Determination of Elastodynamic Model Parameters using a Recurrent Neuro-Fuzzy System

Andreas Nürnberger, Arne Radetzky[*], and Rudolf Kruse
University of Magdeburg, Faculty of Computer Science (FIN-IWS),
Universitätsplatz 2, D-39106 Magdeburg, Germany
Tel. +49-391.67.11358, Fax +49-391.67.12018
email: {andreas.nuernberger, rudolf.kruse}@cs.uni-magdeburg.de
[*]University of Braunschweig, Faculty of Computer Science
38100 Braunschweig, Germany
Tel. +49.531.391.2125, Fax +49.531.391.9502
email: a.radetzky@umi.cs.tu-bs.de

ABSTRACT: Elastodynamic models are used for real-time simulation of deformable objects in virtual reality applications. To obtain a realistic simulation, the physical parameters of the model must be defined appropriately. Furthermore, the usability of elastodynamic models in virtual reality applications depends on the used simulation algorithm to a great part, since interactions with the simulated object have to be done in real-time. In this paper we present a learning algorithm which determines the parameters of a spring-mass model. The algorithm is based on a recurrent neural network and can be initialized by use of a fuzzy system. The presented algorithm uses the positions of specific object points during discrete time steps to learn the required parameters. The time series data for learning can be derived, for example, by a time dependent optical measurement of an object under influence of external forces.

KEYWORDS: system identification, recurrent network, neuro-fuzzy, spring-mass model, virtual reality.

MOTIVATION

Nowadays computer assisted surgery is already used for procedure training and operation planning [2]. Most of the utilized methods are based on static visualization techniques. To improve the benefit for surgical training a visual convincing static modeling of the operation scenario and the involved tissues is not sufficient, because tissues can be deformed at contact and transections can be made [21]. Especially for the interaction with medical devices, such as laparoscopic instruments, it is necessary to simulate the deformation of tissue under the influence of collision forces. Of course, this has to be done in real-time. To fulfill these requirements, spring-mass models [3][22] can be used.
To simulate deformable tissues in surgical simulation, traditional deformation models, e.g. based on the (exact) computation of differential equations, are difficult to use, since often a model of the physical system can not be obtained by conventional approaches. This can be caused by problems in constructing a system of differential equations and (most often) in finding the required parameters. Furthermore, the simulation of such a model description is usually very time-consuming and the use in real time applications is impossible. A further problem is that even small or local changes of the model structure often require a rebuild of the whole model. Some approaches try to resolve these problems by pre-processing elementary deformations (see, for example, [1]). However, this is usually very time consuming for complex objects.
Different approaches use artificial neural networks to train and simulate the behavior of physics-based model structures (e.g. [4]). These approaches have the disadvantage that incisions could not be made during the simulation process, which is necessary for the simulation of surgical training procedures. Besides the neural networks have to be trained off-line by use of observation data of physics based models, even if some expert knowledge about the physical model is available. Another approach to solve some of the problems mentioned above is the use of neuro-fuzzy systems.

Simulation using neuro-fuzzy systems

Neuro-fuzzy systems combines the capabilities of neural networks (e.g. the ability to learn) and fuzzy systems (e.g. interpretability) [7][16]. In [12] we proposed a simulation model, which was motivated by a combination of a fuzzy system and an artificial (recurrent) neural network, a so-called hybrid neuro-fuzzy system. The fuzzy system enables the definition of system parameters by use of prior expert knowledge. To define the visual response of the tissue's model, for example, medical terms can be used [14].

In this paper we present an appropriate learning algorithm for our model. The algorithm uses the positions of specific object points during discrete time steps to learn the required physical parameters. The time series data for learning can be derived, for example, by a time dependent optical measurement of an object under influence of external forces. So, the sometimes unconvincing (biological) realism of manual defined spring-mass models [1] can be resolved.

Vectorized neural networks

A vectorization of a (feedforward or recurrent) neural network can be done by replacing the (scalar) processing elements of conventional networks (net input function *net*, transfer function *f* and output function *o*) by 'vector processing' units and by vectorizing the connections between the neurons. The weights of the network remain scalars.

