

NeSS: a Simulation Environment for Behavioral Design of Neural Networks for Prediction and Control

Cesare Alippi, Fabio Casamatta, Leonardo Furlan, Andrea Pelagotti, Vincenzo Piuri
Department of Electronics and Information
Politecnico di Milano
20133 Milano, Italy

Abstract

The *NeSS* (Neural Systems Simulator) environment is presented in this paper: it is a flexible software package which has been developed to support, analyze and model dynamic non-linear systems for prediction, system identification and control applications, by providing both classical and innovative approaches within a flexible and high-level framework. The behavior of each system is easily defined in a graphic way by interconnecting parametrized atomic objects (e.g., algebraic functions and neural networks), whose behaviors can be either predefined or identified by means of a learning procedure. Neural networks play a relevant role in *NeSS*: rich and easily expandable libraries are given which support different neural structures and learning algorithms.

1. Introduction

The design of the control scheme for a given plant requires a fine modeling of the whole process; in general, this is performed by exploiting the available a priori knowledge, determining physical-driven equations, and identifying the unknown parts with suitable identification and prediction techniques. Identification and prediction of some parts of the plant can nowadays be successfully solved even when the system is strongly non linear by relying on new information technologies such as the ones provided by neural networks [1] and fuzzy logic [2].

Neural paradigms are parallel distributed processing models composed of simple non-linear computational units interconnected to constitute a network. Their increasing interest and popularity in several application fields is strictly related to the proved flexibility and effectiveness in solving complex non-linear tasks [3], e.g., learning a function, predicting time series, controlling and identifying a process (therefore enriching more classical system identification models with non-linear ones).

A neural network evolves and learns the behavior of a system by adapting its internal parameters without requiring any information about the internal behavior of the system itself. Learning is based on a set of input-output pairs coming from the system to be identified (generally, the input-output pairs refer directly to the inputs and the outputs of system) and it is carried out by minimizing an error function which measures the discrepancy between the process output and the one provided by the neural model.

A modern approach to deal with a model-based description of a process is to decompose, according to a top-down philosophy, the whole system in sub-parts and treat each of them as a separate module. Each module can be seen as a black box connected with the outside world via input and output ports; its behavior is characterized by the mathematical description ruling its own input-output relationship.

Some relationships can be directly modeled by analytical equations (algebraic, differential, et cetera), while others are either computationally too complex or the relationship itself is analytically unknown. In the latter case, an identification procedure needs to be taken into account: it may consist either of traditional linear approaches or, whenever the system is hardly non-linear, neural networks. As a consequence, the whole system is described as a mixture of algorithmic and neural elements: for each part, the approach more suited to capture its behavior is adopted.

The integrated presence of neural and algorithmic descriptions makes the simulation and the design of the control architecture a difficult task; the difficulty increases when hybrid controllers (i.e., neural and PID-like controllers) are envisaged, since the training of the neural controller evolves in a closed-loop configuration.

To deal with such models, dedicated CAD tools have been developed; they mainly deal with classical approaches. The most sophisticated of them support the design of hybrid systems with an off-line training modality, i.e., the neural component is trained separately and, once the optimal weights configuration has been determined, it is cast into the plant's model as an identified module. Examples of these neural networks software tools can be found in [4-6] and in the web site [7]. However, two drawbacks are associated with off-line training:

- generation of the training pairs is not straightforward since, in general, we do not know the controller's behavior,
- good performance in validation does not imply a sufficient accuracy once the model is inserted in the final working configuration.

Conversely, these problems do not arise in on-line identifications.

To provide a flexible software environment with powerful on-line identification tools and a friendly user interface, we developed *NeSS*, the *Neural System Simulator* presented in this paper.

The behavior of the process is defined graphically by interconnecting each of its modules by using a graphical editor. Modules are represented as black boxes with input and output connections and are linked each other to generate the functional graph of the system. Each component is functionally defined by a vector Φ of parameters representing the physical insights of the element; for example, for a sinusoidal function $y = A \sin(\omega t + \varphi)$, the vector contains the variables $\Phi = [A, \omega, \varphi]$. Such parameters may be either fixed or time-dependent.

