NST in 8+ϵ pages

Helge Ritter, September 95

Department of Information Science
Bielefeld University email: helge@techfak.uni-bielefeld.de

1 Synopsis

NSToffers a framework for constructing artificial neural systems from a comprehensive library of reusable software components, called "units". Each NST-unit implements a particular function in the form of an object that can be executed (and be subjected to a small number of additional operations, such as initialization, adaptation, loading or saving its current state).

The range of available units ranges from different types of single neurons over neuron layers to entire network modules of different types. Additional units implement operations such as control flow, access of data files, transformation, display and plotting of various types of data and many others.

Most units have inputs and outputs that allow to connect them with other units. In this way, one can form "circuits" that combine the functions of several units in a desired way. Once a generally useful circuit has been built, it can be "encapsulated" and made available as a new unit.

2 NST and C

NSTis strongly based on C. The framework of NSTfollows closely many concepts that are familiar from the C-language. Basically, each C-program is a block of statements in which further blocks may be nested:

```
program = { {...} ... {{...}{...}} }
```

Executing a block causes execution of its sub-blocks in the order of their appearance. Each block can have local variables. Function declarations and parameter lists are the mechanisms to invoke a block via a name and to copy values into some of its local variables.

In NST, the equivalent of a block is a *unit*. Like a block in C, a NST-unit can contain an ordered sequence of subunits. Likewise, when a unit is executed, the sequence of its subunits is executed. However, NST-units provide the following important extensions over C-blocks:

- while a C-block is always newly created when it is executed, and ceases to exist after execution is completed, creation and execution of NST-units are strictly separated.
- As a consequence, a NST-unit has a persistent existence and – unlike C-blocks – multiple instances of the same NST-unit can exist and memorize different histories.
- in addition to simple execution, NSTallows to apply to each unit an (extensible) number of additional "methods", including saving and loading its current status to/from file, adaptation (for neural network units) and various unit-specific control operations.

These are common features of any object-oriented approach and are shared with other approaches, such as C++. NSTcombines these features with a convenient framework for combining the functions of separate units by connecting them into circuits:

- like in C, a parameter list is the mechanism to pass information for the construction of a NST-unit. For the separate execution of units, a second, separate mechanism is provided in the form of connectors.

- instead of global auxiliary variables that must be explicitly allocated in order to transfer information between C-functions, NST-units exchange information through "wires" that are automatically created when matching connectors are joined.

- NSTprovides the concept of "operator units". While a normal unit can only execute its subunits, an operator unit can additionally invoke execution (or other methods) for a subset of its successor units (the "operands") and exchange information with them according to a pattern that it (the operator unit) can control. The familiar constructs for execution flow control, such as repeated execution of a block, can be viewed as special cases of the more general operator concept.

Besides being powerful, this combination of features also forms a very good basis for a visual programming language (see Sec. 6)

3 Using NST

Solving a task with NSTinvolves three steps:

1. Express the solution to the task in the form of a circuit diagram of simpler functional units.

2. Try to break up complex units into simpler ones, hopefully ending up only with units that are provided in NST(otherwise, see Sec. 10)

3. Write a corresponding C-program.

Before we consider an example, let us fix some language. Each NST-unit has a number of inputs and outputs. Inputs and outputs of a unit are numbered from zero (separately). Inputs and outputs of different units can be connected (if their types match, cf. below) and are thus frequently referred to as "connectors". Each connector consists of one or several "pins". Again, pins of the same connector are numbered from zero. Pins cannot be decomposed further, but there are five different types that can hold a single float value ("scalar float" pin type), a fixed dimension vector of `float`s, `short int`s or `unsigned char`s ("float vector", "int vector" and "byte vector" type) or a variable length character string ("string type"). For historic reasons, the term "field" is equivalent to "connector".

4 An Example

Consider the task of time series prediction for a sequence of scalar values $x(t)$, $t = 0, 1, \ldots$. We assume that each new prediction $\hat{x}(t)$ is to be made from a pair of past values $x(t-d)$ and $x(t-d+1)$. We wish to make the prediction by training an adaptive unit with a portion of the time sequence where all values are known.

A suitable block diagram or "circuit" for performing this task is shown in Figure 1. The leftmost unit labeled `data_gen` represents our data source. Its upper connector provides the desired target value $x(t)$ that is used during training. Its lower connector provides the data from which the prediction $\hat{x}(t)$ for $x(t)$ is to be made (i.e., the vector $(x(t-d), x(t-d+1))^T$).

