The *SL* Simulation and Real-Time Control Software Package

Stefan Schaal

Computational Learning and Motor Control Laboratory
Computer Science and Neuroscience
University of Southern California, Los Angeles, 90089-2520

1 Introduction

SL was originally developed as a Simulation Laboratory software package to allow creating complex rigid-body dynamics simulations with minimal development times. It was meant to complement a real-time robotics setup such that robot programs could first be debugged in simulation before trying them on the actual robot. For this purpose, the motor control setup of ***SL*** was copied from our experience with real-time robot setups with vxWorks (Windriver Systems, Inc.)—indeed, more than 90% of the code is identical to the actual robot software, as will be explained later in detail. As a result, ***SL*** is divided into three software components: 1) the generic code that is shared by the actual robot and the simulation, 2) the robot specific code, and 3) the simulation specific code. The robot specific code is tailored to the robotic environments that we have experienced over the years, in particular towards VME-based multi-processor real-time operating systems. The simulation specific code has all the components for OpenGL graphics simulations and mimics the robot multi-processor environment in simple C-code. Importantly, ***SL*** can be used stand-alone for creating graphics animations—the heritage from real-time robotics does not restrict the complexity of possible simulations.

SL offers the following development features:

- Automatic generation of C-code for rigid body dynamics and kinematics based on simple text files that specify the kinematic structure of the motor system.
- Automatic generation of C-code for simple (!) OpenGL 3D graphics with the ability to add arbitrary beautification by the user afterwards.
- Inbuilt motor control loops for PD, PID, compute-torque, inverse dynamics control, floating base dynamics, contraint dynamics, and balance control.
- Inbuilt inverse kinematics computation and simple Cartesian control loops.
- Easy customization through a variety of preferences files.
- Easy data collection during control and Matlab-based visualization tools for trajectory data.
- Plain C-code implementation for maximal compute-speed and cross-platform compatibility.
- Very easy to extend for user-specific motor control problems.
- Support for generating floating-base simulations (e.g., walking simulations).
- Point-based collision checking.
- Easy definition of objects in the environment with simple geometric shape. Collision checking is automatically performed with respect to these of objects.
- Possible extensions to muscle-based simulations.

These properties of SL make it well suited for applications in the areas like:

- Robot control research and education.
- Almost arbitrary rigid body dynamics simulations, in particular humanoid or animal simulations.
- Studies of motor control form the viewpoint of computational neuroscience.
- Computer graphics with physical animations.
- Studies of motor learning.

- Behavioral motor control studies from the viewpoint of experimental psychology, kinesiology, and motor rehabilitation.

The following pages describe ***SL*** in increasing more detail, starting from the level of the end-user, down to the level of the programmer who wants to change the core of ***SL***.

2 Requirements

2.1 Software Requirements

- OpenGL and GLUT libraries (MesaOpenGL is fine).
- A C-compiler (e.g, the GNU C-compiler)
- Matlab (only for visualization of trajectory data—other visualization tools can be adopted, too.)
- Mathematica (only needed during setup of a new ***SL*** simulation for automatic C-code generation)

2.2 Hardware Requirements

- A fast floating point and graphics computer (e.g., we used iMacs, Apple PowerBooks, Apple G5 computers, Intel Linux computers, DEC Alphas, SUN Solaris, Silicon Graphics SGI)
- For robot control, a VME-bus with multiple computer boards and the real-time operating system vxWorks. It should be possible to port the robot code to other real-time operating systems without too much work.

2.3 Software Directory Setup

In the subsequent sections of this manual, we will assume that the ***SL*** software was installed in the directory tree below. This tree is not mandatory but recommended to make this manual compatible with the software installation. We will refer to items in this directory frequently.

The directories are:

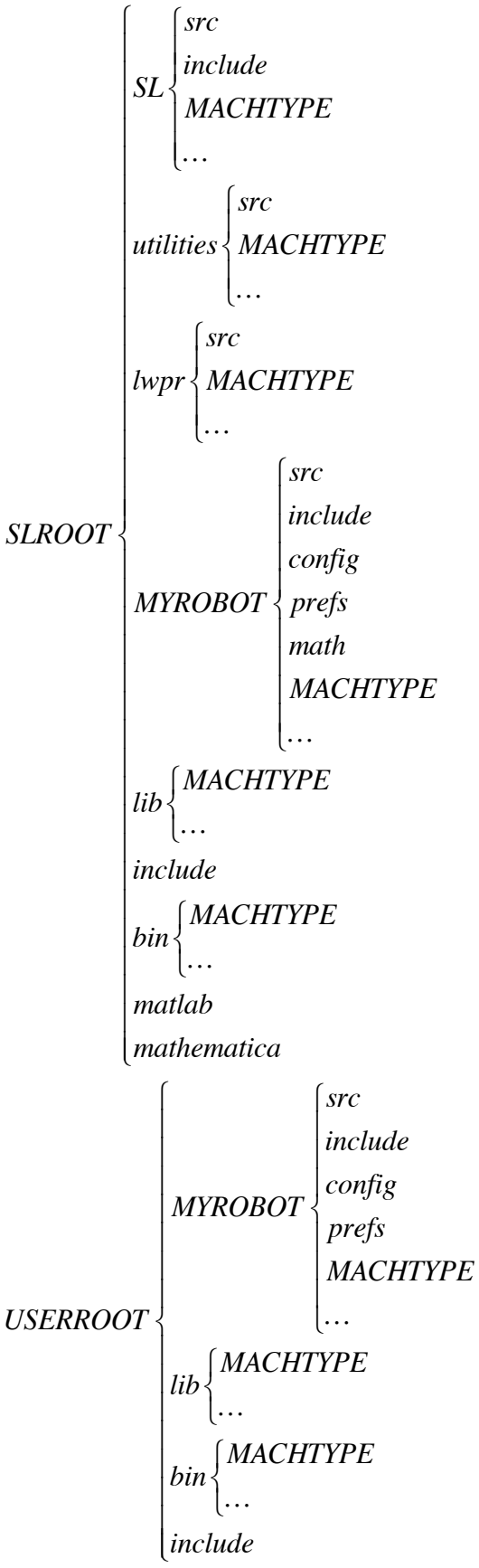

Figure 1: The ***SL*** directory tree.

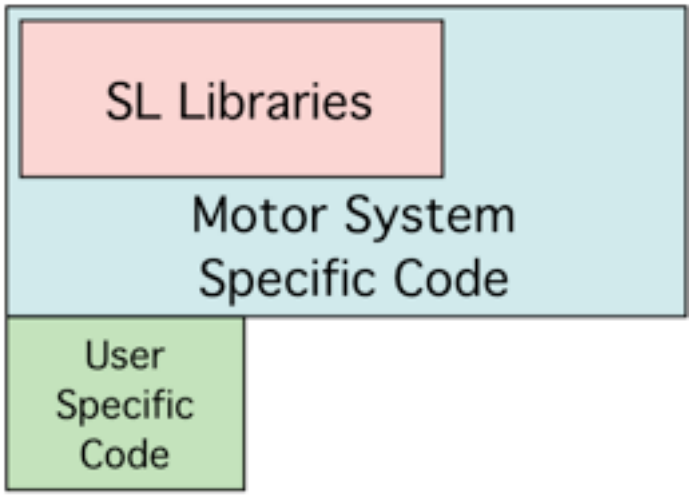

Figure 2: Coarse Software Architecture of an ***SL*** Animation or Robot Control System

SLROOT denotes the root installation of the ***SL*** and its miscellaneous software packages—all these directories exist only once in a computer network. USERROOT is an installation directory that would be repeated in every user's home directory. MYROBOT is the name of the robot/simulation that is to be used—there exist a global MYROBOT directory in SLROOT, and a local version for every user. MACHTYPE denotes an operating specific directory for binary files, compiled under a certain hardware platform (e.g., sparc (for SUN Solaris Sparc), i386 (for Linux on i386), macintosh (for Mac OS X), mips (for SGI mips), etc.). When using our imake-facilities and assuming the your operating system sets the required #define for MACHTYPE, all these directories will be maintained automatically, and compilation is possible on multiple hardware platforms using the same directories—in this case, multiple MACHTYPE directories will be created. Note the unix "tcsh" provides the $MACHTYPE environment variable automatically.

3 The Basic Structure of *SL*

SL has software at three different levels, illustrated in Figure 2:

- *The **SL** Libraries*: they contain the core of the ***SL*** programs and should only be changed by experienced and knowledgeable programmers.
- *The Motor System Specific Code*: this software specifies a particular simulation (or robot). It includes setup files that define the kinematic robot structure, the graphics display, control gains, digital filter coefficients, etc.
- *The User Specific Code*: for every simulation, ***SL*** creates a software library that can be augmented by the end-user with user specific movement tasks. For instance, a reaching task or a pole-balancing task can be created at this level without the need to look at the code of the other levels. This feature makes ***SL*** very friendly for less skilled users. For instance, we have been using ***SL*** for teaching robotics courses, and, within a one to two hours, students had no problems to grasp how to program ***SL***.

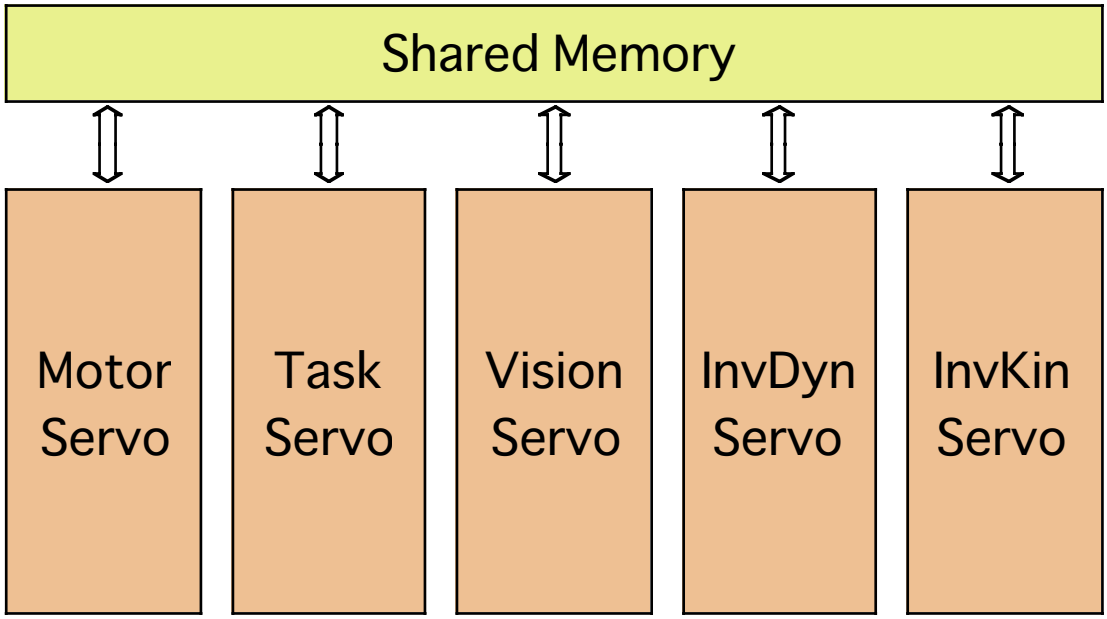

Figure 3: The ***SL*** code is structured to make use of multiple processors, or to mimic them in simulation.

4 The General Software Structure of *SL*

Before explaining how to program in the ***SL*** environment, it is necessary to provide an overview of the entire software structure. As illustrated in Figure 3, ***SL*** was programmed with a multiprocessor environment in mind, where inter-processor communication is accomplished through a shared memory structure. VxWorks realizes such multi-processor environments naturally in a VME bus, and for animations, ***SL*** just mimics this structure in software. Thus, ***SL*** has a modular software structure that helps keeping the programs more transparent.

The elements in Figure 3 have the following meaning:

- *Shared Memory*: Certain variable structures are defined as shared memory structures. In vxWorks, special memory allocation routines make sure that these structures reside in physically shared memory. In simulation, the memory is simply allocated by normal memory allocation functions. Each processor can read/write from/to shared memory. On all processors, shared memory variables are always kept as a local copy to avoid too much communication with the shared memory (this point is irrelevant for simulations, of course).
- *Motor Servo*: This processor or software module handles the low-level input/output with the robot or simulation, and implements the basic feedback control loops. The processor runs a servo at a user-specified frequency that does:
  - Reading of desired position, velocity, and feedforward commands, i.e., $\theta_d, \dot{\theta}_d, \mathbf{u}_{ff}$.

- Computing of PD, PID, PDFF, or PIDFF[1] motor commands $\mathbf{u}$ from the desired quantities $\theta_d, \dot{\theta}_d, \mathbf{u}_{ff}$, and the current state $\theta, \dot{\theta}$ of the robot. Gains from the negative feedback terms are specified in a parameter file, as will be explained later.
- Sending the motor commands to the robot or simulation.
- Putting the current state of the robot $\theta, \dot{\theta}, \ddot{\theta}, \tau, \mathbf{u}_{fb}, \mathbf{u}$, i.e. position, velocity, acceleration, force/torque sensor loads at each degree-of-freedom (DOF), feedback command, and total motor command, into shared memory. Numerical differentiation is employed if one of these signals is not directly available from the robot/simulation, and appropriate digital filters based on 2nd order Butterworth filters are applied (filtering is also enabled for simulations, but can be disabled with the *toggle_filters* UI command to explore how the filters change the quality of control; filter cutoff frequencies can be specified in a parameter file, which also allows to disable filtering altogether by specifying a cutoff of "100"). See also Figure 4 for how signals travel between the control modules.

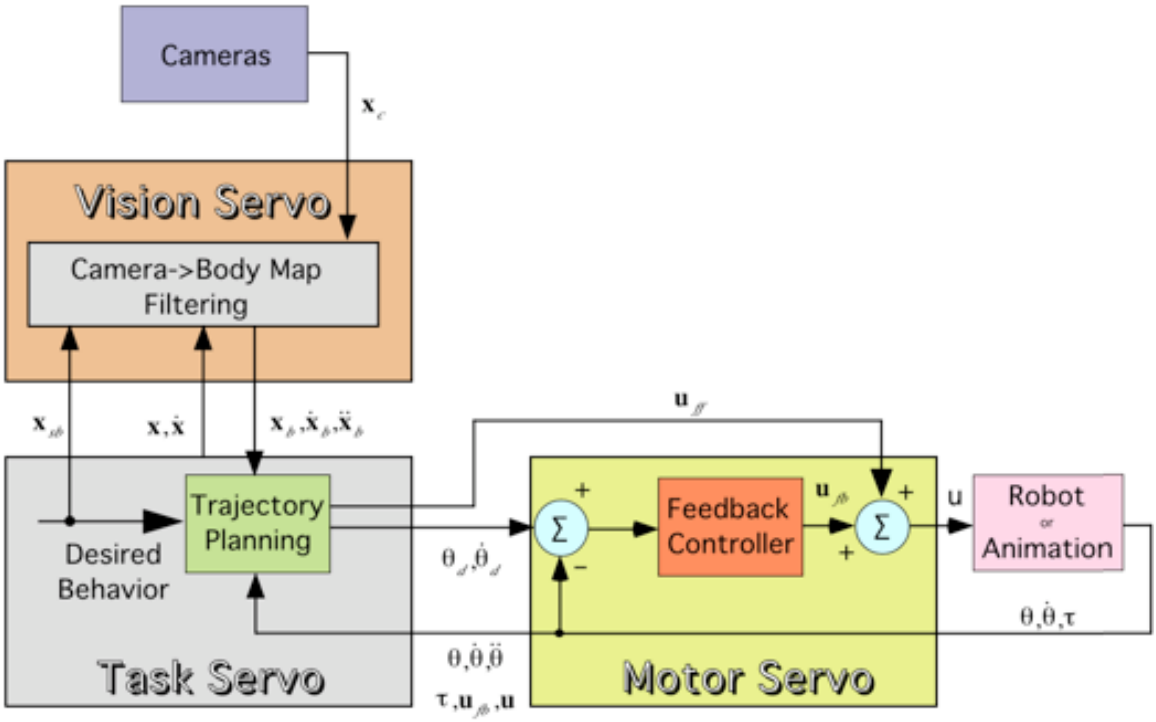

Figure 4: Control solely using the Task Servo. The Task Servo computes desired joint angular information and feedforward commands and passes those directly to the Motor Servo.

– *Task Servo*: This processor or software module allows switching between different motor tasks. The general goal of a motor task is to create appropriate desired position, velocity, and feedforward commands to accomplish a user specified motor behavior, and to send these quantities to the motor servo. From the viewpoint of control theory, the task servo implements the control law, although the PD or PID part of the control law is usually taken over by the motor servo. Note that negative feedback control can be taken over by the Task Servo, too, by sending the current state of

[1] P = proportional; D = derivative; I = integral; FF = feedforward

the robot as desired state to the motor servo, which effectively cancels the PD or PID contribution of the motor servo and only leaves the feedforward command to control the robot. On the other hand, the Task Servo can avoid certain typical computations of the control law by leaving them to additional modules, i.e., the InvDyn Servo and the InvKin Servo (Figure 5, Figure 6)—more information will be given below. In summary, the Task Servo runs a control loop, usually at the same frequency or a lower frequency as the Motor Servo, to accomplish:

- Reading of the current robot state $\boldsymbol{\theta}, \dot{\boldsymbol{\theta}}, \ddot{\boldsymbol{\theta}}, \boldsymbol{\tau}, \mathbf{u}_{fb}, \mathbf{u}$ from shared memory.
- Reading data about visual information from shared memory, i.e., information of the centroids of color blobs $\mathbf{x}_b, \dot{\mathbf{x}}_b, \ddot{\mathbf{x}}_b$ (more information below).
- Computing of useful quantities, e.g., the Jacobian $\mathbf{J}$ of the forward kinematics based on the current state, the Jacobian $\mathbf{J}_d$ of the forward kinematics based on the desired state, the endeffector position and velocity $\mathbf{x}, \dot{\mathbf{x}}$ and angular orientation and velocity $\mathbf{a}, \dot{\mathbf{a}}$, and the 3D positions of all joints.
- Running a user-specific set of functions to compute the task specific desired positions, velocities, and feedforward commands.
- Sending out the desired quantities above to shared memory, to be read by the Motor Servo or other modules.

– *Vision Servo*: The vision servo collects data from camera systems and makes the data available in shared memory. We assume that all visual information comes in form of the centroids of color blobs $\mathbf{x}_c$, based on our previous work with color blob tracking vision systems. Thus, the vision servo converts $\mathbf{x}_c$, i.e., two-dimensional image based

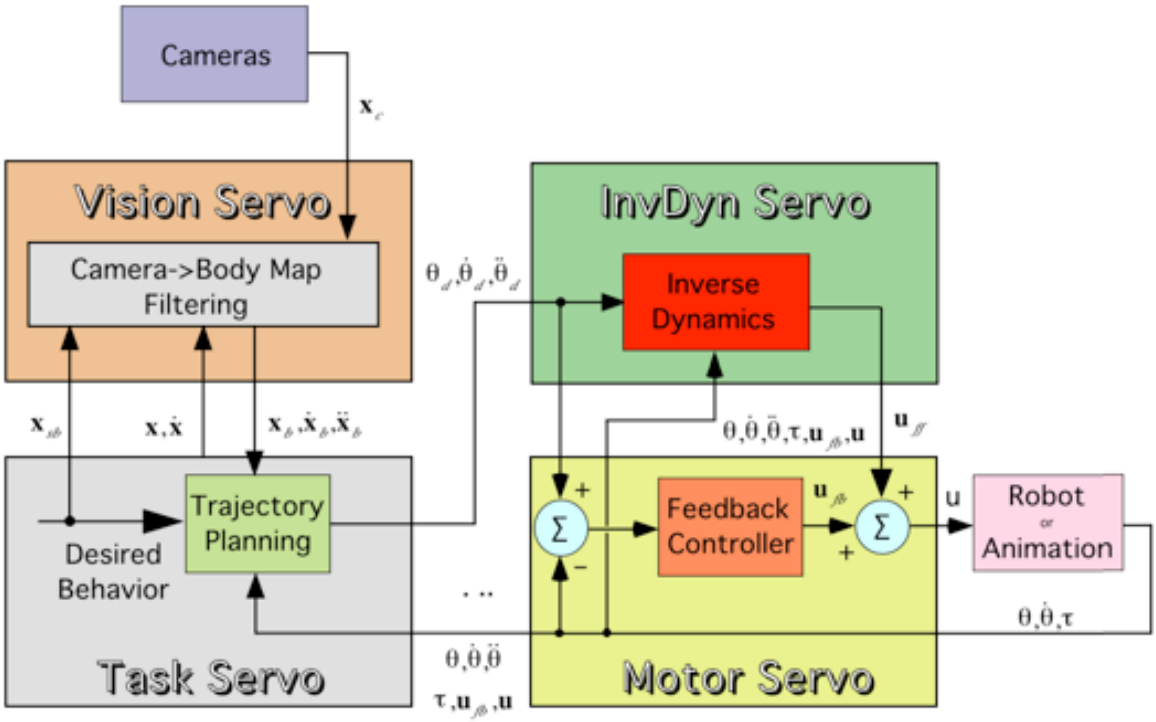

Figure 5: Control using the InvDyn Servo. In contrast to Figure 4, the Task Servo only computes the desired joint angular positions, velocities, and accelerations, passes those to the InvDyn Servo to fill in the feedforward commands.