As far as a neural component is concerned, a library of neural models is provided, containing the most popular and frequently used topologies (e.g., multi-layered feed-forward [1], virtual neurons topology [8]). For each neural element, the user has to define

the topological complexity of the network in terms of number of layers and neurons per layer: in this case, the Φ vector contains the neural network's weights and biases.

A configuration procedure allows to train one or more components in parallel; this implies that parameters contained in each Φ can be identified via learning. The training procedure can be dedicated and tailored to the specific application; according to this philosophy, the function to be minimized and the learning algorithm can be directly created by the user. To ease this task, libraries of pre-defined error functions and training procedures are provided, covering the most common cases.

In section 2, the overall architecture of NeSS is presented, while details related to the data structures and the dynamic system model are described in section 3. In section 4, we present the features provided by the graphical editor and how to design a system; simulation tools are finally described in section 5.

2. The overall architecture

The main guidelines followed in designing NeSS can be summarized as readability, efficiency, portability, and reusability, in order to simplify future software modifications and extensions; for these reasons, the code was written in C++ within the MS-Windows operating system on IBM-compatible personal computers.

The NeSS architecture is composed by four fundamental modules: the user interface, the graphical editor, the simulation engine, and the underlying data structure management. Such partitioning allows portability since graphical aspects are separated from mathematical and algorithmic ones. For example, a change in the host platform will only interfere with graphical primitives; likewise, changes in the simulation engine can be made independently from modifications both of the graphical interface and the data structure.

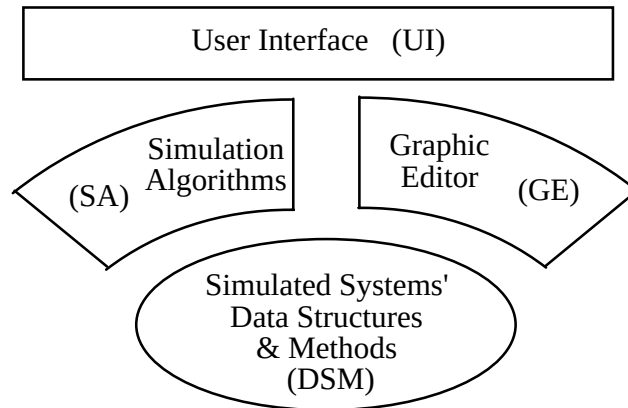


Fig. 1 - The overall architecture of NeSS.

The whole environment results in the layered structure shown in Fig. 1. The information interchange between the user and NeSS is ruled by the user interface (UI). Such a module is designed to provide a friendly and powerful environment, in which the user can easily design and interconnect systems as well as set up and run personalized simulation tasks. The UI offers different pop-up menus which provide the access to all

features supplied by the graphical editor (GE) and the simulation algorithms module (SA).

The graphical editor GE allows to design - easily and intuitively - the model of a generic system. In a window-based environment, the user can draw the internal modularized structure of the system to be modeled, simulate it by configuring all the neighboring components and suitably connect them. Utilities are provided which simplify drawing and linking operations as well as files management.

System simulation is executed by using the algorithms available in the SA module. This module implements all methods which are usually adopted to perform numerical analysis and learning. Simulation procedures can be tailored to the specific application by setting the configuration parameters.

Since NeSS is implemented by relying on the object-oriented philosophy, the system, the functions, and the graphical output windows are defined as objects and can be manipulated by means of the corresponding procedures (i.e., their methods, in the object-oriented terminology). The module DSM groups these objects and their methods together.

3. The system model and the data structure

In control theories, systems are classified in two categories: *static* and *dynamic*. In the former, the outputs depends only on the current values of the inputs and the behavior can be described by algebraic or trigonometric equations. Conversely, dynamic systems' outputs are related also to the internal state of the system itself and, as a consequence, to the history of the system; regressive models with feedback loops including the state variables are therefore required to describe the system behavior. All these kinds of systems can be modeled in NeSS at a high-abstraction level: a system is viewed as a black box characterized by its inputs, outputs, and transfer functions.