During training, these pairs are fed to the adaptable predictor unit, for which we choose a (`bp_unit`). The man page for this unit (available with the `man bp_unit` command) tells us that

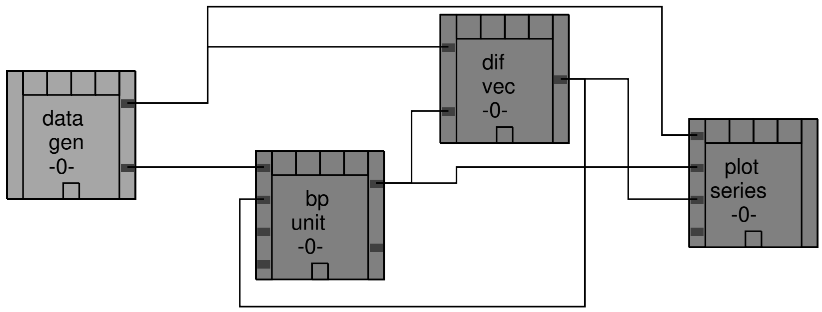

Figure 1: A time-series prediction circuit implemented by the program above.

this unit has an input for the data and for the difference error between its own output and the target value (there are further inputs and outputs, e.g., for specifying a learning rate and for back-propagating error information, which we do not require to use in this simple example). Employing an additional `dif_vec`-unit to form the error difference, we thus arrive at the block diagram shown in Fig. 1.

The additional `plot_series`-unit in Fig. 1 serves to provide a running plot of the time course of the target value $x(t)$, the prediction $\hat{x}(t)$ and the error difference between the two.

The resulting circuit can be read as a data-flow diagram for an equivalent program: units are executed sequentially from left to right, using current values at their input connectors (always located on the left side of a unit) to update the current values at their output connectors (located always on the right side). Some units have only outputs and act as "data sources", others have only inputs and act as "data sinks". Most units have both and serve to accomplish some transformation.

This finishes our first step. For the second step, we see that the `bp_unit`, the `diff_vec` unit and the `plot_vec`-unit are already primitive units that are provided in NST, but there is no `data_gen`-unit. Of course, this unit must be application dependent, since it has to provide the data we wish to use for our prediction. A realistic choice for this unit would be some predefined NST-unit that can read data from a file (such as the `scanf` or the `read_file`-unit). However, for this simple example, and to illustrate another concept (the use of subcircuits within a "container unit"), we build this unit from two simple math-units, the first one (named `wave_gen` to provide a simple sine wave for our sequence (t), and the second (named `delay_unit`), to buffer a fixed number of past values in order to provide the time shifted pair $(x(t-d), x(t-d-1)$. These units are not visible in Fig.1, which only shows the top level of the entire circuit diagram. Instead, the two new units form a little circuit of their own that has been encapsulated in the `data_gen`-unit (this "subcircuit" is just the concatenation of the `wave_gen` and the `delay_unit` and it is not shown; we will see its structure shortly in the program listing below). The unit named `data_gen` is of a general "container type" that can accommodate an arbitrary subcircuit. Such container units can be created with arbitrary configurations of input and output connectors to make data of the subcircuit accessible from outside. In the present example, the unit was created with two output connectors to provide access to $x(t)$ and $(x(t-d), x(t-d-1)$.

We are now ready to translate our circuit diagram into an equivalent C-program (step 3 above). Basically, each unit is translated into a call of a suitable constructor function that returns a handle. Subsequently, each wire is translated into the call of another constructor function that makes a connection. Finally, there are two functions for executing and adapting a single unit or an entire circuit diagram. This leads to the following general program structure:

```
/* NST definitions: */
#include "nst_defs.h"
```

```
/* further includes and defines */
    ...
main() {
   /* call NST constructor functions for units */
   /* connecing created units */
   /* executing created circuits */
}
```

Expanding this scheme for the present example, we arrive at the following program:

```
#include "nst_defs.h"  /* NST-function headers */

unitptr data_generator(int iDelay, unitptr Host) {
   unitptr uBox, uWave, uDelay;

   /*** make container for data_generator: */
   uBox = primitive_unit(0,  2,1,2, Host);

   /*** create subunits:  */
   uWave  = wave_gen(20,20,0,1,uBox);
   uDelay = delay_unit(1,iDelay,uBox);

   /*** make connections: */
   joinfields(uWave,uBox,   out_[1], out_[0]);
   joinfields(uWave,uDelay, out_[1], in_[0]);
   join(uDelay,uBox, out_[iDelay-1],out_[1],0,1);
   join(uDelay,uBox, out_[iDelay-2],out_[1],0,0);

   return uBox;
}

main() {

   int i,iNumSteps=1000;
   unitptr uData, uNeuron, uDif, uPlot;

   init_nst();

   /*** create required NST-units: */
   uData   = data_generator(10,ROOT);
   uNeuron = bp_unit( 2,FUN_IDENT,FUN_ONE,0,ROOT);
   fNstInp(uNeuron,2,0) = 1.0;  /* set learning rate (at input 2) */
   uDif    = dif_vec( 1,ROOT);
   uPlot   = plot_series(0,"NST:plot_series-0 100 690 600 300",
             "%2:-1:1m %5:-1:1m %7:-1:1m", 100., 1.,-1., 1.,ROOT);

   /*** build circuit: */
   joinfields(uDif,   uNeuron, out_[0],in_[1]);
   joinfields(uData,  uNeuron, out_[1],in_[0]);
   joinfields(uData,  uDif,    out_[0],in_[0]);
   joinfields(uNeuron,uDif,    out_[0],in_[1]);
   joinfields(uData,  uPlot,   out_[0],in_[0]);
   joinfields(uNeuron,uPlot,   out_[0],in_[1]);
   joinfields(uDif,   uPlot,   out_[0],in_[2]);

   /*** run: */
   for (i=0; i<iNumSteps; i++) {
         exec_unit(ROOT);
         adapt_unit(uNeuron); /* adapt uNeuron only */
   }
   sleep(3);

} /* end of main() */
```

