Use the `dot()` method if you want a scalar.)

- `printf()` is too slow to run from a servo loop, so it is aliased to the `Ui::Display()` method provided in `UiAgent.h`. The output is delegated to a low-priority task.

However, if the program unexpectedly crashes, any enqueued `printf()` statements will be lost. If you need to force an immediate `printf()` for debugging purposes, use `fprintf(stdout,...)` instead. But use `fprintf()` sparingly: on a QNX machine, an `fprintf()` statement is likely to *cause* a crash because the servo loop can't finish in time.

- One way to debug crashes, which you might need to do while working on your projects at the end of the quarter, is to analyze a core dump. Run “`gdb program_name core_dump`”, and use the “`bt`” command to see a stack trace. On the QNX machines (perse/garuda), the core dump is in `/var/dumps/servo.core`. On the linux machines (pathro/neruda/firebird/PUP/raptor), you can enable core dumps by typing “`ulimit -c unlimited`”. Remember to clean up your core dumps on the computers, or you will run out of disk space!

Global Variables

All the variables and parameters of relevance for the student to access are declared in GlobalVariables.h; All generated code for the assignments and the final project will be located in control.cpp. The student will be able to declare global variables and functions in control.cpp as well.

Note: even though the GUI measures angles in degrees, the internal variables that you'll use in the code use radians. The angles are converted when the data is sent from the GUI to the server.

The gv. denotes a variable of a GlobalVariables instance.

State variables

gv.dof	: Degrees of freedom
gv.curTime	: Current simulator time
gv.tau	: Vector of joint torques [in newton-meters]
gv.q	: Vector of current joint space positions [rad]
gv.dq	: Vector of current joint space velocities [rad / sec]
gv.kp	: Vector of position gains (k_p)
gv.kv	: Vector of velocity gains (k_v)
gv.qd	: Vector of desired joint positions [rad]
gv.dqd	: Vector of desired joint velocities [rad / sec]
gv.ddqd	: Vector of desired joint accelerations [rad / sec ²]
gv.x	: Vector of current operational space positions
gv.dx	: Vector of current operational space velocities
gv.xd	: Vector of desired operational space positions
gv.dxd	: Vector of desired operational space velocities
gv.ddxd	: Vector of desired operational space accelerations
gv.elbow	: Desired elbow configuration for track control mode
gv.T	: Linear transformation for end-effector position/orientation
gv.Td	: Linear transformation for desired end-effector position/orientation

Kinematics & Dynamics Variables

gv.J	: Jacobian
gv.Jtranspose	: Jacobian transpose
gv.A	: Mass matrix. Also called "M" in some robotics classes.
gv.B	: Centrifugal/coriolis vector
gv.G	: Gravity vector
gv.Lambda	: Mass matrix in operational space
gv.mu	: Centrifugal/coriolis vector in operational space
gv.p	: Gravity vector in operational space
gv.singularities	: Bitmap of singularities
gv.E	: Matrix converting linear/angular velocity to configuration parameters
gv.Einverse	: Inverse of g_E matrix

Limit Variables

gv.qmin : Minimum joint positions allowance [rad]
gv.qmax : Maximum joint positions allowance [rad]
gv.dqmax : Maximum joint velocities allowance [rad / sec]
gv.ddqmax : Maximum joint accelerations allowance [rad / sec²]
gv.taumax : Maximum joint torques allowance [newton-meter]
gv.xmin : Vector of minimum operational-space coordinates
gv.xmax : Vector of maximum operational-space coordinates
gv.dxmax : Maximum operational space velocity allowance (scalar)
gv.ddxmax : Maximum operational space acceleration allowance (scalar)
gv.wmax : Maximum angular velocity in operational space (scalar)

Potential-field Variables

gv.jlimit : (simulator only) Joint Limits potential field flag
gv.q0 : Vector of minimum distances to apply joint limit potential fields
gv.kj : Vector of gains for the joint limit potential field
gv.rho0 : Minimum distance to apply potential field controller
gv.eta : Gain for potential field controller
gv.sbound : Boundary of singularity [rad]
gv.line : Structure holding a line for the Line Trajectory controller
gv.numObstacles : Number of obstacles, for the potential field controller
gv.obstacles : Array of obstacles, for the potential field controller

