Chapter 6

Implementation and Results

In order to be able to carry out some experiments on motion synthesis through the specification of motion features, it is essential to have a physically based motion simulation environment. A considerable part of the research efforts reported here have been towards the creation and implementation of such a simulation environment. This chapter presents the details of our implementation of this simulation environment and also reports a number of results from various experiments in motion synthesis carried out using this implementation. As we shall see, the simulation environment is powerful and yet simple and will enable a variety of motion simulation related experimental research and development to be carried out.

In our implementation the entire process of motion synthesis is divided into three phases (cf Figure 6.1). In the first phase an optimal controller is synthesized using the stochastic population hill climbing algorithm. In the second phase the motion is recorded frame by frame by executing the controller and simulating the motion. In the third phase the recorded motion is played back.

![Figure 6.1: Three phases in motion synthesis](image)

117
In what follows we shall explain in detail each of these phases and also illustrate our method with a few representative examples of movements of articulated figures. Since limited computational resources were available for this work, we have made the following simplifying assumptions:

- The system simulates movements restricted to two dimensions.
- Only tree structured articulated figures are considered.
- The only external interacting object modelled is the ground.
- Links of the articulated figures are connected using joints of a single type, namely pin joints.
- In a single simulation we consider the motion of a single articulated figure.

It is important for us to emphasize here that the simplifications and assumptions stated above are not in any way inherent limitations of our approach. The implementation is highly modular, and is flexible enough to easily incorporate extensions so that movements of multiple 3D articulated figures with different kinds of joints in a complex 3D virtual environment can be handled.

### 6.1 Controller Synthesis

The different components of the controller synthesis process are shown in Figure 6.2. The animator specifies the physical and the geometric structure of the articulated figure and with motion features the movement task to be performed. Once the motion features are defined the fitness function is composed and the search is undertaken using stochastic population hill climbing algorithm for a suitable controller. The output of the controller synthesis stage is a controller with values assigned to all its parameters. We shall now explain each of the components and their functions.
Figure 6.2: Different components in controller synthesis phase
6.1.1 Geometry Model

The geometry model stores all the relevant geometric information about the articulated figures and the environmental objects. The system allows the animator to describe the geometric structure of an articulated figure. Each link is specified with respect to the coordinate system attached to it (cf Figure 6.3). The specification includes, the length of each link and its relation with other links.

Geometric Description Script

A link type is defined as:

```
objname <name_of_the_object>
path
pts x1 y1
pts x2 y2
```

Once an object of this type is defined, many instances of this object type can be created where each object instance is a link as follows:

```
instance <name_of_the_object> <link_name>
```

Attachment of two links can be indicated to the system by the use of:
attach <link_name> to <link_name>

Local (body) transformations (translation, rotation etc.) on the link can be done using:

\[ \text{tf trans } <\text{link_name}> x y \]

Environmental object geometry is also part of the geometry model. As already mentioned we have chosen to model only ground interactions. Hence we accept position and orientation of the ground plane as specified by the animator. The plane of the ground can be edited to change its orientation. In particular, the orientation of the normal vector can be adjusted. This allows sloped ground planes to be tested with the articulated objects.

### 6.1.2 Physical Model

The physical model stores animator specified physical properties of each individual link such as mass, inertia, position of the centre of mass etc. Although for visualization purposes links are modelled as two dimensional entities, physically they are modelled as one dimensional rigid link-segments. This simplifies the computation to a large extent. For example, the inertia tensor gets transformed from a $3 \times 3$ matrix into a scalar. Typically the inertia is specified in the coordinate system attached to the centre of mass. Each link has a unique parent link and one or more child links. Further, for each articulated figure there is a special link called as the root link. The root link is attached to the observer’s frame of reference. Each child link is attached to the parent link at the origin of the child link. By convention the parent of the root link is assumed to be 0 (cf Figure 6.3). In order to detect collision between the articulated figure and the ground, several points on the body of the articulated figure are identified. These points are called *monitor points*. Monitor points are continuously monitored during the course of simulation. Whenever any of the monitor points are found to collide with the floor, an appropriate response force is computed by the simulator module and applied at the monitor point.
Physical Description Script

A link is defined as:

`link <link_num> <parent_num> <attach_x> <attach_y> <mass> <inertia> <cmass_x> <cmass_y>`

where, `<link_num>` is a number given to this link in the articulated body. `<parent_num>` is the number of the parent link to which this link is connected. Root link always has `<parent_num>` as 0. `<attach_x>`, and `<attach_y>` are the coordinates of the point on the parent link where this link is connected to it. `<cmass_x>` and `<cmass_y>` represent the center of mass of the link. If the link is fixed at some place say ground or roof it is specified as:

`fix <link_num> <fix_x> <fix_y>`

where `<fix_x>` and `<fix_y>` are the coordinates of the fixed point on this link.