The propagation process in such networks can be done in the same way as in conventional (feedforward or recurrent) neural networks. Also, existing learning procedures can be used to train them, since the principle structure, data flow and processing remain unchanged. An example of a modified learning algorithm for a vectorized recurrent network – to which we will refer in the description of the learning algorithm for the spring-mass model – is presented in the following.

The vectorization of a network can be useful to simplify the representation of specific problems. For example, if multiple inputs should be jointly processed, e.g. the weights should be modified accordingly, or if vector operations, e.g. scalar products, are needed. Vectorized networks can be transferred into conventional scalar networks if they do not use any specific vector operations. This can be done by creating a copy of each neuron and its connections – which must share the same weight (even during learning) – for every dimension.

Vectorized BPTT.

A popular learning method for recurrent neural networks is backpropagation through time (BPTT) [17][18]. The main idea of this method is to transfer the recurrent network into a feedforward network by unfolding it in time [6]. Then, a conventional backpropagation learning method can be applied. The vectorized version of this algorithm can be described as follows. The propagation function of a vectorized network, can be defined as

$$\vec{o}_j(t) = \vec{f}(n\vec{e}t_j(t)) = \vec{f}(\sum_i \vec{o}_i(t) w_{ij} + e\vec{x}t_j(t)) \tag{2.1}$$

where $\vec{o}_j(t)$ is the output of neuron *j* at time *t* and $e\vec{x}t_j(t)$ the (external) input to the neuron (if any).

Let E be the total error, respectively the cost function to be minimized, of all output neurons k over all time steps *t*=0, ..., *T*:

$$E = \sum_{t=0}^{T} E(t) = \sum_{t=0}^{T} \left[\frac{1}{2} \sum_k (\vec{E}_k(t))^2 \right] \tag{2.2}$$

with

$$\vec{E}_k(t) = \begin{cases} \vec{p}_k(t) - \vec{o}_k(t) & \text{if node k has a target output } \vec{p}_k \text{ at time } t, \\ 0 & \text{otherwise.} \end{cases} \tag{2.3}$$

be an error measure for node k. Then the error gradient can be derived as usual by defining

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = -\eta \sum_{t=1}^{T} \frac{\partial E(t)}{\partial n\vec{e}t_j(t)} \frac{\partial n\vec{e}t_j(t)}{\partial w_{ij}} \,. \tag{2.4}$$

The error signal $\delta(t)$ for every neuron can be defined as

$$\vec{\delta}_j(t) = -\frac{\partial E(t)}{\partial n\vec{e}t_j(t)} = -\frac{\partial E(t)}{\partial \vec{o}_j(t)}\frac{\partial \vec{o}_j(t)}{\partial n\vec{e}t_j(t)} \tag{2.5}$$

Thus, with

$$\vec{A}_j(t) = \vec{E}_j(t) + \sum_k \vec{\delta}_k(t+1)w_{ik}$$

the specific error signals can be obtained:

$$\vec{\delta}_j(t) = \begin{cases} \vec{f}\,'(n\vec{e}t_j(t))\vec{E}_j(t) & \text{if } t = T, \\ \vec{f}\,'(n\vec{e}t_j(t))\vec{A}_j(t) & \text{if } 1 \le t < T. \end{cases} \tag{2.6}$$

With equation (2.4) the gradients can be derived after the propagation over all time steps:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = \eta \sum_{t=1}^{T} \vec{\delta}_j(t)\vec{o}_i(t-1) \tag{2.7}$$

It should be noted, that the only difference to the standard BPTT algorithm is the use of the scalar product in equations (2.2), (2.6) and (2.7). Thus, for example, the error rate $\delta(t)$ depends on the vector component of the local error vector in direction to the vector of the local gradient of the transfer function *f*.

