NST Programming Guide

Helge Ritter, May 96[1]

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

Abstract

This manual provides the main documentation for implementing NST-units at the C-Level. The contents of this document are valid for NST 4.6 or higher. Earlier NST versions may need (minor) changes. For the location of the demo sources see the appendix. [2]

Contents

[1] Compiled: March 28, 2002, Id: nstdoc.tex,v 1.2 2001/02/21 14:13:19 walter Exp

[2] the latex- and ps-files `nstdoc.tex` and `nstdoc.ps` for this document are currently located in `/vol/nst/man/...`

1 Overview

This manual provides the necessary information to extend NST by user-specific units. For basic background, see the NST-primer.

NST offers an extensible set of objects, called *units*, that implement a wide range of functions. New functions can be derived either by combining available units into *data flow diagrams* (“circuits”) that can be encapsulated to form a new unit, or by programming new units directly in C.

This manual mainly describes *how to program new units at the C-level*. This is the method of choice when a new algorithm shall be made available as an NST unit and when this algorithm is such that it cannot be built as a dataflow diagram of already available units, or when such realization would be awkward or significantly less efficient than its direct implemenation in C.

Otherwise, if a new function can be implemented in the form of a dataflow diagram of available units, the graphical editor *Neo*is the method of choice to construct this dataflow diagram graphically and to test and debug it interactively. The C-code that implements the final data flow diagram in the form of a new NST-unit can then be generated automatically by *Neo*.

Sometimes, the most concise representation of a new function is in terms of a dataflow diagram plus some additional operations that would be most conveniently formulated at the C-level. Then the above two approaches can be combined.

The rest of the manual provides the necessary information how to program new units at the C-level. To this end, sections 2 and 3 describes how NST's objects are implemented in C. Section 4 then describes the input/output interface of NST units, the available data types and how values can be set or retrieved. Section 5 describes how to use private variables to preserve state information across calls. Section 6 gives further details about the implementation of the basic methods that each NST-unit comes with. Section 13 provides further examples to illustrate the material of the preceding sections. Section 18 summarizes additional technical information (should be skipped at first reading).

Reading of the first six sections, perhaps with a glance at sections 11 and 14, should be sufficient to implement most algorithms in the form of NST-units.

The remaining sections provide additional information that is required to make fuller use of the versatility of NST. Section 8 describes the important concept of operator units and how to implement them. Section 9 describes how to implement units that perform 2D-graphics operations. Section 11 gives information how to implement warning, error and traceing messages. Section 14 discusses some points concerning the design of a NST-unit. Section 15 considers efficiency issues.

While a recipe-type format has been preferred, some sections also provide background information about the inner workings of NST that can be skipped at first reading. These sections appear in small print.

2 Primitive Units

The basic object in NST is called a *unit*. Unlike a procedure, a unit is a persistent object that has associated with it some persistent data and a number of methods. While C++ offers the class construct for a direct implementation of such objects, NST's C-implementation uses the familiar `struct` concept together with function pointers to implement its objects.

A NST-unit is implemented as an instance of a structure of the C-type `unittype`. This structure is defined in `nst_types.h` and has the following main internals:

```
typedef struct unittype{
   /* pointers to input and outputs: */
   field **in;  /* pointer to array of field pointers for input interface */
   field **out; /* pointer to array of field pointers for output interface */
   ...
   /* function pointers to class-specific methods: */
   int (*exec)(struct unittype*);  /* main method */
   ... further methods for adaptation, loading, saving, removal ...
   /* pointers to class specific private variables: */
   void *var;
   ...
   /* further auxiliary pointers */
} unittype, *unitptr;
```

Listing 1: Outline of the `unitptr` struct

The `unitptr` structure provides access to all properties of a unit. To make an instance of a unit of a certain class requires to create an instance of the above structure and to initialize its interface, data and function pointers suitably to reflect the particular properties of the chosen class of unit.

These steps must be carried out be the constructor function for the chosen class of unit. Thus, the main step in the implementation of a new unit is the implementation of its constructor function.

The implementation of the constructor function for a new class of units is simplified by the two predefined NST-constructor functions `void_unit()` and `primitive_unit()` [3] that make a generic "default unit" that then must only be modified (by reassigning some or all of its function and data pointers) to implement the desired class behavior.

! The main difference between `primitive_unit`s and `void_unit`s is the the former is mainly intended as container object for a C-based implementation of a unit, while the latter is intended to make data-flow diagrams of other NST-units available as a new unit by encapsulating them. This is reflected in slightly different methods for both kinds of units: the void unit passes all method invocations down to its enclosed subunits, while the primitive unit leaves it entirely to the implementor what happens to any subunits it may

[3] There are variants `void_unit1..3()` and `primitive_unit1..3()` of both functions that offer different parameter lists to specify the interface of the created unit, see the man pages

contain. Since the implementation of a primitive unit often does not use any NST-subunits at all, this is often not an issue.

For reasons to be discussed shortly (section 2) each NST-unit requires for its creation the specification of an already existing *host unit*. By convention, (a `unitptr` to) this host unit must be specified as the last argument parameter of each NST-constructor function, and newly written constructor functions should also adhere to this rule. An exception is the special unit `ROOT`, which is automatically present after NST has been initialized with the `init_nst()` call and is available as a host unit to start with until additional units have been created.

To illustrate the basic scheme, we implement a unit with no inputs and no outputs that when executed just prints a message to the standard output:

```
#include <stdio.h>
#include "nst.h"   /* necessary NST headers */

int what_myunit_shall_do(unitptr u) {  /* myunit's method */
   printf("Hello world!\n");
   return OK;   /* default return value for NST-units */
}

unitptr myunit(unitptr uHost) {  /* the constructor function */
   unitptr uNew = primitive_unit(0,0,uHost);   /* make generic unit */
   if (!uNew) return NULL;                 /* return NULL if failure */
   uNew->exec = what_myunit_shall_do; /* make uNew use above method */
   label_unit(uNew,"myunit");  /* assign a print name to myunit     */
   return uNew;                /* return handle to created instance */
}

main() {  unitptr u;
   init_nst();          /* must precede any other NST call */
   u = myunit(ROOT);    /* make instance of u, using host unit ROOT */
   exec_unit(u);        /* invoke u's main method once */
}
```

Listing 2: *Hello world*-example

Here, `what_myunit_shall_do()` implements the actual operations to be carried out by any unit created by the constructor function `myunit` if "executed" via the `exec_unit()` function. This "exec"-method is always the main method of a NST-unit and must always be implemented as a function of the fixed type `int(*)(unitptr)` with return value `OK`. The `exec` method of a useful unit, however, would access input values, do some computation and write results back to any output values. Before we can write such functions, we must discuss the input/output interface of a NST unit (section 4) and how units that have inputs and outputs can be made. Further, we must consider how such units can preserve internal state information between `exec_unit()`-calls (section 5) and how additional methods (section 6) can be defined.

Note, however, that even with the above example we have gone a bit beyond the absolute minimal scheme by inserting a check against a failure of the `primitive_unit()`-routine (which may occur, e.g., if an argument parameter such as `uHost` has an illegal value), and by giving the created unit a print name. Both operations are not strictly required, however, they should be considered as almost mandatory good practice.

3 Non-primitive Units

The above illustrates the case of a "primitive unit". By this term, we mean a NST-unit whose methods are implemented at the C-level. NST also offers a way to make non-primitive units that derive their methods from the methods of one or several simpler NST-units (which themselves may be primitive or not).

The implementation of a non-primitive unit requires *only* the writing of its constructor function which constructs the unit from a number of other units; its methods are then automatically derived from the methods of these units without the need for an explicit coding of any `exec` or other methods.

To achieve this, NST allows a unit to act as a "container" that encloses a (sequentially ordered) set of "subunits". In fact, each NST-unit (except for the special unit `ROOT`) must be the subunit of some other, already existing unit. The already existing unit is the "host unit" that must be specified (as the last argument of any NST constructor function, cf. 2) whenever a new NST unit is created. The newly created unit is the appended to the (possibly empty) list of already existing subunits of the specified host unit.

How is this related to the implementation of non-primitive NST-units that use existing units to derive their function? NST offers a special class of container units (with the constructor function `void_unit()` that returns an initially void container) which have as their `exec` method *the sequential execution of the* `exec`*-methods of their subunits*. This composition law extends also to all other methods that are defined for NST units (there is, however, a subtlety for the `adapt`-method which we shall discuss later) and offers a simple scheme for building units with methods that are sequentially composed of the methods of other, simpler units: the constructor function of such a unit first creates a suitable void container unit; next it creates the appropriate sequence of subunits with the container unit as their common host unit; finally it returns the `unitptr` to the container unit.

For instance, the implementation of a non-primitive unit that makes use of several instances of the primitive unit `myunit` implemented above in order to print `hello world` several times on each invocation, would look like

```
   unitptr more_hello(int iHowOften, unitptr uHost) {
      unitptr uNew = void_unit(0,0,uHost);
      if (!uNew) return NULL;  /* failure */
      /* now make iHowOften subunits of uNew: */
      while (iHowOften-- >0) myunit(uNew);
      label_unit(uNew,"more_hello");
      return uNew;
}
```

Listing 3: More *Hello world*

Note that there is no need to keep the pointers returned by the `myunit` calls for later reference. All that is needed is the pointer `uNew` to the created container unit, which is returned with the last statement.

Information about the host and the subunits of a unit is kept in five `unitptr` fields in the `unittype` struct:

```
typedef struct unittype{
   ...
   struct unittype
     *host,          /* pointer to this unit's host unit */
     *first_subunit, /* ptr to first subunit, NULL if none */
     *last_subunit,  /* ptr to last subunit, NULL if none */
     *prev, *next,   /* ptr to predecessor/successor of this unit
                        in the list of subunits of host, NULL if none */
   ...
} unittype;
```

Listing 4: Further fields in the `unitptr` struct

Thus, to traverse the list of subunits of a unit `u` in, say, reverse order, one can use a program fragment such as

```
for (uSub=u->last_subunit; uSub; uSub=uSub->prev) {
    ... do something with uSub ...
}
```

Listing 5: Iterating over the subunits of a unit `u`

The above listing is given to show what really happens; however, it is better programming style to access the fields of the `unitptr` struct not directly but only via the predefined macros `nst_host_of(u)`, `nst_first_subunit_of(u)`, `nst_last_subunit_of(u)` and `nst_next(u)`, `nst_prev(u)`. Also, NST provides the functions `nst_exec_subunits_of(u)`, `nst_adapt_subunits_of(u)`, `nst_ctrl_subunits_of(u)` as a convenience to iterate one of the basic methods over all subunits of a unit `u`.

Most implementations of a non-primitive unit from a collection of subunits would require some communication among the subunits and with the "outer world". This requires the subunits and their container ("host") unit to have inputs and outputs, a topic that we are going to discuss in the next section. The necessary communication would then be set up by connecting pairs of inputs and outputs appropriately to form a "circuit" or data flow diagram that performs the desired task. We do not describe this any further here, since such circuit diagrams can be graphically constructed and interactively tested with the graphical editor *Neo*. Once a circuit diagram works as desired, the C-code for its constructor function can then be generated automatically; this renders the coding of non-primitive units by hand rather obsolete.

! There is an exception to this: sometimes it may be convenient to follow a mixed implementation strategy and to achieve part of the operations of a unit by a circuit diagram or by just executing some subunits, and part by explicit C-statements. In this case, one should first write a constructor function that creates the necessary circuit diagram within a `primitive_unit` instead of in a `void_unit` (you can use *Neo*for this, if you are willing to replace the `void_unit()` in *Neo*'s output by a `primitive_unit` by hand). Unlike the `void_unit`, the `primitive_unit` does not by default iterate its `exec`, `adapt` or `ctrl`-method over its subunits (however, it does so for its `load, save` and `remove` method). This gives you full control to implement in the `primitive_unit`'s `exec` (and, likewise, `ctrl` and `adapt`) method(s) whatever sequence of operations that is required. Here, we illustrate such mixed strategy for the above `more_hello` unit, prepending a header and appending a trailer line to the output of its subunits (this is only to illustrate the technique; for this simple example this approach would hardly be sensible)

```
int exec_for_more_hello(unitptr u) { int iReturn;
   printf("this line is prepended!\n");
   iReturn = nst_exec_subunits_of(u); /* if ok, yields iReturn=OK */
   printf("this line is appended!\n");
   return iReturn;
}

   unitptr more_hello(int iHowOften, unitptr uHost) {
      unitptr uNew = primitive_unit(0,0,uHost);
      if (!uNew) return NULL;  /* failure */
      /* now make iHowOften subunits of uNew: */
      while (iHowOften-- >0) myunit(uNew);
      /* set exec method: */
      uNew->exec = exec_for_more_hello;
      label_unit(uNew,"more_hello");
      return uNew;
}
```

Listing 6: Illustration of mixed implementation strategy

Note that for the other methods `load`, `save` and `remove` the `void_unit` and the `primitive_unit` share the same behavior, since this is what is desired in the majority of cases. If full control even over these methods is desired, one must store a private copy of the host unit's `first_subunit` and `last_subunit` pointers, after which these may be set to `NULL`, thereby disabling any automatic iteration of *any* methods over its subunits.

! When a unit is executed (adapted/ctrl'ed), it executes its subunits as long as these return an non-negative return value. The default return value is `OK` (which is zero). If a subunit returns the special reserved value `NST_RETURN` (which is negative), this is taken as a signal not to execute (adapt/ctrl) the remaining subunits and instead to continue execution (adaptation/ctrl) with the next unit at the level of the subunit's host (currently, any other negative return value is taken as an error indicator). Therefore, the special return value `NST_RETURN` allows to escape from the execution (adaptation/ctrl) of all subunits at some intermediate point in their processing sequence. NOTE: this does not apply for the `load`, `save` and `remove` methods. Their invocations are always extended to all subunits.

4 Inputs and outputs

Units such as `more_hello` are hardly useful, since, after having been created, they provide no way to exchange data with their environment.

Thus, NST units can be equipped with an interface that offers zero or more *input fields* for data input, and zero or more *output fields* for data output.

To create instances of units with a desired structure of their interface (the number of inputs and outputs and their types) makes use of the "generic" NST constructor functions `void_unit()` and `primitive_unit()` from which all other NST units are derived. The desired interface must be specified at the creation time of a unit via the the parameter list of these functions; its structure remains fixed throughout the remaining lifetime of the created unit.

! Variants `void_unit1..3()` and `primitive_unit1..3()` of the standard constructor functions exist in order to cover different needs for the specification of an interface (for an overview, see `man void_unit`).

Before embarking on a more systematic overview, we show the code for a variant of the "Hello world"-unit that now has a single input that allows to specify and alter the to-be-printed text after the unit has been created:

```
#include <stdio.h>
#include "nst.h"        /* necessary NST headers */
#include "nst_strg.h"   /* necessary NST headers */

int what_myunit_shall_do(unitptr u) {  /* myunit's method */
   char *pcThisIsMyInputString = ptNstInp(u,0,0);
   printf("%s\n", pcThisIsMyInputString);
   return OK;   /* default return value for NST-units */
}

unitptr myunit(unitptr uHost) {  /* the constructor function */
   unitptr uNew = primitive_unit(NST_INP,NST_STRING,10,NST_HOST,uHost);
   /* define & make a unit with a string input of initially 10 chars */
   if (!uNew) return NULL;            /* return NULL if failure */
   uNew->exec = what_myunit_shall_do; /* make uNew use above method */
   label_unit(uNew,"myunit");  /* assign a print name to myunit     */
   return uNew;                /* return handle to created instance */
}

main() {  unitptr u;
   init_nst();         /* must precede any other NST call */
   u = myunit(ROOT);   /* make instance of u, using host unit ROOT */
   exec_unit(u);       /* prints the empty string "" */
   NstInpStrCpy(u,0,0,"hello world");  /* set input string */
   exec_unit(u);       /* prints "hello world" */
   NstInpStrCpy(u,0,0,"H-E-L-L-O W-O-R-L-D"); /* set input string */
```

```
    exec_unit(u);          /* prints "H-E-L-L-O W-O-R-L-D" */
}
```

Listing 7: *Hello world*-example with string input

Fields. The entire set of input fields of a unit `u` is implemented as an array of elements of the NST-predefined C-type `field` (the pointer `u->in` points to the first element in this array). An analogous organization exists for the output fields of a unit (here the first element is at `u->out`). The functions `nst_inputs_of(u)` and `nst_outputs_of(u)` return the number of elements in these arrays. The macros `nst_input_f(u,k)` and `nst_output_f(u,k)` return `field*` pointers to the field struct that represents input (output) field `k` of a unit `u`.

