The Creature Library Tutorial (1040 Long Form)

Robert Ringrose

July 18, 1995

Copyright (C) 1992, MIT Leg Lab
All rights reserved

This document is intended as instructions on the basic use of the Creature Library, a library of C subroutines for the creation of physical simulations. There is an appendix which describes how to use the resulting simulation. I assume a working knowledge of the C programming language.

It does not describe the use of Legplot or Anim.

1 Background

In the Leg Lab, we study actively balancing dynamically stable legged robots (robots that keep their balance as they run, walk, or do acrobatic manouvers). Generally, we simulate the robots on a computer to test the control algorithms before trying them on a physical robot. We also find that these simulations are useful for controlling physically realistic computer animation. All of our simulations use rigid-body dynamics.

The Creature Library was designed to solve several problems which came up as the simulation process evolved.

The first simulations in the Leg Lab were coded directly. Later we got a package called SD/Fast[1] which would automatically write the equations of motion for a physical system, and we implemented an interface which allowed adjustment of variables while the simulation was running.

At the same time, we were developing legplot, a tool which allowed further analysis of the simulation results, and anim, another tool which allowed us to combine simulation results into three-dimensional animated scenes.

The Creature Library integrates the use of all three of these tools (SD/Fast, legplot, and anim) in a way which helps prevent inconsistencies and makes it easier to create new creatures.

2 How It Works, In Brief

You write a C program which, using Creature Library subroutines, describes a creature. Once this is compiled and linked into the library, you can run it to write the entire simulation except the control system (for our machines, this is a finite state controller). Makefiles are provided which will compile, link, and run your C program, and then compile the resulting simulation.

The resulting simulation includes:

- servomechanisms in the joints (can be suppressed)
- user-definable external forces at specified points
- ground contact at specified points
- integrator and physical simulator
- code which can be included into the animator to display the creature

[1] SD/Fast is a product of Symbolic Dynamics, inc.

- code which can be included into legplot to display a "cartoon" of the creature
- user interface
- hooks for user-defined ground forces, external forces, servomechanisms
- locations for the control system and its variables.
- user-modifiable code for initialization

The simulation is missing:

- the control algorithm
- stable initial parameters

3 Motivation

Initially, new simulations were created by copying other simulations and modifying them (adding, altering, or removing links and joints). The problem with this approach is that you never know if you made all the appropriate changes everywhere.

One of the goals of the Creature Library is that everything which defines the simulation should end up in one place. For the structure of the creature, this place is in the C program which includes the library. If you change something in that program, you know the changes will be propogated.

Additionally, you know the generated code has been tested and is not likely to be causing errors. The generated code is more uniform than the code which results when every simulation is created individually, so a person who has worked extensively on a simulation can look at someone else's simulation and rapidly figure out what is going on.

Finally, it is easier to modify the creature (longer legs, different mass properties) when the description is in a single fairly small file.

4 Constructing A Creature

There are two phases in the creation of a creature. First you have the basic structure, including the links, joints, and connections between them. Then you have the actual shapes, masses, and moments of inertia for the different links.

I generally create something with the correct basic structure but blocks for the shapes, and once I have verified with the animator that all the blocks are where I think they should be, joints point the right direction, and joints are spaced properly, I make more complex link shapes.

Here is a sample C program which creates a creature called "test".

```
/* create_test.c */
#include <cl_lib.h>

main(argc, argv)
  int argc;
  char *argv[];
{
  command_line(argc, argv);

  begin_species("test");
  new_link("link1");
  begin_shape();
  translate(0.0, 0.0, -1.0);
  shape(SBRICK, 0.5, 0.45, 1.0);
```

```
  end_shape();

  joint_pin("tjoint", 'z');
  new_link("link2");
  begin_shape();
  shape(SBRICK, 0.45, 0.5, 1.0);
  end_shape();
  end_species();
}
```

This program is a full-fledged C program and can call any of the standard C library routines, include other files, and have its own subroutines. For example, if you are creating a quadruped you could have a create_leg subroutine (which might be passed the leg number) which would add a leg.

5 Calling Conventions

Let's go through the sample code and show how it evolved:

First, the header file. Without the #include, you can't access the library routines.

```
/* create_test.c */
#include <cl_lib.h>
```

The include file, /home/ll/include/cl_lib.h, contains extern declarations for each of the functions you can use from the Creature Library. If you are unsure of the parameters which should go to a function, or of the options available (for example, the different kinds of joints or shapes), this file has them in gory detail.

It is good to tell the compiler where to start, so add main.

```
main()
{
}
```

The library has some initialization it needs to do, so tell it you're starting a new species and let it know when you're finished (be aware it does an exit(0) at the end of end_species, so code after it won't be executed).

```
main()
{
  begin_species("test");
  end_species();
}
```

We'd like to be able to use command-line options, but the only way for the library to see them is to actually pass them to the library ourselves with the command_line procedure. Calling command_line is optional, but allows the library to check the species name against the creating filename and permits you to specify which files you want created (discussed in more detail later).

```
main(argc, argv)
  int argc;
  char *argv[];
{
  command_line(argc, argv);
  begin_species("test");
  end_species();
}
```

The structure of the test object is a solid mass connected by a pin joint to another solid mass. We will need two links and a joint. The library assumes each joint is attached to the last link you created, and each link is attached to the last joint you created; to do anything else, see the section on PARENTING. The links and joints need different names so they can be referred to unambiguously. We can set up the basic structure with

```
begin_species("test");
new_link("link1");
new_joint("tjoint");
new_link("link2");
end_species();
```

It doesn't know what kind of joint tjoint is. We want it to be a pin joint, rotating about the "z" (vertical) axis, so we would add set_pin_joint('z') after new_joint("tjoint"). This pair of routines (new_joint and set_joint) come together so often that there is an alternate routine which does both of them in order. So, the link-joint structure becomes

```
begin_species("test");
new_link("link1");
joint_pin("tjoint", 'z');
new_link("link2");
end_species();
```

We have completed the link-joint structure. We could create the simulation now, except that we haven't given any geometry to the links. This means that they will have no size and no mass, and the results will be quite boring (or interesting, depending on your point of view). Actually, SD/Fast won't allow you to have a link with no mass or moment of inertia at the end of a link chain so it won't write the dynamics.

To give it some substance, we need to specify shapes for the link. Normally, I write a C subroutine for each shape and call them as necessary. I do this because the shapes get complex enough that they clutter the creature's structure. To tell it you are defining a shape, add begin_shape and end_shape after the link.

```
new_link("link2");
begin_shape();
end_shape();
end_species();
```

None of the identity, rotate, translate, or shape calls within a begin_shape() end_shape() pair will affect the positioning of the joints. This is so that you can change how something looks without risking alteration of the underlying structure. The process of generating specific shapes with translate() and shape() is fairly complicated and will be covered in GRAPHICS and ORIGINS. For this example, we'll present the code to create a pair of bricks without explaination.

```
begin_species("test");
new_link("link1");
begin_shape();
translate(0.0, 0.0, -1.0);
shape(SBRICK, 0.5, 0.45, 1.0);
end_shape();
joint_pin("tjoint", 'z');
new_link("link2");
begin_shape();
shape(SBRICK, 0.45, 0.5, 1.0);
end_shape();
end_species();
```

6 One-Time Setup

Now that you have a create_test.c file in an otherwise empty directory, check your .login and .rhosts files to make sure it will run. There are a few environment variables which need to be set.