For each “box” in Figure 1 there is one creation call for a corresponding NST-unit in `main()`, and for each wire there is one `joinfields` or (possibly) several `join` calls. The leftmost “box” labeled `data_gen` provides at each execution step the current sine value $x(t)$ (upper connector), and two delayed values $x(t-9)$ and $x(t-8)$ (lower connector). As explained above, it is built from a small “subcircuit” of two simpler NST-units. These are created, connected and encapsulated as a new unit in subroutine `data_generator`. The `bp_unit` is configured as a simple linear perceptron and learns the prediction. For this, it has to receive at its second input connector the difference of the correct and its own output value. This difference is computed by the `dif_vec` unit. Finally, the rightmost unit plots the time course of three curves: the correct value $x(t)$, the prediction output of the `bp_unit`, and the error difference computed by the `dif_vec` unit.

5 Compilation

To compile the above source file (its name being, say `predict.c`), requires to include the NST-include files. These contain some architecture specific parts, therefore, they must be “told” the architecture by defining one of the flags `iris4d, irix5, sun4, rs6000` or `intel` (the correct value is in the variable `$ARCH`). In addition, there are different graphics options, requiring the inclusion of additional libraries. Below is a table of the neccessary combinations for each architecture. To get along with the linewidth, we use the abbreviation (`$(SAME)` for `-o predict predict.c -I/vol/nst/include -L/vol/nst/lib -lnst -lmalloc`, which is the same for all architectures.

architecture	how to compile
SGI iris4d	`cc -Diris4d $(SAME) -s -lgl_s -lfm_s -lm -lbsd -lsun -lX11_s`
SGI irix5 with OpenGL	`cc -Dirix5 -DOGL $(SAME) -s -lGL -lGLU -lm -lX11_s`
SGI irix5 with X11	`cc -Dirix5 -DNST_X11 $(SAME) -L/vol/ni/bin -lX11 -lm`
Sun-OS	`gcc -Dsun4 -DNST_X11 $(SAME) -L/vol/ni/bin -lX11 -lm`
IBM AIX	`cc -Drs6000 $(SAME) -lgl -lm -lbsd`
PC Linux	`gcc -Dlinux -DNST_X11 $(SAME) -L/vol/ni/bin -lX11 -lm`

On those machines that do not offer the `GL`-graphics library, the graphics routines use X11 and are restricted to 2d-graphics operations (with the exception of the `plot3d`-unit which offers some emulation for 3d-line and mesh drawings). As a benefit in this case, everything can be run over the network. When using X11, NSTprovides a subset of the GL-functions through its own X11-interface. This set is neither complete [1] nor is a precise emulation aspired. It is just intended to be sufficient for implementing NST's graphics units. However, by including the header file `glX11.h`, these functions can also be invoked by their `GL`-name in other contexts. If a complete and faithful emulation is required, one can link e.g. the `Ygl`-library. However, in this case one MUST NOT include `glX11.h` to avoid name conflicts[2]

6 The graphical editor NEO

Actually, step 3, the translation of a circuit diagram into an equivalent C-program, is rather straightforward. Therefore, it has been automated: the editor *Neo*allows to construct the circuit diagram graphically and will write the equivalent C-program for you (you even can do without the C-program altogether, since *Neo*can run the circuit diagram directly, without compiling; this is convenient for interactive experimentation).

[1]the available routines are defined in `nst_X11.h`

[2]omitting `glX11.h` puts a prefix `nst_` in front of all NST-GL-emulation functions.

For instance, you may wish to build the circuit diagram in Fig.1 with *neo*and let *neo*write the NST-program for you. Then you can compare the listing with the above source.

Probably you can find out how to use *Neo*just by experimentation[3]

A description of most features of *Neo*can be found in the postscript file `/vol/ni/man/ps/neomanual.ps`, a brief synopsis is in `neo_primer.ps` that includes more recent extensions is in the same directory.