THE SPRING-MASS MODEL NETWORK

The network is defined analogous to the construction of a physical spring-mass model [22]: According to a physical 3D spring-mass mesh, groups of neurons calculating the spring dynamic are connected to groups of neurons calculating the mass node dynamic (see [8]). In the following, the structure of the recurrent neural network is defined slightly different from the definition in [8] to simplify the derivation of the learning algorithm.
The group of 'node-neurons' consists of three neurons: The 'position-neuron' calculates the position *p* of the assigned node based on its current position and its velocity, the 'velocity-neuron' calculates the velocity *v* based on the nodes velocity and its acceleration and the 'acceleration-neuron', which calculates the nodes acceleration *a* based on the forces of the connected springs and external forces. With t_c be the time constant for the simulation, these neurons are defined as:

$$\vec{o}_p(t) = \vec{f}_p(n\vec{e}t_p(t)) = t_c\vec{o}_v(t) + \vec{o}_p(t-1) \tag{3.1}$$

$$\vec{o}_v(t) = \vec{f}_v(n\vec{e}t_v(t)) = t_c\vec{o}_a(t) + \vec{o}_v(t-1) \tag{3.2}$$

$$\vec{o}_a(t) = \vec{f}_a(n\vec{e}t_a(t)) = w_m(\sum_i w_i\vec{o}_{f_i}(t) + ext_f(t)) \tag{3.3}$$

where o_{f_i} is the force of the connected spring *i*. w_i defines the direction of the force (see Figure 1), $w_m := m^{-1}$ and *m* is the mass of the node, and ext_f is an external force applied to the node.
The group of 'spring-neurons' consists of three neurons: The 'position-force-neuron' calculates the static spring force *s* based on the positions of the connected nodes, the 'viscosity-neuron' calculates the spring viscosity *c* based on the relative velocities of the connected nodes and the 'force-neuron', which combines these forces to a single output force *f* of the spring. These neurons are defined as:

$$\vec{o}_f(t) = \vec{f}_f(n\vec{e}t_f(t)) = c_c\vec{o}_c(t) + s_c\vec{o}_s(t) \tag{3.4}$$

$$\vec{o}_s(t) = \vec{f}_s(n\vec{e}t_s(t)) = \vec{f}_s(\vec{o}_{p_1}(t) - \vec{o}_{p_2}(t)) \tag{3.5}$$

$$\vec{o}_c(t) = \vec{f}_c(n\vec{e}t_c(t)) = \vec{f}_c(\vec{o}_{v_1}(t) - \vec{o}_{v_2}(t)) \tag{3.6}$$

where c_s defines the physical spring constant, s_c the viscosity constant, and f_c, f_s the spring- and viscosity functions of the spring. Connected groups of two node-neurons and one spring-neuron are depicted in Figure 1.
(Remark: The transfer functions in (3.5) and (3.6) are more complex and could be implemented by networks with multiplicative connections (see, for example, [11]). Since these functions are not needed for the derivation of the learning algorithm, we omit it for simplification.)

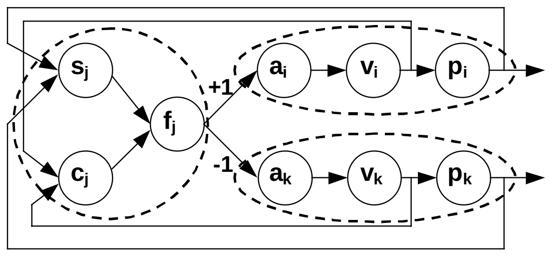

Figure 1. Two nodes (i and k) connected by a spring (j)

