MCW *AFNI* — Plugins α

Robert W. Cox, Ph.D.
rwcox@mcw.edu

User Written Extensions to *AFNI*

A **plugin** is an external software package that can be read into *AFNI* at run time, and be called by the user at any time. An "fill in the form" interface can be created for each plugin: the user fills in the input options relevant to the desired operation, and then presses the "`Run`" button to actually execute the plugin routine. *AFNI* provides routines for the plugin to create the interface form, and to read data from it. Functions and macros are also provided for the plugin to manipulate *AFNI* datasets.

This version of the plugin package for *AFNI* is being released partly for the purpose of testing the development of new plugins. The programmer interface to plugins and *AFNI* datasets is not complete, and new features will undoubtedly evolve as needs become apparent.

This document is written for the `C`-literate *AFNI* user who wishes to write a plugin. Sample plugins can be found in the MCW *AFNI* 2.00 distribution in the files `plug_*.c`. The file `afni_plugin.c` contains the source code for the parts of *AFNI* which are specific to dealing with plugins.

There is a lot of information in this manual. It is not necessary to absorb all of it before beginning to write your first plugin. The fastest way to proceed is to read §1, skim the rest of this document, and then use an existing plugin as a model.

Contents

1 *AFNI* Datasets: Background Information

The central type of data structure in *AFNI* is the dataset. In MCW *AFNI* 1.99 and later, a dataset can have many forms. Since virtually all plugins will manipulate datasets, a thorough understanding of the possible types of datasets is necessary for a plugin author.

1.1 Storage of Datasets on Disk

A dataset `.BRIK` file contains only image data. However, it is not necessary that a dataset actually have a `.BRIK` file. This is due to the **warp-on-demand** feature of *AFNI* : images can be displayed by being transformed from a **parent** dataset that *does* have a `.BRIK` file.

There were two reasons for the development of this capability. The first reason was speed of interaction: it takes about a minute to transform a typical 3D dataset from original to Talairach coordinates (for example). It only takes a second to do a single slice. I didn't want to make the users wait that minute when the "Transform Data" button is pressed, only to find that a mistake had been made and the transformation would have to be recomputed. The second reason was conservation of memory. A typical functional dataset is gathered at a much coarser resolution than an anatomical reference. For display purposes, the functional dataset must be transformed to the anatomical dataset's grid. If this could not be performed "on demand", then every functional dataset would have to be interpolated to the 3D grid of its anatomical reference. This would consume large amounts of memory, large amounts of disk space, and large amounts of CPU time.

The *AFNI* "Write Brick" buttons call the warp-on-demand routines once for each slice in the output dataset, and write the slices to disk to form the new `.BRIK` file.

This presents a problem for the plugin author. To deal with all possible datasets that might be presented to it, a plugin would have to be able to call the warp-on-demand routines and deal with datasets in this slice-at-a-time fashion. This can be quite complex. As an alternative, I have set up the plugin interface so that the plugin author can specify that only datasets with actual 3D `.BRIK`s be passed to the plugin routines. Then the plugin code can be confident it will only have to deal with collections of 3D arrays.

⋆⋆⋆ ⟶ For the present, plugins must deal only with the datasets that actually have `.BRIK` files. I have not yet documented the methods required for a plugin to deal with warp-on-demand datasets.

Plugin authors also need to be aware that dataset bricks may not be stored in memory at any given moment. To overcome this, the macro `DSET_load` must be called to read the dataset `.BRIK` file into memory, if it is not already present.

Another complication is that dataset brick arrays can be stored in one of two fashions. The first method is via the usual `malloc` of workspace. The second method uses the Unix `mmap` function, which maps the `.BRIK` file directly into memory address space. *This mapping is done in readonly mode*; that is, a plugin must not attempt to write into a brick array of a `mmap`-ed file. If a plugin wishes to modify an existing dataset, then it must force the dataset brick arrays to be stored in `malloc` mode, rather than `mmap` mode. A routine is provided to do this.

1.2 Dataset Types

All datasets have a **type** attached. Each type is categorized as being **anatomical** (which can be displayed in the background) or as being **functional** (which can be displayed as the color overlay).

At present, all anatomical types are treated equally. The different names (`spgr`, `epan`, ...) are just for the user's convenience. There are at present five functional dataset types, which are *not* all treated equally:

`fim` Functional Intensity
— one value is stored per voxel

`fith` Functional Intensity + Threshold
— two values are stored per voxel:
— the first is "intensity" (defined arbitrarily);
— the second is "threshold", which is a number between -1.0 and 1.0, which can be used to select which voxels are considered "active".

`fico` Functional Intensity + Correlation
— two values are stored per voxel:
— the first is "intensity";
— the second is a correlation coefficient (between -1.0 and 1.0), which can be used to select "active" voxels at a given significance (p) value.

`fitt` Functional Intensity + t-test
— two values are stored per voxel:
— the first is "intensity";
— the second is a t statistic, which can be used to select "active" voxels at a given significance.

`fift` Functional Intensity + F-test
— two values are stored per voxel:
— the first is "intensity";
— the second is an F statistic, which can be used to select "active" voxels at a given significance.

The plugin interface allows the author to specify which dataset types can be passed. For example, if one wishes to do some calculation specific to the correlation coefficient values, then allowing only `fico` datasets into the plugin would be sensible. The alternative is to allow arbitrary datasets into the plugin, and then test them for type. For example, this might necessary if one wishes to write a plugin that could optionally apply a threshold to a dataset before further operations. If no threshold is desired by the user, then it would make little sense to exclude `fim` datasets from entry.

1.3 Dataset Bricks and Sub-bricks

An *AFNI* dataset contains one or more 3D sub-brick arrays. For example, a 3D+time dataset is essentially just an array of 3D sub-brick arrays, one at each time index. (Note that there is no provision at present for more than one sub-brick array at each time index. This may change in the future!) Each sub-brick is mapped to a single contiguous block of memory; however, adjacent sub-bricks will not necessarily be adjacent in memory.

Sub-bricks can be arrays of type `byte` (a `typedef` for `unsigned char`), `short` (16 bit signed int), `float`, or `complex` (a `struct` containing two `floats`). In places, I refer to these basic datum types as "atomic types". At present, all MCW *AFNI* programs will create datasets in which all sub-bricks are of the same atomic type. This is not required by *AFNI* itself, and may change in the future.

It is likely that a plugin author will wish to restrict the input to contain only sub-bricks of a certain atomic type. The interface definition routines allow this. At MCW, we deal mostly with short bricks. The plugin interface also allows the choice of input to be restricted to 3D or 3D+time datasets.

Note that `byte` and `short` bricks may also have attached a `float` scaling factor. This is used to allow compact storage of values that are really `floats`. For example, the correlation coefficient sub-brick of a `fico` dataset (as generated by the `3dfim` program) will normally be stored as `shorts` in the range $[-10000, 10000]$, with a scaling factor of 0.0001 attached to the brick.

1.4 Dataset idcodes

⋆⋆⋆ ⟶ **Customizing idcodes**

As of *AFNI* 1.99, all datasets have a (hopefully) unique identifying string embedded in them — this is called the idcode. An example is `MCW_FCKWJALORIV`. The 4 letter prefix (`MCW_` in this example) is given by the C macro `IDCODE_PREFIX`, which can be defined in the `machdep.h` file (changing this will require recompiling the entire MCW *AFNI* package). If it is not otherwise defined, then `MCW_` will be used for the prefix. The rest of the idcode string (a total of 16 characters) is generated pseudorandomly (by combining the computer nodename with the time of day).

Even within a single run of *AFNI*, one cannot assume that a pointer to a dataset `struct` will remain constant. *AFNI* may deallocate and reallocate dataset pointers during its run (this will happen when one of the "`Rescan`" buttons is used, for example). The only way to uniquely retain a hold on a dataset is to use its idcode. For that reason, the plugin interface routines return a pointer to an idcode for each dataset that the user inputs. The plugin must then call a function to get the pointer to the actual dataset `struct`. If a plugin wishes to remember which datasets it was called with in previous executions, it should save the idcodes, not the dataset pointers.

The use of idcodes presents a problem when copying datasets. If the PLUTO_copy_dset routine is used to make a working copy of a dataset, then a new idcode will be generated. But if a user uses the Unix cp command to copy a dataset's .HEAD and .BRIK files, this new dataset will have the same idcode as the original. This will cause confusion, at best, if both datasets are then loaded into the same *AFNI* run.

