Datamanip Toolbox Manual

Chapter 3

Library Function Descriptions

Chapter 3 - Library Function Descriptions

A. Introduction

This chapter contains the man3 pages for the function calls in the kdatamanip and ksegops libraries that are contained in the Datamanip toolbox. These man3 pages are grouped into two library sections. Within each library, there are two categories of functions, each presented in its own section. The first section contains the library calls that are associated with the kroutines in the Datamanip toolbox, and the second section contains utility functions. The man3 pages in each section are alphabetically ordered by function name.

ALL programs that utilize these function calls MUST include the statement:

```
#include <datamanip.h>
```

A.1. The kdatamanip Library

The kdatamanip library (libkdmanip.a) contains general data manipulation operators including pointwise arithmetic, statistics calculations, data conversion, histogram, data organization, and size operators. All kdatamanip functions are written to support the relationships defined by the polymorphic data model. This data model provides for general 5-dimensional data (width, height, depth, time, elements), with any combination of value, map, mask, location, and time data segments. The kdatamanip operators are implemented using the polymorphic data services, which is a part of the Khoros 2.0 kappserv library.

A.2. The ksegops Library

The ksegops library (libksegops.a) contains general data segment operators. These functions do not enforce the relationships defined by the polymorphic data model. Operations include moving, copying, inserting and deleting segments, data conversion, ASCII and raw data import, and data object comparison. The ksegops operators are implemented using the data management services, which is a part of the Khoros 2.0 kdms library.

A.3. Side Effects

The function calls in kdatamanip and ksegops should have no side effects on the source objects passed into them, and should only affect those segments and attributes of the destination object that the operation *needs* to modify. For example, if a destination object is passed in that already has value, mask, and geometry data, and a multiplication operation is performed on the source value data, the destination mask and geometry data will not be modified, and will remain intact.

Each function call frees all memory that has been allocated within the call, and closes all temporary objects that were opened within the call. The function call will *not* free memory or close objects that were passed into it.

A.4. DISCLAIMER — WARNING

The kdatamanip and ksegops libraries are in their first prototype stages. The kdatamanip library is also in the process of being updated to fully support the polymorphic data model. Therefore, the behavior and API of these functions may change in the future.

A.5. On-Line Access to man3 Pages

Any library function man3 page can be accessed by typing the following command.

```
% kman <function_name>
```

The keywords (-k) option in kman allows the user to provide a keyword string for searching. When invoked with the *-k* option, kman will perform the search and provide a list of the names and short descriptions of functions found by the search.

```
% kman -k "keyword string"
```

B. kdatamanip Library Functions

B.1. kroutine Function Calls in the kdatamanip Library

B.1.1. lkarith1() — *perform single operand arithmetic on data object*

Synopsis

```
int lkarith1(
   kobject src_obj,
   char    *function,
   kobject dst_obj)
```

Input Arguments

`src_obj`
: the input object to be processed

`function`
: the function to be applied

Output Arguments

`dst_obj`
: the output object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

The following is the library routine that performs a unary function on a data object. The routine takes an input data object, a function name, and an output data object in which to store the result of the function.

The supported functions are:

abs	Output is Absolute Value of input
sqrt	Output is Square Root of input
cbrt	Output is Cube Root of input
neg	Output is Negative of input
recip	Output is Reciprocal of input
not	Bitwise NOT operation
clear	Bitwise CLEAR - set all valid bits to 0
set	Bitwise SET - set all valid bits to 1
sin	Output is the sine() of the input
asin	Output is the arc sine() of the input
cos	Output is the cosine() of the input
acos	Output is arc cosine() of the input
tan	Output is tangent() of the input
atan	Output is arc tangent() of the input
sinh	Output is hyperbolic sine() of the input
asinh	Output is hyperbolic arc sine() of the input
cosh	Output is hyperbolic cosine() of the input
acosh	Output is arc hyperbolic cosine() of the input
tanh	Output is hyperbolic tangent() of the input
atanh	Output is hyperbolic tangent() of the input
sinc	Output is sin(x)/x of the input
ceil	Output is ceiling of the input
floor	Output is floor of the input
trunc	Output is interger truncate of the input
fract	Output is fractional part of the input
lgamma	Output is the lgamma function of the input
erfc	Output is 1.0-erf(x)
erf	Output is error function of x where erf(x)=2/sqrt(pi)*integralfrom0

Therefore if lkarith1() were called with:

```
lkarith1(i, "sin", o);
```

then the following would be performed:

```
o = sin(i);
```

B.1.2. lkarith2() — *perform pointwise dual operand arithmetic on data object(s)*

Synopsis

```
int lkarith2 (
   kobject src1,
   kobject src2,
   double  real,
   double  imag,
   char    *function,
   int     use_subpos,
   kobject dest)
```

Input Arguments

`src1`
: the input object to be processed

`src2`
: the second input object (if NULL then uses the constants "real" and "imag")

`real`
: real value to be instead of the second data object

`imag`
: imaginary value to be instead of the second data object

`function`
: the function to be applied

`use_subpos`
: if TRUE, offset src2 by its subobject position

Output Arguments

`dest`
: the output object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

The lkarith2 library routine performs an arithmetic operation on two source data objects. The routine takes two input data objects (or a single data object and a complex constant), a function name, and a destination data object in which to store the result of the function.

The supported functions are:

add	dest = src1 + (src2 or val)
sub	dest = src1 - (src2 or val)
subfrom	dest = (src2 or val) - src1
mult	dest = src1 * (src2 or val)
div	dest = src1 / (src2 or val)

divinto	dest = (src2 or val) / src1
pow	dest = src1 raised to power of (src2 or val)
absdiff	dest = | src1 - (src2 or val) |
mod	dest = remainder (src1/(src2 or val))
atan2	dest = atan(src1,(src2 or val)) = atan(src1/(src2 or val))
hypot	dest = hypot(src1,src2)=sqrt(src1*src1+src2*src2)
minimum	dest = lower value between src1 and (src2 or val)
maximum	dest = higher value between src1 and (src2 or val)
ldexp	dest = ldexp(src1,(src2 or val)) = src1 * 2**(src2 or val)

Where val is defined by the real and imag parameters passed into lkarith2.

Therefore if lkarith2() were called with two input data objects:

```
lkarith2(src1, src2, 0.0, 0.0, NULL, "add", dest);
```

then the following would be performed:

```
dest = src1 + src2
```

If lkarith2 were called with a single data object and a constant:

```
lkarith2(src1, NULL, 10.0, 0.0, NULL, "add", dest);
```

then the following would be performed:

```
dest = src1 + 10.0
```

If the use_subpos parameter is TRUE, the subobject position attribute of src2 is applied as an offset before the final destination object size is determined, and before the arithmetic operation is performed. The subobject position of src1 is not applied. If the use_subpos parameter is TRUE, both objects are aligned at (0,0,0,0,0).

If either source has mask data, the destination will have a mask that is the logical and of the source masks. Masks are padded with 1.

If a single source object is supplied, and it has map data, then the operation is performed on the map. If both source objects are supplied with combinations of map and value data, then the objects are operated on as described in the kapu_upgrade_objects() man page.

If the first source object has location or time data, they will be transferred to the destination object. If the object is resized, uniform location attributes will not be modified. If the location data is curvilinear or rectilinear, padded may occur using the src1 object's location/time pad attributes.

B.1.3. lkbessel() — *compute bessel function of data object*

Synopsis

```
int lkbessel (
   kobject i,
   int     fn_type,
   int     order,
   kobject o)
```

Input Arguments

`i`
: the input object to be processed

`fn_type`
: bessel fn. of the 1st/2nd kind

`order`
: order of the bessel fn.

Output Arguments

`o`
: the output object

Returns

TRUE (1) on success, FALSE (0) on failure

Description

Library Routine for kbessel. The following is the library routine that performs the bessel function on a data object. The routine takes an input data object, user-specified function kind, user-specified function order, and an output data object in which to store the result of the function.

Therefore if lkbessel() were called with:

lkbessel(i, 1, 0, o)

o = J0(i)

and if it were called with:

lkbessel(i, 2, 1, o)

o = Y1(i)

B.1.4. lkbitwise() — *perform dual operand bitwise operations on data object(s)*

Synopsis

```
int lkbitwise (
   kobject i1,
   kobject i2,
   long    sval,
   char    *function,
   kobject o)
```

Input Arguments

`i1`
: the input object to be processed

`i2`
: the second input object (if NULL then uses the constant "sval")

`sval`
: the constant to be instead of the second data object

`function`
: the function to be applied

Output Arguments

`o`
: the output object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

The following is the library routine that performs a binary bitwise operation on two data objects. The routine takes two input data object (or a single data object and a constant), a function name, and an output data object in which to store the result of the function.

The supported functions are:

and	o = i1 & (i2 or sval)
or	o = i1 | (i2 or sval)
xor	o = i1 ^ (i2 or sval)
nand	o = (~i1) | (~(i2 or sval))
nor	o = (~i1) & (~(i2 or sval))
andrev	o = i1 & (~(i2 or sval))
andinv	o = (~i1) & (i2 or sval)
orrev	o = i1 | (~(i2 or sval))
orinv	o = (~i1) | (i2 or sval)
rs	o = i1 >> by (i2 or sval)
ls	o = i1 << by (i2 or sval)

srs	Signed Right Shift by (i2 or sval)
sls	Signed Left Shift by (i2 or sval)
rr	Right Rotate by (i2 or sval)
lr	Left Rotate by (i2 or sval)

Therefore if lkbitwise() were called with two input data objects:

lkbitwise(i1, i2, 0, "and", o);

then the following would be performed:

o = i1 & i2

If lkbitwise were called with a single data object and a constant:

lkbitwise(i1, NULL, 10, "and", o);

then the following would be performed:

o = i1 & 10

B.1.5. lkblend() — *blend two data objects using the specified ratio*

Synopsis

```
int lkblend(kobject src1_obj,
            kobject src2_obj,
            kobject gate_obj,
            kobject ratio_obj,
            double  ratio,
            kobject dest_obj)
```

Input Arguments

`src1_obj`
first source object to be blended

`src2_obj`
second source object to be blended

`gate_obj`
optional operation gating object

`ratio_obj`
optional object specifying blending ratios

`ratio`
value of the blending ratio

Output Arguments

`dest_obj`

destination object to be written into

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

lkblend blends two objects in the specified ratio. The blending ratio of the objects can be specifed as a constant or an input object can be used for supplying the ratios. The constant is used only if the ratio object is NULL. An optional input object can be used to specify regions in which the operation is not to be performed. The value segment of the gating object is used to determine if the blending on a point is to be gated.