LEARNING CONSIDERATIONS

The main problem of the propagation process [13] and especially a probable learning algorithm is the large number of time steps between two attractors of the network (when the external forces and the internal forces are compensating each other). Furthermore, it must be possible to learn by use of time series data, which is obtained by changing external forces, e.g. data obtained by an optical measurement of a deformable object under influence of a continuously increasing external force. Thus – the very efficient learning algorithm – backpropagation through time (BPTT) [17][18], cannot be used. A further problem of this learning method is that the (position) error of each node can increase drastically over the – in most cases very long – time sequences, even if the network parameters (e.g. spring constants) are only slightly different from the 'required' settings. Thus, the total error obtained from the output of the 'unfolded' network cannot be used for efficient learning.
Likewise, the use of (offline) real-time recurrent learning (RTRL) [23] is not recommended due to the problems mentioned above. The online variant with 'teacher forcing' [23] seems to be more appropriate. This learning method modifies the weights of the network after each time step and sets the output $o_k(t)$ of a node k to the desired output $p_k(t)$, if available, after the derivation of the error $E(t)$. Thus, the network is 'teached' to join the targets. The intention is to follow the trajectories of the nodes more exactly. Nevertheless, training the network by this method is very time consuming, since the time delay between the modification of the weights (respectively masses, viscosity and spring constants) and the output of the relevant position neurons can be very large.
Schmidhuber presented a combination of RTRL- and BPTT-like learning [19]. The idea of this algorithm was to split the learning process in blocks consisting of n-time steps. Every block is trained by a combined BPTT/RTRL method starting at time step t_0. After training of the first block, the process starts over with the next block until the end of the time sequence is reached. Unfortunately, all of these gradient based learning methods have difficulties to learn if the time delay between an input and the required response is very high [20]. This is caused by the exponentional diminishing of the error signal during backpropagation. The specific structure of the considered network shows quite well one problem in learning: If there is a large error at the final time step T, it can be useless to start backpropagation at this time step, since the (relative and absolute) error of the positions can lead to unusable error values. This can result in a divergence of the learning method or, in a less worse case, in an undesired change of the weights. Sometimes, this can be compensated by a very low learning rate, but then resulting in a bad performance.
For these reasons we tried to combine the advantages of the above mentioned learning methods with a problem specific learning algorithm for the considered network structure.

THE LEARNING ALGORITHM

The goal of the learning method is to derive the parameters of the physical spring-mass model, the masses m_j and the constants c_i and s_i. Therefore, the learning algorithm just modifies the weights defined by these constants during learning.
Due to the considerations mentioned above, the presented learning algorithm performs a single backpropagation step in time after propagation of each time step, and modifies the weights – different to the BPTT algorithm – immediately. To prevent a great deviation from the required positions a 'teacher forcing' strategy is used: After a fixed number of time steps, the outputs $o_p(t)$ (respectively the activations) of the 'position-neurons' are set to the required values $p(t)$.
The learning algorithm uses the error function defined by equations (2.2) and (2.3), where $o_k(t)$ is the position of node k and $p_k(t)$ the position of the node k in the exact physical model at time-step t. Thus, an error is only defined for the 'position-neurons' in the group of 'node-neurons'. According to the gradient-descent method, the error rates for the respective neurons are derived as follows (see also equations (2.5) and (2.6)).

Let *p(t)* be the desired output at time *t* for a 'position-neuron' *p*, then the error rate $\delta_p(t)$ is defined as:

$$\vec{\delta}_p(t) = -\frac{\partial \vec{E}(t)}{\partial n\vec{e}t_p(t)} = \vec{p}(t) - \vec{o}_p(t) \tag{3.7}$$

For the error rates of a 'velocity-neuron' *v* and a 'acceleration-neuron' *a* we obtain:

$$\vec{\delta}_v(t) = -\frac{\partial \vec{E}(t)}{\partial n\vec{e}t_v(t)} = \vec{\delta}_p(t)t_c = (\vec{p}(t) - \vec{o}_p(t))t_c \tag{3.8}$$

$$\vec{\delta}_a(t) = -\frac{\partial \vec{E}(t)}{\partial n\vec{e}t_a(t)} = \vec{\delta}_v(t)t_c = (\vec{p}(t) - \vec{o}_p(t))t_c^2 \tag{3.9}$$

With equation (3.9) the weight-gradient for w_m can be defined as:

$$\begin{aligned}\Delta w_m(t) &= \eta\delta_a(t)o_f(t) \\ &= \eta(\vec{p}(t) - \vec{o}_p(t))t_c^2 o_f(t)\end{aligned} \tag{3.10}$$

where η is a learning rate. Since t_c is constant, we define

$$\Delta w_m^*(t) = \eta_m(\vec{p}(t) - \vec{o}_p(t))o_f(t)\,. \tag{3.11}$$

with η_m be a learning rate. The error rates for the adaption of the spring constant can be obtained in the same way:

$$\vec{\delta}_f(t) = -\frac{\partial \vec{E}(t)}{\partial n\vec{e}t_f(t)} = -\frac{\partial \vec{E}(t)}{\partial \vec{o}_f(t)}\frac{\partial \vec{o}_f(t)}{\partial n\vec{e}t_f(t)} \tag{3.12}$$