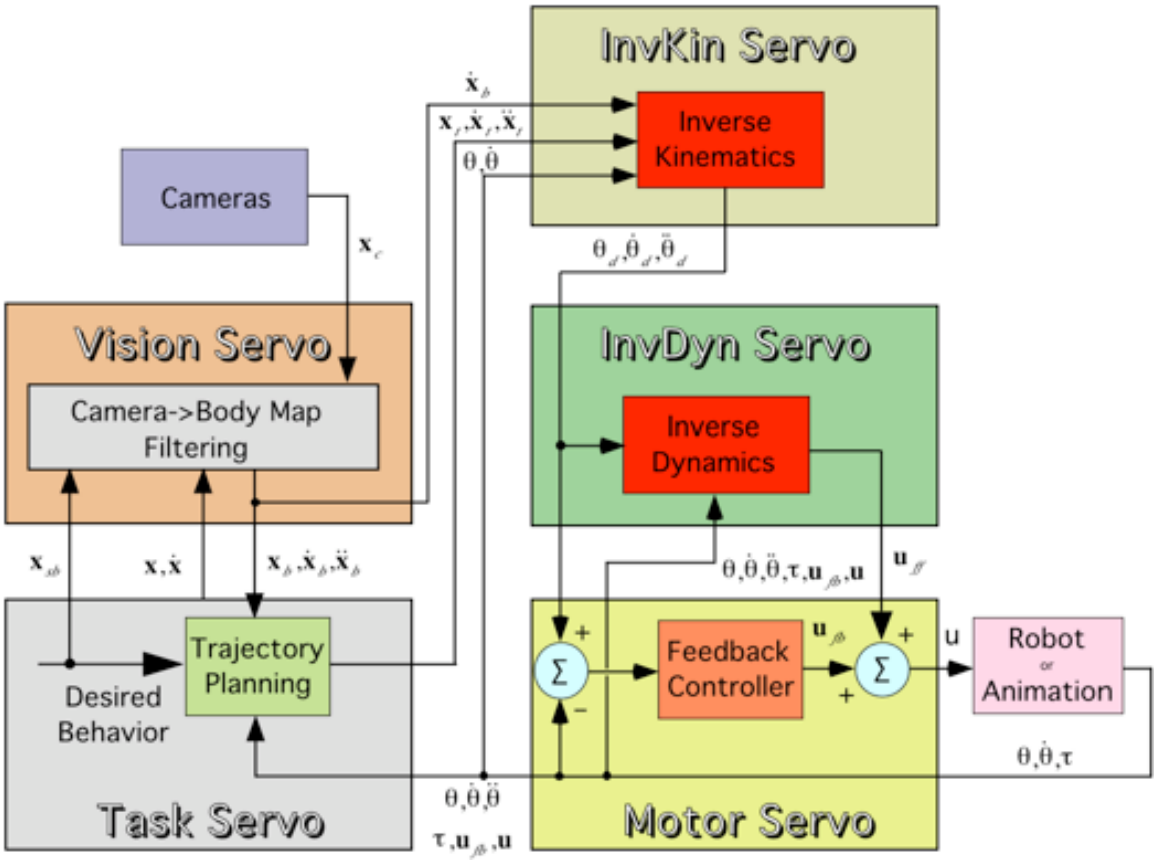

Figure 6: Control using the InvKin Servo. The Task Servo only creates information about a target in Cartesian space, and passes this information on to the InvKin Servo for conversion to desired joint space trajectories.

coordinates of the centroids, to 3D information, $\mathbf{x}_b, \dot{\mathbf{x}}_b, \ddot{\mathbf{x}}_b$ and provides these quantities at a rate of 60Hz (i.e., video rate) to the shared memory. In simulation, the centroid information can be generated by user specific functions and sent to the Vision Servo as simulated camera input $\mathbf{x}_{sb}$—the vision servo will copy $\mathbf{x}_{sb}$ to $\mathbf{x}_b$ and numerically differentiate these quantities to obtain derivatives. In summary, the Vision Servo does:

- Data collection from a (simulated) vision system.
- Data filtering with 2nd order Butterworth digital filters or Kalman filters.
- If necessary, transformations from camera space to robot-centric space.
- If desired, short time predictions to compensate for the delays in visual information processing.

Filtering is also performed on simulated data in order to generate realistic visual delays. The filter cutoff frequencies can be specified in task specific parameters files, as will be described later. These parameter files also allow eliminating filtering altogether.

– *InvDyn Servo*: The inverse dynamics servo is used to create inverse dynamics model-based feedforward commands on a separate processor. When using an analytical model of the inverse dynamics, the computations are fast enough in real-time to run

on the Task Servo. For learning of inverse dynamics models, a separate processor is needed. The inverse dynamics servo simply receives desired position, velocity, and acceleration commands and computes the appropriate feedforward commands either based on compute-torque techniques, or true inverse dynamics controllers. The desired position, velocity, and feedforward commands are then sent to the Motor Servo. Additionally, the Invdyn Servo can also learn the inverse dynamics model based on statistical neural network techniques. Figure 5 illustrates the control block diagram with the InvDyn Servo.

- *InvKin Servo*: In analogy with the inverse dynamics servo, the inverse kinematics servo is used to create a control policy that is based on Cartesian states. The Cartesian information about the target $\mathbf{x}_t, \dot{\mathbf{x}}_t, \ddot{\mathbf{x}}_t$ is sent to the inverse kinematics servo, and here it is converted to desired joint angle trajectories $\boldsymbol{\theta}_d, \dot{\boldsymbol{\theta}}_d, \ddot{\boldsymbol{\theta}}_d$. The InvKin Servo can also learn the inverse kinematic from visual data of the endeffector using statistical neural networks. The InvKin Servo passes the desire joint angular data to the InvDyn Servo for adding in the feedforward commands. This InvKin Servo is currently under development and does not exist yet in the ***SL*** libraries.

Given these different components, the basic control block diagrams of control can be visualized as shown in Figure 4, Figure 5, and Figure 6. The meaning of the variables used in these control diagrams was introduced in the text above. Note that arrows in and out of the servos are drawn to clarify which processor puts which variable into shared memory—unfortunately that made the routing of the arrows a bit less clear than possible. Whenever an arrow leaves a servo, a corresponding copy of the variables to shared memory occurs. Analogously, whenever an arrow enters a servo, the servo actually copies these variables from shared memory into a local data structure. All data structures will be discussed below in more detail. Switching between the different control modes is easily accomplished by calling a C-function, also described below.

5 The User Specific Code Level of *SL*

Given the explanations about the three different control modes in ***SL***, the task of a user specific program is to create appropriate input variables for either of the control modes. Just to repeat, these input variables are:

- *Task Servo Only Mode*: desired joint angle positions and velocities, and desired feedforward commands, i.e., $\boldsymbol{\theta}_d, \dot{\boldsymbol{\theta}}_d, \mathbf{u}_{ff}$.
- *InvDyn Servo Mode*: desired joint angle positions, velocities, and accelerations, i.e., $\boldsymbol{\theta}_d, \dot{\boldsymbol{\theta}}_d, \ddot{\boldsymbol{\theta}}_d$.
- *InvKin Servo Mode*: desired Cartesian target position, velocity, and acceleration, i.e., $\mathbf{x}_c, \dot{\mathbf{x}}_c, \ddot{\mathbf{x}}_c$.

Note that the Task Servo Only Mode can perform the same computations as the InvDyn Servo Mode and the InvKin Servo Mode. It is thus the most flexible, at the cost of more programming work in order to repeat the frequently occurring inverse kinematics and inverse dynamics computations.

The way ***SL*** allows to add a user specific motor task requires minimal knowledge of the entire ***SL*** software. Each such task involves writing three simple C-functions (see also Figure 7):

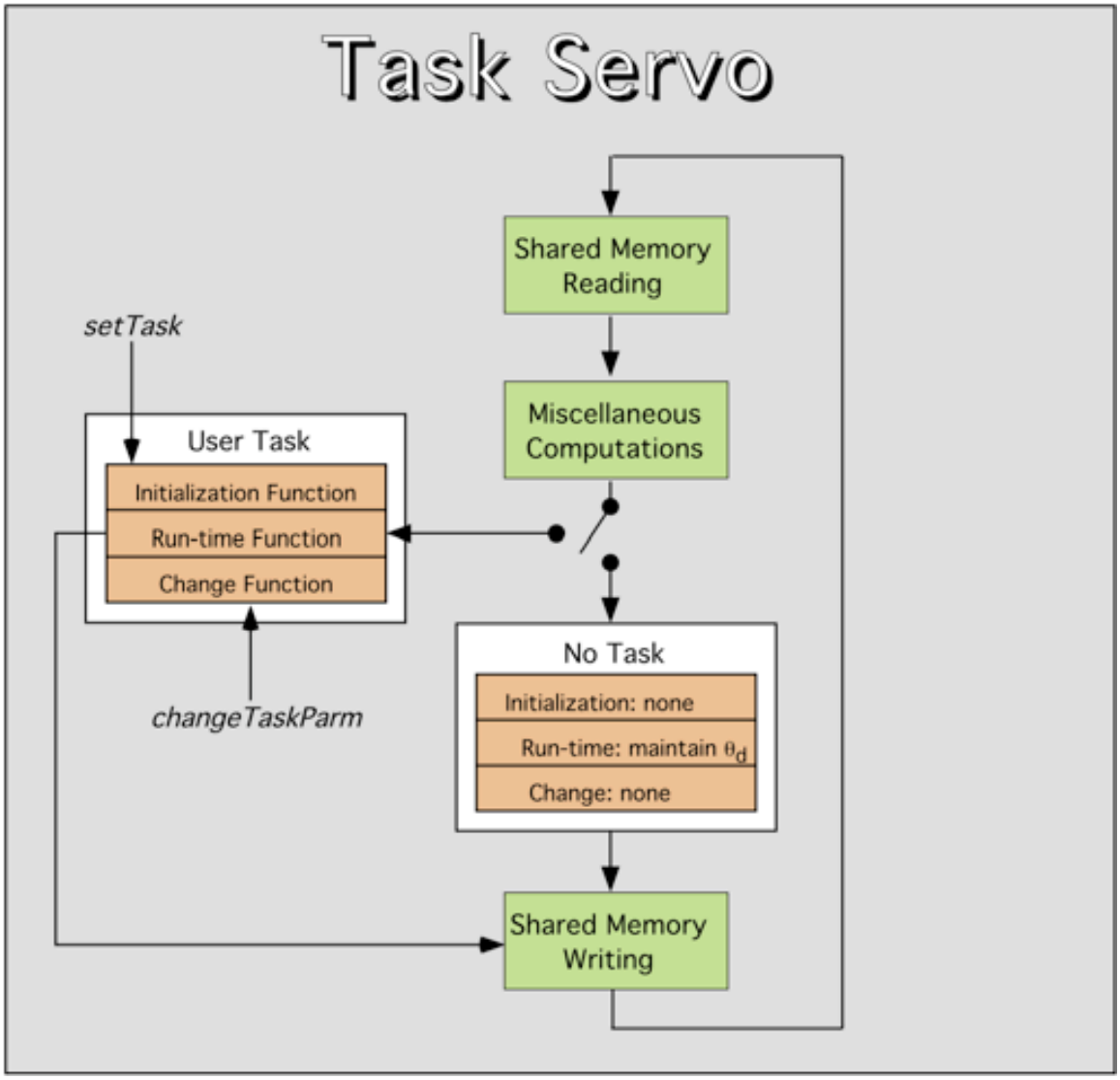

Figure 7: A more detailed view of the loop running on the Task Servo. A software switch allows users to activate their own task to create motor commands. By default, the robot (animation) runs the "No Task", i.e., a task which simply maintains the current posture.

1. *The initialization function*: This function initializes all variables needed to perform the motor task, and it commands the robot (or animation) into a defined posture from which the task is to be started. This function is called when the *setTask* command is typed in the user interface and the specific task is selected from the ensuing menu. No other task must be running when *setTask* is called, which is enforced by the software. Note that the initialization function is *not* time critical, i.e., all computations are executed in a parallel process independent of the ongoing robot control servo loops—for simulations, of course, the latter point is irrelevant.
2. *The run time function*: After successful initialization, the pointer to the run time function is added to the Task Servo loop and executed at the frequency of the

Task Servo loop (see switch in Figure 7). The run-time function *does* require time critical computations, i.e., on a real robot it must not take longer than the period of the Task Servo loop (e.g., about 2 milliseconds in most of our systems). For simulations, real-time issues do not matter as long as the task is never to be executed on a real robot.

3. *The change parameter function*: After successful initialization, the pointer to this function is linked to the UI command *changeTaskParm*. In a parallel process, task specific parameters (e.g., the speed of executing a movement) can be changed interactively through this facility. Care should be taken in the *change parameter function* to guarantee that a change of a parameter does not lead to a catastrophe on the real robot. A good habit is to use local temporary variables for user interactions and only copy those to the final variables after appropriate checking for the appropriate range of a variable.

Figure 7 illustrates the outline of the servo loop running on the Task Servo. As described in the next section, several tasks are already provided by default in ***SL*** and their source code in SLROOT/src can serve as a template how to write new tasks.

5.1 Inbuilt Tasks in every *SL* Motor System

By default, ***SL*** provides the following tasks from the *setTask* UI command:

- The GOTO_TASK: When evoked, the task controls the robot from the current posture to a target posture by using a constant velocity profile and by linearly interpolating the feedforward commands between the current and the target position. This task is the most basic way of moving the robot to a given target.
- The SINE_TASK: This task reads a user provided parameter file from `MYROBOT/prefs`, e.g., `default.sine`. Each line of this parameter file contains (in the sequence below):
  - The name of a DOF as defined in `SL_user_common.c`.
  - The number of superimposed sine waves.
  - The zero position of the sine wave A_0.
  - The amplitude A_1 of sine wave one.
  - The phase offset φ_1 of sine wave one.
  - The frequency f_1 of sine wave one.
  - The latter three pieces of information are repeated for every additional sine wave—up to 3 superimposed sine waves are possible.

  This results in a desired trajectory of DOF *i* as:

$$\theta_{d,i} = A_0 + \sum_{r=1}^{3} A_r \sin(2\pi f_r + \varphi_r).$$

  Desired velocity and accelerations are derived analytically and the task is executed with inverse dynamics control. The SINE_TASK is useful to run the robot/simulation for an extended time in order to test the quality of control of some DOFs (i.e., the quality of tracking), or to collect data for learning applications, etc.
- The TRAJ_TASK: This task tracks a desired trajectory that is provided in a data file, e.g., `reaching.traj`. The format of this file is described in Section 5.4.4. For every DOF that has a desired trajectory in this file, tracking is performed using either feedforward commands provided in this file, on-line feedforward commands from the inverse dynamics computation, or no feedforward commands. Non-specified DOFs

are kept at the default posture. The TRAJ_TASK automatically down-samples or up-samples the given desired trajectories in order to provide a target at every servo cycle. The TRAJ_TASK is useful to test arbitrary trajectories that were generated off-line. For instance, open-loop tasks like the Japanese manipulation task of Kendama (Miyamoto, Schaal, Gandolfo, Koike, Osu, Nakano, Wada, & Kawato, 1996 can be realized in this way, or the results of learning research can be tested even if it is too hard to implement the learning system on-line.

- The GOTO_CART_TASK: Using inverse kinematic control, this task controls the endeffector of the robot/animation in a straight line in Cartesian space to a Cartesian target position. A minimum jerk trajectory in Cartesian space realizes smooth, human-like motion. Inverse dynamics control is used to accomplish the movement.
- The TRACKING_TASK: Using inverse kinematic control, this task controls the endeffector of the robot/animation to track a simulated or real visual target. This task is currently still under development and not included in standard ***SL*** distributions.

In the next section, we will introduce some of the most important C-variables and functions of ***SL*** to allow the use to create her/his own motor tasks. We will also provide an example of how to generate a simple task in SL, and how to link it into the entire ***SL*** software.

5.2 Important C-Variables of *SL*

SL has a set of global data structures that provide access to all important information about the simulation or robot. Assuming that the core libraries of ***SL*** were installed in the director `SLROOT`, the directory `SLROOT/include` contains two important header files, `SL.h` and `SL_user_common.h` (see Appendix for a recent version of these headers files—but those may not be quite up-to-date anymore). `SL.h` declares all relevant variable types and structures for ***SL***, while `SL_user_common.h` contains all the global variables that can be accessed by the user. It should be noted that on a real robot with multiple processors, these variables are only assigned meaningful values if their contents were computed locally on a processor, or if it was read from shared-memory. In simulation, all these variables are globally shared, such that the user will always have access to appropriate values.

Two more files are important, `SL_user.h` and `SL_user_common.c`. The files are specific for a particular animation or robotic system, and should reside in a directory `MYROBOT/include` and `MYROBOT/src`, respectively. Two examples of these files for a 10 DOF robot arm are in the Appendix. It should be clear from reading these two files that they simply instantiate various parameters that define a robot or simulation, in particular a numbering scheme for joints and links, a corresponding naming of these elements, and various constants including the number of degrees-of-freedom, the number of endeffectors, etc.

`SL_user_common.h` contains the most relevant information for the following discussions since it declares all global variables. There are short comments in `SL_user_common.h` to explain each variable, such that we will not go into the details of all of them—the ***SL*** libraries are getting improved all the time, such that this documentation is never per-

fectly up-to-date. Nevertheless, some of the most important variables deserve a brief discussion:

```
SL_Jstate      joint_state[N_DOFS+1];          /* current states */
```

This array of structures contains the current joint states of the robot, i.e., $\theta, \dot{\theta}, \ddot{\theta}, \theta, \mathbf{u}_{fb}, \mathbf{u}$, as explained in Figure 4, Figure 5, and Figure 6. Obviously, the user is not supposed to overwrite the contents of these variables. Note that all our arrays in ***SL*** start with index "1", in contrary to most C programming style in computer science, but in accordance to matrix algebra, e.g., as in Matlab.

```
SL_DJstate     joint_des_state[N_DOFS+1];      /* desired states */
```

This array of structures contains the desired state in joint coordinates, i.e., $\theta_d, \dot{\theta}_d, \ddot{\theta}_d, \mathbf{u}_{ff}, \mathbf{u}_{ext}$. This array needs to be assigned appropriately by each user task. Note that $\mathbf{u}_{ext}$ allows the inverse dynamics to compensate for an external torque at each joint, e.g., as it could result from a Jacobian-transpose computation that takes an endeffector force and maps it into joint space.

```
SL_endeff      endeff[N_ENDEFFS+1];           /* endeffector structure */
```

Each endeffector has a specific structure that allows simulating objects attached to the endeffector in terms of their inertia, mass, and center of mass (all expressed in local coordinates of the endeffector coordinate system, but NOT necessarily in center-of-mass coordinates).

```
SL_Jstate      joint_sim_state[N_DOFS+1]; /* state of the sim. robot */
```

This array of structures contains the current state of a *simulated* system. The values from this array serve as sensory input to the simulation, while the simulation writes torques to this array which are subsequently used in numerical integration of the equations of motion.

```
SL_DJstate     joint_default_state[N_DOFS+1];  /* posture for startup */
```

This array of structures contains the default posture of robot/simulation—only position variables are used, e.g., `joint_default_state[n].th`. The robot/simulation starts up by commanding this state to the controller. The UI command *go0* automatically servos the robot/simulation to this posture. The array is initialized from the configuration file `SensorOffsets.cf`, which resides in the `MYROBOT/config` directory of each user (more about this in Section 0).

```
double         joint_range[N_DOFS+1][3+1];   /* info on joint limits */
```

This matrix contains information about kinematic quantities of each joint:

- `joint_range[n][MIN_THETA]`: the minimum of the joint range
- `joint_range[n][MAX_THETA]`: the maximum of the joint range
- `joint_range[n][THETA_OFFSET]`: an offset value, to be subtracted from the sensor readings. This value is only relevant for real robot where the sensory reading at the zero position does not necessarily coincide with the user definition of the zero posture.