The common Leg Lab login file (/home/ll/.login) will set these environment variables. If you have access to /home/ll, I suggest simply putting "source /home/ll/.login" in your ~/.login file so that future changes will automatically take effect.

If you don't have access to /home/ll/.login, or don't want to source it, edit your ~/.login file so that it includes the following lines:

```
setenv SDKEYDIR /home/ll/bin/sun4/sd_examples_BX21
```

(SDKEYDIR may change with other versions of SD/Fast)

```
setenv SIM_SOURCE_FILES /home/ll/sim_source_files/
setenv MATERIALS_DATA /home/ll/anim/gx_iris/material.h
set path = ($path /home/ll/bin)
```

Save your ~/.login file and either source it or log off and on again.

You will also need a ~/.rhosts file. It should have a line for each machine you are likely to run create_test from with the machine name and your user ID. Alternately, this file can have lists of hosts (netgroups). For example, my .rhosts file has

```
+@aihosts ringrose
```

which allows me to freely rlogin between all the AI lab machines. Additionally, the .rhosts file must be owned by you (chown YOUR-ID ~/.rhosts) and should only be writable by you (chmod 0644 ~/.rhosts). Having your current machine in your .rhosts file will enable you to log into other machines without giving your password. The Creature Library needs this ability to run SD-Fast, since SD-Fast will only run on tibia.ai.mit.edu.

The changes to your ~/.login and ~/.rhosts files need only be done once.

7 Creating The Simulation

Go to the directory with create_test.c and type

```
setup_simulation test
```

(replace "test" with the creature name if you're making something else). This has to be done the first time you create a simulation. If you modify or correct it, as long as you don't change the species name, you will not need to run setup_simulation again. Setup_simulation will fail (and tell you so) if you don't have a .rhosts file or do not have the environment variables set.

Finally, type

```
make
```

to create the simulation. If you just did a setup_simulation, there will be a stub makefile which will compile and run create_test so that it has the "real" makefile, and then it will do another make. If this stub makefile fails (because of an error in your code or otherwise), you should do another setup-simulation.

8 In Summary

- If you have never used CL before, you need to modify your .login and .rhosts files.
- If this is a new creature, the species name has been altered, or the compile failed right after running setup_simulation, you need to run setup_simulation.
- If the creature has been modified, or setup_simulation has just been run, or a compile failed and the problem has been corrected, you need to run make.

9 Coordinate Systems

The Creature Language uses a right-handed coordinate system, where +x and +y are parallel to the ground and positive z is "up". Generally, positive x is also considered "forward". Gravity is, by default, in the negative z direction at $9.81m/s^2$. Positive rotations are defined by the right-hand rule. Thus, a positive x rotation rotates from y to z around the origin, a positive y rotation rotates from z to x, and a positive z rotation rotates from x to y.

10 Parenting (Attaching Things)

Every creature has a "base" link, which is attached to the ground. Since loops of links aren't implemented, each link has a joint closer to the base link (or, in the case of the base link, the ground) and each joint has a link closer to the base link. That joint/link closer to the base link is the parent.

To determine a link's parent: Normally, a link's parent is the joint created most recently. A call, before the next link or joint is begun, to

```
set_parent("jointname");
```

will change this. Passing GROUND to set_parent will fix the link to the ground, useful for bases of systems attached to the ground. Passing FREE will create a new six-degree-of-freedom joint attached to the ground.

These six-degree-of-freedom joints, effectively three sliders (x, y, z) and three pins (yaw, pitch, roll), is normally used to measure position and orientation with respect to the ground. Unless you directly apply a force to the joint, the joint will not affect the motion of your simulated object.

In the case, as with link1 in "test" earlier, of no recently-created joint, it is treated as if you had called set_parent(FREE).

To determine a joint's parent: Normally, a joint's parent is the link created most recently (regardless of joints created previously). A call, before the next link or joint is begun, to

```
set_parent("linkname");
```

will change this. Passing GROUND to set_parent will attach the joint to the ground. If there is no previously created link, it is treated as if you had called set_parent(GROUND). It is an error to set a joint's parent to FREE.

A more complicated link-joint structure requiring set_parent calls is a biped with telescoping legs (figure 10). Calls to routines for offsets and shapes are left out to emphasize the link-joint structure, since they don't affect it.

```
new_link("trunk");
joint_ball("hip1", "hip1_yaw", "hip1_roll",
           "hip1_pitch");
new_link("u_leg1");  /* upper leg */
joint_slider("leg1", 'z');
new_link("l_leg1");  /* lower leg */
joint_ball("hip2", "hip2_yaw", "hip2_roll",
           "hip2_pitch");
set_parent("trunk"); /* Attach hip2 to the trunk */
new_link("u_leg2");
joint_slider("leg2", 'z');
new_link("l_leg2");
```

So, with the exception of the hip2 ball joint, everything is attached to the joint/link above it.

The same tree structure can be created with

```
new_link("trunk");
joint_ball("hip2", "hip2_yaw", "hip2_roll",
           "hip2_pitch");
```

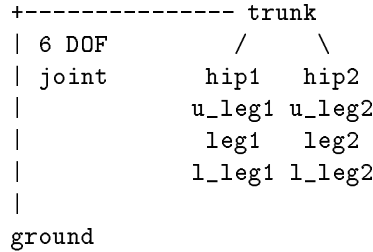

Figure 1: Connectivity structure for sample two-legged creature

```
joint_ball("hip1", "hip1_yaw", "hip1_roll",
           "hip1_pitch");
new_link("u_leg1");
joint_slider("leg1", 'z');
new_link("l_leg1");
new_link("u_leg2");
set_parent("hip2");  /* Attach u_leg2 to hip2 */
joint_slider("leg2", 'z');
new_link("l_leg2");
```

The possible joints in the Creature Library are the same as those in SD/Fast. Although most of them can be done as combinations of pins and sliders. For details on using more complex joints, see section 15

- joint_pin: rotation about an axis. For example, a simple hip.
- joint_slider: movement up and down an axis. Length zero means that the point where the slider is attached to the parent and the point where the slider is attached to the child are the same in world space. Any telescoping leg.
- joint_cylinder: movement up and down and rotation about the same axis.
- joint_universal: Two pin joints, at right angles to each other.
- joint_planar: Two sliders, at right angles, and a pin joint in the third axis. Confines motion to a plane.
- joint_gimbal: Three pin joints, each at right angles to the previous.
- joint_ball: Like a gimbal, but without gimbal lock. Unlike pin joints, which if they go through several complete rotations can have angles greater than PI, a ball joint's angles will wrap from +PI to -PI. WARNING: velocities (qd) in and torques applied to a ball joint are given in body-fixed coordinates. The torques should be in the CHILD's coordinates.
- joint_6dof: Three sliders, then a ball joint. Allows complete freedom of movement between the two links (or the link and the ground). WARNING: velocities (qd) in and torques applied to a ball joint are given in body-fixed coordinates.

11 Origins

Links will have their center of mass determined by the shape and density of the graphic representation, or you can call

```
set_com_offset(x, y, z);
```

to set the location of the center of mass relative to the parent joint. This will not affect the drawing of the link, only the properties passed to SD/Fast for use in calculating the dynamics.