Monitor points are used to monitor the points on the body which are expected to collide with the ground (cf Figure 6.4). These are defined as:

`monitor <link_num> <monitor_pt_num> <pt_x> <pt_y>`

6.1.3 Feature model

The feature model stores feature values specified by the animator. Each feature is specified by a key word followed by a value and the allowable range for that feature. For example:

`feature <value> <min> <max>`
6.1.4 The Simulator

In addition to the articulated figure's geometric information the user can directly set parameters for controlling different aspects of simulated environment such as simulation time, magnitude and direction of gravity. Gravity control allows the direction and magnitude of the gravitational acceleration vector to be modified. By setting the components of the gravity vector to zero, gravity can be effectively turned off.

The equations of motion for our articulated bodies are too complex to derive by hand. In addition we wish to have an efficient implementation, since these equations need to be solved at every time step. We were able to integrate into our simulation environment “Dynacomp” [99] a public domain dynamics compiler that symbolically computes equations of motion in the form \( Ax = b \) given a physical description of the articulated figure as input. The output of the dynamics compiler gives the symbolic value for each element of matrix \( A \) and
Figure 6.5: A relation between free-body diagram and the link-segment also to the vector $b$. Values of common subexpressions are precalculated to avoid unnecessary calculations. The implementation uses the recursive Newton-Euler formulation (See section 3.3.2). This is $O(n^3)$ in the number of links and is quite suitable for $n < 10$. The values of $A$ and $b$ are output as lines of "C" code so that the equations of motion can be compiled. The LU decomposition method is used to solve the linear system of equations for the accelerations. $A$ and $b$ are dependent on the internal torques applied at the joints, external forces, the physical properties of the links, and the state of the links. $x$ represents the vector of unknown accelerations. The accelerations are numerically integrated using the simple Euler method to determine new velocity and position of the links. The time step is chosen to be in the range of 0.001 to 0.0005 in order to overcome the stiffness problem.

In our model, the creatures are treated as free bodies in space (cf Figure 6.5). That is all the forces and torques acting on an individual segment are added up to compute the total force and torque acting on the segment. Apart from the
internal forces acting at the specified key joints, the only other external forces modelled are the gravitational force and the reaction force exerted by the floor. The acceleration of each individual link-segment is computed by calculating all forces/torques acting on the segment. The forces exerted by the floor on the articulated figure are calculated using stiff spring and damper model. We favour this approach as it is simple and flexible and also the same formulation of the equations of motion can be used throughout the entire simulation. There is no need to model and compute the magnitude of the impulsive forces that occur upon impact with the ground. The external forces exerted on the figure are computed at the points of contact with the floor which are typically the monitor points (cf Figure 6.6). The position and velocity of these monitor points on the articulated figure are used to compute the external forces as follows:

\[ F_x = -(m_x - p_x)k_p - v_xk_v \]
\[ F_y = -(m_y - p_y)k_p - v_yk_v \]

where \((m_x, m_y)\) is the present position of the monitor point and \((p_x, p_y)\) is the point of initial contact with the floor. Typically spring and damper constants chosen are \(k_p = 10^5 N/m\) and \(k_v = 10^3 N/m\). This creates a suitably stiff floor that functions effectively when used in a simulator with a sufficiently small time step. A simulation script is used to define the various parameters used in simulation.
Simulation Script

The time step used in simulation is set using

```
set dtsim <step_size>
```

The initial state the articulated figure is set using:

```
state <x> <dx> <y> <dy> <th1> <dth1> <th2> <dth2> ....
```

where
- `<x>` is the x position of the origin of the root link
- `<dx>` — x speed
- `<y>` — y position
- `<dy>` — y speed
- `<th1>` — th1 angle of the link 1
- `<dth1>` — dth1 angular speed of link 1