There is no simple solution to this problem, since datasets can refer to each other (*e.g.*, a dataset transformed to Talairach coordinates contains an idcode that points to its parent dataset). When a dataset is copied, does the user want its children to point to the copy or to the original parent? Should a copy of a dataset that points to a parent still point to that parent? (If so, then two datasets will have the same parent. If not, what is the parent of the copied dataset to be?) As a partial solution, you can use the auxiliary program 3dnewid to attach a new idcode to a dataset. This will not change the idcodes it has that point to other datasets (its 'parents'), nor will this change the idcodes inside other datasets that point to the altered dataset (its 'children').

2 Accessing Data in a Dataset

There are a number of functions and macros that provide access to the data inside a dataset. Alternatively, one can directly access the struct data elements; this is necessary in some cases, since at present the dataset struct type is not fully encapsulated behind a wall of interface routines.

Many of the routines and data types start with the string THD_ — this stands for "3 dimensional dataset". The macros are defined in the file 3ddata.h and the functions in 3ddata.c. *Do not modify these files, since they are at the core of the entire* MCW AFNI *package.*

⋆⋆⋆ ⟶
Not a dog

Other routine names described below start with the string PLUTO_ — this stands for "PLugin UTility Operation". These are routines (or macros) that were developed specifically for the use of plugins.

2.1 Pointer to the Dataset

The type MCW_idcode is a typedef for a struct that contains the idcode string described above. When a user calls a plugin and provides a dataset, the plugin retrieves a "MCW_idcode *" by using the routine PLUTO_get_idcode. To get a pointer to an actual dataset structure, then the appropriate code fragment is:

```
MCW_idcode * idc ;
THD_3dim_dataset * dset ;
idc  = PLUTO_get_idcode(plint) ;
dset = PLUTO_find_dset(idc) ;
```

(The plint variable will be explained in §4.) The pointer dset is needed for all further access to the dataset contents.

If two idcodes are obtained, the plugin can test them for equality using the macro `EQUIV_IDCODES(*idc1,*idc2)`, which will return 1 if the idcodes pointed to by `idc1` and `idc2` are equivalent. Note that the macro takes as input actual `MCW_idcode structs`, not pointers `MCW_idcode *`, which is the return type of `PLUTO_get_idcode()`.

2.2 Loading and Unloading Data

The macro `DSET_load(dset)` will call the appropriate routine to read all the sub-bricks of a dataset into memory, if need be. The macro `DSET_unload(dset)` will free the memory space occupied by the input dataset. Note that *AFNI* may execute `DSET_unload()` on a dataset after the plugin returns control to the main program.

2.3 Dataset Dimensions and Orientation

The spatial extent of a dataset is stored in a substructure called a `THD_dataxes`. Some relevant elements of this structure can be accessed by:

`dset->daxes->nxx`	Number of voxels in the x direction
`dset->daxes->xxorg`	x coordinate of center of voxel 0
`dset->daxes->xxdel`	voxel size in x direction (may be negative)

The x coordinate of the center of the i^{th} voxel in the x direction is

$$\texttt{dset->daxes->xxorg} + \texttt{dset->daxes->xxdel} \times i \quad \text{for } i = 0 \ldots \texttt{dset->daxes->nxx-1}.$$

Similar variables exist for the y and z directions, with the `xx`'s in the variable names replaced by `yy`'s and `zz`'s, respectively.

For many purposes, a plugin does not need to know the orientation of the dataset arrays. If such a need is encounted, the orientation is also stored in the `dset->daxes` structure. If you are operating on AC–PC or Talairach coordinate data, this will always be $-x$ = Right, $+x$ = Left, $-y$ = Anterior, $+y$ = Posterior, $-z$ = Inferior, and $+z$ = Superior. In original coordinates, the data will be stored in whatever slice orientation was used to form the dataset in program `to3d`. For example, the orientation of the x axis can be found by looking at the variable `dset->daxes->xxorient`, which will have one of the values

`ORI_R2L_TYPE` = Right-to-Left
`ORI_L2R_TYPE` = Left-to-Right
`ORI_P2A_TYPE` = Posterior-to-Anterior
`ORI_A2P_TYPE` = Anterior-to-Posterior
`ORI_I2S_TYPE` = Inferior-to-Superior
`ORI_S2I_TYPE` = Superior-to-Inferior
`ORI_GEN_TYPE` = General orientation (not used at present)

The last code is intended for the long-awaited but oft-postponed day when *AFNI* supports the creation of datasets with arbitrary axes orientations.

2.4 Dataset Sub-brick Arrays

The data in the p^{th} sub-brick of a dataset is accessed by the macro `DSET_ARRAY(dset,`p`)`. This will return a "`void *`" which must be cast to the appropriate pointer type. If the pointer returned is `NULL`, then the dataset is not loaded into memory (if it does not have a `.BRIK` file, this will certainly be the case; otherwise, `DSET_load(dset)` should be executed).

Unless the plugin interface specified that only datasets with one particular brick type were allowed, a plugin must be able to determine what type of data is stored in a sub-brick. This is done with the macro `DSET_BRICK_TYPE(dset,`p`)`. The value returned will be one of the following integers:

`MRI_byte`	meaning `unsigned char`
`MRI_short`	meaning `short int`
`MRI_float`	meaning `float`
`MRI_complex`	meaning `complex`, which is defined by `typedef struct { float r,i ; } complex ;`

The (i, j, k) voxel in the dataset is at location

$$i + \texttt{dset->daxes->nxx} \times j + \texttt{dset->daxes->nxx} \times \texttt{dset->daxes->nyy} \times k$$

in the array returned by `DSET_ARRAY`.

The sample plugin `plug_3ddot.c` shows an example of manipulating datasets of general type. In general, this could get quite complex. One way to deal with this (as in `plug_3ddot.c`, `3dmerge.c`, and other codes) is to make a `float` copy of a dataset brick, and operate on that, only converting the results to the desired output type when done.

2.5 Miscellaneous Dataset Items and Actions

A glance at `3ddata.h` and `3ddata.c` will show that there are many components to an *AFNI* dataset. Only some of them are outlined here.

Usage	(Type): Description
`DSET_BRICK_TYPE(dset,`p`)`	(`int`): See §2.4
`DSET_BRICK_BYTES(dset,`p`)`	(`int`): How many bytes in sub-brick p
`DSET_PRINCIPAL_INDEX(dset)`	(`int`): Index (p) of "principal value" in the dataset (currently is always 0)
`DSET_THRESH_INDEX(dset)`	(`int`): Index (p) of the "threshold value" if the dataset has one; will be -1 if the dataset has no such data (currently is 1 for `fith`, `fico`, `fitt`, and `fift`)

DSET_PREFIX(dset)	(char *): Pointer to filename prefix string
DSET_FILECODE(dset)	(char *): Pointer to filename prefix+view string
DSET_IDCODE(dset)	(MCW_idcode *): Pointer to dataset idcode
DSET_BRICK_FACTOR(dset,p)	(float): Scale factor to apply to apply to each element of the p^{th} sub-brick. If zero, no scaling applies.
DSET_NUM_TIMES(dset)	(int): Number of time points in dataset (will be 1 except for 3D+time datasets)
DSET_NVALS_PER_TIME(dset)	(int): Number of sub-bricks at each time point (currently is always 1)
DSET_NVALS(dset)	(int): Total number of sub-bricks in dataset
DSET_GRAPHABLE(dset)	(int): Returns 1 if this dataset is 3D+time, and can be loaded into memory
ISVALID_3DIM_DATASET(dset)	(int): Returns 1 if dset points to a valid dataset
dset->view_type	(int): Determines coordinate system dataset has been transformed to: VIEW_ORIGINAL_TYPE = +orig VIEW_ACPCALIGNED_TYPE = +acpc VIEW_TALAIRACH_TYPE = +tlrc
ISANAT(dset)	(int): Returns 1 if dataset is an anatomical type
ISFUNC(dset)	(int): Returns 1 if dataset is a functional type
dset->func_type	(int): Code indicating what type the dataset is: will be one of the ANAT_*_TYPE or FUNC_*_TYPE codes in 3ddata.h. **N.B.**: It is necessary to use the ISANAT macro (say) together with dset->func_type to precisely determine the dataset type, since the ANAT_*_TYPE and FUNC_*_TYPE codes overlap; for example FUNC_FIM_TYPE and ANAT_SPGR_TYPE are both 0.
DSET_write(dset)	(none): Writes dataset to disk (.HEAD and .BRIK)
PLUTO_output_header(dset)	(none): Write dataset .HEAD to disk

