The Legged Robot 3D Simulator: Description and Programming Guide

Josep M. Porta
Institut de Robòtica i Informàtica Industrial (UPC-CSIC)
Gran Capità 2-4, 08034, Barcelona, SPAIN
e-mail: jporta@iri.upc.es

IRI-DT-2000/4

August 1, 2000

Contents

1 Introduction **6**
1.1 Objective . . . 7
1.2 General Structure . . . 7
1.3 Installation Issues . . . 9

2 User Interface **12**
2.1 The On-Line Interface . . . 12
2.1.1 The Main Panel . . . 12
2.1.2 The Robot Menu . . . 14
2.1.3 Move leg Panel . . . 16
2.1.4 Set angles Panel . . . 17
2.1.5 Set positions Panel . . . 18
2.1.6 Balances Panel . . . 19
2.1.7 Sensors Panel . . . 22
2.1.8 Move Robot Panel . . . 23
2.1.9 Load Robot Panel . . . 23
2.1.10 Legend Panel . . . 24
2.1.11 Change View Panel . . . 24
2.1.12 Traces Panel . . . 25
2.1.13 Camera Panel . . . 26
2.1.14 The World Menu . . . 27
2.1.15 Load Ground Panel . . . 27
2.1.16 Load Landmarks Panel . . . 27
2.1.17 Save Landmarks Panel . . . 28
2.1.18 Manage Landmarks Panel . . . 28
2.2 The Off-Line Interface . . . 30
2.2.1 Ground Configuration . . . 30
2.2.2 Landmark Configuration . . . 33
2.2.3 Robot Configuration . . . 33

3 Programming a Robot Controller **39**
3.1 The User-Defined Controller Structure . . . 39
3.2 The Robot . . . 40
3.2.1 The Object . . . 40
3.2.2 Setup Functions . . . 41

3.2.3 Structure Functions 42
3.2.4 Positioning Functions 42
3.2.5 Motor Functions 43
3.2.6 Gesture Functions 43
3.2.7 Balance Functions 44
3.2.8 Camera Control Functions 46
3.2.9 Execution Control Functions 47
3.2.10 Sensor Functions 48
3.2.11 Landmarks Functions 49
3.2.12 Frame of Reference Conversion Functions 50
3.2.13 Trace Functions 50
3.2.14 Visualization Functions 52
3.2.15 Transform Functions 52
3.2.16 Landmark Functions 57
3.2.17 An Example: A simple controller 58

4 Future Extensions 65

List of Figures

1.1 *The Argos six-legged robot and its corresponding simulation.* 6
1.2 *The simulator logical structure.* 8
1.3 *The processes of the simulator. Each oval represents an independent process* 8

2.1 *The simulator initial appearance.* 13
2.2 *The main panel.* 14
2.3 *The tools panel.* 15
2.4 *The move leg sub-panel.* 16
2.5 *The body frame of reference (dashed lines) and a leg frame of reference (dotted lines).* 16
2.6 *The set angles sub-panel.* 17
2.7 *The set position sub-panel.* 18
2.8 *The balances sub-panel.* 20
2.9 *The references sub-panel.* 21
2.10 *The tensions sub-panel.* 21
2.11 *The sensors sub-panel.* 22
2.12 *The move robot sub-panel.* 23
2.13 *The load robot sub-panel.* 23
2.14 *The legend robot sub-panel.* 24
2.15 *The change view sub-panel.* 24
2.16 *The traces management sub-panel.* 25
2.17 *The camera control sub-panel.* 26
2.18 *The load ground sub-panel.* 27
2.19 *The load landmarks sub-panel.* 27
2.20 *The manage landmarks sub-panel.* 29
2.21 *The new landmark sub-panel.* 29
2.22 *The move landmark sub-panel.* 29
2.23 *Top view of the control grid of a ground definition.* 31
2.24 *The* `step.grd` *ground.* 31
2.25 *The contents of the* `step.grd` *ground configuration file.* 32
2.26 *A landmarks configuration file.* 33
2.27 *The frame of reference attached to the robot's body.* 34
2.28 *The leg structure. Angles point toward its positive sense.* 35
2.29 Configuration file for a long hexapod robot. 37
2.30 Configuration file for a spider-like hexapod robot. 38

3.1 *The body frame of reference (dashed lines), two leg frame of references (dotted lines), and the world frame of reference (solid lines) in an arbitrary position.* 40
3.2 *The reference frame of reference.* 41
3.3 *Graph of states for the example controller.* 58

Abstract

We present a description of a simulator of legged robots. The simulator allows the definition of different models of robots as well as different shapes for the ground. The robots are provides with many physical sensors (touch sensors in feet, inclinometers, an active camera, ...), virtual sensors (stability sensor, ...) and different motors that can be commanded either individually or coordinately. The controller of the robot is a separate module that the user can re-program at his convenience interfacing with the simulated robot as he would done with a real robot and without worrying about the internal structure of the simulator. Additionally, the simulator provides tools to monitor the workings of the robot controller: for instance it allows the drawing of traces for those mobile element (body, feet, ...) that the user want to monitor along the time. As a last feature, the simulator includes the definition of landmarks identifiable with the robot camera (if they are at sight) and that can be useful to test navigation algorithms for legged robots [13].

Up to now, only the kinematic of the robots is simulated and no dynamic effect is taken into account. Additionally, we assume detectable but traversable ground with infinite friction and we also assume that individual leg movement can not produce robot's displacements (leg movements have to be explicitly coordinated to produce a robot's body movement [4]). This allows to accelerate the simulation and, in general, it is sufficient to test the robot controllers.

In this document the simulator is described either from the point of view of the end user and from the point of view of the programmer. In the first place we present the general structure of the simulator and how it can be installed. Then, we include the description of the user interface, explaining how the controllers can be executed to observe its effects and how the different parameters of the robot and the ground can be displayed and modified on line. Next we describe how to program a new robot controller detailing how to interface with robot's sensors and actuators from a user program. Finally we outline future extensions of our work.

Chapter 1

Introduction

In general (but not always [14]), legged robots are special purpose robots constructed in research laboratories for experimental reasons and that result difficult to built, very expensive, and not very robust.

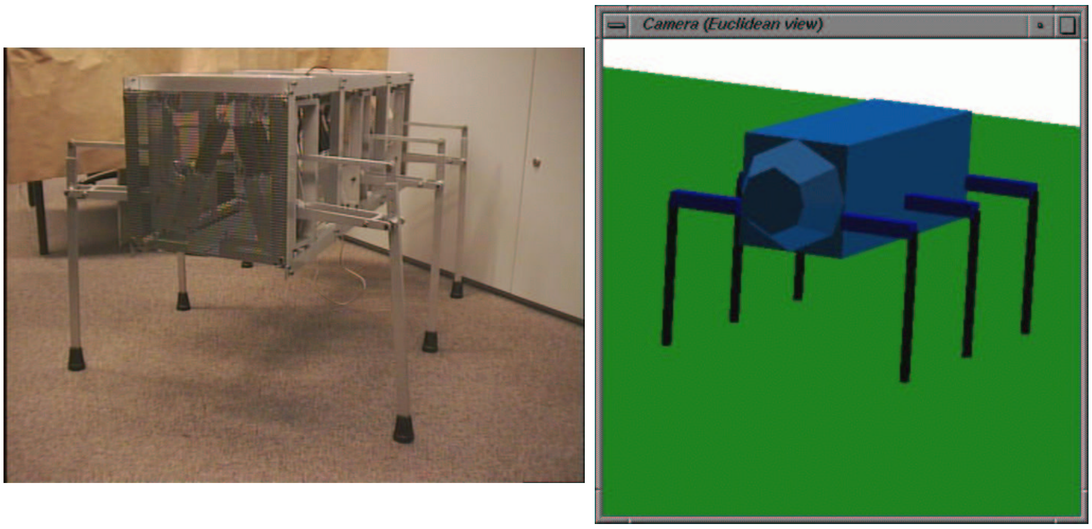


Figure 1.1: *The Argos six-legged robot and its corresponding simulation.*

Performing experiments with a legged robot is a tedious and dangerous process since experiments tend to last long periods and because any programming error can result in a damage of the robot. For these reason it seem reasonable to test tentative controllers in simulation previously to test them in the real robot. However, simulator can not be as accurate as reality and can not replace it at all [3]. In general the more the realistic the simulator the more useful it becomes. Trivial simulators for legged robots can only deal with flat terrain and the robot's legs are reduced to a two degrees of freedom (dof) mechanism that move legs up and down or forward and backward. These simulators are only useful to study the generation of patterns of steps. For instance, we use one of those simple simulators in [8] and in [10]. To test more complex controllers, more realistic simulators are needed. The drawback is that the more the realistic the simulator the more the complex and slow it becomes. Consequently, we have to meet a trade off between fidelity to the reality and usefulness. In the simulator presented in this report, we include the simulation of abrupt terrain (it is the only terrain in which it make sense the use of legged robots) but in the most simple way: Leg-ground interaction are detected with a simple touch sensors (no force sensors are provided), ground friction is considered to be infinite (so all leg to ground contacts are valid footholds), and the robot body can only be propelled if the supporting legs are explicitly moved in a coordinated way. Additionally we only consider the kinematic aspects of the robot (leg positioning) and we do not simulate dynamic aspects (inertias in leg movements, gravity, …). In general this is not a drawback since in the main part of the legged robots only use statically stable gaits. For those aiming to produce dynamic gaits more complex simulators must be use [12].

1.1 Objective

The objective of our simulator is that of providing a minimal tool for testing controllers for legged robots with four or more legs, with different leg distributions, and walking on abrupt terrain performing simple navigation task based on vision sensors.

1.2 General Structure

Our simulator implements two main objects: the robot and the world (figure 1.2). The robot includes either the emulation of the mechanical parts of the robot (legs, sensors, …) as well as the execution of the robot controller. As far as the world is concerned, it implements the ground and the visual landmarks emulation. All these objects can be initialized by the user by setting the corresponding configuration files and they can be controlled on line through the user interface provided by the simulator.

The different objects interact between them:

- The robot controller queries the physical robot to get the sensor values and provides low level commands to be executed by the robot.
- The physical robot receives the low level commands from the user (either from the controller of directly from the on-line user interface), executes them and updates the robot sensor values according to the result of the execution.
- The ground object is used by the robot object to determine the activation value of the touch sensors of legs.
- The set of landmarks is queried by the robot to select those that are visible at a given moment and to determine their position in the robot's frame of reference.
- The user interface can send low level commands to the robot and can modify the ground and landmark setup (change the ground shape, add or delete landmarks, …).

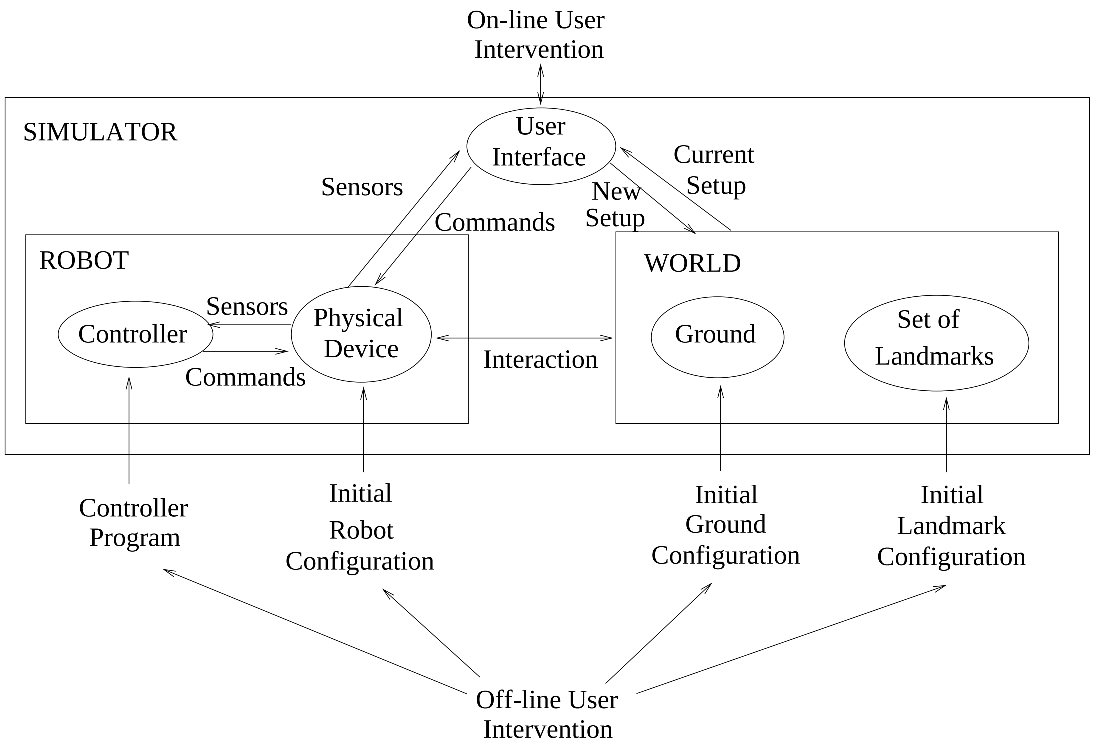


Figure 1.2: *The simulator logical structure.*

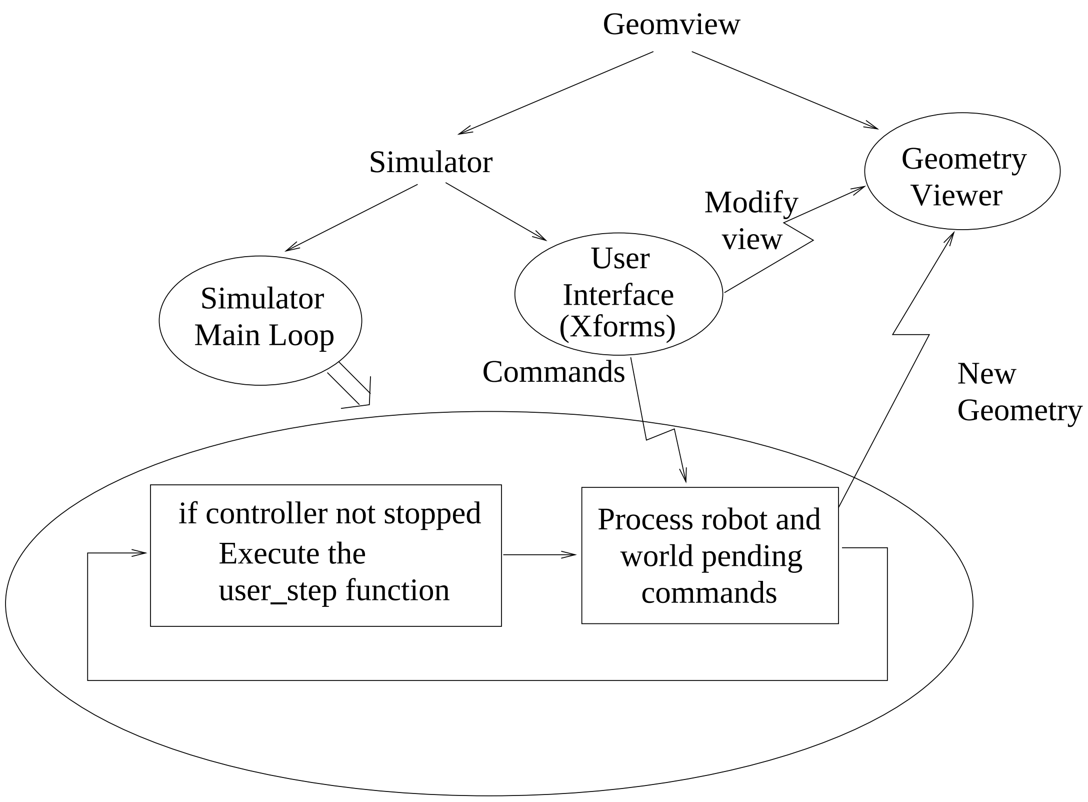


Figure 1.3: *The processes of the simulator. Each oval represents an independent process*

Two kinds of simulators are possible according to who owns the control of the simulation execution: the user controller or the simulator routines. In the first approach, the user controller constitutes the main program of the simulation and has to decide when to pass control to the robot and world update routines. An example of this approach is the Webots simulator [5]. In the second approach, the simulator routines passes the control to the user controller form time to time (in general once each time slice). The user controller has to use these opportunities to command low level commands to the simulated robot that are executed as soon as the user routines return the execution control to the simulator code. An example of this approach, that is the one taken by the simulator presented here, is the Kephera simulator [6].

When running, three processes are executed in parallel (see figure 1.3). The first one implements the main loop in charge of executing the user defined controller from which low level commands are derived. This process uses these low level commands to update the state of the robot and the world for the next time slice. The second parallel process executes the user interface from which the state of the robot and the world can be modified on line. Finally, the third parallel process is the one in charge of displaying the geometry (that is changed when the robot or the world are updated). This process also allows to change the view on the scene according to the user preferences.

The simulator is executed as an external module of Geomview [7]. This program is in charge of the geometry management and the external module implements either the user interface and the main loop processes.

1.3 Installation Issues

The legged robot controller requires of the following items to work properly:

- A machine running under Solaris (Sun) or Irix (SGI) operating system. Up to now the simulator has not been tested under Linux but it should work.
- The simulator package.
- The Geomview 3D visualization program properly installed [7].
- The Xforms library installed [16].
- The GNU C compiler.