Simulation time can be set by:

```
sim <time_for_simulation>
```

### 6.1.5 Controller representation

The choice of representation for the solutions plays a crucial role in the success of the evolutionary algorithm. Ideally, the representation should be compact enough so that the motion synthesis problem can be solved in a reasonable time, without sacrificing generality. Compactness is achieved by having fairly powerful rule based controller representations that need a small number of states and hence a small number of parameters. In our implementation, we have used the pose control graph and PD control law given by the equation

\[ \tau = k_p(\theta_d - \theta) - k_v \dot{\theta} \]

where \( k_p \) is the spring constant, \( k_v \) is the damper constant, \( \theta \) and \( \dot{\theta} \) are the current angle and the angular velocity respectively and \( \theta_d \) is the rest angle (equilibrium position) of the spring.
The advantage of this controller is that the torque function gets automatically defined once the target joint angle is specified. To execute a particular movement of the joint, it is necessary to define a number of target joint angles. The motion synthesis problem is then converted to that of synthesizing the function $\theta_d(t)$. If the articulated figure has $m$ actuators, it amounts to synthesizing $m$ functions of the type $\Theta_d(t) = (\theta_d^1(t), \theta_d^2(t) \ldots \theta_d^m(t))$. The simplest way to solve the problem is to choose a piece-wise continuous function. This function could be a constant, could be linearly varying or could be more complex with continuous basis functions such as splines [20], sinusoids or wavelets [64] (cf Figure 6.7). For the sake of simplicity, for our experimentation purposes, we have used piece-wise constant functions.

To synthesize a motion sequence for duration $T$, we divide $T$ into several phases or states. Each phase will be associated with a set of parameters such as

$$i, \tau_d, \kappa_p, \kappa_v$$

where

- $i$ — $i^{th}$ joint actuator
- $\theta_d^i$ — desired angle for the $i^{th}$ joint
- $k_p^i$ — spring constant corresponding to the $i^{th}$ joint
- $k_v^i$ — damping constant corresponding to the $i^{th}$ joint
- $t$ — time duration of the phase.

If there are fifteen phases in a motion sequence, the number of unknowns to be

127
determined are $15 \times 4 = 60$. We have mentioned earlier, that, as the simulation duration increases, the search time increases exponentially. When we consider periodic motion, such as walking, running, hopping etc., one can reduce the search time. In periodic motion, after every period $t_{\text{per}}$, the motion is repeated to fill the simulation time $T$. Unknown parameters depend only on the number of phases in the period $t_{\text{per}}$ (cf Figure 6.8). In order to reduce the search time further, one can fix the values $t, k^i_p, k^i_v$ a priori. The time period $t$ could be either derived from a previously synthesized key-framed version of the motion or from live or video data of a creature similar in size and shape [103]. Values of spring constants can be estimated depending on the mass of the body. For heavier links, higher values of spring constants are required in order to protect the springs from a possible collapse (spring failure). However, values should not be so high that they would generate such high torques at the joints that unexpected motions are caused. For more details refer to Appendix A.

### 6.1.6 Performance Metric

For our experimental purpose, we have considered only a few features to characterize the motion. Although this simplifies the implementation considerably, it fully retains the essence of our methodology. Our task is to synthesize periodic motions such as hopping, running and walking. Some of the features that
we have built into our implementation and experimented with are:

- external energy \((E)\)
- horizontal distance travelled by the centre of mass \((D)\)
- intermediate postures \((\theta_{s,i})\), fully or partially specified

The specific performance metric is as follows:

\[
J = w_1 \sum_{s=1}^{\text{samples}} \sum_{j=1}^{\text{joints}} \left( 1 - \left( \theta_{s,j}^o - \theta_{(s,j)} \right) \right) + w_2 \left( 1 - \frac{(E_o - E)}{E_{\text{max}}} \right) + w_3 \left( 1 - \frac{(D_o - D)}{D_{\text{max}}} \right)
\]

where,

- \(w_1, w_2, w_3\) are weights, assigned to the features depending on their relative importance, the value ranging between 0 and 1.
- \(\theta_{(s,j)}\) is the angle at joint \(j\) for posture,
- \(E\) is the external energy
- \(D\) is the horizontal distance travelled
- \(E_{\text{max}}, D_{\text{max}}\) are the maximum expected external energy and horizontal distance, respectively. These values are used for normalization of the two quantities.
- \(\theta_{(s,j)}^o, E_o, D_o\) are the feature values specified by the animator.

For more intuitive explanation of various terms used in the performance metric, refer to Appendix A.