2.6 Creating a New Dataset

Creating a completely new dataset is moderately complex. Under most circumstances, however, a plugin will usually wish to make an "empty copy" of an existing input dataset — an empty copy contains no sub-bricks, but contains all the geometric and type information of a dataset. This is done with the call

```
THD_3dim_dataset * old_dset , * new_dset ;
new_dset = EDIT_empty_copy( old_dset ) ;
```

Function EDIT_empty_copy takes as input a pointer to a dataset and returns as output a pointer to a new dataset where all of the control information about the dataset (except its idcode) has been copied. The plugin can then use EDIT_dset_items (described below) to change whatever it needs to about the new dataset. The routine EDIT_substitute_brick can then be used to attach new sub-brick arrays to the new dataset:

```
EDIT_substitute_brick( new_dset , p , MRI_type , ptr ) ;
```

where p is the index of the sub-brick to replace; MRI_type is one of the MRI_ type codes given earlier (type is byte, short, float, or complex)—see §2.4; and ptr is a pointer to an array of values of that type, which must have been created using malloc (not XtMalloc). If ptr is NULL, then appropriate space will be malloc-ed, which can later be accessed using DSET_ARRAY(new_dset,p).

The function PLUTO_copy_dset (in afni_plugin.c) can also be used to create a "full copy" of a dataset:

```
THD_3dim_dataset * old_dset , * new_dset ;
char * new_prefix ;
new_dset = PLUTO_copy_dset( old_dset , new_prefix ) ;
```

The new dataset will have all the sub-bricks copied as well. This routine can be used as an example of how to create a dataset.

2.7 Changing Dataset Internal Items

If one wishes to change the contents of dataset control items, it might seem simplest to do this directly by accessing the appropriate fields of the THD_3dim_dataset structure. This is confusing, however, and another method is available. The routine EDIT_dset_items takes as input a dataset pointer (such as dset above), and a paired list of subitems to modify and their new values. (The call syntax is similar to XtVaSetValues.) For example, to change a dataset memory allocation type to malloc (so that the plugin can modify the sub-brick arrays), and at the same time change the dataset filename prefix, the appropriate call would be

```
char * new_prefix ;
ierr = EDIT_dset_items( dset,
                          ADN_malloc_type, DATABLOCK_MEM_MALLOC ,
                          ADN_prefix , new_prefix ,
                        ADN_none ) ;
```

Here, the return code ierr is the count of the number of errors that happened during EDIT_dset_items (this should be 0). The two items on the second line are the first pair of inputs, which in this case are a code, ADN_malloc_type, indicating the dataset item to modify, and the new value of that item, DATABLOCK_MEM_MALLOC (a symbolic constant defined in 3ddata.h). The two items on the third line are the second pair of inputs: in this case, ADN_prefix indicating that the next input is a new prefix for the dataset's disk files, and new_prefix, a pointer to the new prefix string. The last line is the special value ADN_none, indicating that no more inputs to EDIT_dset_items are present.

The following table shows all the values that can be altered using EDIT_dset_items. It is not possible to alter items that affect the size of a dataset after the dataset actually has sub-bricks attached to it. That is, EDIT_empty_copy(dset) can be used to create a dataset with no sub-bricks, then EDIT_dset_items can be used to alter the dataset control parameters at will, but once EDIT_substitute_brick is used to actually attach data sub-bricks to the new dataset, then altering parameters such as ADN_nxyz is illegal. Such parameters are marked with a superscript $\star$ in the table.

Control Code	Data Type	Meaning
ADN_prefix	char *	Changes the prefix for the dataset filenames.
ADN_directory_name	char *	Changes the name of the directory where the dataset will be stored.
ADN_brick_fac	float *	Changes the scaling factors attached to all sub-bricks. The input is a float array of length DSET_NVALS(dset). The p^{th} entry contains the scale factor for the p^{th} sub-brick. If this scale factor is zero, then the sub-brick values will not be scaled; otherwise, they will be multiplied by this factor before being used in *AFNI* .
ADN_malloc_type	int	Changes the way in which the sub-brick arrays are stored in memory. Must be one of the following codes: DATABLOCK_MEM_MALLOC = use malloc DATABLOCK_MEM_MMAP = use mmap DATABLOCK_MEM_ANY = let *AFNI* choose Note that if this parameter is changed, then the dataset must have DSET_load() executed on it before the sub-bricks will be available in memory.
*ADN_datum_all	int	Changes the atomic type stored in all sub-bricks. Should be one of the following codes—see §2.4: MRI_byte MRI_short MRI_float MRI_complex Note that EDIT_substitute_brick can change the atomic type of a sub-brick later.
*ADN_datum_array	int *	Changes the atomic type stored in all sub-bricks. Unlike ADN_datum_all, the input is an array of length DSET_NVALS(dset) which contains MRI_type codes for each sub-brick individually.
*ADN_nvals	int	Changes the number of 3D sub-bricks stored in the dataset.

*ADN_nxyz	THD_ivec3	Changes the number of points stored along each axis of the dataset (*e.g.*, `nxx`).
ADN_xyzdel	THD_fvec3	Changes the spacing (mm) between points along each axis of the dataset (*e.g.*, `xxdel`).
ADN_xyzorg	THD_fvec3	Changes the offset (mm) to the center of the 0^{th} voxel along each axis of the dataset (*e.g.*, `xxorg`).
ADN_xyzorient	THD_ivec3	Changes the orientation codes for each axis of the dataset (*e.g.*, `xxorient`).

ADN_type	int	One of the following codes:
		`HEAD_ANAT_TYPE HEAD_FUNC_TYPE`
		`GEN_ANAT_TYPE GEN_FUNC_TYPE`
		This values is used with `ADN_func_type` to determine how the dataset is processed. `ANAT` types are displayed as the background grayscale images in *AFNI*. `FUNC` types are displayed as the color overlay images.

ADN_view_type	int	Determines the coordinate system that this dataset has been transformed to:
		`VIEW_ORIGINAL_TYPE` = `+orig`
		`VIEW_ACPCALIGNED_TYPE` = `+acpc`
		`VIEW_TALAIRACH_TYPE` = `+tlrc`.

ADN_func_type	int	If `ADN_view_type` is an `ANAT` type, then this should be one of:
		`ANAT_SPGR_TYPE ANAT_FSE_TYPE ANAT_EPI_TYPE`
		`ANAT_MRAN_TYPE ANAT_CT_TYPE ANAT_SPECT_TYPE`
		`ANAT_PET_TYPE ANAT_MRA_TYPE ANAT_BMAP_TYPE`
		`ANAT_DIFF_TYPE ANAT_OMRI_TYPE`
		If `ADN_view_type` is an `FUNC` type, then this should be one of:
		`FUNC_FIM_TYPE FUNC_THR_TYPE FUNC_COR_TYPE`
		`FUNC_TT_TYPE FUNC_FT_TYPE`
		Note that if `ADN_func_type` is changed, the number of sub-bricks appropriate for this dataset may change also. You must do this explicitly using `ADN_nvals` at the same time.

ADN_stat_aux	float *	The `float *` argument must point an array of length `MAX_STAT_AUX`, which will contain the auxiliary statistical parameters needed for the `fico`, `fitt`, or `fift` dataset.
ADN_ntt	int	Changes the number of time points in the dataset. **N.B.**: You must also change `ADN_nvals` when you change this!
ADN_ttorg	float	Changes the origin of time for 3D+time datasets.

ADN_ttdel	float	Changes the time step for 3D+time datasets.
ADN_ttdur	float	Changes the duration of data acquisition for each time point. This parameter is currently unused by any *AFNI* program.
ADN_tunits	int	Changes the units that are used for time in a 3D+time dataset. Legal values are: UNITS_MSEC_TYPE UNITS_SEC_TYPE UNITS_HZ_TYPE.

Note that the THD_ivec3 and THD_fvec3 types are defined in vecmat.h (which will be included when you #include "afni.h"). They are used to encapsulate 3-vectors of integers and floats, respectively. (By the way, ADN stands for Afni Dataset Name.)