Each input or output field can hold a record of one or several *pins* The number of pins in a field `pField` is called its *field dimension*. Its value is returned by the macro `nst_dim_of_f(pField)`. The macro(s) `dim_of_X(u,f)` (`dim_of_Y(u,f)`) return(s) the dimension of input (output) field in terms of its position `f` in the input (output) interface a unit `u`

! There is one caveat, however: for fields that consist of a single float vector pin (cf. below), `dim_of_X/Y()` will not return the expected value of 1, but instead *minus the number of elements in the float vector*. `u->in[f]` and `u->out[f]` point to the f-th input/output field of `u`.

Pins. Each field is implemented as an array of *pins*. Each pin corresponds to one data item in the record that is represented by the field. The data item associated with a pin depends on the type of the pin and can be a single float value, a short int, an unsigned char or a null-terminated character string. Function(s) `pNstInp(u,f,i)` (`pNstOut(u,f,i)`) return(s) a pointer to the pin at position `i` in the input (output) field at position `f` of unit `u`.

Here is a summary of the above-mentioned access functions for the inputs of a unit (an analogous set of functions is defined for the outputs):

name	return value
iNstInpNum(u)	number of input fields of unit u
pNstInpField(u,f)	pointer to f-th input field
iNstInpDim(u,f)	dimension (= nr of pins) of f-th input field
pNstInp(u,f,i)	pointer to i-th pin of f-th input field

Data types. The data item held in a pin can be of one of the six basic types `NST_SCALAR`, `NST_FLOAT`, `NST_INT`, `NST_BYTE` `NST_STRING` or `NST_ANY`. These correspond to C-data types as described in Table 1.

NST_SCALAR	a single float value
NST_FLOAT	an array of float values (“float vector”)
NST_INT	an array of short int values (“int vector”)
NST_BYTE	an array of unsigned char values (“byte vector”)
NST_STRING	a variable-length null-terminated character string
NST_ANY	for future extension to handle arbitrary data types

The number of data elements in the data item of a pin is returned by the function `nst_dim_of_p(p)` and is called its *pin dimension*. Likewise, `nst_size_of_p(p)` returns the byte size of the data array held in pin `p` (for a pin of type `NST_STRING`, this is the string length plus one).

! It has been turned out to be convenient for the usage of many NST-units to implement them in such a way that a n-dimensional input (output) can either appear in the form of a field with n separate pins of a single

NST_SCALAR float value, or as a field with a single pin of type NST_FLOAT with n elements. Which alternative is chosen is decided with the call of the constructor function and a recommended convention is that a positive value for the parameter that determines the dimension of such a field chooses the first (NST_SCALAR) variant, while a negative parameter value $-abs(n)$ is taken to specify the second (NST_FLOAT) variant (with $+abs(n)$ elements in the vector). Note also that this convention is congruent with the dimension values that are returned by the dim_of_X/Y()-macros explained earlier. [4] Implementing a unit in this way is usually more work (since the access macros for the scalar and the vector fields differ, and an efficient implementation will also make use of the additional possibilities for optimization in the case of the latter), but also yields more flexibility for its use.

The function nst_type_of_p(p) returns the type of the data in the pin pointed to by p. The return value is one of the integer constants NST_SCALAR, ... NST_ANY.

! Many existing NST-routines use the function t_of_p(), which also returns the type of a pin. However, instead of the integer constants NST_SCALAR.. this function returns a different set of character constants NST_SCALAR_T='s',... and should be reserved for NST's internal use.

Access of data elements. Macro pfNstInp(u,f,i) returns the address of the first data element in the data item of a float vector pin at position i of the f-th input field of a unit u. Data elements are stored in simple array order, thus, the k-th data element in the float vector is given by pfNst(u,f,i)[k]. Analogous macros pcNstInp, piNstInp, ptNstInp exist for the other vector data types and for the string type. To access the single float value of a scalar float input pin, use fNstInp(u,f,i). A corresponding set of macros fNstOut,pfNstOut,.. is provided for output pins.

Here is a table of these access macros for accessing a pin of the f-th input field of a unit u:

name	return value
fNstInp(u,f,i)	value of i-th pin (pin must be scalar)
pNstInp(u,f,i)	pointer to i-th pin (all pin types)
pfNstInp(u,f,i)	start of float array in i-th pin, which must be NST_FLOAT
piNstInp(u,f,i)	start of short int array in i-th pin, which must be NST_INT
pcNstInp(u,f,i)	start of unsigned char array in i-th pin, which must be NST_BYTE
ptNstInp(u,f,i)	start of character string in i-th pin, which must be NST_STRING

! Note that neither NST_FLOAT,.. nor NST_FLOAT_T,.. are C-data type specifiers. Instead, they denote integer or character constants that are used to pass type information to NST-functions, such as primitive_unit() for making units with pins of a specified type (see below).

As an example of the use of these macros, we consider a code fragment that sets all input values of a given unit u to zero, whatever their type etc. may be:

```
#include "nst.h"
#include "nst_strg.h"
unitptr u; /* the NST-unit that we consider */
...
u = ... /* some NST-creation function here */;
...
for (f=0; f<nst_inputs_of(u); f++) {
   pin *pPin;   /* pin is a NST-defined type */
   for (i=0; i<nst_input_pins(u,f); i++) {
      pPin = NstInp(u,f,i);
      iDim = nst_dim_of_p(pPin);
      switch (nst_type_of_p(pPin)) {
         case NST_SCALAR: fNstInp(u,f,i)=0; break;
```

[4] the documentation refers in many places to fields that have only a single pin of one of the vector types NST_FLOAT..NST_BYTE as “packed fields”.

```
         case NST_FLOAT:
            for (k=0; k<iDim; k++) pfNstInp(u,f,i)[k]=0;
            break;
         case NST_INT:
            for (k=0; k<iDim; k++) piNstInp(u,f,i)[k]=0;
            break;
         case NST_CHAR:
            for (k=0; k<iDim; k++) pcNstInp(u,f,i)[k]=0;
            break;
         case NST_STRING:  /* strings are special: */
            NstInpStrCpy(u,f,i,""); break;
         default:  exit (0); /* error: unknown type! */
      }
   }
}
...
```

Listing 8: Accessing values in the NST-interface

Setting string values. At this point we remark that setting values for pins of type `NST_STRING` requires some care, since setting of strings requires a mechanism that dynamically adjusts for the varying length of strings. To ensure that the appropriate adjustments are made, changes to pins of type `NST_STRING` should, therefore, be made via the functions `NstInpStrCpy(u,f,i,pcStr)` and `NstInpStrCat(u,f,i,pcStr)`, which copy or append `pcStr` (in)to the string for an input pin of string type (cf. Listing 4). Similar functionality, but with the possibility of an additional offset is provided by the functions `NstInpStrCpy1, NstInpStrCat1` and a mirror set of analogous functions exists for output pins.

! NST's string manipulation functions (located in `nst_strg.c` use *dynamic strings* that automatically reallocated to the next integer multiple of `NST_STRBLKSIZ` (default=20) bytes if need arises. However, this requires that all write accesses occur via NST's string handling functions (see the man page `NstStr` for a description).

5 Private Variables

A scheme for handling persistent objects should also offer a way to preserve private state information across invocations of each object. While this is again directly offered in the class data type of C++, NST uses a simple implementation in C that is based on the definition of a separate `struct type` for each class (that needs such private variables). This approach is somewhat analogous to the way methods are implemented: again a pointer (named `var`) in the `unitptr` struct is set, this time to an instance of the `struct type` that must have been defined to hold all private data that need are needed by the class. The necessary steps, including the allocation and proper initialization of an instance of the struct that is defined to hold the private variables needed by the class, must also be carried out in the constructor function.

To clarify the approach, we again return to our "Hello world" example, using the version with the string input field shown in Listing 4. We now want to extend this such that each unit of the class `myunit` keeps track of the average length of all text lines it has printed, as well as of the total number of invocations of the `exec`-method since its creation.

Here is a suitably extended version of the listing 4 that offers the new feature:

```
#include <stdio.h>
#include "nst.h"   /* necessary NST headers */

typedef struct var_myunit{
   int iNumInvocations;
```

```
   float fAvgStrLen;
} var_myunit_t;

int what_myunit_shall_do(unitptr u) {   /* myunit's method */
   char *pcThisIsMyInputString = ptNstInp(u,0,0);
   var_myunit_t *pVar = nst_var_of(u);
   /* define & set pointer to private var struct */
   printf("%s\n", pcThisIsMyInputString);
   /* update private variables with new entry: */
   pVar->fAvgStrLen = (pVar->iNumInvocations*pVar->fAvgStrLen
       + strlen(pcThisIsMyInputString))/(pVar->iNumInvocations+1);
   pVar->iNumInvocations++;
   return OK;   /* default return value for NST-units */
}

unitptr myunit(unitptr uHost) {  /* the constructor function */
   var_myunit_t *pVar;
   unitptr uNew = primitive_unit(NST_INP,NST_STRING,10,NST_HOST,uHost);
          /* makes a unit with a string input of initially 10 chars */
   if (!uNew) return NULL;            /* return NULL if failure */
   uNew->exec = what_myunit_shall_do; /* make uNew use above method */
   /* make instance of var_myunit_t struct to hold private data: */
   uNew->var  = pVar = CALLOC(1,var_myunit_t);
                              /* make private var struct instance */
   pVar->fAvgStrLen = pVar->iNumInvocations = 0;  /* initialize */
   label_unit(uNew,"myunit");  /* assign a print name to myunit     */
   return uNew;                /* return handle to created instance */
}
...
```

Listing 9: *Hello world*-example with private variables

Again, a macro `nst_var_of(u)` is provided to access the pointer value `u->var`. Note that when a NST-unit `u` is removed (by invoking `remove_unit(u)`, NST automatically frees the object at the address pointed to by `u->var`. Thus, there is no need to write extra code for freeing the `var_myunit_t` instance that is associated with each instance of class `myunit`. Only if `var_myunit_t` contains pointers that reference other memory objects, these are not freed automatically by NST's default remove routine; the responsibility to care about their proper cleaning up is entirely within the implementation of the new unit and a corresponding non-default remove routine may be necessary in these cases.

! Many existing NST-units use a second way of storing and retrieving private variables: NST has three macros `vinit()`, `vput()`, `vget()` to initialize a private *variable stack* for each unit and to put and retrieve data objects of arbitrary type on/from it. With these macros, the storage of the two variables `fAvgStrLen` and `iNumInvocations` of the above example could have been made without resort to the definition of an extra `var_unit_t`-struct:

```
int exec_myunit(unitptr u) {
    int iNumInvocations; float fAvgStrLen=0;
    ...
    vinit(u);  /* prepare private var stack for retrieval */
    fAvgStrLen      = vget(u,float); /* first var to get */
    iNumInvocations = vget(u,int);   /* last var to get */
    ...
}
...

unitptr myunit(...) {
    int iNumInvocations=0; float fAvgStrLen=0;
    ...
    vput(uNew,int,iNumInvocations); /* last var to get must be put first! */
    vput(uNew,float,fAvgStrLen);    /* first var to get must be put last! */
    ...
```

```
        return uNew;
    }
```

Note that variables must be gett'ed in the inverse order in which they were putt'ed earlier. Note also that the `vput()`-macro requires a variable identifier as its last argument; a constant will not do. While this scheme is convenient and handy (and includes further conveniences explained in the man pages of vput and vget), if there are only a few private variables, it get error prone for larger numbers, and the `struct`-base approach should then be preferred.

! Important: you cannot use the old and the new scheme simultaneously for the same unit! Either you use `nst_var_of()` *or* you use the `vput/vget` macros (You can skip the following, but the reason for this restriction is that `vput(u,t,adr)` does not set simply `u->var` to `adr`; instead it creates an auxiliary memory block holding references to the next queue element and to (a copy) of the object referenced by `adr`. As a result, `nst_var_of(u)` does not return the address of the first element putted by a `vput`. Conversely, `vget` does not return the address set by a previous assignment `nst_var_of(u)=adr` (see `nst6.[ch]` for details)).

Having now discussed all important points to implement new NST units, we now show as a more realistic example the "wrapping" of a C-subroutine from the Numerical Recipes software collection as a NST-unit. This also illustrates how to deal with the particular scheme of Numerical Recipes to implement arrays via vectors of pointers with index shifts such that all indices have the "fortran" range of 1..n.

For our example, we implement the `gaussj` subroutine as a NST-unit. The `gaussj` subroutine takes four arguments:

```
gaussj(float **ppfA, int iN, float **ppfB, int iM)
```

It inverts a `iN x iN` matrix `ppfA` "in place" and applies the inverted matrix to a second matrix of `iM` `iN`-dimensional column vectors that specify the `iM` right-hand sides $\vec{y}$ of `iM` linear systems $\mathbf{A}\vec{x} = \vec{y}$ to be solved. Thus, after return of the routine matrix `ppfB` contains the `iM` result vectors.

Below we show the code for a wrapper that makes this subroutine usable in the form of a NST-unit with two input fields. The first input field has a single float vector pin of `iN*iN` elements for the matrix $\mathbf{A}$, the second input field has a single float vector pin of `iN*iM` elements for the matrix of the `iM` right hand side column vectors $\vec{y}$. Two analogous output fields provide the computed result values. (the code assumes linkage with `gaussj.o` from the Numerical Recipes routines; if you just want to compile the thing, you can replace the `external gaussj(..)` declaration by a corresponding dummy routine that does nothing).

```
#include <stdio.h>
#include "nst.h"          /* necessary NST headers */
#define TESTMODE          /* to compile also main() below */

typedef struct var_mygauss{
   int iN,iM;  /* the matrix dimensions */
   float **ppfA, **ppfB;  /* to access Numerical Recipes - type matrices */
} var_mygauss_t;

extern void gaussj(float **ppfA, int iN, float **ppfB, int iM); /* { return; } */

int exec_mygauss(unitptr u) {  /* myunit's method */
   var_mygauss_t *pVar = u->var;
   memmove(pfNstOut(u,0,0), pfNstInp(u,0,0), nst_size_of_p(pNstInp(u,0,0)));
   memmove(pfNstOut(u,1,0), pfNstInp(u,1,0), nst_size_of_p(pNstInp(u,1,0)));
   gaussj(pVar->ppfA,pVar->iN,pVar->ppfB,pVar->iM);
   return OK;
}

int remove_mygauss(unitptr u) {
   var_mygauss_t *pVar = u->var;
```

```
   free(pVar->ppfA+1); free(pVar->ppfB+1);  /* +1 to undo prior offsets! */
   return nst_remove0(u);
}

unitptr mygauss(int iN, int iM, unitptr uHost) {  /* the constructor function */
   unitptr uNew;  int i;
   var_mygauss_t *pVar;
   uNew = primitive_unit(
             NST_INP,NST_FLOAT,iN*iN, /* NxN matrix A as single vector */
             NST_INP,NST_FLOAT,iN*iM, /* NxM matrix B as single vector */
             NST_OUT,NST_FLOAT,iN*iN, /* dito for outputs */
             NST_OUT,NST_FLOAT,iN*iM, NST_HOST,uHost);

   if (!uNew) return NULL;            /* return NULL if failure */
   pVar = CALLOC(1,var_mygauss_t);
   pVar->iN = iN; pVar->iM = iM;
   pVar->ppfA = CALLOC(iN,float*)-1; /* ptr to iN rows, offset -1     */
   pVar->ppfB = CALLOC(iN,float*)-1; /* makes first el start at pos 1 */

   /* set pointers (offset -1 makes each row vector start at index of 1) */
   for (i=0; i<iN; i++) { /* assign row vectors, each iN or iM els long  */
      pVar->ppfA[1+i] = pfNstOut(uNew,0,0) + i*iN - 1;
      pVar->ppfB[1+i] = pfNstOut(uNew,1,0) + i*iM - 1;
   }  /* offset "-1" makes each row vector start at index value of 1 */

   for (i=1; i<=iN; i++) pVar->ppfA[i][i]=1; /* init to identity matrix */
   uNew->exec = exec_mygauss;  /* make uNew use above method        */
   uNew->var = pVar;           /* set private variable ptr          */
   label_unit(uNew,"mygauss"); /* assign a print name to myunit     */
   return uNew;                /* return handle to created instance */
}

#ifdef TESTMODE

main() { /* test for Ax = b, A = 2x2, b = 2x1 given below */
   extern unitptr print_mat(int,int,char*,char*,unitptr);
   unitptr uGauss,uShowInverse,uShowSolution;  int i,iN=2,iM=1;

   float afA[4]={ 1, 2, 3, 4 }, afB[2] = { 0, -2};
   init_nst();         /* must precede any other NST call */
   uGauss = mygauss(iN,iM,ROOT);   /* make instance of u, using host unit ROOT */
   /* set input data: */
   for (i=0; i<iN*iN; i++) pfNstInp(uGauss,0,0)[i]=afA[i];
   for (i=0; i<iN*iM; i++) pfNstInp(uGauss,1,0)[i]=afB[i];
   exec_unit(uGauss);
   /* we use two instances existing NST-units to display the result: */
   uShowInverse  = print_mat(iN,iN,"Inverse   90 940 200 50"," %8.3v",ROOT);
   uShowSolution = print_mat(iM,iN,"Solution 300 940  75 50"," %8.3v",ROOT);
   joinfields(uGauss,uShowInverse,out_0,in_0);
   joinfields(uGauss,uShowSolution,out_1,in_0);
   exec_unit(uShowInverse); exec_unit(uShowSolution);
   sleep(5);
}
#endif TESTMODE
```

Listing 10: Wrapping a Numerical Recipes routine as a NST-unit

Here, the struct type `var_mygauss_t` holds as private variables the address pointers `ppfA`, `ppfB` to the two required arrays, as well as the two dimensions `iN, iM`. Since routines from Numerical Recipes represent a matrix as an array of pointers to the row vectors of the matrix, we must allocate two `iN`-dimensional float pointer arrays (since both matrices A, B have `iN` rows) and set `ppfA`, `ppfB` to the starting address of these arrays.