The user hierarchically creates the model of his system (e.g., a plant, a machinery): the top-level system is partitioned into sub-systems, internally connected each others and externally communicating via *connectors*. Connectors represents the system interface, i.e., the primary input and output signals exchanged by the whole system with the external environment. In turn, each sub-system can be recursively partitioned in lower-level sub-systems, until elementary components are reached. These components, called *functions*, are the basic atomic building blocks composing the system; for example, they can be sums, products, logic operators, and neural nets. The system's transfer function is therefore described hierarchically by interconnecting the transfer functions of the atomic components. In Fig. 2, a simple system characterized by the transfer function $y = \sin(x_1 + x_2)$ is shown.

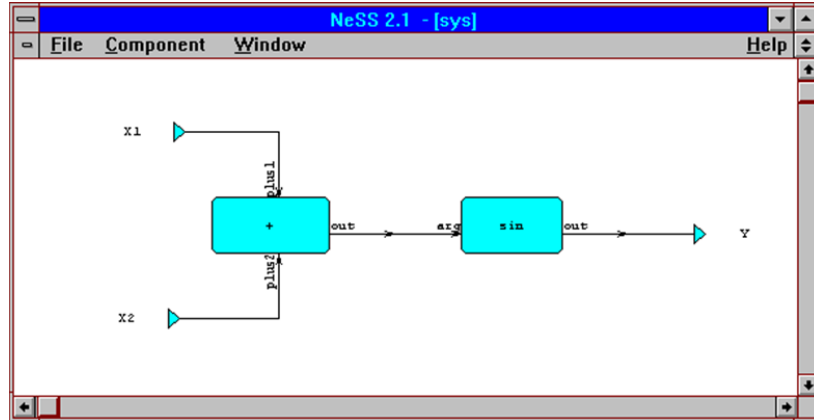


Fig. 2 - Graphical structure of the function $y = \sin(x_1 + x_2)$.

The internal representation of a dynamic system in NeSS must be coherent with this characteristic view. A comprehensive and homogeneous data structure must identically treat functions, connectors, and systems as mathematical operators characterized by a behavioral description of their input-output representation.

The data structure representing a dynamic system can be naturally described by using an object-oriented description of a class hierarchy based on inheritance, as shown in Fig. 3. Such a structure takes advantage from the dynamic binding mechanism which is used to activate member functions by using pointers to objects created previously. Each object will be briefly analyzed in the following.

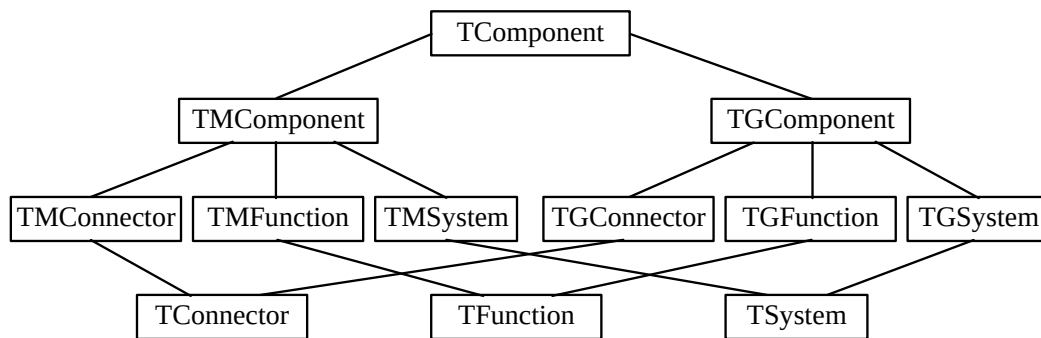


Fig. 3 - Architecture of the DSM module.