To install the simulator package copy the file legsim.tar.gz[1] in one of your directories and type

- `gunzip legsim.tar.gz`
- `tar xf legsim.tar`

This creates a directory called `sim` containing:

[1]Available at `http://www-iri.upc.es/people/porta/legsim.tar.gz`.

- **Some makefiles:** The common part is in `makefile` and the parts depending on the machine are in `make.sun`, `make.sgi`). This second one is the one to be used as a base to personalize when working under Linux. In this case, some of the directory names that appear in the makefile should be modified (for instance, the `/usr/openwin/` directory should be changed to `/usr/X11R6/`)
- **Some scripts:** `compile` and `sim`. They are provided to easy the compilation and execution of the simulator with different robot controller. Their use is explained below.
- **The main program:** It is named `lrs` for Legged Robot Simulator. This program is invoked from `Geomview` as an external module and implement the main loop of the simulator as well as the user interface management. If you run it directly you will obtain the same user interface, no graphic output, and a long sequence of geometric commands appearing in the standard output (these are the ones received and processed by Geomview during a normal execution). In the distributed package, this program is no present since it would be generated the first time the simulator is compiled.
- **Simulator configuration file:** `sim.cfg`. In indicates which ground, robot and set of landmarks to load at startup.
- **The `grounds` directory:** Containing ground descriptions. Its syntax is described in section 2.2.1.
- **The `robots` directory:** Includes robot descriptions. Its syntax is described in section 2.2.3.
- **The `landmarks` directory:** Includes pre-defined sets of landmarks. Its syntax is described in section 2.2.2.
- **The `obj` directory:** Used to store the objects resulting form the compilation. Deleting the files in this directory (for instance executing `make clean` in the command line) forces the complete re-compilation of the simulator code.
- **The `doc` directory:** Contains this document.
- **The controller directories:** Each user programmed controller is stored in a different directory. Two controllers are provided with the simulator: `user_skel` (that is just an empty controller) and `user_tripod` that contains the controller described in section 3.2.17.

To compile a new controller you can use:

`compile` *directory_name*

where *directory_name* is the name of the directory containing the controller to be compiled. For instance you can type

`compile` *user_tripod*

to compile the simulator with the controller described in section 3.2.17.

To run the simulator just type

```
sim
```

in the command line.

Chapter 2

User Interface

This chapter is a brief description of how the user can interact with the simulator. There are two form of possible interaction. The first one is on-line and consist in controlling the execution the simulator: starting/stopping the robot controller, changing the point of view over the robot, changing the robot, changing the ground, ... The other kind of user intervention with the simulator is off-line and consists in setting different configuration files (ground, robot, landmarks) and selecting which ones to load at startup[1].

2.1 The On-Line Interface

Once the simulator is started (moving to the simulator directory and typing `sim` in the command line) three window appear in the screen (see figure 2.1).

The largest window is the *robot display window*. This window shows the view of a camera looking to the world in which the robot is supposed to move. The robot display window includes the ground (the large square inside the window) that in the example figure is a flat surface, the robot (in the middle of the window), and, optionally, a set of landmarks (the small balls around the robot). Initially the camera show a top view of the world and the robot, including all the ground. This can suppose to draw a very small robot (some times almost invisible) if the ground is large. The position of the camera can be changed using a set of tools described below.

The other two windows displayed at start up are the *simulator main panel* (described below) and the *robot camera window*. Of course, this last one is only displayed if the robot is equipped with a camera. The position of this camera with respect to the robot is defined in the robot configuration file and its orientation (pan and tilt) can be changed either from the user controller or from the on-line interface.

2.1.1 The Main Panel

The *simulator main panel* has six control buttons, the view interval control dialog, and two sub-menus (the robot and the world ones described in the following sections). Each one of the control buttons have the following function:

[1]However, all of them can be changed on-line if necessary.

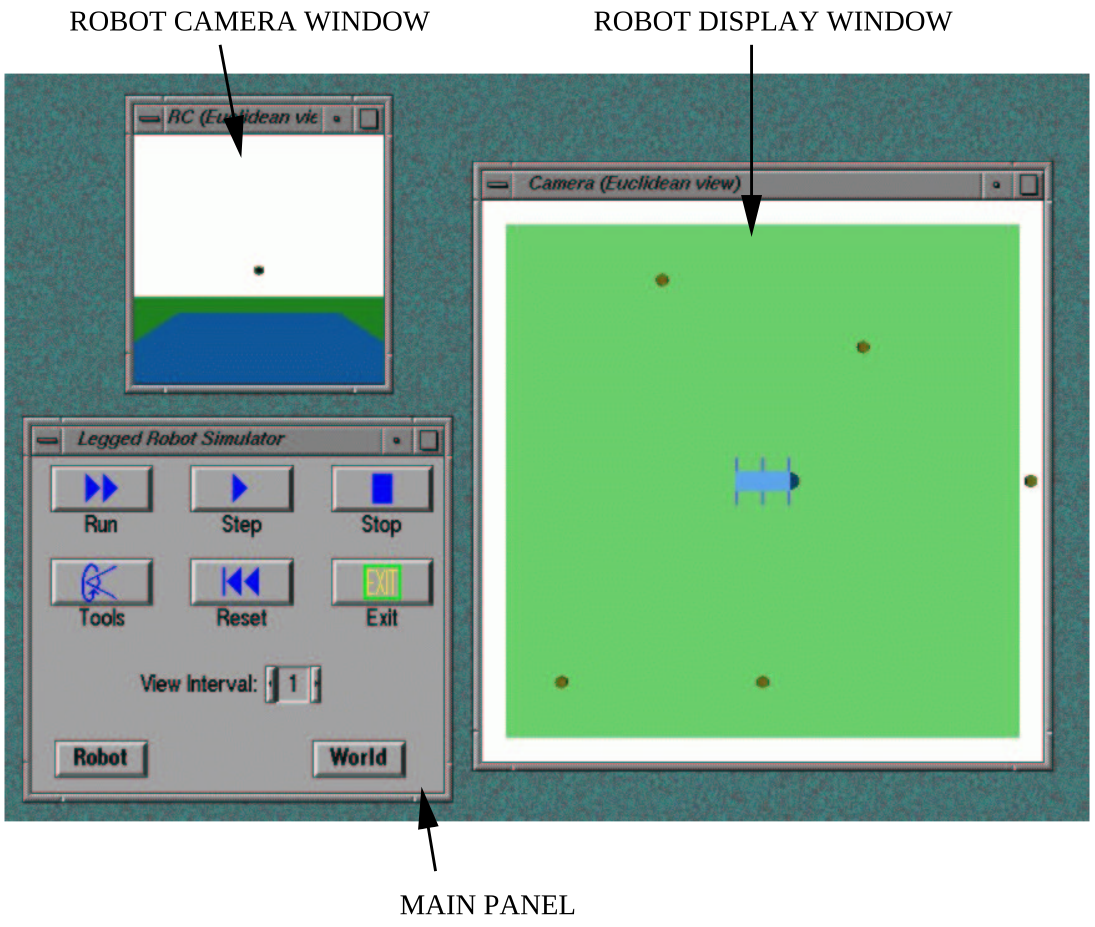


Figure 2.1: *The simulator initial appearance.*

- **Run**: Allow the execution of the user defined controller. From the moment this button is pressed the simulator executes the function `usr_step` once each time slice and after its execution, the robot model is updated (and displayed if required).
- **Step**: Executes once the user defined controller and the robot and world update. This button is useful to monitor the workings of the robot controller in detail. If this button is pressed, the visualization interval is set to 1 so all frames are displayed.
- **Stop**: Stops the execution of the user defined controller (and so, stops the robot).
- **Tools**: Show a panel (see below) to select tools to move the camera that is focusing the world and the robot.
- **Reset**: Moves the robot (and the camera looking to it) to the initial position.
- **Exit**: Closes the simulator.

Pushing the tools button opens the motion tools panel (figure 2.3). This panel allow us to choose between three tools to change the point of view over the world and the robot:

- **Zoom**: Approach or moves away the world to/from the camera.

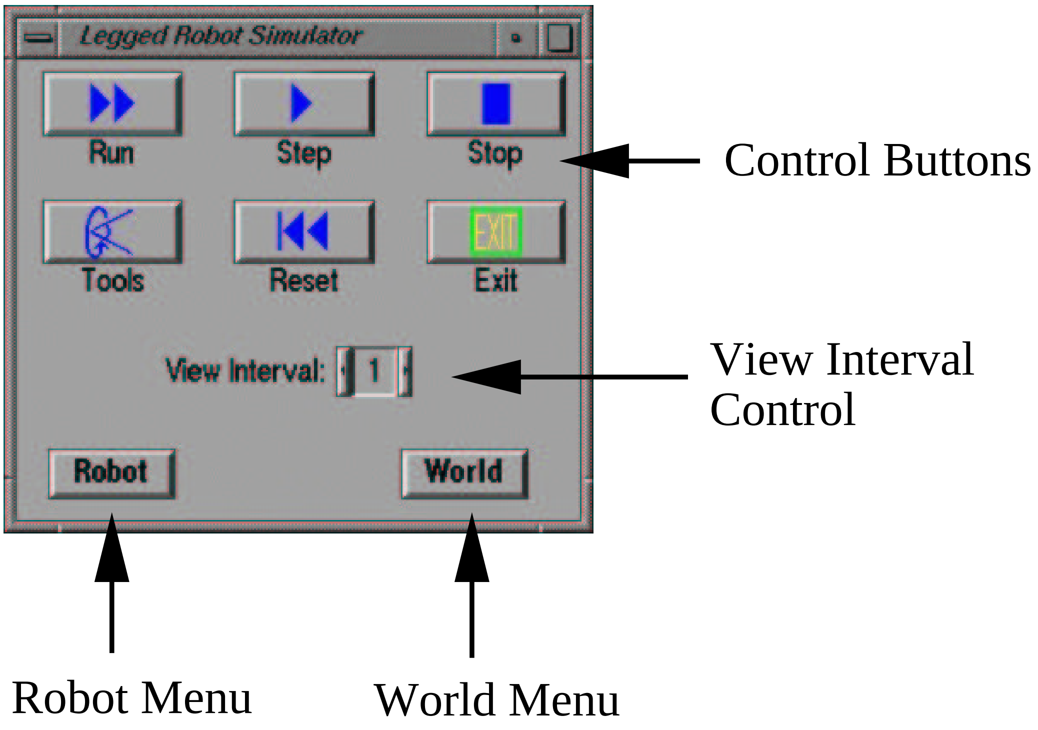


Figure 2.2: *The main panel.*

- **Fly**: Translates the world up and down of left and right (as seen from the camera).
- **Orbit**: Rotates the world.

Once one command is selected (the corresponding button is shown pushed) we can move the camera by pushing the left mouse button over the robot display window and moving the mouse. The movements produced in this way have inertia (they continue even when the mouse button is released). To stop a movement just click with the mouse over the robot display window.

The three provided commands are a subset of the commands provided by Geomview. The rest of Geomview commands can also be used by typing the appropriate shortcuts (see the Geomview manual for details).

Observe that while the robot display window and the robot camera window are provided by Geomview, the main panel and the rest of sub-panels have been designed and implemented by our selves. We have use the Xform library and the source codes are provided with the simulator. Consequently, they can be modified to fit your preferences. However, this could be a hard work and we hope the default interface to be useful enough. May be the strongest restrictions of the provided interface is that we limit the robots to have at most 8 legs.

2.1.2 The Robot Menu

The robot menu groups the set of functionalities that allows to configure and control the robot. The options of this menu are:

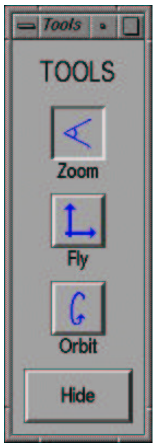


Figure 2.3: *The tools panel.*

- Move leg: To increment/decrement the position of a leg.
- Set angles: To change the angle of each articulation of each leg.
- Set positions: To query/modify leg positions.
- Balances: The balance query and gesture execution.
- Sensors: To get the values of the sensors.
- Move Robot: To move the robot in the world.
- Load Robot: To load a new robot model.
- Legend: To show a legend explaining the color code used to display the robot.
- View: To move the camera to some predefined positions.
- Traces: To manage the traces (if any defined).
- Camera: To control the robot's camera.

Once an option of this menu is selected a sub-panel would appear. We describe the different sub-panels in sections 2.1.3 to 2.1.13.

2.1.3 Move leg Panel

This sub-panel allows us to move each leg individually in the different Cartesian dof. The components X, Y and Z can be incremented or decremented by pushing the corresponding button (+ to increment and - to decrement). The increment/decrement step is fixed and set to 1 unit[2]. On the left of the panel there are two selectors to choose the leg to move and the frame of reference to which the X, Y and Z coordinates are referred. Two frame of references are available (see figure 2.5), the body frame of reference (attached to the robot's body and common to all legs) and the leg frame of reference (attached to the leg hip and particular for each leg). The leg frame of reference result from a rotation around the Z axis and a translation along axes X, Y, and Z of the body frame of reference so that the leg hip is reached. The button `Hide` hides the panel.

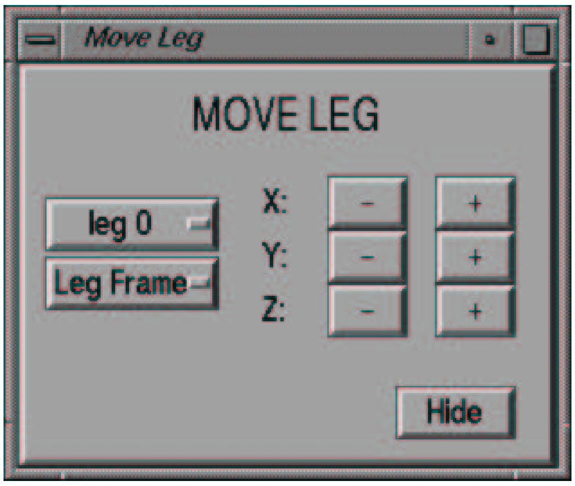


Figure 2.4: *The move leg sub-panel.*

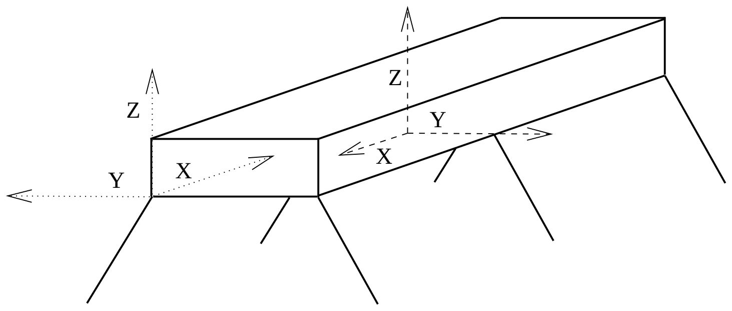
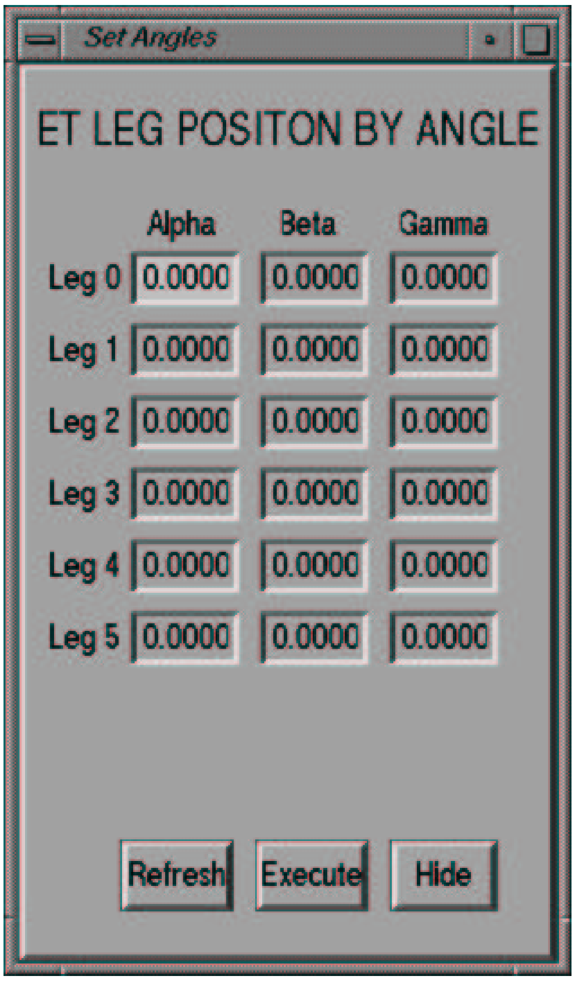


Figure 2.5: *The body frame of reference (dashed lines) and a leg frame of reference (dotted lines).*

[2]All measures used in the simulator are referred to a base unit not specified and that can be centimeters, inches or whatever the user prefers

2.1.4 Set angles Panel

This sub-panel allows to change the position at which each joint of each leg is. Three angles for each leg can be modified: α, β, and γ (see section 2.2.3 for details about this angles and its bounds). We can change an angle by directly typing it in the corresponding text square and pushing the `Execute` button. The `Refresh` button updates the values of the angles and `Hide` button hides the panel.