7 Accessing inputs and outputs of NST-units:

Using NST-units within a C-program requires to know how to access the data elements within the pins of each unit. For this, NSTprovides a set of macros. Each macro name consists of a prefix to indicate the type of the pin that is to be accessed, followed by either `NstInp(u,f,p)` or `NstOut(u,f,p)` to access the $p-$th pin of the f-th input or output field of unit `u`, resp.

Here is a table of the output type macros (the input ones are analogous, with `Y` replaced by `X`).

NST pin type	macro name	stands for:
float vector	pfNstOut(u,f,p)	ptr to first el of float vector
int vector	piNstOut(u,f,p)	ptr to first el of (short) int vector
byte vector	pcNstOut(u,f,p)	ptr to first el of (unsigned) char vector
string	ptNstOut(u,f,p)	ptr to start of var len string
any	pvNstOut(u,f,p)	void ptr to first data element, any type
scalar float	fNstOut(u,f,p)	value of scalar float pin
pin	pNstOut(u,f,p)	ptr to full NST-pin struct
scalar float	Y(u,f,p)	same as fNstOut(u,f,p)
vector float	YP(u,f,i)	same as pfNstOut(u,f,0)[i]

(the last two ones are there for historical reasons). Note that in order to access a single element in a vector pin1 the ptr given by the macros still must be dereferenced, e.g., `piNstInp(u,1,2)[3]` is the element at position 3 (the fourth element) of the `int` vector pin at position 2 (the third pin) in field 1 (the second field) of unit `u`.

The types of two pins determine whether they can be connected with a `join`-call: this requires that the two pins are either both of type “string” or “scalar float”, or, if their types differ, that both pins are of the vector type and have the same dimension (in this case, there will be an automatic conversion to the destination type)[4].

Note also that at present a vector pin of type “float” with a single element and a “scalar float” pin are considered as different and are incompatible (this may change in a future release).

8 Available units:

To translate a task into a circuit diagram requires to know which units are available and what they do. Usually this information can be obtained with `man NameOfUnit`, or, more conveniently, as HTML-hypertext (the entry point is the file `/vol/nst/man/html3/manindex.html`). The appendix provides a synopsis (for an updated version enter `man nstfuns`.

There are rather direct counterparts of many C-constructs, some of them are listed below:

[3] *Neo*runs on the SGI-workstations with full 3D-Graphics support, and on the POWER CHALLENGE, the SUN workstations and the LINUX-PCs with 2D graphics; there is also compatible 2D-version that will run on a LINUX-laptop with a LCD-monochrome display and 3D-version on the the RS6000-workstations.

[4] the `splitter` and `merger` units can be used to convert a field of `dim` scalar float pins into a float vector of corresponding size (or the reverse), which can then be connected to any of the remaining other vector types

	C	NST
data types	structs	composite fields
	arrays	vector pins of fields
control ops	for-loop	iterator
	switch	switch_op
	condition	if_op
	return	return_unit
io-functions	printf	fprintf_unit
	scanf	scanf_unit
math functions	`+,-, ...` `<math.h>`	sum_vec, mul_vec,
	... sin_vec, cos_vec	

Recent additions also include units (`str_dispatch_unit` and `svc_unit`) to make services that were written for the communication tool SCOTT available under NSTand within *neo*, as well as units (`export_data, import_data`) that support a simple form of interprocess communication via shared memory (the latter still very alpha).

9 How does NST work internally?

NSThas to do three tasks: manage creation and removal of instances of units, connect units into circuits, and finally, care about doing something real by executing, adapting, loading, saving etc. circuits or parts thereof.

1. For the creation of units, NSTprovides a large collection of *constructor routines* (cf. Appendix). Each constructor routine can make instances of a particular type of unit. Most constructor routines expect some parameters that further define the properties of the created unit. Return value is always a handle of type .unitptr that serves for all further references to the created unit. This handle points to a data structure unittype that holds all necessary information about the unit (how many inputs and outputs it has, pointers to their values and other private variables etc.) This includes six function pointers to C-subroutines that implement the six methods execution, adaptation, loading, saving, control and removal that each type of unit offers (these are invoked by `exec_unit(handle)` .etc) The corresponding 6 C-subroutines together represent the functionality of a NST-unit. When your write your own NST-units, you must provide these 6 subroutines, together with a constructor function that makes and properly initializes a unittype struct, directs the six function pointers to your routines and returns the address of the struct as the handle.