### 6.2 Stochastic Population Hill Climbing Algorithm (SPHC)

The SPHC algorithm is an evolutionary programming algorithm [30] that can be distinguished from genetic algorithms, primarily by the fact that it uses only the mutation operator and does not use a crossover operator.
Like all genetic algorithms SPHC uses a population of solutions. Each solution in the population is perturbed randomly at each iteration, using the mutation operation with probability 1. The resulting solutions are compared with their original solutions and the better ones are kept for the next generation. Periodically, a reseeding operator is applied which selects the top half of the population and copies them into the bottom half of the population, refocusing the search on the most promising solutions in the population. The full algorithm is shown in Figure 6.9

```
Initialize population
Evaluate each solution in the population
for generation = 1 to number_of_generations
    for each individual solution in the population
        Randomly perturb the solution
        Evaluate the new solution
        if the new solution is better than the old one then
            Replace the old solution with the new one
        end if
    end for
    if(generation mod reseed_interval) = 0 then
        Rank order the population
        Replace bottom 50% of the population with top 50%
    end if
end for
```

Figure 6.9: Stochastic population hill climbing (SPHC)

**Mutation Operator**

The mutation operation is the backbone of our SPHC algorithm. In every iteration all solutions go through this operation. Since a slight change in parameters can change the motion drastically, it is necessary to apply the mutation operation with care. We have selected to mutate only one parameter
at a time with only a small change in original value. The parameter to mutate is selected randomly with all the parameters having equal probability. This was found quite suitable through experiments, as it helps the algorithm to fully explore the region near to the existing solution. If we try to mutate more than one parameter at a time, the solution may jump from one region to another without exploring the current one. As the function is multimodal, it may actually be the case that the optimal solution is in the vicinity of solution being mutated. Each time mutation is called either the selected parameter goes through a creep operation\(^1\) or all its parameters are randomized from scratch (cf Figure 6.10).

\begin{verbatim}
Randomly select one of the states in pose-control graph to be modified
Randomly select operation to be applied on selected state
IF operation is creep operation THEN
    Randomly select one of the creep operations and apply
ELSE
    Generate all the parameters in the new state randomly from scratch
\end{verbatim}

Figure 6.10: Mutation operation

6.2.1 A Parallel SPHC

Controller synthesis is computationally a very expensive process. The reasons are two fold. Firstly, the time taken for a single simulation is large. For example, an eight second simulation took around two minutes on a VAX 8600 machine. The simulation time is directly proportional to the complexity of the creature i.e number of links and the monitor points. Secondly, due to the fact that search space is very large, the search algorithm has to make many

\(^1\)The creep operation is used here to modify the parameter by a very small factor. For more details refer to Appendix A.
simulations before locating a suitable controller. The overall time taken by the synthesis process can be reduced considerably if we parallelize the search task. In SPHC algorithm we have a population of solutions which are to be modified and checked separately. All the evaluations and mutations are independent. We can take advantage of this independence property in parallelizing the algorithm.

In the best case, if we have the number of processors equal to the population size, we can run all the simulations separately on each of the processors, achieving maximum parallelism at the granularity of a single simulation. If the processors are lesser in number than the population size, a good scheduling policy has to be implemented to achieve considerable amount of parallelism. Since simulation time for each candidate solution in the population is the same, it is easy to parallelize the search process.

We have implemented the algorithm on a networked environment. There are several heterogeneous workstations connected on the network, each having a different load average at any time instance. Also there are varying communication delays on the network. In addition to parallelizing the code to distribute the evaluations on different processors, we have to handle the problem arising due to varying communication delays. We have adopted a very simple solution to the problem. We treat the population of controllers as one common pool of tasks which are allocated to the set of processors. To start with, all the processors are allocated one candidate each for evaluation. As soon as any of the processors becomes free, it is allotted a new candidate.

To handle the parallelization problems in the network environment we have made use of a system called Parallel Virtual Machine (PVM) [27].

Application programs view PVM as a general and flexible parallel computing resource that supports a message passing model of computation. This resource may be accessed at three different levels, the transparent mode in which tasks are automatically executed on the most appropriate hosts, the architecture-depend mode in which the user may indicate specific architectures on which particular tasks are to be executed, and the low-level mode in which a particular
host may be specified. Such layering permits flexibility while retaining the ability to exploit particular strengths of individual machines on the network.