The fundamental base class of the NeSS data structure for the system model is *TComponent*, as defined in Fig. 4: it is characterized by features inherited by other basic objects (e.g., alphanumeric identifiers uniquely specify component and I/O primitives, thus allowing operations related to memory storage). Systems, connectors, and functions inherit then their own features from the same base class *TComponent*.

```

class TComponent
// Base class for all components.
{
    public:
        char Name[LENGHT_COMP_NAME + 1];

        TComponent(void);
        virtual ~TComponent(void);
        virtual void Read(istream& is);
        virtual void Write(ostream& os);
        friend istream& operator >> (istream& is, TFunction* pFunction);
        friend ostream& operator << (ostream& os, TFunction* pFunction);
};

```

Fig. 4 - Declaration of TComponent.

Inheritance has been exploited in NeSS also in a finer way by using *multiple inheritance*. Mathematical and graphical properties, which are shared by all components, are separated in two intermediate-level classes (*TMComponent* and *TGComponent*), which inherit from the same base class *TComponent*.

The *TMComponent* class (see Fig. 5) adds virtual methods to the base class, in order to support mathematical processing (e.g., primitives to allocate memory during computation), search for components which are connected to the input ports, and computation of the output vector. These features are virtual, i.e., their definitions are postponed till their actual instantiation: in fact, they must be implemented according to the particular requirements of the object inheriting from this class.

```

class TMComponent : public TComponent
// This class inherits from TComponent and specialises it for mathematical computation.
{
    public:
        TMComponent(void);
        virtual ~TMComponent(void);
        virtual void FindInputComp(void)=0;           // It looks for components which
supply inputs.
        virtual int StatusMemoryAlloc(int) = 0;       // It allocates memory to store the status
during
        // the computation.
        virtual void StatusMemoryFree(int) = 0;       // It frees the status memory.
        virtual void Compute()=0;                    // It computes the output
vector.
};

```

Fig. 5 - Declaration of TMComponent.

TGComponent specializes *Tcomponent* (see Fig. 6) by adding features for graphical processing. It provides variables for graphical display (e.g., shape and screen position) and lists of connected ports at each side of the component box (*GPortUpList*, *GPortDownList*, *GPortLeftList*, *GPortRightList*). Besides, this class supplies methods to manage object drawing (*Draw*), connections drawing (*DrawConnections*, *Disconnect*), port modification (*FindPort*, *DeletePort*), and other geometric primitives (*InternalPoint*, *GetRect*). The method *Draw* is declared as virtual: it draws only parts shared by all types of components, i.e., the name and the I/O ports; as a consequence, classes which inherit

from TGComponent must add new functions to this method to complete their drawing abilities.

```

class TGComponent : public TComponent
// This class inherits from TComponent and specialises it for graphic operations.
{
    protected:
        int x,y;           // Coordinates on the screen.
        int w,h; // Dimensions.

        virtual void Read(istream& file);
        virtual void Write(ostream& file);

    public:
        TGPortList GPortUpList;           // Lists of I/O ports.
        TGPortList GPortDownList;
        TGPortList GPortRightList;
        TGPortList GPortLeftList;

        TGComponent();
        virtual ~TGComponent();
        virtual void Draw(HDC dc);          // It draws the component in the current
position.
        void DrawConnections(HDC dc); // It draws the component's connections.
        BOOL InternalPoint(int x, int y); // It checks if a point is internal to the component's
square.
        RECT GetRect();                    // It gets the occupation
area.
        TGPort* FindPort(char* portName); // It looks for the I/O port.
        void DeletePort(TGPort* pGPort);
        void Disconnect();                // It
deletes all connections with other components.
};

```

Fig. 6 - Declaration of TGComponent.

Properties of TMComponent and TGComponent are then extended and specialized for individual components by another level in the class hierarchy. The classes dealing with mathematical properties are *TMConnector*, *TMFunction*, and *TMSystem*; they provide data and methods to support simulations. The classes addressing the graphic features are *TGConnector*, *TGFunction*, and *TGSystem*; they are data structures and methods for graphic descriptions of dynamic systems (i.e., mainly the operations related to the graphical editor).