! The Numerical Recipes provide a set of utility functions for the allocation and deallocation of such arrays, as well as an error printing routine. These routines are used by most other Numerical Recipes programs and are collected in `nrutil.c`. To simplify the usage of NST together with Numerical Recipes routines, `libnst.a` contains an analogous set of routines, however with error messages integrated into NST's error messaging system. Thus, using a Numerical Recipes routine requires to link with the object file for that routine only; a further linking with `nrutil.o` *is not necessary*.

Since Numerical Recipes routines always use 1 as their starting index value, we must use shifted address values so that the element addressed by an index value of 1 is the initial element of the actually allocated memory range. Note that these pointer arrays do not yet contain the memory locations for the array elements themselves. Instead of allocating them extra (e.g., with the routines in `nrutil.c` of the Numerical Recipes Package), a better way here is to directly use the elements already allocated in the input and output arrays `pfNstInp(uNew,0,0)[]`, `pfNstInp(uNew,1,0)[]`,`pfNstOut(uNew,0,0)[]` and `pfNstOut(uNew,0,0)[]` of the created unit. Since `gaussj()` overwrites the given array data with the computed result values, we direct `ppfA` and `ppfB` to the output arrays `pfNstOut(uNew,0,0)[]` and `pfNstOut(uNew,1,0)[]`. This requires that in the exec-method the input values in `pfNstInp(uNew,0,0)[]` and `pfNstInp(uNew,1,0)` are first copied into the corresponding output arrays `pfNstOut`, so that the subsequent call of `gaussj(ppfA,iN,ppfB,iM)` can work on the proper input data. Note that this economic scheme requires that we represent the input and output data as contiguous arrays (i.e, as one of NST's "vector" data types) and that we adopt a convention for the arrangements of matrix elements in the input and output arrays of the NST unit that is compatible with the convention in Numerical Recipes: matrix elements are arranged in row-wise order, concatenated into one long vector that starts with the topmost row $a_{11}, a_{12}, \ldots a_{1n}$, continuing with the next $a_{21}, a_{22}, \ldots$, and finally, ending with a_{nn} as the last element. For these reasons, this is also the convention used by existing NST-units that involve matrix operations.

! 1. The allocation of the auxiliary pointer arrays requires to provide a `remove` method that frees the allocated memory when the unit is destroyed. This point is discussed in the next section **??**, but we include the necessary code here to make the example complete. Note also that we must not forget to undo the prior address offsets to the pointers, since the `free()`-routine expects to *get the same address that an earlier allocation routine has returned!*.

! 2. There is a pitfall if we wish to adopt the reverse scheme, namely to direct the `ppfA` and `ppfB`-pointers to the *input* arrays `pfNstInp` to do the in-place computation there, and copy the result *thereafter to the outputs*: when NST units are connected, NST just *maps* the memory locations of the source unit's outputs into the destination unit's interface and discards any previously allocated memory there. While this is efficient, it has the side effect that *the memory address of values in the input interface of a NST unit cannot be assumed to stay fixed*. Thus, with the second scheme we cannot set the `ppfA`, `ppfB`-pointers once for all in the constructor function, but instead *must* set them each time anew in the `exec` function itself. This makes the first scheme the preferred choice.

The `main()`-program creates an instance of `mygauss`, sets the inputs to some test data and displays the inverted matrix and the solution vector (i.e., `iM=1`), using two instances of NST's `print_mat`-unit to display the result in two small `X`-windows (instead of printing it). It also illustrates the technique of offering a conditionally compiled `main()` program with testing statements for a new implementation, that can be disabled with a `#undef TESTMAIN` directive at the beginning of the file.

! The previous unit would do a full matrix inversion each time it is executed. In many cases, the data at the inputs will only change from time to time, and it would be convenient to have a means to de-activate the computations of the unit, when nothing interesting happens. To this end, many NST-units are implemented with an extra input field, usually with a single pin of type `NST_FLOAT` and their method routines are written such that they perform their computations only if this *"control pin"* has a non-zero value. In the past, there was the tacit convention that such control pin should always be the first scalar float pin in the last input field. The new version of NST provides the routine `nst_def_ctlpin(u,f,i)` that *defines* pin i of input field f of unit u to be the control pin; the similar routine `nst_init_ctlpin(u,f,i)` has the same effect but in addition sets the pin value to 1. When a unit shall offer a control pin for disabling its computation, one of these routines should always be used, since then NST "knows" what the control pin is and can automatically skip any computations for the unit. In addition, *Neo*will then show the control pin in a different shape, making it easier to know which units can be disabled and which cannot.

6 Further NST-defined Methods

Unlike C++ classes, that can be defined with arbitrarily many methods, NST-units have a fixed default set of methods. We have so far mainly discussed the `exec` method, which is considered to be the "main method" for each class.

! Of course, by introducing private variables that are suitable function pointers, the set of methods for a NST unit can be arbitrarily extended. However, one of the virtues of NST-classes is the high degree of uniformity that results from the restriction to only five fixed methods. Thus, the mentioned technique should, if at all, be used very sparingly, since in most cases the same effect can be achieved by a thoughtful design of the unit's exec and ctrl methods, in particular in combination with settings at the unit's input field.

The remaining NST methods are the following:

6.1 Remove Method

The `remove_unit(u)` call invokes the remove method for a NST-unit `u`. The semantics of this method should be to release all resources that are private to the unit `u`. This includes all its private variables, data in its interface and – possibly – the closing of any files or windows that "belong" to this unit; it also includes the removal of all subunits that `u` may have. The units that are returned by the `void_unit` and the `primitive_unit()` constructors (including their variants) have as their remove routine NST's default remove routine `int nst_remove0(unitptr)`. In all cases in which the implementation adds no private variables, or can do with a private variable struct that does not contain any pointers, this default remove routine will cover all necessary actions. Only if the private variable struct introduces references to additionally allocated memory objects, it becomes necessary to replace the default remove routine by a class-specific one. The necessary steps follow the general pattern

```
int remove_myunit(unitptr u) {
  ... class specific clean-up actions for u here ...
  [but leave u, its var-struct and its interface untouched!]
  /* finally, invoke NST default remove routine for clean-up
     of u itself, its var-struct, subunits and interface: */
  return nst_remove0(u);
}

...

unitptr myunit(...) { unitptr uNew;
   ...
   uNew = primitive_unit(...);
   ...
   uNew->remove = remove_myunit;  /* replace default remove */
   ...
   return uNew;
}

...
```

Listing 11: General pattern for the Remove method

(a more detailed example will be provided in section 13).

6.2 Load and Save Methods

The `save_unit(fp,u)` call invokes the save method for a NST-unit `u`. The semantics of this method should be to save all internal state information of unit `u` to the file pointed to by the `FILE* fp` such that a subsequent `load_unit(fp,u)` call from the same file location `fp` (and for the same unit `u`) can restore the state that was valid at the time of the previous invocation of save.

Again, the units that are returned by the `void_unit` and the `primitive_unit()` constructors (including their variants) come with predefined default load and save routines `nst_load0(FILE*,unitptr)` and `int nst_save0(FILE*,unitptr)` that load and save the *the subset of all scalar and text string values* in the input interface of the created unit. Only if a unit needs to keep additional state information it becomes necessary to replace the default routines by extra written routines (which must be of the type `int(*)(FILE*,unitptr)`) and to set the `unitptr load` and `save` function pointers to these routines.

! 1. The saving and loading of the interface variables of a unit is accomplished with the NST functions `ivread()` and `ivwrite()`. The variables that are saved/loaded by these routines include only those on the input side, since any output variables will be restored anyway from the saved inputs after the first invocation of a unit. The exclusion of an automatic save for the vector type pins has been made to avoid huge save files for circuits with high-dimensional vectors, such as image data, that are in most cases no essential state information for a unit.

! 2. Currently, all save routines write their data in formatted form. This has the advantage of readable and editable save files, but also introduces some roundoff for float values. If huge amounts of data are to be saved, or roundoff is intolerable, on should write a customized save routine that uses unformatted data. However, on should avoid the occurrence of nullbytes in a NST-save file, since some routines for the remote execution of NST-units rely on treating NST-save files as long strings. To avoid this, the routine pair `nst_encode()` allows to translate binary data into suitable nullbyte-free format (at a small increase in size); `nst_decode()` allows to translate back into the original binary form.

A suitable pair of save/load routines for the `myunit` of Listing 6.2 must save/load the private variables of the `pVar` struct. A simple implementation is shown below:

```
...

int save_myunit(FILE *fp, unitptr u) {
   var_myunit_t *pVar = nst_var_of(u);
   ivwrite(fp,FFMT,u);  /* saves data of u's interface */
   fprintf(fp,"%d %g\n",pVar->iNumInvocations,pVar->fAvgStrLen);
   return OK;
}

int load_myunit(FILE *fp, unitptr u) {
   var_myunit_t *pVar = nst_var_of(u);
   ivread(fp,FFMT,u);   /* reads interface data saved by ivwrite() */
   fscanf(fp,"%d %g",&(pVar->iNumInvocations),&(pVar->fAvgStrLen));
   return OK;
}

...

unitptr myunit(...) { unitptr uNew;
   ...
   uNew->save = save_myunit;
   uNew->load = load_myunit;
   ...
   return uNew;
}
```

Listing 12: Save and Load methods for the unit of Listing

A more realistic example is given in section 13.

6.3 Ctrl Method

Sometimes, the single `exec` method is not sufficient for a convenient implementation of a class. Often, there is a need to define a set of additional auxiliary or control operations, such as re-initializing an object, clearing something etc. The `ctrl` method is provided to implement such additional operation. It is invoked with the call `ctrl_unit(iMode,u)` where the first (integer valued) parameter is provided for the choice of one of a discrete set of alternatives. The default control routine that NST provides does nothing; thus, any control operations must be entirely implemented from scratch.

Again we make use of our *Hello world*-unit to illustrate the implementation of a simple `ctrl`-method: the call `ctrl_unit(iMode,u)` shall print the values of the two state variables `iNumInvocations` and `fAvgStrLen` when `iMode` has the value `MYUNIT_INFO` (some constant value to be defined with the implementation of the class `myunit`), and do nothing otherwise:

```
...
#define MYUNIT_INFO 92547   /* some int not used as mode by other classes */

int ctrl_myunit(int iMode, unitptr u){
   var_myunit_t *pVar = nst_var_of(u);
   switch (iMode) {
      case MYUNIT_INFO:
         printf(" # of past invocations: %d\n",pVar->iNumInvocations);
         printf(" average string length: %g\n",pVar->fAvgStrLen);
         break;
      /* .. maybe further choices here .. */
      default:  break; /* no op */
   }
   return OK;
}
...
unitptr myunit(...) { unitptr uNew;
  ...
  uNew->ctrl = ctrl_myunit;
  ...
  return uNew;
}
```

Listing 13: A control routine for `myunit`

6.4 Adapt Method

The `adapt_unit(u)` call invokes the adapt method for a NST-unit `u`. This method has been introduced to especially support the implementation of NST classes for adaptive units, such as neural networks. Thus, the semantics of this method should be to carry out any adaptive changes to internal state variables (such as the weights of a neural network) that result from the currently valid inputs.

In many cases, adaptation in a chain of interconnected units (such as in a sequence of consecutive neural network “layers”) A B C .. D depends on the previous execution of each unit in the forward direction A B C .. D to compute from the outputs of its predecessor unit valid inputs for its successor in the chain. Then, during a “backward pass”, the final output of the last unit D is compared with a “target value”, and the observed error difference is “backpropagated” along the chain, but in the inverse direction D .. C B A. At each stage, the error signal computed by the successor unit (say, C) is sent to an “error input” of the previous unit (here, B), which uses it to perform an adaptation step for its internal weights and which also computes from it a transformed error signal that then is valid and sent to the error input of its predecessor unit (here, A).

Therefore, the `adapt` method has been defined in a way that differs in an important respect from the definitions of the other NST-methods: when `adapt_unit(u)` is invoked for a `void_unit` that has subunits, the `adapt` method is invoked for the *last* subunit *first*, then for the next-to-last subunit and so on, with the *first* subunit adapted *last*. The same reversed traversal is used for the `adapt_subunits_of()` routine, which must be used in order to apply the `adapt` method for the subunits of a primitive unit.

Apart from this reversal in execution order, all that has been said for the `exec` method applies in an analogous fashion for the `adapt` method. An example of the implementation of an `adapt` routine is given in section 13.

7 Template-based Programming

Many parts of the code for a NST-unit are of a rather stereotypic pattern. Thus, a good way to implement a NST-unit is from a template program in which once for all the numerous necessary but tedious little things like extensive error checks, skeleton texts for comments and documentation, testing sections and so on have been laid down.

The shell script `mku` generates such a skeleton; it should preferably used as a quick starting point for the implementation of a new unit. It produces also a comment section for a description of the unit and its interface such that a manual page can be produced by another shell script `mma` (`mma` is an improved version of the Manual page generator `mmp`; it also allows to produce man pages in nroff and HTML-format).

`Mku` also generates a suitable makefile `name.mak` and a second source file `neo_name.c` that contains a skeleton with wrapper code for integrating the new unit into the graphical editor *Neo*.

8 Operator units

In many cases, one would like to use functions or, more generally, NST-units as parameters of other units. For example, we might want to have a unit `gradient_op` that computes the gradient of a function $f(\vec{x})$. $f(\vec{x})$ could be given as a C-function or it could itself be implemented as a NST-unit with an input field for the argument vector $\vec{x}$ and an output field for the scalar result value $f(\vec{x})$. An obvious solution would be to implement the gradient computing unit with a creation function that takes a suitable function pointer or a `unitptr` to an instance of the unit that computes $f(\vec{x})$, leading to the following scheme (for the case that $f()$ is implemented as a NST-unit):

```
typedef var_gradient_op1_t { unitptr uFun; } var_gradient_op_t;

unitptr gradient_op1(
   int iDim,  /* dimension of x and of gradient vector */write a creation
   unitptr uFun, /* unit that implements f(x) */
   unitptr uHost /* host unit */
) {  var_gradient_op_t *pVar;
  uNew = ...
  ...
  pVar = CALLOC(1,var_gradient_op_t);
  pVar->uFun = uFun;  /* let uFun be one of the private variables */
  uNew->var  = pVar;
  ...
  return uNew;
}

int exec_gradient_op1(unitptr u) {
  var_gradient_op1 *pVar = nst_var_of(u);
  ...
```

```
  /* use pVar->uFun to compute gradient */
  ...
}
```

Listing 14: Example of an operator unit: poor version

However, this scheme is rather inflexible: it requires to specify the unit that is to be used for the computation of `f()` already at the time when the `gradient_op1` unit is created. Later, `uFun` can be replaced by a different unit only if `uFun` is overwritten by the new unit, since the `gradient_op1`-unit will always look at the fixed address given when it was created. Finally, the `gradient_op1` unit could not be easily used within the graphical editor *Neo*: for a unit to be useable under *Neo*requires that all values for its creation parameters can be entered from the keyboard; obviously, this is hard to achieve for pointer values, such as `uFun`. Had we used a function pointer instead, the situation would be very similar.

To achieve a more flexible binding, NST provides the concept of *operator units*. An operator unit is like a normal NST unit; however, when it executes (adapts,ctrls...) it may issue execution (adapt, ctrl...) calls to other units that are its (usually) successors in the NST-execution sequence. These units are called the *operands* of the operator unit.

This is a very natural scheme: for the gradient, we now can define a `gradient_op` unit that acts much like the gradient operator ∇: it is simply created before the unit `uFun` (i.e., the `gradient_op` and the `uFun` instance share the same host unit, and `uFun` is successor of `gradient_op`) that implements the to-be differentiated function. Execution of the `gradient_op` unit then uses its first successor unit as its operand to compute the value of the derivative, which can be, e.g., provided at output field `out_0` of the `gradient_op` unit.