2. Part of the data pertaining to a unit is a pointer for each pin (input and output) that tells the address of the value associated with the pin. When NSTconnects an output pin a of a unit A to an input pin b of a unit B, it just copies the address referenced by the pointer of the source pin .a into the pointer of the destination pin b. Additionally, each pin keeps a list of its current readers, thus .a adds the new reader .b to its list, and .b looks into its list to see if it has to propagate the received address further downstream to its readers, since these must use the same value that pin .b uses (and recursively to the readers readers etc.). [5] The necessary pointer gymnastics and bookkeeping is provided by the two functions `join` (for two pins) and `joinfields` (for all pins of two fields)

By this pointer mechanism, each unit has a set of pointers that allow direct access to its input data and the destinations of the result data, obviating the necessity of any time consuming transport operation.

3. Besides just creating all units, NSTalso puts in some additional structure: each unit is viewed as a container, that can contain other units as subunits. This gives rise to a hierarchical nesting of units, much like a directory tree. Each constructor routine requires the specification of the host unit within which the new unit shall be constructed as a subunit. Initially, the only available container ist

[5] In addition, .b frees the memory for its now no longer needed previous value, unless this pertained already to an output pin of another unit (if .b somewhen gets totally disconnected again, it allocates a private variable again).

the ROOT unit that is made by the init_nst call. To execute (adapt,...) an entire circuit diagram is accomplished by executing (adapting,...) the single unit that contains it.

10 Extending NST

Sometimes, you may wish units that are not yet built into NST. If the task is small and not time critical, the fastest way is to use the prog_unit: its creation routine accepts the text of a C-program (the syntax is restricted to a reasonable subset of C) and returns a unit that performs the operations of the specified program (this uses a C-interpreter that is part of NSTand yields about 10% of the speed of compiled code).
Example:

```
prog_unit(4, 4,
" for (i=0; i<NX; i++) printf(\"pin[%3.f] = %8.3f\\n\",Y(i)=X(i));",
uHost)
```

creates a unit with an input field and and output field of 4 pins each that just copies its data from input to output, but prints them also.

In many cases, the time-critical loops are due to matrix- or vector operations. The matrix_algebra-unit is suitable for these cases. It resembles the prog_unit, but allows to specify matrix operations in a Mathcad-like language and performs most matrix computations almost as fast as compiled code (since it essentially calls suitable C-subroutines). Supported operations include matrix multiplication, inverse, eigenvalues and -vectors, singular value decomposition, sorting and many more.

If the task is still more complex and cannot conveniently be broken up, you must write the required units directly in C, but not from scratch: the tool mku writes a skeleton program for you into which you just have to insert the missing special pieces of C-code (leaving still enough work over). If you call mku with the option -v, you will find additional comments in the created template to explain how to complete it to a full definition of a unit. mku also leaves behind a makefile for compiling the new unit (the makefile notices the proper architecture automatically). You can also make your new unit available within *Neo*. The file neo_private.c provides a template from which you probably can guess how to proceed (for more details you may wish to consult the *Neo*Manual).

A third way to proceed is to write the new function in the form of a SCOTT-service. Such services can then be invoked, using the str_dispatch_unit and/or the svc_unit in NSTor *neo*. This approach has the advantage that such services can be dynamically exchanged during runtime, without any need of recompiling the program that uses them.

The last two approaches can be combined: a function, first written as a NST-unit, can then be “wrapped” into a SCOTT-service with a minimum of additional effort. The necessary support for this is provided with the tool senwa. When such a service is invoked from NSTvia the svc_unit, the service appears as an instance of the original NST-unit, however, with the additional benefit of network-transparent invocation and dynamic replacement. This provides a maximum of modularity and flexibility and is thus the recommended approach.

11 Hardware devices:

NSTalso tries to make the use of special hardware things easier: there are a number of units that provide access to hardware devices, such as the datacube, the androx boards. the force sensor, the puma robot, the camera head and the manus hand. To use these, you must have access to a proper version of libnst.a (and of *neo*) where these things have been compiled in. To make a version of `libnst.a` and/or *neo*for a particular configuration (choice of architecture and hardware devices for which support is compiled in), edit the `Makefile.shared` in your `src` directory and request compiling in the proper hardware by setting the associated flags `MANUS, DATACUBE, RCCL, PUMA, ANDROX`. In addition to these "native" NST-units, there are now many `SCOTT`-services for the various hardware devices available. These have the advantage that they need not be compiled into your current version of `libnst`. Instead, they can be used either standalone or from within NSTor *neo*, and in both cases access can be locally or remote.

12 Checking out NST

NSTis administrated with the concurrent version system `cvs`. To obtain a fresh set of sources, enter `cvs co nst`. This will make the entire NST-source tree in your current directory. Then `cd` to subdirectory `src` and enter `make info`. Then follow the steps on the display. However, you can always use the public version of NST(as described in Sec.4). Then you need not bother about making your own `libnst.a` at all.