B.1.6. lkclip() — *clip or threshold values in the data object*

Synopsis

```
int lkclip (
   kobject src_obj,
   kobject lc_obj,
   kobject uc_obj,
   double  lval,
   double  uval,
   double  fval,
   double  tval,
   int     l_flag,
   int     u_flag,
   int     use_false,
   int     use_true,
   int     op,
   kobject dst_obj)
```

Input Arguments

`src_obj`
: the source object to be processed

`lc_obj`
: input object containing lower cutoff values

`uc_obj`
: input object containing upper cutoff values

`lval`
: lower clip/threshold value

`uval`
: upper clip/threshold value

`fval`
: value assigned to points that are outside of the clip/threshold range (ie. values less than lower cutoff and/or greater than upper cutoff)

`tval`
: value assigned to points that are within the clip/ threshold range (ie. values greater than or equal than the lower cutoff and less than or equal to the upper cutoff)

`l_flag`
: if set, use lower cutoff value (lval or lc_obj)

`u_flag`
: if set, use upper cutoff value (uval or uc_obj)

`use_false`
: if set, use fval as lower output value foR clip

`use_true`
: if set, use tval as upper output value for clip

`op`
: operation: clip = 0, threshold = 1

Output Arguments

`dst_obj`
: the destination object

Returns

TRUE (1) on success, FALSE (0) on failure

Description

Library Routine for kclip This function clips or thresholds the values in the source object, src_obj and stores the result in the destination object, dst_obj.

Upper and/or lower cutoff values can be specified as a combination of the input constants (lval, uval) and the objects (lc_obj, uc_obj). l_flag and u_flag are flags that define which cutoffs are to be used. If no cutoffs are flagged, lkclip prints an error message and retuns FALSE. If a valid cutoff object exists, it will be used instead of the corresponding constant value.

The operation flag (op) defines whether the data should be clipped (op=0) or thresholded (op = 1).

Clipping: When clipping, if the use_false and/or use_true flags are set, the fval and/or tval passed in will be assigned as the lower result and the upper result for resulting clipped data. Otherwise, if use_false is FALSE, any data clipped to the lower cutoff is set to the lower clipping value (lower result = lc or lc_obj) instead of fval. Likewise, if use_tval is FALSE, data clipped to the upper cutoff is set to the upper clipping value (upper result = uc or uc_obj). Clipping results are assigned according to the following rules.

Clipping Assingnment Rules: 1. When lower cutoff < upper cutoff If the data value is less than the lower cutoff, set the output value to "lower result". If the data value is greater than the upper cutoff, set the output value to "upper result". Otherwise, the data value remains the same.

2. When upper cutoff < lower cutoff If the data value is less than the upper cutoff and it is greater than the lower cutoff, set output value to the "lower result". Otherwise, the data value remains the same.

Thresholding: Thresholding results are assigned according to the following rules.

Thresholding Assingnment Rules: 1. When lower cutoff < upper cutoff If the data value is less than the lower cutoff, or it is greater than the upper cutoff, set the output value to fval. Otherwise, set the output value to tval.

2. When upper cutoff < lower cutoff If the data value is less than the lower cutoff and it is greater than the upper cutoff, set output value to fval. Otherwise, set the output value to tval.

Map Data: If all source objects contain map data and no value data, the operation is done directly on the map data. If only a souce object is passed in, and it contains map data, the operation is performed directly on the map. Otherwise, if map and value data exist, the data is mapped before the operation is performed. If the destination object does not have the necessary map and/or value segments, they are created for it.

Location & Time Data: If more than one source object is supplied (src_obj, lc_obj, uc_obj), and any of them contains time or location data, lkclip will fail at this time.

Mask Data: Data is masked according to the masking mode attribute of the source data objects. The resulting object mask is the logical AND of any input masks that exist. If the destination object does not have a mask, one is created for it.

B.1.7. lkcmplx() — *perform a unary complex function on data object*

Synopsis

```
int lkcmplx(
   kobject i,
   char    *function,
   kobject o)
```

Input Arguments

`i`
: the input object to be processed

`function`
: the function to be applied

Output Arguments

`o`
: the output object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

The following is the library routine that performs a unary complex function on a data object. The routine takes an input data object, a function name, and an output data object in which to store the result of the function.

The supported functions are:

conj	Output is Complex Conjugate of the input
p2r	Output is Polar Coordinates to Rectangular Coordinates of the input
r2p	Output is Rectangular Coordinates to Polar Coordinates of the input

Therefore if lkcmplx() were called with:

lkcmplx(i, "p2r", o);

then the following would be performed:

o = polar_to_rect(i);

B.1.8. lkcompare() — *perform comparison operation on data object(s)*

Synopsis

```
int lkcompare (
   kobject i1,
   kobject i2,
   double  val,
   char    *function,
   double  tval,
   double  fval,
   double  tol,
   kobject o)
```

Input Arguments

`i1`
: the input object to be processed

`i2`
: the second input object (if NULL then uses the constant "val")

`val`
: the constant to be instead of the second data object

`function`
: the function to be applied

`tval`
: the value to be assigned if comparison is TRUE

`fval`
: the value to be assigned if comparison is FALSE

`tol`
: the plus/minus tolerence in which to do the comparison (not functional yet)

Output Arguments

o
: the output object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

The following is the library routine that performs a comparison operation on two data objects. The routine takes two input data object (or a single data object and a constant), a function name, and an output data object in which to store the result of the function.

The supported functions are:

eq	o = i1 == (i2 or val)
ne	o = i1 != (i2 or val)
gt	o = i1 > (i2 or val)
ge	o = i1 >= (i2 or val)
lt	o = i1 < (i2 or val)
le	o = i1 <= (i2 or val)

Therefore if lkcompare() were called with two input data objects:

lkcompare(i1, i2, 0.0, 1.0, 0.0, "eq", o);

then the following would be performed:

o = (i1 == i2) ? 1.0 : 0.0

If lkcompare were called with a single data object and a constant:

lkcompare(i1, NULL, 10.0, 1.0, 0.0, "eq", o);

then the following would be performed:

o = (i1 == 10.0) ? 1.0 : 0.0

B.1.9. lkextract() — *extract rectangular region from data object*

Synopsis

```
int lkextract(
   kobject src_obj,
   int     w_off,
   int     h_off,
```

```
int    d_off,
int    t_off,
int    e_off,
int    w_size,
int    h_size,
int    d_size,
int    t_size,
int    e_size,
int    subpos,
kobject dst_obj)
```

Input Arguments

`src_obj`
the source data object to be processed

`w_off`
origin of region on width dimension

`h_off`
origin of region on height dimension

`d_off`
origin of region on depth dimension

`t_off`
origin of region on time dimension

`e_off`
origin of region on elemens dimension

`w_size`
region size on width dimension

`h_size`
region size on height dimension

`d_size`
region size on depth dimension

`t_size`
region size on time dimension

`e_size`
region size on elements dimension

`subpos`
if TRUE, set dst_obj KPDS_SUBOBJECT_POSITION attribute to (w_off,h_off,d_off,t_off,e_off). Otherwise set it to (0,0,0,0,0).

Output Arguments

`dst_obj`
the destination data object

Returns

TRUE (1) on success, FALSE (0) on failure

Description

Library Routine for kextract.

Lkextract is based on implicit indexing. This means that if location or time data exist, the extract operation is not done in terms an interpretation of the location/time data values, but in terms of the implicit indexing of these data (which is specified by the width, height, depth, time, elements indices of the polymorphic data model).

Lkextract extracts from the source data object (src_obj) a region specified by the w_size, h_size, d_size, t_size, and e_size parameters and places the extracted data into the corresponding segment(s) in the destination object (dst_obj). The region will be extracted starting at the point defined by (w_off, h_off, d_off, t_off, e_off). This point will be the origin (0,0,0,0,0) of the destination object. The size attributes of the destination object's segment(s) are set according to the <dimension>_size input parameters.

If the source data object has mask, location, time, and/or value data segments, the specified region is extracted from each existing segment in the source and placed in the corresponding segment of the destination object. If the segment does not already exist in the destination object, it will be created.

If the offset (defined by w_off, h_off, d_off, t_off, e_off) and/or the extracted region size, fall outside the indexing bounds, or size, of the data segments, the object segments will be padded with the values pad value attribute of the source object, which can be set prior to calling lkextract.

If the destination object contains segments not present in the source object, they will automatically be resized according to the size input parameters (to maintain the integrety of the data with respect to the polymorphic data model). The offset will be the origin (0,0,0,0,0), not the. offset values passed in. If padding of these segments is necessary, the padding values will be determined from the pad attribute of the destination object, which can be set prior to calling lkextract.

If the subpos parameter is TRUE, lkextract sets the KPDS_SUBOBJECT_POSITION attribute of the destination object to the subregion origin coordinates (w_off, h_off, d_off, t_off, e_off). If subpos is FALSE, the KPDS_SUBOBJECT_POSITION is set to (0,0,0,0,0).

Error Checks: The size parameters passed in must be greater than zero, lkextract will fail if they are not. Lkextract will also fail if the source and destination objects passed in are not valid. Any offset value is legal.

Restrictions

Only operates on value, mask, location, and time segments.

B.1.10. lkfft() — *perform multidimensional FFT on data object*

Synopsis

```
int lkfft (
        kobject in_obj,
            int     center,
            int     w,
            int     h,
            int     d,
```

```
            int     t,
            int     e,
            int     scale_factor,
            int     direction,
            int     datatype,
            kobject cmplx_out)
```

Returns

TRUE (1) on success, FALSE (0) on failure

Description

Performs a multidimensional FFT on the input object. The FFT can be requested on up to 5 dimensions. The direction of the FFT can be specified. The same routine performs both the forward and the inverse FFT. The result of the FFT operation can be scaled by either a factor of 1 (no scaling), 1/N or 1/sqrt(N) where N is the size of the FFT. Centering on the FFT/IFFT is also available as an option. lkfft performs the FFT operation on all the data primitives in the object. For example, if the FFT is requested along the width for a 5-D object then it is taken along all width primitives in the object. The size of the dimension for which FFT is requested must be a power of two and greater than 1. The output data object is always complex. It can either be single precision complex or double precision complex depending on the value of the parameter datatype.

B.1.11. lkflip() — *reflect data in object along specified dimensions*

Synopsis

```
int lkflip( kobject src,
            int num_dim,
            long *dim_list,
            kobject dest )
```

Input Arguments

`num_dim`
: No. of dimensions to operate on

`dim_list`
: list of dimensions on which to operate

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Library Routine for kflip. This routine performs data reflection in one, all, or any combination of directions. The directions supported are: KWIDTH, KHEIGHT, KDEPTH, KTIME, KELEMENTS

B.1.12. lkgate() — *write data 1 or data 2 to destination depending on gate*

Synopsis

```
int lkgate(
    kobject src1_obj,
    kobject src2_obj,
    kobject gate_obj,
    double  gate_val,
    int     gate_logic,
    kobject dest_obj)
```

Input Arguments

`src1_obj`
: First source object

`src2_obj`
: Second source object

`gate_obj`
: Gating object which determines the output

`gate_val`
: Value defining when to perform gating operation

`gate_logic`
: Gating logic: if gating element = gate_val, (0) use src1_obj, (1) use src2_obj

Output Arguments

`dest_obj`
: Destination object which contains results

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

lkgate takes two source objects and writes out source object 1 or source object 2 to the destination object depending on the value of the gating object. A non-zero gating value implies that the values taken on by source object 2 are propogated to the destination object. Source object 1 values are used if a zero gating value is found.