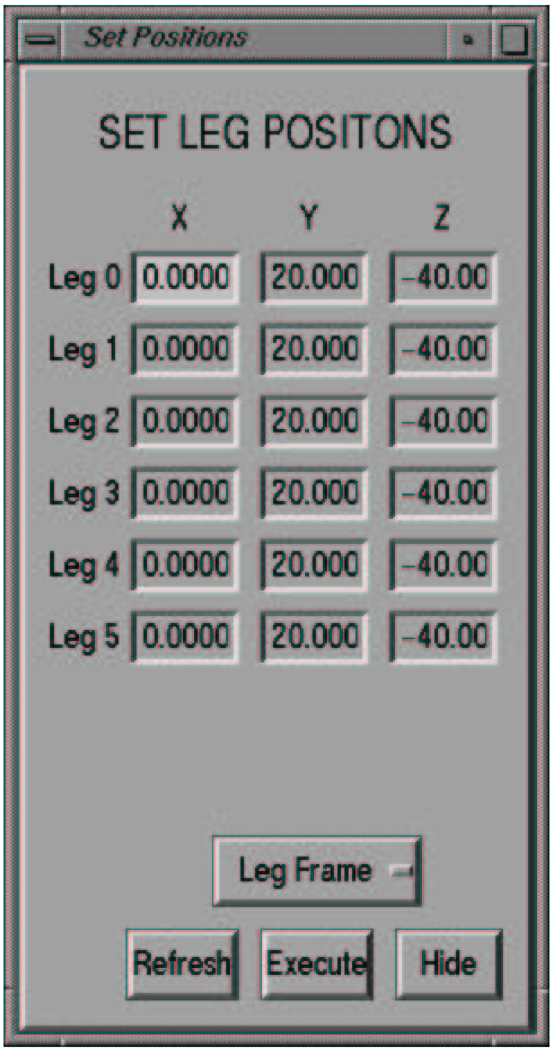
ET LEG POSITON BY ANGLE

	Alpha	Beta	Gamma
Leg 0	0.0000	0.0000	0.0000
Leg 1	0.0000	0.0000	0.0000
Leg 2	0.0000	0.0000	0.0000
Leg 3	0.0000	0.0000	0.0000
Leg 4	0.0000	0.0000	0.0000
Leg 5	0.0000	0.0000	0.0000

Figure 2.6: *The set angles sub-panel.*

2.1.5 Set positions Panel

This sub-panel provides a way to query leg positions and an alternative form to move each leg. Leg positions can be queried by selecting a frame of reference: leg, body, world, or reference (see section 3.2.1 for details on those frames of reference). In general, the positions are automatically updated but you can press the `Refresh` button to ensure that you are viewing the actual positions. In the body and leg frame of references, leg positions can be changed by typing the new positions in the corresponding text boxes and pressing the button `Execute`. Finally, the `Hide` button hides this panel.



	X	Y	Z
Leg 0	0.0000	20.000	-40.00
Leg 1	0.0000	20.000	-40.00
Leg 2	0.0000	20.000	-40.00
Leg 3	0.0000	20.000	-40.00
Leg 4	0.0000	20.000	-40.00
Leg 5	0.0000	20.000	-40.00

Figure 2.7: *The set position sub-panel.*

2.1.6 Balances Panel

This section surveys a set of panels that allows to query the current balance values and to execute gestures to modify them.

A balance is a measure of the position of the robot's body with respect to feet positions along a given dof. These measures are evaluated by comparing each leg position at a given moment with respect to a reference position that is at a fixed place with respect to the body.

The difference of the position of a single leg with respect to its current reference position along a given dof is called the *tension* of this leg. There is a physical analogy that can illustrate the meaning of the tension. If we imagine that each leg is attached to the reference position by an ideal spring, then the tensions correspond to the forces and moments that the spring would exert on the robot's body.

Formally, if $p^i = (p^i_x, p^i_y, p^i_z)$ is the current position of a leg and $r^i = (r^i_x, r^i_y, r^i_z)$ is its reference position, we can compute as many different tensions as possible dof in the Euclidean space (i.e., Tx, Ty, Tz, Rx, Ry, Rz):

$$
\begin{aligned}
Tension^i_{Tx} &= p^i_x - r^i_x \\
Tension^i_{Ty} &= p^i_y - r^i_y \\
Tension^i_{Tz} &= p^i_z - r^i_z \\
Tension^i_{Rx} &= p^i_z r^i_y - p^i_y r^i_z \\
Tension^i_{Ry} &= p^i_x r^i_z - p^i_z r^i_x \\
Tension^i_{Rz} &= p^i_y r^i_x - p^i_x r^i_y
\end{aligned}
$$

A balance measure is the sum for all legs of the tension in a given dof.

Normally, the reference position of a leg should be the central position of its workspace (the one reached when all motor angles are zero) but the user can select another initial reference position and he can change the reference from the robot controller using the appropriate set of functions (see section 3.2.7).

Obviously, tensions (and so balances) are different according to frame of reference to which leg and reference positions are referred (it is like computing the resulting force exerted by the virtual springs at different points). In our simulations we use a special frame of reference located at the center of gravity of the polygon conformed by the reference positions (see figure 3.2) since, in this way, tensions are independent of measures as for instance the height of the robot.

A gesture is a coordinated movement of legs in one dof consisting in applying the same transformation matrix to a given set of legs. The execution of a gesture (as any leg movement) produce a modification of the tensions values.

The balance panel (figure 2.8) shows the current balance value in the six possible dof. The buttons + and - for each dof allow to execute positive and negative gestures in the corresponding dof. The executed gestures have a fixed magnitude (0.15 units for translations and 0.15 degrees for rotations). After a gestures is executed, the balance values are updated. If this is not the case, the `Refresh` button can be pushed to update them. The small top buttons allow to select the set of legs that are affected by gestures.

The selector just under the balance values allow to choose whether the gestures have effect of the robot's body or not. In general an infinite friction between legs and ground is simulated and when legs are moved using gestures, the result is that legs keep in the same position as they are and the robot's body is displaced. When we choose the gestures to have no effect on the body, then no friction is simulated and legs slip on the ground without displacing the robot's body.

The `Reference` button shows a sub-panel with the current values of the reference positions expressed in the robot frame of reference (remember that leg positions can be queried using the panel described in section 2.1.5). The references panel can be close by pushing the `Hide` button.

The `Tension` button shows a sub-panel with the current tensions for all legs and dof. As usual, in this panel, the `Refresh` update the tensions values and the `Hide` button closes the panel.

Finally, the `Hide` button closes the balance panel.

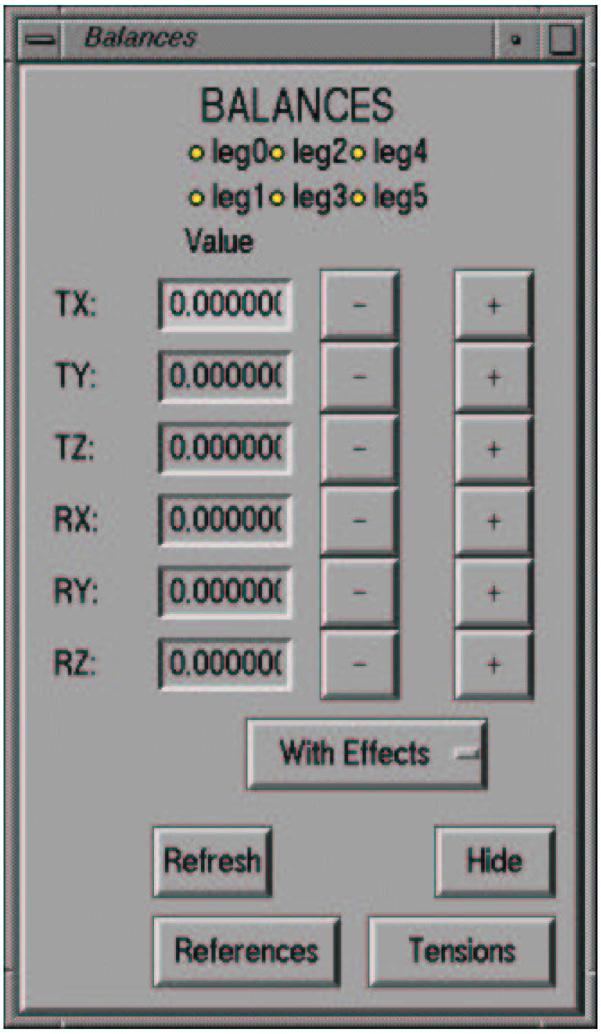
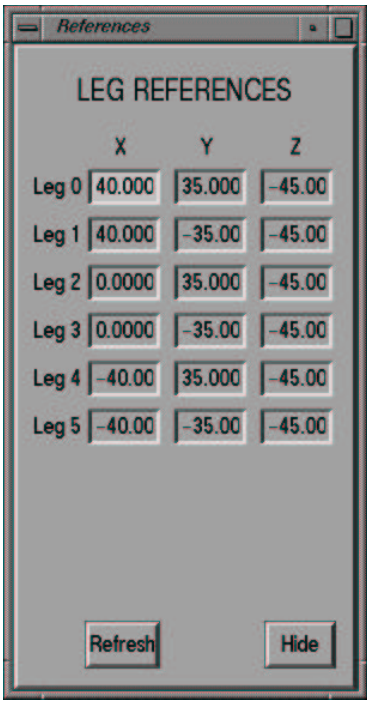


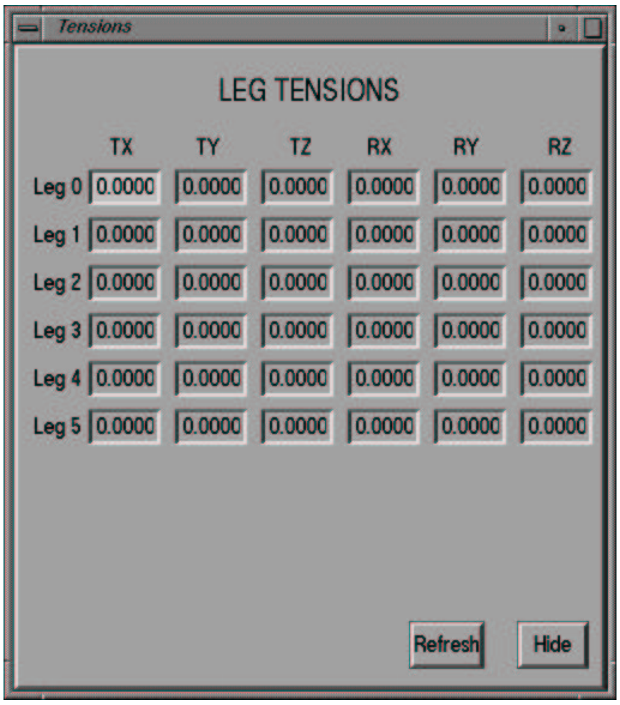
Figure 2.8: *The balances sub-panel.*



LEG REFERENCES

	X	Y	Z
Leg 0	40.000	35.000	-45.00
Leg 1	40.000	-35.00	-45.00
Leg 2	0.0000	35.000	-45.00
Leg 3	0.0000	-35.00	-45.00
Leg 4	-40.00	35.000	-45.00
Leg 5	-40.00	-35.00	-45.00

Figure 2.9: *The references sub-panel.*



LEG TENSIONS

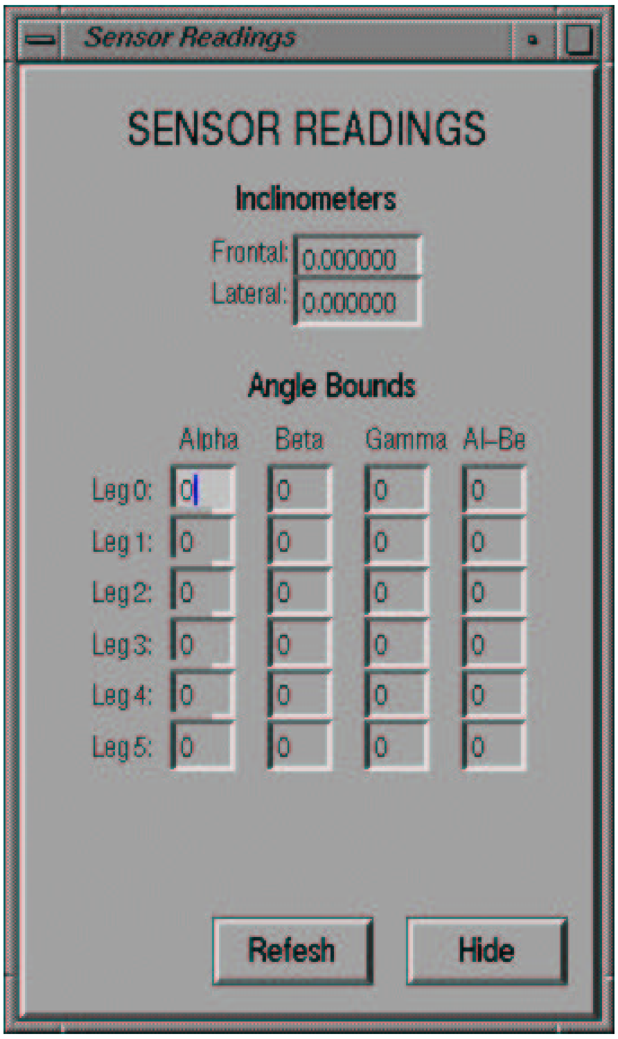
	TX	TY	TZ	RX	RY	RZ
Leg 0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Leg 1	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Leg 2	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Leg 3	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Leg 4	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Leg 5	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Figure 2.10: *The tensions sub-panel.*

2.1.7 Sensors Panel

This panel shows the values of the robot sensors (except feet touch sensors whose value are directly indicated by the corresponding leg color, see section 2.1.10, and for the robot camera whose output is drawn in an always visible private window). Up to now the robot is equipped with two inclinometers (one along the robot's body and the other transversal to it) and sensors to detect when a leg reaches the end of the range of any of its joints. Those sensors are active if the angular distance between the joint position and the end of its range is below 1 degree. There is one sensor for each one of the three joints of each leg (α, β, and γ) and an additional sensor to monitor possible restrictions that involve both α and β. See section 2.2.3 for a detailed description about the leg configuration.

The `Refresh` and `Hide` buttons perform the usual tasks.



SENSOR READINGS

Inclinometers

Frontal: 0.000000
Lateral: 0.000000

Angle Bounds

	Alpha	Beta	Gamma	Al-Be
Leg 0:	0	0	0	0
Leg 1:	0	0	0	0
Leg 2:	0	0	0	0
Leg 3:	0	0	0	0
Leg 4:	0	0	0	0
Leg 5:	0	0	0	0

Figure 2.11: *The sensors sub-panel.*

2.1.8 Move Robot Panel

This panel provides tools to displace (translate and rotate) the robot in the world. These tools are useful to position the robot in a given position to initiate a gait. The `Legs to Ground` button adjusts leg positions so that all of them are in contact with the ground. Of course, this is only done if the ground surface is inside the leg workspace. If this is not the case the leg is not moved and it is task of the user to move the robot so that the ground surface becomes reachable.

The usual `Hide` button to close the panel is also provided.

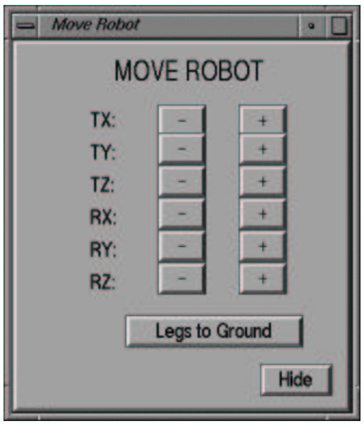
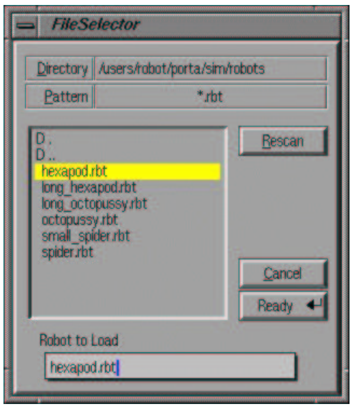


Figure 2.12: *The move robot sub-panel.*

2.1.9 Load Robot Panel

When this option is selected the Xform standard file browser (see figure 2.13) is shown and the user can select a new robot description file. If an inexistent file name is entered nothing is done. If the selected file contains an invalid robot description then the simulator can crash or enter in an infinite loop.



Directory /users/robot/porta/sim/robots
Pattern *.rbt

```
D .
D ..
hexapod.rbt
long_hexapod.rbt
long_octopussy.rbt
octopussy.rbt
small_spider.rbt
spider.rbt
```

Robot to Load
hexapod.rbt

Figure 2.13: *The load robot sub-panel.*

2.1.10 Legend Panel

This option shows an image to explain the code of colors used to display the robot.