2.8 Header Attributes

As you should know by now, *AFNI* datasets are stored in two files, with names ending in .HEAD and .BRIK. The .HEAD file is plain ASCII and contains **attributes**, which are named arrays. When *AFNI* reads in a .HEAD file, it firsts reads all the attributes in, then searches them for the ones needed to define a dataset. Other attributes will be ignored by *AFNI*, but still will be in memory, and will still be written to disk when the dataset output routine is called. This means that a plugin can attach new attributes to a dataset, and that these will be preserved when the dataset is written out and read back in.

There are three types of attributes: string, float, and int. They are stored in structs defined below:

```
typedef struct {             typedef struct {             typedef struct {
      int    type ;                int    type ;                int    type ;
      char * name ;                char * name ;                char * name ;
      int    nch ;                 int     nfl ;                int    nin ;
      char * ch ;                  float * fl ;                 int   * in ;
} ATR_string ;               } ATR_float ;                } ATR_int ;
```

A string attribute can be added to a dataset with the routine

```
THD_set_string_atr( dset->dblk , name , str ) ;
```

where name is a char * that points to a NUL-terminated string which will be used to identify the attribute, and str is a char * that points to the NUL-terminated string to store as the attribute's value. A float attribute can be added to a dataset with the routine

```
THD_set_float_atr( dset->dblk , name , nfl , fl ) ;
```

where nfl is the number of floats to be stored in the attribute, and fl is a float * that points to the values to be stored. Similarly,

```
THD_set_int_atr( dset->dblk , name , nin , in ) ;
```

is used to add an `int` attribute to a dataset, where `nin` is the number of integers to store and `in` is an `int *` pointing to the values to be stored.

Attributes will be saved to the dataset's `.HEAD` file the next time the dataset is written to disk. To force the `.HEAD` file `only` to be written out, call

```
PLUTO_output_header( dset ) ;
```

The function call `THD_find_string_atr(dset->dblk,name)` will return a pointer of type "`ATR_string *`" if an attribute is found that has the same `name`. If `NULL` is returned, then no match was found. Otherwise, the `nch` and `ch` entries of the `ATR_string struct` contain the count of characters and the character array, respectively. Similar routines and remarks apply to finding and using `float` and `int` attributes.

3 Plugin Libraries

Plugins are C routines compiled and linked into the shared (or dynamic) library format. On most Unix systems, the suffix for such files is `.so`, but on HP–UX systems, the file suffix is `.sl`. (HP–UX also uses a different API to access such shared libraries, but *AFNI* will take care of that for you.)

Plugins can be compiled into shared library format using the `Makefile` distributed with MCW *AFNI* 2.00. For example, `make fred.so` will compile `fred.c` into `fred.o`, and then appropriately run `ld` on `fred.o` to produce `fred.so`. At this time, I have only verified that my compilation procedure and shared library interface code works on Linux 1.2.13, SGI IRIX 5.3 and 6.2, HP–UX 9.05 and 10.0, and Sun Solaris 2.5. Making plugins work on other systems will require appropriate editing of the platform-specific Makefiles and also of the `machdep.h` file.

⋆⋆⋆ ⟶ **System specifics**

AFNI will search for plugin libraries when it starts. It will look in the directories specified in the shell environment variable `AFNI_PLUGINPATH` (this is a colon-separated list of directories — for an example, type `echo $PATH` to see what your current executable path is). If this variable does not exist, then *AFNI* will use the `PATH` variable instead. This makes it possible to put the plugin libraries in the same location as the *AFNI* binaries.

AFNI will attempt to load all shared library files it finds. System libraries will not work properly, since they will not have the required routine `PLUGIN_init` (see the next section). Such libraries will then be unloaded. If more than one copy of a correct plugin is found, *AFNI* will load them all. This will probably confuse the user. (It confuses me.)

4 Creating a Plugin

There are three issues to address in creating a plugin:

- Creating a user interface.
- Getting data out of the the user interface when the plugin is called.
- Doing something useful and communicating it back to *AFNI*.

Each plugin must have one routine named `PLUGIN_init`. *AFNI* will call this routine to get the specification of how the plugin interface should look. This routine should be prototyped

```
PLUGIN_interface * PLUGIN_init( int ncall )
```

The output type "`PLUGIN_interface *`" is a pointer to a type defined in `afni_plugin.h`. This file will be among the header files loaded with the statement `#include "afni.h"`, which must be at the top of every plugin source file. The return value must be constructed using the routines described later. A variable of type `PLUGIN_interface` encapsulates the way a plugin will be called from *AFNI*.

Each plugin library can define more than one interface. That is the purpose of the `ncall` input to `PLUGIN_init`. On the first call to `PLUGIN_init`, `ncall` will be 0. On each subsequent call, `ncall` will be one larger. This will continue until `PLUGIN_init` returns `NULL`. After that, `PLUGIN_init` will never be called.

Plugin interfaces are called up by the user from the `Plugins` menu button in *AFNI*, located at the bottom of the `Datamode` control panel. If this menu button is not present, this means that *AFNI* did not find any plugins when it started up.

In principal, a plugin need not define any interfaces. It could simply use the call to `PLUGIN_init` to start itself, and then it could get all the information it needs directly from *AFNI*, or from the user by popping up its own windows. For this reason, the `PLUGIN_init` functions are called only after *AFNI* has read in all its inputs, has initialized X11, and is just about to pop up the first controller window.

At present, such a "standalone" plugin would have to use something like `XtAppAddTimeOut` or X11 events to get control after `PLUGIN_init` returns. It would be possible for a plugin to spawn threads to process data while *AFNI* runs in the "foreground", but anyone contemplating this approach should be aware that *AFNI* and Motif are not thread-safe — a thread cannot directly call a routine in *AFNI* or Motif to get some information or perform some task. (I have thought about adding a "heartbeat" to *AFNI* , so that a plugin could register to be called from the *AFNI* main thread every cycle (say 100 ms). This would allow the plugin main thread to service requests from other threads through the use of mutexes and condition variables. Any comments on this idea?)

A plugin interface can simply specify that the plugin be called immediately when its menu item is picked. In this case, it the the plugin code's responsibility to get whatever inputs it needs from the user — perhaps by popping up its own control window. This might be appropriate for a plugin that provided some new graphical capability; for example, a volume rendering tool (anyone want to write this one for me?). None of the sample plugins use this method of plugin activation.

All the sample plugins use the facility that *AFNI* provides to create a popup control box, which lets the user enter control parameters and then press a "Run" button to actually call the plugin code. The routines described below let the plugin author specify what kind of parameters (*e.g.*, strings, numbers, or datasets) are required by the plugin. When the user chooses the plugin from the *AFNI* menu, this control box will be popped up to the display. If a plugin's needs fit into this paradigm, a plugin need have no direct interaction with the user — the routines in `afni_plugin.c` will do these chores.

4.1 Creating a Plugin Interface

A quick summary of the procedure, which is to be used from within the `PLUGIN_init()` function:

Step 1: Use function `PLUTO_new_interface` to create the initial `PLUGIN_interface` data structure.

Step 2: Use function `PLUTO_add_option` to create an option line in the *AFNI* interface menu. There is no built-in limit to the number of option lines that may be added to an *AFNI* interface menu. (An "option line" is a line of "chooser" widgets in the interface window. It may or may not be optional, depending on how you call this routine.)

Steps 2(abcdef): Use functions

`PLUTO_add_number`, `PLUTO_add_string`, `PLUTO_add_dataset`,
`PLUTO_add_dataset_list`, and `PLUTO_add_timeseries`

to add plugin parameter choosers to the most recently created option line. Up to 6 choosers may be added to an option line. When one option line is finished, return to Step 2 to create the next.

Step 3: When done, return the new `PLUGIN_interface *` to *AFNI*.

