suplib

Edition 1.5.1, for version 1.5.1
6 January 2014

Diab Jerius

Table of Contents

1 Copying 1

2 Introduction 3

3 Ranges of numbers 5

3.1 rangef_parse 5
3.2 range_perror 7
3.3 rangef_del 9
3.4 rangef_in 9
3.5 rangef_new 10
3.6 rangel_parse 11
3.7 rangel_count 13
3.8 rangel_del 13
3.9 rangel_dump 14
3.10 rangel_new 15
3.11 rangel_next 15
3.12 rangel_minmax 16
3.13 rangel_rep_maxval 17
3.14 rangel_rep_minval 17
3.15 rangel_rep_val 18
3.16 rangel_reset 19

4 Strings and tokens 21

4.1 str_dup 21
4.2 str_join 21
4.3 str_prune 22
4.4 str_rep 22
4.5 str_tokcnt 23
4.6 str_tokenize 24
4.7 str_tokenize_free 25
4.8 str_tokq 26
4.9 str_tokq_restore 28
4.10 str_tokqcnt 28
4.11 str_tokbqenize 29
4.12 str_tokbqenize_free 31
4.13 str_tokbq 31
4.14 str_tokbq_init 33
4.15 str_tokbq_free 34
4.16 str_tokbq_restore 34
4.17 str_tokbqcnt 35
4.18 str_trim 36
4.19 str_trunc 37

4.20 str_dtokcnt ... 37
4.21 str_dtoksplit ... 38
4.22 str_varscan ... 39
4.23 str_interp ... 40
4.24 tokmatch ... 41
4.25 tokcmp ... 42
4.26 tokqsplit ... 43
4.27 toksplit ... 43
4.28 unescape ... 44
4.29 unquote ... 45

5 Parsing Keyword_value pairs ... 47
5.1 keyval_st ... 47
5.2 keyval_perror ... 50
5.3 keyval_cmp ... 51

6 Timing processes ... 53
6.1 clearTime ... 53
6.2 startTime ... 53
6.3 elapsedTime ... 54
6.4 currentTime ... 54
6.5 incTime ... 55
6.6 diffTime ... 56
6.7 stringTime ... 56

7 I/O handling ... 59
7.1 fget_rec_new ... 59
7.2 fget_rec_delete ... 59
7.3 fget_rec_read ... 60
7.4 fget_rec ... 61
7.5 fget_rec_append ... 62

8 File/Directory processing ... 65
8.1 base_name ... 65
8.2 searchpath ... 65

9 Routines helpful for Debugging ... 67
9.1 debug_init ... 67
9.2 dbflag ... 67
9.3 die ... 68
9.4 hexdump ... 69

10 Handling units ... 71
10.1 Setting up the structures ... 71
10.2 units_parse ... 72
10.3 units_cvt ... 73

11 List manipulation 75
11.1 bnd_bsearch 75
11.2 partition 76

12 1D and 2D Image manipulation 79
12.1 ave_dev_err 79
12.2 center_variter 80
12.3 weightpos 82
12.4 wtvar 83

13 Statistical Calculations 85
13.1 gsmirn 85
13.2 gsmirn2 86
13.3 stcalc 87
13.4 kolmogorov 89

1 Copying

Except where otherwise noted, the following copyright applies:

2 Introduction

`suplib` is a collection of various routines which are of general interest. It is grouped by topic into several sub-libraries, each with its own header file.

3 Ranges of numbers

Often one is faced with dealing with sets, or ranges of numbers. They can be a pain to parse, as well as to iterate over.

The `rangel` family of routines work on ranges of integers. Parsing of the ranges is done with `rangel_parse`. Iterating through them is done with `rangel_next` and `rangel_reset`. There are several routines for manipulating the minimum and maximum range values.

The `rangef` routines work on ranges of real numbers. Parsing of range specifications is done with `rangef_parse`. To determine if a real is within a range list, use `rangef_in`.

Errors are handled via `range_perror`.

3.1 rangef_parse

parse a list of floating point ranges

Synopsis

```
#include <suplib/range.h>

RangeErr rangef_parse(
  RangeFList **rfl,
  char *range_spec,
  RangeOpts opts,
  double minval,
  double maxval,
  long *where
);
```

Parameters

`RangeFList **rfl`
: returned list of ranges

`char *range_spec`
: string containing range to parse

`RangeOpts opts`
: range options

  Possible values for a `RangeOpts` are as follows: `Range_SORT`, `Range_MERGE`, `Range_INCOMPLETE`, `Range_UNSIGNED`

`double minval`
: replacement value for incomplete start

`double maxval`
: replacement value for incomplete end

`long *where`
: position in string where error occurred

Description

`rangef_parse` translate a list of ranges of the form `"r1,r2,r3,r4"` into start-end pairs. a range has the format:

```
[start]{ ':' | '(' | ')' | '~' }[end]
```

where start and end are optional limits on the range.