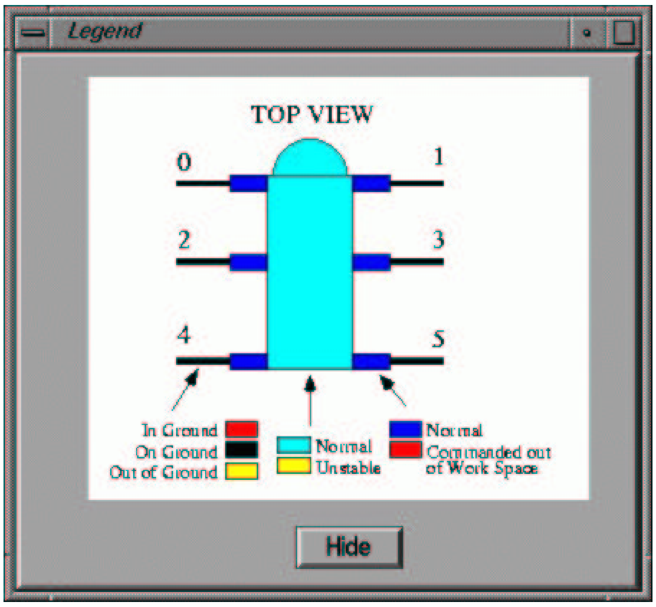


Figure 2.14: *The legend robot sub-panel.*

2.1.11 Change View Panel

This panel give the user the change to move the camera focusing the robot in the world to some useful predefined positions: top view of the world (Z+), view from the left (Y+), front view (X+), ...

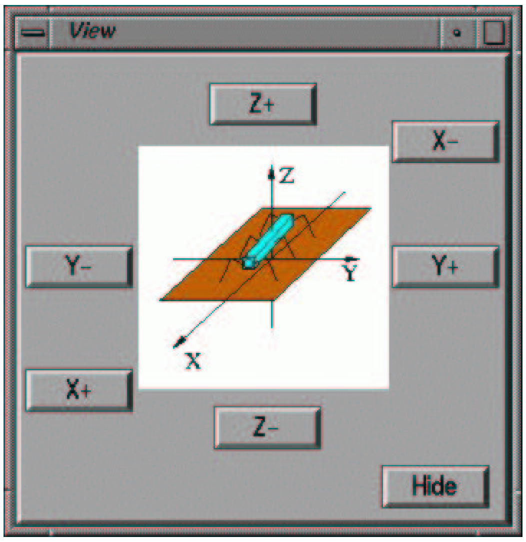


Figure 2.15: *The change view sub-panel.*

2.1.12 Traces Panel

This panel gives tools to manages the user-defined traces. A trace is a polygonal line whose points can be defined by the user in successive time slices. A trace is drawn as a set of balls of the same color and size connected by lines. Traces are intended to monitor the displacement of any part of the robot. They can be used to mark the path followed by the robot, the future passing points for the robot, the turning centers of the robot path, the footholds used by a given leg, ... Traces are defined and enlarged from the robot controller. This panel allow to show/hide each trace as well as to reset them but not to define new traces or delete existing traces.

The information related to each trace is given in a single line that includes the trace name. Usually traces are not shown because this slows the workings of the display engine of the simulator. To show a trace the user have to activate the small round button on the left of the corresponding trace name. The `Reset` button on the right of the trace name reset the trace (i.e., deletes all the points added to the trace until this moment).

The `Hide` button closes the trace sub-panel.

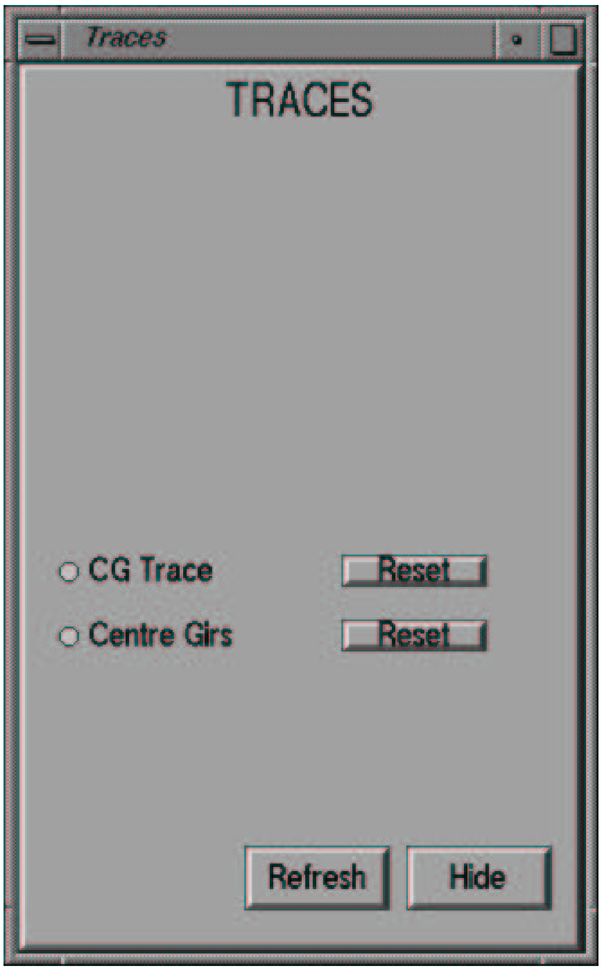


Figure 2.16: *The traces management sub-panel.*

2.1.13 Camera Panel

This panel shows the pan an tilt of the robot's camera and provides tools to modify them. Of course, this panel is only available if the robot has camera.

The two sliders can be used to modify the pan and the tilt. The `Refresh` button updates the current values of the pan and tilt of the camera. It can be necessary to push it to view the value this variables when the pan/tilt are changed not from this control panel. The `Reset` button sets the pan and the tilt to zero. Finally, the `Hide` button closes the trace sub-panel.

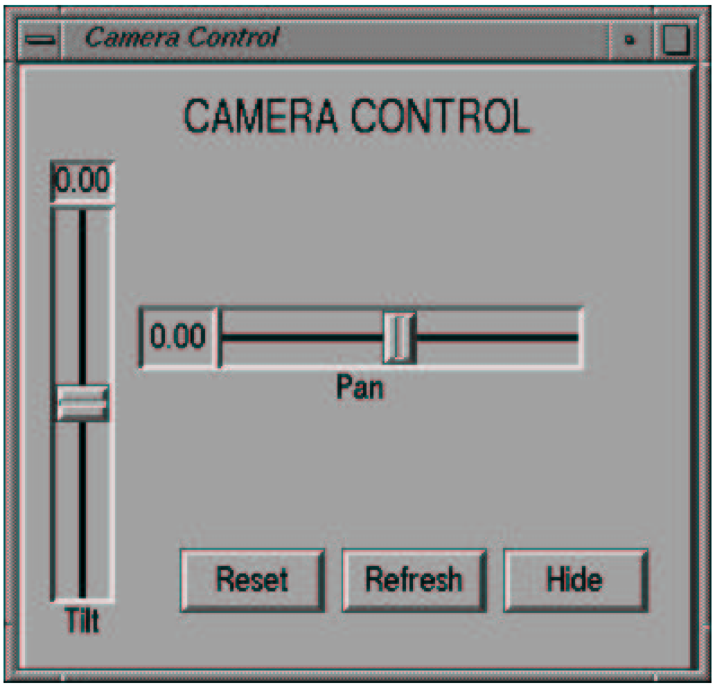


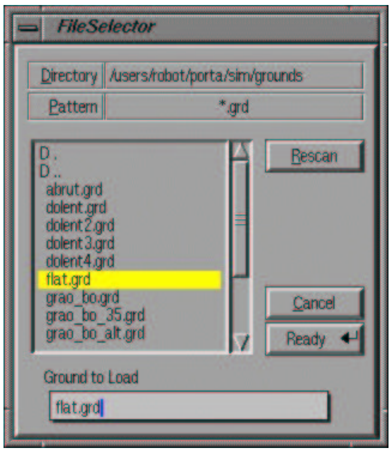
Figure 2.17: *The camera control sub-panel.*

2.1.14 The World Menu

The options of this menu affect the world in which the robot moves. As explained, the world object includes both the ground and a (maybe empty) set of landmarks.

2.1.15 Load Ground Panel

This option enables a sub-panel to select a new ground file. The ground file syntax is described in detail in section 2.2.1. The simulator includes some examples of grounds and a small utility to generate new grounds.



Directory: /users/robot/porta/sim/grounds
Pattern: *.grd

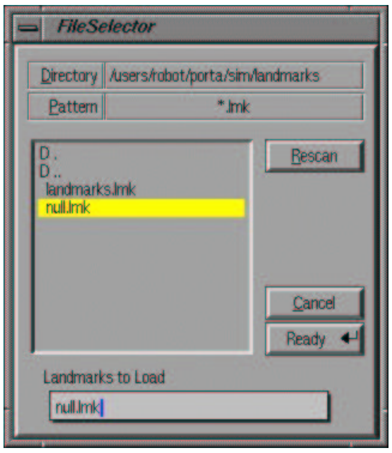
```
D .
D ..
 abrut.grd
 dolent.grd
 dolent2.grd
 dolent3.grd
 dolent4.grd
 flat.grd
 grao_bo.grd
 grao_bo_35.grd
 grao_bo_alt.grd
```

Ground to Load: flat.grd

Figure 2.18: *The load ground sub-panel.*

2.1.16 Load Landmarks Panel

This option shows a sub-panel to select a new set of landmarks file to replace the current one. See section 2.2.2 for more details about landmarks files.



Directory: /users/robot/porta/sim/landmarks
Pattern: *.lmk

```
D .
D ..
 landmarks.lmk
 null.lmk
```

Landmarks to Load: null.lmk

Figure 2.19: *The load landmarks sub-panel.*

2.1.17 Save Landmarks Panel

This option shows a sub-panel to save the current set of landmarks. The displayed panel is similar to that of figure 2.19.

2.1.18 Manage Landmarks Panel

This option allows to add, delete, and move landmarks. The landmarks management panel (figure 2.20) displays the list of the current landmarks and provides the following tools:

- **New:** To create a new landmark. This button displays the sub-panel of figure 2.21 that allows to input the data for a new landmark: identifier and position. Once this data is input, pressing the button `Ok` adds the new landmark to the current set and closes the sub-panel. It is not possible to add a new landmark with the same identifier of an already existing landmark. The button `Hide` simply closes the panel.
- **Move:** This options opens the move landmark sub-panel (figure 2.22) in which a new position of the landmark can be input (the landmark identifier is shown but can not be modified). Again, pressing the button `Ok` enables the position modification and the button `Hide` closes the panel. The landmark to be moved is the one selected on the landmark list. A landmark can be selected with a single mouse click over its identifier.
- **Delete:** Deletes the currently selected landmark.
- **Mark:** Changes the color of the selected landmark from yellow (the default color) to red.
- **Unmark:** Changes the color of the selected landmark from red to yellow.

A double click over a landmark identifier opens a panel showing the information relative to the corresponding landmark (identifier and position). This panel is similar to the move landmark sub-panel of figure 2.22 (but in this case no data can be modified) and can be closed with the corresponding `Hide` button.

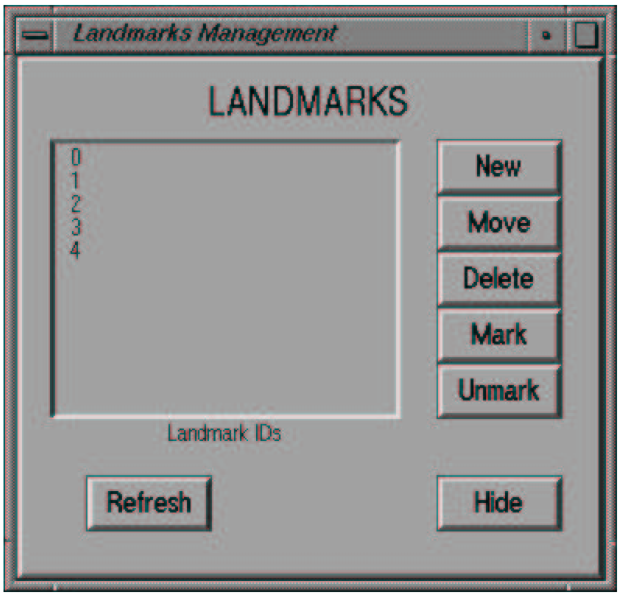


Figure 2.20: *The manage landmarks sub-panel.*

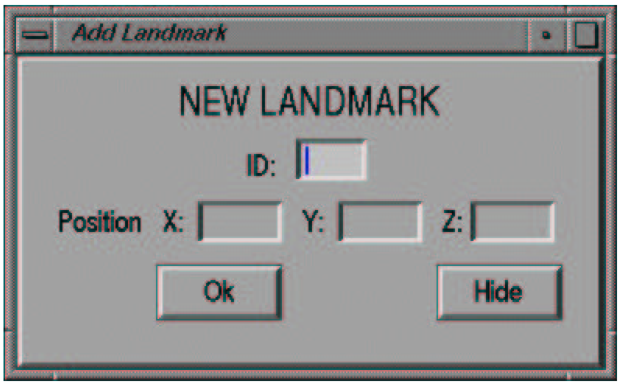
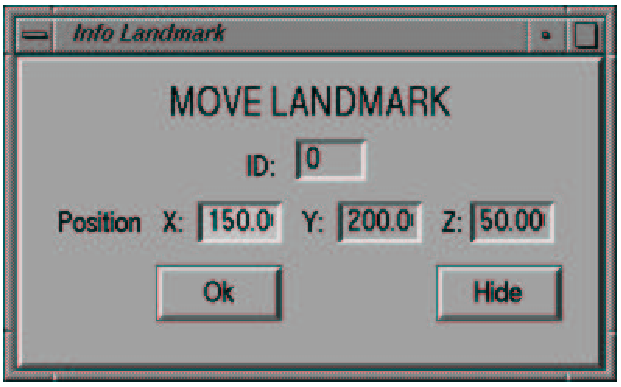


Figure 2.21: *The new landmark sub-panel.*



ID: 0

Position X: 150.0 Y: 200.0 Z: 50.00

Figure 2.22: *The move landmark sub-panel.*

2.2 The Off-Line Interface

The off-line interface of the simulator consists in a set of files that allow to define grounds, landmarks sets, robots, and robot controllers. How to program a robot controller is described in detail in chapter 3. In the following subsections the ground, the landmarks and the robot configuration file formats are described. In the file `sim.cfg` the user can indicate which ground, landmark, and robot files to load at startup. However, the on-line interface described before allow the user to load a new file of any of these types whenever required.

All this configuration files are text files in which information fields (numbers, positions, ...) are preceded by text descriptions ended by the symbol `':'`. The contents of the description field does not matters and can be changed to fit user preferences but the symbol `':'` should be there in all the cases to separate between text and input data otherwise the simulator would fail (i.e. crash or enter in an infinite loop) when reading the file.

2.2.1 Ground Configuration

The ground is simulated with a triangular mesh with control points regularly distributed in the X-Y plane of the world and symmetrically placed around the Z axis.

The ground configuration file contains a description of this mesh. First, we have to define the dimensions of the ground along the X and Y axis: `lx`, `ly`. The ground is placed in the rectangular area comprised between points

$$(-lx/2, -ly/2)$$

and

$$(lx/2, ly/2)$$

Then we have to define the density of the mesh. To do that we indicate how many tiles (`tx` and `ty`) are required to cover `lx` and `ly` distances. The number of control points to be placed along `lx` and `ly` are `tx+1` and `ty+1` respectively.

Finally, we have to indicate the height of the mesh at each control point. Figure 2.23 shows the a top view of a control grid for a ground definition. The numbers indicate the order at which heights should be input.

When loaded by the simulator the control mesh is triangulated in a trivial way (i.e., connecting each three contiguous vertexes as shown in figure 2.23) and it is displayed using a smoothing coloring technique so that it looks like a realistic ground.

The ground directory of the simulator includes the small utility `new_ground` that generates ground configuration files according to a template file named `ground.tpl` and with the user desired shape. The fields of the template file are the size of the ground to be generated, the density of the mesh, and the maximal height (Z coordinate). The result is a ground where heights are those given by the function `ground_function` defined in the file `source/ground_function.c`. This function can be modified to generate ground as required for each application. If it is modified, the `new_ground` program has to be re-compiled typing

```
make new_ground
```

in the command line once placed in simulator main directory.

