NAME

get_ivec_param, get_int_param, get_real_param, get_str_param,
get_param, get_param_nc, sget_param, do_ivec, fixup_ivec,
gft_set_single, gft_set_multi,
gft_out, gft_out_brief, gft_out_bbox, gft_out_full, gft_out_set_bbox,
gft_outm, gft_outm_brief, gft_outm_bbox, gft_outm_full,
gft_read_brief, gft_read_full, gft_read_shape, gft_read_rank, gft_read_name,
gft_close, gft_close_all,
gft_read_id_gf, gft_read_id_int, gft_read_id_float, gft_read_id_str,
gft_write_id_gf, gft_write_id_int, gft_write_id_float, gft_write_id_str,
gft_read_id_str_p, gft_read_id_int_p, gft_read_id_float_p,
gft_read_2idata, gft_read_1idata, gft_read_idata,
gft_write_id_str_p, gft_write_id_int_p, gft_write_id_float_p,
gft_write_2idata, gft_write_1idata, gft_write_idata

RNPL I/O routines

SYNOPSIS

```
#include <bbhutil.h>

int get_ivec_param(const char *p_file, const char *name,
                   int *p, int size);
int get_int_param(const char *p_file, const char *name,
                  int *p, int size);
int get_real_param(const char *p_file, const char *name,
                   double *p, int size);
int get_str_param(const char *p_file, const char *name,
                  char **p, int size);
int sget_ivec_param(const char *string, const char *name,
                    int *p, int size);
int sget_int_param(const char *string, const char *name,
                   int *p, int size);
int sget_real_param(const char *string, const char *name,
                    double *p, int size);
int sget_str_param(const char *string, const char *name,
                   char **p, int size);

int get_param(const char *p_file, const char *name,
              char *type, int size, void *p);
int get_param_nc(const char *p_file, const char *name, int def,
                 char *type, int size, void *p);
int sget_param(char *string, const char *name,
               char *type, int size, void *p, int cs);

int do_ivec(const int it, const int niter, int *ivec);
void fixup_ivec(const int min, const int max, const int lev, int *ivec);

void gft_set_single(const char *nm);
```

```
void gft_set_multi(void);

int gft_out(const char *func_name, double time, int *shape, int rank,
            double *data);
int gft_out_brief(const char *func_name, double time, int *shape, int rank,
                  double *data);
int gft_out_bbox(const char *func_name, double time, int *shape, int rank,
                 double *coords, double *data);
int gft_out_full(const char *func_name, double time, int *shape, char *cnames,
                 int rank, double *coords, double *data);
int gft_out_set_bbox(double *coords, int rank);

int gft_outm(const char *func_name, double *time, int *shape, const int nt,
             int *rank, double *data);
int gft_outm_brief(const char *func_name, double *time, int *shape, const int nt,
                   int *rank, double *data);
int gft_outm_bbox(const char *func_name, double *time, int *shape, const int nt,
                  int *rank, double *coords, double *data);
int gft_outm_full(const char *func_name, double *time, int *shape, char **cnames,
                  const int nt, int *rank, double *coords, double *data);

int gft_read_brief(const char *gf_name, int level, double *data);
int gft_read_full(const char *gf_name, int level, int *shape, char *cnames,
                  int rank, double *time, double *coords, double *data);
int gft_read_shape(const char *gf_name, const int level, int *shape);
int gft_read_rank(const char *gf_name, const int level, int *rank);
int gft_read_name(const char *file_name, const int n, char *name);

int gft_close(const char *nm);
int gft_close_all(void);

int gft_read_id_gf(const char *gfname, int *shape, int *rank, double *data);
int gft_read_id_int(const char *pname, int *param, int nparam);
int gft_read_id_float(const char *pname, double *param, int nparam);
int gft_read_id_str(const char *pname, char **param, int nparam);

int gft_write_id_gf(char *gfname, int *shape, int rank, double *data);
int gft_write_id_int(char *pname, int *param, int nparam);
int gft_write_id_float(char *pname, double *param, int nparam);
int gft_write_id_str(char *pname, char **param, int nparam);

int gft_read_id_str_p(const char *file_name, const char *param_name,
                      char **param, int nparam);
int gft_read_id_int_p(const char *file_name, const char *param_name,
                      int *param, int nparam);
int gft_read_id_float_p(const char *file_name, const char *param_name,
                        int *param, int nparam);
int gft_read_2idata(const char *file_name, const char *func_name,
                    int *shape, int rank, double *datanm1, double *datan);
int gft_read_1idata(const char *file_name, const char *func_name,
                    int *shape, int rank, double *datan);
```

```
int gft_read_idata(const char *file_name, const char *func_name,
                   int *shape, int rank, double *data);

int gft_write_id_str_p(const char *file_name, const char *param_name,
                       char **param, int nparam);
int gft_write_id_int_p(const char *file_name, const char *param_name,
                       int *param, int nparam);
int gft_write_id_float_p(const char *file_name, const char *param_name,
                         double *param, int nparam);
int gft_write_2idata(const char *file_name, const char *func_name,
                     int *shape, int rank, double *datanm1, double *datan);
int gft_write_1idata(const char *file_name, const char *func_name,
                     int *shape, int rank, double *datan);
int gft_write_idata(const char *file_name, const char *func_name,
                    int *shape, int rank, double *data);
```