With equation (3.3) we obtain

$$\begin{aligned}\frac{\partial \vec{E}(t)}{\partial \vec{o}_f(t)} &= \sum_a \frac{\partial \vec{E}(t)}{\partial n\vec{e}t_a(t)}\frac{\partial n\vec{e}t_a(t)}{\partial \vec{o}_f(t)} \\ &= -\sum_a w_a\vec{\delta}_a(t)\end{aligned} \tag{3.13}$$

So, $\delta_f(t)$ is defined as

$$\vec{\delta}_f(t) = \sum_a w_a\vec{\delta}_a(t) \tag{3.14}$$

With (3.14) the weight-gradients for the spring constants c_s and s_s can be defined:

$$\Delta w_c(t) = \eta\vec{\delta}_f(t)\vec{o}_v(t) \tag{3.15}$$

$$\Delta w_s(t) = \eta\vec{\delta}_f(t)\vec{o}_p(t) \tag{3.16}$$

Where η is a learning rate. The error rates for the 'static-force-neurons' are defined as:

$$\begin{aligned}\vec{\delta}_s(t) &= \delta_f(t)w_1 f_s'(net_s(t)) \\ &= f_s'(net_s(t))\sum_a \vec{\delta}_a(t)\end{aligned} \tag{3.17}$$

The error rates for the 'viscosity-neurons' are obtained in the same way, therefore we define:

$$\vec{\delta}_c(t) = f_c'(net_c(t))\sum_a \vec{\delta}_a(t) \tag{3.18}$$

Finally the error rates for the 'position-neurons' have to be derived.

$$\vec{\delta}_p(t) = -\frac{\partial \vec{E}(t)}{\partial n\vec{e}t_p(t)} = -\frac{\partial \vec{E}(t)}{\partial \vec{o}_p(t)}\frac{\partial \vec{o}_p(t)}{\partial n\vec{e}t_p(t)} \tag{3.19}$$

With

$$\begin{aligned}\frac{\partial \vec{E}(t)}{\partial \vec{o}_p(t)} &= \sum_{i\in\{s,c\}} \frac{\partial \vec{E}(t)}{\partial n\vec{e}t_i(t)}\frac{\partial n\vec{e}t_i(t)}{\partial \vec{o}_p(t)} \\ &= -\sum_s \vec{\delta}_s(t) - \sum_c \vec{\delta}_c(t)\end{aligned} \tag{3.20}$$

$\delta_p(t)$ is defined as

$$\vec{\delta}_f(t) = \sum_s \vec{\delta}_s(t) + \sum_c \vec{\delta}_c(t) \tag{3.21}$$

The learning algorithm adapts the weights by use of equations (3.10), (3.15), and (3.16) for every node. If there is no time series data *p(t)* for an (inner) node available, the error is propagated back from the spring neurons by use of equations (3.18) and (3.21). Then the calculation of the error rate of the 'position-neuron' (equation (3.7)) is replaced by

$$\vec{\delta}_p^*(t) = \sum_s \vec{\delta}_s(t) + \sum_c \vec{\delta}_c(t) \tag{3.22}$$

A pseudocode description of the algorithm is presented in Figure 2.

```
reset iteration counter;
do
    for t := 1 to T
        reset time step counter;
        propagate net for one time step;
        derive error $E(t)$;
        derive error rates $\delta$;
        derive weight gradients $\Delta w$;
        modify weights;
        if time step counter > teacher forcing limit then
            set node positions to required positions p(t);
        end;
    end;
    increment iteration counter;
until (error is sufficiently small
    or max. number of iterations is reached)
```

Figure 2. Learning algorithm for spring-mass model network

If the object, which has to be simulated, is consisting of homogenous material and the network structure was suitably defined, all spring constants (*c* and *s*) can be learned jointly. Thus, the performance and the stability of the learning algorithm can be improved to a great part. Even areas of the same material can be jointly processed, by slight modifications of the algorithm. Furthermore, in many cases, the mass m_j of the nodes can be predefined, by just dividing up the total mass of the object between the mass nodes of the network.