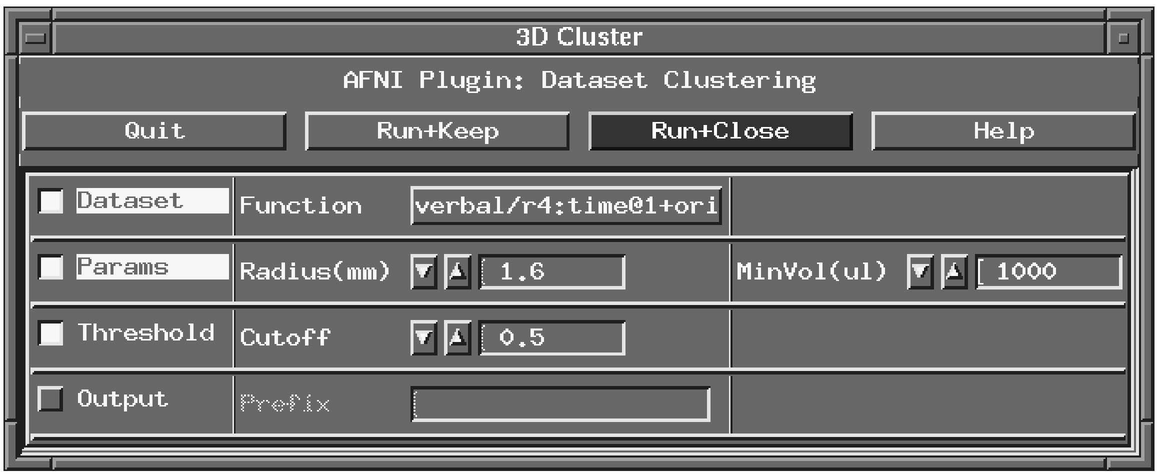
For each `PLUTO_add_*` routine, there is a corresponding `PLUTO_get_*` routine to retrieve the user's inputs from the popup window.

These interface creation routines are documented below. Examples of their usage found in the sample plugins. Note that there is currently no facility for passing other kinds of information to a plugin.

The figure below shows a popup control box created by application of the above procedures. At the top are four control buttons:

`Quit`	To close the popup without running the plugin;
`Run+Keep`	To run the plugin and keep the popup open;
`Run+Close`	To run the plugin and close the popup window;
`Help`	To popup a window with the help string provided by the plugin author when `PLUTO_new_interface` was called.

Below, there are four option lines, labeled `Dataset`, `Params`, `Threshold`, and `Output`. The first option line has a dataset chooser, the next two have number choosers, and the fourth (which has been toggled off by the user) has a string chooser.



Sample plugin interface window (from `plug_clust`).

4.1.1 `PLUTO_new_interface`

This routine is used to initialize a new interface to a plugin. Its prototype is

```
typedef char * cptr_func() ;  /* generic function returning "char *" */

PLUGIN_interface * new_PLUGIN_interface( char * label ,
                                         char * description ,
                                         char * help ,
                                         int call_type ,
                                         cptr_func * call_func ) ;
```

The arguments are

`label`	C string to go on the menu button that activates this interface (will be truncated to 15 characters).
`description`	C string to go on the interface control panel popped-up when the button above is pressed—this has no meaning if `call_type = PLUGIN_CALL_IMMEDIATELY`. In the figure above, this string was `"Dataset Clustering"`.
`help`	C string to be popped up if the user presses `Help` on the interface control panel—this has no meaning for `call_type = PLUGIN_CALL_IMMEDIATELY`. If this is `NULL`, then there will be no help available.
`call_type`	C `int` that describes how the plugin is to be called from *AFNI* : `PLUGIN_CALL_IMMEDIATELY` means to call `call_func` as soon as the activating button is pressed; `PLUGIN_CALL_VIA_MENU` means to have *AFNI* popup a menu to control the input parameters passed to `call_func`.
`call_func`	Function that *AFNI* should call when the user activates this plugin. The routine will be passed the pointer "`PLUGIN_interface *`" created herein, which can be interrogated with the `PLUTO_get_*` routines. The `call_func` should return a "`char *`", which is `NULL` if everything is OK, and points to an error message that *AFNI* will display if something bad happened. See the sample plugins for examples of how this works.

The value returned by `PLUTO_new_interface` is the pointer to the new interface `struct`. This value should also be the return value of the `PLUGIN_init` routine. This pointer will be called `plint` in the sample code fragments below. Note that the three input strings (`label`, `description`, and `help`) are copied by *AFNI*.

⋆⋆⋆ ⟶ plint

4.1.2 `PLUTO_add_option`

This routine is called to add an "option line" to a plugin interface control box. An option line may or may not be optional, depending on the value of `mandatory`. The prototype for this function is

```
void PLUTO_add_option( PLUGIN_interface * plint ,
                       char * label , char * tag , int mandatory ) ;
```

The arguments are

`plint`	The value returned by `PLUTO_new_interface`.
`label`	C string to go at the start of the option line. In the figure above, the first option line label is `"Dataset"`.

`tag`	C string that will be passed to the plugin's `call_func` to identify this option line. Under most circumstances, `tag` can be the same as `label`. Only if you create multiple option lines with the same `label` and wish to be able to distinguish them inside the plugin will the use of `tag` be necessary.
`mandatory`	C `int` saying whether or not this option line has required inputs. If `mandatory=1`, then this option line will always be passed to the `call_func`. If `mandatory=0`, then the user can toggle this line off, in which case the values on that line will not be passed to the `call_func`.

After this function is called, then up to 6 choosers can be added to the option line by calling the following routines.

4.1.3 `PLUTO_add_number`

This routine is called to add a number chooser to the most recently created option line. Numbers are always returned to the plugin `call_func` in floating point format. Numbers can be restricted to a finite set, or be editable by the user. The prototype for this function is

```
void PLUTO_add_number( PLUGIN_interface * plint , char * label ,
                       int bot , int top , int decim ,
                       int defval , int editable      ) ;
```

The arguments are

`plint`	The value returned by `PLUTO_new_interface`.
`label`	C string to go at the left of the chooser.
`bot, top, decim`	C `int`s specifying the range of allowable numbers. `decim` is used to specify the decimal shift leftwards; for example, `decim=2` means that the legal range of this chooser is `0.01*bot`...`0.01*top`.
`defval`	C `int` specifying the default value for the chooser when it is first popped up. If decim is nonzero, then the actual default value is `defval/10**decim`.
`editable`	C `int` specifying if the user can edit (type in) the value. If `editable=1`, then the number can be selected either with the arrow controls or by typing. If `editable=0`, then a fixed menu of numbers is all that is available to the user (from `bot/10**decim` to `top/10**decim`, inclusive).

In the figure above, the number choosers on the second and third option lines are all `editable`.

4.1.4 `PLUTO_add_string`

This routine is used to add a string chooser to the most recently created option line. The string to be entered can either be selected from a fixed list or can be free-form (typed in). The prototype for this function is

```
void PLUTO_add_string( PLUGIN_interface * plint , char * label ,
                       int num_str , char ** strlist , int defval ) ;
```

The arguments are

`plint`	The value returned by `PLUTO_new_interface`.
`label`	C string to go at the left of the chooser.
`num_str`	C `int` providing the count of how many strings are given in `strlist`. If this is 0, then instead of being presented with a list of fixed strings, the user will have to type in a string. At present, the maximum allowed value for `num_str` is 34 (`PLUGIN_MAX_STRING_RANGE` in `afni_plugin.h`).
`strlist`	`strlist[i]` is a pointer to the i^{th} string value that the user is to choose from, for $i = 0 \ldots$`num_str-1`.
`defval`	If `num_str > 0`, then this is an integer from 0 to `num_str-1` indicating which string in `strlist` is the default value. If `num_str == 0`, this gives the width of the field that the user has to type in the string. (I usually use 19 for this case.)

In the figure above, the string chooser on the fourth option line was created with `num_str=0`, and so is a "type in" field (in this case, used to specify the prefix of the output dataset).

4.1.5 `PLUTO_add_dataset`

This routine is used to add a dataset chooser to the most recently created option line. The dataset to be chosen can be restricted in various ways, so that the plugin need not deal with the full range of possibilities described in §1. The prototype for this function is

```
void PLUTO_add_dataset( PLUGIN_interface * plint , char * label ,
                        int anat_mask , int func_mask , int ctrl_mask ) ;
```

The `_mask` inputs are bitwise ORs (`|`) of dataset type masks. These are used to specify the types of datasets that can be passed to the plugin. The first two `_mask` inputs below cannot both be zero (for then no dataset would be allowed into the plugin!).

`plint`	The value returned by `PLUTO_new_interface`.
`label`	C string to go at the left of the chooser.