Joints, by default, are at the same place as the parent link's parent joint. However, they can moved with either

```
set_joint_offset(x, y, z)
```

which places the joint relative to the parent link's parent joint, or

```
set_joint_offset_com(x, y, z)
```

which places the joint relative to the parent link's center of mass (either calculated or given explicitly with set_com_offset). In the previous biped example, to spread the hips out 0.1 m in the y direction, you would use

```
new_link("trunk");
joint_ball("hip1", "hip1_yaw", "hip1_roll",
           "hip1_pitch");
set_joint_offset(0.0, 0.1, 0.0);
new_link("u_leg1");
joint_slider("leg1", 'z');
new_link("l_leg1");
joint_ball("hip2", "hip2_yaw", "hip2_roll",
           "hip2_pitch");
set_joint_offset(0.0, -0.1, 0.0);
set_parent("trunk");
new_link("u_leg2");
joint_slider("leg2", 'z');
new_link("l_leg2");
```

or you could use

```
new_link("trunk");
joint_ball("hip2", "hip2_yaw", "hip2_roll",
           "hip2_pitch");
set_joint_offset(0.0, -0.1, 0.0);
joint_ball("hip1", "hip1_yaw", "hip1_roll",
           "hip1_pitch");
set_joint_offset(0.0, 0.1, 0.0);
new_link("u_leg1");
joint_slider("leg1", 'z');
new_link("l_leg1");
new_link("u_leg2");
set_parent("hip2");
joint_slider("leg2", 'z');
new_link("l_leg2");
```

Note that nothing has been done to translate the lower legs down. Frequently, with sliding legs, we don't translate the lower legs and then set up the geometry so the slider value corresponds to how far the leg is extended. But this is simply a convention, not something required or enforced by the library.

12 Graphics

The basic graphics routine is shape(). All shapes are created pointing in the positive z axis (up), with the center of the base at 0, 0, 0 link co-ordinates. The origin in link co-ordinates can be moved, moving the link and all links and joints attached to it, by using set_joint_offset on the link's parent joint.

You begin a graphical description of a link with

```
begin_shape();
```

Once the graphical description is begun, you can add shapes to the link by listing the translations and rotations to be applied (calls to translate() and rotate() routines) and then calling shape() with appropriate parameters (the possible shapes are listed further on in this section). These calls will not affect the shapes or positions of other links. If you want more than one shape associated with a link, simply continue the list of translations and rotations and then call shape() again. If you want to add another shape but not make it subject to the previous translations and rotations, simply call identity(). This routine cancels (for all shapes following it) the effects of previous calls to translate() and rotate().

Finally, end the graphical description of the link with a call to

```
end_shape();
```

use_color() will change the color of all shapes created after it. Colors are declared in the file /home/ll/anim/gx_iris/material.h as arrays; simply pass the array name as a string to use_color(). For example,

```
use_color("aluminum_material");
```

will make all shapes after the use_color have the color declared in the array aluminum_material. The default color is white. **Warning:** the case is important. If you do not use the same case as in material.h, the animator will get a bus error when it loads your creature.

To figure out where a shape really is and its orientation:

Start at the call to shape(). Find the previous identity() or begin_shape() call. Place the shape with the center of the base at 0, 0, 0 (link co-ordinates) pointing in the positive z direction. Go through the list of calls, applying the rotation (about 0,0,0) or the translation to the shape.

As an example, take the code

```
new_link("link2");
begin_shape();
translate(0.0, 0.0, 0.5);
rotate('y', 90.0);
shape(SBRICK, 0.45, 0.5, 1.0);
end_shape();
```

and see where the brick ends up.

```
 _
| |     1) place brick (width 0.45m, depth 0.5m,
| |        height 1m)
|.|     2) go to begin_shape call


 _
| |     3) translate the brick (0.5m in the +z
| |        direction, up)
|_|
 .
   ___ 4) rotate the brick 90 degrees about the
 . |   |   y axis to get final position.
   +---+   (Note that the y axis points into the page)
```

The density of the polygons displayed by the animator can be altered with a call to the routine use_polygon_density(density) where the density is some number greater than zero. Density 1.0 is "standard" — you might wish to increase the density for very large shapes or shapes where you have aliasing problems, and you might wish to decrease the density for unimportant shapes. Use_default_polygon_density returns to the default. The polygon density attribute propogates in the same manner as the physical density attribute (see *Groups* and *Mass Properties*).

The different shapes:

- shape(SBRICK, x_length, y_length, z_length); A brick of the given lengths in x, y, and z. 0,0,0 is the center of the base of the brick, not the center of mass of the brick.
- shape(SCYLINDER, height, radius); A cylinder with the center of the base at 0,0,0 of the given height and radius.
- shape(SCONE, height, base_radius); A cone, with the center of the base at 0,0,0 and the given height and base radius.
- shape(SFCONE, height, base_radius, top_radius); A trucated cone, with the center of the base at 0,0,0 and the given radii at the base and top.
- shape(SELL_FCONE, height, base_x, tip_x, y_x_ratio); Equivalent to shape(SGEN_FCONE, height, base_x, base_x*y_x_ratio, tip_x, tip_x*y_x_ratio); The horizontal cross-section is an ellipse.
- shape(SGEN_FCONE, height, base_x, base_y, tip_x, tip_y); A truncated cone with the center of the base at 0,0,0 and the given radii in the x and y directions at the base and top. The horizontal cross-section is an ellipse.
- shape(SGEN_FPYRAMID, height, base_x, base_y, tip_x, tip_y); A four-sided pyramid with the given widths. 0,0,0 is the middle of the bottom side.
- shape(SSPHERE, radius); Create a complete sphere of the given radius, with the center at 0,0,0.
- shape(SELLIPSOID, x_radius, y_radius, z_radius); Create a sphere where the different axis have different radii.
- shape(SHEMISPHERE, radius); Create a hemisphere of the given radius. The cross-section in the xz and yz planes will be a half circle, while the cross-section in the xy plane will be a full circle.
- shape(SHEMIELLIPSOID, x_radius, y_radius, z_radius); As hemisphere, except the radii are different so the cross-sections become ellipsoid.
- shape(SARC_TORUS, angle_start, angle_end, major_radius, minor_radius); Create a section of a torus. Angle_start and angle_end are measured in degrees in the xy plane, with zero degrees being the x axis. The center (major radius) is at 0,0,0. There are no endcaps.
- shape(SARC_HOOP, angle_start, angle_end, major_radius, minor_radius); Not currently functional
- shape(SBBRICK, x_length, y_length, z_length, bevel_width); A brick of the given lengths in x, y, and z with bevelled corners. 0,0,0 is the center of the base of the brick, not the center of mass of the brick.
- shape(STUBE, height, radius, inner_radius); A tube with the given height, outer radius, and inner radius. 0,0,0 is the center of the base of the tube, which extends vertically.
- shape(SGEN_TUBE, height, base_x, base_y, tip_x, tip_y, inner_base_x, inner_base_y, inner_tip_x, inner_tipy); A generalized tube, where the inner and outer sections are ellipses with the given radii. 0,0,0 is the center of the base of the tube, which extends vertically.
- shape(SGEN_COLUMN, height, num_points, positions); *height is a double, num_points is an integer, and positions is a pointer to a list of doubles (x and y positions of points).* A generalized column, where the cross-section is specified by an array of x and y positions relative to 0,0,0 and with the given vertical height.
- shape(SCOMPLEX, filename, scalex, scaley, scalez, right_handed); *filename is a string, and right_handed is an integer 1 or 0.* Any complex shape, the structure of which is taken from the given file. For consistent functioning, the filename should be absolute (start with a "/"). The shape is scaled in x, y, and z before any calculations. Right_handed should be 1 if the exterior polygons of the shape are right-handed, and 0 if they are left-handed. The shape should not be self-intersecting, should not have bow-tie polygons, and should be closed, or the mass properties will not be accurate.