The following lines provide a simple implementation for the constructor function part of the `gradient_op` class:

```
unitptr gradient_op(
  int iDim, /* dimension of gradient */
  float fStepSize,  /* step size for numerical approximation of gradient */
  unitptr uHost)
{  unitptr uNew;
   if (!(uNew = primitive_unit(
      NST_INP, NST_FLOAT, iDim,    /* vector input pin for x    */
      NST_INP, NST_SCALAR, 1,      /* scalar input pin for fEps */
      NST_OUT, NST_FLOAT, iDim,    /* vector output pin for gradient */
      NST_HOST, uHost))) return NULL; /* failure */
   fNstInp(uNew,1,0) = fStepSize; /* initialize step size: */
   nst_operands_of(uNew) = 1;     /* this specifies # of operands */
   uNew->exec = exec_gradient_op;
   label_unit(uNew,"gradient_op");
   return uNew;
}
```

Listing 15: Example of an operator unit

Here, the new macro call `nst_operands_of(uNew)=1` stores in the `operands` field of `uNew` that `uNew` has 1 operand unit. This flags `uNew` as an operator. However, it is entirely in the responsibility of the exec method to query the value of `nst_operands_of(uNew)` and, if not zero, to take appropriate action that makes `uNew` behave like an operator.

In the present example, the `exec` method should invoke the exect method of the successor unit to compute the gradient. We will describe two different implementation schemes. The first scheme assumes that we already have the required algorithm in the form of a `C`-function from some suitable library, and that we want to make this algorithm available as a NST-operator unit. The second scheme illustrates how to proceed with an implementation from scratch.

For the first approach, lets us assume that we have a function

```
void comp_grad(
    int iDim, float pfX,    /* dimension and value of argument vector x */
    float *pfGrad,          /* returns computed gradient vector */
    float fEps,             /* step size parameter etc. */
    float (*f)(int,float*) /* ptr to the to-be-differentiated function */
);
```

As is typical for such routines, `comp_grad` requires the to-be-differentiated function to be specified by a function pointer, just the feature that we wanted to avoid!

The solution is to wrap the execution of the operand unit, which is assumed to implement $f(\cdot)$, suitably so that its computation can be made available for `comp_grad` via a function pointer. The exec-function for `gradient_op` would then just make the necessary call of `comp_grad()`. Here's a way how this can be set up:

```
static unitptr guOpndOfGrad; /* auxiliary variable */

static float operand_as_function(int iDim, float *pfX) {
   /* fetch input values for operand unit from vector pfX[]: */
   memcpy(pfNstInp(guOpndOfGrad,0,0),pfX,iDim*sizeof(float));
   exec_unit(guOpndOfGrad);  /* now operand has computed f(X) */
   return fNstOut(guOpndOfGrad,0,0); /* return operand's result */
}

int exec_gradient_op(unitptr u) {
   if (u->next==NULL) error_msg(0,"gradient_op",0,"operand is missing!");
   else {
      guOpndOfGrad = u->next;    /* temporarily use u->next as operand */
      enable_exec_of(guOpndOfGrad);  /* (explained in text) */
      comp_grad(ABS(dim_of_X(u,0)),  /* iDim inferred from input field */
                   pfNstInp(u,0,0),  /* main input vector pfX from inp0 */
                   pfNstOut(u,0,0),  /* deposit gradient result in out0 */
                   fNstInp(u,1,0),   /* take step size eps from inp1 */
               operand_as_function); /* ptr to operand wrapper function */
      disable_exec_of(guOpndOfGrad); /* (explained in text) */
   }
   return OK;
}
```

Listing 16: Exec-method for the gradient operator

As before, the argument pointers of `comp_grad` are just set up to make the routine work directly on the data in the input and output interface of the unit (auxiliary arrays would have to be set up in the case of `NST_SCALAR` fields to pass the data as contiguous arrays to `comp_grad`). The function `operand_as_function` wraps the computation of the operand unit into the call of a `C`-function of the type required by the last argument parameter of `comp_grad()`. However, since the parameter list of this function cannot have a slot for the to-be chosen operand unit (which will be different for different instances of `gradient_op`!), it must be accessed via a global parameter `guOpndOfGrad` that is set to the correct address just before the `exec`-method is going to call `comp_grad()`. This mechanism implements the dynamic binding for operand of the gradient.

! For the sake of a compact example, we require any operand of the `gradient_op` to have a `NST_FLOAT` pin at position 0 of its input field 0 for the value of the vector $\vec{x}$ from which the exec method of the operand evaluates $f(\vec{x})$. Likewise, we require this result to be returned at a single `NST_SCALAR` float pin at position 0 of output field 0. In a robust implementation the exec-method should at least check for the presence of these pins and their types in the operand unit and take appropriate error actions in case of a mismatch, since an attempt to access a non-existing pin usually results in a fatal error.

We still need to explain the `nst_en/disable_exec_of()`-calls that bracket the main part of the exec-routine: in their absence, after completion of the exec-method for a `gradient_op` instance `u` the normal NST-execution flow would continue with an invocation of the exec method for the next unit. However, this is just the operand unit `u->next` that has already done its job under the control of `u's` exec method! Clearly, the natural semantics for the operator-operand scheme is that an operand of `u` should participate in the execution flow *only under the control of the operator u's* exec method, not subsequently and additionally on its own. This is achieved by setting with the macro call `nst_disable_exec_of(guOpndOfGrad)` a *disable*-flag (implemented as a bit in `guOpndOfGrad->property` of `guOpndOfGrad`'s `unitptr`-struct) for `u`'s operand, before `u`'s exec-method returns. However, this then leads to the necessity of resetting this flag with a corresponding `nst_enable_exec_of()` call before we can exec an operand from within the operators exec method. This explains the displayed "bracketing" of any access to the operand(s) of an operator unit with a pair of `nst_en/disable_exec_of()`-macro calls. Similar remarks apply for implementing the `adapt`-method; here we need to invoke a `nst_en/disable_adapt_of()`-pair.

! The macro `nst_enable_exec_and_adapt_of(u)` does a `nst_enable_exec_of()` and a `nst_enable_adapt_of()` in a single step. A similar macro exists for the corresponding disable operation.

! Since the disable-flag disables execution/adaptation of a unit only for a single `exec_unit` or `adapt_unit`-call, an operator must "refresh" the disabling of its operands *at each call*. Thus, even if the operator unit traverses in its `exec` (`adapt`) method a branch where nothing is to be done with the operands, it *still must apply a* `nst_disable_exec_of()` (`nst_disable_adapt_of()`) to all of them! The simplest way to achieve this is a call to the routine `nst_disable_operands_of(u)`, which does the necessary disabling for both the adapt and the exec method for all operands in a single step.

There are some important variations to the above scheme. First, an operator `u` may have more than a single operand. In this case, `u->operands` should contain their number and execution of all operands in their normal NST-order is achieved with

```
...
iRemaining = u->operands;
for (uOpnd=u->next; uOpnd && iRemaining; uOpnd=uOpnd->next, iRemaining--) {
    ...
}
```

Listing 17:

(the `uOpnd!=NULL` requirement safeguards against a premature end of the operator "chain" due to missing units).

! NST allows to define operators whose first operand is at some offset position relative to the operator unit (the storage and obeyance of the offset is in the responsibility of the programmer). When defining such units, they should be flagged by oring their property flag `u->property` with `P_admits_opnd_offset` (see the `ctrl_op`-operator for an example).

Second, while the above example accessed the input and output pins of the operand in a "wireless fashion" directly, a different approach is to exchange the necessary data via NST-connections that are set up between the operator and its operands by the user. This allows to use the same operator with operands that expect their inputs at different input fields and/or provide their results at different outputs. On the programming side, it requires that the operator must be provided with suitable input and output fields for the communication with its operand(s), and that the invocations of the operands and the updating of the operator outputs (for setting the operands inputs) or reading of the operator inputs (to get the operands outputs) are properly synchronized by the operator's exec method.

We illustrate this approach in the context of the second way of implementing the `gradient_op`-unit. Here, we assume that the constructor function has been modified such as to make an operator unit with the following interface description:

`inp0[iDim]` input vector $\vec{x}$ at which to compute gradient `inp1[1]` (single scalar float pin) to receive result value $f(\vec{x})$ from operand

`inp2[1]` step size parameter ϵ for gradient computation

`out0[iDim]` final output vector $\nabla f(\vec{x})$ (=result)

`out1[iDim]` to be used as argument value $\vec{x}$ by operand unit.

For the computation of the gradient we use a simple difference approximation $\nabla f(\vec{x})_i = (f(\vec{x} + \epsilon\hat{e}_i) - f(\vec{x}))/\epsilon$. Thus, we need to make `iDim+1` execution calls of the operand unit; in addition, before each call we must provide a different value at output 1 for the input vector $\vec{x}$ for the operand. This requires to connect the operand unit input to this output field. Likewise, the operand output that holds the result $f(\vec{x})$ must be connected to the extra input field 2 of the operator unit. than field 0. The following listing shows an implementation example for this approach:

```
int exec_gradient_op(unitptr u) {  /* direct implementation */
   unitptr uOperand;
   int i,iDim=ABS(dim_of_Y(u,0));
   float fFunctionValue,fFunctionValue1,fEps,fInvEps,fTemp,*pfX;
   fInvEps  = 1.0/(fEps=fNstInp(u,1,0));
   /* cp desired x to output for operand unit: */
   memmove(pfNstOut(u,1,0),pfNstInp(u,0,0),iDim*sizeof(float));
   if (uOperand = u->next) {
      /* now uOperand = 1st successor unit of u */
      nst_enable_exec_of(uOperand);  /* ensure that exec_unit will work */
      exec_unit(uOperand);
      fFunctionValue = fNstInp(u,2,0);  /* this input receives f(x) */
      /* now compute iDim difference approximations to derivatives: */
      pfX = pfNstOut(u,1,0); /* to speed up multiple accesses */
      for (i=0; i<iDim; i++) {
         /* first replace x by value shifted along axis i: */
         fTemp = pfX[i]; pfX[i] += fEps;
         /* execute operand to compute associated function value: */
         exec_unit(uOperand);
         /* get result from uOperand: */
         fFunctionValue1 = fNstInp(u,2,0);
         pfX[i] = fTemp; /* restore original input */
         /* and compute approximate derivative: */
         pfNstOut(u,0,0)[i] = (fFunctionValue1-fFunctionValue)*fInvEps;
      }
      nst_disable_exec_of(uOperand);  /* protect uOperand from next exec */
   }
   return OK;
}

...

main() {  /* test code, using operand f(x) = |x|, grad f = x/|x| */
   unitptr uGrad, uOpnd;
   init_nst();         /* must precede any other NST call */
   uGrad = gradient_op(2,0.0001,ROOT);  /* make instance */
   uOpnd = euc_len(-2,ROOT);      /* computes f(x)=sqrt(x0*x0 + x1*x1) */
                                  /* iDim = -2 to have NST_FLOAT inp */
   /* connect uOpnd and uGrad properly: */
   joinfields(uGrad,uOpnd,out_1,in_0);  /*    x --> operand */
   joinfields(uOpnd,uGrad,out_0,in_2);  /* f(x) --> operator */
   pfNstInp(uGrad,0,0)[0] = 3;    /* take gradient at */
   pfNstInp(uGrad,0,0)[1] = 4;    /* x = (3,4) */
   exec_unit(uGrad);              /* invoke operator uGrad */
   printf("\n gradient = (%g %g)\n",
               pfNstOut(uGrad,0,0)[0],pfNstOut(uGrad,0,0)[1]);
}
```

Listing 18: Gradient operator, second implementation.

! 1. There is a subtle difference between the `exec_unit(u)` call and the direct call `(*(u->exec))(u)`: only the first form can be disabled; the second call *will always execute*. However, when using `(*(u->exec)(u)` instead of `exec_unit(u)`, note that also variable type conversions are only automatic for `exec_unit(u)` and *not for* `(*(u->exec)(u))`. Similar remarks apply to the `adapt`-pair of routines.

! 2. The save and load methods of an operator unit should save and load state information only for the operator unit; the operands should save/load their state information via their own save and load routines (i.e., there are no `enable/disable`-macros to be used for these methods. The same holds for the ctrl and remove methods.

Whenever a more complex operation is to be implemented, one should always consider whether implementing it in the form of an operator unit could lead to more generality or a simplified, more natural and perhaps more elegant design.

! The operator scheme also allows to work with variable length data, such as linked lists: to this end, an operator iterates its operand(s) a corresponding number of times, providing a new list element at each iteration.

Often, the number of operands can be an additional argument parameter of the creation routine, and a value of 0 for this could be used to indicate a request for the creation of a (perhaps more restricted) non-operator version of the unit.

9 Units for 2D-Graphics

NST facilitates the implementation of new graphics units by providing a number of units (called "window hosts") that offer windows with 2D or 3D scaling and optional coordinate systems. These windows can be accessed by newly written units (that become then "clients" of a window host) and NST provides a simple and uniform scheme for identifying a window host from another unit and for accessing any scaling or zoom information that may be associated with the window host's window(s). In this way, the burden of window management and scaling operations is encapsulated in the window host units and the implementation of a window client becomes simplified.

This scheme offers several advantages: it easy to implement different drawing operations in different drawing clients that can combine their drawing in the same host window. Second, the implementation of new window clients can concentrate on the drawing operations and need not worry about window handling. Third, the modularity of a drawing is reflected in a similar modularity of the window clients that are responsible for these parts. Whenever one or several of these window clients changes, their shared window host becomes notified and in turn can issue suitable redraw requests to the remaining window clients in order to make an updated version the entire drawing. In addition there is also a means to redirect the drawing from a window host to a postscript file.

Since all this is "done behind the scenes", the implementation of a new drawing client NST-unit is very easy: (i) the constructor function needs an additional call to select a window host. (ii) The drawing operations in the `exec` method must be bracketed between `nst_begin_draw()` and `nst_end_draw()` calls (passing a handle acquired in the constructor function when selecting the window host) and use a set of provided GL-style drawing functions. These steps are discussed in more detail below.

A window client selects its window host either by a name (a text string) that must match the title string of the to-be-selected window host (this is the preferred mechanism), or, alternatively, by a relative position offset in the NST-execution order (positive offsets select successor units, negative offsets predecessor units; only window host units are included in the count).

[5]

Since some operations are particularly frequent, NST offers a number of drawing hosts (such as the `write_pix`, `plot_vec`, `plot_xy` and `plot_series` units) that have in addition built in some fixed standard operations, such as drawing a coordinate system and plotting a graph. They can be viewed as drawing hosts that have at least one fixed "implicit" drawing client that is responsible for the standard operations. In addition, the `input_window` and `output_window` units (which have a window that is normally not accessible as a window host for other units) provide a special widget type ("named area", format token `%width:height=name`) to reserve rectangular subregions of their window for use as host windows by other units (including the redirection of the entire graphics output of another window host instead of having a separate window).

! The basis for this is that NST keeps a global list of windows that are currently in use (i.e., that are opened or that are closed, but referenced by some drawing client; closed windows that are not referenced by some drawing client are not held in the window list).

Each list entry (of type `wdw_qentry`) holds information about the window, such as its screen location, its pixel size and any scaling or zoom that is in force. In addition, each list entry maintains also a list of all NST-units that are drawing clients and contribute to the drawing that is visible in the window. List entries are C-structs of type `wdw_qentry_t` and access functions for them are defined in `nst_aux5.c`.

! The NST window queue described above is conceptually different from a second window queue that is used in `nst_X11.c` to keep track of X11-specific information for each window. This latter queue contains only entries for windows that are actually open, while the NST-queue also may contain entries of windows that have been closed but that belong to window hosts that later may reopen them.