All these value are initialized from the file `SensorOffsets.cf`.

```
double         u_max[N_DOFS+1];              /* actuator output limits */
```

This array contains the maximal absolute output of the actuators for each joint, assuming that this value applies both for positive and negative saturation. All values are initialized from `Gains.cf`

```
SL_VisionBlob blobs[MAX_BLOBS+1];          /* blob info from vision */
```
This array contains information about color vision blobs. For each blob, 3D information about position, velocity, and acceleration is provided, i.e., $\mathbf{x}_b, \dot{\mathbf{x}}_b, \ddot{\mathbf{x}}_b$ in Figure 4, Figure 5, and Figure 6. Additional, for each blob a status flag is given, e.g., `blobs[n].status`, that indicates whether the blob information is valid (e.g., if the blob was occluded, the status will be "0", if it is valid, the status will be "1").

```
Matrix       J;                            /* kinematic Jacobian */
```
This matrix contains the geometric Jacobian (e.g., cf. Sciavicco & Siciliano, 1996) of all endeffectors, computed based on the position values in `joint_state`.

```
Matrix       Jdes;                /* Jacobian based on des. state */
```
This matrix contains the geometric Jacobian (e.g., cf. Sciavicco & Siciliano, 1996) of all endeffectors, computed based on the desired position values in `joint_des_state`.

```
Matrix       link_pos;                      /* Cart. pos of links */
```
This matrix contains the Cartesian positions of all joints, endeffectors, and other extremities (e.g., the top of the head, the heel of the foot, etc.) of the robot/animation. The computations are based on `joint_state`. Note that the dimensionality of `link_pos` is [N_LINKS] by [3+1]. The number N_LINKS is not identical to the number of DOFs, i.e., N_DOFS, since one physical joint can have multiple DOFs, and some extremities do not add a DOF. The files SL_user.h and SL_user_common.c contain information about these elements. It will be explained later how these elements are created for a new simluation/robot.

```
Matrix       link_pos_des;             /* desired cart. pos of links */
```
This matrix contains the Cartesian positions of all joints, endeffectors, and other extremities (e.g., the top of the head, the heel of the foot, etc.) of the robot/animation. The computations are based on `joint_des_state`.

```
SL_Cstate    cart_state[N_ENDEFFS+1];     /* endeffector state */
SL_Corient   cart_orient[N_ENDEFFS+1];    /* endeffector orientation */
```
These structured arrays contain the Cartesian position and orientation of the endeffectors, their velocities, and their accelerations, i.e., $\mathbf{x}, \dot{\mathbf{x}}, \ddot{\mathbf{x}}, \mathbf{a}, \dot{\mathbf{a}}, \ddot{\mathbf{a}}$, based on `joint_state`. Note that accelerations are not computed as this is analytically too expensive and rarely used. The user could add these values, e.g., by numerical differentiation of the velocities and appropriate filtering. The position of the endeffectors is also available somewhere in the `link_pos` array. The file SL_user_common.c contains a variable `link2endeffmap` that tells ***SL*** which of the `link_pos` elements correspond to which endeffector. Endeffector velocities are computed from the geometric Jacobian, i.e., $\dot{\mathbf{x}} = \mathbf{J}(\theta)\dot{\theta}$.

```
SL_Cstate cart_des_state[N_ENDEFFS+1]; /* endeff.state from des.state */
SL_Corient cart_des_orient[N_ENDEFFS+1]; /* endeff.orient f. des.state*/
```

These structured arrays contain the Cartesian position and orientation of the endeffectors, their velocities, and their accelerations, i.e., $\mathbf{x}_d, \dot{\mathbf{x}}_d, \ddot{\mathbf{x}}_d, \mathbf{a}_d, \dot{\mathbf{a}}_d, \ddot{\mathbf{a}}_d$, based on `joint_des_state`. Note that accelerations are not computed as this is analytically too expensive and rarely used. The user could add these values, e.g., by numerical differentiation of the velocities and appropriate filtering. The position of the endeffectors is also available somewhere in the `link_pos_des` array. The file SL_user_common.c contains a

variable link2endeffmap that tell *SL* which of the link_pos_des elements correspond to which endeffector. Endeffector velocities are computed from the geometric Jacobian, i.e., $\dot{\mathbf{x}}_d = \mathbf{J}_{des}(\phi_d)\dot{\phi}_d$. Important: cart_des_state and cart_des_orient are not to be confused with cart_target_state and cart_target_orient explained below. The _des_ notation just indicates that the variables were calculated based on the desired joint states, but not that these variables serve as desired values in a Cartesian control loop. cart_target_state and cart_target_orient are the target states for this Cartesian control loop, used in inverse kinematics computations.

```
SL_Cstate cart_target_state[N_ENDEFFS+1]; /* endeff. Target state */
SL_Corient cart_target_orient[N_ENDEFFS+1]; /* endeff. Target orient */
```

These structured arrays contain the target Cartesian position and orientation of the endeffectors, their velocities, and their accelerations. These pieces of information are needed for using the InvKin Servo to perform motor control with inverse kinematics.

```
char          current_vision_pp[30];  /* vision post processing specs */
```

This string contains the name of a parameter file for vision post-processing. Every user specific motor task can specify in a simple ASCII file how the visual data should be filtered, etc. The Task Servo passes the name of the parameter file with the help of current_vision_pp to the Vision Servo, which then reads the parameter file and activates the appropriate post processing. These issues will be explained later in more detail.

```
SL_OJstate joint_opt_state[N_DOFS+1];  /* rest state for optimization */
```

This structured array specifies a rest posture for the robot/animation and weights for the importance of the rest posture for every DOF. This information is used in inverse kinematics computations to resolve the redundancy of the motor system. See http://www-slab.usc.edu/publications/ for papers on how we compute inverse kinematics. The variables in joint_opt_state are initialized from SensorOffsets.cf.

```
SL_link       links[N_DOFS+1];   /* specs of links: mass, inertia, cm */
```

This structured array specifies the dynamic variables for each link, i.e., the inertia matrix in local coordinates of the link (but usually not the center-of-mass coordinates), the mass of the link, and center of mass position in local coordinates of the link. Actually, we do NOT specify the center of mass position but rather the mass*cm, i.e., the center of mass position multiplied by the mass. This representation is numerically more efficient. The variables in links are initialized from LinkParameters.cf for real robots, and from LinkParametersSim.cf for animations. Normally, the same parameters can be used both for a robot and an equivalent simulation, but idiosyncrasies of real physical system sometimes make this impossible.

```
Blob3D  raw_blobs[MAX_BLOBS+1];/* raw blobs 3D of vision system */
Blob2D  raw_blobs2D[MAX_BLOBS+1][2+1];/*raw blobs 2D of vision system */
```

These variables contain the raw position information about color blobs from the vision system. Normally, stereo information comes from two cameras, both delivering 2D images—these values are stored in raw_blobs2D. The two camera images are then fused into 3D Cartesian information in world or robot-centric coordinates, e.g., by means of learning algorithms. The final result is stored in raw_blobs. Lastly, the Vision Servo converts raw_blobs into blobs, i.e., the filtered Cartesian information about the blobs, including velocities, accelerations, and possible post-processing.

```
int whichDOFs[N_DOFS+1] /* which DOFS does a servo compute */
```

On every servo, these variables are initialized from `WhichDOFs.cf`. This configuration file will be explained later. The idea of `whichDOFs` is that a robot system may have many CPUs, each of which computes only motor commands for a subset of the DOFs. `whichDOFs` specifies for which DOFs a CPU computers motor commands and allows thus sharing of labor among many processors. For simulations, normally all DOFs are computed on the Task Servo, the InvDyn Servo, and the InvKin Servo. The Vision Servo is not concerned with DOFs, and the Motor Servo must take care of every DOF anyway for the final control loop.

```
SL_Cstate    base_state;/* cartesian state of base coordinate system */
SL_quat      base_orient;/* cartesian orientation of base coord. Sys. */
```

These variables contain the position and orientation of the robot/animation base coordinate system as seen from the world coordinate system. Such information is needed in simulations with floating bases, e.g., walking animations.

```
SL_uext       uext[N_DOFS+1];     /* external forces on every DOF */
```

In order to simulate contact with the environment, contact forces and torques can be specified through `uext` for every DOF. Contact forces/torques are in world coordinates.

5.3 Important C-Functions of *SL*

C-functions can either be available for programming or for UI interactions. The UI *man* command displays all UI interface commands available in a simulation. Under vxWorks, every C-function that is not declared `static` is also available through UI command-line commands (although it needs to be explicitly added to the ***SL*** *man* UI command database). We will first describe generically available UI function (the user can easily create additional UI function), and second functions that are only available to the programmer.

5.3.1 User Interface (UI) Function

Including the header file `man.h` in a C-file allows adding a function's pointer to the *man* UI interface by using the function

```
addToMan(char *abr, char *string, void (*fptr)(void))
```

(see SLROOT/src/man.c for an explanation of the function call). Adding a function in this way has two results. First the name in `abr` will be listed together with the help text `string` when the *man* command is executed, and second the simulation interface will accept `abr` as a command on the command line by calling the corresponding function given by `fptr`. For a real robot running out of vxWorks, the latter point is irrelevant since every non-static C-function becomes a command line command anyway.

Several UI commands are implemented by default in ***SL***. The most important are:

```
    man -- prints all help messages
  reset -- reset state of simulation
setTask -- changes the current task of robot
     st -- short for setTask
   redo -- repeats the last task
changeTaskParm -- allows changing of parameters of the last(current) task
    ctp -- short for changeTaskParm
    go0 -- go to default posture
```

```
     go -- go to a specific posture
goVisTarget -- move one endeff to blob1
  where -- print all current state information
where_des -- print all desired joint information
 cwhere -- cartesian state of endeffectors
 lwhere -- cartesian state of links
  linfo -- axis,cog,orgin info of each link
 bwhere -- cartesian state of vision blobs
rbwhere -- current state of vision blobs
rbwhere2D -- current state of 2D vision blobs
where_base -- current state of base coordiante system
where_misc -- current state of miscellanious sensors
   setG -- set gravity constant
    scd -- start collect data
 stopcd -- manuall stop collect data
   scds -- start collect data, automatic saving
  mscds -- start collect data, automatic saving of multiple files
saveData -- save data from data collection to file
outMenu -- interactively change data collection settings
 status -- displays information about the servo
 freeze -- freeze the robot in current posture
   step -- step commands to a joint
   stop -- kills the robot control
     ck -- change controller
where_off -- sensor readings without offsets
where_raw -- raw sensor readings
toggle_filter -- toggles sensor filtering on and off
freezeBase -- freeze the base at orgin
```

Most of these commands are self-explanatory, particularly after trying them in the simulator. We will explain some of the more complex commands in later sections.

5.3.2 Programmable Functions

The following functions are useful in creating new tasks. In general, all these functions can be found in the various header files of SLROOT/include, or in the source files of SLROOT/src. Behind each function it is indicated whether it can be called from a time critical loop, i.e., with real-time performance.

```
int   check_range(SL_DJstate *des); (real-time ok)
```

Checks a desired state array for violations of the joint range and replaces values out of range with the permissible min/max values.

```
void SL_InverseDynamics(SL_Jstate *cstate,
             SL_Djstate *state,SL_endeff *endeff); (real-time ok)
```

Adds the inverse dynamics feedforward command to the desired state of the robot, i.e., `state`. This function assumes that `state` already contains the proper desired position, velocity, and acceleration for every DOF. The array of endeffector information is passed to include possible external loads at the endeffector to the inverse dynamics computations. If the current state `cstate` is passed as the NULL pointer, the inverse dynamics is solely computed based on the desired state, i.e., in a compute-torque control mode. If `cstate` is the current state of the robot/animation, the function computes a proper inverse dynamics control command, i.e., using the current position and velocities from `cstate` and

only the desires acceleration from `state`. See Sciavicco & Siciliano, 1996 for more information on inverse dynamics control.

`int go0_wait(void);`
This function can be called when the robot/animation is in NO_TASK mode. It starts the GOTO_TASK of ***SL***, which controls the system to acquire the default posture. This C-function waits until the GOTO_TASK is terminated before control is given back to the calling program. Note that this task is useful in the initialization function of a task (see the example in next section). The GOTO_TASK linearly interpolates in time between the current state of the robot and the desired state in order to go to the desired state with constant velocity. The speed of the movement is preprogrammed in the task.

`int go_target_wait(SL_DJstate *target);`
This function is equivalent to `go0_wait` but allows giving a target state for the GOTO_TASK controller. Only position and feedforward commands are taken into account from the `target` variable, i.e., linear interpolation in time is performed between the current and target position, and the current and target feedforward command.

`int go_target_wait_ID(SL_DJstate *target);`
This function is equivalent to `go0_target_wait` but uses inverse dynamics computations to provide a good feedforward command during the GOTO_TASK.

`void freeze(void); (real-time ok)`
This function freezes the robot/animation at the current position and terminates the ongoing task by enabling the NO_TASK. For this purpose, desired velocity and acceleration specifications are zeroed, while the current feedforward command is maintained. `freeze` is typically used when an error occurs in a task and the task is prematurely aborted, or it can be typed from the UI to manually stop the execution of a task.

`int send_raw_blobs(void); (real-time ok)`
This function is used when simulating visual information about color blobs (cf. Figure 4, Figure 5. Figure 6). Such a simulation creates information in the variable `raw_blobs` and, whenever this variable is updated, calls `send_raw_blobs` in order to transfer this information to the Vision Servo which then overwrites the visual information from camera input.

`int go_cart_target_wait(SL_Cstate *ctar,int *stat, double mt);`
In analogy to `go_target_wait`, this function servos the robot/animation to a Cartesian target using the GOTO_CART_TASK. See the section on inverse kinematics for a more detailed description of what the arguments of this function mean.

`int stop(char *); (real-time ok)`
This function is only really relevant for real robots. It implements an emergency stop of the robot with power shutdown.

`int setServoMode(int type); (real-time ok)`
This function allows switching between the three different control modes of ***SL***, i.e. MOTORSERVO (Figure 4), INVDYNSERVO (Figure 5), or CARTSERVO (Figure 6) mode. Switching can only take place when the robot is in the NO_TASK, i.e., it is typically part of the initialization function of a task.

`int init_pp(char *name);`

This function passes the variable `name` to the Vision Servo, which then reads the parameter file `name` from MYROBOT/prefs in order to initialize a new vision post-processing procedure. See Section 0 for more information.

```
void addVarToCollect(char *ptr,char *name,char *units,
                     int type, int flag);
```

SL has a very flexible way to allow the user to collect variables of interest during motor control. This data collection facility is described in detail in Section 5.4.4. In essence, `addVarToCollect` allows adding a pointer of a variable of interest to a data collection database such that this variable can be store in a data buffer during real-time motor control. The buffer can be written to a file and inspected by Matlab routines.

```
void scd(void); (real-time ok)
```

Data collection is triggered by this function. Data collection automatically terminates after an interactively specified time period. Often it is useful to make `scd` the last command of the initialization function of a task in order to trigger data collection with the moment that the run-time function of the task gets started.

```
void addTask(char *tname, int (*init_function)(void),
             int (*run_function)(void), int (*change_function)(void));
```

This function is called in order to add a task to the executable set of tasks of a ***SL*** simulation or robot controller. In simulations, `addTask` is used in the C-function `initUserTasks`() to link a user task to the entire ***SL*** code. In vxWorks, `addTask` can just be called from the script file that loads the compiled object containing the task (vxWorks allows real-time linking, while normal C-implementations do not). The example in the next section will make the usage of `addTask` rather clear.

```
int  inverseKinematics(SL_DJstate *state, SL_endeff *endeff,
       SL_OJstate *rest, Vector cart, iVector status, double dt);
        (real-time ok)
```

This function implements the inverse kinematics transformation of a Cartesian velocity vector to a joint space velocity vector. The function will be described in more detail in a later section.

```
void addUserDisplayFunc(void (*fptr)(void));
```

This function allows adding a pointer to a function to ***SL***, such that this function is called whenever a redraw of the graphics is required. Obviously, this function is only useful for simulations. It allows to add task-specific graphics to an ***SL*** simulation, e.g., a manipulated object like a ball or a pole for the task of pole balancing.

5.4 An Example of a User-Specific Task

The following example creates a small sinusoidal movement of all the DOFs around the midrange of each DOF. Additionally, the tip of the endeffector will be displayed as a yellow sphere, created by an OpenGL function, such that it will become clear how to add additional, task-specific graphics to the simulator. Note that we added extensive comments in the C-code to make it self-explanatory. We also used various functions from our C-utilities library `libutility.a` (the header file for `utility.h`, is in the appendix, while the corresponding C-function can be found in the directory UTILITIES-ROOT/src).

5.4.1 General Declarations at the Head of the C-file

The following piece of code is an example of what is usually the prologue of the C-file containing all the functions below.

```
/*=========================================================================
=========================================================================

                              example_task.c

=========================================================================
Remarks:

      a simple task to generate sinusoidal movements

=========================================================================*/

/* header files */
#include "stdio.h"
#include "math.h"
#include "string.h"
#ifndef VX
#include "strings.h"
#endif
#include "math.h"
#include "vx_headers.h"

/* private includes */
#include "SL.h"
#include "SL_user.h"
#include "tasks.h"
#include "task_servo.h"
#include "kinematics.h"
#include "dynamics.h"
#include "collect_data.h"
#include "shared_memory.h"

/* a fix for DEC alpha computers */
#ifdef alpha
#undef alpha
#endif

/* defines */

/* local variables */
static double frequency;
static double amp;
static SL_DJstate target[N_DOFS+1];
static double start_time;

/* global functions */
void add_example_task(void);

/* local functions */
static int  init_example_task(void);
static int  run_example_task(void);
static int  change_example_task(void);
static void display_ball(void);
```

5.4.2 *The Initialization Function*

```
/*************************************************************************
**************************************************************************
  Function Name        : init_example_task
  Date         : Dec. 1997

  Remarks:

  initialization for example task

**************************************************************************
  Paramters:  (i/o = input/output)

       none

 *************************************************************************/
static int
init_example_task(void)
{
  int    j, i;
  char   string[100];
  int    ans;
  int    flag = FALSE;
  static int firsttime = TRUE;


  /* a handy way of doing one-time initializations */
  if (firsttime) {

/* reset the frequency flag */
       firsttime = FALSE;

/* initialize the frequency */
frequency = 1.0;

/* initialize the amplitude */
amp = 0.1;

  }


  /* load vision post processing */
  strcpy(current_vision_pp,"tracking_task.pp");


  /* if this is not vxWorks, a ball is displayed at the endeffector */
#ifndef VX
  display_ball();
#endif

  /* go to the mid range posture */

  /* first, just make sure that the variable “target” is zeroed */
  bzero((char *)&(target[1]),N_DOFS*sizeof(target[1]));

  /* create the midrange posture */
  for (i=1; i<=N_DOFS; ++i) {
    target[i]=(joint_range[i][MAX_THETA]-joint_range[i][MIN_THETA])/2.0;
  }
```

```
  /* use the goto task with inverse dynamics computation to servo the
     system to the desired posture */
  if (!go_target_wait_ID(target))
    return FALSE;

  /* ready to go */
  ans = 999;
  while (ans == 999) {
    if (!get_int("Enter 1 to start or any key to abort ...",ans,&ans))
      return FALSE;
  }

  if (ans != 1)
    return FALSE;

  start_time = task_servo_time;

  /* collect data during the first seconds of the task */
  scd();

  return TRUE;

}
```

5.4.3 The Run-Time Function

```
/*************************************************************************
**************************************************************************
  Function Name        : run_example_task
  Date         : Dec. 1997

  Remarks:

  run the task from the task servo: REAL TIME requirements!

**************************************************************************
  Paramters:  (i/o = input/output)

  none

 *************************************************************************/
static int
run_example_task(void)
{
  int j, i;
  double task_time;
  double omega;

  /* some handy variables precomputed */
  task_time = task_servo_time - start_time;
  omega     = 2.0*PI*freq;

  /* compute the sinusoidal desired trajectories, using the variable
     task_servo_time, a generic variable of SL */

  for (i=1; i<=N_DOFS; ++i) {
    joint_des_state[i].th = target[i].th + amp * sin(omega*task_time);
    joint_des_state[i].thd = amp * omega * cos(omega*task_time);
    joint_des_state[i].thdd = -amp * omega * sin(omega*task_time);
```

```
  }

  /* compute the inverse dynamics feedforward command */

  SL_InverseDynamics(joint_state,joint_des_state,endeff);

  /* This is all that is needed. The variable joint_des_state will be
     sent to the Motor Servo automatically */


  return TRUE;

}
```

5.4.4 The Change Parameter Function

```
/*****************************************************************************
******************************************************************************
  Function Name        : change_example_task
  Date         : Dec. 1997

  Remarks:

  changes the task parameters

******************************************************************************
  Paramters:  (i/o = input/output)

  none

 *****************************************************************************/
static int
change_example_task(void)
{
  double aux;

  /* note that, at least for real robots, it is utmost important to
     catch any errors of the user */

  get_double("Frequency of Sinusoid",freq,&aux);
  if (aux >= 0.0 && aux < 5)
    freq = aux;

  get_double("Amplitude of Sinusoid",amp,&aux);
  if (aux >= 0.0 && aux < 0.4)
    amp = aux;

  return TRUE;

}
```

5.4.5 Adding a Task Specific Graphics Function

```
#ifndef VX
/*****************************************************************************
******************************************************************************
 Function Name : display_ball
 Date          : Aug. 99

 Remarks:
```

```
        displays a ball at the tip of the first endeffector

 ************************************************************************
 Paramters:  (i/o = input/output)

     none

 ************************************************************************/
#include "GL/glut.h"
static void
display_ball(void)
{
  static int firsttime = TRUE;
  GLfloat  col[4]={(float)1.0,(float)1.0,(float)0.0,(float)1.0};

  /* the following automatically adds the display function to the
     SL graphics routines */

  if (firsttime) {
    extern void addUserDisplayFunc(void (*fptr)(void));
    firsttime = FALSE;
    addUserDisplayFunc(display_ball);
  }

  /* here is the drawing rountines */
  glPushMatrix();
  glTranslated((GLdouble) cart_state[1].x[_X_],
               (GLdouble) cart_state[1].x[_Y_],
               (GLdouble) cart_state[1].x[_Z_]);

  glColor4fv(col);
  glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, col);
  glutSolidSphere(0.03,8,8);
  glPopMatrix();

}

#endif
```

5.4.6 Linking a Task to a Simulation/Robot

First, we create a brief function that adds the task to the *setTask* database of the Task Servo:

```
/************************************************************************
************************************************************************
Function Name  : add_example_task
Date           : Feb 1999
Remarks:

adds the task to the task menu

************************************************************************
Paramters:  (i/o = input/output)

none

************************************************************************/
void
```

```
add_example_task( void )

{
  addTask("Example Task", init_example_task,
          run_example_task, change_example_task);
}
```

5.4.6.1 Adding the Task in a Simulation

In simulations, every user has the file `initUserTasks.c` in `USERROOT/MYROBOT/src`. All that happens in this file, is that a function `initUserTask()` is created that is automatically called by ***SL*** during initialization. Every new task is added here as exemplified below.

```
/*=========================================================================
=========================================================================

                              initUserTasks.c

=========================================================================
Remarks:

         Functions needed to initialize and link user tasks for the
         simulation

=========================================================================*/

#include "SL.h"
#include "SL_user.h"
#include "string.h"

/* global variables */

/* local variables */


/*************************************************************************
*************************************************************************
Function Name  : initUserTasks
Date           : June 1999

Remarks:

      initialize tasks that are not permanently linked in the simulation
      This replaces the <ltasks facility in vxworks since we cannot
      do on-line linking in C.

*************************************************************************
Paramters:  (i/o = input/output)

  none

*************************************************************************/
void
initUserTasks(void)

{
  extern void add_example_task();

  /* this is all what is required to add the task */
  add_example_task();
```

```
}
```

5.4.6.2 Adding the Task in vxWorks

In vxWorks, the compiled task can be loaded into the current CPU with the load command, which will automatically link the code with the code already running on the CPU. On the command line, the user can just execute `add_example_task` interactively in order to make the new task visible in the *setTask* menu. Alternatively, all these command can be embedded in a script file, e.g., called `ltasks` in most of our implementation. The script file is triggered from the vxWorks command line by executing `<ltasks`.