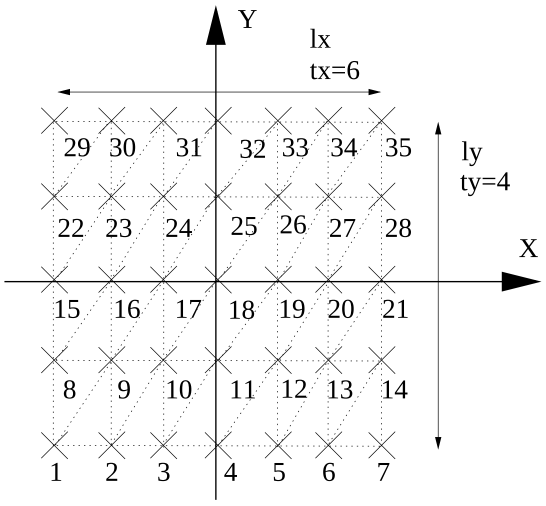


Figure 2.23: *Top view of the control grid of a ground definition.*

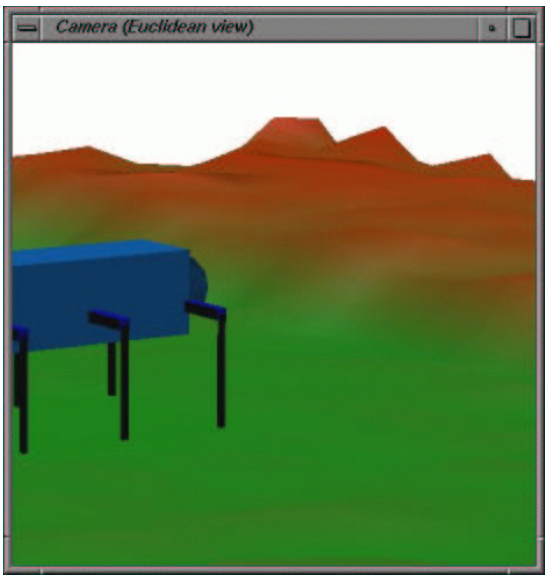


Figure 2.24: *The* `step.grd` *ground.*

Length x (lx): 80.000000
Length y (ly): 80.000000
Tiles x (tx): 30
Tiles y (ty): 30

23.74	13.11	18.62	10.94	6.66	0.31	2.06	19.15	14.81	0.32	8.03	10.03	15.28	10.68	2.81	0.46	3.36	7.11	10.30	11.41	7.10	16.22	37.97	38.75	40.67	30.60	25.06	29.42	48.64	46.53
2.07	18.593	20.83	5.68	5.41	17.04	0.64	3.05	10.22	15.80	2.32	1.13	14.08	3.31	3.19	9.84	12.6320		7.68	12.85	10.31	16.06	32.05	40.67	31.63	27.85	29.80	41.98	33.67	27.51
16.33	7.109	14.62	7.69	15.38	16.67	2.12	0.41	7.29	2.38	9.70	0.35	10.55	7.63	10.98	7.63	9.09	5.89	4.07	12.17	8.91	11.98	40.29	35.13	31.84	26.54	42.80	32.31	34.36	48.03
3.819	8.378	6.28	15.95	10.75	11.21	8.56	10.47	9.40	6.44	10.93	6.28	11.38	13.90	11.82	12.03	3.87	8.95	12.82	5.73	2.26	0.53	38.06	38.63	25.34	43.66	41.15	25.43	43.03	3.32
3.113	6.751	13.72	1.40	3.79	6.57	3.64	1.49	14.39	6.61	10.93	1.13	4.60	0.84	8.30	9.39	4.57	1.34	1.94	7.42	4.76	9.68	32.93	41.41	39.72	35.88	41.25	40.02	33.39	30.61
20.60	4.748	18.74	11.97	6.72	2.11	12.83	7.39	6.69	2.00	1.43	6.30	7.37	0.27	2.30	4.60	11.06	7.20	10.38	6.26	13.00	6.27	38.84	25.10	34.35	38.59	42.57	38.54	31.78	42.95
2.99	12.131	9.3619		4.24	0.34	2.70	0.09	8.03	5.30	11.10	0.41	0.18	5.43	3.65	1.02	8.60	1.52	7.90	5.55	9.66	10.34	29.04	37.84	37.56	40.01	39.27	38.02	35.07	26.55
0.56	5.127	17.36	0.43	10.30	4.04	8.80	1.18	1.73	6.70	4.23	6.50	7.12	8.04	8.21	0.16	3.12	5.15	0.71	2.50	9.86	11.74	27.31	29.29	26.40	39.82	34.30	41.50	38.48	34.83
8.348105		1.432512		7.46	3.20	3.10	0.37	1.10	8.39	6.69	6.70	3.96	3.34	2.73	1.19	6.99	5.26	0.01	5.01	5.95	0.40	28.48	25.35	36.49	32.54	37.20	31.77	33.69	28.76
9.83	6.549	3.28	0.26	7.74	2.39	4.84	4.19	4.78	7.40	2.75	3.64	6.25	6.93	4.60	0.98	5.68	1.63	0.45	6.63	4.45	3.23	29.53	3.70	34.36	37.41	31.24	31.01	31.44	40.81
15.24	4.557	11.21	0.85	12.19	12.81	5.29	0.06	7.06	5.08	2.11	4.99	4.23	4.95	4.13	2.03	0.22	1.01	1.77	6.17	5.50	8.38	32.65	35.96	25.31	35.46	35.61	27.27	36.81	29.27
6.80	11.243	4.37	5.34	1.74	4.45	2.34	0.23	1.82	6.55	6.04	2.10	0.30	1.88	4.18	4.03	0.46	4.28	4.33	1.96	2.90	0.50	26.23	35.07	32.75	34.50	27.36	37.61	39.20	36.01
7.79	9.569	4.94	7.28	4.50	1.68	7.00	8.31	1.64	0.54	4.89	0.39	0.81	1.86	0.66	3.04	3.02	2.45	2.37	1.92	0.74	0.91	24.28	25.13	27.29	3.62	28.78	33.08	33.85	25.01
17.05	1.738	10.91	1.39	9.45	7.95	2.08	7.63	6.40	4.18	5.31	4.45	2.55	0.00	0.00	0.00	0.00	2.40	0.37	3.31	2.24	1.16	27.43	25.09	25.48	29.91	35.11	39.70	30.30	28.63
3.66	5.633	6.6581		2.86	5.27	1.34	0.22	6.89	2.60	4.44	0.87	2.34	0.00	0.00	0.00	0.00	2.38	1.10	2.21	5.56	1.70	3.04	3.78	26.60	31.26	27.85	27.50	33.47	26.94
15.66	1.582	9.17	12.90	3.52	6.93	6.48	5.38	1.38	4.39	3.46	1.39	2.15	0.00	0.00	0.00	0.00	2.70	1.68	2.82	4.28	5.06	3.05	30.49	28.68	25.21	38.05	39.76	29.91	3.97
4.24	15.255	6.44	5.25	11.20	3.86	6.61	2.83	7.53	2.33	5.13	3.00	2.15	0.00	0.00	0.00	0.00	0.98	3.75	3.39	6.21	2.74	25.62	27.00	30.52	25.91	37.89	39.10	32.95	37.88
16.92	11.747	1.16	0.94	5.02	4.06	6.61	5.02	7.73	3.95	5.80	4.92	1.97	2.87	1.05	0.78	2.31	1.75	5.16	2.21	6.38	7.19	24.49	28.08	32.83	35.40	29.16	38.02	34.07	30.84
12.98	2.168	0.88	4.40	4.07	9.72	8.79	8.23	0.85	2.98	4.25	4.35	1.52	4.40	0.99	4.01	0.21	2.05	3.53	2.72	0.12	0.33	32.59	32.20	34.35	27.45	27.68	29.19	25.38	3.30
6.51	14.373	14.29	14.63	4.09	0.46	10.79	8.11	4.98	3.43	2.01	1.80	4.89	3.42	0.74	2.10	0.79	1.86	6.50	3.03	2.27	7.08	26.76	32.47	27.09	3.35	27.82	35.89	29.49	31.65
6.86	1.315	16.19	6.31	4.84	1.58	9.76	0.19	7.15	8.57	5.28	4.01	4.63	2.73	2.81	3.01	3.02	6.39	0.15	2.42	6.16	10.04	25.95	34.36	26.95	3.59	32.73	30.28	26.27	38.32
16.49	17.418	3.96	2.76	3.08	9.47	8.65	8.93	4.23	3.63	1.11	7.13	5.54	8.10	6.62	4.74	2.80	0.78	3.18	9.61	6.89	4.67	30.97	37.24	38.43	28.70	26.53	31.25	30.76	43.26
11.43	0.846	0.74	9.88	3.70	8.77	9.44	1.79	11.15	7.23	1.05	0.85	2.82	0.68	5.31	3.80	3.49	5.81	0.96	9.74	7.42	5.24	34.99	35.83	25.81	38.07	35.06	27.79	39.75	36.42
17.06	3.668	17.34	4.11	6.73	8.34	2.82	12.31	7.93	1.72	4.19	11.02	7.29	4.68	0.23	1.42	2.26	10.62	4.57	6.60	10.30	9.44	3.96	34.01	27.62	38.49	38.45	38.09	30.19	39.68
1.45	8.085	16.51	12.53	0.26	4.5248		12.19	4.35	11.57	9.90	0.22	5.70	8.51	2.67	5.18	6.57	10.70	5.36	4.27	5.19	6.63	28.83	40.12	36.34	26.97	35.70	37.37	27.33	32.61
20.09	17.221	18.26	5.50	14.04	9.09	1.22	6.65	2.82	6.36	0.92	6.30	9.97	7.04	4.12	11.40	3.88	1.30	5.3314		7.42	11.87	37.30	35.01	25.59	29.43	32.59	36.48	42.94	45.48
4.62	0.197	3.06	14.96	1.09	9.11	12.70	11.06	5.56	0.07	10.3248	3.818		9.60	0.26	0.01	7.89	9.95	5.17	9.76	2.38	15.76	40.42	35.02	38.07	41.30	40.88	25.55	37.27	37.42
15.32	11.357	1.03	6.71	11.31	11.94	10.14	4.73	9.21	14.41	3.82	8.25	3.11	0.63	1.97	4.72	3.39	14.99	7.21	0.38	3.97	10.90	29.39	35.92	28.93	40.60	36.91	29.74	27.01	43.76
11.97	14.167	2.00	12.98	7.82	3.13	10.13	4.95	3.67	0.15	2.42	11.34	10.50	3.68	13.28	14.13	0.02	7.49	14.84	9.93	12.11	1.75	32.50	41.31	29.62	32.69	35.41	35.07	36.03	29.15
12.88	3.044	9.22	8.86	0.83	19.88	8.15	11.94	17.57	5.64	1.55	1.69	3.83	11.97	4.94	4.3688		4.20	11.18	10.83	12.36	5.78	28.22	42.76	45.25	28.64	40.29	40.47	28.45	3.30

Figure 2.25: *The contents of the* `step.grd` *ground configuration file.*

```
Number of Landmarks:  5

Id:  0
Position:  150 200 50

Id:  1
Position:  400 0 50

Id:  2
Position:  0 -300 50

Id:  3
Position:  -300 -300 50

Id:  4
Position:  -150 300 50
```

Figure 2.26: *A landmarks configuration file.*

2.2.2 Landmark Configuration

The landmark configuration file has a simple syntax. We only have to indicate how many landmarks are included in the file and describe each landmark. To define a landmark we only have to give and identifier (a number) and the landmark position (x, y, z) in world coordinates. In figure 2.26 you can see an example of landmarks configuration file.

2.2.3 Robot Configuration

A robot description consist in five separate sections:

1. Body description.
2. Head description.
3. Camera description.
4. Leg description: That includes the leg positioning and the individual leg setup (bump tolerances, dimensions, and joint limits).
5. Leg reference description.

The body description

To describe the shape of the robot's body we have to determine a polygon in the X-Y plane of the robot's (Figure 2.27) and two points along the Z axis (Z min and Z max) that determine the bottom and the top surface of the body.

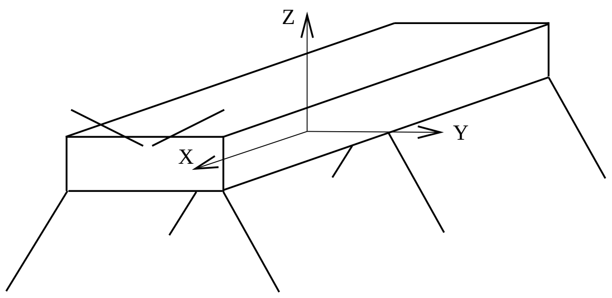


Figure 2.27: *The frame of reference attached to the robot's body.*

The head description

We can add a ball (that represents a head) to the robot's body. We have to indicate whether or not the robot has head and if it is the case where is to be placed the head, and its size (i.e., radius)

The camera description

The robot can be equipped with a camera. The camera description section of the robot configuration file includes fields to indicate whether or not the robot has camera, where to place the camera (in the robot's frame of reference), and the vision angle of the camera.

The camera is always placed looking towards the positive X axis but is orientation can be changed from the robot controller using the appropriate set of functions or from the on-line user interface.

The vision angle can any form 0 (the camera field becomes a single plane) to 180 (the camera field includes all the surroundings of the robot). Unfortunately, too small or to large vision angles are not accepted by Geomview and the robot camera window is not adapted to them. Although the visualization is not the expected, the internal workings of the simulator are the correct ones for all angles.

Legs description

All robot's legs are instances of a single mechanism (the well known pantograph mechanism [1]) but the simulator allow to fix the parameters of this mechanism for each particular robot.

Figure 2.28 shows a schematic representation of a leg with its attached frame of reference. The three controllable dof of the leg correspond to the angles α, β, and γ. α is the angle between segment 1 and the axis Y, β is the angle between segment 2 and the vertical, and γ corresponds to the rotation angle around the Z axis. Observe that if we modify angle α segment 1 is moved but segment 2 is also move so that angle β is maintained.

In the leg positioning section of legs description we have to indicate the point at which legs are attached to the body and the leg orientation. Leg placements must be given

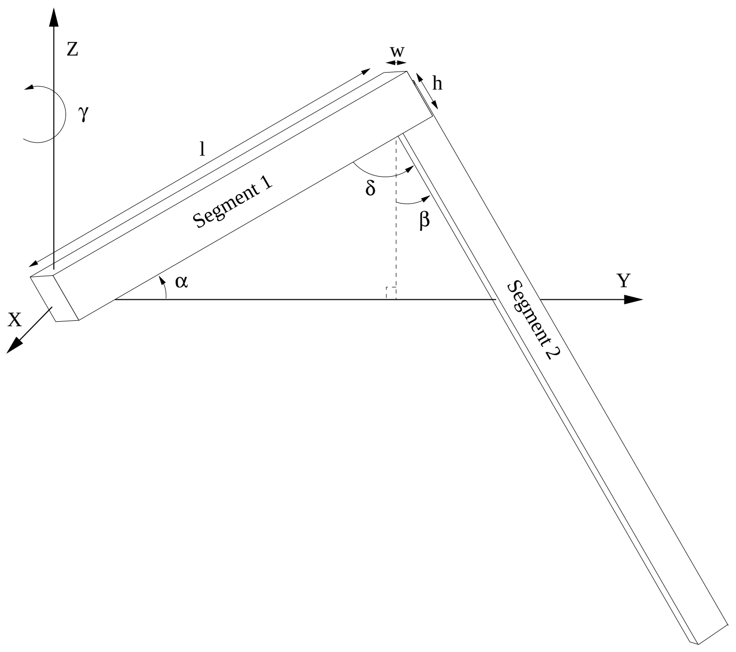


Figure 2.28: *The leg structure. Angles point toward its positive sense.*

clockwise (from a top view of the robot) starting from any leg and for each each one of them we have to indicate:

- An identifier: Just a number from 0 to $number\_of\_legs - 1$ to refer to the leg. This is the leg number used in all the robot functions described in next chapter.
- An attaching point: The (x, y, z) coordinate of the robots body to which the leg is attached.
- A rotation angle around the Z axis of the robot along which segment 1 is aligned.

After the global description of the leg arrangement we have to include some configuration parameters that are used for all legs. First, we have to indicate the leg bump tolerance. This parameter is used to determine whether a leg is touching of inside the ground. This measure is suppose to correspond to the passive adaptation capacity of the leg. Next we can can indicate the sizes of each leg segment. For each segment three variables can be set: the length (l in figure 2.28), the wide (w),and the height (h). The last leg configuration subsection includes restrictions to the ranges of each dof (angles α, β, and γ). There is an additional restriction that can be imposed over the angle between segments 1 and 2 (angle δ in figure 2.28).

The leg reference description

In this section of the robot configuration file we have to choose the reference position for each leg. This position is given in the leg frame of reference (and so independently of the particular leg positioning). These reference positions are the ones initially used to evaluate the tensions and balances (see sections 2.1.6) but the user can change them from the robot controller using the appropriate set of functions (see section 3.2.7)

Examples of robot configuration files

Here we provide two robot configuration files following the syntax described before. The first one corresponds to a long hexapod robot and the second one to a spider-like hexapod robot. Both files are included in the simulator package.

```
Robot Setup (all angles in degrees)
   Body Setup
      Number of points:  4
      Points (x y):
         40 -15
         -40 -15
         -40 15
         40 15
      Z min:  -15
      Z max:  15
   Head Setup
      Has head?  (0/1):  1
      Position (x y z):  40 0 0
      Size (radius):  15
   Camera Setup
      Has Camera?  (0/1):  1
      Camera position (x y z):  0 0 25
      Vision Angle (0..180):  60
   Legs Setup
      Leg positioning
         Number of legs:  6
         Legs positioning (n x y z a) clockwise top view:
            1 40 -15 -5 180
            3 0 -15 -5 180
            5 -40 -15 -5 180
            4 -40 15 -5 0
            2 0 15 -5 0
            0 40 15 -5 0
      Individual Leg setup
         Leg bump tolerance :  2.5
         Segment 1
            Length:20 Width:4 Height:4
         Segment 2
            Length:40 Width:2 Height:2
         Angle alpha limits:  -60 60
         Angle beta limits:  -40 60
         Angle gamma limits:  -60 60
         Limits of angle delta (between segments 1 and 2):  30 150
   Reference position Setup
      Reference (x y z):  0.0 20.0 -40.0
```

Figure 2.29: Configuration file for a long hexapod robot.

```
Robot Setup (all angles in degrees)
   Body Setup
      Number of points:  4
      Points (x y):
         0 -15
         -13 -7.5
         -13 7.5
         0 15
         13 7.5
         13 -7.5
      Z min:  -15
      Z max:  15
   Head Setup
      Has head?  (0/1):  1
      Position (x y z):  40 0 0
      Size (radius):  15
   Camera Setup
      Has Camera?  (0/1):  1
      Camera position (x y z):  0 0 25
      Vision Angle (0..180):  60
   Legs Setup
      Leg positioning
         Number of legs:  6
         Legs positioning (n x y z a) clockwise top view:
            3 0 -15 -5 180
            5 -13 -7.5 -5 120
            4 -13 7.5 -5 60
            2 0 15 -5 0
            0 13 7.5 -5 300
            1 13 -7.5 -5 240
      Individual Leg setup
         Leg bump tolerance :  2.5
         Segment 1
            Length:20 Width:4 Height:4
         Segment 2
            Length:40 Width:2 Height:2
         Angle alpha limits:  -60 60
         Angle beta limits:  -40 60
         Angle gamma limits:  -60 60
         Limits of angle delta (between segments 1 and 2):  30 150
   Reference position Setup
      Reference (x y z):  0.0 20.0 -40.0
```

Figure 2.30: Configuration file for a spider-like hexapod robot.