ENVIRONMENT

Some of the BBHUtil routines make use of environment variables. The important variables are: BBHHOST, JSERHOST, ISOHOST, BBHHDF, BBHSDF, and BBHSV. BBHHDF, BBHSDF, and BBHSV are simply set or unset. BBHHOST is set to the IP address (or name) of a machine which is running a server which accepts SDF data. JSERHOST is set to the IP address or name of a machine running the scivis visualization server. ISOHOST is set to the machine running the iso-surface generator for visualizing 3D data with scivis.

By default, all routines (except parameter fetching) use SDF files. To use HDF files instead, set BBHHDF. As noted below, this has no effect on the initial data reading and writing routines. Also, the _outm routines do not support HDF.

If you wish to send data to a visualization server, set BBHHOST. If you also wish to send data to an SDF server, set BBHSDF. If instead, you wish to send to scivis BBHSV.

DESCRIPTION

BBHUtil provides a set of C and FORTRAN callable functions which read and write the data files which are read and written by rnpl generated programs. These include parameter files, initial data files, and evolution data files.

Parameter Files

A parameter file is an ASCII file containing arbitrary text along with lines of the form name := value, where name is the name of a parameter and value is the parameter's value. There are four types of parameters, integer, floating point, string, and ivec.

Integer, floating point, and string parameters are all straightforward. Here are some examples of how they would appear in a parameter file:

```
intp := 10
intv := [ 1 2 3 6 7 ]
floatp := 1
floatp2 := 10.1
floatv := [ 1 2.3 5.0 7.2 ]
stringp := "name.sdf"
stringv := [ "string1" "string 2" "string3_" ]
```

The routines for fetching these types of parameters are get_int_param, get_real_param, and get_str_param. The first argument is the name of the parameter file. The second is the name of the parameter. The fourth parameter is an array of ints, doubles, or strings, and the last is the number of elements in the array. If the parameter in the file has fewer elements than the routine asks for, an error will result.

An ivec is an index vector. One use of an ivec is for output control. In a parameter file, an ivec would appear as:

```
ivec1 := *
ivec2 := *-*/10
ivec3 := 1,7,9,10-17/2,18-27,30-*/10
```

An ivec looks like v or v-v or v-v/n, where v is either an integer or * and n is an integer. An ivec can also be any comma-separated sequence of such. The way an ivec is interpreted, depends on how the ivec is being used. In the examples above, if we were using the ivecs for output control, the would be interpreted as follows:

ivec1 says to output at every time step. ivec2 says to output every tenth time step starting at the first time and continuing to the last. ivec3 says to output at time steps 1, 7, and 9, then every other time step from 10 to 17, every time step from 18 to 27, and every tenth time step from 30 to the last time step.

Internally, an ivec is represented by an integer array. The routine for fetching ivecs is get_ivec_param. The size parameter tells the length of this array. If it is too short to handle the ivec in the parameter file, an error will result.

Once an ivec is fetched, there are two routines for manipulating it: fixup_ivec and do_ivec. The arguments to fixup_ivec are the smallest value the ivec can hold, the largest value, the convergence level, and the ivec itself. The min and max values are used for replacing the *'s (if any) in the ivec. In the examples above, these would be the first and last iteration. The convergence level is useful for allowing the same ivec to be interpreted differently at different resolutions. For instance, the ivec

```
output := *-10/4,21
```

would remain unchanged for convergence level 0, and would become

```
output := *-20/8,42
```

for convergence level 1. Basically, all numbers get multiplied by 2^lev.

The routines sget_str_param, sget_int_param, sget_real_param, and sget_ivec_param, work the same as those described above, except they read from a string instead of a file. The first argument is a string of the form name := value, not a file name.