If the source objects have map data, the data is mapped before the gating process. If the destination object has a map, it will be destroyed.

If any of the source objects have masks, the destination object will also have a map. If none exists on the destination, one will be created. The resulting mask will be the result of logically ANDing the source masks.

B.1.13. lkgconst() — *set value segment of a data object to a constant*

Synopsis

```
int lkgconst(
   int     wsize,
   int     hsize,
   int     dsize,
   int     tsize,
   int     esize,
   double  real,
   double  imaginary,
   int     datatype,
   kobject object)
```

Input Arguments

`wsize`
The wsize of the object being created

`hsize`
the height of the object being created

`dsize`
the depth of the object being created

`tsize`
the size of the objects time dimension

`esize`
the objects element dimension size

`real`
the real component of the constant value

`imaginary`
imaginary component of the constant value

`datatype`
datatype value (see above)

Output Arguments

`object`
the output object

Returns

TRUE (1) on success, FALSE (0) on failure

Description

Library Routine for kgconst. lkgconst creates a constant valued value segment for the destination data object. The resulting data size is specified by the wsize, hsize, dsize, tsize, and esize parameters. The data type of the value segment is specified by the datatype parameter, and can be KBIT, KBYTE, KUBYTE, KSHORT, KUSHORT, KINT, KUINT, KLONG, KULONG, KFLOAT, KDOUBLE, KCOMPLEX and KDCOMPLEX. The value segment will be set to the real and imaginary constant levels specified by the "real" and "imaginary" parameters.

Side Effects

If the value segment already exists, then all data in the segment will be overwritten with the constant value in the data type specified in the function call and the size will be changed to the specfied wsize, hsize, dsize, tsize and esize.

B.1.14. lkgenloc() — *generate location data*

Synopsis

```
int lkgenloc(
   double  xbegin,
   double  ybegin,
   double  zbegin,
   double  xend,
   double  yend,
   double  zend,
   int     grid,
   int     wid,
   int     hgt,
   int     dep,
   int     datatype,
   kobject dst_obj)
```

Input Arguments

`xbegin`
: the x-axis begin point

`ybegin`
: the y-axis begin point

`zbegin`
: the z-axis begin point

`xend`
: the x-axis end point

`yend`
: the y-axis end point

`zend`
: the z-axis end point

`wid`
: width size to use for the constructed data

`hgt`
: height size to use for the constructed data

`dep`
: depth size to use for the constructed data

`datatype`
: data type to use for the constructed data

Output Arguments

`dst_obj`
: the output object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine will generate explicit location data for a specified object.

This routine will create a location segment of the specified width, height, and depth using the specified data type. The dimension size of the created data is assumed to be three to correspond to (x,y,z). It is assumed that the object initially does not have an explicit location segment.

The explicit location data is specified by three sets of begin and end values, which together compose explicit corner markers for volume in three space (x,y,z). Typically, the x-axis corresponds uniformly to the implicit width dimension on the polymorphic data model, with the y-axis and z-axis corresponding to implicit height, and depth, respectively.

Data is generated uniformly from those specified corner markers, however the explicitness of the generated data can be specified by the grid parameter. A KUNIFORM grid will dictate that the generated data consist simply of setting the uniform corner markers on the output data set. A KRECTILINEAR grid will dictate that the generated data consist of three arrays of locations, an x-array, a y-array, and a z-array, with the data within those arrays changing uniformly from the specified (x,y,z) begin and end values. A KCURVILINEAR grid will dictate that the generated data consist of a four dimensional array of location data. Essentially, this data set is a three dimensional array of (x,y,z) points.

The data is generated and stored as one block, so generating large location data sets are subject to the memory limitations of your machine.

Restrictions

Destination object must not contain location data

B.1.15. lkgnoise() — *generate noise in value segment of data object*

Synopsis

```
int lkgnoise(
            int width,
        int height,
            int depth,
            int t,
            int elements,
            char *noise_type,
```

```
              int datatype,
              double noise_spec1,
              double noise_spec2,
              kobject object)
```

Input Arguments

`width`
: The width of the object being created

`height`
: the height of the object being created

`depth`
: the depth of the object being created

`t`
: the size of the objects time dimension

`elements`
: the objects element dimension size

`noise_type`
: a string defining the noise type to be generated

`datatype`
: an int defining the datatype

`noise_spec1`
: double specifying noise characteristic

`noise_spec2`
: double specifying noise characteristic

Output Arguments

`object`
: the output object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Library Routine for kgnoise. lkgnoise creates noisy value segment for a data object. The value segment will be set to the dimension of width* height * depth * time * elements that were specified. The data type of the value segment is specified by the datatype int and can be KBIT, KBYTE, KUBYTE, KSHORT, KUSHORT, KINT, KUINT, KLONG, KULONG, KFLOAT, KDOUBLE, KCOMPLEX and KDCOMPLEX. The value segment will be set to the noisy data having the distribution specified by the noise_type string. The noise distribution can be Gaussian, Rayleigh, Uniform, Exponential or Poisson. The parameters associated with a particular distribution can be specifed by noise_apec1 and noise_spec2 (optional depending on the distribution).

B.1.16. lkgsin() — *generate sinusoidal value data*

Synopsis

```
int lkgsin(int w_size, int h_size,
           int d_size, int t_size,
           int e_size, int sum,
           double *ampl,double w_num,
           double h_num, double d_num, double t_num,
           double e_num, double w_phase,
           double h_phase, double d_phase,
           double t_phase, double e_phase,
           int data_type, kobject out_obj)
```

Input Arguments

`w_size`
: The width of the out_obj being created

`h_size`
: the height of the out_obj being created

`d_size`
: the depth of the out_obj being created

`t_size`
: the size of the out_objs time dimension

`e_size`
: the out_objs element dimension size

`sum`
: flag, if TRUE sine is calculated along each dimension else sine is calculated on the sum of the coordinates

`ampl`
: pointer to amplitude of the sine being generated. If sum is TRUE, an amplitude is needed for each dimension else only one amplitude value is needed. wnum - number of sine cycles along width hnum - number of sine cycles along height dnum - number of sine cycles along depth tnum - number of sine cycles along time enum - number of sine cycles along elements

`w_phase`
: phase offset along width

`h_phase`
: phase offset along height

`d_phase`
: phase offset along depth

`t_phase`
: phase offset along time

`e_phase`
: phase offset along elements

`data_type`
: datatype of the output out_obj

Output Arguments

`out_obj`

the output out_obj

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Library Routine for kgsin. lkgsin creates a value segment filled with sinusoidal values for the data out_obj. The value segment will be set to the the dimension of w_size * h_size * d_size * t_size * e_size that were specified. The data type of the value segment is specified by the datatype parameter, and can be KBIT, KBYTE, KUBYTE, KSHORT, KUSHORT, KINT, KUINT, KLONG, KULONG, KFLOAT, KDOUBLE, KCOMPLEX and KDCOMPLEX. The value segment will contain either a sum of 5 sinusoids, one for each dimension or the sine of the sum of the coordinates at each point depending on the value of the parameter "sum".

B.1.17. lkhisto() — *compute histograms for data objects*

Synopsis

```
int lkhisto(
  kobject in_obj,
  kobject gate_obj,
  int w, int h, int d, int t, int e,
  double min,
  double binwidth,
  int bins,
  int oob,
  kobject out_obj)
```

Input Arguments

`in_obj`
: object to be histogrammed

`gate_obj`
: optional gate object (ROI) w,h,d,t,

`e`
: flags that, if set, indicate that the processing unit for independent histograms should include the given axis

`min`
: minimum side of most negative bin

`binwidth`
: width of each bin

`bins`
: number of bins

`oob`
: out-of-bounds flag; append two extra bins to the histogram to count those values that fall out of the histogram bounds

Output Arguments

`out_obj`
contains histograms for the processing units in the data

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

lkisto computes histogram information for data objects. The unit of data to be used to compute each histogram can be selected to lie along any combination of the axes 5D data model.

For example, it is possible to compute a histogram for each HxD plane in a data set comprised of a sequence of multiband volumes.

Output histograms are stored as columns in the output object, one column per unit of data, in WHDTE order.

The output data type is UNSIGNED LONG.

If a gating object is supplied, then only input data that has a corresponding non-zero value in the gating object is histogrammed.

The out-of-bounds flags can be used to catch any counts that lie outside of the histogram array.

B.1.18. lkhistops() — *perform histogram equalization and stretching*

Synopsis

```
int lkhistops(
  kobject in_obj,
  char *function,
  int invertflag,
  int w, int h, int d, int t, int e,
  kobject out_obj)
```

Input Arguments

`in_obj`
object to be processed

`function`
a string containing either "equalize" or "stretch"

`invertflag`
if non-zero, invert greylevels after processing w h d t

`e`
flags that, if set, indicate that the processing unit for independent histograms should include the given axis

Output Arguments

`out_obj`
: output object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Perform histogram equalization and stretching with optional inversion. Equalization occurs using the standard algorithm.

Stretching is done between the most negative data value (mapping to the most negative value possible for the input data storage type) and the most positive data value (mapping to the most positive value possible for the input data storage type).

No span limits are applied, as was the case in the K1.5 vhstr routine.

Restrictions

KBYTE, KUBYTE, KSHORT, KUSHORT data only

B.1.19. lkimpulse() — *generate impulse value data*

Synopsis

```
int lkimpulse(int w, int h,
              int d, int t,
              int e, int wsp,
              int hsp, int dsp,
              int tsp, int esp,
              int w_num, int h_num,
              int d_num, int t_num,
              int e_num, int w_off,
              int h_off, int d_off,
              int t_off, int e_off,
              double imp_rval, double imp_ival,
              double back_rval, double back_ival,
              int data_type, kobject dst_obj)
```

Input Arguments

`w`
: The width of the object being created

`h`
: the height of the object being created

`d`

the depth of the object being created

`t`
: the size of the objects time dimension

`e`
: the objects element dimension size

`wsp`
: the spacing between two impulses in width direction

`hsp`
: the spacing between two impulses in height direction

`dsp`
: the spacing between two impulses in depth direction

`tsp`
: the spacing between two impulses in time direction

`esp`
: the spacing between two impulses in elements dir

`w_num`
: Number of impulses along width

`h_num`
: Number of impulses along height

`d_num`
: Number of impulses along depth

`t_num`
: Number of impulses along time dimension

`e_num`
: Number of impulses along elements dimension woff - Location of first impulse in the width direction hoff - Location of first impulse in the height direction doff - Location of first impulse in the depth direction toff - Location of first impulse in the time direction eoff - Location of first impulse in the elements direction

`imp_rval`
: Real value assigned to the impulse

`imp_ival`
: Imaginary value assigned to the impulse

`back_rval`
: Real value assigned to the background

`back_ival`
: Imaginary value assigned to the background

`data_type`
: Data type of the output object

Output Arguments

`dst_obj`
: output object containing impulse data

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine creates impulses at locations controlled by the various parameters passed to it. The value segment is created if one does not already exist and is set to size w*h*d*t*e. The datatype of the value segment is set according to the variable data_type. It can be either one of bit, byte, unsigned byte, short, unsigned short, integer, unsigned integer, long, unsigned long, float, double, complex and double complex. The value assumed by the impulse generated can be specified by imp_rval and imp_ival which denote its real and imaginary component respectively. The value assumed by the background can also be specified by the back_rval and back_ival parameters which denote it's real and imaginary components. Three parameters are available to control the location of the impulses. The spacing between two impulses, the starting point of the first impulse and the number of impulses can be specified for each of the five dimensions in the object.