The above classes contain also methods to compute the output of a system given its inputs as well as the gradient of its outputs with respect to the free parameters; this features are particularly useful for neural modeling. An output buffer is associated to each component so as to store the last computed vector. The evaluation of the output of any component in the system hierarchy is performed in three steps:

1. the input vector is built by exploiting the information contained in the output buffers of the components connected to the inputs of the considered component,
2. the output vector is then computed by applying the rules associated with the component itself,

3. the output vector is finally stored in the output buffer of the component.

As examples of these extensions, we present the function-type components in Figs. 7 and 8. TMFunction specializes TMComponent by defining the procedure *compute()*, which generates the output vector from the inputs by applying the component transfer function. In TGFunction, a new procedure *Draw* is defined: it calls TGComponent::Draw and then completes the drawing by tracing the rounded block which represents the object on the screen.

```
class TMFunction : public TMComponent
// This class defines for a generic system the data structure and methods needed for the mathematical
// simulation.
{
    public:
        TMFunction(void);
        virtual ~TMFunction(void);
        virtual void FindInputComp(void);           // It looks for components which supply
inputs.
        virtual int StatusMemoryAlloc(int);         // It allocates memory to store the status
during
        // the computation.
        virtual void StatusMemoryFree(int);         // It frees the status memory.
        void Compute(void);
};
```

Fig. 7 - Declaration of TMFunction.

```
class TGFunction : public TGComponent
// This class defines for a generic system the data structure and methods needed for the graphic
// operations
{
    protected:
        void Read(istream& file);
        void Write(ostream& file);
    public:
        TGFunction();
        virtual ~TGFunction();
        void Draw(HDC dc);
};
```

Fig. 8 - Declaration of TGFunction.

Final classes used in the system model are *TConnector*, *TFunction*, and *TSystem*: these classes inherit both from the mathematical (TM) and the graphical (TG) classes, i.e., object in these classes are in the C++ *is-a* relationship both with the mathematical and the graphical classes. For example, *TFunction* (see Fig. 9) inherits both from *TMFunction* and *TGFunction*: it provides all procedures exported by the two classes and is the base class for the hierarchy of elementary operators. In NeSS, the user can choose among about fifty elementary functions, algebraic and logic operators, and other several widely-used functions. In the present version, we implemented two types of neural networks which are significantly used in identification and control: the multi-layered perceptron (either with sigmoidal or linear outputs) and the memory neuron networks [8].

```

class TFunction : public TGFunction, public TMFunction
// This class inherits from TGFunction and TMFunction. It defines the complete data structure and methods
// for a system.
{
    protected:
        virtual void Read(istream& is);
        virtual void Write(ostream& os);
    public:
        TFunction *pParent;    // Pointer to the system whose GCompList contains "this".

        TFunction();
        TFunction(TFunction* Parent);
        virtual ~TFunction();
};

```

Fig. 9 - Declaration of TFunction.

For a *TSystem* object, the computation method produces, in a recursive way, a sequence of calls to the computation methods of operators and to the *TSystem* components contained in the object itself. All these calls are executed by exploiting the polymorphism mechanism.

NeSS has been designed as an "open" system in order to deal efficiently with particularly complex mathematical models. In such a case, a dedicated procedure can be written in any programming language and put into the operators class library, so that simulation can be speeded up considerably.

4. The graphic editor

The structure of a dynamic system can be easily defined by the user through the graphic editor GE, which provides several drawing features and filing management support.

In NeSS, the system graphic representation adheres to the hierarchical structure of the system itself. The entity at the top level is the *project*, which represents the whole system: it contains the information related to the system specifications and interconnections and the simulation options. To each project is associated a window, in which the system model can be designed by placing and interconnecting various components. New components can be instanced by selecting them in the menu *Component* (see Fig. 10). A connection between two blocks can be drawn by selecting the blocks with the mouse. Connections are relocatable around the blocks in order to support possible restructuring both of the graphic representation and of the signals that must be linked.