To illustrate the implementation of a simple drawing client, let us consider the example of a unit with $iNum$ input fields, each of dimension 2, to draw a polygon of $iNum$ points.

```
typedef struct var_drwpoints {
   nst_winfo_t *pDrawInfo;
   int iNumPts,iCleared /* flag */;
} var_drwpoints_t;

int exec_drwpoints(unitptr u) {
   var_drwpoints_t* pVar=nst_var_of(u);
   if (!CTL_in(u,0)) return OK;
   if (nst_begin_draw(pVar->pDrawInfo)){  float fX,fY; int i;
      for (i=0; i<pVar->iNumPts; i++) {
         nst_dat_to_pix2f(fNstInp(u,i,0),fNstInp(u,i,1),&fX,&fY);
         if (nst_window_is_open()) {
            if (i==0) move2(fX,fY); else draw2(fX,fY);
         }
      }
      nst_end_draw(pVar->pDrawInfo);
      pVar->iCleared=0;  /* see below */
   }
   return OK;
}

unitptr drwpoints(char *pcName, int iNumPts, char *pcFmt, unitptr uHost) {
   int i,aiInp[MAXFIELDS],aiOut[1]={0};
   unitptr uNew;
   var_drwpoints_t *pVar;
   /* set elements of interface description arrays: */
   aiInp[0]=iNumPts+1;  /* nr of input fields */
   for (i=0; i<iNumPts; i++) aiInp[1+i]=2; /* and their dimensions */
   aiInp[iNumPts+1] = 1;  /* the control field */
   if (!(uNew=primitive_unit1(aiInp,aiOut,uHost))) return NULL;
```

[5]under *Neo*, the positional attachment is dynamic: when new units are inserted or deleted, these changes are automatically taken into account

```
   label_unit(uNew,"drwpoints");
   pVar = CALLOC(1,var_drwpoints_t);
   pVar->pDrawInfo = nst_make_winfo(pcName,uNew);
   pVar->pDrawInfo = nst_init_winfo(pcFmt,"asomCFG",pVar->pDrawInfo);
   pVar->iNumPts = iNumPts;
   uNew->var = pVar;
   uNew->exec = exec_drwpoints;
   uNew->ctrl = ctrl_drwpoints;
   uNew->remove = remove_drwpoints;
   CTL_in(uNew,0) = 1;
   return uNew;
}

main() {  unitptr uDraw,uDraw2,uWdwHost; int i,iNumPts=4;
   float afPoints1[]={10,10, 90,10, 50,90, 10,10};
   float afPoints2[]={40,30, 60,30, 60,50, 40,50, 40,30};
   init_nst();         /* must precede any other NST call */
   uWdwHost = write_pix(100,100,"UseThis 100 900","",ROOT);
   uWdwHost = write_pix(100,100,"NotUsed 220 900","",ROOT);
   uDraw = drwpoints("UseThis",iNumPts,"",ROOT);
   uDraw2 = drwpoints("UseThis",iNumPts+1,"",ROOT);
   for (i=0; i<iNumPts; i++) {
       fNstInp(uDraw,i,0) = afPoints1[2*i];
       fNstInp(uDraw,i,1) = afPoints1[2*i+1];
   }
   for (i=0; i<iNumPts+1; i++) {
       fNstInp(uDraw2,i,0) = afPoints2[2*i];
       fNstInp(uDraw2,i,1) = afPoints2[2*i+1];
   }
   exec_unit(ROOT);       /* prints "" */
   nst_XFlush();
   sleep(2);
   remove_unit(uDraw2);
   nst_XFlush();
   sleep(3);
}
```

Listing 19: A class for drawing a polygon into a window

Here, the main new ingredient in the constructor function is the line

```
...
pVar->pDrawInfo = nst_make_winfo(pcName,uNew);
pVar->pDrawInfo = nst_init_winfo(pcFmt,"asomCFG",pVar->pDrawInfo);
...
```

The first call selects the window host unit that shall be used later. Which window host is selected is determined by the string `pcName` (the last argument is always the current window client unit).

If the string can be read as an integer, e.g., `"-2"` or `" 1 "`, the window host will be, e.g., the second predecessor or the first successor unit that has a window (i.e., only the subset of NST units that qualify as a window host are included in the count) [6]

If the string can not be read as an integer, e.g., `"1.2"` or `intensity`, the window host will be that NST unit that has the specified string in its title (NST complains, if no such unit exists; if there are several ones, NST will pick one of them). Again, selection is restricted to the subset of NST units that qualify as a window host.

! We do not discuss the implementation of new window host units here; readers that are interested in this issue might have a look at the source code of the `write_pix`-unit. To test, whether a unit qualifies

[6]currently, window hosts are all `plot_xxxx`-units, the `plot3d`-unit and the `write_pix`-units; The `input_window` and the `output_window`-units are counted only if they contain *named areas* (cf. above)

as a window host or not or to set this property you can use the macros `nst_is_window_host(u)` and `nst_set_window_host(u)` (they access a flag in the `u->property` field).

The handle returned by `make_winfo()` must be kept and is used in the `exec, ctrl etc` methods of the window client.

The second call `nst_init_winfo()` is optional; it initializes flags in `*pDrawInfo` that enable or disable features such as autoscaling, offsets, zooming etc. The necessary informations for these settings are expected in the "format string" `pcFmt`. The expected format and the available options are as described below. The second argument parameter of `nst_init_winfo` specifies which subset of the available options shall be used (option specifications in `pcFmt` with command letters not in the string will be ignored).

The `nst_init_winfo()` call is optional, and handling of options can be done entirely separately. However, this means that, e.g, things such as scaling must be handled by specially written code. Therefore, it is recommended at least to admit the autoscaling option (letter `"a"`). `nst_init_winfo(pcFmt,"a",..)` will allow its activation by a `%a` in the format string. `nst_init_winfo("%a","` will enforce it, independent of `pcFmt`.

Here is a listing of currently available options:

`%a`: automatic scaling (currently for `plot_xy`, `plot_vec`, `plot_series` and `write_pix`-units.). If this option is specified, the routine checks, whether any special coordinate system is defined for the window host and if so, writes the necessary scaling information into `pDrawInfo`. This enables the routines `nst_dat_to_pix()` and `nst_pix_to_dat()` later to do the appropriate conversions. The presence of this option overrides any other scaling option.

`%f1:f2s`: specify that a window pixel has a size of f1 screen pixels along the horizontal direction, and of f2 screen pixels along the vertical direction.

`%f1:f2o` add an offset vector (f1,f2)

`%iC` use initial color i. Default is i=1, i.e. white.

Then, when drawing in the `exec_unit` routine, all that needs to be done is to enclose all drawing operations in a pair of `nst_begin_draw(p)` and `nst_end_draw(p)` calls, where `p` is the handle acquired with `make_winfo()` in the constructor routine. All the enclosed drawing operations will then refer automatically to the selected window host's window. The available drawing routines are a subset of the GL-drawing routines, here we just give a brief enumeration:

! The `nst_begin_draw()` routine makes the window to which the given `pWinfo`-handle refers the current window for drawing and temporarily saves the previous current window on a stack. The return value is the `unitptr` to the window host of the selected window or `NULL` in the case of failure. In this case, the graphics context remains unchanged. The `nst_end_draw()` call restores the previous window context.

```
move2i(),move2f(),draw2i(),draw2f(),rdr2i(),rdr2f(),
cmov2(),cmov2i(),charstr(),clear(),color(),rect(),
rectfi(),circ(),rectwrite(),pnt2()
```

and a few others listed in `nst_X11.h`. (these names can optionally also be prefixed with `nst_`)

A small number of additionally supported functions is

`mark(),circ(),nst_arrowfi()`

Scaling is in (unzoomed) window pixels. To use the scaling that is in effect for selected window host, one must use the functions `nst_dat_to_pix2f()` and `nst_pix_to_dat()` that transform from data to pixel coordinates or the reverse (there is also a `2i`-function for integer return values). If the window host is a `plot3d`-unit, the necessary perspective transformation from data to pixels are `nst_dat_to_pix3f()` and `nst_dat_to_pix3i()` (for `float` or `int` return values, resp).

Note that the drawing commands are executed only if `nst_window_is_open()` is true. This is necessary to avoid undesireable effects when the unit is executed while the window has been closed.

The main() program illustrates the use of the `drwpoints` unit with a little example. It first creates two window units of class `write_pix` before the `drwpoints` unit. They differ in their title string (given in the third argument of the `write_pix`-routine, followed by the coordinates for the lower left window corner), and the first argument of the `drwpoints`-routine specifies that the first window shall be used.

We now turn to the `ctrl`-method of a window client. When the window host receives the signal `NST_W_REDRAW`, it forwards the signal
`NST_W_INTERNAL_REDRAW` to all units that draw into its window (it can do so by inspecting the NST-window queue, which contains the necessary information). Thus, a unit that draws into the window of a window client should in its `ctrl`-method react to the signal `NST_W_INTERNAL_REDRAW` and redraw its graphics (this may require to save drawing data in private variables). Likewise, it should react to the signal `NST_W_CLEAR` by clearing any stored drawing information such that a subsequent redraw produces nothing. Then, it should call the `nst_finish_clear()`-function to issue a `NST_W_REDRAW`-signal for its window host. This then allows to selectively remove its graphics contribution from the entire drawing.

The necessary code in the `ctrl_unit` routine for our example unit would look as given below:

```
int ctrl_drwpoints(int iMode, unitptr u) {
   var_drwpoints_t *pVar = nst_var_of(u);
   switch (iMode) {
      case NST_W_INTERNAL_REDRAW:
         if (!pVar->iCleared) exec_drwpoints(u);
         break;
      case NST_W_CLEAR:
         pVar->iCleared=1;
         nst_finish_clear(pVar->pDrawInfo);
         break;
      /* .. maybe further choices here .. */
      default:  break; /* no op */
   }
   return OK;
}
```

Listing 20: Ctrl method for 2D-graphics

This also shows how a flag `pVar->iClear` is used to diable redraws after a clear. The clear flag is reset at the next `exec_drwpts()`-call (see above).

Finally, the `nst_winfo_t` struct must be discarded in the `remove_unit` routine. Use the `nst_free_winfo()` routine for this:

```
int remove_drwpoints(unitptr u) {
   var_drwpoints_t *pVar = nst_var_of(u);
   ...
   nst_free_winfo(pVar->pWinfo);
   ...
   return nst_remove0(u);
}
```

Listing 21: Remove method for 2D graphics

After the call to `nst_free_winfo()` all calls `ctrl_unit()` for the unit `u` are ineffective. Thus, if any control calls are needed during the remove procedure, either issue them *before* the `nst_free_winfo()`-call, or call the control routine directly, without using the generic `ctrl_unit()`. DON't forget the `nst_free_winfo()`-call! Otherwise you may be bound to see VERY unexpected behavior of subsequent NST calls.

10 Input Devices

For accessing the input devices mouse and keyboard NST provides the function

`nst_locate(int *piX, int *piY, char *iCh, int *piBut, int iMode)` Queries mouse and keyboard device and returns in `*piX...piBut[]` or a subset thereof the read values (`*piX,*piY` horizontal and vertical pixel location of mouse ptr, `iCh` char value of last key pressed, `piBut[0..2]` = status of left, middle or right button). Which of these values are updated with return values is determined by the binary representation `(b6,b5,b4,b3,b2,b1,b0)` of `ABS(iMode)`: `b0` selects the first argument, `b1` the second and so on. E.g., to query only the values for the keyboard and the middle mouse button, `iMode` must be 4+16=20. Positive values of `iMode` make the call blocking, (i.e., the routine waits until a key or button has changed state) negative values lead to a non-blocking call.

If the `nst_locate()`-function is used within a `nst_begin_draw()`-`nst_end_draw()` pair, its return values `*piX,*piY` automatically yield the pixel location relative to the lower left corner (`(0,0)`) of the selected graphics window.

11 Error handling

NST provides the two routines `nst_error()` and `nst_warning()` to issue diagnostic messages. They handle their arguments in same manner as the familiar `printf()`-function: the first argument must be a format string that optionally can contain format tokens that allow the printing of further arguments. The conventions for this format string are same as for the `printf()`-function.

To facilitate the printing of diagnostics, three non-standard format tokens are additionally recognized in the format string:

`%L`: is equivalent to one of the strings `"Warning in line #`*num*` "` or `"Error in line #`*num*` "` (depending on which routine was chosen), where in both cases *num* is replaced by the value of `ERROR_LINE`.

`%F`: is equivalent to the text `"of file `*`file`*`"`, where *file* will be replaced by value of `ERROR_FILE`

`%S`: is equivalent to `%Li%F(in "%s")`, and consumes one string argument for `%s`

For example, the call

```
nst_warning("%S:\nargument %d has illegal value (=%g)!",
            "funnybug",2,fValue);
```

will give rise to a warning text such as

```
Warning in Line 124 of file "TheFile" (routine "funnybug"):
argument 2 has illegal value (=4.56)
```

NST allows to control the amount of printed diagnostics by assigning with each diagnostic message a *print threshold* and comparing it against the value in the global variable `NST_TRACE`. When execution encounters a diagnostic message with a print threshold below `NST_TRACE`, the message will be printed; otherwise, it will be suppressed.

The print thresholds of messages issued via `nst_error()` and `nst_warning()` have the fixed values of -2 and -1, respectively, so that they will always be printed, even when `NST_TRACE` has its default value of zero.

The routine `nst_message(int,char*,..)` is similar to `nst_warning()` and `nst_error()`, but allows to issue diagnostics with an arbitrary print threshold. Its main purpose is to insert additional diagnostics with positive print thresholds that can then selectively be activated to supplement the information given by `nst_error()` and `nst_warning()`.

In many cases, a fixed global value for `NST_TRACE` would be inconvenient. Therefore, each NST-unit can have a *private trace level*. It can be changed with the function call `nst_trace(int iLevel, unitptr u)`. After this call, the variable `NST_TRACE` will temporarily assume the value `iLevel` while any of the methods of unit `u` or one of its subunits executes (the trace value of the subunits can be selectively changed by further calls of `nst_trace`).

! The aforementioned routines supersede the old `error_msg(int,char*,int,char*)`-routine, which offers a somewhat more restricted interface. However, the old routine will be kept for compatibility. Warnings and error messages generated by the old routine will have the same print threshholds of -1 and -2 as if they were generated by the newer `nst_warning()` and `nst_error()` routines and the effect of `NST_TRACE` on their printing will also be the same.

The entire output that is produced by any of the aforementioned routines is not printed directly. Instead, it is passed as a null-terminated string to NST's *error handler*. The default for this function, which is of the type `int(*)(char*)`, is to print to the standard error output. The function `nst_set_error_handler(int(*)(char*))` allows to replace the default error handler by a user-defined routine, which must be passed as a function pointer. This provides full control over the processing of NST's diagnostic output (but NOTE that only the subset of diagnostics selected by `NST_TRACE` are seen!)

The call `nst_set_error_handler(NULL)` restores the default.

A number of further auxiliary functions is useful for the checking of legal arguments:

`is_not_unit(unitptr u)` checks, if `u` points to a legal unit (when NST creates a unit `u`, it sets `u->copy=u` and destroys that relation when the unit is removed later; this is used to distinguish valid from invalid units).

12 Debugging

A `#define NST_DEBUG` at the beginning of a file (before the `#include nst.h`) will activate additional diagnostics, including full runtime checking of argument values whenever one of the `NstInp()/NstOut()` macros is invoked (this will slow down execution and thus should only be used during development). Any range errors then will produce a message that specifies the source line where the problem occurred.

NST uses the global variables `ERROR_LINE` and `ERROR_FILE` to indicate where an error may have occurred. A macro call `error_spotting(Routine)` is provided that wraps routine `Routine` with some extra statements that on each invocation of `Routine` (within the rest of the file below the point of the `error_spotting` macro call) set `ERROR_LINE` and `ERROR_FILE` to the current `__LINE__` and `__FILE__`. Then, when a problem occurs, the NST error output routines (described above) will print these values, thereby allowing to spot the error (there is little use to print `__LINE__` and `__FILE__` themselves, since this would just yield some location within the error printing routine!). The file `nst_errm.h` contains a list of `error_spotting()` calls for the most frequent NST-functions. These are also activated by a `#define NST_DEBUG`.

! `error_spotting(F)` works by replacing `F` with `(ERROR_LINE=__LINE__,ERROR_FILE=__FILE__,F)`. This treats `F` as a function pointer and does the necessary assignments before assuming the value of `F`.

Thus, `F` must not be a macro! Also, the substitution must be carried out non-recursively; the full definition contains an additional `##` for this (however, not all compilers seem to treat this accurately)

Some further functions/settings may be helpful to spot bugs:

describe_unit(u,iLevel) describes interface and connection of a unit `u`. Verbosity increases from `iLevel=1` to `iLevel=3`. The negative values extend the call recursively to subunits.

describe_queue(u) prints information about the subunits of unit `u`.

describe_pin(p) prints information about a pin.

TRACE_ON=1 turns on verbose messages in many (older) units, as well as in the `exec_unit`, `ctrl_unit` and `adap_unit` routines.

dprint(V) prints the value of a integer variable `V` together with line information. Only activated when `#define NST_DEBUG`.

gprint(V) prints the value of a `double` or `float` variable `V` together with line information. Only activated when `#define NST_DEBUG`.

tprint(V) prints the value of a text string variable `V` together with line information. Only activated when `#define NST_DEBUG`.

set_nst_error_handler() allows to pass error messages to a user defined handler (see `nst_dbug.c` for details).

13 Further Example

This section complements the very simple *Hello world* example of the previous sections with an example of a more realistic complexity.

We implement a class to simulate a single "backpropagation-neuron" with `iNum` input values $\vec{x}$, the same number of weight values $\vec{w}$, and one scalar output value y. The to-be implemented transformation equations for this unit are

$$s(\vec{x}) = \sum_{i=0}^{n-1} w_i x_i)$$

$$y(\vec{x}) = f(s(\vec{x}))$$

for the `exec` method and

$$\vec{z} = h(s(\vec{x}))(y^{(target)} - y(\vec{x}))\vec{x}$$

$$w^{(new)} = w^{\vec{(old)}} + \epsilon\vec{z}$$

for the `adapt`-method. Here $f(\cdot)$ is a user-specified "sigmoidal" activation function and $h(\cdot) = f'(\cdot)$ must be its derivative.