Side Effects

If the value segment already exists, then all data will be overwritten with the impulse data of the specified datatype. The size of the object will also be changed to the size requested in the function call.

B.1.20. lkkmeans() — *Perform K-means clustering*

Synopsis

```
int lkkmeans(kobject in_obj1,
             kobject in_obj2,
             int n,
             int k,
             int map_flag,
             int spectrum_flag,
             kobject out_obj1,
             kobject out_obj2,
             kobject out_obj3,
             kobject out_obj4,
             kfile *statsfile
)
```

Input Arguments

`in_obj1`
: input object for clustering

`in_obj2`
: input object for initial cluster centers (if non-NULL)

`n`
: maximum number of iterations to be performed

`k`
: number of clusters to generate

`map_flag`
: (1 : generate map on out_obj1 containing the cluster centers, 0 : do not generate map on out_obj1)

`spectrum_flag`
: (1 : generate special map on out_obj1 containing information required by spectrum [overrides map_flag setting] , 0 : do not generate spectrum compatible map on out_obj1)

Output Arguments

`out_obj1`
: output object to contain cluster numbers in value segment

`out_obj2`
: if non-NULL, contains cluster centers stored row by row in the value segment

`out_obj3`
: if non-NULL, contains cluster variances stored row by row in the value segment

`out_obj4`
: if non-NULL, contains cluster vector count stored row by row in the value segment

`statsfile`
: if non-NULL, an ASCII record of statistics is written to the associated file

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

lkmeans accepts an input data object containing vectors of equal size and performs the K-means clustering algorithm on the vectors. The length of each vector is determined by the elements (E) dimension of the input data object.

The K-means algorithm is based on minimization of the sum of the squared distances from all points in a cluster to a cluster center. The user chooses K initial cluster centers and the input vectors are iteratively distributed among the K cluster domains. New cluster centers are computed from these results, such that the sum of the squared distances from all points in a cluster to the new cluster center is minimized.

Although the K-means algorithm does not really converge (in a continuous space), it may converge in a discrete space or a practical upper limit can be chosen for convergence. The user has the option of specifying the maximum number of iterations using the n argument.

There are two ways to specify the initial cluster centers. If the in_obj2 argument is non-NULL, then the cluster centers are read from that object. The vectors are assumed to be stored along the E direction. Only the first K centers (as specified by the -k argument) will be read. If the in_obj2 argument is NULL, then the first K vectors in the in_obj1 object will be used as the initial cluster centers.

It should be noted that it is possible to specify an initial cluster that lies at a sufficient distance from all input vectors that it will have no vectors assigned to it during a pass of the K-means algorithm. If this happens, lkkmeans will reinitialize the value of that cluster to the mean value of a moving pair of the existing cluster centers, thus avoiding degeneracy.

If map_flag and spectrum_flag are false, the out_obj1 object will contain a value segment specifying the cluster number to which each input vector was assigned. If map_flag is true, then a map segment will also be generated. The final cluster centers will be stored row by row in the map. The values in the value segment can be interpreted as "pointing" to a particular row in the map where the associated cluster for that input vector can be found.

If spectrum_flag is true, then the out_obj1 output object will contain a special map segment (regardless of map_flag) with additional information required for use with the spectrum program in the most general sense. Here, not only the cluster centers are stored, but so are the number of vectors associated with each cluster and the packed upper triangle of the covariance matrix for each cluster. See the spectrum manual for additional information on how this data is used and the additional capabilities that become available when the extra data supplied by setting spectrum_flag to true.

If out_obj2 is non-NULL, the associated data object will contain the cluster centers (mean vectors), stored row by row in the value segment. The dimensions of the value segment will be WxHx1x1x1 where W is the number of elements in each mean vector and W is the number of clusters.

If out_obj3 is non-NULL, the associated data object will contain the cluster variances, stored row by row in the value segment. The dimensions of the value segment will be WxHx1x1x1 where W is the number of elements in each vector of variances and W is the number of clusters.

If out_obj4 is non-NULL, the associated data object will contain the cluster membership counts, stored row by row in the value segment. The dimensions of the value segment will be 1xHx1x1x1 where H is the number of clusters. The membership counts simply state the number of vectors that were present in the input object that were assigned to each of the final cluster centers.

If statsfile is non-NULL, the an ASCII file will be written to the associated file descriptor which will contain statistics obtained during the execution of lkkmeans. The output includes the following information: Total Number of K-means Iterations, Total Number of Clusters, Number of Vectors Per Cluster, Cluster Center Values, Cluster Center Variance Values, Trace of Covariance Matrix.

Results obtained by the K-means algorithm can be influenced by the number and choice of initial cluster centers and the geometrical properties of the data.

For the out_obj1, out_obj2, and out_obj3 output objects, the data will be stored as type KDOUBLE. For the out_obj4 output object, the data will be stored as type KINT. For the out_obj1 output object, the value data will be stored as type KSHORT and all map data as type KDOUBLE.

lkkmeans was converted from the K1.5 lkmeans routine, which was written by Tom Sauer and Charlie Gage, with assistance and ideas from Dr. Don Hush, University of New Mexico, Dept. of EECE. Significant modifications were made to the algorithm by Scott Wilson during conversion to K2.

Restrictions

Complex data types are not supported.

B.1.21. lklinearop() — *perform linear operation (convolution/correlation) on data object*

Synopsis

```
int lklinearop( kobject src,
                kobject kernel,
                int filtering_type,
                double real_pad_value,
                double img_pad_value,
                int  *offset,
```

```
                    int  size_type,
                    int  reflection,
                    int  num_dim,
                    long *dim_list,
                    int  *sub_size,
                    int  *sub_pos,
                    int  upcast,
                    kobject dest )
```

Input Arguments

`filtering_type`
: linear (0) or circular (1)

`real_pad_value`
: real value for padding

`img_pad_value`
: imaginary value for padding

`offset`
: array of kernel offset values (location of hot-spot), these 5 values (w,h,d,t,e) override those stored in the kernel object. If offset is NULL, then the default values stored with the kernel object are used.

`size_type`
: truncate (0) or extend the result of the operation (output object) to include full range of support (1)

`reflection`
: reflect (for convolution) (1) or do not reflect (0) the kernel

`num_dim`
: number of directions to operate on

`dim_list`
: array of directions on which to operate (use KWIDTH,KHEIGHT,KDEPTH,KTIME,KELEMENTS)

`sub_size`
: array specifying the sub-object size, these 5 values (w,h,d,t,e) specify a subregion of the input object on which to operate

`sub_pos`
: array specifying the sub-object position, these 5 values specify the origin of the sub-object in (w,h,d,t,e)

`upcast`
: force upcast of input data type to KDOUBLE or KDCOMPLEX before processing (1), or allow upcast of data types to the best fitting type in {KUBYTE, KLONG, KDOUBLE, KDCOMPLEX} (0) (look out for loss of precision)

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Library routine for klinearop. Performs convolution or correlation operations on one, all, or any combination of directions. The directions supported are: width, height, depth, time, elements. Operates on sub-objects as well. The sub-objects parameters suppored are: sub-object position and size in the supported direction.

B.1.22. lklocxform() — *transform location data*

Synopsis

```
int lklocxform(
   kobject src_obj,
   float   xrot,
   float   yrot,
   float   zrot,
   float   xtrans,
   float   ytrans,
   float   ztrans,
   float   xscale,
   float   yscale,
   float   zscale,
   kobject dst_obj)
```

Input Arguments

`src_obj`
: the input object with location data

`xrot`
: rotation about the x axis

`yrot`
: rotation about the y axis

`zrot`
: rotation about the z axis

`xtrans`
: translation in the x direction

`ytrans`
: translation in the y direction

`ztrans`
: translation in the z direction

`xscale`
: scaling in the x direction

`yscale`
: scaling in the y direction

`zscale`
: scaling in the z direction

Output Arguments

`dst_obj`
: the output object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine performs a homogeneous coordinate transformation on the location data in the source object, and saves the transformed location data into the destination object.

The source object must have location data, and the destination object must not have location data for this routine to function properly. This routine will create location data of the proper grid type in the destination.

The coordinate transformation can be specfied by nine parameters which specify an x-y-z rotation, an x-y-z translation, and an x-y-z scaling. The rotations are all performed with respect to the (0,0,0) origin. A scaling matrix, a translation matrix, and three rotation matrices are constructed from these specified parameters and composited into a single transformation matrix in the following order :

[1] Scaling
[2] X Rotation
[3] Y Rotation
[4] Z Rotation
[5] Translation

A value of 1.0 should be used for any scaling parameters where no scaling is desired. A value of 0.0 should be used for any rotation parameters where no rotation is desired or for any translation parameters where no translation is desired.

The location points are interpreted to be in three-space for the transformations. Explicit location data located in dimensions higher than three will be lost during the transformation. The transformation will be performed using the datatype KFLOAT.

In general, the grid type of the source object will be preserved in the destination object. For example, uniform location data, when transformed and stored in the destination object, will still be uniform location data. The exception to this rule is rectilinear data, which must be converted to curvilinear data in the event of a rotation. In this case the destination location data will be stored as explicit curvilinear data points.

Reference : Foley & VanDam "Computer Graphics : Principles and Practice", Second Edition, 1990, Chapter 5.

B.1.23. lklogexp() — *perform log or antilog function on data object*

Synopsis

```
int lklogexp (
   kobject i,
   double  base,
   int     offset,
   int     inv,
   kobject o)
```

Input Arguments

`i`
: the input object to be processed

`base`
: the log/anti-log base if function is NULL

`offset`
: whether to subtract/add 1 from the result

Output Arguments

`o`
: the output object

Returns

TRUE (1) on success, FALSE (0) on failure

Description

Library Routine for klogexp The following is the library routine that performs a log/antilog function on a data object. The routine takes an input data object, user specified base, offset flag, and an output data object in which to store the result of the function.