13 Group

A graphical object and its associated mass properties can be separated from a link and used like a shape with the begin_group, end_group, and group procedures.

```
begin_group("groupname");
```

... description of a group of shapes, exactly as within a begin_shape() end_shape() pair

```
end_group();
```

From there on, you can treat this "group" in the same way you would any single shape, except that you add it with

```
group("groupname");
```

rather than a call to shape().

By placing calls to set_moi(), set_moment_of_inertia(), set_com_offset(), or set_mass() within the begin_group() end_group() pair, you can set the various mass properties of a group (see the descriptions of set_moi, set_com, and set_mass later). When you later use the group within a link, the given mass properties will be included in the calculations for that link (if you don't provide mass properties, they will be calculated for you from the graphic description). This means that you can approximate the mass properties of a complicated shape without having to calculate the mass properties of an entire link yourself, since when you put a group within a link it will use the given mass properties to compute the mass properties of the whole link.

Densities and colors within a group will, if set, remain what they are set to. However, any density or material which is set to default values will become set to whatever the current values are when the call to group() is made. You can force the use of the default color or density by calling use_default_color() or use_default_density(). Note that if you actually want the default color or density regardless of values outside of the group, you need to call use_color with the default color and use_density with the default density.

You cannot nest the declaration of a shape or a group. In other words, a call to begin_group() or begin_shape() cannot occur within a begin_group() end_group() pair, or within a begin_shape() end_shape() pair. The same holds for end_group() and end_shape().

You can, however, put a call to group() within a begin_group() end_group() pair if you have completed the declaration for that group already. For example:

```
begin_group("centered_cube");
translate(0.0, 0.0, -0.5);
shape(SBRICK, 1.0, 1.0, 1.0);
end_group();

begin_group("rotated_cube");
rotate('y', 45.0);
group("centered_cube");
end_group();
```

and later, within a link:

```
begin_shape();
rotate('z', 30.0);
group("rotated_cube");
translate(0.0, 0.0, 2.0);
group("centered_cube");
end_shape();
```

One could also include more shape() calls and set the masses, colors, and moments of inertia, but that would only confuse the example.

14 Mass Properties

The Creature Library will calculate and use the mass properties (mass, center of mass, and moment of inertia tensor) of the shape created within begin_shape and end_shape. A creature's density is usually about that of water, so the default is $1000.0 kg/m^3$ (water density).

One way to alter these properties is to call

```
use_density(new_density);
```

where new_density is a float (kg/m^3). Any future calls to shape() will create objects with the indicated density. The density can be zero or even negative, although massless or negative mass links should be avoided. The header file cl_lib.h has pre-defined constants for a number of materials, including water, aluminum, steel, brass, titanium, heavy and light wood, ivory, mammilian bone, acrylic, teflon, polyethylene, concrete, and Earth (average planetary density). They are defined as WATER_DENSITY, ALUMINUM_DENSITY, STEEL_DENSITY, and so on.

Alternately, you can override the mass or moment of inertia calculations completely and use your own numbers with

```
set_mass(mass_in_kg);
```

to set the mass,

```
set_com_offset(x, y, z);
```

to set the link's offset of the center of mass from the previous joint, and either

```
set_moment_of_inertia(x, y, z);
```

or

```
set_moi(x1,x2,x3,y1,y2,y3,z1,z2,z3);
```

to set the moment of inertia. Set_moment_of_inertia expects three three-vectors of floats, and sets the moment of inertia of the current link. set_moi is the same, except that it doesn't require that the input columns be in arrays. These subroutines will also set the mass and moment of inertia of a group (see the section on groups).

15 Complex Joints

The complex joints (cylinder, universal, ball, planar, etc.) implemented by SD/Fast are available through calls to routines such as set_cylinder_joint and set_universal_joint. Simply call them instead of set_pin_joint. Because you frequently get new_joint() followed by set_cylinder_joint(), there are corresponding joint_cylinder routines which combine the two.

With simple joints (pins and sliders) the name of the joint is the same as the name for the corresponding variable. However, with more complex joints there are several degrees of freedom. For those, you need to specify a joint name (for parenting) and variable names for each degree of freedom. Within the simulation, you can access each degree of freedom by the variables, but when describing the creature you can refer to the entire joint as a whole.

The different complex joints:

set_pin_joint() :

- char axis — The axis ('x', 'y', 'z').

Defines the current joint as a pin joint about the given axis.

set_slider_joint() :

- char axis — The axis ('x', 'y', 'z').

Defines the current joint as a sliding joint along the given axis.

set_cylinder_joint() :

- char *pinname — The name of the pin subjoint.
- char *slidername — The name of the slider subjoint.
- char axis — The axis ('x', 'y', 'z').

Defines the current joint as a cylinder joint along the given axis, with the pin and slider subjoints having the given names.

set_universal_joint() :

- char *majorname, *minorname — The names of the major and minor pin subjoints.
- char majoraxis, minoraxis — The axis ('x', 'y', 'z') for the major and minor subjoints.

Defines the current joint as a universal joint. The major axis must be perpendicular to the minor axis.

set_planar_joint() :

- char *firsttrans, *secondtrans — The names of the first and second translation subjoints.
- char *rotate — The name of the rotation subjoint
- char firstaxis, secondaxis, rotateaxis — The axis ('x', 'y', 'z') for the various subjoints

Defines the current joint as one which confines motion to a plane. The second planar axis must be perpendicular to the other two.

set_gimbal_joint() :

- char *firstname, *secondname, *thirdname — The names of the subjoints within the gimbal joint.
- char firstaxis, secondaxis, thirdaxis — The axis ('x', 't', 'z') for each subjoint.

Defines the current joint as a gimbal joint. Each axis must be perpendicular to the ones next to it.

set_ball_joint() :

- char *yawname, *rollname, *pitchname — The names for the yaw, pitch, and roll subjoints.

A ball joint is internally tracked as a quaternion, so there is no gimbal lock. Thus, the values for yaw, pitch, and roll will by limited to $+/-\pi$.

set_6dof_joint() :

- char *trans1name, *trans2name, *trans3name, *yawname, *rollname, *pitchname — The names for the various subjoints.
- char trans1axis, trans2axis, trans3axis — The axis ('x', 'y', 'z') for the translational subjoints.

Defines the current joint as a six degree of freedom joint. These joints do not actually limit movement, but can be used for measurements and for application of arbitrary forces.