A suitable interface for this class can have the following structure:

`inp_0[iNum]` (input field 0) `iNum` scalar values for the input pattern $\vec{x}$.

`inp_1[1]` (input field 1) a single scalar float value for the difference error $y^{(target)} - y(\vec{x})$.

`inp_2[iNum]` (input field 2) a float vector pin for the weights

`inp_3[1]` (input field 2) a single scalar float value for the learning rate ϵ.

`out_0[1]` (output field 0) a single scalar float value for the output $y(\vec{x})$.

`out_1[iNum]` (output field 1) `iNum` scalar elements for the back-propagated error signal $\vec{z}$.

The following call of the `primitive_unit` constructor function creates an empty container unit with such an interface:

```
uNew = primitive_unit(
       NST_INP, NST_SCALAR, iNum, /* for main input */
       NST_INP, NST_SCALAR, 1,    /* for error difference input */
       NST_INP, NST_FLOAT, iNum,  /* for weight vector */
       NST_INP, NST_SCALAR, 1,    /* for learning rate eps */
       NST_OUT, NST_SCALAR, 1,    /* for main output */
       NST_OUT, NST_SCALAR, iNum, /* for backpropagated error  */
       NST_HOST, uHost);
```

Listing 22: Interface specification for a NST-unit

The effect of the creation routine is that memory for all the interface variables has been allocated and can be accessed using the NST functions described in section 4 above.

! There is a special property to the `NST_SCALAR` declaration: if the following number is a positive integer $n > 0$, n scalar float pins will be inserted in the current field. However, for $n < 0$, `NST_SCALAR, n` *becomes equivalent to* `NST_FLOAT, -n`. This facilitates the definition of units that make scalar pins for positive dimension specifications, and "packed fields" (i.e. a single vector pin) for negative dimension specifications.

The following listing adds the necessary lines to implement the exec and adapt methods for the bp unit.

```
#include <stdio.h>
#include "nst.h"

typedef struct var_bpunit{/* struct for private variables of class bpunit */
   float fSum_wx;      /* scalar product s = vec w * vec x */
   float (*f)(float); /* ptr to user-specified activation function */
   float (*h)(float); /* ptr to derivative of activation function */
} var_bpunit_t;

#define weight_of(u)   pfNstInp(u,2,0)

int exec_bpunit(unitptr u) { int i,iDim;
   var_bpunit_t *pVar = nst_var_of(u);
   pVar->fSum_wx = 0;
   if ((iDim=dim_of_X(u,0))>0) for (i=0; i<iDim; i++)
      pVar->fSum_wx += fNstInp(u,0,i)*weight_of(u)[i]; /* s += xi*wi */
   else for (i=0; i<-iDim; i++)
      pVar->fSum_wx += pfNstInp(u,0,0)[i]*weight_of(u)[i];
   fNstOut(u,0,0) = (*(pVar->f))(pVar->fSum_wx);  /* y = f(s) */
   return OK;  /* OK */
}

int adapt_bpunit(unitptr u) { int i,iDim;
   var_bpunit_t *pVar = nst_var_of(u);
   float fDel = (*(pVar->h))(pVar->fSum_wx)*fNstInp(u,1,0); /* h(s)*error */
   float fEps = fNstInp(u,3,0);
   if ((iDim=dim_of_X(u,0))>0) for (i=0; i<iDim; i++) {
      weight_of(u)[i] += fEps*(fNstOut(u,1,i)=fNstInp(u,0,i)*fDel);
   } else {
      float *pfX=pfNstInp(u,0,0), *pfErrOut=pfNstOut(u,1,0);
      for (i=0; i < -iDim; i++)
      weight_of(u)[i] += fEps*(pfNstOut(u,1,0)[i]=pfNstInp(u,0,0)[i]*fDel);
```

```
   }
   return OK;  /* OK */
}

unitptr bpunit(
   int iNum,           /* number of inputs */
   float (*f)(float), /* ptr to activation function to use */
   float (*h)(float), /* ptr to derivative of activation function */
   unitptr uHost      /* host unit of created instance */
) {
   var_bpunit_t *pVar=CALLOC(1,var_bpunit_t);
   unitptr uNew = primitive_unit(..arguments as above..);
   if (!uNew) { free(pVar); return NULL;}  /* safeguard against failure */
   uNew->exec  = exec_bpunit;
   uNew->adapt = adapt_bpunit;
   pVar->f = f; pVar->h = h; /* set user-specified functions */
   nst_var_of(uNew) = pVar;
   fNstInp(uNew,3,0) = 0.5;  /* initialize learning rate to 0.5 */
   label_unit(uNew,"bpunit");
   return uNew;
}

main() { .. (see file bp0.c) .. }
```

Listing 23: Implementation of bpunit

Note that the exec and adapt method contain an `if .. else ..` with separate code for the cases of `iNum>0` (main input and error output consist of fields with scalar float pins) and `iNum<0` (main input and error output consist of float vectors).

We chose function pointers in the private variables struct to access user-defined activation functions in the exec and adapt routines (here, it makes little sense to change these functions during the lifetime of an instance). The exec method computes $f(s)$ and keeps $s = \vec{x} \cdot \vec{w}$ in the state variable `fSum_wx`. Later, in the backward pass, when the adapt method is invoked, it can then evaluate $f'(s)$ without repeating the (time-consuming) computation of $\vec{x} \cdot \vec{w}$.

The adapt method changes the `iNum` weight values and computes a backpropagated error signal that can – via `out_1` – be handed down to the previous `bp_unit`, if there is any. The weights also belong to the state variables. However, here we have chosen not to accomodate them in the `var_bpunit_t`-struct (a perfectly good method), but instead to define the unit with an additional input field that is used as a store for the weights. This has two advantages: first, one can change the weights any time from outside, since they are input variables; second, one need not care about their allocation and deallocation when the unit is created or removed.

! Since the weights sit in an input pin, it seems that they cannot be read from other NST-units (e.g., for the purpose of visualizing them, or computing some statistics). All that needs to be done, however, is to create a void unit with an output pin of matching dimension and type and to connect this pin to the input pin holding the weights. This will make the weights "visible" for other unit in a (readable!) output pin, since both pins of a connected output-input pair reference the same data. For comparison, had we implemented the weights in an output pin, we could have read them directly by other units; however, there would be no way to let other NST-units write into the weights. [7]

! The above scheme offers even more flexibility: it allows several `bp_units` *to share a weight set* by connecting their weight inputs to a common output pin of matching dimension (e.g., belonging to a `void_unit`). This offers a very natural scheme for implementing methods like "unfolding in time", where the weight changes within a chain of networks must be averaged into a weight pattern that is kept the same for all members in the chain.

[7] This is not strictly true; the `reflect_unit` provides a way to achieve this; however, it is subject to restrictions and it was mainly implemented to remedy this limitation for the existing adaptive units, which were implemented in the initial days of NST, using their output pins for holding the weights.

For a really useful adaptive unit, we still need to implement a pair of load and save methods, since we want to be able to preserve the weight values in file, after an adaptive unit has been trained.

! Had we implemented the weights in an input field with pins of the type `NST_SCALAR`, the saving would have been automatic (cf. section **??**). However, usually weights are sufficiently numerous to prefer an implementation as float arrays, which is both more space and also more access-efficient that the `NST_SCALAR` type.

Here is a suitable pair of routines (which would require to set `uNew->save` and `uNew->load` in the constructor function accordingly):

```
int save_bpunit(FILE *fp, unitptr u) {
   int i,iDim = ABS(dim_of_X(u,0));
   int iError=ivwrite(fp,FFMT,u);
   if (iError<0) return iError;
   fprintf(fp,"\n# Weights are:\n");  /* insert a comment */
   for (i=0; i<iDim; i++) fprintf(fp," %g",pfNstInp(u,2,0)[i]);
   fprintf(fp,"\n");
   return OK;
}

int load_bpunit(FILE *fp, unitptr u) {
   int i,iDim = ABS(dim_of_X(u,0));
   int iError=ivread(fp,FFMT,u);
   if (iError<0) return iError;
   skip_comments(fp,'#');  /* to skip inserted comment line(s) */
   for (i=0; i<iDim; i++) {
      if (fscanf(fp,"%g",&(pfNstInp(u,2,0)[i]))!=1) {
         error_msg(0,"load_bpunit",i,"read error for weight %d!");
         break;
      }
   }
   return OK;
}
```

Listing 24: Save and Load methods for `bpunit`-example:

Both routines make use of the NST-functions `ivread()` and `ivwrite()` to read/write information about the unit's interface (`FFMT` is a predefined format constant for this; don't care about it). A negative return value indicates some error and leads to premature termination. The remaining statements insert/skip a comment line into the file, then they write/read the values in(to) input field 2 of the unit. The `error_msg()` call is explained in section 11.

Still, an implementation of a `bpunit` might provide a control routine with mode for, e.g., zeroing or randomizing the unit's weights. Since the implementation of such routine raises no really new issues, we skip it and end this section with a few concluding remarks.

! Regarding the ctrl-method of adaptive units, file `nst0.h` defines some mode constants that should be used within the ctrl method, if possible. The defined mode constants include `NST_RESET` and `NST_INIT`. The semantics of any provided action(s) for these should be to bring the unit back into a state it was in initially. If there are several possibilities, the "strongest" initialization should always be reserved for `NST_INIT`.

14 General design considerations

This section discusses some points that deserve attention when implementing a NST-unit that is meant for serious re-use. Many of these points are not strictly NST-specific, but are of general importance when attempting to increase software re-usability.

Simplicity Try to provide a design that is as simple as possible to use. Provide generality in the form of options that let complexity (such as additional fields) appear only when explicitly requested. Prefer simple field types over fields composed of many different types (this facilitates Neo's task when having to figure out how to join wires when a circuit is remade with new parameters).

Familiarity Prefer structures that are likely to be already familiar. This can simplify both use and documentation! For instance, functions like `NstStrCat`, `NstStrCpy` or units like `switch_op`, `for_loop`, `if_op` that generalize well-known things from C are easier to learn than entirely new combinations. For similar reasons, arguments of constructor functions should preferably be arranged in some regular order: number of operands, dimension specifications, option string, further non-standard arguments, host unit (this should *always* be last).

Modularity Favor as much modularity as possible. If a unit starts to get complex, check if its function cannot be broken up into simpler parts. Exploit the operator concept to "factor out" complexity into operand units while still obtaining generality. At the implementation level, try to implement the necessary functions as a collection of C-routines that then are called and interfaced by NST's method routines (good examples of this are given in Secs. 5 and 8).

Robustness Include checks for illegal arguments. Ensure that illegal arguments do not result in undefined behavior. Often, it is convenient if "special" illegal values (such as `NULL`, the empty string etc.) will signal the choice of particular default settings. For functions that create objects (such as NST constructor functions), use a return value of `NULL` to indicate failure.

Feedback Provide informative error messages and warnings when something goes wrong. Instrument your code with additional diagnostic messages that can be "woken up from silency" by raising a trace level. Prefer the use of `nst_error()`, `nst_warning()` and `nst_message()` functions for this (with the trace level sensititivity already built-in) and avoid direct `printf`'s. This makes it easier to maintain global control of diagnostic messages, e.g., for redirecting or otherwise postprocessing them.

Extensibility Plan for later extensibility without the need to compromise compatibility. A simple but effective measure is to provide a string argument parameter for passing options ("option string"). Even if not required at the outset, this allows to incorporate later extensions without having to revise the parameter list of the function.

! The NST-system routine `nst_options()` can be used to parse an option string that consists of text with interspersed options that are marked with a beginning (`%`)-character. Each such option can be of the form `%a:b..:dX` where `a..d` are up to four numeric option (float or integer) parameters and a final "option character" `X` that identifies what the option is for. The "idiom" for stepping through such command tokens of a format string `pcFmt` when, say, the command letters may be from the set `"abZ$s="`, is:

```
for (pc=nst_options(pcFmt,"abZ$s=",&iArg,&fArg0,&fArg1,&fArg2,&fArg3));
     pc && *pc;
     pc=nst_options(++pc,"abZ$s=",&iArg,&fArg0,&fArg1,&fArg2,&fArg3)){
   switch (pc[0]) {  /* pc[0] is the command char */
     case 'a' :  ...   break;
     ...
     case '=' :  ...   break;
     default:  ... illegal command ...
   }
} /* for */
```

Listing 25: parsing a NST-option string

If less than four arguments are needed, the unused `fArg`'s may be specified as `NULL` pointers. Another useful routine for option parsing is the `ascanf()` routine in `nst0.c`

Naming Well-chosen, informative names can greatly facilitate the use of a system. At the source level, exploit the freedom in the choice of names for bringing code closer to readable documentation, not to optimize typing! A poor choice for a name is difficult to correct later.

At a more formal level, adherence to a naming style may further help to minimize unnecessary complexity. The following is a recommendation that tries to maximize conformance with already existing NST-code: use the "Hungarian" notation with capitalized letters at word boundaries for all variables and for routines for which explicit denotation of return types is of significant help (examples: `iNum`, `pfNstInp(u,f,k)`, `apcArg[]`). Use lower case names with the underscore as word separator for all other objects (thus, mainly routine names, e.g. `void_unit(), init_nst()` (this suggestion is a compromise that tries to maximize the conformance with already existing code).

! Existing NST-code uses the following letters to indicate types via Hungarian notation: `'c' char, 'i' int, 'f' float 'c', 'u' unitptr, 'v' void, 'a' array, 'p' pointer, 's' struct, 't' text string (='pc')`. A `'g'` indicates that a variable is global. Usually, the different types of `int` (long,short,signed,unsigned) and `char` are not further distinguished. Note the difference between array and pointer (an array address refers to automatic or statically allocated storage and should not be used as an lvalue, while a pointer is more general).

The following are a number of more NST-specific, points:

Dimension specifications If a parameter specifies a dimension for a subset of float input or output fields, most NST-units follow the convention that a positive dimension gives rise to field with a corresponding number of scalar float pins, while a negative dimension is interpreted as specifying a field with a single float vector pin with a number of elements given by the absolute value of the specification[8]

By-reference arguments If a constructor function uses by-reference arguments (such as pointers to arrays or character strings), make a private copy of all elements (the `CALLOC(N,T)` macro allocates contiguous memory for `N` elements of type `T`, initialized to zero, and returns a pointer). A constructor function *should never rely on the persistence of any variables outside its own scope*[9] Note that this forbids the use of address pointers to achieve dynamic binding to parameters. For those parameters that need remain changeable after the unit has been created, provide suitable input fields (it is then often convenient to provide argument parameters in the constructor function *for the initialization* of these input fields). If a constructor function needs to reference function parameters or other units, check if the use of pointers can be avoided by implementing the unit as an operator with suitable operands. If a unit needs to access files prefer the use of file names over file pointers unless there is compelling reason to do otherwise.

Beware of static variables Keep in mind that usually several instances of a unit must be able to coexist simultaneously without interfering with each other. Thus, the methods of a unit can not use static variables to conserve state information across calls. Any such information must be kept using private variables, as explained in Sec. 5.

Matrices If a pin represents a rectangular $(i+1) \times (j+1)$ matrix, elements should be arranged row-wise, starting with the first row (i.e., in the sequence $a_{00}, a_{01} \ldots a_{0j}, a10, \ldots aij$).

Image Data If a pin represents a pixel rectangle, pixel should again be arranged row-wise, but this time starting with the lowest row (such that the bottom left corner pixel comes first, then its right neighbor and so on in row-wise order, the upper left corner pixel coming last). That is,

[8] since the access macros `fNstInp/Out` and `pfNstInp/Out` for both cases differ, you must provide separate processing branches in all methods that access these fields in order to obtain this flexibility!

[9] A rare exception is the `array_of`-unit, see its man-page why.

the pixel at image position (x, y) is indexed by $x + (j+1)y$ (remember that $j+1$ was the width of the pixel rectangle). As a result of this convention, displaying a matrix as a NST-pixel image will show the matrix flipped upside down.

Control field At least the more complex units should have as their last input field a scalar "control field" that allows to disable the `exec` and `adapt` operations on the unit when its value is 0. The semantics of other values can be chosen at will; usually a value of 1 should correspond to the "default operation". The routines `nst_def_ctlpin(u,f,i)` and `nst_init_ctlpin(u,f,i)` should be used to tell NST that pin i of input field f is used for this purpose. Usually, f specifies the last field of a unit. The macro `CTL_in(u,i)` yields the value of the `i`-th pin of the last field of a unit (assuming it to be of scalar float type).

Graphics Units that access graphics windows should do so as late as possible, i.e., not at creation time, but at the first exec or ctrl operation. This ensures that the units can be created even when the window does not yet exist (of course, this does not apply to units that create their own window(s)). Also, units that do 2D or 3D drawing should react to the predefined signals `NST_W_CLEAR` and `NST_W_REDRAW` by erasing or redrawing their contribution of the graphics. How to achieve this for the case of 2D-graphics is explained above.