Therefore if lklogexp() were called with:

lklogexp(i, 2.0, TRUE, FALSE, o);

o = log2(i)

Therefore if lklogexp() were called with:

lklogexp(i, KE, FALSE, TRUE, o);

o = e**(i)-1

B.1.24. lkmsquish() — *compress map to one column by means of Average, RMS or MAX*

Synopsis

```
int lkmsquish(kobject in_obj, int map_flag, int type, kobject out_obj)
```

Input Arguments

`in_obj`
: input data object

`map_flag`
: Pull value data thru resultant map (1), or update only the map segment (0)

`type`
: Compute the grey level as the average (1), RMS (2), or MAX (3) of the components of each row in

the existing map

Output Arguments

`out_obj`
output data object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

lkmsquish compresses a map down to one column, effectively converting the map to a "greyscale" map. An example of this operation is encountered when trying to print a pseudocolor RGB image on a B/W laser printer: the RGB color information must be somehow converted to greyscale in order for the printer to know how to dither the image during printing.

lkmsquish is very happy to operate on simple 3-column maps, such as would be encountered when processing a color image in any of the standard coordinate systems (ntscRGB, etc). However, lkmsquish is not limited to 3 column maps - it will operate on maps with any number of cloumns.

The conversion from multiple columns to a single column is done by computing a grey level for each row of the map, one at a time. For type=1, the grey level for a row is the average of the columns in the row. For type=2, the grey level is the RMS value of the columns in the row. For type=3, the grey level is the maximum value found in the colums of the row. Clearly, these schemes are not optimal in any sense, but one of them will usually give an acceptable result.

Ordinarily (map_flag=0), the map is the only part of the input object that is modified. The value data is not touched; it still points to various rows in the map just as before the compression. If map_flag=1, the value data is pulled thru the map and the map segment itself is removed. If mask data is present, then this operation proceeds as described in the man page for kmapdata(1).

Restrictions

Restrictions on data or input as applicable

B.1.25. lknoise() — *introduce noise in source object*

Synopsis

```
int lknoise(
   kobject src_obj,
   char    *function,
   int     add_flag,
   double  spec1,
   double  spec2,
   kobject dst_obj)
```

Input Arguments

`src_obj`
: the input object to be processed

`function`
: type of noise to be introduced

`add_flag`
: indicates whether to add or replace w/noise

`spec1`
: first paramater for distribution

`spec2`
: second paramater for distribution

Output Arguments

`dst_obj`
: the output object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

The following is the library routine that performs the addition of noise to a data object or the replacement of the value segment of a data object with noise. The routine takes an input data object, an eplace oper- optional gating data object, a noise type, specs related to the noise, a flag indicating the add ation, and an output data object in which to store the result. The following types of noises are supported by this routine:

Gaussian Noise	(Mean, Variance)
Rayleigh Noise	(Variance)
Uniform Noise	(Maximum, Minimum)
Poisson Noise	(Amount of time, Variance)
Exponential Noise	(Variance)

The right hand side above indicates the paramaters needed to specify each kind of noise. Therefore if lknoise() were called with:

lknoise(i, NULL, "ray", TRUE, 1.0, 2.0, o);

then the following would be performed:

o = i + Rayleigh Noise of variance 1.0

It may be noted here that the value passed in the second spec (2.0) is ignored since it only 1 value is needed for specifying a Rayleigh distribution.

Side Effects

Destroys the map segment on the destination object if the source object contains both map and value data. Also the size and datatype of the value segment is set to the size and datatype of the the mapped source object.

B.1.26. lknormal() — *normalize regions using minimum & maximum of each region*

Synopsis

```
int lknormal(
   kobject src_obj,
   double  norm_lower,
   double  norm_upper,
   int      apriori_flag,
   double  data_min,
   double  data_max,
   int     w_flag,
   int     h_flag,
   int     d_flag,
   int     t_flag,
   int     e_flag,
   int     whole_flag,
   char    *data_type_string,
   kobject dst_obj)
```

Input Arguments

`src_obj`
: the input data object to be normalized lower_norm - the upper normalization value upper_norm - the upper normalization value

`apriori_flag`
: if TRUE, uses specified data_min and data_max

`data_min`
: predetermined minimum data value

`data_max`
: predetermined maximum data value

`w_flag`
: if TRUE, include width in normalization unit

`h_flag`
: if TRUE, include height in normalization unit

`d_flag`
: if TRUE, include depth in normalization unit

`t_flag`
: if TRUE, include time in normalization unit

`e_flag`
: if TRUE, include elements in normalization unit

`whole_flag`
: if TRUE, normalize whole data set at one time datatype - the data type of the output data object

Output Arguments

`dst_obj`

the output data object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

lknormal normalizes the data within regions to the lower_norm and upper_norm parameters. Regions are defined by TRUE settings of w_flag, h_flag, d_flag, t_flag, e_flag, and whole_flag. The minimum and maximum values in each region are used to define the normalization over that region. If whole_flag is TRUE, the normalization region will be the entire data set, and w_flag, h_flag, d_flag, t_flag, and e_flag settings are overridden. If none of the flags are true, the default is whole_flag.

If the apriori flag is TRUE, the data will not be searched for minimum and maximum values but instead, the data_min and data_max parameters provided will be used. This eliminates a required pass through the data before normalization is performed.

The data type of the output object is specified by the data_type_string parameter, and should be one of the following values:

Data Type	Abbreviation	Data Type
bit	bi	`KBIT`
byte	by	`KBYTE`
unsigned byte	un by	`KUBYTE`
ubyte	uby	`KUBYTE`
short	sh	`KSHORT`
unsigned short	un sh	`KUSHORT`
ushort	ush	`KUSHORT`
integer	in	`KINT`
unsigned integer	un in	`KUINT`
uint	ui	`KUINT`
long	lon	`KLONG`
unsigned long	un lon	`KULONG`
ulong	ul	`KULONG`
float	fl	`KFLOAT`
double	do	`KDOUBLE`
complex	co	`KCOMPLEX`
double complex	do co	`KDCOMPLEX`
dcomplex	dc	`KDCOMPLEX`
Propagate Input Type - will not change the data type		

If the source object (src_obj) contains value data and no map, normalization is performed on the value data. Likewise, if the source object contains map data and no value data, normalization is performed on the map.

If the source object has both map and value data, certain region conditions must be met for lknormal to succeed.

1. The first condition is that the normalization region must be defined by at least width and height.
2. If the map depth, time, and elements dimensions are all one, the normalization region must be the entire data set.
3. If the map depth, time, or elements dimensions are greater than 1, the corresponding dimension does not have to be part of the normalization region.

Masks: Currently, the validity mask is only transferred from the source to the destination. However, if there is a mask, the KPDS_MASKED_VALUE_PRESENTATION is used by kpds. As a future enhancement, invalid data will not be figured into the normalization calculations.

lknormal ignores location, and time data.

Restrictions

The validity mask is simply transferred from the source to the destination. Invalid data will be included in the normalization calculations.

Does not support complex input types yet.

B.1.27. lkorient() — *reorient object to new dimensions*

Synopsis

```
int lkorient (
   kobject src_obj,
   int     *src2dst_map,
   kobject dst_obj)
```

Input Arguments

`src_obj`
: the input object to be processed

`src2dst_map`
: dimension mapping from src_obj to dst_obj

Output Arguments

`dst_obj`
: the output object

Returns

TRUE (1) on success, FALSE (0) on failure

Description

The following is the library routine that performs data reorientation with respect to the 5 polymorphic dimensions, width, height, depth, time, and elements. The routine takes an input data object and a mapping, defined by the src2dst_map array, which specifies how each dimension in the old object will be mapped to the new object. More than one dimension from the input object can be mapped to the same dimension in the output object.

The src2dst_map array is a set of five integers that can have one of the following values:

KORIENT_WIDTH
KORIENT_HEIGHT
KORIENT_DEPTH
KORIENT_TIME
KORIENT_ELEMENTS

src2dst_map[0]

defines where the WIDTH dimension of the source object will be mapped to in the destination object. src2dst_map[1] defines where the HEIGHT dimension of the source object will be mapped to in the destination object. src2dst_map[2] defines where the DEPTH dimension of the source object will be mapped to in the destination object. src2dst_map[3] defines where the TIME dimension of the source object will be mapped to in the destination object. src2dst_map[4] defines where the ELEMENTS dimension of the source object will be mapped to in the destination object.

If the source object has map data, the map data cannot have depth, time, or elements dimensions of greater than one. The map width and height dimensions can be any size.

If the source object has rectilinear or curvilinear location or time data, lkorient will fail. If the source object has uniform location data, lkorient will continue, but the location data will not be modified to reflect the reorientation.

If the object has mask data, it will be reoriented along with the value data.

lkorient will fail if the source object does not contain value data.

B.1.28. lkpad() — *perform the padding of a data object*

Synopsis

```
int lkpad (
   kobject src_obj,
   int     w_size,
   int     h_size,
   int     d_size,
```

```
int     t_size,
int     e_size,
int     w_off,
int     h_off,
int     d_off,
int     t_off,
int     e_off,
double  real,
double  imag,
int      valid,
kobject dst_obj)
```

Input Arguments

`src_obj`
Source data object.

`w_size`
Size of width dimension after padding.

`h_size`
Size of height dimension after padding.

`d_size`
Size of depth dimension after padding.

`t_size`
Size of time dimension after padding.

`e_size`
Size of elements dimension after padding.

`w_off`
Offset of src_obj in width dimension.

`h_off`
Offset of src_obj in height dimension.

`d_off`
Offset of src_obj in depth dimension.

`t_off`
Offset of src_obj in time dimension.

`e_off`
Offset of src_obj in elements dimension.

`real`
Real part of pad value.

`imag`
Imaginary part of pad value.

`valid`
if TRUE, padded data is considered valid

Output Arguments

`dst_obj`
Destination object (padded) data object.

Returns

TRUE (1) on success, FALSE (0) on failure

Description

Library Routine for kpad. lkpad performs the padding of a data object. The offset parameters define where the origin of the source object will lie with respect to the output object. Specifying an offset greater than zero can be used to generate a border around the original data. Negative offset values result in truncation of data lying in the region (0-w_off, 0-h_off, 0-d_off, 0-t_off, 0-e_off) For example, given an offset of (-1,-1,0,0,0), the object will effectively be shifted by -1 in the width and height dimensions, and that data truncated. If the offset plus the original data size is greater than the specified destination size, truncation of the data will occur.

The imag and real input parameters are the pad values used. If the input data object type is not complex, the imaginary pad value will be ignored. For data objects that are not of type double, pad values will be cast so that the output object is the same type as the input object.

Mask data corresponding to padded data will be padded non-zero if the valid flag is TRUE, and padded with zeros if the valid flag is FALSE. Mask data corresponding to the source object will be transferred unaltered.

If curvilinear or rectilinear time or location data exist, it is unclear how to alter the dimensionality, and lkpad will fail. If uniform location data exist lkpad will not fail, but it will not alter the location data in any way. yet.

lkpad does not modify map data - it directly pads the value data. The data is not mapped first.

B.1.29. lkprdata() — *print object data in ASCII format to file*

Synopsis

```
int lkprdata(
   kobject in_obj,
   kfile   *ascii_out,
   int     print_attributes,
   int     print_index,
   int     print_location,
   int     print_time,
   int     print_value,
   int     print_mask,
   int     print_map_index,
   int     print_map,
   int     map_value_data,
   int     replace_flag,
   char    *replace_string)
```

Input Arguments

`in_obj`
: Input data object.

`print_attributes`
: Flag, if set, print object attributes.