`anat_mask` C `int` controlling which kind of anatomical datasets are allowable. Chosen from the list in `3ddata.h`, which is currently

`ANAT_SPGR_MASK ANAT_FSE_MASK ANAT_EPI_MASK ANAT_MRAN_MASK`
`ANAT_CT_MASK ANAT_SPECT_MASK ANAT_PET_MASK ANAT_MRA_MASK`
`ANAT_BMAP_MASK ANAT_DIFF_MASK ANAT_OMRI_MASK`

and `ANAT_ALL_MASK`, which will allow any anatomical dataset. Entering 0 for `anat_mask` will mean that no anatomical datasets will be choosable.

`func_mask` C `int` controlling which kind of functional datasets are allowable. Chosen from the list in `3ddata.h`, which is currently

`FUNC_FIM_MASK FUNC_THR_MASK FUNC_COR_MASK`
`FUNC_TT_MASK FUNC_FT_MASK`

and `FUNC_ALL_MASK`, which will allow any functional dataset. Entering 0 for `func_mask` will mean that no functional datasets will be choosable.

`ctrl_mask` An additional mask to specify further exactly which datasets should be choosable:

`SESSION_ALL_MASK`	If this is set, then the choice of datasets will be drawn from all sessions now loaded into *AFNI*. By default, only the "current" session will be included.

`DIMEN_3D_MASK`
`DIMEN_4D_MASK`
`DIMEN_ALL_MASK`
	Masks that define whether 3D and/or 3D+time (4D) datasets are allowable.

`WARP_ON_DEMAND_MASK`	If this is set, then datasets that may not have a `.BRIK` file will be included in the list of datasets to choose from. In this case, the plugin must be ready to deal with the warp-on-demand routines that return one slice at a time. By default, only datasets with actual `BRIK`s will be included.

`BRICK_BYTE_MASK`
`BRICK_SHORT_MASK`
`BRICK_FLOAT_MASK`
`BRICK_COMPLEX_MASK`
`BRICK_ALLTYPE_MASK`
`BRICK_ALLREAL_MASK`
	Masks that define what type of data should be stored in the sub-bricks of the allowable datasets.

Note that entering 0 for `ctrl_mask` means that no datasets will be choosable. At the least, one of the `DIMEN_` masks must be chosen, and one of the `BRICK_` masks must be chosen.

The dataset chooser is shown on the first line of the figure above. It comprises a label to the left and a pushbutton to the right. Clicking on the pushbutton causes a list of eligible datasets to pop up. When the user makes a selection, the name of the selected dataset will be shown on the pushbutton.

4.1.6 PLUTO_add_dataset_list

This routine is used to create a dataset list chooser in the most recently created option line. Such a chooser allows the user to select multiple datasets. At this writing, no serious sample plugin uses this—the "Testing" plugin in plug_power.c has a trivial example, which was used to debug this facility. The prototype for this function is

```
void PLUTO_add_dataset_list( PLUGIN_interface * plint , char * label ,
                             int anat_mask , int func_mask , int ctrl_mask ) ;
```

The arguments are exactly the same as in PLUTO_add_dataset.

4.1.7 PLUTO_add_timeseries

This routine is used to add a timeseries chooser to the most recently created option line. This allows the user to choose from the list of *.1D files that were read in at program startup, or were stored into the timeseries list later (via PLUTO_register_timeseries, or from a graph window). The prototype for this function is

```
void PLUTO_add_timeseries( PLUGIN_interface * plint , char * label ) ;
```

By now, the arguments should be self explanatory. A logical extension of this routine would be PLUTO_add_timeseries_list. Perhaps someday. Then again, perhaps not.

4.2 Getting Data from a Plugin Interface

When the routine (call_func) associated with a "PLUGIN_interface *" is called, then the first thing the plugin must do is extract the information that the user put into the interface window (in the "chooser widgets"). This is done with various routines that start with PLUTO_get_. They all take as input a variable of type "PLUGIN_interface *", referred to as plint below.

The sample plugins show how to use these functions. Note that they are designed assuming that the plugin action function knows the layout of the option lines and choosers as set up by the PLUGIN_init function. In all the sample plugins, the first part of the plugin action routine is essentially a list of calls to PLUTO_get_something, with each "something" being a copy of the PLUTO_add_something that was used to create a chooser widget in the PLUGIN_init function. After all the user input parameters have been acquired and tested, then the actual computations begin.

4.2.1 PLUTO_get_label(plint)

Returns the "char *" label supplied by PLUTO_new_interface. This could be used to decide which plugin interface was calling the action function, if the same call_func were used for more than one call to PLUTO_new_interface. None of the sample plugins use this function (or the next one).

4.2.2 `PLUTO_get_description(plint)`

Returns the "`char *`" `description` supplied by `PLUTO_new_interface`.

4.2.3 `PLUTO_get_optiontag(plint)`

Returns the "`char *`" tag specified for the next option line selected by the user. An option line that is not selected (is toggled off) will be skipped by this function. `NULL` is returned when the chosen options are exhausted. It is necessary to call this routine to be able to get values from chooser widgets in the next option line. If the tag value is not desired, then the macro `PLUTO_next_option(plint)` can be used. This is the usual way to advance to the next option line when it is known in advance what it will be (when the option line is `mandatory`). Note that is necessary to use `PLUTO_next_option(plint)` or `PLUTO_get_optiontag(plint)` to advance to the first option line, before trying to extract user inputs via `PLUTO_get_number(plint)`, etc.

In this routine, as in all the "`PLUTO_get_`" functions, the return value is a pointer (here, `char *`). The memory targeted by this pointer should not be altered by the plugin code. After the plugin returns control to *AFNI*, this memory will be `XtFree`-d.

4.2.4 `PLUTO_get_number(plint)`

Returns the next number from an option line number chooser. This will always be a `float`. (If the next item on the current option line is not a number chooser, or there is no next item, then the special value `BAD_NUMBER` will be returned.)

4.2.5 `PLUTO_get_string(plint)`

Returns a "`char *`" pointing to the next string from an option line string chooser. (If `NULL` is returned, then the next item on the option line is not a string chooser, or there is no next item.)

If the string returned is to be used as a dataset prefix, the routine `PLUTO_prefix_ok(str)` can be used to check if the string `str` is acceptable. If zero is returned by this function, then `str` is not acceptable as a dataset prefix (*e.g.*, it contains an illegal character).

If the string returned was chosen from a fixed set, then the function call

```
int ii ;
ii = PLUTO_string_index( str , num_str , strlist ) ;
```

will return `ii` such that `strcmp(str,strlist[ii])==0)`. If `ii==-1`, then `str` was not found in `strlist`.

4.2.6 PLUTO_get_idcode(plint)

Returns an "MCW_idcode *" from an option line dataset chooser. If the value returned is NULL, then the user did not make a choice. (If the next item on the option line is not a dataset chooser, or there is no next item, NULL will also be returned.) Normally, the first thing that one does with this is to pass it to PLUTO_find_dset, which will return a pointer to the actual dataset (or NULL if the idcode is illegal).

```
MCW_idcode * idc ;
THD_3dim_dataset * dset ;
idc  = PLUTO_get_idcode(plint) ;
dset = PLUTO_find_dset(idc) ;
```

If dset is NULL, then the user did not make a choice.

4.2.7 PLUTO_get_idclist(plint)

Returns an "MCW_idclist *" from an option line dataset list chooser. This is a pointer to a structure which contains a list of MCW_idcodes. The following 3 macros are used to access the contents of the MCW_idclist struct:

PLUTO_idclist_count(idclist)	=	number of MCW_idcodes in the list
PLUTO_idclist_next(idclist)	=	returns an MCW_idcode * that points to the next dataset idcode on the list (returns NULL if past end of list)
PLUTO_idclist_reset(idclist)	=	resets the MCW_idclist so that PLUTO_idclist_next() will start again at the first dataset in the list

One way to use the value returned by PLUTO_get_idclist() is thus to repeatedly call PLUTO_idclist_next(), then PLUTO_find_dset(), until the dataset pointer returned is NULL. This signals that the list of datasets picked by the user has been exhausted.