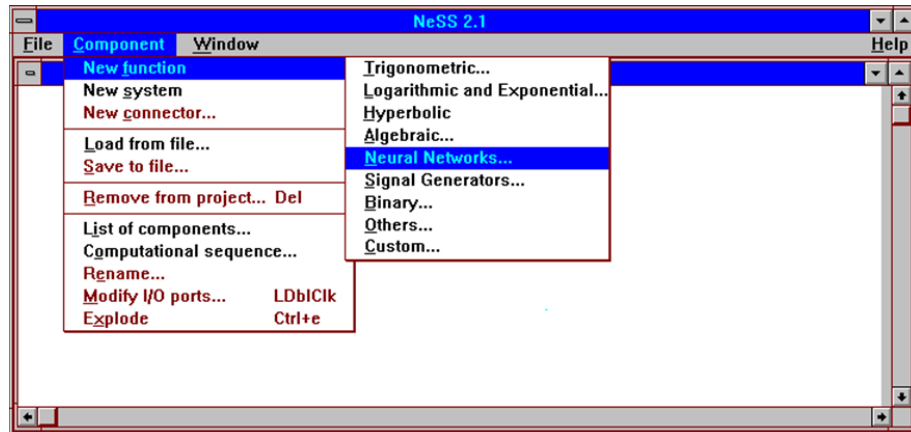


Fig. 10 - Types of function.

The graphical editor allows to define the project architecture by drawing the blocks representing the components in the window working space and by connecting them through interactive mouse-based operations. Systems are represented by rectangles, functions by rounded rectangles, and connectors by small triangles which suggest the direction of the port (i.e., input or output), as shown in Fig. 11. Functions and connectors are terminal objects, i.e., they are leaves in the tree representing the hierarchy of the project's components.

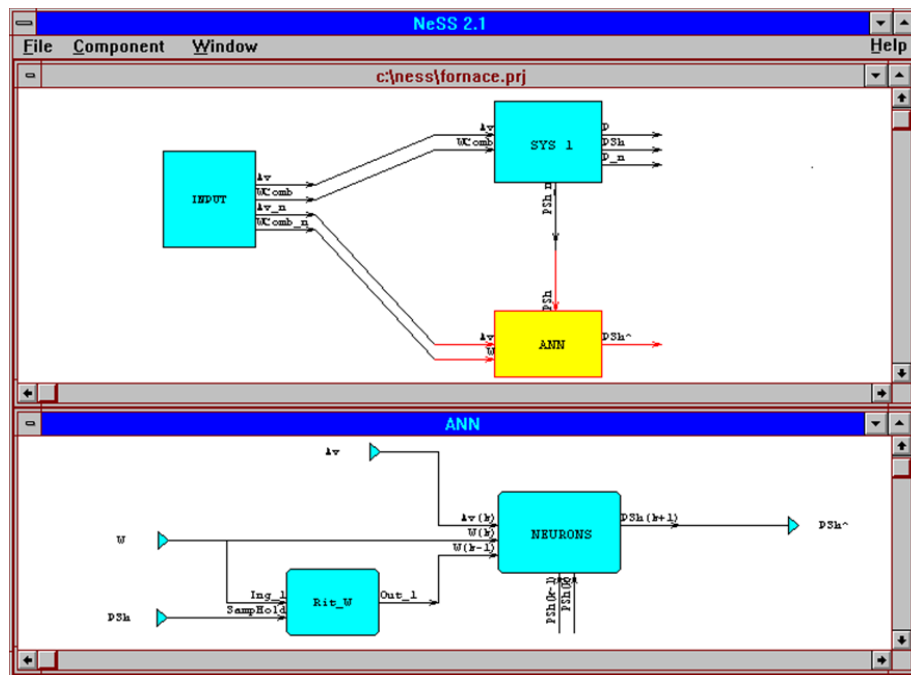


Fig. 11 - The project's architecture (upper part) and the subsystem's internal structure (lower part).

The view of blocks representing system objects can be expanded to occupy a window; each of them contains the internal components used to implement the system transfer function, i.e., the lower level components. This approach can be recursively used

to design a complex plant architecture, which can be therefore viewed as a tree whose root is the project itself and whose leaves are functions and connectors. An example is given in Fig. 11, where the upper window is associated with the project, while the lower window contains the exploded view of one of its components.