While parallelizing an application on a multiuser network environment we have to deal with several problems not existing on a parallel computer. Here the effect is on both communication and computational performance of the program. As the machines are of different power, if we divide the problem into identical pieces one for each machine then the application will run as slow as the task on the slowest machine. If the tasks coordinate with each other, then even the fast machines will be slowed down waiting for the data from the slowest machine. The long message latency across the network also affects the performance of the application. As the performance of the network and the machines are dynamically changing the conditions are difficult to reproduce, and hence it is difficult to debug the application.

There are multiple ways by which we can distribute the tasks amongst different processors on the network.

In the simplest case if we have the number of processors equal to the number of tasks, we can assign them one each statically. Here the assignment may occur offline even before the job is started. This kind of distribution is only useful when all the tasks have to be running together and also there is communication between them. In our application we do not have this kind of requirement and hence we will not consider this scheme any further. It also requires that the number of processors be equal to the number of tasks. This is not practical in our case as we generally have a very large population of tasks.

The other scheme, which we have implemented is based on the method known as Pool of Tasks paradigm. It is typically implemented in a master/slave implementation where the master programs creates and holds the “pool” and farms out tasks to slave programs as they fall idle. The pool is implemented as a queue and if the tasks vary in their sizes then the larger tasks are placed near the head of the queue. With this method all the slave processes are kept busy as long as there are tasks left in the pool. Since tasks start and stop at arbitrary times with this method, it is better suited to applications which require
no communication amongst slave programs and the only communication that takes place is with the master or through data files.

Our requirement exactly matches with this model. Each generation in SPHC algorithm has a population of solutions to be evaluated, like a pool of tasks. We have a limited number of processors, so initially each processor is given a solution to be evaluated. As soon as any one of the processors finishes the task (as it happens frequently due to the difference in network load and computational power of machines) it is assigned another member of the population for evaluation. This way all the processors are kept busy.

The main SPHC algorithm when ready with all the solutions to be evaluated, makes a call to the master program. Simulation programs are kept on different machines which take part as slaves. These slave programs are spawned under the control of the master. PVM provides a library routine which allows the processes to be spawned on different machines on the network. The master then sends the appropriate data to this spawned task through message passing routines provided by PVM. We require to pass the structure representing the solution(controller). The slave program then runs the simulator with this controller, calculates the fitness of the solution and returns the fitness to the master. Master keeps one to one correspondence between the solution it had earlier passed to the slave and the fitness it returned. This is done by passing the solution number in the population also as a message to the slave. The slave then returns the same number. This way each fitness value produces its identification to the master process. This avoids the need for processes to finish in the order they were spawned. This is continued till there is no solution left in the pool to be evaluated. Once all the solutions are evaluated the master passes them with their fitness to the SPHC algorithm. The parallel form of SPHC algorithm is shown in Figure 6.11 and the master program which distributes the tasks is shown in Figure 6.12

We achieved considerable performance enhancement using this parallel SPHC algorithm. For example controller synthesis tasks which would take about 8 hours of elapsed time on a single CPU took just 1 hour when the tasks were
Initialize population
Call Master to Evaluate in parallel all the solutions in the population
for generation = 1 to number of generations do
    for each individual solution in the population do
        Randomly perturb the solution
        Call Master to Evaluate in parallel all the solutions in the population
        for each individual solution in the population
            if the new solution is better than the old one then
                Replace the old solution with the new one
            end if
        end for
    end if
end for
if (generation mod reseed-interval) = 0) then
    Rank order the population
    Replace bottom 50% of the population with top 50%
end if
end for

Figure 6.11: Parallel stochastic population hill climbing (SPHC)
Determine the number of hosts currently available
Mark all the available hosts as free
while all the currently running tasks are not over OR there are tasks left to be evaluated do
  while there are tasks left to be evaluated do
    if there is a free host then
      spawn the next task on the host
      send the solution to be evaluated to the spawned task with its number in population
      mark host as busy
    end if
  end while
if there are tasks running currently then
  wait for any of the task to get over
  get the host corresponding to this task
  mark host as free
  wait for slave message containing fitness value and solution number
  store the fitness value corresponding to the solution number
end if
end while

Figure 6.12: Master program distributing the solutions on different hosts
farmed out to 8 workstations. The main feature of the master program is that it is independent of the number of hosts currently in the configuration. So we can add as many hosts as we want in the configuration and get better and better performance. The PVM system provides library calls by which we can identify the situation when a host is added into the configuration, or deleted from the configuration. We can use this feature to implement fully dynamic configurability.