Reset and Init The signal `NST_RESET` should usually reset internal variables to "starting values". A similar effect should have `NST_INIT`, however, if there are different choices, `INIT` should be "stronger" than `RESET`. E.g., for a unit that offers the function of a buffer, `RESET` might reset access to the start of the buffer, and `INIT` might in addition also erase the contents of the buffer.

File-i/o Preferably use `nst_fopen()` and `nst_fclose()`. They work like the familiar `fopen()/fclose()`, but `nst_fopen()` will (in read mode) in addition also try to find the file in a sequence of other directories that may be specified in the Unix environment variable `NSTPATH`. The `nst_fopen0()` is similar, but allows additional use of `gzip`'ed or `compressed` version of the specified file (with automatic decompression via a pipe; however, this makes backward file pointer movements impossible!).

NST interface Don't *write* into *input* pins of a unit when the pin is connected to the outputs of another unit: this will overwrite the other unit's current output values and will thus have the severe side effect to change its outputs *also for any other reader!* (there are a few, very special cases when this effect may be desired; one example is the sharing of weight adjustments among a set of adaptable units, cf. the discussion on p.**??**). However, writing into inputs is safe within the construction function of a unit: at this time, the inputs are still unconnected.

! The NST function call `w_of_p(p)` returns the address of any writer pin of a pin `p`, or `NULL`, if the pin is unconnected. It may be used to inquire if writing into an input is safe.

Of course, writing into output pins of a unit is always ok. However, for text strings, you must use the `NstOutStrCpy` and `NstOutStrCat` functions (necessary to inform other units about possible memory reallocations due to string lenght changes). In general, don't redirect any pointers in the NST-interface "by hand" unless you know exactly what you are doing!

Optimization for units for general use, it is desireable to take some optimization measures into account. These are explained more fully in the next section; they are particularly recommendable for image processing units.

15 Performance Issues

NST was written such as to make the execution and adaptation of units fast. In particular, data between connected units are not copied in most cases; instead access is by pointer reference.

The price for this has been increased complexity in the `join()` and `joinfields()` routines that set up connections between units. Thus, dynamical "rewiring" of NST-units is a comparably costly operation (also since extensive error checking is involved). Still more expensive is the creation of entire new units.

! Copying of data, however, occurs if a connection runs between fields of different type. If an output field of type `NST_BYTE` is connected to two input fields of type `NST_FLOAT`, there will be *one* copy and conversion operation into a "conversion memory area" from which the *NST_FLOAT* readers can access the converted data by reference. The necessary conversion memory is automatically allocated whenever fields of different type are connected; it is freed when the last such connection is cut. This all is fully automatic; the only visible effect is that fields of different type can be simply connected. For details, inspect `nst8.c`.

15.1 Optimization of data access

As regards performance, the vector types `NST_FLOAT`, `NST_INT` and `NST_BYTE` allow faster access, since here all elements are contiguous in memory. The `NST_SCALAR` type allows arbitrarily "scattered" elements; this gives utmost flexibility in connecting pins of the same field to widely different places but comes at the price that each element must be accessed separately and room for optimization via pointer arithmetics is more limited.

The straighforward way to access the elements of a vector pin at position 0 in field `f` of a unit `u` would be: Instead of a loop

```
 for (i=0; i<ABS(dim_of_Y(u,f)); i++) {
    pfNstOut(u,f,0)[i] = ...
}
```

Listing 26: A suboptimal loop

(note that we here assume a "packed fieldd" that has only a single vector pin at position 0, and that the `dim_of_Y()` macro returns the negative number of elements in this case) we better factor out the repeated operations and use the more efficient

```
pfStart = pfNstOut(u,f,0);
pfEnd   = pfStart + ABS(dim_of_Y(u,f));
for (pf=pfStart; pf<pfEnd; pf++) {
    *pf = ...
}
```

Listing 27: Optimized version of loop

A similar remark applies to input fields. Even for a `NST_SCALAR` field, some optimization is possible: instead of

```
 for (i=0; i<dim_of_Y(u,f); i++) {
    fNstOut(u,f,i) = ...
}
```

Listing 28: Unoptimized loop with scalar float field

use

```
pin **pp,**ppStart,**ppEnd;
...
ppStart = &pNstOut(u,f,0);
ppEnd   = ppStart + dim_of_Y(u,f);
for (pp=ppStart; pp<ppEnd; pp++) {
    /* the above yields *pp = pNstOut(u,f,0..dim-1) */
    *pp->s[0] = ...
}
```

Listing 29: Optimized loop for scalar float field.

(this makes use of the fact that the float value that is associated with a pointer `p` to a scalar float pin is given by `p->s[0]`).

15.2 Optimization of pixel operations

Operations on image data can often be speeded up drastically when it is possible to restrict the computation to a subregion of the image. NST offers a simple scheme to propagate the specification of a bounding rectangle along a chain (or tree) of image processing units. The underlying idea is that the first unit in the chain, which acts as the source of the image data, compares successive image frames to update the size and location of a bounding rectangle that encloses the area where pixels have changed. Then, if this bounding rectangle is passed down to the other units along the chain, each of them can restrict its processing operations to the enclosed subregion. Of course, those units that perform non-local operations (such as, e.g., convolutions), may have to change the bounding rectangle accordingly, before they pass it on to their successors.

In order to make this scheme convenient, NST allows to store the necessary information about a bounding rectangle in the fields that hold also the image data. In this way, passing of image data is automatically accompanied by the passing of four additional numbers describing a bounding rectangle (these data are stored separately from the array holding the image data and can be ignored, if no such optimization is desired).

To make a unit participate in this optimization scheme requires to initialize in the constructor function the bounding rectangle mechanism for those input and output fields of the unit that shall use it. Then, the exec-method must read the bounding rectangle(s) data from the appropriate input fields, restrict its pixel operations accordingly, and write (a possibly modified bounding rectangle) specification into the appropriate output fields.

Of course, the benefit of this method will be restricted to the initial portion of the processing chain that consistes entirely of units that use this scheme. If the passing on of the subrectangle is broken at some point, subsequent units cannot benefit any further (the default area of the bounding rectangle is the entire image). However, the unit `subrect_of_change` can be inserted in the processing chain to recompute a bounding rectangle, if it has been lost or if the image source does'nt provide one.

This may sound somewhat involved, but it is actually rather simple and can lead to drastic speed-ups, particularly for image sequences. We illustrate the procedure with the simple example of a unit that thresholds the intensity values of an image. The image input (output) is at input (output) field 0, the threshold value is at input field 1.

Here is the “plain version”,

```
int exec_thr_imag(unitptr u){
   float *pfPixInp = pfNstInp(u,0,0);
   float *pfPixOut = pfNstOut(u,0,0);
   float *pfEnd    = pfPixInp + nst_dim_of_p(pNstInp(u,0,0));
   float fThreshold = fNstInp(u,1,0);
   for (; pfPixInp < pfEnd; pfPixInp++)
      *pfPixOut++ = (*pfPixInp > fThreshold)
                   ?  *pfPixInp
                   : 0;
   return OK;
}

unitptr thr_imag(int iDx, int iDy, /* image size */
                 unitptr uHost) {
unitptr uNew = primitive_unit(
                   NST_INP, NST_FLOAT, iDx*iDy,
```

```
                  NST_INP, NST_SCALAR, 1,
                  NST_OUT, NST_FLOAT, iDx*iDy,
                  NST_HOST, uHost);
if (!uNew) {
   nst_error("%S: failed to create unit!\n","thr_imag");
   return NULL;
}
uNew->exec = exec_thr_imag;
label_unit(uNew,"thr_imag");
return uNew;
}
```

Listing 30: Thresholding of an image

Note that we have copied the threshold value and pointers to the start of the input and output image data into separate variables to avoid the need of a separate call of the `NstInp/Out` macros for each pixel, since each such call would involve traversal of a pointer chain of length 3.

A version that uses the bounding rectangle mechanism looks as follows:

```
int exec_thr_imag(unitptr u){
   float *pfPixInp = pfNstInp(u,0,0);
   float *pfPixOut = pfNstOut(u,0,0);
   float *pfEnd;
   float fThreshold = fNstInp(u,1,0);
   int iXll,iYll,iDx,iDy,iSubDx,iSubDy,iY,iStride;
   /* retrievel lower left corner (iXll,iYll) and
      size iSubDx, iSubDy of bounding rectangle
      associated with input X_in:
   */
   nst_get_roi(&iXll, &iYll, &iSubDx, &iSubDy,
                 &iDx,  &iDy,  u->in[0]);
   /* restrict threshold operation to subrectangle:
      (now more convenient with a double loop):  */
   pfPixInp   += iXll + iYll*iDx; /* offset to lower left */
   pfPixOut   += iXll + iYll*iDx; /* corner of rectangle  */
   iStride    = iDx - iSubDx;     /* nr of pixels from right
              margin of current line to left margin of next */
   for (iY = iYll; iY < iYll+iSubDy; iY++) {
      pfEnd = pfPixInp + iSubDx;
      for (; pfPixInp < pfEnd; pfPixInp++) {
         *pfPixOut++ = (*pfPixInp > fThreshold)
                      ?  *pfPixInp
                      : 0;
      }
      pfPixInp += iStride;
   }
   /* pass rectangle info on to output out_0
      [here: same size as at input] */
   nst_set_roi(iXll,iYll,iSubDx,iSubDy,u->out[0]);
   return OK;
}

unitptr thr_imag(int iDx, int iDy, /* image size */
                 unitptr uHost) {
unitptr uNew = primitive_unit(
                  NST_INP, NST_FLOAT, iDx*iDy,
                  NST_INP, NST_SCALAR, 1,
                  NST_OUT, NST_FLOAT, iDx*iDy,
                  NST_HOST, uHost);
if (!uNew) {
   nst_error("%S: failed to create unit!\n","thr_imag");
   return NULL;
}
```

```
uNew->exec = exec_thr_imag;
nst_init_roi(iDx,iDy,uNew->in[0]);
nst_init_roi(iDx,iDy,uNew->out[0]);
label_unit(uNew,"thr_imag");
return uNew;
}
```

Listing 31: Thresholding of an image, optimized with bounding rectangle

Most things will be evident from the comments. The `iStride` value results from the linear layout of images as `iDy` rows of length `iDx` each. Thus, the increment between identical horizontal positions of successive rows is `iDx`, and since `iSubDx` pixels of the preceding row have already been processed, the required increment is `iDx-iSubDx`. The required image size parameters `iDx` and `iDy` are for convenience returned with the `get_restriction` call. The exec method will update its output only within the area of the bounding rectangle. This is perfectly ok, since initially the bounding rectangle was the entire image, and after that only the changed parts need updating.

The demo program `pix1.c` demonstrates the dramatic effect of this optimization: its `main()` program prepares a simple test image which is then displayed 200 times, each time with only a single pixel changed. If `pix1` is called without any command line parameter, the optimization is disabled (the above `thr_imag` unit gets no `roi` information between calls). As a result, the tiny change invokes a recomputation of the entire thresholded image and a redisplay of the entire image region by the attached `write_pix()` unit, clearly a very inadequate solution for the present case.

If instead some command line parameter (value does not matter) is given, an additional call of `nst_set_roi()` is activated (this would normally be carried out automatically by the predecessor unit feeding the `thr_imag` unit) and informs `thr_imag` about a (here 1x1!) bounding rectangle of the changed pixel. This allows the unit to restrict its internal computations to the tiny 1x1 pixel region and to pass this information further on to the attached `write_pix()` unit. The `write_pix()`-unit has also been implemented with the present scheme; thus it will now restrict its screen update to the changed pixel region only, which is, of course, much faster for the single pixel than for the entire image before.

! A current limitation is that there is only a single roi per image now. A future extension in this regard will probably have an internal list of roi's which can be traversed with multiple calls to `nst_get_roi()` unit a return value of 0 indicates that the list is exhausted. Likewise, the list passed to the successor will be filled with multiple `nst_put_roi()` calls. Thus, use of the extended scheme will only require to replace the current

```
nst_get_roi(..);
.. do operations on roi ..
nst_set_roi(..);
```

by

```
while (nst_get_roi(..)) {
  .. do operations on roi ..
  nst_put_roi();
}
```

Thus, code written in the indicated way is prepared to take advantage of significant further optimization with only very little modification.

! A similar chaining scheme at the level of fields/pins is a likely candidate to integrate list and other data structure processing into a future NST-version in such a way that existing code will remain fully compatible and can be made to benefit from the extended capabilities with very little modification.

16 Wrapping units for use with *Neo*

This subject is covered in detail in the Neo-Tutorial. The `mku`-utility generates a skeleton of the required Neo wrapper code besides the code skeleton for the unit itself. For convenience, here we just summarize the main steps of how to proceed:

1. Invoke `mku` to generate template programs `myunit.c` and `neo_myunit.c` (`mku myunit`).
2. Flesh out the template program `myunit.c` to contain the necessary code for the to-be-implemented unit (as described in this manual)
3. Flesh out the template `neo_myunit.c` to contain the necessary wrapper code for *Neo*. This mainly consists in defining the layout of one or two interface windows and substituting their widget values into a call of the NST-creation routine of the new unit (explained in the *Neo*Tutorial).
4. Edit the file `neo_user.c` to specify a folder into which the new unit goes (requires inserting a line `ADDICON(myunit,NameOfFolderWhereToGo)`).
5. Add `myunit.o` and `neo_myunit.o` to the objects to be linked into *Neo*(e.g., add `myunit.o` to `NSTALPHA` and `neo_myunit.o` to `NEOALPHA`)
6. Remove the old `neo_user.o` and make a new version of *Neo*with `make neo`

17 Compilation

[this section needs update] To compile a NST source file (its name being, say `myunit.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, SOLARIS` or `intel` (the correct value is in the variable `$ARCH`). In addition, there are different graphics options, requiring the inclusion of additional libs. Below is a table of the neccessary combinations for each architecture. To get along with the linewidth, we use the abbreviation (`$(SAME)` for `-o myunit myunit.c -I/vol/nst/include -L/vol/nst/lib -lnst`, which is the same for all architectures.

architecture	how to compile
Sgi iris4d	`cc -Diris4d $(SAME) -s -lgl_s -lfm_s -lm -lX11_s`
Sgi irix5 with X11	`cc -Dirix5 -DNST_X11 $(SAME) -lX11 -lm`
Sgi irix5 X11 & OpenGL	`cc -Dirix5 -DOGL -DNST_X11` $(SAME) -lGL -lGLU -lX11 -lm
Sun-OS	`gcc -Dsun4 -DNST_X11 $(SAME) -lX11 -lm`
Sun-Solaris	`cc -Xc -xO2 -DSOLARIS -DNST_X11 -I/usr/openwin/include` $(SAME) -lnsl -lX11 -lm
Sun-Solaris X11 & OpenGL	`cc -Xc -xO2 -DOGL -DNST_X11 -I/usr/openwin/include` -DSOLARIS $(SAME) -lGL -lGLU - ldga -lX11 -lXmu -lnsl -lm
Ibm AIX	`cc -Drs6000 $(SAME) -lgl -lm -lbsd`
Ibm AIX	`cc -Drs6000 $(SAME) -lgl -lm -lbsd`
PC Linux	`gcc -DINTEL -DNST_X11 $(SAME) -lX11 -lm`

To use compile a `SCOTT` interface into NST, add a `-DSCOTT` switch.

There is now a unit that supports parallel (threaded) execution of a number of operand units. If this feature is to be used, add a `-DHAS_THREADS` switch (requires to add `-lthread` on Solaris machine and similar libs on other machines; only tested on Solaris).

18 Additional Technical Information

The following is a "dump" of some technical information that awaits a more didactic documentation. It is not strictly needed for implementing NST-units; however, it may occasionally be a useful reference.

18.1 NST-Kernel

The NST kernel is organized into nine functional "layers":

nst0.c	general routines
nst1.c	pins
nst2.c	fields
nst3.c	units
nst4.c	connections
nst5.c	methods
nst6.c	variables
nst7.c	file-i/o
nst8.c	type conversions
nst.c	master file, includes the above

NST Kernel routines

18.2 The `field` Data type

NST creates for each instance of a unit a structure of type `unittype` that collects all important information about the instance. The constructor routines `mk_u, void_unit` and `primitive_unit` are located in `nst3.c` and create raw (`mk_()`) or initialized instances of the `unittype`. Fields `ninputs, noutputs` of this struct contain the number of input and output fields of the associated unit.