13 Further information

Postscript versions of some useful documents are in `/vol/ni/man/ps`: `NstPrimer.ps` (=this document), `neo_primer`, `neomanual.ps`, `nstmanpages.ps` and `nstmanpages_index.ps` contain what their name indicates. The HTML-link `http://www.techfak.uni-bielefeld.de/techfak/persons/nils/ni/nst.html` is a good entry point to HTML-versions of a major portion of the NST-documentation, including some demos that can be interactively started. In `/vol/ni/share/neo/tutorial` there is an extensive set of NST-examples plus explanation text for getting acquainted with most NST-features. These examples are encoded not directly as C-source files; instead, each file is a *description* of a circuit in a format readable by *Neo*. *Neo*can then be used to read the description, build and display the corresponding NST-circuit (similar as in Fig.1) and experiment interactively with it. Only if C-source code is expressly desired, *Neo*'s `MakeSrc` option allows to generate it. Still further examples (somewhat less well maintained) are in `/vol/ni/share/neo/demos`.

The 8 pages are used up now, and there remains only an epsilon of space for the Appendix.

14 Appendix

15 General NST units:

array_of make array of units from operand template
ctrl_op apply sequence of control ops to successor units
counter count exec calls
delay_unit delay buffer for data sequence
export_data export data into shared memory segment
extract_subvec extract subset of vector elements for new vector
fileio_op load/save data of operand units
for_loop simple iterator unit
function_box create unit that applies a set of functions
if_op conditionally execute successor unit(s)
import_data import data from shared memory segment
iterator iterate successor unit(s) and collect data
matrix_algebra very versatile unit that evaluates matrix formulas
merger create unit to merge several input fields
merge_multi generalized merger
no_op omit execution of subunit(s)
one_op execute operand(s) only firsttime
primitive_unit create a primitive unit and return pointer
prog_unit unit that interpretes C program
matrix_algebra unit that interpretes matrix program
rpc_unit make remote nst unit
primitive_unit create a primitive unit
primitive_unit1,2,3 dito, but different versions of argument list
reflect write values ("reflect") into output field
remake_unit remake successor unit with new parameters
return_unit return from execution/adaptation of subunits
sleep_unit wait for a specified time

splitter create unit to split a single input field
split_multi generalized splitter
som_op create unit to make self-organizing map
svc_unit access SCOTT service
switch_input, switch_output switch connection between units
switch_op switch execution (analogous to C-statement)
thread_op threaded execution of operands
time_unit get elapsed CPU time
var_def define named array of variables
var_get get element from named array of variables
var_set set element in named array of variables
virtual_unit make other unit callable in this place
void_unit create a void unit
void_unit1,2,3 dito, but different versions of argument list

16 General NST functions:

dim_of_X dim_of_Y macros to retrieve dimension of a field
exec_unit execute a unit
adapt_unit adapt a unit
ctrl_unit control operation for unit
remove_unit remove a unit
join join pins of two units
joinfields join two fields of two units
joinvar attach C-variable to pin of a unit
queue_unit requeue unit and return previous position in queue.
swap_units swap two units
str_dispatch dispatch string message to SCOTT service
X,Y-macros access variable of pin

17 General math units with vector result:

base_vec i-th canonical basis vector
bin_vec binarize vector
cos_vec cosine of vector elements
const_vec provide constant vector
log_vec logarithm of vector elements
exp_vec exponential function of vector elements
fermi_vec fermi function of vector elements
sin_vec sine of vector elements
tan_vec tangens of vector elements
tanh_vec tanh of vector elements
lim_vec limit vector elements to range
min_vec create unit that forms minimum of two vectors
normalize_vec create unit that normalizes a vector
out_prd, out_div create unit to compute outer product/quotient
pow_vec power of vector elements
rnd_vec create random vector generator
scale_vec scale vector elements to given range
scl_vec scales a vector by a given factor
sub_avg subtract average from vector
sum_vec create unit that adds two vectors
sup_vec create unit for vectorial superposition
thr_vec create unit that threshholds a vector
trf_vec create unit that transforms a vector (vector result)
wav_vec create unit to compute weighted average of vector

18 General math units with scalar result:

dif_len create unit that computes vector distance
euc_len create unit that computes euclidean length of vector
min_val create unit that computes minimum element of vector
max_val create unit that computes maximum element of vector
scl_prd create unit to compute scalar product
sqr_val create unit that computes square of norm of vector
trf_val create unit that computes transform of vector (scalar result)
cos_ang cosine of angle between two vectors