The remaining parameter routines are not callable from FORTRAN. get_param and get_param_nc take the same arguments. These are the name of the parameter file, the name of the parameter, a string representing the parameter type ("long", "double", "string", or "ivec"), the number of elements expected in the parameter, and a pointer to the parameter. The difference between these routines is that get_param is case-sensitive and get_param_nc is not. Note that when getting an ivec, the current size of the ivec should be passed to the routine, not the number of elements expected.

sget_param works the same as these routines except it looks for the parameter in a string instead of a file. Hence the first argument is a string instead of a file name. Also, sget_param takes an extra integer argument at the end. If this is 1, the routine is case-sensitive, if it is 0, the routine is not.

All routines return 0 if there is some sort of syntax error with the input or a memory error. They return -1 if the desired parameter wasn't found in the file or string, and 1 if the parameter was correctly read.

Below are some example calls. Especially note how the strings are passed. Also, the ivec must be dynamically allocated, not allocated from the stack.

```
int i1,*i2,ret;
double f1,*f2;
char *s1,**s2;
int *iv;

/* must allocate memory for arrays */

ret=get_param("p_file","nx","long",1,(void *)&i1);
ret=get_param("p_file","bl","long",5,(void *)i2);
ret=get_param("p_file","m","double",1,(void *)&f1);
ret=get_param("p_file","v","double",1,(void *)f2);
ret=get_param("p_file","tag","string",1,(void *)&s1);
ret=get_param("p_file","coms","string",1,(void *)s2);
ret=get_param("p_file","output","ivec",5,iv);
```

Data Files

The I/O routines have multiple targets. The targets come in two basic types and are chosen with environment varibles (see above). The first type of target is direct-to-server. Data is sent using TCP/IP to a server which is running either locally or on a remote machine. Currently, three servers are supported: scivis, vs, and explorer. The second type of target is a data file. Grid function data is written to or read from a disk file. Most of the routines support both HDF and SDF files (see below), while a few only support SDF. File I/O is done in two modes: single file and multi file. In single file mode, all data is written to or read from a single file. This file can have an arbitrary name, i.e. the name passed into the function is used unchanged. To enter single file mode, call gft_set_single and pass the name of the file. To enter multi file mode, call gft_set_multi.

Most of the I/O routines take the grid function name as a parameter. In multi file mode, one file is created for each grid function. The file names are derived from the grid function name in the following way: alphanumeric characters, _, ; and / are kept and all other characters are killed. The file name will be given the extension .hdf if BBHHDF is set, or .sdf otherwise. If the grid function name already has the appropriate extension, another will not be added.

gft_out, gft_out_brief will output the grid function named func_name. Time gives the time at which the grid function had the values given in data. The rank and shape provide the size of the grid function. The coordinates will be named x,y,z and the coordinates are given a default bounding box $[-1.0, 1.0]$.

gft_out_bbox performs the same function as gft_out. In addition, the coordinate values are taken from the bounding box coords. Coords is a one dimensional array the first two values of which provide the min and max for the first coordinate, the next two values provide the min and max for the second coordinate, etc. In general, the minimum for the nth coordinate is stored in position $2(n-1)$ and the maximum is in position $2(n-1)+1$.

gft_out_full works as the above except the coordinate names are given explicitly in cnames and the coordinate values are given explicitly in coords. The first shape[0] values of coords are the values of the first coordinate, the next shape[1] values are for the second coordinate, etc. cnames is a string which contains the coordinate names separated by a '|'. For a 3D grid function, cnames would be something like "x|y|z".

gft_outm, gft_outm_brief, gft_outm_bbox, and gft_outm_full are for outputting multiple data sets with one call. This is not terribly useful when output is directed to a file, but is very useful when sending to a visualization server because it greatly reduces handshaking and thus speeds transmission. The parameter nt specifies the number of data sets being sent in the call. The time parameter is a vector of length nt which contains the time of each data set. Rank is a vector of length nt which holds the rank of each data set. The shape parameter has length equal to the sum of the elements of rank. The first rank[0] elements of shape are the shape of the first data set, the next rank[1] elements ar the shape of the second data set, etc. That is, shape is the concatenation of all the individual shapes. Similarly, coords is the concatenation of the bounding boxes for gft_outm_bbox and the coordinates for gft_outm_full. Cnames is an array of strings with the names of the coordinates of the first data set followed by the names of the coordinates of the second data set, etc. The data parameter is just the concatenation of all the data sets. Note: these routines do not support HDF.