6.3 Simulation of Motion and Sampling of Frames

Once the controller is obtained, it is plugged into the simulator to generate the motion and the sample frames at required rates. The default sampling rate is 25 frames per second of simulation. The frames are recorded in the format:

```
show <x> <y> <th1> <th2> <th3> ...
```

where

- `<x>` is the x position of the root link.
- `<y>` is the y position of the root link.
- `<th1>` is the orientation of the root link.

... Each individual frame is recorded as one command line and basically contains the values for each of the DoFs of the articulated figure.

6.4 Motion Playback

The frames computed and stored by the simulator are played back in real-time by "anix" a public domain animation server for X–windows [98]. This is an X-Windows program, and can produce real time output on a screen for display purposes or in a postscript file format for documentation purposes. The input to anix program is the output file created by simulator. At the end of display of each frame, the frame is erased by setting `<erase>` flag to true.

```
aniset erase t
```
6.5 Experimental Details

We describe below the structure and motion behaviour of three creatures that we have experimented with. These are named as Luxo, Pogo and Walker. Among these creatures, Luxo is the simplest creature made of just three links and two actuator joints. It is a one legged virtual creature. Appearance wise it is similar to a lamp and the only mode of locomotion available to it is hopping. Pogo a two legged virtual creature, is made of 5 links and 4 actuator joints. Appearance wise it is similar to a dog. Dynamically it is more stable than Luxo and can demonstrate different gaits such as walking and running. Walker is a human like virtual creature but without hands or head. It is made of 7 links and 6 actuator joints. In comparison to both Luxo and Pogo it is dynamically very unstable and this makes it very difficult to get a good controller which results into a steady walk.

The motion synthesis process is primarily based on the SPHC algorithm which takes time proportional to the number of generations \( G \), the size of the population \( M \), the amount of time to simulate for each trial, and the accuracy of the integration. This is because each individual controller's motion behaviour must be evaluated by simulation of the dynamics. Even if the simulation can be done faster than real time, it still must be performed for roughly \( M \times G \) different controllers. In our computation, an 8 second controller simulation took approximately two minutes on VAX 8600. Though this may not be particularly fast in terms of CPU time, it is very efficient in human animator time.

For the documentation of the Animation System, refer to Appendix B. Details of the scripts describing for synthesizing

6.5.1 The Luxo creature

Figure 6.13 shows the geometric structure of Luxo.

Table 6.1 shows the allowable ranges of joint angles (in degrees) defining the internal configurations of Luxo.
Figure 6.13: The articulated figure – Luxo (a lamp like creature)

<table>
<thead>
<tr>
<th>Joint</th>
<th>Min (deg.)</th>
<th>Max (deg.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>-300</td>
<td>-240</td>
</tr>
<tr>
<td>A2</td>
<td>360</td>
<td>210</td>
</tr>
</tbody>
</table>

Table 6.1: Range of angles for Luxo

Table 6.2 shows the physical properties of different links of Luxo.

<table>
<thead>
<tr>
<th>Link</th>
<th>Mass (kg)</th>
<th>Inertia (kg.m²)</th>
<th>cmassₓ (m)</th>
<th>cmassᵧ (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>0.15</td>
<td>0.00312</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>L2</td>
<td>0.10</td>
<td>0.00208</td>
<td>0.25</td>
<td>0.0</td>
</tr>
<tr>
<td>L3</td>
<td>0.30</td>
<td>0.00625</td>
<td>0.25</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 6.2: Physical properties of Luxo
Geometric Description Script for Luxo

```plaintext
# polyline definition of link object
objname baselink
path
pts -0.27 0.02 0
pts 0.27 0.02 0
pts 0.27 -0.02 0
pts -0.27 -0.02 0

objname middlelink
path
pts -0.02 0.02
pts 0.52 0.02
pts 0.52 -0.02
pts -0.02 -0.02

objname toplink
path
pts -0.02 0.02
pts 0.42 0.02
pts 0.30 -0.15
pts 0.47 -0.15
pts 0.42 -0.02
pts -0.02 -0.02

# instance of links
instance baselink link1
instance middlelink link2
instance toplink link3

# relative coordinate frame definition
attach link1 to world
attach link2 to link1
attach link3 to link2

# placement of each link
tf trans link2 0.0 0.0
tf trans link3 0.5 0.0
```
物理参数描述：

```
# physical parameters of the link
link 1 0 0.0 0.0 0.15 0.003123 0 0
link 2 1 0.0 0.0 0.10 0.002082 0.25 0
link 3 2 1.0 0.0 0.30 0.006246 0.25 0

# specification of monitor points
monitor 1 1 -0.27
monitor 1 2 0.27
```