16 Joint Stops

You can limit a joint's motion by calling limit(variable_name, min, max). Whenever the joint passes the minimum end stop, the force/torque applied is either the force normally applied by the servo, or the force from a PD servo with spring constant ls.k_stop and damping ls.b_stop, whichever is greater. The maximum end stop is the same, with the signs reversed as necessary.

If you want more control over the springs and dampers, you should instead call limit_pd(variable_name, min, max, spring_constant, damping). This acts exactly as a call to limit, except that the strings given for spring constant and damping are used in the PD servo. Three examples are:

- limit_pd("legz_1", -1.0, 0.5, "ls.k_stop", "ls.b_stop");

  This is exactly equivalent to limit("legz_1", -1.0, 0.5);
- limit_pd("legz_1", -2.0, -1.0, "ls.my_k_stop", "ls.my_b_stop");
- limit_pd("legz_1", -2.0, -1.0, "2.5*ls.k_stop", "2.5*ls.b_stop");

17 Servos

Servomechanisms will be added automatically at each degree of freedom unless you specify otherwise. Each servo has a state, held in the simulation variable servosw.JOINT_NAME. State zero (the initial state) is always limp (no applied torque or force). To add another servo state, you would make a call to servo after creating the joint and before creating the next link. For example,

```
joint_ball("hip1", "hip1_yaw", "hip1_roll",
           "hip1_pitch");
servo(1, "hip1_yaw", PD_SERVO, "ls.k_hip1",
      "ls.b_hip1");
servo(2, "hip1_yaw", PD_SERVO, "ls.high_k_hip1",
      "ls.b_hip1*2.0");
servo(3, "hip1_yaw", PD_FF_SERVO,
      "ls.k_hip1", "ls.b_hip1", "ls.ff_hip1");
```

means that in your control system, or directly from the simulation interface (see INTERFACE), if you set servosw.hip1_yaw to 1.0, you will have a position-damping servo with spring constant determined by the variable ls.k_hip1 and damping determined by ls.b_hip1 (whatever values those variables happen to hold). It will try and servo to the desired position (determined by q_d.hip1_yaw). If you set servosw.hip1_yaw to 2.0, you will again have a position-damping servo, but with different spring and damping values. If you set it to 3.0, you will have another position-damping servo, the same as the first, except that it has a feed-forward torque (ls.ff_hip1). Finally, if you set servosw.hip1_yaw to 0.0 the servo will be limp (the default).

Although that example only dealt with hip1_yaw, the other degrees of freedom (hip1_roll and hip1_pitch in this case) can have their own sets of servo states, independant of hip1_yaw.

Each of the servo types takes a series of parameters. In the servo call, they would be strings after the servo type which are textually substituted for the appropriate parameters within the simulation.

- LIMP_SERVO - no parameters
- PD_SERVO - k (spring constant) and b (damping). tau = -k*(position error) - b*(velocity)
- PD_BALL_SERVO - k and b. As PD_SERVO, but deals with the fact that ball joints limit the angles to +/- PI
- PD_FF_SERVO - k (spring constant), b (damping), and ff (torque). tau = -k*(position error) - b*(velocity) + ff
- PD_PLUS_SERVO - k, b, other_k, other_b, other_value, other_desired, other_derivative, second_b, and second_derivative. This servo calculates its torque as two PD servos and an additional damping term.

- LEG_SERVO - k_compress, b_compress, k_extend, and b_extend. Another PD servo, but with different values for extension and compression forces.
- LEG_R_SERVO - k_compress, b_compress, k_extend, b_extend, and spring_len. Another PD servo, but with a 1/r force applied on extension.

The routines which actually implement these servomechanisms can be found in servo.c (once the simulation is made) if there is any doubt about the forces applied.

Using those routines as models, you can create your own servomechanisms. Simply write the routine (placing it somewhere safe, like control.c) using the existing ones as a guide, and then in the creating program call

```
user_servo(2, "hip1_yaw", "my_servo_routine", 3,
           "ls.param1", "ls.param2", "ls.param3");
```

replacing my_servo_routine with your routine name, 3 with the number of parameters other than *tau, q, qd, and q_d (which are assumed), and the parameters (ls.param1, ls.param2, ls.param3) with the parameters you actually want passed to your new servo routine.

Alternately, you can call no_automatic_servos(). No servomechanisms will be placed at any joints, but whenever servos should be updated there will be a call to servo1() (located in control.c). You can then place your own servo code there - all it has to do is set the tau structure to the forces to be applied.

18 Ground Contact

Arbitrary contact is not checked for. However, you can designate a point on a link as a ground contact point with

```
ground_contact("foot", x, y, z);
```

or

```
mobile_ground_contact("foot", "1.0", "2.0", "ls.z_pos");
```

By calling ground_contact, you designate a specific point on an an arbitrary vector from the link origin as a ground contact point. As necessary, this point will be checked to see if it on the ground. When it lands, the contact point will be recorded and a force determined by the ground contact model will be applied.

Note that in the second version the ground contact points are strings which are simply copied into the appropriate positions in the generated simulation. They can be constants, variables, or functions. Be aware, however, that if you place a function call within the strings indicating the position it should be as efficient as possible.

Contact forces are provided, in general, by a spring–damper system (with a torque spring for rotation). This means that the feet will penetrate the ground a little, but for our purposes this is acceptable.

Information on a ground contact point is kept in a structure gc_"contact_name" with the fields

- fs — The current status of the "foot switch", 1.0 if the foot is down and 0.0 if it is in the air.
- td_x, td_y, td_z, td_th — The position and rotational orientation at touchdown.
- at_x, at_y, at_z, at_th — The current position and rotational orientation.
- k_x, k_y, k_z, k_th — The spring constants in each direction.
- b_x, b_y, b_z, b_th — The damping in each direction.
- f_x, f_y, f_z, t_th — The forces and torques applied.
- nomlen_z — Used as the nominal spring length in some ground contact models.
- model — The ground contact model in use.

The model itself can be altered at any time by changing the model field of the ground contact's structure (for example, gc_foot.model). The initial model is zero, which means that no forces will be applied. Currently implimented values are:

- 0 — No forces will be applied
- 1 — a linear spring-damper system without foot torque.
- 2 — attached to the ground. As long as the point is on or below the surface of the ground (z=0.0), there will be a force pulling it back to whatever is designated as the touchdown point (at_x, at_y, at_z, at_th).
- 3 — a nonlinear spring-damper system (the spring forces are a quadratic function of displacement) without foot torque.
- 4 — a linear spring-damper system with foot torque.
- Greater than 100 — If the model number is greater than 100, a routine called user_gcontact will be called.

There is a stub for the user_gcontact routine in control.c (where it won't go away). It is intended to allow you to create your own ground contact models. At the time this routine is called, the position and angle in the contact structure are correct, and the position and velocity is passed as parameters. This routine should set the forces as necessary for your ground contact model and return.

19 Spring Constants

Once the Creature Library knows the mass properties of the simulation, it will create suggestions for critically damped springs. Those suggestions are in the files suggestions and Ssuggestions. The suggestions file is easier to read, while the Ssuggestions file is loadable directly into the simulator, but they both contain the same information. For each joint, assuming the rest of the simulation is rigid, it calculates the spring and damping values for critically damped oscillation at 1 Hertz. It performs this calculation three assuming the simulation is free-floating, that the parent of the joint is fixed in place, and that the child of the joint is fixed in place. You should choose appropriately.