The default bounding box (for gft_out, gft_outm, gft_out_brief, and gft_outm_brief) can be set by calling gft_out_set_bbox.

gft_read_brief returns the grid function named gf_name. Level runs from 1 to the number of time levels stored in the file.

gft_read_full takes the grid function name, desired time level, and rank and returns complete information including the grid function, time, shape, coordinate names, and coordinate values. Storage for the returned values must be preallocated. The coordinate names are packed (see gft_out_full above). Note: these read routines may not behave as expected in single file mode.

gft_read_shape reads the shape of the levelth data set in the file containing the grid function gf_name.

gft_read_rank reads the rank of the levelth data set in the file containing the grid function gf_name.

gft_read_name reads the name of the nth data set in the file file_name.

gft_close closes the data file whose name is passed as a parameter.

gft_close_all closes all open data files. Note: if using HDF files, this routine *must* be called.

gft_read_id_int, gft_read_id_float, and gft_read_id_str are for reading parameters from SDF initial data files. The routines take the parameter name (pname), preallocated storage for the parameter (param), and the number of elements in param (nparam). The value(s) of the parameter are returned in param.

gft_read_id_gf reads a grid function from an SDF initial data file. It takes the grid function name (gfname) and returns the shape, rank, and values (data) of the grid function.

gft_write_id_int, gft_write_id_float, gft_write_id_str, and gft_write_id_gf work like their read counterparts.

Note: gft_set_single must be called with the initial data file name before the above initial data routines are used. Also, the routines are sequential, so the values must be read in the order in which they were written.

The following routines support reading and writing to HDF initial data files. RNPL generated programs now exclusively use SDF initial data files. It is strongly suggested that these routines not be used, however, they are supported for historical reasons.

gft_read_id_str_p, gft_read_id_int_p, and gft_read_id_float_p are for reading parameters from HDF initial data files. The routines take the file name, parameter name, and number of parameters (array size) and return the parameter values. Sufficient storage must be preallocated and passed in.

gft_read_2idata reads two time levels of initial data from an HDF initial data file. File_name is the name of the initial data file. Func_name is the name of the grid function. Shape and rank contain the size of the data sets. Datan is the data set with the later time, and datanm1 is the data set at the earlier time. Storage must be preallocated.

gft_read_1idata works the same way except for grid functions with only one time level of initial data.

gft_read_idata will read one time level of data from an HDF initial data file for a function with any number of time levels. Func_name must carry the time level information as well as the function name. For instance if the grid function name is "Phi" then the first time level (earliest) would be requested with a func_name of "Phi[0]" while the next would be "Phi[1]" and so on.

gft_write_id_str_p, gft_write_id_int_p, and gft_write_id_float_p work like their read counter parts, except they write data to an HDF initial data file.

gft_write_2idata, gft_write_1idata, and gft_write_idata work like their read counter parts, except they write data to an HDF initial data file.

All (non void) functions return 1 upon success and 0 on failure. Error messages are sent to stderr. The routines take care of opening or creating files as necessary.

SDF Description and Format

SDF is a new file format for storing grid function data. SDF files have the following properties: portability–files written on one machine can be read on another, speed– SDF I/O is very fast, small memory requirements, compactness–there is little overhead beyond the grid data, and flexibility–SDF files can store uniform and non-uniform grid data, as well as output from adaptive mesh refinement codes. Further, SDF files can be concatenated to produce a valid SDF file, making them good for parallel I/O.

Note: the following file format description is subject to change. Please do not attempt to read and write SDF files directly. Use the library routines.

Each SDF file consists of a series of data sets. All data is either an 8 bit character, or a 64 bit IEEE floating point number. Data is stored in network (big-endian) order. On machines which don't use IEEE floating point (some Crays for instance), the data must be converted to IEEE before writing and after reading (this is done automatically by the library). On little-endian machines (Intel), the floating point numbers must be byte-swapped (this is done automatically by the library).

Each data set is divided into a header and a body. The header consists of all the fixed length data. It is defined as:

```
typedef struct sdfh {
  double time;
  double version;
  double rank;
  double dsize;
  double csize;
  double pnlen;
  double cnlen;
  double tglen;
} bbh_sdf_header;
```

The body contains all the variable length data and is defined as:

```
typedef struct sdfb {
  char *pname;
  char *cnames;
  double *bbox;
  double *shape;
  char *tag;
  double *c;
  double *d;
} bbh_sdf_body;
```

The sizes for each of the body fields are provided in the header.

FILES

bbhutil.h	Header file for BBHUtil routines.
libbbhutil.a	Library file.

SEE ALSO

rnpl(1)

The RNPL Reference Manual
The RNPL User's Guide