EXAMPLE

To demonstrate the usability of the presented algorithm we used a very simple but typical measurement problem: An object is deformed over a period of time by a constant external force.

As object we simulated a simple cube with diagonal springs, consisting of 27 nodes and 86 springs (see Figure 3) with predefined 'physical' parameters for the masses and springs. The bottom nodes of the cube were fixed to the ground during simulation. To the external node in the center of the top layer a constant force was applied. The deformation of the object was simulated for a period of 250 time steps. After this period, the object's nodes rest in a final position – an attractor or energy minimum. The positions of the nodes on the discrete time steps were stored as learning data.

After this generation of test data the learning process was started. For the first sample the mass, spring and viscosity constants were randomly chosen in an interval defined by [0.2x, 5x] around the original settings x. The algorithms was initialized with a learning rate $\eta = 0.01$ for the spring constants c_i and s_i, and $\eta_m = 0.1$ for the mass constants m_j. A 'teacher forcing' limit of ten iteration steps was chosen. The algorithm was terminated after 100 learning cycles (each cycle is a complete propagation through time) with an error E = 1.21. The error after each training cycle is depicted in Figure 4. As expected, due to the low amount of (different) training data, the algorithm did not retrieve the 'physical' parameters of the model. The resulting constants defining c_i, s_i and m_j were still spread around their 'exact' settings. This can lead to an undesired behavior of the object in 'untrained' situations.

The second example was initialized as above, but with a learning rate of $\eta = 0.1$ for the spring constants. Furthermore, the masses were predefined and not modified during learning. The algorithm was again terminated after 100 learning cycles, with an achieved error E = 0,033. The error after each training cycle is depicted in Figure 4. Again, the algorithm did not retrieve the 'physical' parameters of the model. The resulting constants were still spread around their 'exact' settings.

In the last example we learned the spring constants *c* and *s* jointly for all springs. The algorithm was initialized as above, but the masses were predefined and an offset was added to the spring and mass constants to prevent an accidental parameter match. The algorithm was again terminated after 100 learning cycles. The algorithm achieved an error of E = 0,138. This is slightly higher than in the other examples. The error after each training cycle is depicted in
Figure 4. In this case the algorithm was able to retrieve the physical parameters of the model, with an error of about less than 0.1%.

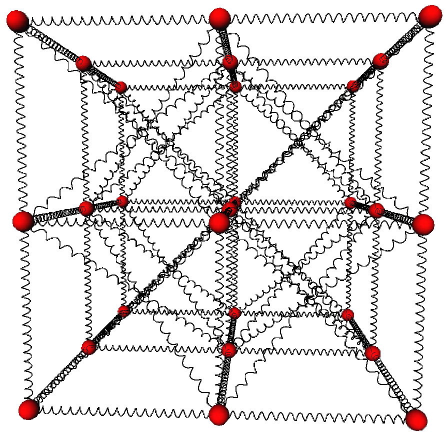
Figure 3. Elastic cube used for learning