Returns

It returns a code indicating whether an error ocurred. (See `suplib/range.h` for the possible errors, and codes. It also returns the location of the error in the range specification via the passed `where` variable.

Possible values for a `RangeErr` are as follows:

`RangeErr_OK`
: no error

`RangeErr_NOMEM`
: out of memory

`RangeErr_INCOMPLETE`
: incomplete range

`RangeErr_ERANGE`
: number out of bounds

`RangeErr_ILLNUM`
: not a number

`RangeErr_NEGNUM`
: negative number

`RangeErr_OFLOWSTART`
: overflow of start value

`RangeErr_NONPOSCOUNT`
: non-positive count

`RangeErr_OFLOWEND`
: overflow of end value

`RangeErr_INTERNAL`
: internal error

`RangeErr_ORDER`
: start greater than end

`RangeErr_EMPTY`
: float range is empty set

`RangeErr_MAXERR`

The Range Operators Are As Follows

‘:’ range from start to end, inclusive of both.

‘(’ range from start to end, exclusive of end.

‘)’ range from start to end, exclusive of start.

‘~’ range from start to end, exclusive of both.

The handling of the ranges depends upon the flags set in the *opts* arguement. *opts* is set to the logical OR of zero or more of the following values

`Range_SORT`
: The resultant ranges will be sorted in ascending order by their lower bound

`Range_MERGE`
: Adjacent ranges will be merged, if possible. This implies `Range_SORT`.

`Range_INCOMPLETE`
: If set, incomplete ranges, i.e., those without an explicit start or end value, are accepted. In this case, implied starts are set to the value of the passed *minval* argument, while implied ends are set to the value of the *maxval* argument.

`rangef_parse` creates a `RangeFList` object and returns it via the *rfl* parameter.

Author

Diab Jerius

3.2 range_perror

output an error message associated with a `RangeERR`

Synopsis

```
#include <suplib/range.h>

void range_perror(
  FILE *fout,
  RangeErr error,
  const char *spec,
  long where
);
```

Parameters

`FILE *fout`
: where to print the error message

`RangeErr error`
: which error to print

  Possible values for a `RangeErr` are as follows:

  `RangeErr_OK`
  : no error

  `RangeErr_NOMEM`
  : out of memory

  `RangeErr_INCOMPLETE`
  : incomplete range

  `RangeErr_ERANGE`
  : number out of bounds

  `RangeErr_ILLNUM`
  : not a number

  `RangeErr_NEGNUM`
  : negative number

  `RangeErr_OFLOWSTART`
  : overflow of start value

  `RangeErr_NONPOSCOUNT`
  : non-positive count

  `RangeErr_OFLOWEND`
  : overflow of end value

  `RangeErr_INTERNAL`
  : internal error

  `RangeErr_ORDER`
  : start greater than end

  `RangeErr_EMPTY`
  : float range is empty set

  `RangeErr_MAXERR`

`const char *spec`
: the specification which caused the error

`long where`
: where in the spec the error ocurred

Description

Error messages are returned by the `range` routines via an integer code of type `RangeERR`. This routine acts much like the system `perror` routine. It takes as input the output stream, the error code, the range specification passed to the range parsing routines, and the *where* argument returned by those routines. It prints an error message to the passed output stream.

Author

Diab Jerius

3.3 rangef_del

destroy a `RangeFList` object.

Synopsis

```
#include <suplib/range.h>

RangeFList *rangef_del(RangeFList *rl);
```

Parameters

`RangeFList *rl`
: Not Documented.

Description

This routine frees all memory associated with a `RangeFList` object.

Returns

It returns `NULL`.

Author

Diab Jerius

3.4 rangef_in

determine if a number is within a list of ranges

Synopsis

```
#include <suplib/range.h>

int rangef_in(
  RangeFList *rl,
```

```
  double t
);
```

Parameters

`RangeFList *rl`
: Not Documented.

`double t`
: Not Documented.

Description

`rangef_in` determines if a passed number is encompassed by one of the ranges in a passed `RangeFList`.