5.5 Collecting Data During a Task and Displaying It in Matlab

SL has a simple, fast, and efficient way of collecting from arbitrary variables of interest during control. The basic idea of this data collection facility is straightforward: in real-time applications, it is usually not possible to write data to files on disk during control due to slow disk access time relative to the real-time requirements. Thus we opted to store data in a big matrix in the computer memory and dump this matrix to a file after data collection. Writing to memory can be accomplished easily in real-time, and dumping of the matrix can be done as a parallel process. Despite none of these issues are important for simulations, the ability to off-line analyze data files is usually much more versatile than on-line data display without storing of data.

5.5.1 Data Collection Functions

The C-programs `src/collect_data.c` and `include/collect_data.h` in the `SLROOT/SL` directory provide several C-functions to customize the data collection facility. The key C-functions are:

```
void initCollectData( int freq );
```

The function initializes the data collection routines and is automatically called by ***SL***. The argument `freq` inform the program at which maximal frequency data can be collected.

```
void addVarToCollect(char *ptr,char *name,char *units,
                     int type, int flag);
```

This function allows adding the pointer of a variable to the data collection facility. Note the variable must be in STATIC memory, i.e., not a local variable. Along with the variable, a name is passed that uniquely identifies the variable, the units of the quantity of the variables are specified in a character string, and the type (`DOUBLE`, `FLOAT`, `INT`, `SHORT`, `LONG`) needs to be given. The argument `flag` indicates whether ***SL*** should update the file `MYROBOT/prefs/sample_script` after adding the new variable. This file contains the names of all variables that are currently available for data collection and it can serve as a template to create data collection scripts (see below). For simulations `flag` can always be TRUE. For real-time implementations, disk access should usually be minimized such that it wiser to dump all the variables names explicitly after `addVarToCollect` was called for all desired variables by using the function `updateDataCollectScript`.

```
void updateDataCollectScript( void );
```

Writes all currently available names of variables for data collection to the file `MYROBOT/prefs/sample_script`.

`void writeToBuffer(void);`
SL calls this function internally. At every call, all active variables for data collection are stored in the row of a data collection matrix.

`void saveData(void);`
Save the previously collected data matrix to a file name dxxxxx, where xxxxx is an integer number automatically generated by ***SL***.

`void scd(void);`
Starts the data collection. This command is also available as a user interface command.

`void scds(void);`
Starts the data collections and automatically calls `saveData` after the data buffer is full. This command is also available as a user interface command.

`void mscds(int num);`
Collects multiple data files one after each other and saves the resulting data matrices automatically to disk. This command is also available as a user interface command.

`void stopcd(void);`
Terminates on-going data collection. Normally, data collection terminates automatically after the data buffer is full.

`int  dscd(int parm);`
Starts data collection with an initial delay. This command is most useful on the real robot when other processes need to be initiated before data collection starts (e.g., the robot needs to be given an object).

`void outMenu(void);`
This user interface command starts a three point menu that allows changing specifics of data collection. First, the frequency of data collection can be adjusted. Second, a new data collection script file (see below) can be activated. Third, the duration of data collection can be adjusted.

`void changeCollectFreq( int freq );`
This function allows changing the maximal frequency at which data collection can be performed. The frequency information is primarily needed to provide the user with information about the data collection in the data collection files.

5.5.2 Data Collection Script Files

Data collection script files are simple ASCII files containing the names of the variables that should be collected. These scripts should reside in the MYROBOT/prefs directory. Arbitrarily many script files can be generated. A script file contains a blank delimited list of variable names, using the names that were provided in `addVarToCollect`. The file MYROBOT/prefs/sample_script contains the variables that are currently available for data collection. Note, however, that if the user add additional variables without updating the data collection script (see above), sample_script may not be complete. An example of a data collection script could look like:

```
R_EB_th
R_EB_thd
```

```
R_EB_u
R_EB_load
R_EB_des_th
R_EB_des_thd
R_EB_uff
```
This script collects position, velocity, command, load, desired position, desired velocity, and feedforward commands of the right elbow of an anthropomorphic robot. SL automatically provided all these variables, i.e., they need not to be added by the user manually.

5.5.3 The Format of Data Collection Files

The contents of data collection files consists of an ASCII header followed by the data as a binary (IEEE big endian) FLOAT matrix. The ASCII header is composed of blank delimited items of the form:

1. The size of the data matrix (#rows times #columns) (INT).
2. The number of columns of the data matrix (i.e., the number of saved variables) (INT).
3. The number of rows of the data matrix (i.e., the length of the time series) (INT).
4. The sampling frequency of the data (FLOAT).
5. For every saved variable, the name of the variable (CHAR array) followed by the units of the variable (CHAR array). Note neither name nor unit must have blanks—use underscore to create delimiters.

These ASCII quantities can be inspected with any text editor, well, since they are ASCII. After point 5, the data matrix (FLOAT) is stored in binary format enforcing IEEE big endian binary format. The data matrix is written row-wise, i.e., the first row, the second row, etc. This latter point is important when reading the data in other programs. Inspect `SLROOT/SL/src/collect_data.c` for how the data is saved to file. As an example of how the data files can be decoded by another program, look at the Matlab files explained in the next section.

5.5.4 Displaying Data Collection Files in Matlab with MRDPLOT

Given a Matlab 5.0 installation(or higher) on your computer, you can use our MRDPLOT functions to display data files collected from the robot/simulation. You need to start Matlab and make sure that `SLROOT/matlab` is in your access path in Matlab. At the Matlab command window prompt, type `mrdplot`. The two Matlab windows illustrated in Figure 8 will be displayed.

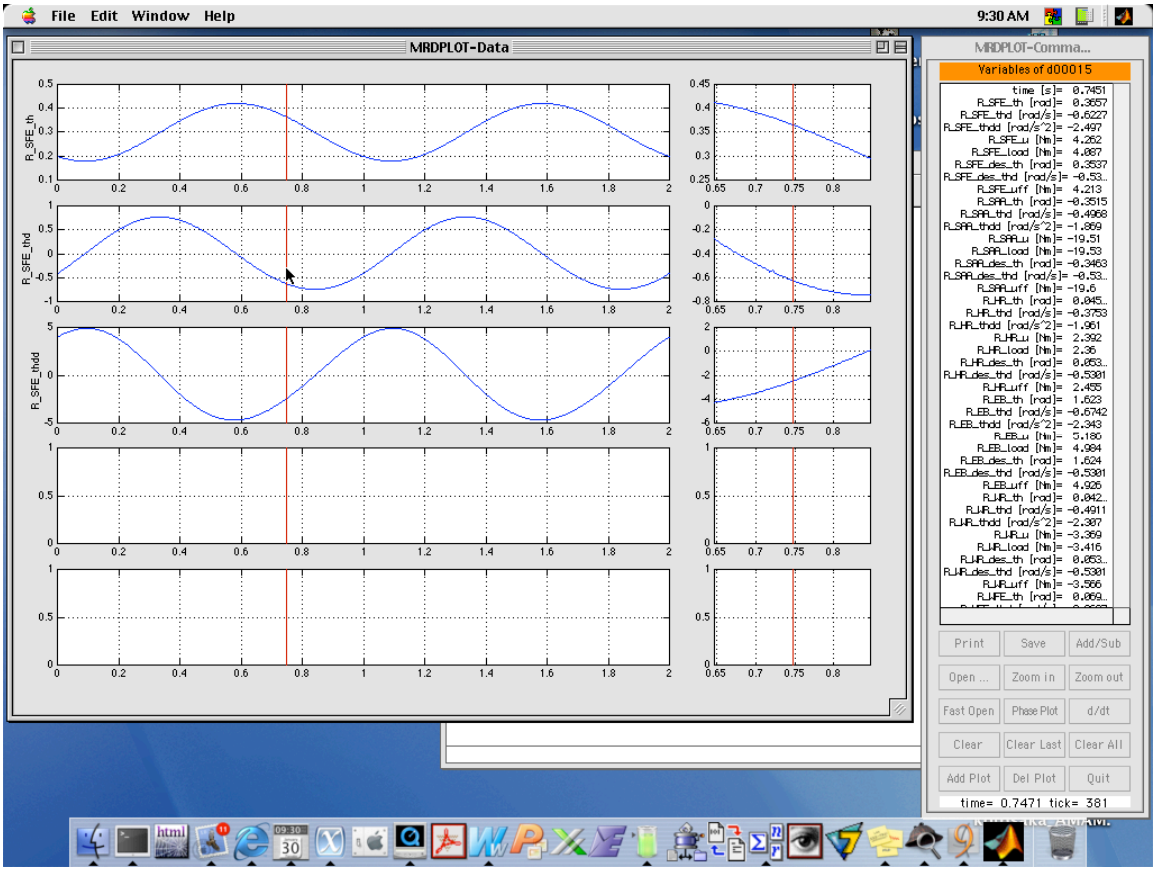

Figure 8: Screenshot of MRDPLOT windows in Matlab in Mac OS X environment.

The MRDPLOT-Data window is to display the time series of different variables stored in a data file. There is a zoom window on the right side of the MRDPLOT-Data window that displays the area around the current position of cursor (red bar on the left) in a magnified way. The cursor can be shifted by mouse-clicking in any of the graphic windows: this will shift the cursor exactly to this position and refresh the zoom windows.

The MRDPLOT-Command window contains the variables available for display as well as several commands how to manipulate the display. A variable is added to the MRDPLOT-Data window by first clicking on its name in the MRDPLOT-Command window, and then clicking on the desired graphics window in the MRDPLOT-Data window where the time series should be added. The important feature of MRDPLOT is that several variables can be displayed in parallel (i.e., 5 in the example of Figure 8), and, moreover, every graph can also have multiple data traces overlaid. For example, this latter feature is very useful to compare desired and actual trajectories of a DOF, while displaying multiple traces in parallel allows, for instance, diagnosing the cause of an instability in the system by inspecting which trajectory has the first occurrence of the instability.

The possible data manipulations of the MRDPLOT-Command window are explained in the following paragraphs. Note that all click-and-choose operations that require mul-

tiple mouse clicks have a time-out, i.e., if the user does not perform the next required mouse click within 5 seconds, the entire operation is reset. This feature will become obvious if you operate MRDPLOT too slowly.

5.5.4.1 MRDPLOT-Command Features

`Print`
Prints the current MRDPLOT-Command window to the default printer.

`Save`
Saves the current data to dxxxxx file (see above). This is useful if additional variable were created with the data manipulation buttons described below.

`Add/Sub`
Adds or subtracts two data time series. This feature is operated by first clicking the `Add/Sub` button, and then clicking the names of the two variables to be added or subtracted. The second time series is added/subtracted to/from the second one. A pop-up window will let you choose between addition and subtraction.

`Open…`
A new dxxxxx data file is read into the Command window, using a pop-up dialog window. The old data is discarded. Current visualizations in the Data window are refreshed with the new data. This latter feature requires that the new data file has data columns in the same order as the previously displayed data file and is very helpful to quickly debug many data files collected with the same data collection script file (see above).

`Zoom in`
Increases the amplification of the zoom window in the MRDPLOT-Data window.

`Zoom out`
Decreases the amplification of the zoom window in the MRDPLOT-Data window.

`Fast Open`
When collecting data files the `collect_data` facility, we automatically maintain a file names `last_data` in the directory where the data is stored. This file contains the number of the last data file stored, i.e., if the last data file was `d00123`, `last_data` will contain "123". Whenever a new data file is stored with `saveData`, this counter is automatically incremented. `Fast Open` first tries to open the `last_data` file, determine the name of the last stored data file (i.e., `d00123` in our example), and then open this file automatically in MRDPLOT. This feature allows avoiding the tedious open-dialog to read a new data file and is very handy to rapidly re-run simulations/robots, store data, and display data. If the "fast-open" fails, a normal interactive Open-dialog is entered.

`Phase Plot`
Two time series can be displayed in a phase plot. First, click the button, then selection the first and the second variable by clicking its name in the Command window. A new window will pop up displaying the phase plot with variable one as x-axis, and variable 2 as y-axis.

`d/dt`
This feature creates a new variable which is the time derivative of an existing variable. First, click the button, then the variable. A pop-up dialogue allows you to filter the time derivative with a Butterworth filter. Enter 0 or 100 for the filter cutoff frequency (this

number is in percent of the Nyquist frequency, i.e., half of the sampling frequency) to avoid filtering altogether. Note that you need the signal processing toolbox to be installed to use the filter. We may program our own Butterworth filter in the near future to avoid the need of this toolbox.

`Clear`
Clears one of the graphs of the Data window. First, click the button, then the graph you want to clear. Either the zoom or the non-zoom part of the graph can be clicked to identify the graph.

`Clear Last`
Removes the last added time series in a graph. This is useful if a graph displays multiple time series in superposition. First, click the button, then the graph in which you wish to eliminate the last time series. If only one time series is displayed in the graph, this function is identical to `Clear`.

`Clear All`
Click this button to clear all the graphs in the Data window.

`Add Plot`
Click this button to add an additional graph in the Data window.

`Del Plot`
Click this button to delete the top most graph in the Data window.

`Quit`
Terminates MRDPLOT

5.5.4.2 Utilities to convert data files

We offer two Matlab utilities to convert back-and-forth between the dxxxxx data format and regular Matlab matrices.

`mrdplot_convert`
Use "`help mrdplot_convert`" in the Matlab command window to obtain help on how to use the function. You will get the following text:

```
[D,names,units,freq] = mrdplot_convert(fname)
  converts an MRDPLOT binary file into a Matlab matrix. If fname is
  given, the file is processed immediately. If no filename is given,
  a dialog box will ask to located the file
  The program returns the data matrix D, the names of the columns and
  their units, as well as the sampling frequency
```

For example, execute `[D,names,units,freq]=mrdplot_convert('d00123')` to obtain the contents of the example file above.

`mrdplot_gen`
Use "`help mrdplot_gen`" in the Matlab command window to obtain help on how to use the function. You will get the following text:

```
mrdplot_gen(data,names,units,freq,fname)
  writes a data matrix into MRDPLOT binary file. The names
  matrix contains the variable names, units contains the units
  and freq the sampling frequency.
```

For example, execute `mrdplot_gen(D,names,units,freq,'newdata')` to generate a new data file called `newdata`.

5.6 The Configuration Files of *SL*

Many parameters in ***SL*** are robot specific, and they are kept in ASCII parameter files to simplify the tuning of these parameters, e.g., as commonly needed for gains. Configuration files are kept in `USERROOT/MYROBOT/config`, i.e., they are local to every user. We also suggest to have default configuration files in `SLROOT/MYROBOT/config` that serve as template for the user configuration files, and users either copy them for their purposes or create a symbolic link to them, if they do not want to make any changes.

In general, all configuration files are parsed according to some keywords, which is most of the time the name of a DOF as specified in `SL_user_common.h`. After each keyword, a fixed number of parameters follows. Thus, the order of keywords is not important in configuration files, while, however, the order of the parameters after the keyword is important. For each configuration file, we give an example from an actual 10 DOF anthropomorphic robot.

Configuration files are read only once at the time of startup of the robot/simulation.

5.6.1 Configuration Files for Simulations and Robots

`SensorOffsets.cf`

For every DOF, the permissible range of motion, characterized by a minimum and maximum value, is provided. Next, the desired position for a default posture of the simulation is given, i.e., the posture the simulation/robots starts out with when starting the simulation/robot or when using the `go0` command (see above). Three more values complete the dataset after each keyword: these are the joint angles for a rest position, the weight of the rest position, and an offset value. The offset value is only needed in real robots to offset the readings of real sensors by a fixed value in order to define a zero posture. For simulations, these values are not used and can be set to zero. The rest position and the weight are used for inverse kinematics computations. Redundancy is resolved by using deviations from the rest position as a penalty in an optimization criterion (Tevatia & Schaal, 2000). The larger the (strictly positive) weight, the more importance will be given that a DOF stays close to the rest position. Note that optimization is only done in the NULL space of the Jacobian of the forward kinematics, i.e., the optimization criterion is only used to resolve the redundancy of the robot/simulation and does not affect the endeffector position. Also note that weights cannot be chosen arbitrarily large as this will lead to numerical instabilities (e.g., maximum around 100-500, depending on the servo rate).

```
/* this file contains the specification for position offsets and min/max
   position values (min and max AFTER offset was subtracted.
   Additionally, the file allows to set a default posture for the robot.
   Note: the file is parsed according to keywords, after the keyword,
         the values need to come in the correct sequence */

/* format: keyword, min , max, default, rest, weight, offset  */
```

```
  R_SFE  -0.43 1.48    0    0     1    0.753
  R_SAA  -1.35 0.36 -0.3   -0.3     1    2.026
  R_HR   -1.65 1.45    0    0     .2   0.108
  R_EB    0.53 2.25 1.57    1.57  .5   0.124
  R_WR   -1.48 1.6     0    0     .2  -0.071
  R_WFE  -0.76 0.95    0    0     .5    1.140
  R_WAA  -0.67 0.82    0    0     2    1.327

  R_TL   -0.5 0.5      0    0     1    1.846
  R_TV   -0.5 0.5      0    0     1    1.273
  R_FAA  -0.5 0.5      0    0     1    1.310
```

SensorFilter.cf

In this file, the cutoff values for filtering position, velocity, acceleration, and load cell data are specified. ***SL*** uses 2nd order Butterworth filters for smoothing the sensory signals. Cutoff values need to be between 0 and 100, denoting the percent of the Nyquist frequency that should be used for cutoff. For instance, if the servo rate is 500Hz, the Nyquist frequency is 250Hz, and a cutoff value of 10 would become a 25Hz cutoff frequency. If the cutoff frequency is set to 100 or zero, not filtering is performed. Note that all filters are applied to raw data, i.e., if the same cutoff value is specified for position, velocity, acceleration, and load, then all the signals will be delayed by the same amount. This is usually very helpful to assure that differential relationships between position, velocity, and accelerations remain correct, i.e., that velocity is the time derivative of position, etc.

```
/* this file contains the filter specifications for all sensory data.
   For each DOF, cutoff frequency in % of the Nyquist frequency for a
   second order butterworth filter are given */
   Note: the file is parsed according to keywords, after the keyword,
         the values need to come in the correct sequence */

/* format: keyword, cutoff th, cutoff thd, cutoff thdd, cutoff load  */

  R_SFE  5 5 5 5
  R_SAA  5 5 5 5
  R_HR   5 5 5 5
  R_EB   5 5 5 5
  R_WR   5 5 5 5
  R_WFE  5 5 5 5
  R_WAA  5 5 5 5

  R_TL   5 5 5 5
  R_TV   5 5 5 5
  R_FAA  5 5 5 5
```

Gains.cf

The gain configuration file specifies position, velocity, and integral gains for every DOF. Additional, a maximal value is provided at which must not be exceeded by a control value. This latter feature mimics a saturation of an actuator in simulations

```
/* this file contains gains and max control for each DOF
```

```
   Note: the file is parsed according to keywords, after the keyword,
         the values need to come in the correct sequence */

/* format: keyword, gain_th, gain_thd, gain_int, max_control  */

  R_SFE  40 6     0.01  100
  R_SAA  40 4     0.01  100
  R_HR   30 2     0.01  100
  R_EB   30 2     0.01  100
  R_WR   20 0.5   0.01  100
  R_WFE  10 0.4   0.01  100
  R_WAA   8 0.35  0.01  100

  R_TL   10 0.05  0.01  100
  R_TV   10 0.05  0.01  100
  R_FAA  10 0.05  0.01  100
```

WhichDOFs.cf

The WhichDOFs configuration file allows to specify for which DOFs a processor is supposed to generate motor commands. After the keyword of the name of each processor, the names of the DOFs are listed that are to be used by it. There are separate entries for simulations and real robots. For most simulations, all DOFs would be listed after each processor name, except for the VisionServo, which does not create motor commands directly. For real robots, it may be useful to generate some motor commands on one processor, while others are computed somewhere else. A typical example is a humanoid robot where oculomotor control can be performed by a specialized oculomotor processor, while postural control will be taken care of in the normal way. In such a case, for instance, the task servo may not list the eye DOFs, and a separte EyeServo would get an entry in WhichDOFs to take care of eye motor commands. For most users, WhichDOFs will always list all DOFs for every processor.

```
/* this file contains a list of DOFs that are computed by each processor
   Note: the file is parsed according to keywords, after the keyword,
         the values need to come in the correct sequence */

/* format: keyword, <list of joint names for a processor> */

task_servo
              R_SFE  R_SAA  R_HR  R_EB  R_WR  R_WFE  R_WAA
task_sim_servo
              R_SFE  R_SAA  R_HR  R_EB  R_WR  R_WFE  R_WAA
              R_TL   R_TV   R_FAA
motor_servo
              R_SFE  R_SAA  R_HR  R_EB  R_WR  R_WFE  R_WAA
              R_TL   R_TV   R_FAA
motor_sim_servo
              R_SFE  R_SAA  R_HR  R_EB  R_WR  R_WFE  R_WAA
              R_TL   R_TV   R_FAA
vision_servo
vision_sim_servo
invdyn_servo
              R_SFE  R_SAA  R_HR  R_EB  R_WR  R_WFE  R_WAA
```

```
invdyn_sim_servo
              R_SFE   R_SAA   R_HR   R_EB   R_WR   R_WFE   R_WAA
```