模拟脚本：

```
# simulation time step
set dtsim 0.001

# symbolic code for equations of motion
dyn luxo

# symbolic code for monitor points
mon mon_luxo 2

# number of state variables
set state_size 10

# initial values of state variables
state 0.0,0.0,0.0,0.0,0.0,0.0,0.0,-0.69,0.0,4.665,0.0

# name of the output file
set dispfile luxo.out

# simulation time (sec)
sim 10.0

quit
```
Animation Script for Luxo

```plaintext
# initialize server
init

# read the values x y th th1 th2 for a frame
tf trans link1_1 .2 0
tf rot link1 Z .3
tf rot link2 Z .4
tf rot link3 Z .5

# initial position of link1 on the screen
tf trans world 10 4

# scale the articulated figure
tf scale world 3 3

# set degree mode to false
aniset degmode f

# erase the frame at the end of the display
aniset erase t
```

The controller for Luxo has been designed using a two node pose control graph. The parameter space is ten dimensional. The ratio $k_p/k_v$ is chosen as 0.1.

$$[\theta_1, \theta_2, k_p^1, k_p^2, t_1] \quad [\theta_1, \theta_2, k_p^2, k_p^1, t_2]$$

Five posture features to synthesize hopping motion are listed in Table 6.3. The postures are approximately 0.2 sec apart in time. The value of distance to be travelled in a single hop is given as 0.4(m) and external energy as 2.3(Nm). Just in order to convince ourselves that the SPHC algorithm will indeed find the optimum, we synthesized the same motion i.e optimizing the same performance metric by randomly choosing many different sets of initial populations. Figure 6.14 shows two such cases of the progress of the SPHC search algorithm in finding the hopping motion controller for Luxo. In the first case the desired controller is found after 40 generations with a population size of 50. In the second case, more or less a similar controller was found after only 25 generations with the same population size.
Figure 6.14: Synthesis of two different controllers for Luxo

Figure 6.15: Luxo hopping

Figure 6.16: Phase diagram for hopping Luxo
Figure 6.17: Variation in torques for hopping Luxo

Figure 6.18: Variation in joint angles for hopping Luxo

Figure 6.19: Variation in total energy of hopping Luxo
values have been chosen so as to synthesize walking as well as running motion. The progress of the search algorithm in finding the walking motion controller is shown in Figure 6.21.

Two different gaits are synthesized for Pogo. In the first experiment a walking gait is obtained by choosing four intermediate postures at time intervals of approximately 0.2 sec as shown in the table 6.6. The value of distance travelled and external energy in a single gait cycle is chosen as 0.6(m) and 0.7(Nm) respectively. The output is shown in the Figure 6.22 below.

Figure 6.23 shows a phase diagram which plots height of centre of mass versus vertical velocity of centre of mass. The trajectories in the phase diagram once again show a periodic behaviour with trajectories being attracted towards an attractor cycle, indicating stable behaviour. Figure 6.24 shows the variation in total energy for walking Pogo.

In the second experiment a running gait is obtained by choosing three intermediate postures (Table 6.7) with value of distance travelled in a single gait cycle as 0.5(m) and external energy as 0.8(Nm).
Table 6.5: Physical properties of Pogo

<table>
<thead>
<tr>
<th>Link</th>
<th>Mass ((kg))</th>
<th>Inertia ((kg.m^2))</th>
<th>(cmass_x) ((m))</th>
<th>(cmass_y) ((m))</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>0.15</td>
<td>0.003123</td>
<td>0.25</td>
<td>0.0</td>
</tr>
<tr>
<td>L2</td>
<td>0.10</td>
<td>0.002082</td>
<td>0.125</td>
<td>0.0</td>
</tr>
<tr>
<td>L3</td>
<td>0.10</td>
<td>0.002082</td>
<td>0.125</td>
<td>0.0</td>
</tr>
<tr>
<td>L4</td>
<td>0.10</td>
<td>0.002082</td>
<td>0.125</td>
<td>0.0</td>
</tr>
<tr>
<td>L5</td>
<td>0.10</td>
<td>0.002082</td>
<td>0.125</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Figure 6.21: Synthesis of a controller for Pogo