4.2.8 PLUTO_get_timeseries(plint)

A timeseries is one or more 1-dimensional arrays of floats. If a timeseries chooser is placed in an option line, then the user can select from the list of *.1D files read in when *AFNI* starts. PLUTO_get_timeseries() returns a pointer of type MRI_IMAGE * from an option line timeseries chooser. This points to a ubiquitous and general image struct of a type that is used heavily throughout *AFNI*. Full documentation of the MRI_IMAGE type is beyond me at this time. For the purposes of reading a timeseries from a plugin chooser, the following code fragment will serve:

```
MRI_IMAGE * tsim ;
int ntime , nvec ;
float * vec ;

tsim = PLUTO_get_timeseries(plint) ;
if( tsim == NULL ) return "No Timeseries Input" ;

ntime = tsim->nx ;            /* number of points in each vector */
nvec  = tsim->ny ;            /* number of vectors */
vec   = MRI_FLOAT_PTR(tsim) ; /* vec[i+j*nx] = i'th point of j'th vector */
                              /* for i=0..ntime-1 and j=0..nvec-1 */
```

As the sample above indicates, `PLUTO_get_timeseries()` may return `NULL`, which means that the user did not select a time series.

4.3 Computing Something Useful

Utility is in the eye of the beholder, of course. One of the most common things that a plugin will do is to create a new dataset. The low-level mechanics of this have been discussed in §2.

AFNI provides a routine to automate the creation of one kind of dataset. This function will take as input a 3D+time dataset and return a `fim` dataset, with the intensity in each voxel computed from that voxel's time series. All that the plugin must do is provide a function that takes as input a time series and returns as output the desired intensity value. The output dataset will have its single sub-brick stored as `short`s, with a scaling factor attached (*à la* `DSET_BRICK_FACTOR`).

The prototype for this 3D+time to 3D dataset function is

```
typedef void generic_func() ;
THD_3dim_dataset * PLUTO_4D_to_fim( THD_3dim_dataset * old_dset ,
                                    char * new_prefix ,
                                    int ignore , int detrend ,
                                    generic_func * user_func ,
                                    void * user_data )
```

The inputs to this routine are

- `old_dset` Pointer to the input 3D+time dataset. This dataset must *not* be warp-on-demand.
- `new_prefix` C string that will be the new dataset's filename prefix.
- `ignore` C `int` specifying the number of points at the beginning of each time series that will be ignored in all calculations.
- `detrend` C `int`: if this is 1, then each voxel time series will have its mean and slope removed before being passed to `user_func`.

`user_func` Function provided that computes the output value at each voxel, given that voxel's time series. The details of this function are documented below.

`user_data` A pointer to any data that needs to be passed to `user_func`. This would normally be a pointer to a `struct` that contains parameters for the `fim` calculation.

The function `user_func` should be prototyped as follows:

```
void user_func( double tzero , double tdelta , int npts , float ts[] ,
                double ts_mean , double ts_slope ,
                void * ud , float * val ) ;
```

The arguments to `user_func` are:

`tzero` time at `ts[0]`

`tdelta` time at `ts[1]` (*i.e.*, `ts[k]` is at `tzero + k*tdelta`); `tzero` and `tdelta` will be in seconds if this is truly 'time'

`npts` number of points in `ts` array

`ts` one voxel time series array, `ts[0]` ... `ts[npts-1]`; note that this will always be a `float` array, and that `ts` will start with the `ignore`$^{\text{th}}$ point of the actual voxel time series.

`ts_mean` mean value of ts array

`ts_slope` slope of ts array; this will be inversely proportional to `tdelta` (units of 1/sec); if `detrend` is nonzero, then the mean and slope will been removed from the `ts` array

`ud` the `user_data` pointer passed in here—this can contain whatever control information the user wants

`val` pointer to return value for this voxel; note that this must be a `float`

Before the first time series is passed, `user_func` will be called once with arguments

```
( 0.0 , 0.0 , nvox , NULL , 0.0 , 0.0 , user_data , NULL )
```

where `nvox` = total number of voxel time series that will be processed. This is to allow for some setup (*e.g.*, `malloc` of workspace). No value should be returned in this call; in fact, the return pointer `val` will be `NULL` for this special call, which is how this call can be distinguished from the `nvox` calls that follow.

After the last time series is passed, `user_func` will be called once again with arguments

```
( 0.0 , 0.0 , 0 , NULL , 0.0 , 0.0 , user_data , NULL )
```

This is to allow for cleanup (*e.g.*, `free` of `malloc`). Note that the only difference between the initial and final "notification" calls is the third argument.

If an error occurs, PLUTO_4D_to_fim will return a NULL pointer; otherwise it returns a pointer to the newly created fim dataset. An example of the use of this dataset creating function can be found in plug_stats.c.

5 Sending Information to the User and to *AFNI*

5.1 Sending a Dataset Back to *AFNI*

Editing an Existing Dataset in Place

One way to send the results of a computation back to *AFNI* is to edit an existing dataset in place. The sample plugin plug_clust.c shows how this can be done. If this is done, then *AFNI* must be told to redisplay images (otherwise, it won't know that the plugin altered an existing dataset). This is done with the routine PLUTO_force_redisplay(), which takes no arguments, but simply instructs each *AFNI* image and graph window to redraw itself.

Editing a dataset in place will destroy the data that is stored in its .BRIK file. Careful consideration should be given as to whether this is a desirable capability to give the user. As an alternative, the routine PLUTO_copy_dset can be used to create a complete copy of a dataset, which can then be edited without destroying the original–see §2.6.

Another complication can arise with editing a dataset in the +tlrc view. Suppose that this dataset .BRIK file was originally transformed from a +orig view dataset. Now suppose that the +tlrc dataset .BRIK is altered by a plugin. When the Define Datamode controls are set to View Data Brick, then the edited data will be seen. When they are set to Warp on Demand, the data will be transformed directly from the unedited +orig .BRIK, which will no longer be the same as the +tlrc .BRIK. This can be very confusing.

Creating a New Dataset

Another way to get information back into *AFNI* is to create a new dataset and send it back with the routine PLUTO_add_dset. This will put the dataset into the current session of the *AFNI* controller window from which the plugin was invoked, and (optionally) make it the current dataset for viewing.

```
PLUTO_add_dset( plint , dset , flag ) ;
```

where plint is the "PLUGIN_interface *" passed into the plugin call_func, dset is the pointer to the new dataset, and flag takes on one of the values below:

DSET_ACTION_MAKE_CURRENT	Set the dataset to be the current dataset for viewing.
DSET_ACTION_NONE	Leave the current dataset for viewing as it is.

Putting a dataset into this specific session may not always be desirable — I'm open to suggestions on other ways to specify where a new dataset should be placed. PLUTO_add_dset will write the dataset .HEAD and .BRIK files to disk, so the plugin need not bother to perform this task. In addition, if the new dataset is in the `+orig` view, `PLUTO_add_dset` will create the warp-on-demand children datasets in the `+acpc` and `+tlrc` views.

If a plugin changes the name of one or more pre-existing datasets, the titles in the *AFNI* windows may be wrong. The routine `PLUTO_fixup_names()` (which takes no arguments) will tell *AFNI* to redraw all those titles. An example of how this is used is in `plug_rename.c`.

5.2 Displaying Miscellaneous Information

Some plugins may not wish to send data back to *AFNI* itself, but simply need to open their own windows to communicate something to the user.

Text Strings

If all that is needed is to popup a string in a window, the functions below can be used:

```
PLUTO_popup_message( plint , str ) ;
PLUTO_popup_transient( plint , str ) ;
```

The first function will leave the message box on the screen until the user clicks the mouse inside the box. The second function will destroy the message box after 30 seconds if the user does not click in it before that amount of time has elapsed.

X11 Access

The function `PLUTO_beep()` (which takes no arguments) will cause the display bell to ring. If for some reason a plugin needs direct access to the X11 display on which *AFNI* is running, this is given by the macro `PLUTO_X11_display` (which takes no arguments). For example, this could be used to open up a new window using `XtVaAppCreateShell()`.

Progress Meters

A plugin that performs a lengthy calculation may wish to inform the user of the progress of the computations. The following routines let a plugin control a progress meter like the one displayed when one of the *AFNI* "`Write`" buttons is used to output a dataset:

`PLUTO_popup_meter(plint)`	Creates the progress meter atop the titlebar of the plugin's user interface window.
`PLUTO_popdown_meter(plint)`	Destroys the progress meter. In any case, this will be executed when the plugin `call_func` returns control to *AFNI*.
`PLUTO_set_meter(plint,perc)`	Sets the progress meter to `perc`% complete, where `perc` is an `int` from 0 to 100 (inclusive). The meter is initially at `perc=0`.