Functions to be modified in control.cpp

The following functions are run once every time a different control mode is invoked :

void InitControl()	: Runs before the first servo loop
void initFloatControl()	: Float Control Mode
void initOpenControl()	: Open-Loop Control Mode
void initNjholdControl()	: Joint Space Non-Dynamic Hold Mode
void initJholdControl()	: Joint Space Dynamic Hold Mode
void initNholdControl()	: Operational Space Non-Dynamic Hold Mode
void initHoldControl()	: Operational Space Dynamic Hold Mode
void initNjmoveControl()	: Joint Space Non-Dynamic Move Mode
void initJmoveControl()	: Joint Space Dynamic Move Mode
void initNjgotoControl()	: Joint Space Non-Dynamic Velocity Saturation Mode
void initJgotoControl()	: Joint Space Dynamic Velocity Saturation Mode
void initNjtrackControl()	: Joint Space Non-Dynamic Cubic Spline Track Mode
void initJtrackControl()	: Joint Space Dynamic Cubic Spline Track Mode
void initNxtrackControl()	: Cartesian Space Non-Dynamic Cubic Spline Track Mode
void initXtrackControl()	: Cartesian Space Dynamic Cubic Spline Track Mode
void initNgotoControl()	: Operational Space Non-Dynamic Velocity Saturation Mode
void initGotoControl()	: Operational Space Dynamic Velocity Saturation Mode
void initNtrackControl()	: Operational Space Non-Dynamic Cubic Spline Track Mode
void initTrackControl()	: Operational Space Dynamic Cubic Spline Track Mode
void initPfmovementControl()	: Potential Field Move Mode
void initLineControl()	: Potential Field Line Track Mode
void initProj1Control()	: User-defined Final Project Control Mode 1
void initProj2Control()	: User-defined Final Project Control Mode 2
void initPsroj3Control()	: User-defined Final Project Control Mode 3

The following control functions are be continuously executed (computing and sending torque values at the clock rate) as long as a associated control mode is active:

```
void PreprocessControl() : Runs before the control function
void PostprocessControl() : Runs after the control function
void noControl() : No Control, all Torque set to 0.0
void floatControl() : Arm Floats (gravity compensation)
void openControl() : Open-Loop (no feedback) control (caution !)
void njholdControl() : Arm holds current joint position (Non-Dynamic)
void jholdControl() : Arm holds current joint position (Dynamic)
void nholdControl() : Arm holds current end-effector position (Non-Dynamic)
void holdControl() : Arm holds current end-effector position (Dynamic)
void njmoveControl() : Joint Space, Non-Dynamic Feedback Control
void jmoveControl() : Joint Space, Dynamic Feedback Control
void njgotoControl() : Arm joints move to desired angles with vel.sat.(Non-Dynamic)
void jgotoControl() : Arm joints move to desired angles with vel.sat.(Dynamic)
void njtrackControl() : Arm joints move following cubic spline trajectory.(Non-Dynamic)
void jtrackControl() : Arm joints move following cubic spline trajectory.(Dynamic)
void nxtrackControl() : End-effector moves to cartesian coordinates (Non-Dynamic)
void xtrackControl() : End-effector moves to cartesian coordinates (Dynamic)
void ngotoControl() : End-effector follows trajectory with vel.sat. (Non-Dynamic)
void gotoControl() : End-effector follows trajectory with vel.sat. (Dynamic)
void ntrackControl() : End-effector follows a cubic spline trajectory (Non-Dynamic)
void trackControl() : End-effector follows a cubic spline trajectory (Dynamic)
void pfmoveControl() : Potential Field Method, manipulator avoids obstacles
void lineControl() : Line Tracking Method (with potential field)
void proj1Control() : User-defined Final Project Control Mode 1
void proj2Control() : User-defined Final Project Control Mode 2
void proj3Control() : User-defined Final Project Control Mode 3
```

The following function is useful for debugging. It will execute whenever you type **pdebug** at the “CS225A:>” prompt:

```
void PrintDebug() : Print debugging information in response to the pdebug command
```