Chapter 3

Programming a Robot Controller

In this chapter we describe how to program a controller for the simulated robot. We first provide a general overview of a controller structure. Next we describe each one of the functions available to manipulate the robot from a user controller. Finally we provide a simple controller example to illustrated the previous explanations.

3.1 The User-Defined Controller Structure

A robot controller consist in a set of functions that are compiled and linked with the rest of the simulator objects. The most important of these functions is the `usr_step` that, when the robot is running, is called at each time slice. This function should implement the control rule for the robot, or what is the same, give a set of low level commands to be executed next. Observe that the effects of these low level commands are not observable until they are actually executed and this happens just after the `usr_step` routine concludes.

Each robot controller is contained in a private directory. This directory must contain a header file (`user.h`), a program file (`user.c`), and a compilation rule file named `dependencies` that enumerates the modules used by `user.c`. The program file `user.c` must define at least the following set of functions:

- `usr_init`: Called when the simulator is started.
- `usr_reset`: Called each time the simulator is reset.
- `usr_step`: Called once at each time slice if the user controller is enabled.
- `usr_close`: Called when the simulator is closed.

All these function receive as a parameter a robot object (the one to be controlled) and a user definable type `t_user` that can be useful to store parameters or the state of the controller in a given time slice. This type is defined in `user.h` and the functions to manipulate it should be provided by the user in the program file `user.c`. The functions to manipulate the robot object are detailed in section 3.2.

The `user_skel` directory provides an empty controller that can be useful as a base to fill in when a new controller wants to be defined.

To compile a new controller you can use:

```
compile directory_name
```

where *directory_name* is the name of the directory containing the controller to be compiled.

To run the simulator with the currently included controller just type

```
sim
```

3.2 The Robot

Next, we enumerate and describe the robot object and its associated set of functions grouped according the its functionality.

3.2.1 The Object

A legged robot has a body and four or more legs. The body shape can be described as a polygon in the X-Y plane of the robot's reference frame sweeped along an interval in the Z axis. In our simulator each leg has three degrees of freedom (dof) and the same structure for all robots. However, some parameters of the leg structure (size, joint bounds, ...) can be instantiated for each robot. Each one of leg degrees of freedom is controlled by an independent motor. The most elementary method to move a leg is to command an angle directly to one of these motors. If this angle is inside the user provided bounds, the motor instantaneously moves to the desired angle (we assume that motors can move at any speed). However, the angular form of control is not intuitive (for instances, to move a leg along a straight line at least two motors have to be properly coordinated). For this reason, we provide functions to command each leg in two Cartesian frames of references (see figure 3.1): the leg one (particular for each leg) and the robot one (common for all legs). Additionally, methods are provided to get leg positions in another two frame of references: the world one and the reference one (defined from the reference positions).

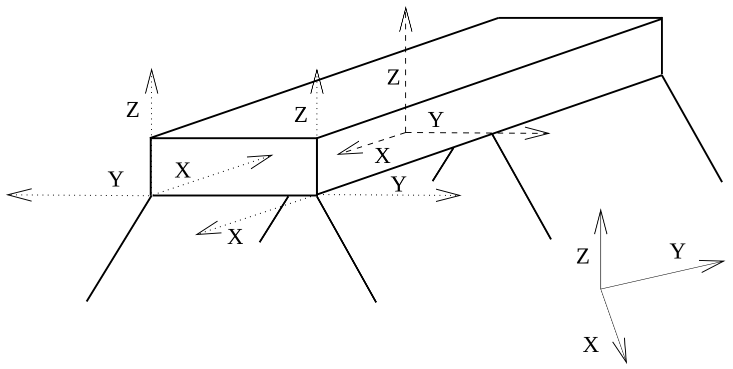


Figure 3.1: *The body frame of reference (dashed lines), two leg frame of references (dotted lines), and the world frame of reference (solid lines) in an arbitrary position.*

The reference position of a leg is the preferred position for that leg. This position normally correspond to the central position of the leg workspace (i.e. when all angles

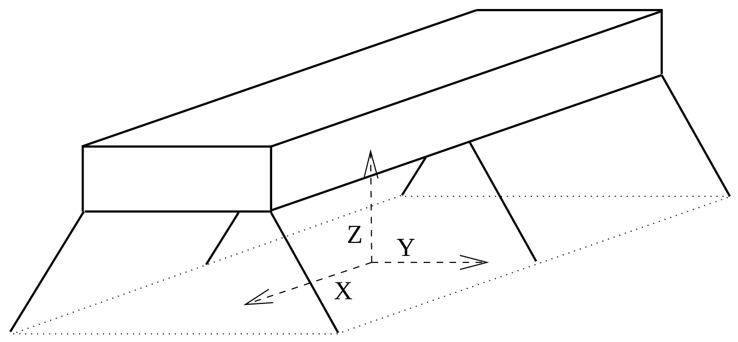


Figure 3.2: *The reference frame of reference.*

are set to zero) but the user can choose it to be a different one. The *reference frame or reference* is a Cartesian frame of reference placed in the center of gravity of the polygon conformed by reference positions (see figure 3.2). This frame of reference is only used to evaluate tensions (see section 2.1.6).

To easy even more the leg movement control, we have included two high level methods to move a leg. The first one moves a set of legs in a coordinated way (applying to all of them the same transformation expressed in the robot's reference frame). This coordinated movement of legs is what we called a *gesture* [4]. The other high level function to move legs allows to move a leg to a given position in a given time (expressed as a number of time slices to achieve the final position).

All robot models are equipped with touch sensors at feet (that detect contact with the ground in any direction) and two inclinometers (one aligned with the X axis of the robot and the second aligned with the Y axis). Optionally, a robot can be provided with a camera. If this is the case, the simulator shows and additional window displaying the images provided by the camera. The pan and tilt of the camera can be queried and modified with the appropriate functions described below.

3.2.2 Setup Functions

These are function to set the global parameters of the robot.

```
void robot_load (char *filename,t_robot *r)
```

This function loads a robot definition from a file named *filename*. Robots are described with the syntax defined int section 2.2.3. If the file does not exists, nothing is done. If it do not correspond to a correct robot description then the simulator stop its execution. Once loaded, the robot is put in its reset position (setting all angles to zero). If necessary, the ground shape is modified so that the robot touches the ground with all legs standing in the reset position.

```
void set_friction (boolean friction,t_robot *r)
```

This function allow to enable (friction=TRUE) or disable (friction=FALSE) the friction of legs. Without friction the robot body can not be propelled. This mode can be useful when positioning the robot.

The type boolean and the constants TRUE and FALSE are defined in the file `boolean.h` provided with the simulator source.

3.2.3 Structure Functions

These are functions to get information about the robot structure.

```
unsigned int number_of_legs (t_robot *r)
```

Returns the number of legs of the robot.

```
unsigned int right_leg_of (unsigned int n,t_robot *r)
```

Returns the number of the leg that is at the right of leg `n` or, what is the same, the leg that is next to leg `n` anti-clockwise (from a top view of the robot).

```
unsigned int left_leg_of (unsigned int n,t_robot *r)
```

Returns the number of the leg that is at the left (i.e., next clockwise) of leg `n`.

3.2.4 Positioning Functions

These are the functions useful to move the robot to a given place in the environment.

```
void robot_reset (t_robot *r)
```

Moves the robot to the reset position that consists first in moving the robot to the origin of the world reference system and then moving all legs to its reference position.

```
void move_robot (unsigned int dof_R3,double v,t_robot *r)
```

Translates and rotates the robot. Possible movements are translation along and rotations around axis X, Y and Z. Those movements are expressed using homogeneous transformations[1]: $Tx(x)$, $Ty(y)$, $Tz(z)$, $Rx(\phi)$, $Ry(\theta)$, $Rz(\psi)$. dof_R3 indicates the transformations to be executed numbered from 0 to 5 and `v` the size of its parameter (x, y, z, ϕ, θ, ψ). The transformation can be named using the six constants (`TX=0, TY=1, TZ=2, RX=3, RY=4,` and `RZ=5`) that are provided in the file `transform.h`. All angles should be in radians.

[1]See section 3.2.15 for more details about homogeneous transformations and functions to manipulate them.

3.2.5 Motor Functions

This set of functions allow to command the motors of the robot at different levels of abstraction.

```
void set_angle (unsigned int leg,unsigned int joint,double angle,t_robot *r)
```

Moves the indicated joint of the given leg to position `angle`. If the angle is out of the mechanical limits of the motor the leg is not moved. The bounds of each motor are indicated by the user in the robot configuration file.

```
void set_leg_position (unsigned int leg,double *position,t_robot *r)
```

Moves the leg to the given position expressed in the particular leg frame of reference. If the position is out of the workspace the leg is not moved.

```
void set_position (unsigned int leg,double *position,t_robot *r)
```

Moves the leg to the given position expressed in the frame of reference attache to the robot's body.. If the position is out of the workspace the leg is not moved.

```
void leg_to_ground (unsigned int leg,t_robot *r)
```

Descend the leg until the ground is touched. If the ground is not reachable, the leg is not moved.

```
void move_to (unsigned int leg,double *position,unsigned int tick,t_robot *r)
```

Takes the leg to the given position in `tick` time slices. The leg is moved a fixed distance each time slice until the target is reached or until the next aimed position is out of the workspace. The leg is moved from the start position to the final one in straight line (in Cartesian coordinates, not in angular ones).

3.2.6 Gesture Functions

This set of functions allow the execution of gestures that move legs in a coordinated way. If the friction is active and three or more legs in contact with the ground are affected by gestures, the execution of a gesture results in keeping feet in the same position as they are (in the world reference frame) and displacing the body of the robot in the opposite direction of the executed gesture.

```
void gesture_effect (unsigned int leg,boolean effect,t_robot *r)
```

Allow to select the legs that are affected by gestures (`effect=TRUE`).

```
void exec_gesture (unsigned int dof_R3,double v,t_robot *r)
```

Executes a gesture. A gesture consist in applying the same homogeneous transformation to all legs affected by gestures. When leg friction is active and three or more legs in contact with the ground are affected by gestures, the inverse of the gesture transformation is applied to the body. The resulting effect is to move the body and to keep feet in the same position as they are. If the legs affected by gestures do not properly support the body or if legs slip over the ground then only the legs are moved and the robot's body remains stationary.

The parameters `dof_R3` and `v` allow to select the transform to apply to legs ($Tx(x)$, $Ty(y)$, $Tz(z)$, $Rx(\phi)$, $Ry(\theta)$ or $Rz(\psi)$) and the size of its parameter (x, y, z, ϕ, θ, ψ). Again, the transformation can be named using the constants `TX, TY, TZ, RX, RY`, and `RZ` defined in the file `transform.h`.

If more than one gesture is issued in the same time slice, then they are executed in the following sequence

$$q^i = (Rz(\psi)Ry(\theta)Rx(\phi)Tz(z)Ty(y)Tx(x))p^i$$

where q^i is the new position of leg i and p^i is its previous position. The effect on the robot's body is just the inverse:

$$(Rz(\psi)Ry(\theta)Rx(\phi)Tz(z)Ty(y)Tx(x))^{-1} = (Tx(-x)Ty(-y)Tz(-z)Rx(-\phi)Ry(-\theta)Rz(-\psi))$$

Taking into account this order is important since gestures are the only way to make the robot advance.

3.2.7 Balance Functions

This set of functions are included in the simulator to easy the implementation of the *balance control* introduced in [4] and extended in [11]. These functions, however, are not useful if other control schemas are adopted.

A balance is a measure of the position of the robot's body with respect to feet positions along a given dof. This measures are evaluated by comparing each leg position at a given moment with respect to a reference position that is at a fixed place with respect to the body. Both the legs and the reference positions are expressed in the *reference frame of reference* introduced in section 3.2.1.

The difference of a leg position with respect to its current reference position along a given dof is called the *tension* of this leg. There is a physical analogy that can illustrate the meaning of the tension. If we imagine that each leg is attached to the reference position by an ideal spring, then the tension correspond the the forces and moments that the spring connected to the leg exert on the robot's body.

Formally, if $p^i = (p^i_x, p^i_y, p^i_z)$ is the current position of a leg in the reference frame and $r^i = (r^i_x, r^i_y, r^i_z)$ is its reference position in this same frame of reference, we can compute and many different tensions as possible dof in the Euclidean space (i.e., Tx, Ty, Tz, Rx, Ry, Rz):

$$\begin{aligned}
Tension^i_{Tx} &= p^i_x - r^i_x \\
Tension^i_{Ty} &= p^i_y - r^i_y \\
Tension^i_{Tz} &= p^i_z - r^i_z \\
Tension^i_{Rx} &= p^i_z r^i_y - p^i_y r^i_z \\
Tension^i_{Ry} &= p^i_x r^i_z - p^i_z r^i_x \\
Tension^i_{Rz} &= p^i_y r^i_x - p^i_x r^i_y
\end{aligned}$$

Normally, the reference position of a leg should be the central position of its workspace (the one reached when all motor angles are zero) but the user can select another initial reference position and he can on-line change the reference position for legs. Formally, if r^i_{init} is the initial reference position given by the user, then the reference position r^i at a given moment is

$$r^i = Rz(r_\psi)Ry(r_\theta)Rx(r_\phi)Tz(r_z)Ty(r_y)Tx(r_x)r^i_{init}$$

where r_ψ, r_θ, r_ϕ, r_z, r_y, and r_x are initially set to 0 and that be changed using the functions described below.

The values of the tensions (and consequently the balance values) can be modified by moving legs (either individually of performing gestures) or by moving the reference positions. In our controllers both methods are used in different circumstances.

```
double get_current_tension (unsigned int leg dof_R3, t_robot *r)
```

Returns the current tension produced by a given leg in the specified dof.

```
double calculate_tension (unsigned int leg dof_R3,double *position,t_robot *r)
```

Returns the tension that a leg would exert on the robot's body in a given dof if the leg was placed at `position`. The position is given in the body frame of reference.

```
double get_balance (unsigned int leg dof_R3,t_robot *r)
```

Returns the adding of tensions in a given dof for all legs.

```
void move_balance_reference (unsigned int dof_R3,double v,t_robot *r)
```

Sets the parameter $r_{dof\_R3} \leftarrow v$

`void move_incr_balance_reference (unsigned int dof_R3,double v,t_robot *r)`

Sets the parameter $r_{dof\_R3} \leftarrow r_{dof\_R3} + v$

`void reset_balance_reference (t_robot *r)`

Sets the values of the reference positions to its initial values (i.e., sets all reference movement parameters r_ψ, r_θ, r_ϕ, r_z, r_y, and r_x to 0).

`void get_central_position (unsigned int leg,double *position,t_robot *r)`

Returns the initial reference position for the given leg (r^{leg}_{init}).

`void get_balance_reference (unsigned int leg,double *position,t_robot *r)`

Returns the current reference position for the given leg (r^{leg}).

3.2.8 Camera Control Functions

The robot can be equipped with an active camera. This set of functions allow to command the camera and get information about its state.

`boolean has_camera (t_robot *r)`

Returns `TRUE` if the robot has a camera.

`void set_pan (double pan,t_robot *r)`

Sets the pan angle for the camera. If `pan` is negative the camera is moved to the right and if positive to the left.

`void set_tilt (double tilt,t_robot *r)`

Sets the tilt angle for the camera. If `tilt` is negative the camera is moved down and if positive up.

`void set_vision_angle (double angle,t_robot *r)`

Allows to modify the aperture of the camera or, what is the same, the angular field-of-view. The angle must be given in radians.

```
double get_pan (t_robot *r)
```

Returns the current pan of the camera.

```
double get_tilt (t_robot *r)
```

Returns the current tilt of the camera.

```
double get_vision_angle (t_robot *r)
```

Returns the current aperture of the camera.