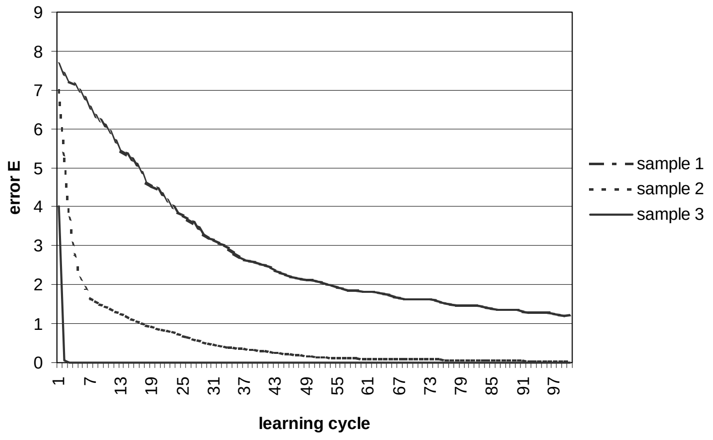

Figure 4. Learning example: Error E

CONCLUSIONS

By use of the learning algorithm, measured data (e.g. derived from an optical measurement of a deformable solid) can be used to derive the parameters of spring-mass models. Furthermore, the learning process can be initialized by use of prior knowledge based on ‘real’ physical parameters or parameters defined by the tool *Elastodynamic Shape Modeler* [14]. Since the existing propagation algorithms for these types of networks can be used in real time applications, the resulting network can be used to simulate objects in virtual environments.
It should be noted, that the ‘physical’ parameters achieved by use of the presented algorithm, need not match the parameters of the physical model, if there is not enough training data with different deformations available. In this case the algorithm may derive a combination of parameters, which follows the same (trained) trajectory, but result in different inner forces. Nevertheless, since the main objective of this algorithm is to find a combination of parameters resulting in a visual convincing simulation of the object, ‘exact’ modeling is not necessary. To improve the learning performance, known homogenous areas should share the same parameters and if the masses of the nodes are known, they should be used to set the mass constants fixed. So, the algorithm is able to derive a more ’exact’ model.
As presented, the learning algorithm can be used to derive the parameters of spring-mass models to obtain a visual convincing real-time simulation in virtual reality applications. The simulation model is already be used in different simulators for minimally invasive surgery in gynaecology [13] and neurosurgery [5][15] (see also Figure 5). Since the deformation behavior of tissues is inhomogeneous, the parameter adaption of the simulating model is very important. First approximations of the parameters can be obtained using surface deformation data of organs, which are already removed from the body. However, these organs have different deformation behaviors because they are not supplied with blood. Nevertheless, real time deformation data of living tissue can be obtained by open MR-tomography or with real-time 3D sonography. With these data, the learning algorithm can be used to improve today’s surgical simulators.

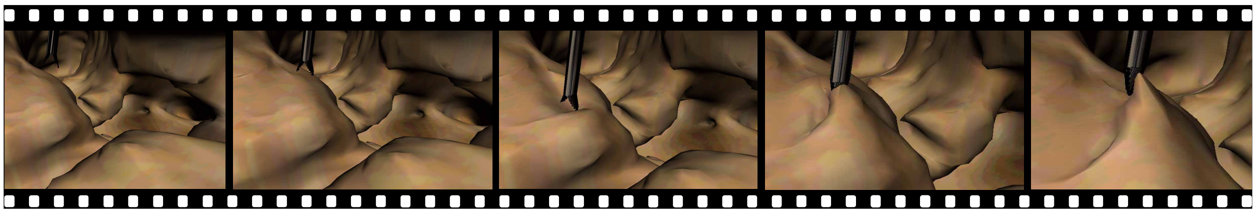
Figure 5. Virtual deformation of the ventricular system in the human brain

REFERENCES

[1] S. Cotin, H. Delingette, and N. Ayache. Efficient Linear Elastic Models of Soft Tissues for real-time surgery simulation, *INRIA report no. 3510, INRIA Sophia Antipolis*, France, 1998.

[2] S. L. Delp, P. Loan, C. Basdogan, et al., Surgical Simulation: An Emerging Technology for Training in Emergency Medicine. In *Teleoperators and Virtual Environments*, 6(4):147-159, 1997.

[3] P. L. Gould. *Introduction to Linear Elasticity*. Springer, New York, 1994.

[4] R. Grzeszczuk, D. Terzopoulos, and G. Hinton. NeuroAnimator: Fast neural network emulation and control of physics-based models, *Proc. of ACM SIGGRAPH 98 in Computer Graphics Proceedings, Annual Conference Series*, pages 9-20, Orlando, FL, July, 1998.