Returns

It returns non-zero if the number is within the list of ranges, zero if not.

Author

Diab Jerius

3.5 rangef_new

create a new `RangeFList`

Synopsis

```
#include <suplib/range.h>

RangeFList *rangef_new(size_t nr);
```

Parameters

`size_t nr`
: Not Documented.

Description

`RangeFList` objects are used to contain a list of integral ranges. `rangel_new` creates such an object for a given number of ranges.

Returns

It returns a pointer to a `RangeFList` object upon success, `NULL` if it ran out of memory.

Author

Diab Jerius

3.6 rangel_parse

parse a list of integral ranges

Synopsis

```
#include <suplib/range.h>

RangeErr rangel_parse(
  RangeLList **rll,
  char *range_spec,
  RangeOpts opts,
  long minval,
  long maxval,
  long *where
);
```

Parameters

`RangeLList **rll`
: returned range list

`char *range_spec`
: string containing range to parse

`RangeOpts opts`
: range options

  Possible values for a `RangeOpts` are as follows: `Range_SORT`, `Range_MERGE`, `Range_INCOMPLETE`, `Range_UNSIGNED`

`long minval`
: minimum value for incomplete start

`long maxval`
: maximum value for incomplete end

`long *where`
: position in string where error occurred

Description

`rangel_parse` translates a list of ranges of the form

```
r1[{,| }r2[{,| }r3]]...
```

into start-end pairs. The handling of the ranges depends upon the flags set in the *opts* arguement. *opts* is set to the logical OR of zero or more of the following values

`Range_SORT`
: The resultant ranges will be sorted in ascending order by their lower bound

`Range_MERGE`
: Adjacent ranges will be merged, if possible. This implies `Range_SORT`.

`Range_INCOMPLETE`
: If set, incomplete ranges, i.e., those without an explicit start or end value, are accepted. In this case, implied starts are set to the value of the passed *minval* argument, while implied ends are set to the value of the *maxval* argument.

`Range_UNSIGNED`
: Only unsigned integers are allowed in the ranges.

Returns

It returns a code indicating whether an error ocurred. See `suplib/range.h` for the possible errors and codes. It also returns the location of the error in the range specification via the passed `where` variable.

Possible values for a `RangeErr` are as follows:

`RangeErr_OK`
: no error

`RangeErr_NOMEM`
: out of memory

`RangeErr_INCOMPLETE`
: incomplete range

`RangeErr_ERANGE`
: number out of bounds

`RangeErr_ILLNUM`
: not a number

`RangeErr_NEGNUM`
: negative number

`RangeErr_OFLOWSTART`
: overflow of start value

`RangeErr_NONPOSCOUNT`
: non-positive count

`RangeErr_OFLOWEND`
: overflow of end value

`RangeErr_INTERNAL`
: internal error

`RangeErr_ORDER`
: start greater than end

`RangeErr_EMPTY`
: float range is empty set

`RangeErr_MAXERR`

A Range Has The Format

```
[start]{':'}[end]
[start]{'/'}count
```

where *start* and *end* are optional limits on the range, and *count* specifies the number of elements in a range. In the first form, if *start* or *end* is missing, *minval* and *maxval* are used appropriately (if the incomplete range flag is set). In the second form, if *start* is missing, the range starts at 1 if no previous range was specified, or directly follows the end of the previous range.

`rangel_parse` creates a `RangeLList` object and returns it via the *rll* parameter.

Author

Diab Jerius

3.7 rangel_count

determine the number of elements in a range list

Synopsis

```
#include <suplib/range.h>

unsigned long rangel_count(RangeLList *rl);
```

Parameters

`RangeLList *rl`
: the range list to count

Description

`rangel_count` counts the number of elements in each range of a range list, returning the grand total.

Returns

It returns the number of elements in the range list.

Author

Diab Jerius

3.8 rangel_del

destroy a `RangeLList` object.

Synopsis

```
#include <suplib/range.h>

RangeLList *rangel_del(RangeLList *rl);
```

Parameters

`RangeLList *rl`
: the `RangeLList` object to delete

Description

This routine frees all memory associated with a `RangeLList` object.

Returns

It returns `NULL`.

Author

Diab Jerius

3.9 rangel_dump

pretty print a `RangeLList`

Synopsis

```
#include <suplib/range.h>

void rangel_dump(
  FILE *fout,
  RangeLList *rl
);
```

Parameters

`FILE *fout`
: where to print

`RangeLList *rl`
: the list to print

Description

`rangel_dump` prints out a simple dump of a `RangeLList` to the passed output stream.