`print_index`
Flag, if set, print indices associated with value, mask, location and time segments.
`print_location`
Flag, if set, print location data.
`print_time`
Flag, if set, print time data.
`print_value`
Flag, if set, print value data.
`print_mask`
Flag, if set, print mask data.
`print_map_index`
Flag, if set, print indices associated with map segment.
`print_map`
Flag, if set, print map data.
`map_value_data`
Flag, if set, map value data through map before printing, and print the mapped values.
`replace_flag`
Flag, if set, substitute replace_string for masked values.
`replace_string`
String to substitute for masked values.

Output Arguments

`ascii_out`
Output file for printing ASCII data.

Returns

TRUE (1) on success, FALSE (0) on failure

Description

Library Routine for kprdata. Prints attributes and data and indices associated with each data segment in formatted ASCII to the print device. The flags passed in define what will be printed.

B.1.30. lkreal2cmplx() — *create a complex object from real source object(s)*

Synopsis

```
int lkreal2cmplx(
    kobject real_obj,
    kobject imag_obj,
    double  real_val,
    double  imag_val,
    kobject dest_obj)
```

Input Arguments

`real_obj`

object to be used as real component

`imag_obj`
object to be used as imaginary component

`real_val`
real pad value, and real component if real_obj is NULL

`imag_val`
imaginary pad value, and imaginary component if imag_obj is NULL

Output Arguments

`dest_obj`
resulting complex data object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

lkreal2cmplx generates a complex destination object, dest_obj, containing complex data from the supplied source object(s), real_obj and imag_obj. The complex data type (float or double) is determined by taking the highest source data type.

Single Source Object: If only one source object is valid, the other component of the complex pair will be the real_val or imag_val passed in. If the source object has map data, the operation will be performed on the map only. For the single source case, location, time, and mask data are not needed or modified.

Two Source Objects: If both source objects are valid, and both contain maps and no value data the operation is performed directly on the map. For the case of two sources with no value data, location, time, and mask data are not needed or modified.

If either source object contains both map and value data, then both objects must have value data, and the data will be mapped before processing.

If the source objects are the same size, then location and time data can be ignored.

If the source objects are different sizes, then location and time data are checked, and if they exist, lkreal2cmplx will only continue if the grid type is uniform. The destination size will the maximum of both source sizes. Padding required by real_obj will be set to real_val, and padding required by imag_obj will be set to imag_val.

If either source object has a validity mask, the destination object will get a mask. If both source objects have masks, they will be combined (logical AND) after processing.

lkreal2cmplx will fail if neither source object is valid; if the source objects do not contain value and/or map data; if either source object is complex; or if the destination object is not valid.

B.1.31. lkreplace() — *replace all occurrences of X with Y in data object*

Synopsis

```
int lkreplace(
          kobject src_obj,
                 kobject gate_obj,
                 double  old_val,
                 double  new_val,
                 int     upper_tol_flag,
                 double  upper_tol,
                 int     lower_tol_flag,
                 double  lower_tol,
                 kobject dest_obj)
```

Input Arguments

`src_obj`
: source object whose values are substituted

`gate_obj`
: operation gating object

`old_val`
: value to be replaced

`new_val`
: value to be substituted

`upper_tol_flag`
: flag to indicate validity of upper tolerance

`upper_tol`
: upper tolerance allowed in old_val

`lower_tol_flag`
: flag to indicate validity of lower tolerance

`lower_tol`
: lower tolerance allowed in old_val

Output Arguments

`dest_obj`
: destination object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

lkreplace replaces each occurence of the value old_val in the object with the value new_val. The operations occur on the map segment if one is present in the source object otherwise the operations occur on the value segment.

The tolerance values allow the value that is being replaced to fall within a given range. the upper_tol and lower_tol values will not be used unless the corresponding flags are TRUE.

The gating functionality will be removed in a future release.

Currently, lkreplace explicitly checks the validity mask, and will not perform the operation if the masked data is invalid. This will also be updated in a future release to act more like lkarith1, where the operation will still be performed, but the mask presentation attributes are used.

B.1.32. lkresample() — *resample the value data of an object*

Synopsis

```
int lkresample(
     kobject i,
     double wfactor,
     double hfactor,
     double dfactor,
     double tfactor,
     double efactor,
     int wcenter,
     int hcenter,
     int dcenter,
     int tcenter,
     int ecenter,
     int resize,
     int interpolate,
     kobject o)
```

Input Arguments

`i`
: the input data object to be resampled

`wfactor`
: the width magnification

`hfactor`
: the height magnification

`dfactor`
: the depth magnification

`tfactor`
: the time magnification

`efactor`
: the element magnification

`wcenter`
: the width center position

`hcenter`
: the height center position

`dcenter`
: the depth center position

`tcenter`
: the time center position

`ecenter`
the element center position
`resize`
reset the dimensions to the magnified size
`interpolate`
the interpolation type

Output Arguments

`o`
the output data object to be resampled

Returns

TRUE (1) on success, FALSE (0) on failure

Description

lkresample is the library routine that actually resamples the object data. lkresample magnifies or reduces the source data object (i) by using the selected interpolation method (interpolate). (At this time, only only pixel replication is supported.) Magnification factors greater than 1 magnify the object, and magnification factors between 0 and 1 reduce the object.

Pixel replication and subsampling are done on a floating point indexing basis, where the data point index is obtained by truncating the cumulative floating point index. Therefore, when the calculated replication factor or the subsampling step size is non-integer, the resulting integer step size varies during the process.

Magnification Factors: individual magnification factors are specified for each dimension by the wfactor, hfactor, dfactor, tfactor, and efactor parameters, and must be values greater than zero.

Center of Action: The magnification or reduction is done about a center point that can be specified to be anywhere in the object (even outside of the object). The center of action coordinates are specified for each dimension by the wcenter, hcenter, dcenter, tcenter, and ecenter parameters.

Destination Object Size: The resultant destination object size will be the same as the source object size if the resize flag is FALSE, or it will be resized to fit the magnified or reduced data set if resize flag is TRUE.

If resize is FALSE, and the source object is reduced (magnification factors < 1.0) then unknown pixels are set to zero. Likewise, if the source object is magnified (magnification factor > 1.0) then input data that would be mapped outside of the destination object are lost.

Data Type: The data type of the destination object's value data is the same as the source object's value data type.

If there is map and value data in the source object, and zero order interpolation (pixel replication) is used, the map is transferred to the destination object. If any interpolation algorithm is used, the value data is mapped before the operation can begin and the destination will not have map data. This program fails if the source object lacks both map data and value data.

This program fails if there is mask data and the operation uses anything other than zero order interpolation. In other words, if value data is being replicated or subsample on grid, then the mask can be correspondingly replicated or subsampled. (Bilinear interpolation can not be performed on mask data.)

If explicit location or time data are available, the sampling algorithm can not resample these data yet, and will fail. This will be implemented later ....

B.1.33. lksampline() — *Sample a data object along an arbitrary line*

Synopsis

```
int lksampline(kobject in_obj,
     double wc1,double hc1,double dc1,double tc1,double ec1,
        double wc2,double hc2,double dc2,double tc2,double ec2,
        int n, int nc, kobject out_obj)
```

Input Arguments

`in_obj`
: input data object

`wc1`
: Coordinate of start of sampling line in width dimension

`hc1`
: Coordinate of start of sampling line in height dimension

`dc1`
: Coordinate of start of sampling line in depth dimension

`tc1`
: Coordinate of start of sampling line in time dimension

`ec1`
: Coordinate of start of sampling line in element dimension

`wc2`
: Coordinate of end of sampling line in width dimension

`hc2`
: Coordinate of end of sampling line in height dimension

`dc2`
: Coordinate of end of sampling line in depth dimension

`tc2`
: Coordinate of end of sampling line in time dimension

`ec2`
: Coordinate of end of sampling line in element dimension

`n`
: number of samples between start and end, inclusive of end points

`nc`
: minimum required neighbor count; if zero, then this is computed as 0.5*(2^dim)+1 where dim is the dimensionality of the source object.

Output Arguments

`out_obj`
output data object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

lksampline is used to sample the data in an object along an arbitrary line and at arbitrary intervals. The value at each sample point is obtained by an inverse distance weighting scheme.

The first sample point is precisely at the starting coordinate and the final sample is precisely at the ending coordinate.

The line may traverse any portion of the 5D data space. If a particular sample on the line lies outside the data space then the values will be interpolated using a padding value of zero.

The output object will be of the same data type as the input object. If a map is present in the input object, then the value data is sent through the map before processing. The output data will thus have no map.

Mask data, if present, is used to help control the interpolation. Value points with a corresponding mask value of zero will not be included in the interpolated value. However, a minimum number of neighboring values with a valid mask are required in order to produce a valid output value; non-valid output values are marked by a zero mask. If the mask segment is not present, then all value points in the source object are assumed to be valid.

The actual minimum required neighbor count is determined by the setting of the "nc" argument. If nc is zero, then the minimum required neighbor count will be computed from the dimensionality of the source data set. For nc=0, the minimum required neighbor count is 0.5*(2^dim)+1 where dim is the dimensionality of the source object (1 for a line, 2 for a plane, 3 for a volume, etc). If nc is non-zero, then the minimum required neighbor count is simply nc. Beware of non-intuitive results when interpolating across mask wise non-convex parts of the data object. The default minimum required neighbor count gives "reasonable" answers in these cases. Reducing the minimum required neighbor count can provide answers that are less and less meaningful.

Note that lksampline samples the data as if it were using a delta function. This implies that you may get significantly different results depending on the position of each sample location. You may wish to lowpass the source data before sampling with lksampline to avoid aliasing. This can be particularly important when dealing with "holey data" or when the sampling involves degenerate geometry (such as sampling a plane object with a sampling line that does not lie in the plane).

lksampline is implemented using *point* data access. A point data access is done for each sample in the line. This implies that a large number of samples may require quite some time to compute, particularly if operating on a higher dimension data set.

Restrictions

Restrictions on data or input as applicable

B.1.34. lkshot() — *corrupt data uniformly with spikes*

Synopsis

```
int lkshot(
   kobject src_obj,
   double  ratio,
   double  real_val,
   double  imag_val,
   kobject dst_obj)
```

Input Arguments

`src_obj`
: Source data object

`ratio`
: Percentage of spikes in the image

`real_val`
: Real value of spike

`imag_val`
: Imaginary value of spike

Output Arguments

`dst_obj`
: Destination object containing spiked data

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Library routine for kshot. lkshot corrupts the input data object uniformly with spikes. The value of the spike is specified by real_val and imag_val, which represent the real and imaginary component of the spike. Imaginary value is used only if spikes are to be inserted in a complex object. Ratio specifies the percentage of points in the input objects to be corrupted with shot noise. The location of points to be corrupted by spikes is generated by a uniform random number generator. If the mask value at a point to be corrupted by a spike is 0, the spike is not inserted at that point but it is counted in the number of points corrupted.

Side Effects

Destroys the map segment on the destination object if the source object contains both map and value data. Also the size and datatype of the value segment is set to the size and datatype of the the mapped source object.