The menu items *File* and *Component* allow to access to the filing management primitives, i.e., project and components loading and saving in a file. To save a project means to store its architecture (i.e., components and interconnections) as well as its state, namely the current value of the dynamic parameters and the global features of the simulation. This allows to stop a simulation, save its state on a file and then restart it exactly from the point at which it was pre-empted. Each component can be independently saved on a file, in order to create a personalized library of reusable objects.

The *Component* menu supplies blocks management facilities such as component renaming, searching in working space, I/O relocation, and creation of new blocks.

5. The simulation engine

NeSS provides a sequential simulator for the dynamic system designed by the user. It includes the simulation engine as well as the auxiliary procedures for simulation setup and presentation of results both in text and graphic formats. The structure of the simulation module is shown in Fig. 12.

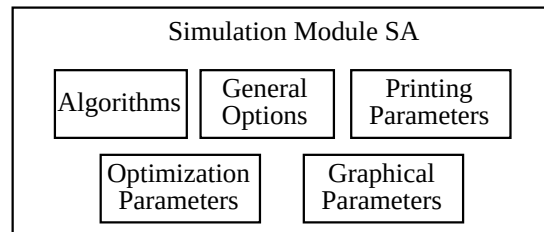


Fig. 12 - The module SA.

Simulation uses information and elementary methods contained in the system data structure. Two kinds of simulation can be generally envisioned: *training* and *validation*. Training allows to tailor the system model on the specific application by adapting the model parameters to the application data. Validation is directed to observe and evaluate the performance of a given system model on a data set or directly on data coming from the real physical system.

In both cases, the user must first select and load the description of the desired system. All components and subsystems are then serially simulated, starting from the primary inputs of the system towards the primary output. This sequential approach avoids possible problems related to synchronization, which constitute the main drawback in parallel simulators working with shared-memory data structures.

Let's consider first the *validation mode*: each dynamic system's configuration (i.e., its parameters) is assumed known, being eventually the result of a suitable training phase. Validation is performed by exciting the top-level system with the user-defined input signals. According to the internal architecture of the system, signals are propagated to the

system's components until the atomic ones are reached. At this point, the computation can be performed by using directly the methods of the atomic components and results may be returned backward to the upper level. The computation propagates from the bottom to the top level until the overall system is solved.

In the *training mode*, the configuration parameters of the elementary operators (including the neural networks) are optimized according to the user-selected goal. Since elementary operators contain methods to compute the first derivatives of the outputs with respect to their parameters, we can define training algorithms using gradient information.

As in the validation mode, the gradient of the top-level system is evaluated recursively by propagating the computation downward to the atomic components. Gradient computation is completed in two steps. In the first one, the error function derivatives with respect to the outputs of the operator under training are computed by using a numerical algorithm to guarantee the simulator generality. In the second step, the derivatives of the operator's outputs with respect to the operator's parameters are computed. The gradient vector is then obtained by multiplying these two derivative matrices. NeSS supports different techniques for gradient computation [9] to match the specific characteristics of the envisioned application. Gradients may be computed according to different granularities of the training data representation, i.e., to different (single or multiple) sizes of the training window on sampled data. We can evaluate the gradient in correspondence with each individual pattern or at the end of the whole training set (batch learning). Besides, both fixed-step gradient and conjugate gradient algorithms have been implemented in the library of the training procedures.

Different training methods have been implemented to take into account effects induced by recursion in dynamic systems. To this end, recurrent neural networks can be configured by using recurrent training algorithms [10].

Simulation options can be set in the menu item *Simulation* by selecting the desired configuration through the dialog windows. Adaptation of each model parameter is performed according to the error between the selected signal and the reference value. If required, no optimization is performed: this is useful to test the outputs of a non-adaptive classical control system. The user can select one or more signals that will be displayed in text and/or graphic formats (see Fig. 13); data outputs can be sampled at a user-defined frequency.