[5] G. Kleinszig; A. Radetzky; C. Wimmer; D. P. Pretschner; L. M. Auer: ROBO-SIM: A Simulator for Minimally Invasive Interventions. In *Scientific Program of the RSNA*, 209(P), p. 682, Chicago, 1998.

[6] M. L. Minsky and S. A. Papert. *Perceptrons* (second expanded ed. 1988), MIT Press, Cambridge, MA, 1969.

[7] D. Nauck, F. Klawonn, and R. Kruse. *Foundations of Neuro-Fuzzy Systems*, John Wiley & Sons Inc., New York, 1997.

[8] A. Nürnberger, A. Radetzky, and R. Kruse. A Problem Specific Recurrent Neural Network for the Description and Simulation of Dynamic Spring Models. *Proceedings of the International Joint Conference on Neural Networks*, pages 468-473, Anchorage, Alaska, May 1998.

[9] F. J. Pineda. Generalization of back-propagation to recurrent neural networks. In *Physical Review Letters 59*, pages 2229-2232, 1987.

[10] F. J. Pineda. Recurrent back-propagation and the dynamical approach to adaptive neural computation, *Neural Computation*, 1:161-172, 1989.

[11] J. B. Pollak. Cascaded backpropagation on dynamical connectionist networks, In *Proc. of the Ninth Conf. Cognitive Sci. Soc.*, p. 391-404, Seattle, WA, 1987

[12] A. Radetzky A. Nürnberger, D. P. Pretschner. A Neuro-Fuzzy Approach for the Description and Simulation of Elastic Tissues. In *Baumeister, M.; Hohn, H. P.; Sklorz, S. [et al.] (eds.): Multimedia Technology in Medical Training, Proc. of the European Workshop*, pages 30-40, 1997.

[13] A. Radetzky A. Nürnberger, D. P. Pretschner. Simulation of elastic tissues in virtual medicine using neuro-fuzzy systems. In Kim, Y.; Mun, S. K. (eds.): *Medical Imaging 1998: Image Display. Proc. of SPIE Vol. 3335*, pages 399-409, 1998.

[14] A. Radetzky, H. Machenski, D. P. Pretschner. A Fuzzy Parameter Estimator to Describe the Elastic Behavior of Deformable Tissues in Surgical Simulation. In *Scientific Program of the RSNA*, 209(P), p. 681, Chicago, 1998.

[15] Radetzky, Ch. Wimmer, G. Kleinszig, M. Brukner, L.M. Auer, D.P. Pretschner, Interactive Deformable Volume Graphics in Surgical Simulation. In Proc. of *International Workshop on Volume Graphics*, Swansea, GB, 1999.

[16] R. Rojas. *Neural Networks - A Systematic Introduction*, Springer, Berlin, 1996.

[17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation, In D. E. Rumelhart, J. L. McClelland, et al. (eds.), *Parallel Distributed Processing*, vol. 1, chap. 8, MIT Press, Cambridge, MA, 1986.

[18] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by backpropagation errors, *Nature*, 323:533-536, 1986.

[19] J. Schmidhuber. A fixed size storage O(n^3) time complexity learning algorithm for fully recurrent continually running networks, *Neural Computation*, 4:243-248, 1992.

[20] J. Schmidhuber. *Netzwerkarchitekturen, Zielfunktionen und Kettenregel*, Habilitation (postdoctoral thesis), Institut für Informatik, Technische Universität München, 1993.

[21] N. Suzuki, A. Hattori, S. Kai, et al.. Surgical Planning System for Soft Tissues Using Virtual Reality. In: Morgan, K.S. (eds.): *Medicine Meets Virtual Reality: Global Healthcare Grid, Vol. 39 of Studies in Health Technology and Informatics*, pages 159-163, IOS Press, Amsterdam ,1997.

[22] Wesolowski, Z (ed.). *Nonlinear Dynamics of Elastic Bodies*, Springer, Wien, 1978.

[23] R. J. Williams, and D. Zipser. Experimental analysis of the real time recurrent learning algorithm, *Connection Science 1*, pages 87-111, 1989.