B.1.35. lkstats() — *calculate statistics of data object*

Synopsis

```
int lkstats (
   kobject src_obj,
   kobject gate_obj,
   kobject stats_obj,
   unsigned long  active_stats,
   unsigned long  aux_info,
   unsigned long  stats_region,
   kfile          *ascii_file,
   int          ascii_flag)
```

Input Arguments

`src_obj`
: source data object from which stats are calculated.

`gate_obj`
: gating data object

`stats_obj`
: data object in which stat results are stored

`active_stats`
: the bits of this flag indicate whether a particular statistic is to be calculated.

`aux_info`
: the bits of this flag indicate whether a particular piece of auxiliary information is to be recorded in the stats_obj.

`stats_region`
: the bits of this flag specify which dimensions make up each region from which statistics are to be computed.

`ascii_flag`
: if this flag is TRUE, output statistics in formatted ASCII to ascii_file.

Output Arguments

`stats_obj`
: destination object for statistics results

`ascii_file`
: output file for printing ASCII data.

Returns

TRUE (1) on success, FALSE (0) on failure

Description

Library Routine for kstats. For the input src_obj, lkstats calculates the set of statistics defined by the flags set in the active_stats variable. The results are stored in the stats_obj data object. The active_stats flag can be built by or'ing the statistics flags defined in include/kdatamanip/kstats.h. The aux_info variable defines additional, non-statistical information, such as object data dimensions and position of the minimum and maximum values, that may also be recorded in stats_obj. The aux_info flag can be built by or'ing the auxiliary information flags defined in include/kdatamanip/kstats.h.

Data with corresponding mask or gate values of zero (gating defined by gate_obj) are not included in any statistics calculations. If the gating object is different sized than the source object, it will be resized to match the source object, padded with 1.

If the input data object contains a map, the data will be mapped before the statistics are calcualted. All output information will reflect this mapping.

The following statistics are available:

mean = 1/N*sum(x(i)) i=1..N
var = 1/(N-1)*sum((x(i) - mean)**2) i=1..N
stddev = sqrt(var) i=1..N
rms = sqrt(1/N*sum(x(i)**2)) i=1..N
skew = 1/N*sum(((x(i) - mean)/stddev) **3) i=1..N
kurtosis=(1/N*sum(((x(i) - mean)/stddev) **4)) - 3 i=1..N
minimum value
maximum value
minimum value coordinates
maximum value coordinates
sum of all points
sum of all positive points
sum of all negative points
number of contributing points
number of positive contributing points
number of negative contributing points
number of zero-valued contributing points
dimensions of data

Statistics are stored in the stats_obj as double float. Each statistic is stored as an element of a N-D vector defined along the value data "elements" dimension of the stats_obj, where N is the number of statistics defined by the active_stats variable.

The order in which the elements of the statistics vector are stored in the statistics data object is given in the object's comment attribute.

When M sets of statistics are calcualted for M multiple regions of the input data object (defined by the flags set in the stats_region variable), the statistics vectors are stored along the width dimension (width=M, elements=N).

The stats_region variable defines how the data object is sliced along the dimensions for calculation of statistics over sub-regions. For example, if the width and height flags are set in the "stats_region" variable, the statistics for each width-height plane will be calculated separately. For an image object this can be viewed as calculating the statistics across the various bands of the image. The stats_region flag can be built by or'ing the region dimensionality flags defined in include/kdatamanip/kstats.h.

While computing variance, the division factor is set to (N-1) in order to obtain a more accurate estimate. For a more detailed explanation refer to any standard statistics text.

B.1.36. lkwiener() — *perform Wiener filtering (restoration) in Fourier frequency domain*

Synopsis

```
int lkwiener(kobject in_obj, kobject psf_obj, double wiener, kobject out_obj)
```

Input Arguments

`in_obj`
: source data object (FFT) psg_obj - PSF data object (FFT)

`wiener`
: Wiener parameter

Output Arguments

`out_obj`
: filtered output data object (FFT)

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Perform restoration of data in the spatial domain by Wiener filtering in the frequency domain. The idea is to deconvolve the blurring function from the blurred data. A few problems exist in performing this operation, as described in great detail in any good image processing book.

A full implementation of a Wiener filter requires detailed knowledge of the signal and noise power spectra. Many times this information is simply unavailable or is very poorly known. In such situations, it is convenient to assign some constant value to the SNR and use that in the Wiener filter implentation. This is a somewhat extreme restriction of the Wiener filter, but it makes the filter very easy to use and can give much better results than the pseudoinverse filter in lkinverse().

This routine implements the filter described in K.R. Castleman, "Digital Image Processing", Prentice-Hall 1979, ISBN 0-13-212365-7, p281 Eq. 2 with the variable SNR term given as the Weiner parameter constant.

Restrictions

Restrictions on data or input as applicable

B.2. Utility Functions in the kdatamanip Library

B.2.1. kcombine_to_complex() — *combines 2 double arrays into a double complex array*

Synopsis

```
int kcombine_to_complex(kdcomplex **c, int num, double *r, double *i)
```

Description

kcombine_to_complex takes a real and imaginary array of type double and returns a kdcomplex array.

B.2.2. kdata_insert() — *insert a value in memory*

Synopsis

```
int kdata_insert(
   kaddr  data,
   int    num,
   int    datatype,
   double real,
   double imaginary)
```

Input Arguments

`data`
: point to the data to be filled

`num`
: location to be inserted into

`datatype`
: data type to fill with

`real`
: real part of data to fill with

`imaginary`
: imaginary part of data to fill with

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to insert a value into memory. A datatype is used to determine how to interpret the real and imaginary values specified.

C. ksegops Library Functions

C.1. kroutine Function Calls in the ksegops Library

C.1.1. lkconvert() — *convert, offset, and/or scale data in specified segment*

Synopsis

```
int
lkconvert(
   kobject source_object,
   char    *segment_name,
   char    *data_type_string,
   double  scale,
   double  real_offset,
   double  imaginary_offset,
   kobject destination_object)
```

Input Arguments

`source_object`
: the object to get the segment from

`segment_name`
: the segment to convert

`data_type_string`
: data type to convert to

`scale`
: value to scale the data by

`real_offset`
: real value to offset data by

`imaginary_offset`
: imaginary value to offset data by

Output Arguments

`destination_object`
: the resulting object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine will convert, offset and/or scale the segment specified by segment_name of the input object source_object and create a new object destination_object containing the modified segment and all the other segments which will be unmodified. The data type to convert to will be passed in data_type_string and should be one of these values:

Data Type	Abbreviation	Data Type
bit	bi	`KBIT`
byte	by	`KBYTE`
unsigned byte	un by	`KUBYTE`
ubyte	uby	`KUBYTE`
short	sh	`KSHORT`
unsigned short	un sh	`KUSHORT`
ushort	ush	`KUSHORT`
integer	in	`KINT`
unsigned integer	un in	`KUINT`
uint	ui	`KUINT`
long	lon	`KLONG`
unsigned long	un lon	`KULONG`
ulong	ul	`KULONG`
float	fl	`KFLOAT`
double	do	`KDOUBLE`
complex	co	`KCOMPLEX`
double complex	do co	`KDCOMPLEX`
dcomplex	dc	`KDCOMPLEX`
Propagate Input Type - will not change the data type		

The scale value is specified by scale and the offset value by real_offset & imaginary_offset.

C.1.2. lkcpfromval() — *copy data from value into specified segment*

Synopsis

```
int
lkcpfromval(
   kobject source_object,
   char    *segment_name,
   int     dimension,
   int     *index_order,
   kobject destination_object)
```

Input Arguments

`source_object`
  object to copy the value segment from

`segment_name`
  segment name to copy to

`dimension`
  dimension of segment (not yet supported)

`index_order`
  index order of segment (not yet supported)

Output Arguments

`destination_object`
object to put the segment into

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine copies the five dimensional value segment from the source_object to segment specified by segment_name, of the destination_object. The data will be copied into the lowest order dimension of the destination segment, then the next order will be used, and the next, and so forth, until all the value data is copied. If the dimensionality of the destination is less than five, then the last dimension of the destination segment will have all the higher order dimensions rolled into it.

If segment_name is one of the polymorphic segments (KDMS_SEGMENT_VALUE, KDMS_SEGMENT_MASK, KDMS_SEGMENT_MAP, * KDMS_SEGMENT_TIME or KDMS_SEGMENT_LOCATION) than the * segment dimensionalities and index orders specified by the polymorphic model will be used. If the segment is not one of the polymorphic segments, then the dimensionality and index order of the destination segment must be specified.

Only the size and data type attribute will be changed, all others will not be transferred to the destination segment. If the segment does not exist in the destination_object and is not a polymorphic segment the dimension and index order will be set. If it is a polymorphic segment the default polymorphic dimension and index order will be used.

Restrictions

Results guaranteed only if dimension of destination segment is 5. Other dimensions are not tested and may produce poor results.

C.1.3. lkcpseg() — *copy segments from source to destination object*

Synopsis

```
int
lkcpseg(
   kobject source_object,
   char   *segment_list,
   kobject destination_object)
```

Input Arguments

`source_object`
object to copy the segment(s) from

`segment_list`
list of segment to copy

Output Arguments

`destination_object`
object to put the segment(s) into

Returns

TRUE (1) if one copied, FALSE (0) if none copied

Description

This routine will copy from the source_object the segments specified by segment_list into the destination_object. Each segment's attributes and data will be copied into the destination_object. If segment_list is NULL it will copy all the segments in the object. The segment_list is a string that contains each segment name separated by space(s).

```
lkcpseg(source_object, "segment1 segment2",
        destination_object);
```

If the segment does not exist an error message is printed, but will not return until an attempt to copy all the segments specified has been done.

C.1.4. lkcptoval() — *copy data from specified segment into value segment*

Synopsis

```
int
lkcptoval(
   kobject source_object,
   char    *segment_name,
   kobject destination_object)
```

Input Arguments

`source_object`
object to copy the segment from
`segment_name`
segment name for segment to copy

Output Arguments

`destination_object`
object to put the segment into

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine copies a segment from the source_object and put it into the destination_object's value segment. The higher dimensions will be rolled into the elements dimension of the value segment. If the dimension is less than 5, the data will be copied into the lowest order dimension of the value segment first, then the next, etc. and the higher dimensions ignored. If the value segment exists in the source object it will be replaced.

Only the size and data type attribute will be changed in the value segment, all others will not be transferred. If the value does not yet exist, then the dimension and index order of the value segment will be set to the defaults: 5 and width, height, depth, time, and elements. If it does exist the index order will not be changed.

Restrictions

Results guaranteed only if dimension of destination segment is 5. Other dimensions are not tested and may produce poor results.