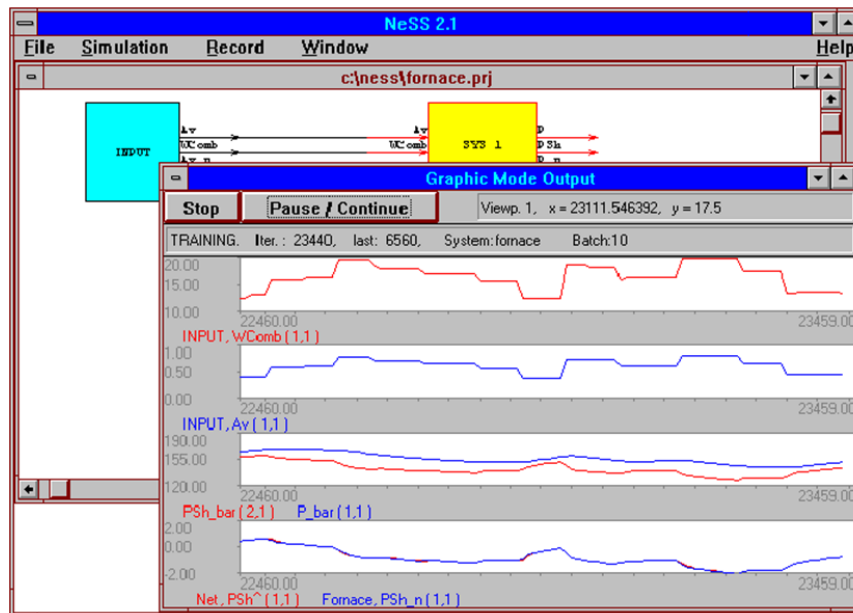


Fig. 13 - Data output in the graphic format.

The menu item *Record* allows the user to save or load files containing the status of the simulation, the data produced during the simulations, and the parameters of the optimized operators. Data can be saved on files in the Microsoft Winword or Excel format, so that further data processing may be performed by using either these tools or others which can load the same file format.

6. Conclusions

NeSS is a new flexible simulation environment which has been proved very effective to design complex heterogeneous systems for signal identification and system control. By supporting both various neural networks and algebraic/trigonometric functions, NeSS allows to create, configure, and simulate complex dynamic non-linear systems in which algorithmic components are integrated with neural ones. Optimization processes have been defined to adapt system parameters in order to minimize a user-defined error function.

Current research is addressing problems related to the connection of the simulator to data coming directly from the real plant in real time. This can be obtained by using I/O devices connected with the personal computer running the simulator. This facility is particularly attractive since it will allow real-time monitoring as well as it will support users' on-line decisions. A second extension of the system will deal with the inclusion of fuzzy components as elementary objects of the modeling environment.

References

- [1] J. Hertz, A. Krogh, R. G. Palmer, "Introduction to the Theory of Neural Computation", Addison-Wesley Publishing Co., 1991.
- [2] H. J. Zimmermann "Fuzzy Set Theory and its Applications", (2nd edition) Kluwer Academic Publishers, Boston, 1991.

- [3] B. Kosko, "*Neural Networks for Signal Processing*", Prentice-Hall International Editions, 1992.
- [4] SIMULINK, MATLAB with SIMULINK, The MathWorks, 1993.
- [5] ECANSE, Environment for Computer Aided Neural Software Engineering, Siemens Austria.
- [6] Neuroshell and NeuroWindows, Ward System Group, Inc.
- [7] Commercial and Shareware neural software tools, see the web site <http://www.emsl.pnl.gov:2080/docs/cie/neural/systems/>
- [8] P. S. Sastry, K. P. Unnikrishnan, G. Santharam, "*Memory Neuron Networks for Identification and control of Dynamical Systems*", IEEE Transactions on Neural Networks, March 1994.
- [9] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, "*Numerical Recipes in C, the Art of Scientific Computing*", 2nd Edition, Cambridge University Press, 1992.
- [10] R. Williams, D. Zipser, "*A Learning Algorithm for continually running fully recurrent networks*", Neural Computation 1, 270-280, 1989.