19 Further math units:

fft_1d 1d fast Fourier transform
matrix_algebra very versatile unit that evaluates matrix formulas.
conjugate_gradient minimize function by conjugate gradient method
corr_mat correlation matrix from sequence of vectors
diag_mat eigenvalues and eigenvectors of symmetric matrix
diffeqn_op integrate differential equation
function_gen generate sequence of function values
ident_mat provide identity matrix
inv_mat invert matrix
jacobian_op compute Jacobian for vector function operand
mul_mat multiply two matrices
mult4x1,mult4x1_n create unit that multiplies a 4x4 matrix with a vector
mult4x4 create unit that multiplies two 4x4 matrices
genalg Generate unit which performs genetic algorithm
polynom_unit evaluate a polynomial
rungekutta4 integrate ode using adaptive step size Runge Kutta procedure
rnd_gen create unit that generates a random vector
serial_comp perform serial computation on sequence elements
soft_min compute soft min of vector
soft_max compute soft max of vector
statistics create unit to compute statistics
voronoi_op compute 2D Voronoi diagram
wave_gen create wave form generator

20 Operations on image rectangles:

centroid perform threshold operations on image
combine combine two image regions
convolution compute convolution of a mask with an image
diff_segment create unit to segment image
data_store store sequence of data vectors in memory
dual_base computes dual base of a set of masks
gabor_conv compute convolution of gabor masks with an image
gabor_to_pic map gabor masks to image region.
gauss gauss filter an image
histogram compute histogram from image
laplace laplace filter an image
locate_peaks locate intensity peaks in an image
mapper iterate operation of unit across image rectangle
median median filter for images (pixel values from 0 .. 255)
merge_image merge of subimages (for merge of parallel threads)
mexican apply "mexican hat"-filter to image
nst_image_store store and retrieve image sequences
nvk_class apply a nvk classifier to images
pic_to_gabor compute convolution of gabor masks with image
remap remap an image to a different resolution
spatial_operator iterate operation of unit across image rectangle
sobel_x sobel_y sobel filter an image
split_image split into subimages (for split into parallel threads)
subrect cut out subrectangle of an image
subrect_of_change find rectangle that bounds changes in image
threshold perform threshold operations on image
read_pix read pixel data from window
warp_op translate, rotate or generally "warp" an image

21 Accessing special hardware devices:

androx_grab unit to grab image from androx board
cam_head control binocular camera head
dq_grab unit to grab data cube image
sun_grab unit to grab image from SUN Video Adapter
grab_axr another unit that grabs frames from an androx board
grab_vca grabs frames from VCA Video Capture Adapter
manus_move create unit to .
write_to_vca writes an image to the VCA Video Capture Adapter image plane
puma1 moving the PUMA560-Robot ("pinocchio") by using encapsulated

22 Displaying data:

color_code translate scalar into color values
clear_window clear a window
draw_ellipse draw an ellipse or an elliptic arc
draw_mesh draw a mesh into a window
draw_pix draw a pixel array into a window
draw_str draw a string into a window
draw_sym draw a symbol into a window
frame3d create unit to render a 3d-coordinate frame
iso_contour create unit to draw isocontours
view3d create unit to provide an interactive 3d viewing window
plot3d subset of view3d-capabilities in absence of GL or OpenGl
plot_xy create unit to generate a xy-plot of data
plot_series create unit to plot a time series
plot_vec create unit to plot a vector
write_pix create unit that displays pixel data in window
mops monitor data packet stream

23 Reading or writing of data files:

load_unit load state of a unit
multi_ppm generate unit for writing multiple image files
nst_wd dump nst-window content to file
open_postscript open file to pixel image in postscript format
printf_unit: formatted print to file, string or stdout
read_feature_file unit to read feature file
read_file create unit for data input from file - read_data
read_pix create unit that reads image pixel values
read_pxmfile read file in pbm, pgm or ppm format
save_unit save state of a unit
window_to_eps print 2D-graphics window contents as an eps-file
write_postscript append pixel image data to postscript file - write_data
write_file create unit for writing data to file
write_pgmfile write pixel data to a pgm-file
write_pix create unit that writes pixel data to window
write_ppmfile write RGB-pixel data to a ppm-file
rim read image sequence from file
scanf_unit formatted scan from file, string or stdin
wim write image sequence to file

24 Units for programming user interfaces:

input_window create unit to manage a window for data input
mouse_xy get mouse location within window
output_window create unit to display a window for data output

25 String Manipulation

const_str provide constant string
print_str print string(s) in window
printf_unit formatted (s)print; including string(s)
scanf_unit formatted (s)scanf; including string(s)
str_dist distance between two strings
NstStrCat concatenate dynamic strings (C-function)
NstStrCat1 concatenate dynamic strings, with offset (C-function)
NstStrCpy copy dynamic strings (C-function)
NstStrCpy1 copy dynamic strings, with offset (C-function)
NstStrDup dynamic string duplicate (C-function)
NstInpStrCpy copy dynamic string into NST input pin (C-function)
NstOutStrCpy copy dynamic string into output pin (C-function)
NstSprintf sprintf() for dynamic destination string (C-function)
NstOutPrintf printf() into NST output pin (C-function)
..more... (see `nst_strg.h`)