An example of the usage of the meter routines is in `plug_stats.c`.

2D Images

3D or 3D+time "images" are handled by creating new datasets and passing them back to *AFNI*. A plugin may wish to display a single 2D image. This can be done by creating an `MRI_IMAGE struct` (discussed earlier in §4, in the context of timeseries choosers). The atomic type of an `MRI_IMAGE` can be any of the *AFNI* atomic types: `byte`, `short`, `float`, or `complex`—see §2.4. The following code fragment shows how to create an image of `shorts` and get a pointer to the image data array:

```
MRI_IMAGE * shim ;
short * shar ;
shim = mri_new( nx , ny , MRI_short ) ;  /* dimensions are nx X ny */
shar = mri_data_pointer( shim ) ;
```

Pixel (i, j) is stored in `shar[`i`+`j`*nx]` for i = `0..nx-1` and j = `0..ny-1`. By default, the image pixels are square. If the pixels are rectangular, then setting the `float` elements `shim->dx` and `shim->dy` to the x and y dimensions of each pixel is necessary. If you wish to control the string displayed in the titlebar of the image window, then use a call like

```
mri_add_name( "Titlebar String" , shim ) ;
```

Once the image has been created, then it is the plugin's responsibility to fill the image array. When that is done, the image can be popped up with the routine

```
void * handle ;
handle = PLUTO_popup_image( NULL , shim ) ;
```

The variable `handle` is a value that identifies the window in which the image has been opened.

`PLUTO_popup_image` makes a copy of the image, so that there is no need to keep the original, unless it is going to be reused (perhaps to display another image). When it is no longer needed by the plugin, the `MRI_IMAGE struct` should be destroyed with the function call `mri_free(shim)`. This will recover the memory `malloc`-ed to store the image data array and control information.

The `handle` can be used to display another image in place of the first one. For example, suppose that after the above code has been executed, the plugin alters the `shar` array (of course, in this case the `shim struct` should not have be freed). It can then force the redisplay of this modified image by

```
PLUTO_popup_image( handle , shim ) ;
```

That is, if the first argument to `PLUTO_popup_image` is `NULL`, a new window is opened and a new `handle` is returned. If the first argument is an old `handle`, then the image in the old window will be replaced with the new image.

The image window can be closed from within the plugin by the call

```
PLUTO_popdown_image( handle ) ;
```

In addition, the user might close the image window at any time. A plugin can detect this by using the call

```
ii = PLUTO_popup_open( handle ) ;
```

If `ii==1`, then the window to which `handle` refers is still open; if `ii==0`, then that window has been closed. It is still valid to use `handle` to display into that window. *AFNI* will reopen the window, if necessary. If the plugin wishes to close a window and free up all the memory associated with `handle` (just two pointers), then the appropriate macro call is

```
PLUTO_popkill_image( handle ) ;
```

The only difference between this and `PLUTO_popdown_image` is that the memory associated with `handle` is freed, and that `handle` itself will be set to `NULL`.

5.3 Sending a Function to *AFNI*

AFNI maintains three lists of `transformation functions`. These are functions that the user can invoke to modify the display of images or graphs. The three categories of functions are

0D Functions	Functions that perform point transformations of `float` arrays; that is, the i^{th} output point only depends on the i^{th} input point.
1D Functions	Functions that take as input a 1D `float` array and overwrite it in place with new set of values. The outputs may depend on the data array in an arbitrary fashion (unlike the 0D case).
2D Functions	Functions that take as input a 2D `float` array and overwrite it in place with new set of values. The outputs may depend on the data array in an arbitrary fashion (unlike the 0D case).

The 0D and 1D functions can be applied to graphs of 3D+time datasets (from the `Opt` menu in the graphing windows). The 0D and 2D functions can be applied to images (from the `Disp` menu in the image windows).

One application of a plugin that does not create any user interfaces would be a set of functions to be registered with *AFNI*. The `PLUGIN_init` routine would just perform the necessary registrations, and then return `NULL`. These functions would then appear on the appropriate menus, but no `Plugin` menu item would appear for the user to invoke.

0D Functions

A sample 0D function is

```
void ssqrt_func( int num , float * vec )
{
   int ii ;
   double val ;

   if( num <= 0 || vec == NULL ) return ;

   for( ii=0 ; ii < num ; ii++ ){
      val = sqrt(fabs(vec[ii])) ;              /* will be nonnegative */
      vec[ii] = (vec[ii] >= 0.0) ? val : -val ; /* output sign = input sign */
   }
}
```

This function takes the "signed square root" of each input point. The number of input values is the first argument `num`. The second argument is the pointer to the array of input values. Note that this array is modified in place. In a 0D function, the output value of `vec[ii]` should only depend on the input `vec[ii]` and not on any other values in the `vec` array. More general transformations of 1D arrays should be registered as 1D functions.

This function already exists in *AFNI*. If it did not, and it was defined in a plugin, then the plugin could register it with *AFNI* with the call

```
   PLUTO_register_0D_function( "SSqrt" , ssqrt_func ) ;
```

Here, `"SSqrt"` is the string that will be used to identify this function on the menus in which it will appear.

1D Functions

A sample 1D function is

```
void median3_func( int num , double to,double dt, float * vec )
{
   int ii ;
   float aa,bb,cc ;

   bb = vec[0] ; cc = vec[1] ;
   for( ii=1 ; ii < num-1 ; ii++ ){
      aa = bb ; bb = cc ; cc = vec[ii+1] ;
      vec[ii] = MEDIAN(aa,bb,cc) ;         /* see mrilib.h */
   }
}
```

This function computes the median-of-3 filter of the input array `vec`. The input `num` is the number of points in the `vec` array. The input `to` is the time-value at the first point `vec[0]`;

`dt` is the time spacing between `vec` points. (These middle two inputs are not used in this example.) Registration of this function with *AFNI* would be accomplished via

```
PLUTO_register_1D_function( "Median3" , median3_func ) ;
```

The sample plugin `plug_lsqfit.c` has an example of creating and registering a 1D transformation function. This function is controlled by the user parameters in the interface window.

2D Functions

A sample 2D function is given at the end of `imseq.c`. Because of its length, only its prototype is given here:

```
void median9_box_func( int nx , int ny , double dx, double dy, float * ar ) ;
```

The inputs are

- `nx` Number of pixels in the x-direction.
- `ny` Number of pixels in the y-direction.
- `dx` Spacing between pixels in the x-direction
- `dy` Spacing between pixels in the y-direction
- `ar` Pointer to `nx*ny floats`; the (i,j) pixel is in `ar[`i`+`j`*nx]` for i = `0..nx-1` and j = `0..ny-1`

Registration of this function with *AFNI* would be accomplished via

```
PLUTO_register_2D_function( "Median9" , median9_box_func ) ;
```

5.4 Sending a Time Series to *AFNI*

It is also possible for a plugin to create a time series and send it to *AFNI*. This time series will then be available from the time series choosers, which can (for example) be used to select the reference function for the *AFNI* internal `FIM` operation.

A time series is just a 2D `MRI_IMAGE` that was discussed above. The first dimension (`nx`) is "time". The second dimension (`ny`) is the number of time series vectors stored. For many purposes, `ny=1` makes the most sense.

The code fragment below outlines the process:

```
MRI_IMAGE * tsim ;
float * tsar ;
tsim = mri_new( nx , ny , MRI_float ) ;
tsar = mri_data_pointer( tsim ) ;
   /*** fill up tsar array appropriately ***/
PLUTO_register_timeseries( "Name" , tsim ) ;
mri_free( tsim ) ;
```

Note that the image can be freed after it is registered with *AFNI*, since *AFNI* will make a copy. The sample plugin `plug_lsqfit.c` has an example of creating and registering a time series which is defined by user parameters (passed in through the interface control window).

6 Call for Ideas

Although a great deal of work has gone into this version of the plugin software (`afni_plugin.c` and `afni_plugin.h` contain nearly 4000 lines of C), it is clearly just a start. Further development will depend on the needs of plugin developers. This is very much a work-in-progress, and will remain so into the indefinite future. I welcome feedback, especially from those that have actually tried to develop useful plugins.

A portrait of the
author as a young man.