If you want the suggestions to use a position other than 0 for the joint, or a frequency other than 1 Hertz,

```
nominal_position("myjoint", rest_position, frequency);
```

will set the rest position (given in radians or meters, just as in the simulation itself) and critically damped frequency (in Hertz).

20 Miscellaneous

20.1 Command line options

```
main(argc, argv)
  int argc;
  char *argv[];
{
  command_line(argc, argv);
```

By adding these parameters to main and calling command_line with them, the resulting program will check that the species name corresponds to the creating file name, and permit you to specify which files to alter.

Valid parameters are:

- +all, -all - create/don't create all files

- +anim, -anim - create/don't create CREATURENAME_anim.c
- +dat, +sdfast, -dat, -sdfast - create/don't create the input to SD/Fast

Similarly, for any individual file (like ground.c) you can use

- +ground, -ground - create/don't create the file ground.c

Parameters are read left-to-right and evaluated in order, and (unless it would create ambiguities) only the first four letters of the filename are important. Firstrun.c and vars.h are too intertwined to produce separately.

To not touch the SD/Fast input, you would use

```
create_test -dat
```

To only modify ground.c, you could use (since four letters are enough)

```
create_test -all +grou
```

Finally, if you are doing a lot of work where you are making cosmetic changes which won't affect the model, you can do "make anim" (to create only the anim file), "make legplot" (to create only the legplot file), or "make graphics" (to make both the anim and the legplot files). These commands work by running

```
create_test -all +anim +legplot
```

and they will also copy them into the proper directories.

They will not execute a "make" in those directories. You have to do that yourself.

20.2 Alternate integrators

```
alternate_integrator("cl_plant2.c");
```

You can use an alternate integrator by putting a call to alternate_integrator somewhere between begin_species and end_species.

Currently, there are only two functioning integrators: cl_plant.c and cl_plant2.c. cl_plant.c uses the integrator which comes with SD/Fast, while the integrator in cl_plant2.c is available for checking and modification. If you specify an alternate integrator, it will look in the current directory and then in the directory indicated by SIM_SOURCE_FILES.

20.3 Gravity

```
set_gravity(x, y, z);
```

This routine will override the default gravity (earth-normal gravity, with "down" in the negative z direction). It is not possible to have different gravities for different links; the last call to set_gravity before the end_species will be the only effective call. Calling set_gravity before begin_species will have no effect.

20.4 Track point

```
set_track_offset(xoffset, yoffset, zoffset);
```

Anim will track the point (x, y, z) if the variables x, y, and z exist, substituting zero if they do not. Calling set_track_offset allows you to specify a constant offset from that tracking point. The tracking point is used when positioning creatures onscreen, finding the camera's fixation point, and finding the dolly position.

20.5 Alternate output file names

```
set_files("test.out", "test.dat", "test_anim.c");
```

This routine is out of date. If you use it, you will receive a warning and the call will be ignored.

20.6 Legplot body size

```
set_legplot_link_size(xmin, xmax, ymin, ymax,
                      zmin, zmax);
```

For a legplot cartoon, each link is represented as a brick for ease of visulation and display speed. Normally, link size is calculated from the moment of inertia entries on the main diagonal, but with this call you can specify the size of the cube representing the link.

20.7 External forces

```
external_force("push", x, y, z);
external_force_com("wind", x, y, z);
```

The first declares an external force point named "push" at the link origin (or an arbitrary vector from there). The second declares an external force point named "wind" at the center of mass of the current link; if you override the calculated location of the center of mass with set_link_offset, the new center of mass will be used. The vector allows you to displace the force point.

External force points cannot be moved, relative to the link, during simulation runs. Their locations on the links are fixed. Upon the declaration of an external force, a routine is added to eforces.c; this routine should be altered so that it sets the external force as desired.

20.8 Suppression of automatic servos

```
no_automatic_servos();
```

If you call no_automatic_servos(), no servomechanisms will be placed at any joints. All calls to servo() or user_servo() will be overridden; instead, whenever servos should be updated there will be a call to servo1() (located in control.c). You can then place your own servo code there - all it has to do is set the tau structure to the forces to be applied.

20.9 User constants

```
user_constant("name", val);
```

This call will cause the given constant to be defined with the appropriate value within the simulation. It is useful for passing parameters such as leg lengths to the actual simulation when you need them for your control algorithm.

The resulting code, in creature.h, is:

```
#define name val
```

20.10 Double-body actuators

```
make_actuator_connection("name", end, x, y, z);
make_actuator_connection_com("name", end, x, y, z);
```

The first declares the end of a double-body actuator names "name" at the link origin (or an arbitrary vector from there). The second is the same, except from the center of mass of the link.

Because it is a double-body actuator, you need to specify the end as an integer 1 the first time you specify the actuator, and a 2 the second. This has the advantage of double-checking the names of different ends.

For each actuator, a subroutine is added to the eforces.c file. Initially, the subroutine doesn't apply any force, but you can alter it to any function you desire. At some point in the future, you will be able to call a subroutine to specify the name of the called routine, so you don't have to modify each stub yourself.

The basic_act structure, with information commonly used to determine the force applied and a place to set the force, is:

```
typedef struct {
  double force;
  double mag;
  double dx;
  double dy;
  double dz;
  double ddx;
  double ddy;
  double ddz;
  double dmag;
  double a;
  double b;
  double c;
} basic_act;
```

The variables containing this information for each actuator are named act_"name" (where "name" is replaced with the name of the actuator specified in make_actuator_connection). Before each call to determine the force (held in the variable force), the fields are updated. The fields dx, dy, and dz contain a vector between the connected points, and the fields ddx, ddy, and ddz contain the derivative of that vector. Mag and dmag are the magnitudes of the difference vector and the derivative, respectively. Force should be set to the force to apply, with a positive force pushing the points apart and a negative one pulling them together. A, b, and c are scratch variables for your own use.

If you alter the fields dx, dy, dz, mag, ddz, ddy, ddz, or dmag the results will not be reliable since they are cached here for use later.

There have been reports of bugs involving the signs of the force near reversal points with the sign, but I have yet to have anyone show me an example.

21 Resulting Simulation

21.1 Where is My Code Safe?

The written simulation consists of many files, some of which are completely re-written each time the simulation is created, some of which are modified, and some of which are left untouched.

In any file, if there are the lines

```
/* BEGIN USER CODE 1 */
```

and

```
/* END USER CODE 1 */
```

you can add your own code between those two comments and it will be kept, even if the file is completely rewritten.

When you add your own code, it is possible (although not recommended) to directly call SD/Fast routines to apply forces and torques to the model. The reason it is not recommended is that the force application routines written by the library set an absolute limit to forces and torques (interactively modifiable by fmax and tmax), making it easier to debug your control.

21.2 Units

By default, the Creature Library uses MKS (Meters, Kilograms, Seconds) units. One can equivalently use CGS (Centimeters, Grams, Seconds), as well. Angles within a "create" file are given in degrees, although the resulting simulation will use radians. The following conversions are provided for convenience; they are defined as constants, and you do the conversion by multiplication:

- DEG_TO_RAD: Degrees to radians.

- RAD_TO_DEG: Radians to degrees.
- FEET_TO_METERS, METERS_PER_FOOT: Feet to meters.
- INCHES_TO_METERS, METERS_PER_INCH: Inches to meters.