3.2.9 Execution Control Functions

Those functions allow to get information about the execution of robot's commands and to control them if necessary.

```
void update_robot (t_robot *r)
```

Execute all pending robot commands.

```
boolean blocked_leg (unsigned int leg,t_robot *r)
```

Returns `TRUE` if the last command received by the leg has tried to move it out of its workspace. When a new correct command is received, this function returns `FALSE`.

```
boolean all_gestures_executed (t_robot *r)
```

Returns `TRUE` if all the gestures commanded in the previous time slice have been correctly executed. A gesture is not executable if it tries to move any leg out of its workspace.

```
boolean all_executed (t_robot *r)
```

Returns `TRUE` if all commands send to the robot in the previous time slice have been correctly executed or, in other words, if no leg has been blocked.

```
boolean position_achieved (unsigned int leg,t_robot *r)
```

Returns TRUE when there is not any move_to command pending to be finished for the given leg.

```
void stop_leg (unsigned int leg,t_robot *r)
```

This function aborts the commands (set_leg_position, set_position, set_angle, exec_gesture) pending to be executed for a given leg.

```
void stop_robot (t_robot *r)
```

Avoid the execution of all pending robot commands as well as stops any on going command. This command not only step leg commands but also stops the commands send to the robot camera.

3.2.10 Sensor Functions

This set of functions allow to get the values of the sensor robots either the proprioceptive ones (giving information about leg positioning) and the no proprioceptive ones (giving information about the environment).

```
double get_angle (unsigned int leg,unsigned int joint,t_robot *r)
```

Returns the angle at which a given joint of a given leg is.

```
void get_leg_positon (unsigned int leg,double *position,t_robot *r)
```

Return the leg position in the leg frame of reference.

```
void get_position (unsigned int leg,double *position,t_robot *r)
```

Return the leg position in the body frame of reference.

```
void get_position_rf (unsigned int leg,double *position,t_robot *r)
```

Return the leg position in the reference frame of reference.

```
void get_world_position (unsigned int leg,double *position,t_robot *r)
```

Return the leg position in the world frame of reference.

```
boolean joint_bounds (unsigned int leg joint bound, t_robot *r)
```

Returns true if the indicated joint of the given leg is in an extreme position of its range of movement (± 1 degree).

```
double get_inclination_x (t_robot *r)
```

Returns the angle (in radians) of the X axis of the robot with respect to the horizontal plane.

```
double get_inclination_y (t_robot *r)
```

Returns the angle (in radians) of the Y axis of the robot with respect to the horizontal plane.

```
boolean touch_sensors (unsigned int leg,t_robot *r)
```

Returns `TRUE` if the leg is in contact with (or inside) the ground.

```
boolean stable (t_robot *r)
```

Returns `TRUE` if the current set of footholds properly support the robot.

3.2.11 Landmarks Functions

The simulator allow to define visual landmarks that can be helpful to implement legged robot navigation algorithms. In this version of the simulator, landmarks are simple balls placed at fixed positions. If the robot is equipped with a camera it can detect a subset of this landmarks. The following functions allow to identify this subset of visible landmarks.

```
unsigned int number_of_visible_landmarks (t_robot *r)
```

Returns the number of landmarks visible for the robot at a given moment.

```
void get_landmark (unsigned int n,t_landmark *l,double *p,t_robot *r)
```

Returns the description (`l`) and the position (`p`) in the robot's frame of reference of the the landmark number `n` visible by the robot at a given moment. The functions to manipulate the type `t_landmank` are described in section 3.2.16.

3.2.12 Frame of Reference Conversion Functions

In the simulator three main frame of references are used: the leg (different for each leg), the robot and the world frame of reference. This functions return homogeneous transforms that allow to convert points expressed in one of those frame of references to another one. No functions are provided to convert from or to the reference frame of reference since it plays a secondary role and is defined mainly for internal purposes. See section 3.2.15 for details about functions to manipulate transformations and points.

```
transform * leg_to_body  (unsigned int leg,t_robot *r)
```

Returns the transformation to transform from the leg frame of reference (of the indicated leg) to body the body frame of reference.

```
transform * body_to_leg  (unsigned int leg,t_robot *r)
```

Returns the transformation to transform from the body frame of reference to a given leg frame of reference.

```
transform * robot_to_world (t_robot *r)
```

Returns the transformation to transform from the robot frame of reference to the world frame of reference. This transformation can be used, for instance, to get the current position of the robot in the world[2]. However, this odometric measure shouldn't be used in the robot controllers if we aim them to be applicable to a real robot since in a real robot the odometry would be much more imprecise of that provided by the simulator.

```
transform * world_to_robot (t_robot *r)
```

Returns the transformation to transform from the world frame of reference to the robot frame of reference.

3.2.13 Trace Functions

A trace is a polygonal line whose points can be defined by the user in successive time slices. A trace is drawn as a set of balls of the same color and size connected by lines. Traces are intended to monitor the displacement of any part of the robot. They can be used to mark the path followed by the robot, the future passing points for the robot, the turning centers of the robot path, the footholds used by a given leg, ...

[2] `get_translation(robot_to_world(robot),p)` returns in p the position of the center of the robot in world coordinates.

`void start_trace (unsigned int n,char *name,double r g b,double size,t_robot *r)`

Defines the parameters of the trace number `n`. In the current version of the simulator at most 10 different traces can be used. The parameters `r`, `g`, and `b` are doubles in the interval $[0, 1]$ that define the color of the ball of the trace. `size` refers to the ball radius. The `name` is the name that appears in the trace control window of the simulator on-line interface to refer to this trace.

`void restart_trace (unsigned int n,t_robot *r)`

Delete all the points of a trace but keeps the trace definition so that new points can be added to it.

`void delete_trace (unsigned int n,t_robot *r)`

Eliminates a given trace. Once deleted no more points can be added to a given trace.

`boolean trace_defined (unsigned int n,t_robot *r)`

Returns `TRUE` if trace number `n` is defined and point can be added to it.

`void add_point (unsigned int n,double *position,t_robot *r)`

Adds a point to a given trace. The point should be given in the world frame of reference.

`void show_trace (unsigned int n,t_robot *r)`

We can use this function to enable the visualization of the set of points of a given traces. By default traces are not shown on-line because this slows the workings of the simulator.

`boolean trace_shown (unsigned int n,t_robot *r)`

Returns `TRUE` if the trace is visible.

`void hide_trace (unsigned int n,t_robot *r)`

Hides the trace. It is advisable to hide traces if we want the simulator to work fast.

`char * get_trace_name (unsigned int n,t_robot *r)`

Returns the name of the trace.

3.2.14 Visualization Functions

The visualization is the slower process of the simulator. To accelerate it some functions are provided to select the visualization interval, or what is the same, the number of time slices between to consecutive refresh of the robot display window.

```
void visualize_each (unsigned int tick,t_robot *r)
```

Sets the visualization interval. It is advisable to set a large visualization interval if we want the simulator to work fast.

```
unsigned int get_visualize_interval (t_robot *r)
```

Returns the current visualization interval.

```
void show_robot (t_robot *r)
```

Forces the refresh the robot display window.

3.2.15 Transform Functions

The simulator provides a set of functions to manipulate points and geometric transforms. These functions can be useful to create inputs for the robot functions described before and to manipulate its outputs.

A point is an array of three `doubles` that represent the X, Y and Z coordinates of the point. Points are used, for instance, to describe leg, traces and landmarks positions. No special point type is defined since it is just a simple array. By its side, transforms are matrix for four rows and four columns. A transform can represent an arbitrary translation or rotation or combinations of them. The type `t_transform` can be used to define transformation variables.

Some constants are defined to easy the definition and manipulation of points and transforms:

- `DIM_R3, DOF_R3`: They refer to the dimensions of a point in R3 (three) and the DOF in R3 (six). `DIM_R3` can be used to define points:
  `double point_variable[DIM_R3];`
- `TX, TY, TZ, RX, RY, RZ`: Used to identify each one of the six dofs of R3.
- `M_PI, M_PI_2`: π and $\pi/2$.
- `DEG2RAD, RAD2DEG`: Radians in a degree and degrees in a radiant respectively. They can be used to change from degrees to radians and from radians to degrees with a simple multiplication.

- `AXIS_X, AXIS_Y, AXIS_Z, AXIS_H`: They refer to axis X, Y and Z as well as to the homogeneous coordinate.

The functions to manipulate transforms are described below.

```
void trs_identity (transform *t)
```

Returns the identity matrix:

$$Id = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
void trs_copy (transform *to,transform *td)
```

Sets `td` equal to `to`: `td` $\leftarrow$ `to`.

```
void trs_tx (double tx,transform *t)
```

Returns a matrix that represents a translation along the X axis:

$$Tx(tx) = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
void trs_ty (double ty,transform *t)
```

Returns a matrix that represents a translation along the Y axis:

$$Ty(ty) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
void trs_tz (double tz,transform *t)
```

Returns a matrix that represents a translation along the Z axis:

$$Tz(tz) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
void trs_txy (double tx,double ty, double tz,transform *t)
```

Returns a matrix that represents a translation along axes X, Y, and Z:

$$Txyz(tx, ty, tz) = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
void trs_rx (double rx,transform *t)
```

Returns a matrix that represents a rotation around the X axis:

$$Rx(rx) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos rx & -\sin rx & 0 \\ 0 & \sin rx & \cos rx & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

`rx` is given in radians. As usual, positive angles represent an anti-clockwise rotations (seen from the positive side of the axis).

```
void trs_ry (double ry,transform *t)
```

Returns a matrix that represents a rotation around the Y axis:

$$Ry(ry) = \begin{bmatrix} \cos ry & 0 & \sin ry & 0 \\ 0 & 1 & 0 & 0 \\ -\sin ry & 0 & \cos ry & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

`rx` is given in radians. As usual, positive angles represent an anti-clockwise rotations (seen from the positive side of the axis).

```
void trs_rz (double rz,transform *t)
```

Returns a matrix that represents a rotation around the Y axis:

$$Rz(rz) = \begin{bmatrix} \cos rz & -\sin rz & 0 & 0 \\ \sin rz & \cos rz & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

`rx` is given in radians. As usual, positive angles represent an anti-clockwise rotations (seen from the positive side of the axis).

```
void trs_symmetry (boolean sx, boolean sy, boolean sz, transform *t)
```

Returns

$$S(sx, sy, sz) = \begin{bmatrix} sign(sx) & 0 & 0 & 0 \\ 0 & sign(sy) & 0 & 0 \\ 0 & 0 & sign(sz) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where the function $sign(a)$ returns -1 if a is `TRUE` and -1 if not. This transformation is useful to perform a symmetry of a point with respect to one or more axis.

```
void trs_create (unsigned int dof_R3,double v,transform *t)
```

Returns a transform to perform a movement in the given dof of R3 of size `v`. This function is just a high level interface to call the `trs_tx`, `trs_ty`, `trs_tz`, `trs_rx`, `trs_ry`, and `trs_rz` routines described before.

```
void trs_product (transform *t1,transform *t2,transform *t3)
```

Performs the product of two transformations: `t3` $\leftarrow$ `t1 t2`. The effect of the transform `t3` is the concatenation of the effects of `t2` and `t1` (in this order).

```
void trs_add (transform *t1,transform *t2,transform *t3)
```

Adds two transformations: `t3` $\leftarrow$ `t1` + `t2`.

```
void trs_transpose (transform *t,transform *tt)
```

Returns the transpose of a transformation matrix: `tt` $\leftarrow$ $\mathtt{t}^T$. This function is specially useful since graphical packages uses transformations matrix that are the transposed of the matrixes as defined here.

```
void trs_inverse (transform *t,transform *ti)
```

Returns the inverse of a transformation matrix: `ti` $\leftarrow$ `t`$^{-1}$.

```
void trs_sequence (double tx ty tz rx ry rx, transform *t)
```

Performs the product:

$$CS = Rz(rx)\ Ry(ry)\ Rz(rz)\ Txyz(tx, ty, tz)$$

This transformation concatenates a given set of movements in the six dof of R3.

```
void trs_parallel (double tx ty tz rx ry rx, transform *t)
```

The above described sequential combination of transformations (CS) apply a given transformation on the frame of reference resulting from the already applied movements. This can result problematic if all movements have been calculated in parallel, in the same frame of reference. The parallel combination of transformations presented here avoids this problem.

Each transformation t produces a variation var_t over a given point p:

$$p_{new} = t\,p = p + var_t\,p = (id + var_t)\,p$$

So $t = id + var_t$ and $var_t = t - id$. The accumulation of the variations proposed by some transformations $t_1, \dots, t_n$ is:

$$accumulated\_var = \sum_{i=1}^{n} var_{t_i} = \sum_{i=1}^{n} t_i - n\,id$$

And when this accumulated variation is applied to a given point we get:

$$\begin{aligned} p_{new} =& p + accumulated\_var\ p = p + (\sum_{i=1}^{n} t_i - n\,id)\,p = \\ & (id + \sum_{i=1}^{n} t_i - n\,id)\,p = (\sum_{i=1}^{n} t_i - (n-1)\,id)\,p \end{aligned}$$

Consequently, the parallel combination of transformations performs the addition:

$$CP = Txyz(tx, ty, tz) + Rx(rx) + Ry(ry) + Rz(rz) - 3Id$$

```
void trs_apply (transform *t, double *pi, double *ps)
```

Performs the product

$$\begin{bmatrix} ps_x \\ ps_y \\ ps_z \\ 1 \end{bmatrix} = t \begin{bmatrix} pi_x \\ pi_y \\ pi_z \\ 1 \end{bmatrix}$$

Observe that `ps` and `pi` only have 3 values and that the homogeneous coordinate (1) is internally added when necessary.

```
void trs_get_translation (double *trans,transform *t)
```

Returns the three first rows of the fourth column of the matrix `t`. This is equivalent to execute `trs_apply(t,<0,0,0>,trans)`.

```
void trs_get_axis (unsigned int i,double *axis,transform *t)
```

Returns the three first rows of the `i`-th column of the matrix `t`. This is equivalent to execute `trs_apply(t,i,trans)` where i has a 1 in the `i`-th position and zeros in the rest of positions.

```
void trs_print (FILE *f,transform *t)
```

Writes the matrix `t` in the file `f`.

```
void trs_printT (FILE *f,transform *t)
```

Writes the transposition of the matrix `t` in the file `f`.

3.2.16 Landmark Functions

The landmark object contains the features associated to each landmark. Up to now, landmarks only have two attributes: and unique identifier and its position in the world. However this set of features can be easily extended (for instance with color, shape, ... attributes).

```
unsigned int lmk_get_id (t_landmark *l)
```

Returns the landmarks identifier.

```
void lmk_get_position (double *position,t_landmark *l)
```

Return the landmark position in the world frame of reference. Observe that this position is not the same as the one returned by the `get_landmark` function described before since this second one is expressed in the robot's frame of reference and changes as the robot moves while the first one does not.

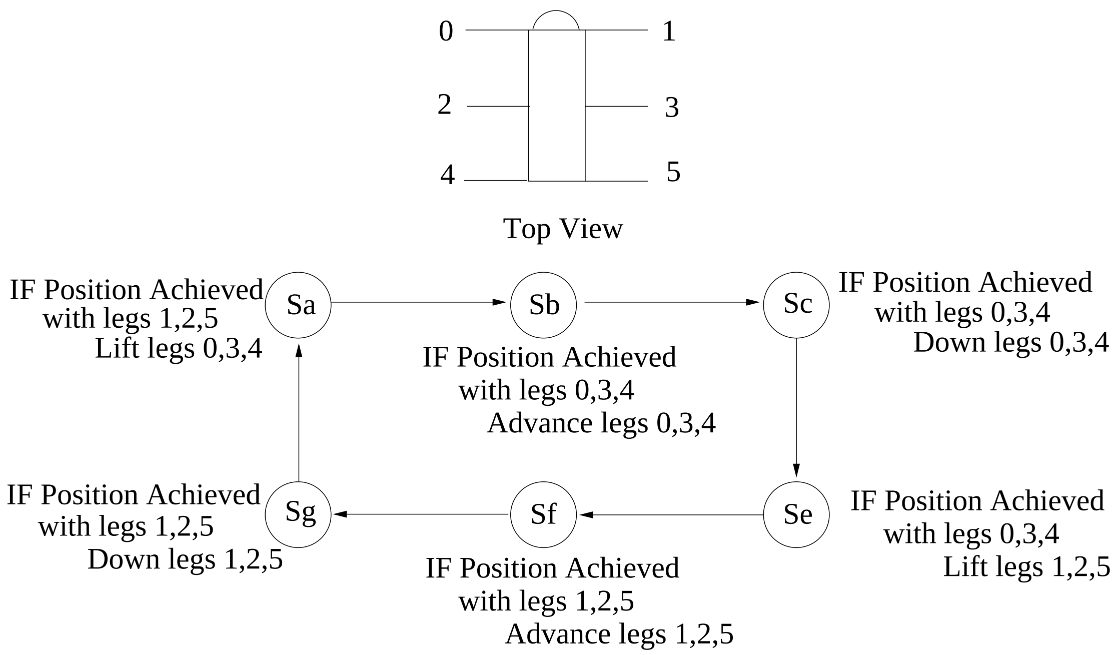

0
1
2
3
4
5
Top View
IF Position Achieved with legs 1,2,5 Lift legs 0,3,4
Sa
Sb
Sc
IF Position Achieved with legs 0,3,4 Down legs 0,3,4
IF Position Achieved with legs 0,3,4 Advance legs 0,3,4
IF Position Achieved with legs 1,2,5 Down legs 1,2,5
Sg
Sf
Se
IF Position Achieved with legs 0,3,4 Lift legs 1,2,5
IF Position Achieved with legs 1,2,5 Advance legs 1,2,5

Figure 3.3: *Graph of states for the example controller.*

3.2.17 An Example: A simple controller

As an example, we present a controller for a long hexapod robot to perform tripod gait on flat terrain. The tripod gait is the fastest stable gait for legged robots. In this gait, two sets of three legs step alternatively (see figure 3.3). This gait can be generated using a finite automata with six states. The first three states (Sa, Sb, and Sc in figure 3.3) are in charge of making a step with legs 0, 3 and 4. The rest of states (Sd, Se, and Sf) do the same for legs 1, 3, 5. The advance movements of legs have to be compensated by a contrary movement of legs on the ground so that the robot advances.

To illustrate the use of the traces we set up seven traces: one for each leg that show the footholds used by each leg and another to show the path followed by the robot.

In this simple controller, we assume that the robot is reseted every time it is stopped.

We start the definition of the controller making a new controller directory in the simulator directory typing:

```
mkdir user_tripod
```

Then we copy the files of the controller skeleton into this directory:

```
cp user_skel/* user_tripod/
```

Then we have to edit the `user.h` and `user.c` files. In the `user.h` file we include the definition of some constants (step height, length, time to perform step movements, . . .) and of the `t_user` type. In this case, this type only contains one integer that represents the state (Sa to Sf).

In the `user.c` file, four functions have to be defined:

- `usr_init`: Ensures that the robot has six legs, start the traces, and calls the `usr_reset` function.
- `usr_reset`: Just reset the traces and sets the `current_state` to the initial state (Sa).
- `usr_step`: Two sub-functions are called. The first one is in charge of stepping legs following the graph represented in figure 3.3. The second one is in charge of making the robot advance as legs move forward. This is achieved by querying a balance value (see section 3.2.7) and executing a gesture with legs in support phase. This mechanism to make the robot advance was first introduced in [2].
- `usr_close`: Deletes the traces.

The contents of the `user.h` and `user.c` files is at the end of this section and they are provided with the simulator package so that you can test and modify them. For instance you can change the step length (AEP) to get a *Follow the leader* gait (you can validate whether or not a given AEP produces such a gait displaying the leg foothold traces). You can also try a negative AEP to see how the robot walks backward.

Once the controller is programmed, we have to configure the simulator so that it loads the correct robot, ground, and landmarks files at start up. This is done editing the `sim.cfg` file in the simulator root directory. In this case this file should contain:

```
    Ground:  grounds/flat.grd
Landmarks:  landmarks/null.lmk
    Robot:  robots/hexapod.rbt
```

This makes the simulator load the hexapod robot, an empty set of landmarks and a flat ground model (all these files are provided with the simulator).

Once this is done we only have to compile the controller typing

```
compile user_tripod
```

and execute it typing

```
sim
```

When the simulator starts, just push the `Run` button to see how the robot evolves under the control of the routines just described.

```
#ifndef CONTROLLERH
#define CONTROLLERH 1

#include "robot.h"

/*Movements*/
#define UP 0
#define ADVANCE 1
#define DOWN 2

/*step constants*/
/*height*/
#define HEIGHT 10
/*length (relative to the reference position)*/
#define AEP 15

/*Time to perform the movements*/
#define T_UP 5
#define T_ADVANCE 10
#define T_DOWN 5

/*States*/
#define Sa 0
#define Sb 1
#define Sc 2
#define Sd 3
#define Se 4
#define Sf 5

typedef struct {
   unsigned int current_state;
} t_user;

void usr_init(t_robot *r,t_user *u);
void usr_reset(t_robot *r,t_user *u);
void usr_step(t_robot *r,t_user *u);
void usr_close(t_robot *r,t_user *u);

#endif
```

```
#include "user.h"

#include "transform.h" /*for TX,TY,...  constants*/
#include "boolean.h" /*TRUE, FALSE,...*/
#include "math.h"

/*Private functions header*/
void advance_robot(t_robot *r,t_user *u);
void step_legs(t_robot *r,t_user *u);

/*Private functions definitions*/
void advance_robot(t_robot *r,t_user *u)
{
   unsigned int i;
   unsigned int n;
   double b;

   /*Only legs on the ground proppel the body
   (i.e.  are affected by the gesture to advance)*/
   n=0;
   for(i=0;i<6;i++)
      {
         if (touch_sensor(i,r))
            {
               gesture_effect(i,TRUE,r);
               n++;
            }
         else
            gesture_effect(i,FALSE,r);
      }
   b=get_balance(TX,r);
   if (fabs(b)>1.0)
      exec_gesture(TX,-b/n,r);
}

void step_legs(t_robot *r,t_user *u)
{
   double pos[DIM_R3];
   double ref[DIM_R3];
   unsigned int set1[3],set2[3];
   unsigned int mov;
   unsigned int i;

   /*See which is the movement to perform*/
   switch(u->current_state)
   {
      case Sa:
      case Sd:
         mov=UP;
         break;
      case Sb:
      case Se:
         mov=ADVANCE;
         break;
      case Sc:
      case Sf:
         mov=DOWN;
         break;
   }

/*Select the set of legs that perform the step (set1)*/
/*and the set of legs that proppel the body (set2)*/
if ((u->current_state==Sa)||(u->current_state==Sb)||(u->current_state==Sc))
   {set1[0]=0;set1[1]=3;set1[2]=4;
   set2[0]=1;set2[1]=2;set2[2]=5;}
else
   {set1[0]=1;set1[1]=2;set1[2]=5;
   set2[0]=0;set2[1]=3;set2[2]=4;}

switch(mov)
   {
      case UP:
         /*If the previous movement is finished*/
         if ((position_achieved(set2[0],r))&&
            (position_achieved(set2[1],r))&&
            (position_achieved(set2[2],r)))
               {
                  for(i=0;i<3;i++)
                     {
                        /*Before raising the leg, mark the foothold*/
                        get_world_position(set1[i],pos,r);
                        add_point(set1[i],pos,r);
                        /*If leg 0 is raised, mark the body position also*/
                        if (set1[i]==0)
                           {
                              trs_get_translation(robot_to_world(r),pos);
                              add_point(6,pos,r);
                           }

                        /*Raise the leg*/
                        get_position(set1[i],pos,r);
                        pos[AXIS_Z]+=HEIGHT;
                        move_to(set1[i],pos,T_UP,r);
                     }
                  u->current_state=(u->current_state+1)%6; /*Next State*/
               }
            break;
         case ADVANCE:
            /*If the previous movement is finished*/
            if ((position_achieved(set1[0],r))&&
               (position_achieved(set1[1],r))&&
               (position_achieved(set1[2],r)))
               {
                  for(i=0;i<3;i++)
                     {
                        /*Advance the leg AEP in front of the reference positon*/
                        get_position(set1[i],pos,r);
                        get_balance_reference(set1[i],ref,r);
                        pos[AXIS_X]=ref[AXIS_X]+AEP;
                        move_to(set1[i],pos,T_ADVANCE,r);
                     }
                  u->current_state=(u->current_state+1)%6; /*Next State*/
               }
            break;
         case DOWN:
            /*If the previous movement is finished*/
            if ((position_achieved(set1[0],r))&&
               (position_achieved(set1[1],r))&&
               (position_achieved(set1[2],r)))
               {
                  for(i=0;i<3;i++)
                     {
                        /*Down the leg*/
                        get_position(set1[i],pos,r);
                        pos[AXIS_Z]-=HEIGHT;
                        move_to(set1[i],pos,T_DOWN,r);
                     }
                  u->current_state=(u->current_state+1)%6; /*Next State*/
               }
            break;
      }
}

/*public functions*/
```

```
void usr_init(t_robot *r,t_user *u)
{
   unsigned int i;
   char name[20];

   /*Ensure six legs*/
   if (number_of_legs(r)!=6)
      exit(-1);

   /*Start the traces (footholds of each leg and body path)*/
   for(i=0;i<6;i++)
      {
         sprintf(name,"Leg %u Footholds",i);
         start_trace(i,name,(double)i/5.0,1.0-(double)i/5.0,0.0,5.0,r);
      }
   start_trace(6,"Robot Path",0.0,0.0,0.0,10.0,r);

   usr_reset(r,u);
}

void usr_reset(t_robot *r,t_user *u)
{
   unsigned int i;

   /*Re-Start the traces*/
   for(i=0;i<7;i++)
      restart_trace(i,r);

   u->current_state=Sa;
}

void usr_step(t_robot *r,t_user *u)
{
   step_legs(r,u);
   advance_robot(r,u);
}

void usr_close(t_robot *r,t_user *u)
{
   unsigned int i;

   /*Delete the traces*/
   for(i=0;i<7;i++)
      delete_trace(i,r);
}
```

Chapter 4

Future Extensions

The presented version of our legged robot simulator (including this documentation) is just a β version that (I am sure) includes many errors, incoherences, and limitations. However, this prototype has resulted useful for us to test our controllers and we hope it can be also useful for other researchers. Our objective when starting this project was that of producing a minimal tool to test legged robot simulator and we think this objective has been achieved. In future versions (if they exists at all) many improvements can be done:

- We can allow more than one robot to be simulated simultaneously.
- Different leg structures could be used. Each leg structure requires of its own direct and inverse kinematic routines. We can offer a limited set of leg structures so that the user can choose between them or we can try to program a set of general kinematic routines (but this is a too complex problem in general).
- The robot to ground interaction should be performed in a more realistic way. A good idea would be that of adding a collision detection engine. In this way not only any leg to ground contact would be detected but also leg to leg or body to ground contacts would be also detected.
- We should provided more powerful (on-line) tools to define ground shapes.
- Better triangulization algorithms could be used to set up the triangular mesh that represents the ground. This can reduce the number of polygons to be managed and would increase the visualization speed.
- It would be a good idea to program the visualization routines directly in OpenGL and not to use `Geomview` as a geometry manager any more.
- The world object could be extended to include other objects (stones, trees, ...) to make more realistic sceneries. Additionally, these objects could be animated (i.e., provided with its own dynamic).
- We can include our autonomous robot programming language [9] into the simulator package so that user controllers can be written using it and not directly in C. This

would be an advantage since our language provides structures suited to program robot controllers (parallel behaviors, wires to communicate behaviors, clock triggers, ...).

- Noise could be added to the robot's odometry so that the values it returns are more realistic.
- The whole code should be re-written in C++.
- The exception handler should be improved so that when an error situation is found the user is better informed about the problem.
- The syntax used in the configuration files should be more flexible.

Uff!! such amount of work to do and this in only a small fraction of it!! But for now, that's all folks.

Index

add_point, 51
all_executed, 47
all_gestures_executed, 47
Angles, 42
Argos, 6

blocked_leg, 47
body_to_leg , 50
Boolean type, 42

calculate_tension, 45
compile, 10, 40, 59
Configuration File
    Ground, 30
    Ground Template, 30
    Landmarks, 33
    Robot, 33
    Simulator, 30
Constant
    AXIS_H, 53
    AXIS_X, 53
    AXIS_Y, 53
    AXIS_Z, 53
    DEG2RAD, 52
    DIM_R3, 52
    DOF_R3, 42, 52
    M_PI, 52
    M_PI_2, 52
    RAD2DEG, 52
    RX, 42, 44, 52
    RY, 42, 44, 52
    RZ, 42, 44, 52
    TX, 42, 44, 52
    TY, 42, 44, 52
    TZ, 42, 44, 52

delete_trace, 51
dependencies, 39
Directory structure, 9

exec_gesture, 44

FALSE, 42
Follow the leader, 59

GCC, 9
Geomview, 9, 14, 34
Gesture, 41, 43
gesture_effect, 44
get_angle, 48
get_balance, 45
get_balance_reference, 46
get_central_position, 46
get_current_tension, 45
get_inclination_x, 49
get_inclination_y, 49
get_landmark, 49
get_landmark, 57
get_leg_positon, 48
get_pan, 47
get_position, 48
get_position_rf, 48
get_tilt, 47
get_trace_name, 51
get_vision_angle, 47
get_visualize_interval, 52
get_world_position, 49

has_camera, 46
hide_trace, 51

Installation, 9

joint_bounds, 49

Landmark, 7
left_leg_of, 42
leg_to_body , 50
leg_to_ground, 43
legsim.tar.gz, 9

lmk_get_id, 57
lmk_get_position, 57
lrs, 10

makefile, 10
make.sgi, 10
make.sun, 10
move_balance_reference, 46
move_incr_balance_reference, 46
move_robot, 42
move_to, 43

new_ground, 30
number_of_legs, 42
number_of_visible_landmarks, 49

Odometry, 50

Panel
  Balances, 19
  Camera, 26
  Change view, 24
  Legend, 24
  Load ground, 27
  Load landmarks, 27
  Load robot, 23
  Main, 12
  Manage landmarks, 28
  Move Landmark, 28
  Move leg, 16
  Move robot, 23
  New Landmark, 28
  References, 19
  Save landmarks, 28
  Sensors, 22
  Set angles, 17
  Set positions, 18
  Tensions, 19
  Tools, 13
  Traces, 25
position_achieved, 48

Reference frame
  leg, 40
  reference, 41
  robot, 40
  world, 40
Requisites, 9
reset_balance_reference, 46
restart_trace, 51
right_leg_of, 42
Robot, 7, 39, 40
robot_load, 41
robot_reset, 42
robot_to_world, 50

set_angle, 43
set_friction, 42
set_leg_position, 43
set_pan, 46
set_position, 43
set_tilt, 46
set_vision_angle, 47
Setup, 7
  Ground, 10
  Landmarks, 10
  Robot, 10
  Simulator, 10
show_robot, 52
show_trace, 51
sim, 9–11, 40, 59
sim.cfg, 59
Simulator Processes, 9
stable, 49
start_trace, 51
stop_leg, 48
stop_robot, 48

Tension, 19, 44
touch_sensors, 49
trace_defined, 51
trace_shown, 51
Transform, 42
  $Rx(\phi)$, 42, 44
  $Ry(\theta)$, 42, 44
  $Rz(\psi)$, 42, 44
  $Tx(x)$, 42, 44
  $Ty(y)$, 42, 44
  $Tz(z)$, 42, 44
transform.h, 44
transform.h, 42
trs_add, 55
trs_apply, 57

trs_copy, 53
trs_create, 55
trs_get_axis, 57
trs_get_translation, 57
trs_identity, 53
trs_inverse, 56
trs_parallel, 56
trs_print, 57
trs_printT, 57
trs_product, 55
trs_rx, 54
trs_ry, 54
trs_rz, 55
trs_sequence, 56
trs_symmetry, 55
trs_transpose, 55
trs_tx, 53
trs_txy, 54
trs_ty, 53
trs_tz, 54
TRUE, 42

update_robot, 47
User Controller, 10
    Structure, 39
User Interface, 7
user.c, 39, 59
user.h, 39, 59
user_skel, 39
usr_step, 13
usr_close, 39
usr_init, 39
usr_reset, 39
usr_step, 39

visualize_each, 52

World, 7
world_to_robot, 50

Xforms, 9, 14, 23

Bibliography

[1] Arikawa K. and Hirose S. (1995): “Study of Walking Robot for 3 Dimensional Terrain” Proceedings of the IEEE International Conference on Robotics and Automation, Vol. 1, pp. 703-708.

[2] Brooks R. A. (1989): “A Robot that Walks: Emergent Behavior from a Carefully Evolved Network” Neural Computation Vol. 1 No. 2, pp. 253-262.

[3] Brooks R. A. (1991): “Intelligence without Representation” Artificial Intelligence 47, pp. 139-159.

[4] Celaya E. and Porta J.M. (1998): “A Control Structure for the Locomotion of a Legged Robot on Difficult Terrain”, IEEE Robotics and Automation Magazine, Vol. 5, No. 2, Special Issue on Walking Robots.

[5] Cyberbotics (1999): “WEBOTS 2.0 Reference Manual”, www.cyberbotics.com

[6] Michel 0. (1996): “Kephera simulator version 2.0 user manual”

[7] Phillips M. et al., “Geomview Manual”, The Geometry Center, University of Minnesota, http://www.geom.umn.edu/software/geomview/.

[8] Porta J.M. (2000): “Rho-Learning: A Robotics Oriented Reinforcement Learning Algorithm”, IRI-DT-2000/3.

[9] Porta J. M. and Celaya E. (1996): “The Behavior Language for PC: User’s Guide”, Institut de Robòtica i Informàtica Industrial, Technical Report IRI-DT-9602.

[10] Porta J.M. and Celaya E. (2000): “Learning in Categorizable Environments”, Sixth International Conference on Simulation of Adaptive Behavior: From Animals to Animats.

[11] Porta J.M. and Celaya E. (2000): “Body and leg coordination for omnidirectional walking in rough terrain”, Third International Conference on Climbing and Walking Robots.

[12] Reichler J. A. and Delcomyn F. (2000): “Dynamics Siumulation and Controller Interfacing for Legged Robots” The Intenational Journal of Robotics Research, Vol. 19, No. 1, pp. 49-58.

[13] Sierra C., Lpez de Mntaras R., and Busquets D. (2000): “Multiagent Bidding Mechanisms for Robot Qualitative Navigation”. Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL).

[14] Sony Corporations (1999): “The AIBO Robot” http://www.world.sony.com/Electronics/aibo/

[15] Torres, A. L. (1996): “Virtual Model Control of a Hexapod Walking Robot”, S.B. Thesis, Department of Mechanical Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts.

[16] The XForms GUI toolkit. http://world.std.com/ xforms/