C.1.5. lkimportasc() — *import ASCII data into specified segment of data object*

Synopsis

```
int
lkimportasc(
   char   *ascii_file,
   kobject source_obj,
   char   *segment_name,
   int     dimension,
   char   *index_order,
   int    *size_vector,
   char   *data_type_string,
   int     start_offset,
   int     skip_factor,
   kobject destination_object)
```

Input Arguments

`ascii_file`
: file name of ASCII file

`source_obj`
: source data object to insert segment in

`segment_name`
: name of segment to create

`dimension`
: dimensionality of segment being created

`index_order`
: index order of segment being created

`size_vector`
: size of segment being created

`data_type_string`

data type of segment being created

`start_offset`

starting point in ascii data to start from

`skip_factor`

skip factor to use while importing ascii data

Output Arguments

`destination_object`

data object to create containing new segment

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine will import an ASCII data file, ascii_file, and insert it into a data object, source_obj, if it is specified. If there is no source object to insert the new segment into, pass NULL for source_obj and the data object specified by the parameter destination_object, will be created. The new segment specified by segment_name will be inserted into this data object. If a source_obj was specified, then all of its data segments and the global object attributes will be copied to the destination.

If the segment_name is one of the Polymorphic segments, the correct dimensionality must be provided :

Segment Name	Dimension
`KDMS_SEGMENT_VALUE`	5
`KDMS_SEGMENT_MAP`	5
`KDMS_SEGMENT_MASK`	5
`KDMS_SEGMENT_LOCATION`	4
`KDMS_SEGMENT_TIME`	1

There are four attributes of the new segment that must be passed in : dimension, index order, size and data type. The number of indicies and sizes specified must be equal to the dimension.

The parameter dimension must be between 0 and 5.

The parameter index_order is a string that contains the indicies of the segment separated by spaces, "elements width height depth time" or "e w h d t". The only allowed values are :

width (w)	mapwidth (mw)
height (h)	mapheight (mh)
depth (d)	mapdepth (md)
time (t)	maptime (mt)
elements (e)	mapelements (me)
dimension (n)	

where the characters in the ()'s are accepted abbreviations.

The parameter size_vector contains the size of the segment to create. All the values must be > 1.

The parameter data_type_string must contain one of the following strings to specify the data type of the segment:

Data Type	Abbreviation	Data Type
bit	bi	KBIT
byte	by	KBYTE
unsigned byte	un by	KUBYTE
ubyte	uby	KUBYTE
short	sh	KSHORT
unsigned short	un sh	KUSHORT
ushort	ush	KUSHORT
integer	in	KINT
unsigned integer	un in	KUINT
uint	ui	KUINT
long	lon	KLONG
unsigned long	un lon	KULONG
ulong	ul	KULONG
float	fl	KFLOAT
double	do	KDOUBLE
complex	co	KCOMPLEX
double complex	do co	KDCOMPLEX
dcomplex	dc	KDCOMPLEX
Propagate Input Type - will not change the data type		

The parameter start_offset is the point in the ASCII source at which to start importing in the ASCII data. All points before this one are ignored.

The parameter skip_factor is the number of points to skip in the ASCII data between reads. If the size of the new segment's lowest index is 3 and the skip factor is 5, then this routine will read in 3 points skip 5 and read in 3 more, etc. until the entire ASCII file read in.

C.1.6. lkimportraw() — *import raw binary data into a data segment*

Synopsis

```
int
lkimportraw(
kobject raw_obj,
kobject add_obj,
int arch,
int type,
char *seg,
int w,
int h,
int d,
```

```
int t,
int e,
kobject out_obj)
```

Input Arguments

`raw_obj`
: raw source data

`add_obj`
: optional object to import the data into

`arch`
: source architecture of the raw data

`type`
: data type of the raw data

`seg`
: string defining which data segment to insert data into w, h, d, t,

`e`
: size of raw data

Output Arguments

`out_obj`
: resulting destination object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

lkimport_raw can be used to read raw binary data and import it to a data segment. lkimport will either create an object with the new segment in it or append the new segment to the optionally supplied add_obj object.

Valid arch types are: KMACH_UNKNOWN, KMACH_LOCAL, KMACH_I80x86, KMACH_SPARC, KMACH_MIPSEL, KMACH_IBMRT, KMACH_RS6000, KMACH_POWERPC, KMACH_MC680x0, KMACH_M88x00,KMACH_MIPSEB, KMACH_PARISC, KMACH_CONVEX, KMACH_APOLLO, KMACH_ALPHA, KMACH_CYBER, KMACH_CRAY, KMACH_VAX, and KMACH_I860.

Valid data types are: KBIT, KBYTE, KUBYTE, KSHORT, KUSHORT, KINT, KUINT, KLONG, KULONG, KFLOAT, KDOUBLE, KCOMPLEX, and KDCOMPLEX

C.1.7. lkinsertseg() — *insert segment(s) from one object into another*

Synopsis

```
int
lkinsertseg(
```

```
    kobject source_object1,
    kobject source_object2,
    char   *segment_list,
    kobject destination_object)
```

Input Arguments

`source_object1`
object to get the segment to insert

`source_object2`
object to insert the segment into

`segment_list`
list of segment(s) to insert

Output Arguments

`destination_object`
object after the segment was inserted

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine will insert a segment(s) specified by segment_list from source_object1 into source_obj2 and return the resulting object in destination_object. Each segment's attributes and data will be copied into the destination_object. If segment_list is NULL it will copy all the segments in the object. The segment_list is a string that contains each segment name separated by space(s).

```
lkinsertseg(source_object1, source_object2,
            "segment1 segment2", destination_object);
```

If source_object2 is not NULL it will be copied using kdms_copy_object() into the destination_object then the segment(s) inserted. If source_object2 is NULL, the segment(s) will be inserted into the destination_object directly. This is the same as calling lkcpseg() as follows:

```
lkcpseg(source_object1, "segment1 segment2",
        destination_object);
```

This routine will actually call lkcpseg() to insert the segment source_obj2 to the destination_object.

C.1.8. lkrmseg() — *delete specified segments from an object creating new object*

Synopsis

```
int
lkrmseg(
    kobject source_object,
```

```
      char   *segment_list,
      kobject destination_object)
```

Input Arguments

`source_object`
: object to delete the segment from

`segment_list`
: name of segment to delete

Output Arguments

`destination_object`
: object without the segment(s)

Returns

TRUE (1) if one deleted, FALSE (0) if none deleted

Description

This routine will copy the source_object to the destination_object then delete the segment(s) specified by segment_list from the destination_object. Each segment's attributes and data will be delete. If segment_list is NULL it will delete all the segments in the object, but leave all object level attributes. The segment_list is a string that contains each segment name separated by space(s).

```
lkrmseg(source_object, "segment1 segment2",
        destination_object);
```

If the segment does not exist an error message is printed, but will not return until an attempt to delete all the segments has been done.

C.1.9. lksegcmp() — *compare two data segments*

Synopsis

```
int
lksegcmp(
   kobject obj1,
   char *name1,
   kobject obj2,
   char *name2,
   char *segment1,
   char *segment2,
   int cast,
   double tolerance,
   int run_silent,
   int print_summary,
   int compare_attributes,
```

```
int print_position,
int print_value1,
int print_type1,
int print_value2,
int print_type2,
kfile * summary_file,
kfile * difference_file)
```

Input Arguments

`obj1`
object containing the first segment for comparison.

`name1`
name of the first object to use in all summary reports.

`obj2`
object containing the second segment for comparison.

`name2`
name of the second object to use in all summary reports.

`segment1`
name of the segment from the first object to compare.

`segment2`
name of the segment from the second object to compare.

`cast`
TRUE if segments of differing data types should be casted to the higher data type before comparison. If this is FALSE and the data types differ, the segments are considered different.

`tolerance`
comparison tolerance to use when comparing each point.

`run_silent`
TRUE implies that nothing should be printed during the entire comparison to either of the summary or the difference files - only the return status of the program is needed as output.

`compare_attributes`
TRUE if the attributes for the given segment should be compared

`print_position`
TRUE if the position of differing points should be printed to the difference file.

`print_value1`
TRUE if the value from the first segment should be printed to the difference file.

`print_type1`
TRUE if the data type from the first segment should be printed to the difference file.

`print_value2`
TRUE if the value from the second segment should be printed to the difference file.

`print_type2`
TRUE if the data type from the second segment should be printed to the difference file.

Output Arguments

`summary_file`
An open kfile in which to print the summary information. Note that this summary will be printed after the specific differences are printed to the difference file. If NULL, then nothing will be printed.

`difference_file`
An open kfile in which to print the specific data differences. Note that this will be printed before the summaries are printed to the summary file. If NULL, then nothing will be printed.

Returns

TRUE (1) if the objects are tolerably identical, FALSE (0) otherwise

Description

This routine will compare the data contained in two data segments point by point, allowing for precise comparisons of data objects independent of file format or machine architecture.

The segments will first be compared to see if they are of the same size and dimensionality. If the data types are different, a comparison can still be done in the higher of the two data types if the cast argument is set.

A +/- comparison tolerance to use when comparing each data point may also be passed in. In cases where a complex comparison is performed, this tolerance will be applied to both the real and imaginary parts of the values.

A summary of the differences found between the segments can be printed to a given summary file. This file will contain the names of the two segments, and will specify if the segments are the same or different. If the sizes, dimensionality, or data types are different, then the differing attributes are printed. If the sizes and dimensionality match, then the number of differing points is printed. The segment level attributes may optionally be compared and printed as part of the summary.

Specific differences found between the segments can be printed to a difference file. This file will contain an entry for each differing point printed in the following format:

```
[POSITION]
< Input #1 Value (Input #1 Data Type)
> Input #2 Value (Input #2 Data Type)
```

Each component of this data point difference may optionally be left out by passing in the appropriate arguments.

Note that the specific differences are generated and printed to the difference file first. The summary is then generated and printed.

If no output is desired in either file, this routine can be told to run silent by setting the run_silent argument.

This routine will compare data "intelligently", never reading in more than a line at a time from both data objects. This allows very large data objects to be compared on machines with limited virtual memory.

Restrictions

Data types can not be printed to the difference file without their corresponding values.

C.2. Utility Functions in the ksegops Library

C.2.1. lksegcmp_attributes() — *compare segment and/or global attributes of two objects*

Synopsis

```
int lksegcmp_attributes(
     kobject obj1,
     kobject obj2,
     char   *segment,
     kfile  *outfile,
     int     run_silent)
```

Input Arguments

`obj1`
: object containing first segment with attributes to compare.

`obj2`
: object containing second segment with attributes to compare.

`segment`
: segment name to compare the attributes for. If NULL, then the object level segments between the two objects are compared.

`run_silent`
: TRUE if no output should be generated to the outfile.

Output Arguments

`outfile`
: An open kfile in which to print the attribute differences.

Returns

TRUE (1) if the attributes all match, FALSE (0) otherwise

Description

This routine compares attributes between two data objects. If a segment is specified, then the attributes for that segment is compared across the two data objects. The size, data type, dimensionality, and index order attributes are ignored. If no segment is specified, then the object-level attributes are compared.

The attribute differences are printed to a specified output file, unless the run_silent argument is set, in which case nothing is printed.

Restrictions

Only defined attributes are compared. User created attributes are not considered in the comparison.

Table of Contents