The fields themselves are implemented as instances of the NST data type `field`. The instances of this data type that belong to a unit are referenced via the pointer arrays `u->in[]` and `u->out[]`. `u->in[i]` (`u->out[i]`) is a pointer to input (output) field `i` (`i=0...ninputs-1`). For each field (input or output) a call of the constructor function `mk_f` (in `nst2.c`) will allocate an instance of the following `struct`:

```
typedef struct def_field{
   pin   **p;   /* ptr to array of pointers to pin */
   int npins;   /* number of elements in array p[] */
   int dim;     /* nr of pins (negative for packed field) */
   int id;      /* positional number of this field in in[]/out[] array */
   unsigned int property;  /* flags about field properties */
   rectangle_t r;  /* optional rectangle info */
   void *host;  /* pointer to unittype struct of which this field is member */
} field;
```

Listing 32: NST field struct

The fields `npins, dim, id, property` and `host` contain information that is shared by all pins of this field. The number of pins for the field is given by `ABS(dim)`. If `dim>0`, the field is a normal field, if `dim<0` it is a "packed field". Such fields are characterized by having a single pin `npins=1` of one of the types `NST_FLOAT`, `NST_INT` or `NST_BYTE` of `-dim` elements[10]

[10]this convention is kept from the days when there was only a choice between fields with several `NST_SCALAR` pins or a single `NST_FLOAT` pin. Then, `dim` was used to specify the number of values, and the sign specified which of the two alternatives was true. Meanwhile, there are more possibilities, but the concept of "packed fields" (that now can also have `short int` or `byte` elements) still is handy in many situations.

18.3 The `pin` Data type

The pins pins themselves are accessible through the `npins` elements of the pointer array `f->p[]` of a `field`.

För an unpacked field there is always `npins = dim`, i.e., each pin `j` has a separate pointer `p[j]` to an instance of a `struct` of type `pin`, defined as follows:

```
typedef struct defpin{
   /* struct for basic pin type: */
   struct defpin *r;  /* ptr to first reader of this pin; NULL if none */
   struct defpin *w;  /* if t & NST_PINSTATUS == 0:
                           ptr to writer of this pin;
                           NULL if none;
                           VPIN, if pin this pin is bound to a C-variable
                         if t & NST_PINSTATUS == 1:
                         ptr to next pin of this pin's readergroup.
                       */
   char t;             /* flag for interpretation of w-field [see above] */
   float *s;           /* ptr to value of this pin */
   pin_extension *ext; /* ptr to additional info for pins other than
                          NST_SCALAR */
} pin;
```

Listing 33: NST pin struct

An instance of this type is created by the pin constructor function `mk_p()` in `nst1.c`.

For pins of type `NST_SCALAR`, `*s` represents the (single) value that is associated with the pin, and `ext` points to the `field` of which this pin is a member. For all other pin types, `s` is the starting address of the memory region in which the elements of the pin are held as a contiguous array of `float`s, `short int`s, `unsigned char`s or `char`s, depending on the type of the pin, and `ext` points to a substructure with additional information about the pin:

```
  typedef struct def_pin_extension{
        /* struct for extension of pin to vector or port type: */
        void *host;  /* ptr to field of which this pin is a member */
        int  iDim;       /* nr of vector elements */
        char native;     /* native type of pin, one of "ifcbt" */
        /* pointers to converted versions of data vector:      */
        float *pf; /* result of conversion native->float       */
        NstInt_t  *pi; /* result of conversion native->int     */
        char  *pc; /* result of conversion native->unsigned char  */
        void  *ps; /* ... */
        /* ptrs to conversion functions for conversions native->xxx: */
        conv_funct_t pFf,  /* converts native->float         */
                     pFi,  /* converts native->int           */
                     pFc;  /* converts native->unsigned char */
   } pin_extension;
```

Listing 34: NST pin_extension structure

Here, `native` specifies the type of the pin, `iDim` the number of data elements (float, short integers or bytes/chars) and `host` now points to the field of which the pin is a member (since the `ext` field is used otherwise).

The pointers `pf,pi,pc` and `ps` are used when the pin is read by pins of a different data type. This will cause NST to allocate additional memory regions to hold properly converted copies of the pin's "native" data that are always starting at address `p->s`. These conversion memory regions can then be accessed via the pointers `pf,pi,pc` and `ps`. More details on this mechanism are given below.

18.4 NST type conversion mechanism

The remaining pointers are used to implement NST's automatic type conversion mechanism: essentally, conversion is installed for a pin `pPin` (of "native" data type t1) whenever it becomes for the first time joined to a reader of a different native data type t2. When such type difference is noticed by NST's kernel `join_pq()` routine, it issues a call `install_type(cNativeTypeOfReader, pPin)`

This has the effect to

1. allocate for the pin `pPin` a block of "conversion memory" of sufficient size to accommodate a copy of its "native" data, converted into the data type requested by the new reader
2. set the corresponding pointer `pf,pi,pc` or `ps` to the newly allocated conversion memory
3. set one of the pointers `pFi`, `pFc` or `pFs` to a suitable conversion routine of type `void(*)(ToType*,FromTyp`
4. set a bit flag in the host unit of `pPin` to indicate to a calling `exec_unit`, `adapt_unit`, `ctrl_unit load_unit` or `save_unit` routine, that some pins of the unit require their conversion routine being called.

The `exec_unit` etc. routines will then make for each input pin of their argument unit the proper conversion calls before executing the body of the unit; before returning to the caller, any conversion routines referenced by the output pins are called. To this end, `exec_unit` etc. calls the routine `do_type_conversions()`, which in turn accesses the necessary conversion functions for a (sub)set of result types for a pin. In this way, all inputs/outputs of the unit are always up-to date in all required type formats, and only one conversion call is spent per exec of a unit.

Installation of a conversion routine and of a conversion memory block may happen separately for each new type that can be requested by a reader of the pin. I.e., each pin can have up to (currently 3) conversion pointer pairs (one to pointer to conversion memory, one pointer to the conversion function) installed.

Likewise, if the last reader of a pin requesting a to-be converted type gets unconnected (again as a result of a join or unjoin), the conversion memory block for this type is freed and the conversion ptr reset to NULL (this is achieved byt the routine `uninstall_type(cTypeToUninstall, pPin)`. Some additional complications arise for connections between fields of the type `NST_SCALAR` and one of the vector types. Their handling is described more fully in `nst8.c`.

18.5 Expandable Pins and Fields

Usually, fields and pins must have fixed dimensions. There is an exception to this: A container unit may have fields that contain *expandable pins*. The dimension of these pins is initially zero (1 for pins of type `NST_SCALAR`) and it is adjusted suitably when connections with fields/pins of subunits of the container unit are made. It is necessary to distinguish 3 types of expandable pins: 1. expandable scalar pins, s0 [=actually expandable scalar fields] 2. expandable vector pins, 3. expandable text pins. Vector pins allow only a single expansion step, after which the dimension is fixed. Scalar fields allow multipile steps. Text pins are by definition always expandable. The precise rules are stated below:

18.5.1 Expandable Text Pins

Since text pins are always dynamically resizeable, expandable text pins are not special in any way; they behave as all other text pins do.

18.5.2 Expandable Vector Pins

An expandable vector pin `p` of a unit `u` is obtained by specifying its type, but with a dimension of zero. It becomes expanded to a nonzero dimension d when it is connected to another vector pin `q` *and*:

a) `q` is a vector pin and has a nonzero dimension d_1;

or

b) `q` is a pin at position 0 of a purely scalar field of d_1 scalar pins.

In both cases, the dimension d is changed to d_1 (and since $d_1 > 0$, `p` looses its expandability)

18.5.3 Expandable Scalar Fields

As long as a field `pF` of a unit `u` contains *only* scalar pins, and is not yet "frozen", it is expandable in the following way: (let d be its current dimension):

1. the first connection operation from a unit that is not a subunit of u will freeze the current dimension d of `pF`.
2. the first connection from a vector pin `q` of dimension $d_1 \geq d$ will freeze the dimension of `pF` to d_1.
   (a) If `q` is `NST_FLOAT` and is the first connection, the s-type will be changed into `NST_FLOAT` of dim d_1.
   (b) Otherwise, it will be changed into `NST_SCALAR` of dim d_1 and a scalar-vector connection is formed.
3. any request to connect a pin with dimension $d_1 > d$ to some scalar pin will increase d to d_1

Expandable pins/fields have been introduced to facilitate for *Neo*the task of code generation for constructor functions: *Neo*can just insert code that makes containers with suitable expandable fields and the actual dimensions will then be adjusted automatically when the subunits are made.

18.6 NST connection handling

When NST connects an output pin `p` of a unit P to an input pin `q` of a unit Q, it just propagates the data address held the source pin `p` to the data address pointer of the destination pin `q`.

Additionally, each pin keeps a list of its current readers, thus `p` adds the new reader `q` to its list, and `q` 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 `q` uses (and recursively to the readers readers etc.). In addition, `q` frees the memory for its now no longer needed previous value, unless this pertained already to an output pin of another unit (if `q` somewhen gets totally disconnected again, it allocates a private variable again). The necessary pointering 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.

Here are some more details of these steps. Information about the connections of a pin `p` with other pins is held in element `w` ("writer to this pin") of the pin-struct, and in a linked list of readers ("readerlist"). The address of the first element of this list is given by `p->r` (`NULL`, if the pin has no reader). To save memory, `w` is multiply used, with a flag `t` denoting which interpretation is valid.

Each pin has its value either from another pin (its *writer*, case 1) below), or oder is "bound" to a C-variable (case 2). The third possibility is the absence of any connected external sources, in which case the pin holds its value in a private memory location. In all cases, the pointer `p->s` specifies the address of the value.

A pin `p` can have zero or more other pins as its readers. In this case, `p` is the "writer" for each of its readers. All readers of a pin `p` form the "readergroup" of this pin. Consequently, all pins within the same readergroup share a common writer pin, and each pin that has a writer, is member of exactly one readergroup (note the difference between the readergroup of a pin `p` [i.e., the set of readers of `p`], and the readergroup *to which a pin belongs* [i.e., the set of pins that share with `p` the same writer!) Each readergroup is implemented as singly linked list. Elements in this list are all pins, that share the same writer `p`. The first element `q` in this list is at the address `p->r = &q` of the shared writer `p`. `q->t & NSTPINSTATUS` indicates that `q` has a successor, and `q->w` is its address. If `q->t & NSTPINSTATUS` is false, `q` is the last element in the list and `p->w` is pointer to the shared writer of the readers in the list.

The other cases 2 and 3 are indicated by the special values `p->w=VPIN` and `p->w=NULL`; in these cases the `p->t` flag is not used.

Connecting of two pins `p` ("source") and `q` ("destination") involves insertion of `q` into the group of readers of `p`. If `q` is already member in some readergroup, it first must be removed from the corresponding list. After this, `q` inherits for its new source `p` the pointer `p->s` that specifies the address of the associated value. If `q` itself has readers, these recursively update their address pointers. In this way, even readers many stations "downstream" have access to the correct input value only "one pointer reference away".

These steps, plus additional steps to assure setting up/releasing any type conversion resources, are carried out by the routines `del_from_rdrgrp`, `join_pq` and `set_adr` in `nst1.c`. They form the basis of all connection handling in NST and are the core of the user-level `join()` and `join-fields()` routines in `nst4.c`, which include additional diagnostics.

19 Common Pitfalls

[this section needs work] Most errors result from incorrect field/pin specifications. The value in element i of a vector pin 0 of field f is `pfNstOut(u,f,0)[i]` *and not* `pfNstOut(u,f,i)`! Compiling with `#define NST_DEBUG` usually helps to many errors easily. Currently, most NST low level functions do no error checking, since they are called frequently and should be fast. Additional compilation of error checking when `NST_DEBUG` is defined should be built in somewhen.

A Demo Sources

The sources of the demo examples discussed in this manual are located in subdirectory `demosrc` of the `nst` directory. Here is a list:

hello0	listing on page 3
hello1	listing on page 3
hello2	listing on page 3
hello3	listing on page 4
hello4	listing on page 5
grad0	listing on page 8
grad1	listing on page 8
numr0	listing on page 5
bp0	listing on page 13
pix0	listing sec 15.2
pix1	listing sec 15.2

Each demo example `name.c` comes with a make file `name.mak` and can be compiled with

```
make -f name.mak
```

The associated makefile `hello0.mak` for making `hello0` from `hello.c` looks like

```
NAME = hello0

ARCH    = linux
NSTDIR  = ..
FLAGS   = -O -D$(ARCH) -DNST_X11
CFLAGS  = -I$(INC) $(FLAGS)
VPATH   = ../nstsrc:../foldersrc
MESA    = -L/usr/lib -lMesaGL -lMesaGLU
SHELL   = /bin/sh
INC = $(NSTDIR)/nstsrc
LIB = $(NSTDIR)/o.$(ARCH) $(MESA) -lnst -lX11 -lXext -lm

$(NAME): $(NAME).c  # require getopt - code for .o
        $(CC) $(CFLAGS) -o $(NAME) $(NAME).c -L$(LIB)

$(NAME).o: $(NAME).c  # require getopt - code for .o
```

Listing 35: Makefile

and needs to be edited if the architecture differs from Intel (set `ARCH` to one of `SOLARIS, irix5,rs6000,sgi,su`
In this case, also additional include files or libraries may be needed. NOTE: no space behind the name (here `hello0`) is permitted!

B Further Information

Postscript versions of some useful documents are in `/vol/ni/man/ps`: `NstPrimer.ps` (=this document), `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`.

C Files

Here is a synopsis of the core NST-files. Those marked with an asterisk are not compiled into `libnst.a` but instead into a separate shared library (together with a wrapper file `neo_name.c`) for use under `Neo`. Using units from these files requires to link the respective objects extra. The source files for `libnst.a` are located in directory `nstsrc`, the remaining files are in directory `foldersrc` (both directories are located in the `NST` root directory).

General NST-units

nst_dbug.c	routines for tracing, debugging and error printing
nst_meta.c	NST `remake_unit`
nst_stdr.c	standard units for control flow (iterators, if-op, switch)
nst_util.c*	various further operator units
nst_prog.c	tiny-C interpreter unit
nst_prog_aux.c	auxiliary routines for the above
nst_strg.c	string handling routines and units
nst_thread.c	operator for parallel execution of NST units
nst_var.c	variable handling

NST-display and GUI units

nst_asci.c	display units for text terminal
nst_contour.c	2D-contour plots
nst_draw.c	2D-drawing routines
nst_iso.c	3D isosurfaces
nst_plot.c	mouse and plot units
nst_scrn.c	NST input and output window units
nst_solid.c*	units for 3D rendering under GL
nst_solid2.c*	more units for 3D rendering under GL
nst_pixel.c*	image manipulation and display
nst_X11.c	X11-graphics interface

Neural Network units

nst_adaptive.c*	adaptive units
nst_neuron.c*	units to build neural networks
nst_psom.c	parametrized self-organizing map unit
nst_psom_aux1,2.c	auxiliary routines for the above
nst_som.c	NST units for self-organizing map

Math units

nst_cjgr.c	conjugate gradient minimizer unit
nst_fft.c	1d-fast Fourier transform
nst_geo.c	geometric transforms for robotics
nst_filter.c*	various image filter units (gabor and other)
nst_funct.c*	mathematical functions I
nst_imk.c	interface to "image awk" `imk`
nst_list.c*	example implementation of user-defined list pin data type and some list ops
nst_math.c	mathematical functions II
nst_math8.c	dito, but subset for byte data type
nst_math16.c	dito, but subset for short int data type
nst_matrix.c*	linear algebra units
nst_matrix_algebra.c	auxiliary routines for the above
nst_matrix_aux.c	auxiliary routines for the above
nst_nr.c	(no units) frequently used numerical routines
nst_numr.c	routines from Numerical Recipes [to be phased out]
nst_numer.c	other routines from Numerical Recipes [to be phased out]
nst_robot.c*	robot related units
nst_vector.c*	2D Voronoi diagrams
nst_voro.c	2D Voronoi diagrams

Various interface units

nst_dcube.c*	interface to data-cube
nst_file.c*	units for file-i/o
nst_ipc.c	units for interprocess communication (alpha)
nst_ipc_aux.c	auxiliary routines for the above
nst_manus.c	unit for controlling manus via MCCL
nst_rpc.c	rpc interface as as NST-unit
nst_rpc.x	rpc definitions for rpcgen
nst_rpc_clnt.c	client stubs for the above (generated by rpcgen)
nst_rpc_svc.c	remote server stubs for the above (generated by rpcgen)
nst_rpc_xdr.c	xdr translation routines (generated by rpcgen)
nst_rpc_xdr.	xdr translation routines (generated by rpcgen)
nst_rpd.c	NST rpc interface, server side
nst_rpd_add.c	extensions to rpc interface
nst_scott.c*	NST–SCOTT interface

Various other units

nst_aux0.c	auxiliary routines for 2D plotting
nst_aux1.c	auxiliary routines for pixel operations
nst_aux2.c	auxiliary routines for text edit windows
nst_aux4.c	auxiliary routines for file-i/o
nst_aux5.c	general graphics auxiliary routines
nst_aux6.c	auxiliary routines for “image awk”
nst_aux7.c	auxiliary routines for parsing
vscanf.c	extended emulation of `vscanf` function