21.3 Variables

Within the generated simulation, there are several structures containing position, torques, servo states, and so on. Some of them are:

- q: position.
- qd: velocity. Be aware that the velocity of a ball joint is given in body-fixed coordinates.
- u: velocity (an artifact of the way the integrator works). This velocity is given in Euler parameters, not Euler angles. You should use qd or CL generated routines to access these angles.
- ud: acceleration.
- Any of the above with a "_d" after it: a desired value. These are used by servos and your control system.
- Any of the above with a "_iv" after it, including desired values. These were once used for initial values, but are out of date and will disappear soon. Just don't touch them.
- tau: the amount of force or torque, depending on the joint type, being applied to the joint.
- servosw: the servo switch for that joint (see SERVOS).
- ground contact: for ecch ground contact point, a variable of type basic_gcontact named "gc_¡name¿" replacing ¡name¿ with the name of the ground contact point
- connected actuators: A variable is added for each connected actuator.

21.4 The "ls" Structure

ls stands for "locomotion structure". Any variables which exist in the code as "ls.variablename" will automatically be added to the structure. These variables, along with the structures such as q, qd, q_d, etc. are all accessible while the simulation is running. So, if you use "ls.foobar" in your control code, next time you do a "make" a line

```
double foobar;
```

will be added to the "ls" structure for you, along with extra code in control_vars.h which will allow you to access the variable through the interface while the simulation is running.

21.5 The Files

- Makefile: This file is only modified when setup_simulation is run. There is a commented-out line of compiler options which uses the debugger and if you change the library include from -lsim to -lsim_g you get the debuggable version of the library as well.
- Makefile.bak: This is a copy of Makefile made automatically when file dependancies are calculated.
- Makefile.old: When setup_simulation is run, the current Makefile will be copied to this file before the new one is created.
- S51.0: Any files consisting of "S" and a number, or just a number, are state files and data files from the simulation.

- control.cskel: A skeleton control system, provided so you know which routines are expected to be in control.c
- control.c: The control system. Every control_dt, control1() (a procedure in control.c) will be called. Also, if automatically generated servos aren't used servo1() will be called every servo_dt so you can write your own servos.

  Because this file is never rewritten, you can make any changes you wish to it. This is also where the code for user-defined servo and ground contact types is (user_servo and user_gcontact, called from servo.c or ground.c).
- control_vars.hskel: Skeleton for control_vars.h
- control_vars.h: Any variables used in the control system which you want to be able to access interactively during run-time should be added to this file, which is never rewritten.
- creature.h: Definitions, externs, and structures used in the simulation. These are accessible to any portion of the simulation. Each link has B_linkname defined to be the SD/Fast body number for it, and each joint has J_jointname and SJ_jointname to be the joint number and subjoint number used by SD/Fast. There are also definitions for PI and conversion factors to or from degrees. This file is rewritten every time
- eforces.c: This file contains procedures to calculate the external forces applied at external force points. Every time an external force point is declared, if there is no corresponding routine it will be added to this file, but code is never removed from here. Routines for calculating double body actuator forces are added here, too, because they were initially implimented as pairs of external forces.
- firstrun.c: This file contains a procedure executed when, interactively, you type "FIRSTRUN". It resets all the variables to (usually) sane values. It is rewritten, but if you change an assignment that change will be kept. For example, you could change the initial height off the ground by changing the line "q.z = 0.0;" to "q.z = 1.0;" and this will be kept as long as there is a variable q.z. This means that if you remove (for example) a contact point the initializations for it will be lost.
- ground.c: This file contains the procedures which govern ground contact, and the predefined ground contact models. It is rewritten, but if the ground contact model is greater than 100 it will call user_gcontact in control.c.
- locomotion.hskel: Skeleton file for locomotion.h
- locomotion.h: This file contains the locomotion (control system) variables. It is rewritten to include all the locomotion variables whenever the source code is modified.
- main.c: This is main body. It has user code segments to allow you to add more interactive commands to the simulation, but aside from those segments is rewritten every time the simulation is created.
- CREATURE.dat: The input file for SD/Fast. Useful for checking the dynamics if you're suspicious.
- CREATURE.out: An ASCII text file describing the creature as the library sees it. Once you create the simulation, this file makes it easy to check some aspects of it.
- CREATURE_anim.c: C code which can be included in anim to display the new creature. Copy it into /home/ll/anim/gx_iris, edit the Makefile there (add CREATURE_anim.c to the list of CRSRCS), and edit modeldef.c so it externs the correct init, build, delete, draw, and pos routines and has an entry for your creature in the array at the bottom of the file. Re-make anim using the new Makefile. You only need to do this when you change the actual look or structure of the simulation; otherwise is won't matter. This file is rewritten.
- CREATURE_dyn.c: C code written by SD/Fast; the dynamics of the simulation.

- CREATURE_info: Information about the simulation, written out by SD/Fast as it creates the dynamics. This file is sometimes useful to verify that your model is correct.
- CREATURE_sar.c: Simplified Analysis Routines written by SD/Fast.
- plant.c: the integrator, or call to SD/Fast's integrator (with glue code). This file is rewritten.
- plant_get_sdfast.c, plant_set_sdfast.c: Glue code to convert from the simulation's structures to the array expected by SD/Fast.
- rs6000, iris, sun4, sun3: Directories where the object files are kept. Not all of these will be present.
- sdlib.c: SD/Fast library routines written by SD/Fast.
- servo.c: code which implements the predefined servomechanisms.
- vars.h: structure which allows the variables to be modified interactively while the simulation is running. The parameters on each line are:
  - The name you will use to refer to the variable interactively.
  - The address of the variable (doubles only).
  - The minimum value (used for display by legplot).
  - The maximum value (used for display by legplot).

  Minimum and maximum values are not used for anything but display.

22 Adding a Creature

22.1 Adding a Creature to Anim

In order for your creature to appear in anim, you need to add it to the animator. This involves editing several files which, if done wrong, can keep the animator from recompiling; but it only needs to be done the first time.

$ANIM_SOURCE: Makefile Add your creature to the CRSRCS list. Make sure that the first character in the line is a tab, not spaces.

$ANIM_SOURCE: modeldef.c Copy the five extern definitions with the string "Camera", and replace "Camera" with your creature name (case is important).

$ANIM_SOURCE: modeldef.c Take the camera creature descriptor

```
{ "CAMERA", 0,
    init_Camera, build_Camera, delete_Camera,
    draw_Camera, pos_Camera, NULL },
```

and copy it, replacing "Camera" with your creature name again. The string "CAMERA"should be in upper case; the rest should be the same case as the declaration of your creature.

Now follow the directions under "Altering a Creature"

22.2 Adding a Creature to Legplot

In order for your creature to appear in anim, you need to add it to the animator. This involves editing several files which, if done wrong, can keep the animator from recompiling; but it only needs to be done the first time.

$LEGPLOT_SOURCE: Makefile Add your creature to the SRCS list. Make sure that the first character in the line is a tab, not spaces.

$LEGPLOT_SOURCE: cartoon.c Find the "Cartoon function declarations"; add init and draw routines for your creature. Case is important.