Figure 6.22: Pogo, walking
Figure 6.23: Phase diagram for a walking Pogo

Figure 6.24: Variation in energy for walking Pogo
Table 6.6: Four posture features for synthesizing a walking motion for Pogo

<table>
<thead>
<tr>
<th>Pose#</th>
<th>$\theta_1$ (rad.)</th>
<th>$\theta_2$ (rad.)</th>
<th>$\theta_3$ (rad.)</th>
<th>$\theta_4$ (rad.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1.9700</td>
<td>5.1936</td>
<td>-0.8497</td>
<td>5.0566</td>
</tr>
<tr>
<td>2</td>
<td>-2.0709</td>
<td>5.2423</td>
<td>-0.6785</td>
<td>5.2343</td>
</tr>
<tr>
<td>3</td>
<td>-1.8970</td>
<td>4.7794</td>
<td>-0.8469</td>
<td>4.7110</td>
</tr>
<tr>
<td>4</td>
<td>-1.5962</td>
<td>4.7227</td>
<td>-1.1903</td>
<td>4.3722</td>
</tr>
</tbody>
</table>

Table 6.7: Three posture features for synthesizing a running motion for Pogo

<table>
<thead>
<tr>
<th>Pose#</th>
<th>$\theta_1$ (rad.)</th>
<th>$\theta_2$ (rad.)</th>
<th>$\theta_3$ (rad.)</th>
<th>$\theta_4$ (rad.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1.0138</td>
<td>5.8728</td>
<td>-1.0830</td>
<td>4.1118</td>
</tr>
<tr>
<td>2</td>
<td>-1.8247</td>
<td>5.4664</td>
<td>-0.9404</td>
<td>4.6641</td>
</tr>
<tr>
<td>3</td>
<td>-1.9589</td>
<td>5.3294</td>
<td>-0.8445</td>
<td>4.7969</td>
</tr>
</tbody>
</table>

The output is shown in the Figure 6.25 below.

Figure 6.25: Pogo, running

Figure 6.26 shows a phase diagram for running Pogo. After initial instable behaviour, the trajectory settles in with a periodic behaviour. Figure 6.27 shows the variation in total energy for running Pogo.
Figure 6.26: Phase diagram for running Pogo

Figure 6.27: Variation in total energy for running Pogo
6.5.3 The Walker creature

Figure 6.28 shows the geometric structure of Walker. Controller for Walker consists of a four node pose control graph.

![Diagram of Walker]

Figure 6.28: The articulated body — Walker (a human like creature)

Table 6.8 shows the allowable ranges of joint angles (in degrees) defining the internal configurations.

<table>
<thead>
<tr>
<th>Joint</th>
<th>Min (deg.)</th>
<th>Max (deg.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>-160</td>
<td>-125</td>
</tr>
<tr>
<td>A2</td>
<td>-10</td>
<td>-45</td>
</tr>
<tr>
<td>A3</td>
<td>60</td>
<td>100</td>
</tr>
<tr>
<td>A4</td>
<td>-160</td>
<td>-125</td>
</tr>
<tr>
<td>A5</td>
<td>-10</td>
<td>-45</td>
</tr>
<tr>
<td>A6</td>
<td>60</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 6.8: Range of angles for Walker

Table 6.9 shows the physical properties of different links.

The result of the simulation of Walker is shown in the Figure 6.29 below.
<table>
<thead>
<tr>
<th>Link</th>
<th>Mass (kg)</th>
<th>Inertia (kg.m²)</th>
<th>cmassₓ (m)</th>
<th>cmassᵧ (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>3.0</td>
<td>0.0625</td>
<td>0.250</td>
<td>0.0</td>
</tr>
<tr>
<td>L2</td>
<td>5.0</td>
<td>0.0260</td>
<td>0.125</td>
<td>0.0</td>
</tr>
<tr>
<td>L3</td>
<td>4.0</td>
<td>0.0208</td>
<td>0.125</td>
<td>0.0</td>
</tr>
<tr>
<td>L4</td>
<td>5.0</td>
<td>0.0260</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>L5</td>
<td>4.0</td>
<td>0.0208</td>
<td>0.125</td>
<td>0.0</td>
</tr>
<tr>
<td>L6</td>
<td>1.0</td>
<td>0.0052</td>
<td>0.125</td>
<td>0.0</td>
</tr>
<tr>
<td>L7</td>
<td>1.0</td>
<td>0.0052</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 6.9: Physical properties of Walker

Figure 6.29: Walker walking