5.6.2 Configuration Files for Simulations Only

`Objects.cf`

Objects in the environment can be created with the help of this configuration file. We currently support only trivial objects like cubes and spheres. Objects will be given a contact model with contact parameters such that ***SL*** can compute contact force when touching an object. Contact checking is performed automatically by ***SL***, but only for vertices of the simulated figure (e.g., the shoulder, the knee, the toe, the elbow, etc. of a humanoid), but not the volumetric connection between two vertices (e.g., the upper arm, etc.). This primitive kind of contact checking is very fast and does not slow down the simulation a lot. For instance, a walking or grasping simulation can easily be generated in this way.

```
/* this file contains a list of object specifications that can be added to the
   graphics scence. Each object must contain blank or tab delimited parameters in
   the following sequence:

       name  (max 20 characters)
        object type (e.g., CUBE=1, SPHERE=2, see SL_openGL.h)
        rgb color values (3 values)
        pos_x pos_y pos_z
        rot_x rot_y_rot_z
        scale_x scale_y scale_z
        contact_model  (e.g., DAMPED_SPRING_FRICTION=1)
        appropriate number of object parameters (N_OBJ_PARMS: see

       SL_openGL.h)
        appropriate number of contact parameters (N_OBJ_PARMS: see

       SL_openGL.h)

*/


floor 1
0.4 0.6 0.9
0.0 0.0 -1.3
0.0 0.0 0.0
4.0 4.0 0.1
1
0 0 0 0 0
0 0 0 0 0
```

`LinkParametersSim.cf`

Every link of the robot/simulation needs to be modeled in terms of its mass, center of mass vector, inertia matrix, and viscous friction parameter of the joint. We do not have separate models for the actuators at this point. For computational efficiency, the center of mass vector is given as in a multiplied form with the mass, i.e., mass*center_of_mass. The center of mass vector is in local coordinates relative to the link's local coordinates

system, which does not move relative to the link. The inertia matrix is the "parallel-axis-theorem-augmented" inertia, i.e., the inertia relative to the origin of the link's local coordinate system (which is usually different from the center of mass). The viscous parameter is a positive number. It is possible to have fake links with no mass and no inertia if the model of the robot is appropriate for this. Modeling of the robot/simulation will be described in the section on creating new simulations.

```
/* this file contains the specification for the dynamic parameters of
   all links */

/* format: keyword, mass, mcmx, mcmy, mcmz, i11, i12, i13, i22, i23, i33, vis */

R_SFE  0.000000  0.332146 -0.319979 -0.000000  0.000000  0.000000 -0.000000  0.000000
-0.000000  0.447291  0.488580
R_SAA -0.000000  0.954889 -0.641299 -0.319979  0.286594 -0.044948  0.119724  0.160697
-0.057820  0.015053  0.450896
R_HR   0.024871 -0.052802 -0.054841  0.953177  0.132839 -0.039734 -0.019548  0.042911
-0.002109  0.024650  0.139699
R_EB   0.312909 -0.342849 -0.083490  0.054235  0.052076 -0.050766 -0.041909  0.105413
-0.010289  0.090583  0.055556
R_WR   0.312775  0.059903  0.071984  0.342954  0.091818 -0.009149  0.000626  0.104177
-0.023716 -0.012890  0.141069
R_WFE  0.423309  0.069862  0.028644  0.078580  0.049361  0.003948 -0.001001  0.029567
-0.000940  0.016148  0.098514
R_WAA  0.423950 -0.062636  0.096681  0.069870  0.019466  0.003221 -0.000951  0.026249
-0.001493  0.010804  0.217428
R_TL   0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
0.000000  0.000000  0.000000
R_TV   0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
0.000000  0.000000  0.000000
R_FAA  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
0.000000  0.000000  0.000000
```

5.6.3 Configuration Files for Robots Only

`LinkParameters.cf`

A real robot can a different LinkParameter file than its associated simulation. Thus, we maintain two separate parameter files for this purpose. Note that under ideal circumstance, the simulation would use the same parameters as the real robot. However, when estimating the link parameters of a real robot, it is sometimes possible to obtain physically impossible link parameters (i.e., negative entries on the diagonal of the inertia matrix) due to unmodeled nonlinearities. To our experience, such implausible parameters work still rather well with the real robot, while the simulation would become unstable immediately. Such parameters need to be corrected for the simulation, thus motivating the LinkParameterSim file above.

`SensorCalibration.cf`

In our work, we experienced both linear and rotary actuators, sensors, and load cells. This configuration file allows specifying important information about such hardware arrangements. The first three parameters simply determine the kind of sensor, actuator, and load cell. A linear load cell denotes a load cell that is, for instance, mounted in sequence with a linear actuator—however, a linear actuator could also be combined with

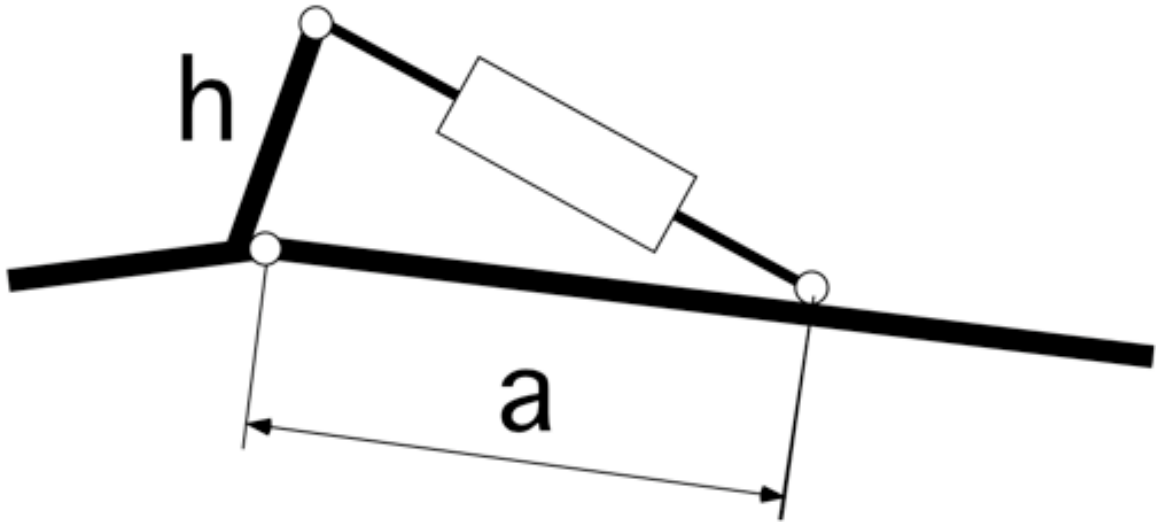

Figure 9: Sketch of linear actuator

a rotary load cell if the linear actuator is used to create a torque about a rotation axis, as shown in Figure 9. The moment arm *a* and length_from_attachment" *h* are illustrated in Figure 9, and are only needed if a linear actuator is used. Additionally, the SensorCalibration file also allows inverting the polarity of position and load cell sensors.

```
/* this file contains the specification such that all linear actuators
   and encoders can be represented as torque and rad values.
   Note: the file is parsed according to keywords, after the keyword,
         the values need to come in the correct sequence */

/* format: keyword,
           linear sensor (1/0),
           linear actuator (1/0),
           linear load cell (1/0),
           moment arm,
           length from attachment of linear actuator to joint axis
          (every linear value is in meters),
           polarity of position sensor (1/-1),
           polarity of load (1/-1)
*/


  R_SFE  0  1  0 0.0508  0.23622  1 1
  R_SAA  0  1  0 0.0508  0.23622  1 1
  R_HR   0  0  0 0       0        1 1
  R_EB   0  1  0 0.03175 0.1397   1 1
  R_WR   0  0  0 0       0        1 1
  R_WFE  0  1  0 0.0254  0.10795  1 1
  R_WAA  0  1  0 0.0254  0.10795  1 1

  R_TL   0  1  0 0.0127  0.06096  1 1
  R_TV   0  1  0 0.0127  0.06096  1 1
```

```
  R_FAA  0  1  0 0.0127  0.06096  1 1
```

5.7 The Preference Files of *SL*

While configuration files are usually tuned only once and then hardly touched again, preference files are files that may be added and modified quite frequently. Preference files specify parameter setting for various tasks of ***SL*** programs. Their naming is usually arbitrary but some conventions may help to make them easier to recognize. In the next sections, we describe the four classes of preference files that are used by ***SL*** by default. Users may want to create tasks that use their own preference files, and we would recommend to store those in the `USERROOT/MYROBOT/prefs` directory.

5.7.1 Data Collection Script Files

The data collection file script files were already described in Section 5.5.2. They usually have the suffix ".script". Arbitrary many data collection files can be generated.

5.7.2 Vision Post Processing Files

Vision post processing files specify how the raw positions of color blobs collected by the vision system should be post-processed. Either Butterworth or Kalman filtering can be applied for every blob (Kalman filters are not implemented yet). For Butterworth filtering, cutoff frequencies in percent of the Nyquist frequency are given in the same way as for the SensorFilter configuration file above. The PREDICT keyword specifies whether the post-processing should predict the position of the color blobs into the future by a certain amount of time. This is useful if the vision data has too much delay—however, prediction is only based on a second-order linear dynamical system, i.e., using the velocity and acceleration information of the blob. As a last point, it is possible to specify that a blob has a specific acceleration in a particular direction, e.g., as useful for objects under gravitational acceleration. Such a specified acceleration would overwrite any acceleration value determined from numerical differentiation of the blob position and velocity.

```
/* this file contains information how to post process the vision blobs.
   Butterworth filtering and Kalman filtering are possible. Additionally,
   coefficents for the acceleration of a blob can be provided, if known */

/* format: keyword, 9 number for butterworth cutoffs, 9 number for
           kalman filter coefficients, 3 number for acceleration vector,
           one number for prediction */

PREDICT
0.033

BLOB1_BUTTER
20   20  20       /* x, y, z position     */
100 100 100     /* x, y, z velocity     */
100 100 100     /* x, y, z acceleration */
```

```
BLOB1_ACCELERATION
0 0 0

BLOB2_BUTTER
20 20 20        /* x, y, z position     */
100 100 100     /* x, y, z velocity     */
100 100 100     /* x, y, z acceleration */

BLOB2_ACCELERATION
0 0 0


BLOB3_BUTTER
20 20 20        /* x, y, z position     */
100 100 100     /* x, y, z velocity     */
100 100 100     /* x, y, z acceleration */

BLOB3_ACCELERATION
0 0 0
```

5.7.3 Sine Files

Sine files are used by the sine_task in order to specify the sinusoidal motion of each DOF. As described in Section 5.1, a *.sine parameter file specifies the number of superimposed sine waves, the mean position of the sine, and then for every sine the amplitude, phase, and frequency. At most 3 sine waves can be superimposed. Non specified DOFs will not move and stay at the default posture of the robot/simulation.

```
/* keyword  #sines offset  amplitude  phase freq */

R_SFE       1   0.3     0.1        0.0   1.0
R_SAA       1  -0.4     0.1        0.0   1.0
R_HR        1   0.0     0.1        0.0   1.0
R_EB        1   1.57    0.1        0.0   1.0
R_WR        1   0.0     0.1        0.0   1.0
R_WFE       1   0.0     0.1        0.0   1.0
R_WAA       1   0.0     0.1        0.0   1.0

R_TL        1   0.0     0.1        0.0   1.0
R_TV        1   0.0     0.1        0.0   1.0
R_FAA       1   0.0     0.1        0.0   1.0
```

5.7.4 Trajectory Files

Trajectory files are files that contain a desired trajectory for tracking, as described in Section 5.1. The file format is the same as specified in Section 5.5.3. When such a file is read by the traj_task, the names of the variables contained in the trajectory file are compared with the joint names of the robot/simulation. If there is a match, the corresponding data for a DOF in the trajectory file will become the desired trajectory for the robot. DOFs that have no desired trajectory in the file will remain at the default position. If for all desired trajectories the first and last point in the trajectory are numerically identical,

SL interprets the trajectory as a rhythmic trajectory and repeats it until the robot/simulation is stopped manually (e.g., using the `freeze` command).

5.8 Contact Models in *SL*

SL includes a simple mechanism for detecting collisions with world objects, and to compute contact forces with these objects with the help of penalty methods, e.g., spring-damper models. More sophisticated contact models could be incorporated if need, however, the ease to setup penalty-based forces made this approach our initial choice.

Every link in the `link_pos` structure can be a contact point. A force or torque acting at this contact point will create forces and moments at the DOF that created an entry in the `link_pos` structure, which enter the simulation in the `uext` variable. The user can also choose to add forces and moments in the `uext` variable manually — contact forces are added to these user created `uext` entries. Note that `uext` is expressed in world coordinates.

For contact force computation, the main issue is contact detection and the computation of the contact forces. Contact detection is rather simple. *SL* only checks whether any of the `link_pos` coordinates insects with an *SL* object. Given that usually rather few objects are used and these objects are of primitive shape like cubes and spheres, contact checking is a fast process. If identified as a contact point, the following information is computed.

The first point of intersection with the object is $\mathbf{p}_{first}$. The current point of intersection is $\mathbf{p}$. The normal from $\mathbf{p}$ to the nearest face of the object is $\mathbf{n}$. The vector $\mathbf{t}$ is computed from $(\mathbf{p} - \mathbf{p}_{first})$ by only keeping the components that are orthogonal to $\mathbf{n}$. Moreover, we compute the relative velocity of $\mathbf{p}$, i.e., $\dot{\mathbf{p}}$, and split it into a normal and tangential velocity, $\dot{\mathbf{n}}$ and $\dot{\mathbf{v}}$. Note that $\dot{\mathbf{v}} \neq \dot{\mathbf{t}}$, as $\mathbf{t}$ is relative to the intial point of contact $\mathbf{p}_{first}$. However, we also compute $\dot{\mathbf{t}}$.

The rational behind the variables is straightforward. $\mathbf{n}, \dot{\mathbf{n}}$ are needed to compute damped spring-like contact forces that avoid strong penetration between the contact point and the object. $\dot{\mathbf{v}}$ is needed for viscous friction tangential to the contact surface. $\mathbf{t}, \dot{\mathbf{t}}$ to simulate Coulomb friction by imagining that Coulomb friction is like a spring in the tangential plane that connects $\mathbf{p}$ to $\mathbf{p}_{first}$. If this spring force becomes too large, as modeled by the friction coefficient, the spring breaks, and only viscous friction remains.

5.8.1 A Damped-Spring Viscous Friction Contact Model

Among the simplest contact models is the linear damped spring with viscous friction in the tangential place of the contact. Forces are computed as:

$$\mathbf{F} = k_n \mathbf{n} + b_n \dot{\mathbf{n}} - b_v \dot{\mathbf{v}}$$

with k_n, b_n, b_v denoting the spring constant, spring damping, and viscous damping coefficients. Note that the sign for spring damping term is correct here, as the normal forces are already computed as reaction forces on the contact point, and not as reaction forces of the contact point on the body.

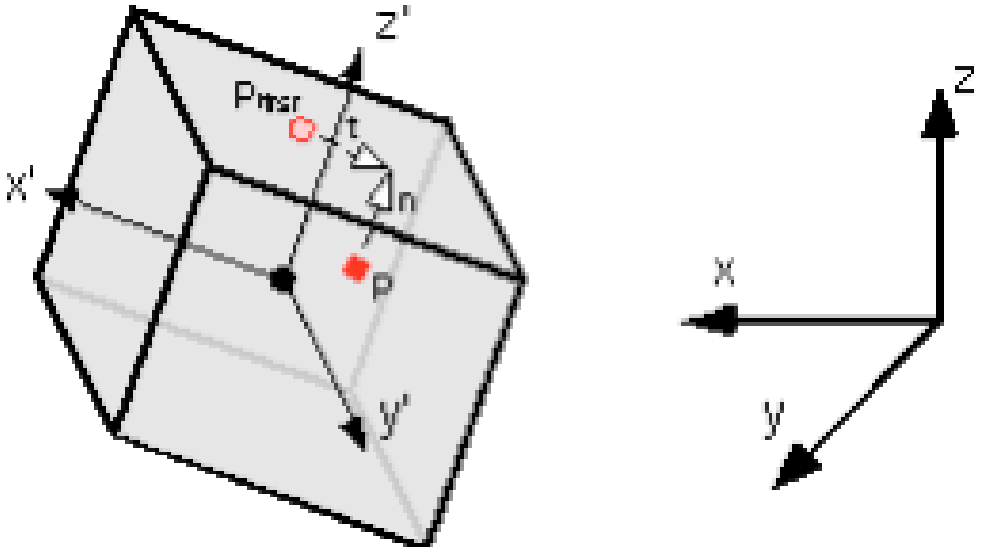

Figure 10: Sketch of variables computed for a contact with an external object.

5.8.2 A Damped-Spring Coulomb Friction Contact Model

In order to add Coulomb friction the forces are computed in a slightly more complex way:

$$\mathbf{F}_{c,s} = k_c \mathbf{t} + 2\sqrt{k_c}$$
$$\mathbf{F}_n = k_n \mathbf{n} + b_n \dot{\mathbf{n}}$$
$$\mathbf{F}_{c,d} = |\mathbf{F}_n| c_d \frac{\dot{\mathbf{v}}}{\|\dot{\mathbf{v}}\|}$$
$$\mathbf{F} = \begin{cases} \mathbf{F}_n - \mathbf{F}_c & \text{if } |\mathbf{F}_c| < c_s \mathbf{F}_n \\ \mathbf{F}_n - \mathbf{F}_{c,d} & \text{otherwise} \end{cases}$$

with k_n, b_n, k_c, c_s, c_d denoting the spring constant, spring damping, spring constant modeling the static friction, the static friction coefficient, and the dynamic friction coefficient.

If the spring modeling the static friction breaks lose, dynamic friction prevails until the tangential velocity of **p** becomes close to zero to allow static friction again. Note that the spring modeling the static friction force is endowed with a critical damping term to avoid oscillations in the tangential plane.

5.9 Rigid Body Orientations in *SL*

All orientations in ***SL*** are expressed as unit quaternions:

$$\mathbf{q} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} \cos\dfrac{\alpha}{2} \\ \mathbf{r}\sin\dfrac{\alpha}{2} \end{bmatrix} \text{ where } \|\mathbf{r}\| = 1$$

where **r** is a unit vector about which the rigid body is rotated with angle α. The rotation matrix from global to local coordinates becomes

$$\mathbf{R} = \begin{bmatrix} -1 + 2q_0^2 + 2q_1^2 & 2(q_1q_2 + q_0q_3) & 2(q_1q_3 - q_0q_2) \\ 2(q_1q_2 - q_0q_3) & -1 + 2q_0^2 + 2q_2^2 & 2(q_2q_3 + q_0q_1) \\ 2(q_1q_3 + q_0q_2) & 2(q_2q_3 - q_0q_1) & -1 + 2q_0^2 + 2q_3^2 \end{bmatrix}$$

and the inverse is the transpose of this matrix.

For converting back-and-forth between derivative information in quaternions and instantaneous angular quantities, the following formulae apply. First define a matrix expansion

$$\mathbf{Q}(\mathbf{q}) = \begin{bmatrix} -q_1 & -q_2 & -q_3 \\ q_0 & q_3 & -q_2 \\ -q_3 & q_0 & q_1 \\ q_2 & -q_1 & q_0 \end{bmatrix}$$

Then, the velocity conversions are:

$$\dot{\mathbf{q}} = \frac{1}{2}\mathbf{Q}(\mathbf{q})\dot{\mathbf{a}} = \frac{1}{2}\mathbf{Q}(\mathbf{q})\omega$$

$$\dot{\mathbf{a}} = \omega = 2\mathbf{Q}^T(\mathbf{q})\dot{\mathbf{q}}$$

where the angular velocity is either denoted as $\dot{\mathbf{a}}$ or ω. Note that the angular velocity is expressed in world coordinates. An alternative formulation exists where the angular velocity is given in local coordinates, which just changes some of the signs in the definition of **Q**. Second derivatives are:

$$\ddot{\mathbf{q}} = \frac{1}{2}\mathbf{Q}(\mathbf{q})\ddot{\mathbf{a}} + \frac{1}{2}\mathbf{Q}(\dot{\mathbf{q}})\dot{\mathbf{a}}$$

For numerical integration of the equations of motion, the Featherstone algorithms return $\ddot{\mathbf{a}}$. This can be integrated to $\dot{\mathbf{a}}$, then converted to $\dot{\mathbf{q}}$, which is finally integrated to **q**. Alternatively, $\ddot{\mathbf{a}}$ can be converted to $\ddot{\mathbf{q}}$, which is then double-integrated to **q**. One paper claims that the latter scheme is numerically more robust.