$LEGPLOT_SOURCE: cartoon.c Find the "Model table". Add a line with the name of your creature, the init and draw routines, and a "Nop" (no operation).

Now follow the directions under "Altering a Creature"

22.3 Altering a Creature

To alter the graphics display of an existing creature, you need to take the following steps:

In the creature's directory: `make graphics`

In $ANIM_SOURCE or $LEGPLOT_SOURCE: `make`

In $ANIM_SOURCE or $LEGPLOT_SOURCE: `anim` or `legplot`, depending on the directory

In $ANIM_SOURCE or $LEGPLOT_SOURCE: if it worked, `make update`

Be careful using `make update`. It will only update the current machine type, and it replaces the commonly used binary with the new one. This means that anyone else using the same program on the same machine type will get a crash if your update succeeds.

23 Common Pitfalls

23.1 Garbage velocities

The velocities are completely (or even slightly) wrong.

Be aware that velocities (qd) for ball joints and the ball joint section of six degree of freedom joints are given in body-fixed coordinates. This avoids singularity problems. The derivatives of the positions (u) are in Euler parameters, not in Euler angles, and shouldn't be modified except for initialization.

23.2 Automatic servos

How do I suppress automatic servos?

Call no_automatic_servos() and put your own servo code in servo1(). This call overrides the effects of all calls to the routines servo() and user_servo().

23.3 Control-C

I can't use control-C to get into dbx.

Right. This is a problem with the simulation interface. The best solution I have found is to put a breakpoint in the routine dbx() and use the simulation command DBX to call that routine. You can then put breakpoints where you suspect the problem lies. Alternately, simply put the breakpoints in before you actually run the program.

23.4 Debug

How do I turn debug on?

In the window you will be compiling, type

```
setenv DEBUG debug
```

To turn it off, type

```
unsetenv DEBUG
```

Be aware that this only affects the files which are compiled after the command. The debugger cannot effectively debug sections of code which haven't been compiled this way.

23.5 Won't Recompile ("Nothing to do for...")

I turned debug on, but now it says "nothing to do for ..." when I say make.

Make doesn't know you set debug on, and there was a successful compile before you turned debug on, so as far as it can tell, it is right.

Go to your simulation directory and type

```
touch create_*.c
```

and make should recompile everything

23.6 Won't Recompile (Anim or Legplot)

It won't recompile Anim or Legplot. Make complains about an error in the Makefile.

When you modify the Makefile for Anim or Legplot to include your new creature, the most common mistake is to start a line with a space instead of a tab. In some ways, make is brain-dead, so it requires that lines which continue previous sets of instructions begin with a tab character.

23.7 Won't Recompile

It won't recompile everything in my simulation. No matter how much I change this file, it won't recompile it.

Somewhere, the file Makefile got altered. One possibility is that your dependancy tree is messed up. Go to the directory with your simulation and type

```
make depend
```

This approach has the advantage of keeping any changes you made to your Makefile.

The brute-force solution (DO NOT DO THIS ANYWHERE BUT IN THE DIRECTORY WITH YOUR SIMULATION) is to go to the directory with your simulation and type

```
setup_simulation NAME
```

where name is the name of your simulation. This will completely wipe out your Makefile and replace it with a new one.

23.8 Trashed Parameters

Although I am passing perfectly good parameters to the routine, if I use dbx to step inside the routine the parameters are garbage.

Check the types of your parameters. All Creature Library routines you should be calling have the types for the parameters included in comments in the file /home/ll/include/cl_lib.h, and if you get the type wrong you will be passing garbage to the routine.

Additionally, although the compiler will do type conversion when it assigns to a variable it will not do type conversion for parameters unless you specifically tell it to. This means that

```
use_density(1000);
```

will probably generate a warning message, since that should turn into an unacceptably large density, while

```
use_density(1000.0);
```

or

```
use_density((float)1000);
```

will work.

23.9 ANSI Prototypes

I can't put the types of the parameters in the parenthesis like ANSI C says I can.

```
int foo(int i){return(i)}
```

won't work.

The Sun C compiler is not ANSI-compliant and can't handle ANSI prototyping. It may work on an IBM or an SGI, but don't do it because it won't port back to the suns.

23.10 Different Machines, Different Answers

My simulation gives different answers on sun or Silicon Graphics machines than it does on an RS6000 like Talus.

The RS6000 use a different amount of accuracy in their floating point calculations. Somewhere, the small amount of difference adds up. Actually, the IBM is right: it has more precision. Change all the floats you can to doubles, and your problem should go away.

23.11 Taking Too Long

It's taking FOR EVER.

Either your model is too complicated, or you're running on a slow machine. You can speed compilation up by turning debugging on. This compilation speed-up is at the cost of optomizing the simulation, so the simulation itself will run more slowly.

If you are simply trying to make cosmetic changes, try "make graphics", "make legplot", or "make anim".

23.12 Simulation Won't Simulate

My simulation doesn't do anything. Even though I said "run 1", t (time) isn't going up.

Either your simulation is very slow or you never gave the simulation command FIRSTRUN, which initializes most variables (such as time steps) to sane values.

23.13 Simulation Commands

What are the simulation commands?

The simulation interface is not really a part of the Creature Library, but Yuri was nice enough to come up with this:

- START: automatically sets the run time to 60 sec
- STOP: stops the simulation
- STEP n: run for n dt's (default is n = 1)
- RUN n: run for n seconds

- FIRSTRUN: sets all the variables to reasonable values. Do this to create an initial S file
- SHOW var-name / HIDE var-name: display the variable and its value on the control panel. HIDE undo's this.
- RECORD var-name / UNRECORD var-name: highlights the variable and marks it for recording in a data file when you do a SAVE
- SEND var-name: marks the variable for piping to another program for realtime display of the simulation data
- SET var-name value: give the variable a value
- SAVE: save the values of the RECORDed variables over the time of simulation in a data file. It also creates a corresponding S file (see SSAVE).
- SSAVE: create a "state" or "S" file containing SETs, RECORDs, and SENDs. This creates a snapshot of the current state of the simulation.
- LOAD: load an S file (see SSAVE).
- STARTGRAPHICS machine: Opens a pipe directly to either anim or legplot running on a given machine. Variables marked with SEND will be sent over this pipe. When connecting to legplot, it will only work if you have given the simulation an individual name (see INDIVIDUAL).
- INDIVIDUAL string: Gives an individual name to the simulation. Not usually necessary unless you use STARTGRAPHICS, but Legplot will not accept a STARTGRAPHICS command unless you have declared an individual name.
- QUIT: exit the simulator

To change a variable and record the values:

```
SHOW q.*   display all variables that begin with ``q.''
SET q.z 3  sets the vertical height to 3 m
RECORD q.* you tell it what variables you want recorded
RUN 0.01   runs the simulation forward for the given
           number of seconds
SAVE       creates a data file as well as an S file
```

To move up in the command history to edit previous commands, press control-p. To move down, press control-n.

23.14 Working Copy of the Creature Library

How do I use the "working copy" of the Creature Library.

Normally, you don't. The working copy of the Creature Library is for testing purposes only.

If you do have to do it, though,

```
cd /home/ll/anim/cl
make working
cd <your simulation directory>
setenv WORK work
setenv SIM_SOURCE_FILES \
        /home/ll/anim/cl/sim_source_files
touch create_*.c
make
```