26 Neural network units:

bp_layer create layer of bp-units.
bp_net create layered bp-net ("MLP").
bp_unit create bp-unit
delta_layer create layer of delta-units.
delta_neuron create delta-neuron unit
delta_unit create delta-unit
hebb_neuron create hebb-neuron unit
hebb_unit create hebb-neuron unit
image_psom image unit of an existing psom
inspect_weights inspect weights of weight pool unit(s)
llm_layer create layer of LLM-units
mlp_net improved version of `bp_net`
neuron_pool general neural network layer
psom (old) parametric self-organizing map
psom2 improved parametric self-organizing map
q_learning Q-Learning in a discrete state-action space
rbf_layer create layer of RBF-units.
rbf_unit create delta-unit
som2d create two-dimensional self-organizing map.
som_builder build general self-organizing map.
som_layer create 2d-self organizing map
vq_unit create vector-quantization unit
weight_pool general weight layer between neuron pools

27 Units for 3D object rendering:

begin_frame start a new object
end_frame discard current object
define_world define world coord frame
frame3d create unit to render a 3d-coordinate frame
get_frame return current object
make_arrow render an arrow
iso_surface render a 3D isosurface
make_cone make a cone shaped object
make_graph create unit to render a graph (2 versions)
make_link render a link-object (3 different types)
make_mesh create unit to render various types of mesh
make_object make object from file description
make_pyramid render a four sided prism with
obj_to_world transform from object to world coord frame
replace_frame replace the current frame by a new one.
rot_frame rotate_frame create unit to rotate a coordinate frame
scale_frame scale coordinate frame
translate_frame create unit to translate
use_material select material for rendering
view3d create unit to provide an interactive 3d viewing window
window_to_world transform from window to world coord frame
world_to_obj transform from world to object coord frame
world_to_window transform from world to window coord frame

28 Robot related units:

dh_tf create unit to compute Denavit-Hardenberg transform
fit_frame fit orientation/position of 3D coord frame to meet constraints
manus_move move fingers of manus hand
mult4x1,mult4x1_n create unit that multiplies a
mult4x4 create unit that multiplies two
puma1 moving the PUMA560-Robot ("pinocchio") by using encapsulated
pumakin forward kinematics of puma arm
pumainvkin inverse kinematics of puma arm
rob_dyn compute forward dynamics of robot arm
rob_invdyn compute inverse dynamics of robot arm
rob_dyn compute forward dynamics of robot arm
rot_trans_mat4x4 create unit that makes a matrix for homogeneous coordinate transformations

29 Special operations: [not all in libnst.a]

boxPntGen generates mesh pnts or uniform random dot points
constr_lay constrained neural layer dynamics
diff_segment create unit to segment image
dual_base computes dual base of a set of masks
getinterval create unit that monitors a data interval or data volume
iCopyComments read and copy comment lines from file
iXdimGetCurrPsom iMdimGetCurrPsom aiKnotsGetCurrPsom fprintPsomDesc get info of current psom
image_psom image unit of an existing psom
loadPsomBlock check description block and load psom unit
savePsomBlock save psom unit description block
ludcmp LU decomposition of matrix
segment train segmentation networks and apply to images
set_nst_inputfilter control acceptance of values in input_window
set_restriction restrict processing to subrectangle
uPsomFromBlock load description block and create psom unit
vec_store store and manipulate vectors

30 NST administration and utility functions:

error_msg (old) issue an error message or warning
fvget read local variables from file
fvinit initialization for fvget, fvput
fvput write local variables to file
init_unit initialization of a unit
init_nst initialize NST-system
invert_matrix Inverse of a matrix
invmtr Inverse of a matrix
mops monitor data packet stream
nst_error issue an error message
nst_message issue a general message
nst_option parse NST option string (former parsefields())
nst_warning issue a warning message
replword script to replace name in file-stream
rim read image sequence from file
wim write image sequence to file
skip_comments skip comment lines
vget retrieve local variable
vinit initialize access to local variable set
vput create and initialize local variable

31 General utilities:

finito file based neural network investigation tool
mksh make skeleton for shell script
mkopt command line option parser generator
mkt tool to make template generator tool
mkc make skeleton C-program
mku make skeleton program for NST unit
mma man-page maker (combines mmp and mmh + improvements)
mmh make manual page in html-format
mmp make manual page in tex and nroff format
imk generate skeleton program for image processing filter
neo the NST graphical editor
option utility Manage command line options, auto document
try try method on file
vDeclareParameter Add new parameter declaration to list low level
vDoc Append output to documentation file
vDeclareFlag, vDeclareInt,vDeclareFloat,vDeclareString Declare command line parameter
vDocArgs,vDocOptions Documentation interface.
vSetHelp Set help text for parameter
vScanArgs Scan command line for arguments
vSetInt, vSetFloat,vSetString set values of parameters