6 The User Interface of *SL*

The ***SL*** simulation user interface consists of a console window and one or multiple graphics windows, depending on the particular use of ***SL***.. A screen shot under Mac OS X is provided in Figure 11. When clicking on the graphics window, a pop-up menu ap-

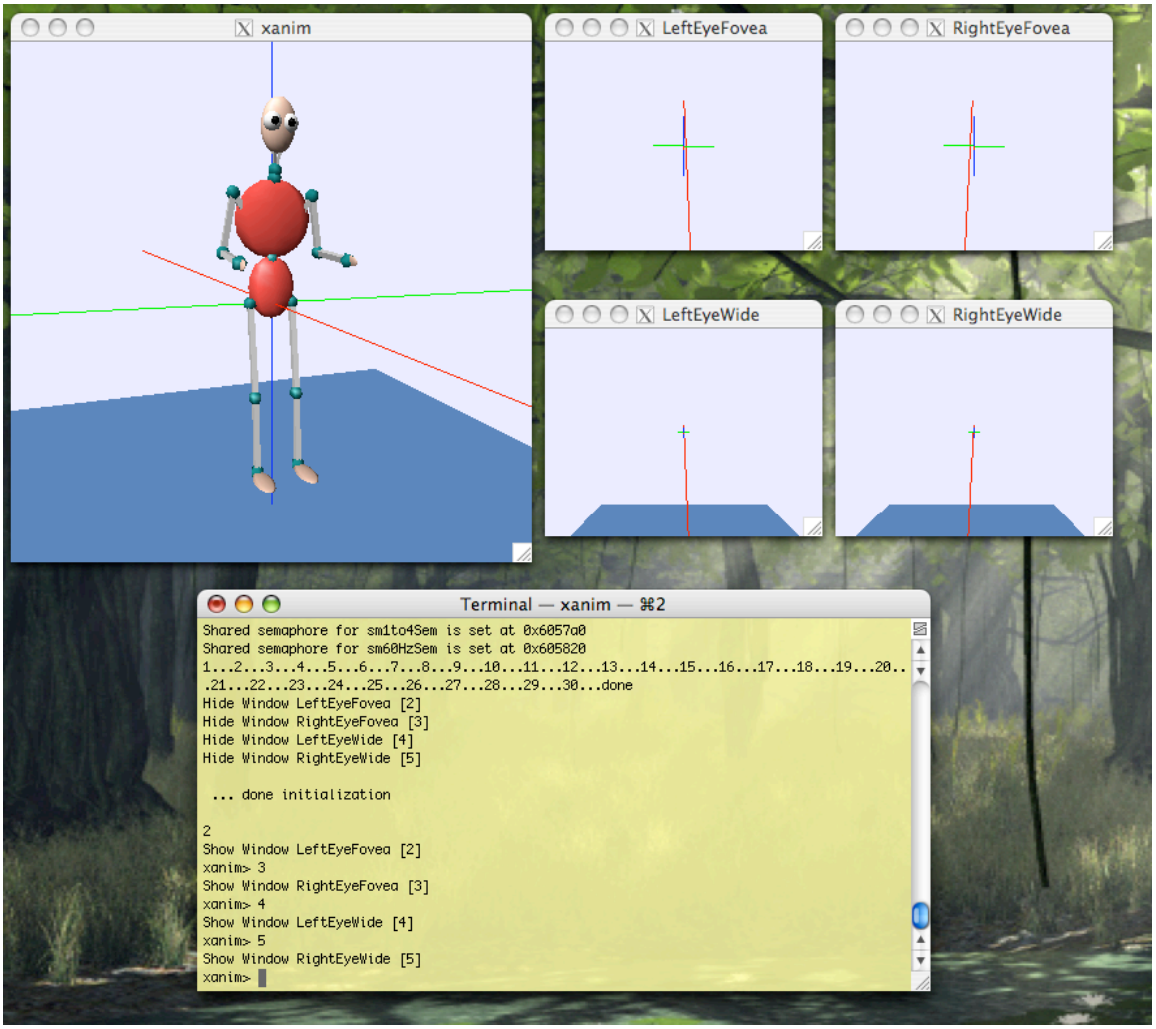

Figure 11: Screen shot of a humanoid ***SL*** simulation.

pears that allows changing the viewpoint of the simulation—the same can be accomplished with the arrow keys of the keyboard. The console window allows typing command as explained in Section 5.3.1. Both console window and graphics window can be made much more elaborate by modifying the code in `SLROOT/MYROBOT/src`, in particular the `SL_user.c` file. Our current goal was primarily computational efficiency and functionality of the ***SL*** user interface, not beauty.

7 Creating New Simulations With *SL*

The experienced ***SL*** programmer can generate rather complex new simulations within a few hours or a day. The easiest way to get started is to duplicate an existing robot/simulation in `SLROOT`, e.g., `SLROOT/MYROBOT`, and then modify the files appropriately. By going through all the files in `SLROOT/MYROBOT` sequentially, the necessary modifications should be rather obvious.

In the following, we will describe a useful sequence of steps to accomplish a new simulation. We will assume that the new simulation is the directory `SLROOT/MYROBOT` to simplify the connection to the previous sections of this manual.

7.1 The Mathematica Input File

The most important file definition file for a new simulation is an ASCII file that describes the kinematic robot structure and various variables for dynamic parameters. From this file, Mathematica can automatically generate all necessary mathematics files that model the dynamics and kinematics of the new robot/simulation.

For every DOF, the mathematica contains a rather simple list of elements that describe this DOF:

```
{
{jointID,{ID=1}},
{jointAxis,{0,0,1}},
{translation,{rx,ry,rz}},
{rotationMatrix,{0,Pi/4,0}},
{successors,{2}},
{inertia,GenInertiaMatrixS["links",ID,1]},
{massCenterMass,GenMCMVectorS["links",ID,1]},
{mass,GenMassS["links",ID]},
{jointVariables,GenVariablesS["state",ID]},
{extForce,GenExtForceS["uex",ID]}
}
```

This list is in Mathematic notation and can be parsed directly by Mathematica. Indeed, the purpose of the Mathematica file is to create a Mathematica-readable description of the robot/simulation and to exploit the symbol manipulation power of Mathematica to generate automatically all C-code that is needed for the mathematical part of a simulation/robot. However, using the power of Mathematica comes at a small, yet annoying, price: we need to strictly adher to syntax of Mathematic. The slightly negative characteristics of this prerequisite are:

- Many curly brackets, similar to the brackets in LISP
- Variable names MUST NOT have any underscore characters "_" or dots "." since these are special syntax characters in Mathematica.
- Array are denote in a double-bracket notation, i.e., A[[1]] is the first coefficient of vector A, A[[1,2]] is the coefficient of row one and column two of matrix A.

Given these constraints, it is easy to specify a robot/simulation structure. The list above has 11 sublists:

- `jointID`: every DOF is given an ID number. Note that this ID number will be used to index into a C array and you should make sure to use only numbers from 1 to N_DOFS, i.e., the number of degrees of freedom of the simulation. (N_DOFS does not include the degrees of freedom of a floating base, only the actuated degrees of freedom). The ordering of the DOFs is otherwise arbitrary, but it is useful to group DOFs according to physical properties, i.e., the right arm DOFs have ID numbers 1 to 7, the left arm 8-14, etc. A special ID number "0" is reserved for the base coordinate system (which can be a floating base).

- `jointAxis`: Every DOF has its own local coordinate system attached to the link it belongs to. Figure 12 illustrates this in an example where the "prime" coordinates are attached to the right link, rotating about axis z'. The `jointAxis` vector is the unit vector that points into the direction of the joint axis as specified in local (=primed) coordinates. In Figure 12, it would be {0,0,1}, as also specified in the Mathematica list above.
- `translation`: this sublist specifies a vector from the previous to the current coordinate system, expressed in the previous coordinate system. In Figure 12, the vector **r** is the translation vector, and it would be expressed in terms of its components {rx,ry,rz} given in the (x,y,z) coordinate system. Essentially, the translation shifts the previous coordinate system's origin to coincide with the new coordinate system's origin.
- `rotationMatrix`: after the translation, the previous coordinate system is rotated to coincide with the current (x',y',z') coordinate system. This is accomplished by three Euler angle rotations, first about the x-axis, then the y-axis, and at last the z-axis. The new (local) coordinate system can be chosen arbitrarily, but it is usually beneficial to chose it such that one of the coordinate axis coincides with the rotation axis of the current DOF, as shown in Figure 12. It is also computationally more efficient to have as many zeros as possible in the specification of rotation angles and translations, as all these zeros will reduce the complexity of ensuing math C-code.
- `successors`: each DOF can have none, one, or multiple DOFs as successor. The successors are specified by their ID numbers (see above). Note that multiple successors mean that there is a branching structure in the robot, e.g., a trunk that has two arms. ***SL*** can easily deal with this, but it is not allowed to have closed loop structures, e.g., DOF1 connects to DOF2, which then connects to DOF3 again. Closed loops are mathematically too involved. The user has to make sure that the branching structure creates "open chains" and that the numbering scheme is reasonable and creates compact C-arrays (i.e., no unused elements in the C-array).
- The list elements `inertia`, `massCenterMass`, `mass`, `jointVariables`, and `extForce` are created automatically from the Gen* functions. Each of the Gen* functions generates appropriate variables that will be used by ***SL*** to denote the mass, the mass multiplied with the center of mass vector, the inertia matrix in local joint coordinates (but not necessarily center of mass coordinates), the variables denoting joint positions, velocities, accelerations, and torques, and finally external forces.

The Mathematica file `RigidBodyDynamics.m` declares all the math functions used by ***SL***. The Mathematica programming language is a bit confusing, particularly since it allows recursive programming. Simple functions like the Gen* function above, however, should be easy to understand. The implementation of dynamics and kinematics equations, however, require a thorough understanding of rigid body dynamics as described by Featherstone (Featherstone, 1987).

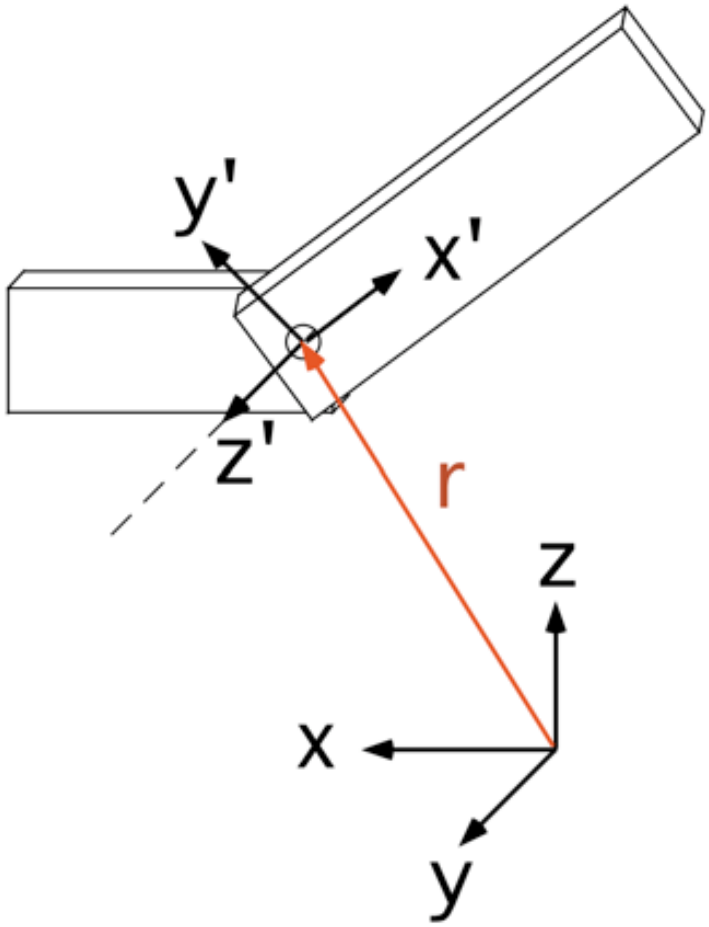

Figure 12: Coordinate transformation from global to link coordinates

A complete robot/simulation is specified by many of the list elements above, describing the transition from coordinate system to coordinate system and the associated DOFs. It is useful to create sketched graphical illustration of the robot kinematics and the local coordinate systems at each point. Note that translations and rotation matrices in the lists can be zero vectors ({0,0,0}) or zero matrices ({{0,0,0},{0,0,0},{0,0,0}}), respectively, which will create more efficient C-code. A joint that has multiple DOFS, e.g., like a ball-joint, can be modeled by several successive roational joints with different rotation axes. For such special joints, there will exist DOFs that have zero inertia and zero mass—these DOFs create "fake" links. The following shows an example of a complete Mathematica file to create a right arm simulation:

```
{ (******************* left arm ******************)
(* left shoulder FE (L_SFE), forward is positive *)
{jointID,{ID=1}},
{jointAxis,{0,0,1}},
{translation,{ZSH,-XSH,YSH}},
{rotationMatrix,{-Pi/2,0,0}},
{successors,{2}},
{inertia,GenInertiaMatrixS["links",ID,1]},
{massCenterMass,GenMCMVectorS["links",ID,1]},
{mass,GenMassS["links",ID]},
{jointVariables,GenVariablesS["state",ID]},
{extForce,GenExtForceS["uex",ID]}
}
{ (* left shoulder AA (L_SAA), outward is negative *)
{jointID,{ID=2}},
{jointAxis,{0,0,1}},
{translation,{0,0,0}},
{rotationMatrix,{-Pi/2,0,0}},
{successors,{3}},
{inertia,GenInertiaMatrixS["links",ID,1]*0},
{massCenterMass,GenMCMVectorS["links",ID,1]*0},
{mass,GenMassS["links",ID]*0},
{jointVariables,GenVariablesS["state",ID]},
{extForce,GenExtForceS["uex",ID]*0}
}
{ (* left HR (L_HR), outward is positive *)
{jointID,{ID=3}},
{jointAxis,{0,0,1}},
{translation,{0,0,0}},
{rotationMatrix,{0,Pi/2,0}},
{successors,{4}},
{inertia,GenInertiaMatrixS["links",ID,1]},
{massCenterMass,GenMCMVectorS["links",ID,1]},
{mass,GenMassS["links",ID]},
{jointVariables,GenVariablesS["state",ID]},
{extForce,GenExtForceS["uex",ID]*0}
}
{ (* left elbow (L_EB), flex is positive *)
{jointID,{ID=4}},
{jointAxis,{0,0,1}},
{translation,{0,0,-UPPERARM}},
{rotationMatrix,{Pi/2,0,0}},
{successors,{5}},
{inertia,GenInertiaMatrixS["links",ID,1]*0},
{massCenterMass,GenMCMVectorS["links",ID,1]*0},
{mass,GenMassS["links",ID]*0},
{jointVariables,GenVariablesS["state",ID]},
{extForce,GenExtForceS["uex",ID]}
}
{  (* left wrist rotation (L_WR), outward is positive *)
{jointID,{ID=5}},
{jointAxis,{0,0,1}},
{translation,{0,0,0.}},
```

```
{rotationMatrix,{-Pi/2,0,0}},
{successors,{6}},
{inertia,GenInertiaMatrixS["links",ID,1]},
{massCenterMass,GenMCMVectorS["links",ID,1]},
{mass,GenMassS["links",ID]},
{jointVariables,GenVariablesS["state",ID]},
{extForce,GenExtForceS["uex",ID]*0}
}
{  (* left WFE (L_WFE), flex is positive *)
{jointID,{ID=6}},
{jointAxis,{0,0,1}},
{translation,{0,0,-FOREARM}},
{rotationMatrix,{0,-Pi/2,0}},
{successors,{7}},
{inertia,GenInertiaMatrixS["links",ID,1]*0},
{massCenterMass,GenMCMVectorS["links",ID,1]*0},
{mass,GenMassS["links",ID]*0},
{jointVariables,GenVariablesS["state",ID]},
{extForce,GenExtForceS["uex",ID]}
}
{  (* left WAA (L_WAA), forward is positive *)
{jointID,{ID=7}},
{jointAxis,{0,0,1}},
{translation,{0,0,0}},
{rotationMatrix,{Pi/2,0,0}},
{successors,{108,8}},
{inertia,GenInertiaMatrixS["links",ID,1]},
{massCenterMass,GenMCMVectorS["links",ID,1]},
{mass,GenMassS["links",ID]},
{jointVariables,GenVariablesS["state",ID]},
{extForce,GenExtForceS["uex",ID]}
}
{ (* the end-effector coordinate system becomes a static system *)
{jointID,{ID=108}},
{jointAxis,{0,0,0}},
{translation,{eff$1$$x[[1]],eff$1$$x[[2]],eff$1$$x[[3]]}},
{rotationMatrix,{eff$1$$a[[1]],eff$1$$a[[2]],eff$1$$a[[3]]}},
{successors,{}},
{inertia,{{0,0,0},{0,0,0},{0,0,0}}},
{massCenterMass,{eff$1$$mcm[[1]],eff$1$$mcm[[2]],eff$1$$mcm[[3]]}},
{mass,{eff$1$$m}},
{jointVariables,{0,0,0,0,0}},
{extForce,{0,0,0,0,0,0}}
}
{  (* left thumb lateral (L_TL), away from palm is positive *)
{jointID,{ID=8}},
{jointAxis,{0,0,1}},
{translation,{-ZTHUMB,-YTHUMB,XTHUMB}},
{rotationMatrix,{-Pi/2,0,0}},
{successors,{109}},
{inertia,GenInertiaMatrixS["links",ID,1]},
{massCenterMass,GenMCMVectorS["links",ID,1]},
{mass,GenMassS["links",ID]},
```

```
{jointVariables,GenVariablesS["state",ID]},
{extForce,GenExtForceS["uex",ID]}
}
{ (* dummy to draw thumb *)
{jointID,{ID=109}},
{jointAxis,{0,0,0}},
{translation,{-THUMB,0,0}},
{rotationMatrix,{0,0,Pi/2}},
{successors,{}},
{inertia,{{0,0,0},{0,0,0},{0,0,0}}},
{massCenterMass,{0,0,0}},
{mass,{0}},
{jointVariables,{0,0,0,0,0}},
{extForce,{0,0,0,0,0,0}}
}
```

There are several important elements in this example that should be noted:

- The list elements use many additional variables, e.g., for coefficiens of he translation vectors. The variables need to be declared in `SL_user.h`, usually with #define statements. The `SL_user.h` example in the appendix has the corresponding definitions.
- At joint ID=7, a branch to three other DOFs is performed.
- Joint ID=108, as one might guess from the number, is special. We call it a "dummy" DOF since it actually has no degree-of-freedom, i.e., the joint variables are all zero, such that the ID number can be chosen arbitrarily. All what happens at this DOF is that the previous coordinate system is translated and rotated in a static way. There are several reasons to create dummy DOFs:
  - They will help create appropriate graphics: without the dummy, one would basiclly not see the endeffector in the animation, e.g., a finger, since from a dynamics point of view all information about the finger was already captured in the previous coordinate system.
  - The endeffector coordinate system can be made more intuitive. While most local coordinate systems are chosen to create as many zeros in the translations and rotations as possible, the endeffector coordinate system is used with respect to external objects. It is thus useful to make it easy for human reasoning, i.e., by having the z-axis of the endeffector coincide with the direction of the finger.
  - While dummy DOFs have normally no influence on the dynamics equations, they are a convenient way to create contact points for collision checking. For instance, joint ID=109 above creates a thumb by translating by the vector `{-THUMB,0,0}` away from the thumb joint. As the orgin of the dummy coordinate system will create a new vertex in the kinematic chain of the robot/simulation, it will become a point for contact checking.
  - Dummies can model variable properties at an endeffector. This trick is used in ID=108. Instead of having static variables for translation, rotation, and mass, we used the variable `eff`. `eff` is a special C-structure, reserved for endeffectors, i.e., dummies that are important for manipulations in the external world (we would normally not consider the toes as an endeffector as they are hardly used for manipulating objects in the external world). The definition of `eff` can be found in `SL.h` in the appendix, called `SL_endeff`. With the help of `eff`, we can give a dummy

flexible properties, e.g., different lengths, differently oriented coordinate systems, and we can even add some mass to the endeffector, mimicking a load attached to the endeffector. In order to use the endeffector C-structure in the Mathematica definition, we had to do some syntactical tricks. For instance, the C-notation for the endeffector mass variable would be `eff[1].m`. Since Mathematic would completely misinterpret such a variable, we used `eff$1$$m` instead. When generating C-code, our Mathematica program now how to replace such $-symbols with the original notation, i.e., "`$`" will be replaced by "`[`" and "`$$`" will be replaced by "`].`". This notation is inconvenient, but not a big deal.

- Note the empty successor list for ID=108 and ID=109, both of which are dummy DOFs.
- Also note that Mathematica commends start with a "`(*`" and end with a "`*)`".

7.2 Generating the Math C-Code

Assuming the Mathematica definitions of the previous section are stored in an ASCII file named `myrobot.dyn`, a simple Mathematica session can create all necessary C-files. You need to start Mathematica on your computer, and issue the command in analogy to the following example:

```
SetDirectory["SLROOT/mathematica"];

<<RigidBodyDynamics.m

SetDirectory["SLROOT/MYROBOT/math"];

OpenGLKinematics["myrobot.dyn","myrobot"]

InvDyn["myrobot.dyn","myrobot",{0,0,-G}]

LinkEndpointKinematics["myrobot.dyn","myrobot"]

GeometricJacobian["myrobot.dyn",{108},"mrobot"]

ForDynArt["myrobot.dyn","myrobot",{0,0,-G}]
```

First, the directory is changed to the appropriate directory containing the `RigidBodyDynamics.m` file. This file is read into Mathematica according using the appropriate "<<" command. Then, we set the directory to the math subdirectory of the new robot/simulation. According to the sequence of commands, Mathematica will create OpenGL graphics, inverse dynamics math, kinematics math, the Jacobian of the kinematics, and the forward dynamics math. Each of the commands uses `myrobot.dyn` as input, and creates special C-header files with the prefix `myrobot`, e.g., `myrobot_ForDyn_math.h`. These files will be included by the appropriate C-files of ***SL***. Since ***SL*** was programmed without a particular robot/simulation name in mind, it will be necessary to create symbolic links to the prefixed header files, e.g., by issuing on a UNIX platform "`ln –s myrobot_ForDyn_math.h ForDyn_math.h`", i.e., the header file without the prefix. The prefix for the header files serves primarily to easily identify to which robot/simulation they belong, given that we assume that many robot/simulations are

created with ***SL***. In SLROOT/matlab, you will find a simple Matlab function that can creat the symbolic links automatically on a UNIX system.

Note that you can obtain a help text on each of the Mathematica functions used above by issuing the command "?<function_name>", e.g., "?GeometricJacobian". Also note that Mathematica requires to press shift-return to trigger a command line command—make sure the read the basics of Mathematica before using it.

For some of the Mathematic functions, additional arguments are needed. The help text in Mathematica will tell you about this. In brief, some function require to define the gravity acceleration vector in global coordinates, and the Jacobian calculation requires to specify which ID numbers in the myrobot.dyn are endeffectors. Multiple endeffectors can be given.

7.3 SL_user.h, SL_user_common.c, SL_user.c

There are three key C-files that require to be adjusted. SL_user.h and SL_user_common.c specify generic aparameters of the simulation/robot. Expamples are in the appendix of this document.

The first file to change is usually SL_user.h. We are going to refer the example of SL_user.h in the appendix in the following elaborations, and explain the key steps in the sequence in which they appear in this example

1. At first, all variables that were used in myrobot.dyn will be defined.
2. Second, an enumeration is given for all the links of the robot/simulation. To be more precise, assuming the robot/simulation is nothing but a stick figure where links connect the vertices of the stick figure, the enumeration of links actually denotes the enumeration of vertices of the robot/simulation. For instance, in a simple human arm, the vertices should be the shoulder joint, the elbow joint, the wrist joint, and the fingertip (assuming the arm has only one finger attached to the wrist). In order to have a fingertip, a dummy DOF needs to be defined in the Mathematica file, as explain above. A joint with multiple DOFs will only create on vertex, e.g., as in the example of the shoulder, which has three DOFs, but only one vertex. Thus, the number of DOFs is not equal to the number of vertices: dummy joints create extra vertices, while fake joints create no vertices. Thus, the enumeration of the RobotLinks in SL_user.h is actually a bit more complicated than expected. For this reason, we decided to leave it to Mathematica to find out what a good enumeration of the robot links is, and leave it to the user to extract this enumeration from the file myrobot_LEKin_math.h and enter it in SL_user.h. Edit myrobot_LEKin_math.h, and search for the variable Xlink. You will find the assignement of Xlink[1], Xlink[2], etc. Just above each of these definitions, there is comment containing the translation vector from the myrobot.dyn file that was used to get to this particular link (or, more precisely, vertex). From this comment, you should be able to infer which vertex of the simulation/robot is refered to. The ordering of Xlink defines the enumeration of RobotLinks, i.e., you must use the same indexing scheme, and the only choice you can make is to give a useful name to the elements of the enumeration.
3. Depending on how many endeffectors you defined in myrobot.dyn, you need to provide indices for these endeffectors, starting at 1 and ending at the number of endef-

fectors. Every DOF in `myrobot.dyn` that contains the `eff` variable will be an endeffector.

4. The next element to modify is the `RobotDOFs` enumeration. This enumeration follows your chosen number scheme in `myrobot.dyn`. Only actuated DOF need to be enumerated here. You can choose intuitive names in the enumeration—in our example be used anatomical notation from human arm anatomy.
5. All following definitions should be rather obvious since the make use of previously generated definitions. The definitions for `N_DOFS_EST` and `N_DOFS_EST_SKIP` are slightly unusual. The definitions are needed to automatically estimate inertial parameters in real robots. We will explain this in a later section on parameter estimation. For simulations, `N_DOFS_EST` should just be the number of actuated joints plus the number of fake joints in `myrobot.dyn`. `N_DOFS_EST_SKIP` can be set to zero in simulations.
6. The definition for `SERVO_BASE_RATE` sets the frequency of integration for simulations, and the servo rate of the motor servo. We usually use values between 500 to 1000Hz. If this rate is very high, the simulation will run rather slowly but have higher numerical robustness. In robots, we need to chose this rate such that we can still compute all necessary quantities in real time. Thus, this rate parameters really depends on the CPU power available and the complexity of the robot/simulation.
7. The `TASK_SERVO_RATIO` allows running the TaskServo at a reduced rate of the MotorServo. `TASK_SERVO_RATIO` thus defines the ratio of `motor_servo_rate/task_servo_rate` and it must be a positive integer number. We usually try to run the MotorServo as fast as possible, but the TaskServo at around 400-500Hz as it is not necessary to create new high level motor motor commands at a very high rate.

The second file to change is usually `SL_user_common.c`. We are going to refer the example of `SL_user.h` in the appendix in the following elaborations, and explain the key steps in the sequence in which they appear in this example.

1. There are four character arrays that the define names to the DOFs, the links (vertices), and endeffectors. The names need to be assigned in accordance with the enumerations in `SL_user.h`, described above, and it should be obvious how to do this.
2. The array `link2endeffmap` defines which of the links (vertices) correspond to which endeffector index. In our example, the enumeration variable HAND, which was defined in the enumeration of `RobotLinks`, will be the endeffector with index 1.
3. As a last point, the function `setDefaultEndeffector` needs to be instantiated. For every endeffector, the endeffector structure needs to be initialized. These values are going to be used in all the mathematics C-code according the the usage of the `eff` variable in `myrobot.dyn`.

The last file to change is `SL_user.c`. This is a longer program, and most of its contents should be rather obvious. There is only one part that really require change, the remainder can be left untouched. The `addUserDisplayFunc` is the place to examine more carefully. The default OpenGL graphics generated by ***SL*** create a simple 3D stick figure which is not necessarily very appealing. For every link of the robot, ***SL*** calls `addUserDisplayFunc` in order to create a graphics element. The switch statement in `addUserDisplayFunc` allows to break out of the default graphics and generate user specific graphics for

each link. The link can be recognized from the ID number that is passed as an argument to `addUserDisplayFunc`, and this ID number coincides with the user defined ID in `myrobot.dyn`. Initially, the user may wish to erase all special switches in `addUserDisplayFunc` and only use the default switch. This will create a decent stick figure. Afterwards, s/he may start modifying the graphics by adding more and more ID specific switches. Such programming can be accomplished with a few hours of studying of the OpenGL manuals.

7.4 user_commands.c

One more C-file, `user_commands.c`, needs to be modified. This file simply contains functions that the user finds handy to use in the command line interface. Primarily, these commands concerning functions that display at which position the robot/simulation current is. ***SL*** already provides several of these commands, e.g., the `where` command, but, for instance, for many DOF simulations, the print-out `where` is just too confusing and requires a more tailored print-out, e.g., only the arm positions, etc. By editing the provided `user_commands.c`, it should be obvious how to modify `user_commands.c`.

7.5 Symbolic Links

Several additional C-files are needed to complete the C-code of the new simulation. We recommend to copy or to make symbolic links to the templates of these files, that reside in `SLROOT/SL/src`. From our experience, these files do not need to be adjusted, but they need to be compiled with the user generated math header files. The relevant files are:

- `for_dynamics.c`
- `inv_dynamics.c`
- `kinematics.c`
- `sensor_proc.c`

7.6 Configuration and Preference Files

The last set of modifications needs to be made to the configuration and preference files. Those files were described above. The modifications in these files are simply concerned with specifying the new DOFs and their appropriate parameter settings for gains, inertial parameters, default posture, etc.

7.7 Compilation

The entire new software needs to be recompiled. The Imakefile facility included in ***SL*** should make this a simple effort. Imakefiles are very user friendly in understanding what is going on.

7.8 User Specific Code

At last, the every user needs to create a new directory for the new robot/simulation, and copy the configuration and preference files.

8 Changing the *SL* Libraries

Advanced users may want to change the ***SL*** libraries files in `SLROOT/SL`. This is not very complicated, but needs to be done very careful to ensure that all simulations/robots that use the ***SL*** libraries are not negatively affected. A complete description of the ***SL*** libraries is beyond the scope of this manual. The way to get started is to look at the Imakefile in `SLROOT/SL/MACHTYPE` and study the programs that are compiled together. The `motor_servo.c`, `task_servo.c`, `vision_servo.c`, `invdyn_servo.c`, and `invkin_servo.c` are the key files that implement the indivual servo loops. Following the function call in those files will reveal the modular elements how ***SL*** is built.

9 Using *SL* for Real-Time Control with VxWorks

Together with vxWorks, ***SL*** can create a multi-processor robot control system. Everything described above is valid for real robots, too, except for some simulation and graphics elements. ***SL*** includes Imakefiles of how to compile for vxWorks. Essentially, the MotorServo, VisionServo, TaskServo, InvDynServo, and InvKinServo are compiled into separte modules that are load on 5 different processors in a VME bus. These processors communicate through share memory. User specific code is simple loaded on top of the preloaded images in TaskServo—vxWorks does real-time linking. The user interface is almost identical to the one in simulation, except that each CPU has its own console window. We may extend the explanations on the vxWorks modules in the near future if there is a sufficient amount of interest.

10 References

Featherstone, R. (1987). *Robot dynamics algorithms*: Kluwer Academic Publishers.

Miyamoto, H., Schaal, S., Gandolfo, F., Koike, Y., Osu, R., Nakano, E., Wada, Y., & Kawato, M. (1996). A Kendama learning robot based on bi-directional theory. *Neural Networks*, **9,** 1281-1302.

Sciavicco, L., & Siciliano, B. (1996). *Modeling and control of robot manipulators*. New York: MacGraw-Hill.

Tevatia, G., & Schaal, S. (2000). Inverse kinematics for humanoid robots, *International Conference on Robotics and Automation (ICRA2000)*. San Fransisco, April 2000.

11 Appendix

11.1 The SL.h Header file

```
/*============================================================================

SL.h

by Stefan Schaal, Jan 1999
============================================================================
Remarks:

Generic definitions to create SL simulations

============================================================================*/

#ifndef _SL_
#define _SL_

#include "utility.h"

/* General header file for rigid body dynamics simulation environment */

#define START_INDEX 1
#define N_CART 3
#define _X_ (0+START_INDEX)
#define _Y_ (1+START_INDEX)
#define _Z_ (2+START_INDEX)
#define _A_ (0+START_INDEX)
#define _B_ (1+START_INDEX)
#define _G_ (2+START_INDEX)

#define N_QUAT 4
#define _Q0_ 1
#define _Q1_ 2
#define _Q2_ 3
#define _Q3_ 4

#define RIGHT 1
#define LEFT  2

/* defines for the servo frequencies */
#define R1TO1 1
#define R1TO2 2
#define R1TO3 3
#define R1TO4 4
#define R1TO5 5
#define R60HZ 60

/* defines for the preference and config files */
#ifdef MACOS
#define CONFIG ":config:"
#define PREFS  ":prefs:"
#else
#define CONFIG "config/"
#define PREFS  "prefs/"
#endif
```

```
#define SENSOR_CALIBRATION_FILE "SensorCalibration.cf"
#define SENSOR_OFFSET_FILE      "SensorOffset.cf"
#define SENSOR_FILTER_FILE      "SensorFilter.cf"
#define GAIN_FILE               "Gains.cf"
#define WHICH_DOFS_FILE         "WhichDOFs.cf"
#define OBJECTS_FILE            "Objects.cf"
#ifdef VX
#define LINK_FILE               "LinkParameters.cf"
#else
#define LINK_FILE               "LinkParametersSim.cf"
#endif

/* defines that are used to parse the config and prefs files */
#define MIN_THETA    1
#define MAX_THETA    2
#define THETA_OFFSET 3

#define LINEAR       1

#define SENSOR       1
#define ACTUATOR     2
#define MOMENTARM    3
#define MOUNTPOINT   4
#define THETA0       5
#define LOADCELL     6

#define DOUBLE2FLOAT 1
#define FLOAT2DOUBLE 2

/* The data structures */
typedef struct { /* joint space state for each DOF */
  double   th;   /* theta */
  double   thd;  /* theta-dot */
  double   thdd; /* theta-dot-dot */
  double   ufb;  /* feedback portion of command */
  double   u;    /* torque command */
  double   load; /* sensed torque */
} SL_Jstate;

typedef struct { /* joint space state for each DOF */
  float   th;   /* theta */
  float   thd;  /* theta-dot */
  float   thdd; /* theta-dot-dot */
  float   ufb;  /* feedback portion of command */
  float   u;    /* torque command */
  float   load; /* sensed torque */
} SL_fJstate;

typedef struct { /* desired values for controller */
  double   th;   /* theta */
  double   thd;  /* theta-dot */
  double   thdd; /* theta-dot-dot */
  double   uff;  /* feedforward torque command */
  double   uex;  /* externally imposed torque */
} SL_DJstate;

typedef struct { /* desired values for controller */
  float   th;    /* theta */
  float   thd;   /* theta-dot */
```

```
  float   thdd;  /* theta-dot-dot */
  float   uff;   /* feedforward torque command */
  float   uex;   /* externally imposed torque */
} SL_fDJstate;

typedef struct {  /* desired values for controller: short version for faster communication */
  char    status; /* valid data or not: needed for multi processing */
  float   th;     /* desired theta */
  float   thd;    /* desired velocity */
  float   uff;    /* feedforward command */
} SL_fSDJstate;

typedef struct { /* desired state for optimization */
  double   th;   /* desired theta */
  double   w;    /* feedforward command */
} SL_OJstate;

typedef struct { /* desired state for optimization */
  float   th;   /* desired theta */
  float   w;    /* feedforward command */
} SL_fOJstate;

typedef struct { /* Cartesian state */
  double   x[N_CART+1];    /* Position [x,y,z] */
  double   xd[N_CART+1];   /* Velocity */
  double   xdd[N_CART+1];  /* Acceleration */
} SL_Cstate;

typedef struct { /* Cartesian state */
  float   x[N_CART+1];    /* Position [x,y,z] */
  float   xd[N_CART+1];   /* Velocity */
  float   xdd[N_CART+1];  /* Acceleration */
} SL_fCstate;

typedef struct { /* Cartesian orientation */
  double   a[N_CART+1];    /* Position [alpha,beta,gamma] */
  double   ad[N_CART+1];   /* Velocity */
  double   add[N_CART+1];  /* Acceleration */
} SL_Corient;

typedef struct { /* Quaternion orientation */
  double   q[N_QUAT+1];    /* Position [q0,q1,q2,q3] */
  double   qd[N_QUAT+1];   /* Velocity */
  double   qdd[N_QUAT+1];  /* Acceleration */
  double   ad[N_CART+1];   /* Angular Velocity [alpha,beta,gamma] */
  double   add[N_CART+1];  /* Angular Acceleration */
} SL_quat;

typedef struct { /* Cartesian orientation */
  float   a[N_CART+1];    /* Position [alpha,beta,gamma] */
  float   ad[N_CART+1];   /* Velocity */
  float   add[N_CART+1];  /* Acceleration */
} SL_fCorient;

typedef struct { /* Vision Blob */
  char       status;
  SL_Cstate  blob;
} SL_VisionBlob;

typedef struct { /* Vision Blob */
```

```
  char        status;
  SL_fCstate  blob;
} SL_fVisionBlob;

typedef struct { /* 2D Vision Blob */
  char        status[2+1];
  double blob[2+1][2+1];
} SL_VisionBlobaux;

typedef struct { /* 2D Vision Blob */
  char        status[2+1];
  float blob[2+1][2+1];
} SL_fVisionBlobaux;

/* a structure for raw 3D vision blobs */
typedef struct {
  char       status;
  double     x[N_CART+1];
} Blob3D;

/* a structure for raw 3D vision blobs */
typedef struct {
  char     status;
  float    x[N_CART+1];
} fBlob3D;

/* a structure for raw 2D vision blobs */
typedef struct {
  char       status;
  double     x[2+1];
} Blob2D;

/* a structure for raw 2D vision blobs */
typedef struct {
  char     status;
  float    x[2+1];
} fBlob2D;

typedef struct { /* Muscle state */
  double   l;   /* length */
  double   ld;  /* velocity (positive = stretch) */
  double   ldd; /* acceleration */
  double   t;   /* Tension Convention: Pull = positive */
} SL_muscle;

typedef struct { /* Muscle state */
  float   l;   /* length */
  float   ld;  /* velocity (positive = stretch) */
  float   ldd; /* acceleration */
  float   t;   /* Tension Convention: Pull = positive */
} SL_fmuscle;

typedef struct { /* Link parameters */
  double   m;              /* Mass */
  double   mcm[N_CART+1]; /* Center of mass multiplied with the mass */
  double   inertia[N_CART+1][N_CART+1];  /* Moment of inertia */
  double   vis;            /* viscous friction term */
} SL_link;

typedef struct { /* Link parameters */
```

```
  float   m;          /* Mass */
  float   mcm[N_CART+1]; /* Center of mass multiplied with the mass */
  float   inertia[N_CART+1][N_CART+1];  /* Moment of inertia */
  float   vis;        /* viscous friction term */
} SL_flink;

typedef struct { /* end-effector parameters */
  double   m;             /* Mass */
  double   mcm[N_CART+1]; /* mass times Center of mass */
  double   x[N_CART+1];   /* end position of endeffector in local coordinates*/
  double   a[N_CART+1];   /* orientation of the tool in Euler Angles (x-y-z) */
} SL_endeff;

typedef struct { /* end-effector parameters */
  float   m;             /* Mass */
  float   mcm[N_CART+1]; /* mass times Center of mass */
  float   x[N_CART+1];   /* end position of endeffector in local coordinates */
  float   a[N_CART+1];   /* orientation of the tool in Euler Angles (x-y-z) */
} SL_fendeff;

typedef struct { /* external forces */
  double   f[N_CART+1];   /* external forces */
  double   t[N_CART+1];   /* external torques */
} SL_uext;

#ifdef __cplusplus
extern "C" {
#endif

  /* variables shared by all SL programs */
  /* external variables */

  /* constants that define the simulation */
  extern const int    n_dofs;              /* number of degrees of freedom */
  extern const int    n_dofs_est;          /* number of DOFs to be estimated */
  extern const int    n_dofs_est_skip;     /* degrees of freedom to be skipped by estimation */
  extern const int    n_endeffs;           /* number of endeffectors */
  extern const int    n_links;             /* number of links in the robot */
  extern const int    n_misc_sensors;      /* number of miscelleneous sensors in the robot */
  extern const int    invdyn_servo_ratio;  /* divides base freq. to obtain invdyn servo freq.*/
  extern const char   vision_default_pp[]; /* default script for vision processing */
  extern const int    max_blobs;           /* maximal number of blobs in the vision system */
  extern const int    d2a_hwm;             /* which D/A hardware is used by motor servo */
  extern const int    d2a_hwt;             /* which D/A hardware is used by the task servo */
  extern const int    d2a_hwi;             /* which D/A hardware is used by the inv dyn servo */
  extern const int    d2a_hwv;             /* which D/A hardware is used by the vision servo */
  extern const int    d2a_bm;              /* which D/A board is used by motor servo */
  extern const int    d2a_bt;              /* which D/A board is used by the task servo */
  extern const int    d2a_bi;              /* which D/A board is used by the inv dyn servo */
  extern const int    d2a_bv;              /* which D/A board is used by the vision servo */
  extern const int    d2a_cm;              /* which D/A channel is used by motor servo */
  extern const int    d2a_ct;              /* which D/A channel is used by the task servo */
  extern const int    d2a_ci;              /* which D/A channel is used by the inv dyn servo */
  extern const int    d2a_cv;              /* which D/A channel is used by the vision servo */


  /* generic external variables */
  extern int           servo_base_rate;     /* base freq. of servos */
  extern int           task_servo_ratio;    /* divides base freq. to obtain task servo freq.*/
  extern char          joint_names[][20];
```

```
extern char          cart_names[][20];
extern char          misc_sensor_names[][20];
extern SL_DJstate    joint_default_state[];
extern SL_OJstate    joint_opt_state[];
extern SL_Jstate     joint_state[];
extern SL_DJstate    joint_des_state[];
extern double        joint_range[][3+1];
extern double        u_max[];
extern SL_endeff     endeff[];
extern SL_Jstate     joint_sim_state[];
extern SL_VisionBlob blobs[];
extern Matrix        J;
extern Matrix        dJdt;
extern Matrix        Jdes;
extern Matrix        Jcog;
extern Matrix        link_pos;
extern Matrix        link_pos_des;
extern Matrix        link_pos_sim;
extern Matrix        joint_cog_mpos;
extern Matrix        joint_cog_mpos_des;
extern Matrix        joint_cog_mpos_sim;
extern Matrix        joint_origin_pos;
extern Matrix        joint_origin_pos_des;
extern Matrix        joint_origin_pos_sim;
extern Matrix        joint_axis_pos;
extern Matrix        joint_axis_pos_des;
extern Matrix        joint_axis_pos_sim;
extern Matrix        Alink[];
extern Matrix        Alink_des[];
extern Matrix        Alink_sim[];
extern int           link2endeffmap[];
extern char          current_vision_pp[];
extern char          blob_names[][20];
extern Blob3D        raw_blobs[];
extern Blob2D        raw_blobs2D[][2+1];
extern char          link_names[][20];
extern SL_Cstate     cart_state[];
extern SL_quat       cart_orient[];
extern SL_Cstate     cart_des_state[];
extern SL_quat       cart_des_orient[];
extern SL_Cstate     cart_target_state[];
extern SL_quat       cart_target_orient[];
extern SL_link       links[];
extern int           whichDOFs[];
extern SL_Cstate     base_state;
extern SL_quat       base_orient;
extern SL_uext       uext[];
extern SL_uext       uext_sim[];
extern double        misc_sensor[];
extern double        misc_sim_sensor[];
extern SL_Cstate     cog;
extern double        gravity;


/* Function prototypes */
void  where(void);
void  where_des(void);
void  cwhere(void);
void  bwhere(int flag);
void  lwhere(void);
```

```
  int   check_range(SL_DJstate *des);
  int   init_commands(void);
  void  initContacts(void);



  void
  SL_ForwardDynamics(SL_Jstate *lstate,SL_Cstate *cbase,
                     SL_quat *obase, SL_uext *ux, SL_endeff *leff);
  void
  SL_InverseDynamics(SL_Jstate *cstate,SL_DJstate *state,SL_endeff *endeff);
  void
  SL_InverseDynamicsArt(SL_Jstate *cstate, SL_DJstate *lstate, SL_Cstate *cbase,
                        SL_quat *obase, SL_uext *ux, SL_endeff *leff);


  void SL_MuscleState(SL_Jstate * state,  /* Current state of structure:
                                             In:  th, thd, thdd */
                      SL_muscle * muscle /* Muscle state
                                            Out: l, ld, ldd */
                      );
  void SL_MuscleTorque(SL_Jstate * state,  /* Current state of structure:
                                              In:  th
                                              Out: u */
                       SL_muscle * muscle /* Muscle state
                                             In: T */
                       );

#ifdef __cplusplus
}
#endif

#endif  /* _SL_ */
```

11.2 The SL_user_common.h Header File

```
/*=========================================================================
  ========================================================================

  SL_user_common.h

  ========================================================================
  Remarks:

  common variables and functions shared by many SL modules. This file
  needs to be included in SL_user_common.c

  ========================================================================*/

#ifdef __cplusplus
extern "C" {
#endif

  /* important variable that define the robot */
  const int  n_dofs = N_DOFS;
  const int  n_dofs_est = N_DOFS_EST;
  const int  n_dofs_est_skip = N_DOFS_EST_SKIP;
  const int  n_links = N_LINKS;
  const int  n_endeffs = N_ENDEFFS;
  const int  n_cameras = N_CAMERAS;
  const int  max_blobs = MAX_BLOBS;
  const int  n_misc_sensors = N_MISC_SENSORS;
  const char vision_default_pp[]=VISION_DEFAULT_PP;
  const int  servo_base_rate = SERVO_BASE_RATE;
  const int  task_servo_ratio = TASK_SERVO_RATIO;

  const int  d2a_hwm = D2A_HWM;
  const int  d2a_hwv = D2A_HWV;
  const int  d2a_hwi = D2A_HWI;
  const int  d2a_hwt = D2A_HWT;

  const int  d2a_bm = D2A_BM;
  const int  d2a_bv = D2A_BV;
  const int  d2a_bi = D2A_BI;
  const int  d2a_bt = D2A_BT;

  const int  d2a_cm = D2A_CM;
  const int  d2a_cv = D2A_CV;
  const int  d2a_ci = D2A_CI;
  const int  d2a_ct = D2A_CT;

  /* global variables */
  SL_Jstate     joint_state[N_DOFS+1];          /* current states */
  SL_DJstate    joint_des_state[N_DOFS+1];      /* desired states */
  SL_endeff     endeff[N_ENDEFFS+1];            /* endeffector structure */
  SL_Jstate     joint_sim_state[N_DOFS+1];      /* the state of the sim. robot */
  SL_DJstate    joint_default_state[N_DOFS+1];  /* posture for startup */
  double        joint_range[N_DOFS+1][3+1];     /* various info on joint limits */
  double        u_max[N_DOFS+1];                /* actuator output limits */
  SL_VisionBlob blobs[MAX_BLOBS+1];             /* blob info from vision */
  Matrix        J;                              /* kinematic Jacobian */
  Matrix        dJdt;                           /* time derivative of Jacobian */
  Matrix        Jdes;                         /* Jacobian based on des. state */
```

```
  Matrix        Jcog;                          /* COG Jacobian (only positions) */
  Matrix        link_pos;                        /* Cart. pos of links */
  Matrix        link_pos_des;                    /* desired cart. pos of links */
  Matrix        link_pos_sim;                    /* simulated cart. pos of links */
  Matrix        joint_cog_mpos;                  /* vector of mass*COG of each joint */
  Matrix        joint_origin_pos;                /* vector of pos. of local joint coord.sys */
  Matrix        joint_axis_pos;                  /* unit vector of joint rotation axis */
  Matrix        joint_cog_mpos_des;              /* vector of mass*COG of each joint based on de-
sireds*/
  Matrix        joint_origin_pos_des;            /* vector of pos. of local joint coord.sys based
on des.*/
  Matrix        joint_axis_pos_des;              /* unit vector of joint rotation axis based on
des.*/
  Matrix        joint_cog_mpos_sim;              /* vector of mass*COG of each joint based on sim-
sate*/
  Matrix        joint_origin_pos_sim;            /* vector of pos. of local joint coord.sys based
on sim.*/
  Matrix        joint_axis_pos_sim;              /* unit vector of joint rotation axis based on
sim.*/
  Matrix        Alink[N_LINKS+1];                /* homogeneous transformation matrices for all
links */
  Matrix        Alink_des[N_LINKS+1];            /* homogeneous transformation matrices for all
links */
  Matrix        Alink_sim[N_LINKS+1];            /* homogeneous transformation matrices for all
links */
  SL_Cstate     cart_state[N_ENDEFFS+1];         /* endeffector state */
  SL_quat       cart_orient[N_ENDEFFS+1];        /* endeffector orientation */
  SL_Cstate     cart_des_state[N_ENDEFFS+1];     /* endeff.state based on des.state */
  SL_quat       cart_des_orient[N_ENDEFFS+1];    /* endeff.orient based on des.state */
  SL_Cstate     cart_target_state[N_ENDEFFS+1]; /* endeff.target state for inv.kin */
  SL_quat       cart_target_orient[N_ENDEFFS+1];/* endeff.target orient for inv. kin */
  char          current_vision_pp[30];           /* vision post processing specification */
  SL_OJstate    joint_opt_state[N_DOFS+1];       /* rest state for optimization */
  SL_link       links[N_DOFS+1];                 /* specs of links: mass, inertia, cm */
  Blob3D        raw_blobs[MAX_BLOBS+1];          /* raw blobs 3D of vision system */
  Blob2D        raw_blobs2D[MAX_BLOBS+1][2+1];   /* raw blobs 2D of vision system */
  int           whichDOFs[N_DOFS+1];             /* which DOFS does a servo compute motor commands
for */
  SL_Cstate     base_state;                      /* cartesian state of base coordinate system */
  SL_quat       base_orient;                     /* cartesian orientation of base coordinate sys-
tem */
  SL_uext       uext[N_DOFS+1];                  /* measured external forces on
                                                 every DOF in local coordinates */
  SL_uext       uext_sim[N_DOFS+1];              /* simulated external forces on
                                                 every DOF, i.e., due to contact
                                                 or simulated virtual forces, in
                                                 global coordinates*/
  double        misc_sensor[N_MISC_SENSORS+1];   /* additional sensors */
  double        misc_sim_sensor[N_MISC_SENSORS+1];  /* additional sensors, simulated */
  SL_Cstate     cog;                             /* center of gravity */

  double        gravity = G;

#ifdef __cplusplus
}
#endif
```

11.3 Example of a SL_user_common.c File

```
/*============================================================================
============================================================================

                              SL_user_common.c

============================================================================
Remarks:

      User file to declare common variables and functions shared by many
      SL modules. This is a good place to declare global variables

============================================================================*/

#include "SL.h"
#include "SL_user.h"

/* global variables */
char joint_names[][20]= {
  {"dummy"},
  {"R_SFE"},
  {"R_SAA"},
  {"R_HR"},
  {"R_EB"},
  {"R_WR"},
  {"R_WFE"},
  {"R_WAA"},

  {"R_TL"},
  {"R_TV"},
  {"R_FAA"}
};

char cart_names[][20]= {
  {"dummy"},
  {"R_HAND"},
};

char blob_names[][20]= {
  {"dummy"},
  {"BLOB1"},
  {"BLOB2"},
  {"BLOB3"},
  {"BLOB4"},
  {"BLOB5"},
  {"BLOB6"}
};

char link_names[][20]= {
  {"dummy"},
  {"SHOULDER"},
  {"SHOULDER_OFF"},
  {"ELBOW"},
  {"ELBOW_OFF"},
  {"WRIST"},
  {"WRIST_OFF"},
  {"HAND"},
  {"FINGER_JOINT"},
  {"FINGER"},
```

```
  {"THUMB_JOINT"},
  {"THUMB"}
};

/* initialization needs to be done for this mapping */
int           link2endeffmap[] = {0,HAND};


/* the following include must be the last line of the variable declaration section */
#include "SL_user_common.h"   /* do not erase!!! */


/*****************************************************************************
******************************************************************************
Function Name  : setDefaultEndeffector
Date           : June 1999

Remarks:

        assigns default values to the endeffector parameters

******************************************************************************
Paramters:  (i/o = input/output)

     none

*****************************************************************************/
void
setDefaultEndeffector(void) {

  int i;

  for (i=1; i<=N_ENDEFFS; ++i) {
    endeff[i].m       = 0.0;
    endeff[i].mcm[_X_]= 0.0;
    endeff[i].mcm[_Y_]= 0.0;
    endeff[i].mcm[_Z_]= 0.0;
    endeff[i].x[_X_]  = 0.0;
    endeff[i].x[_Y_]  = 0.085;
    endeff[i].x[_Z_]  = 0.0;
    endeff[i].a[_A_]  = 0.0;
    endeff[i].a[_B_]  = -PI/2.;
    endeff[i].a[_G_]  = 0.0;
  }

}
```

11.4 Example of a SL_user.h Header File

```
/*============================================================================

  SL_user.h

  by Stefan Schaal, Feb 1999
  ============================================================================
  Remarks:

  robot specific definitions

  ============================================================================*/

#ifndef _SL_user_
#define _SL_user_

/* dimensions of the robot */
#define l0  0.4
#define l1  0.0381
#define l2  0.287
#define l3  0.0025
#define l4  0.2781
#define l5  0.0081
#define l6x 0.03
#define l6y 0.1
#define l6z 0.02
#define l7z -0.05

#define l8  0.10        /* thumb length  */
#define l9  0.05        /* finger length */
#define l10 0.15        /* hand length   */

/* links of the robot */

enum RobotLinks {
  SHOULDER = 1,
  SHOULDER_OFF,
  ELBOW,
  ELBOW_OFF,
  WRIST,
  WRIST_OFF,
  HAND,
  FINGER_JOINT,
  FINGER,
  THUMB_JOINT,
  THUMB
};

/* endeffector information */

#define RIGHT_HAND   1

/* vision variables */

#define CAMERA_1  1
#define CAMERA_2  2

#define BLOB1 1
```

```
#define BLOB2 2
#define BLOB3 3
#define BLOB4 4
#define BLOB5 5
#define BLOB6 6


/* define the DOFs of this robot */

enum RobotDOFs {
  R_SFE = 1,
  R_SAA,
  R_HR,
  R_EB,
  R_WR,
  R_WFE,
  R_WAA,
  R_TL,
  R_TV,
  R_FAA
};

/* number of degrees-of-freedom of robot */
#define N_DOFS  R_FAA

/* N_DOFS + fake DOFS, needed for parameter estimation;
   fake DOFS come from creating endeffector information */
#define N_DOFS_EST 13

/* N_DOFS to be excluded from parameter estimation (e.g., eye joints);
   these DOFS must be the last DOFS in the arrays */
#define N_DOFS_EST_SKIP 3

/* number of links of the robot */
#define N_LINKS THUMB

/* number of miscellaneous sensors */
#define N_MISC_SENSORS 0

/* number of endeffectors */
#define N_ENDEFFS RIGHT_HAND

/* number of cameras used */
#define N_CAMERAS 2

/* number of blobs that can be tracked by vision system */
#define MAX_BLOBS 6

/* vision default post processing */
#define VISION_DEFAULT_PP "vision_default.pp"

/* the servo rate used by the I/O with robot: this limits the
   servo rates of all other servos */
#define  SERVO_BASE_RATE 960

/* divisor to obtain task servo rate (task servo can run slower than
   base rate, but only in integer fractions */
#define  TASK_SERVO_RATIO   R1TO2

/* settings for D/A debugging information -- see dta.h */
```

```
#include  "dta.h"
#define   D2A_HWM     DT1401
#define   D2A_HWT     DT1401
#define   D2A_HWV     0
#define   D2A_HWI     0

#define   D2A_BM      1
#define   D2A_BT      1
#define   D2A_BV      0
#define   D2A_BI      0

#define   D2A_CM      1
#define   D2A_CT      2
#define   D2A_CV      0
#define   D2A_CI      0

#ifdef __cplusplus
extern "C" {
#endif

  /* function prototypes */

#ifdef __cplusplus
}
#endif


#endif  /* _SL_user_ */
```

11.5 The utility.h Header file

```
Error! Cannot open file.
```

11.6 Example of a SL_user.c File

```
/*============================================================================

  SL_user.c

  by Stefan Schaal, Jan 1999
  ============================================================================
  Remarks:

  This program contains all user functions for a particular simulation.

  ============================================================================*/

/* global headers */

#include "stdio.h"
#include "math.h"
#include "stdlib.h"
#include "string.h"
#include "strings.h"

/* openGL headers */
#ifdef alpha
#undef alpha
#endif
#ifdef powerpc
#include <GLUT/glut.h>
#else
#include "GL/glut.h"
#endif

/* user specific headers */
#include "utility.h"
#include "SL.h"
#include "SL_integrate.h"
#include "SL_openGL.h"
#include "SL_user.h"
#include "mdefs.h"
#include "motor_servo.h"
#include "task_servo.h"
#include "vision_servo.h"
#include "invdyn_servo.h"
#include "tasks.h"
#include "dynamics.h"
#include "controller.h"
#include "vx_fake.h"
#include "collect_data.h"
#include "SL_common.h"
#include "shared_memory.h"
#include "kinematics.h"

/* global variables */
int print_Hmat = FALSE;

/* local variables */
static void   (*user_display_function)  (void) = NULL;

/* local functions */
```

```
int
createWindows(void);
static void
myDrawGLElement(int num, double length, int flag);
void
display(void);
void
idle(void);
void
addUserDisplayFunc(void (*fptr)(void));


/* external functions */
extern void initUserTasks(void);


/*****************************************************************************
 *****************************************************************************
 Function Name : initSimulation
 Date          : July 1998

 Remarks:

 initializes everything needed for the simulation.

 *****************************************************************************
Paramters:  (i/o = input/output)

 argc (i) : number of elements in argv
 argv (i) : array of argc character strings


*****************************************************************************/
int
initSimulation(int argc, char** argv)

{

  int i,j,n;
  int rc;
  int ans;

  /* initialize the servos */
  init_task_servo();  /* the first servo sets the sampling
                         rate in collect data */
  read_whichDOFs("task_sim_servo");
  init_motor_servo();
  read_whichDOFs("motor_sim_servo");
  init_vision_servo();
  read_whichDOFs("vision_sim_servo");
  init_invdyn_servo();
  read_whichDOFs("invdyn_sim_servo");

  /* we need the dynamics initalized */
  init_dynamics();

  /* user specific tasks */
  initUserTasks();

  /* create simulation windows */
```

```
  if (!createWindows())
    return FALSE;

  /* reset motor_servo variables */
  servo_enabled            = 1;
  motor_servo_calls        = 0;
  servo_time               = 0;
  motor_servo_rate         = SERVO_BASE_RATE;
  zero_integrator();
  bzero((char *)&(joint_sim_state[1]),sizeof(SL_DJstate)*N_DOFS);
  for (i=1; i<=N_DOFS; ++i) {
    joint_sim_state[i].th = joint_default_state[i].th;
    joint_des_state[i].th = joint_default_state[i].th;
  }

  /* reset task_servo variables */
  servo_enabled           = 1;
  task_servo_calls        = 0;
  task_servo_time         = 0;
  task_servo_errors       = 0;
  task_servo_rate         = SERVO_BASE_RATE/(double) task_servo_ratio;
  setTaskByName(NO_TASK);
  setDefaultPosture();
  changeCollectFreq(task_servo_rate);

  /* reset vision servo variables */
  servo_enabled           = 1;
  vision_servo_calls      = 0;
  vision_servo_time       = 0;
  vision_servo_errors     = 0;
  vision_servo_rate       = VISION_SERVO_RATE;

  /* initialize all vision variables to safe values */
  init_vision_states();
  init_learning();

  /* reset the invdyn servo variables */
  servo_enabled            = 1;
  invdyn_servo_errors      = 0;
  invdyn_lookup_data       = 0;
  invdyn_learning_data     = 0;

  /* initalize objects in the environment */
  readObjects();

  /* assign contact force mappings */
#include "LEKin_contact.h"
  initContacts();

  return TRUE;
}

/*****************************************************************************
 *****************************************************************************
 Function Name : createWindows
 Date          : July 1998

 Remarks:

 initializes graphic windows
```

```
 ***************************************************************************
Paramters:  (i/o = input/output)

 none


***************************************************************************/
int
createWindows(void)

{

  int i;
  int width=400, height=400;
  OpenGLWPtr w;

  /* get a window structure, initialized with default values */
  w=getOpenGLWindow();
  if (w==NULL)
    return FALSE;

  for (i=1; i<=N_CART; ++i) {
    w->eye[i] /= 1.3;
    w->eye0[i] = w->eye[i];
  }
  w->display = display;
  w->idle    = idle;
  w->width   = width;
  w->height  = height;
  if (!createWindow(w))
    return FALSE;

  return TRUE;

}

/***************************************************************************
****************************************************************************
Function Name  : display
Date           : August 7, 1992

Remarks:

        this function updates the OpenGL graphics

****************************************************************************
Paramters:  (i/o = input/output)


***************************************************************************/
void
display(void)

{
  int i;
  static SL_Jstate *state = joint_sim_state;
  static SL_endeff *eff = endeff;
  GLfloat  objscolor[4]={(float)0.,(float)0.,(float)0.,(float)0.0};
  double fscale = 0.01;
```

```
#include "OpenGL.h"

  glColor4fv(objscolor);
  drawObjects();
  drawContacts(fscale);

  if (user_display_function != NULL)
    (*user_display_function)();

  /* glutSwapBuffers(); */

}

/******************************************************************************
******************************************************************************
Function Name  : addUserDisplayFunc
Date           : August 7, 1992

Remarks:

        assigns an additional display function to the openGL updates

******************************************************************************
Paramters:  (i/o = input/output)

   fptr (i): function pointer


******************************************************************************/
void
addUserDisplayFunc(void (*fptr)(void))
{
  user_display_function = fptr;
}

/******************************************************************************
******************************************************************************
Function Name  : myDrawGLElement
Date           : August 7, 1992

Remarks:

        draws a GL element of a particular length in z direction

******************************************************************************
Paramters:  (i/o = input/output)

        num   (i): ID number of element (just for info)
        length(i): length of element
        flag  (i): draw joint element (yes, not)


******************************************************************************/
static void
myDrawGLElement(int num, double length, int flag)

{

  double width=0.03;
```

```
double thumb_width = 0.02;
double finger_width = 0.02;
double hand_width = 0.07;
GLfloat  gray[4]={(float)0.8,(float)0.8,(float)0.8,(float)1.0};
GLfloat  green[4]={(float)0.1,(float)0.5,(float)0.5,(float)1.0};
GLfloat  red[4]={(float)1.0,(float)0.25,(float)0.25,(float)1.0};
GLfloat  blue[4]={(float)0.1,(float)0.1,(float)1.0,(float)1.0};
GLfloat  black[4]={(float)0.,(float)0.,(float)0.0,(float)1.0};
GLfloat  white[4]={(float)1.,(float)1.,(float)1.0,(float)1.0};
GLfloat  yellow[4]={(float)1.0,(float)0.8,(float)0.7,(float)1.0};

if (flag==1) {
  glTranslated(0.0,0.0,length);

  if (num==999) {/* the base coordinate system */
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, blue);
    glColor4fv(blue);
  } else if (num == 8 || num == 9 || num ==10) {
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, white);
    glColor4fv(white);
  } else {
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, green);
    glColor4fv(green);
  }

  if (num == 8 || num == 9 || num == 10) {
    if (solid)
      glutSolidSphere(0.5*width,10,10);
    else
      glutWireSphere(0.5*width,10,10);
  } else {
    if (solid)
      glutSolidSphere(1.2*width,5,5);
    else
      glutWireSphere(1.2*width,5,5);
  }
  glTranslated(0.0,0.0,-length);
}


switch (num) {

case 108: /* the hand */

  glScaled(hand_width,hand_width/3.,length);
  glTranslated(0.0,0.0,0.5);
  glColor4fv(yellow);
  glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, yellow);
  if (solid)
    glutSolidSphere(0.5,8,8);
  else
    glutWireSphere(0.5,8,8);
  break;

case 109: /* the thumb */

  glScaled(thumb_width,thumb_width,length);
  glTranslated(0.0,0.0,0.5);
  glColor4fv(yellow);
  glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, yellow);
```

```
  if (solid)
    glutSolidSphere(0.5,8,8);
  else
    glutWireSphere(0.5,8,8);
  break;

case 110: /* the finger */

  glScaled(finger_width,finger_width,length);
  glTranslated(0.0,0.0,0.5);
  glColor4fv(yellow);
  glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, yellow);
  if (solid)
    glutSolidSphere(0.5,8,8);
  else
    glutWireSphere(0.5,8,8);
  break;

case 8: /* the finger */

  glScaled(thumb_width*1.5,thumb_width*1.5,length);
  glTranslated(0.0,0.0,0.5);
  glColor4fv(gray);
  glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, gray);
  if (solid)
    glutSolidSphere(0.5,8,8);
  else
    glutWireSphere(0.5,8,8);
  break;

case 9: /* the finger */

  glScaled(thumb_width*1.5,thumb_width*1.5,length);
  glTranslated(0.0,0.0,0.5);
  glColor4fv(gray);
  glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, gray);
  if (solid)
    glutSolidSphere(0.5,8,8);
  else
    glutWireSphere(0.5,8,8);
  break;

case 10: /* the finger */

  glScaled(thumb_width*1.5,thumb_width*1.5,length);
  glTranslated(0.0,0.0,0.5);
  glColor4fv(gray);
  glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, gray);
  if (solid)
    glutSolidSphere(0.5,8,8);
  else
    glutWireSphere(0.5,8,8);
  break;

default:

  glScaled(width,width,length);
  glTranslated(0.0,0.0,0.5);
  glColor4fv(gray);
  glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, gray);
```

```
    if (solid)
      glutSolidCube(1.0);
    else
      glutWireCube(1.0);

  }
}

/******************************************************************************
*******************************************************************************
Function Name  : idle
Date           : June 1999

Remarks:

      The function called by openGL whenever processing time available.
      Here, the servo-loops are implemented

*******************************************************************************
Paramters:  (i/o = input/output)


******************************************************************************/
void
idle(void)

{
  int i;
  static double last_draw_time=0;
  int    integrate_method = INTEGRATE_EULER;
  double n_integration    = 1;

  // the core component of the idle task is conveniently included
#include "SL_user_idle_core.h"

  // the numerical integration
#include "SL_user_integrate.h"

  /* refresh display */
  if (servo_time-last_draw_time > window_update_rate) {
    glutPostRedisplayAll();
    last_draw_time = servo_time;
  }

  /* check the keyboard interaction */
  checkKeyboard();
}
```

12 C-Code References