Author

Diab Jerius

3.10 rangel_new

create a new `RangeLList`

Synopsis

```
#include <suplib/range.h>

RangeLList *rangel_new(size_t nr);
```

Parameters

`size_t nr` the number of ranges to allocate space for

Description

`RangeLList` objects are used to contain a list of integral ranges. `rangel_new` creates such an object for a given number of ranges.

Returns

It returns a pointer to a `RangeLList` object upon success, `NULL` if it ran out of memory.

Author

Diab Jerius

3.11 rangel_next

get the next element in the range list

Synopsis

```
#include <suplib/range.h>

int rangel_next(
  RangeLList *rl,
  long *elem
);
```

Parameters

`RangeLList *rl`
: the range to iterate over

`long *elem`
: the returned element

Description

`rangel_next` serves as an iterator over the passed list of ranges, returning the next element in the list. To reset the iterator, Section 3.16 [rangel_reset], page 19. The element is returned via the *elem* argument.

Returns

It returns '`1`' if there is an element left, '`0`' if it has reached the end of the range list.

Author

Diab Jerius

3.12 rangel_minmax

get the minimum and maximum values in a range list

Synopsis

```
#include <suplib/range.h>

int rangel_minmax(
  RangeLList *rl,
  long *min,
  long *max
);
```

Parameters

`RangeLList *rl`
: the range in question

`long *min`
: the minimum range value

`long *max`
: the maximum range value

Description

`rangel_minmax` determines the minimum and maximum values in a range list, storing them in the passed `min` and `max` arguments.

Returns

It returns `1` upon success, `0` if the range was empty.

Author

Diab Jerius

3.13 rangel_rep_maxval

replace matching range maximum values

Synopsis

```
#include <suplib/range.h>

void rangel_rep_maxval(
  RangeLList *rl,
  long maxval,
  long repval
);
```

Parameters

`RangeLList *rl`
: the list to scan

`long maxval`
: the maximum value to match

`long repval`
: the replacement maximum value

Description

`rangel_rep_maxval` replaces the maximum value for those ranges whose maximum values match the passed one with another value.

Author

Diab Jerius

3.14 rangel_rep_minval

replace matching range minimum values

Synopsis

```
#include <suplib/range.h>

void rangel_rep_minval(
  RangeLList *rl,
  long minval,
  long repval
);
```

Parameters

`RangeLList *rl`
: the list to scan

`long minval`
: the minimum value to match

`long repval`
: the replacement minimum value

Description

`rangel_rep_minval` replaces the minimum value for those ranges whose minimum values match the passed one with another value.

Author

Diab Jerius

3.15 rangel_rep_val

replace matching range minimum and maximum values

Synopsis

```
#include <suplib/range.h>

void rangel_rep_val(
  RangeLList *rl,
  long minval,
  long maxval,
  long minrepval,
  long maxrepval
);
```

Parameters

`RangeLList *rl`
: the list to edit

`long minval`
: the minimum value to match

`long maxval`
: the maximum value to match

`long minrepval`
: the replacement minimum value

`long maxrepval`
: the replacement maximum value

Description

`rangel_rep_maxval` replaces the maximum value for those ranges whose maximum values match the passed one with another value. It also replaces the minimum value of the ranges which match the passed minimum value with another value.

Author

Diab Jerius

3.16 rangel_reset

reset the `RangeLList` internal state

Synopsis

```
#include <suplib/range.h>

void rangel_reset(RangeLList *rl);
```

Parameters

`RangeLList *rl`
: the range list to reset

Description

The internal state of the passed `RangeLList` object is set so that the next call to Section 3.11 [rangel_next], page 15 will return the start of the first range.

Author

Diab Jerius

4 Strings and tokens

4.1 str_dup

duplicate a string

Synopsis

```
#include <suplib/str.h>

char *str_dup(const char *string);
```

Parameters

`const char *string`
: the string to duplicate

Description

`str_dup` makes a duplicate copy of a string, allocating space and copying the original into the new space. If the passed pointer is `NULL`, it returns `NULL`.

Returns

Upon success, it returns a pointer to the duplicate string. The user is responsible for freeing this memory. Upon failure (i.e., out of memory) it returns `NULL`

Author

Diab Jerius

4.2 str_join

concatenate strings with delimiters

Synopsis

```
#include <suplib/str.h>

char *str_join(
  const char *delim,
  size_t n,
  ...
);
```

Parameters

`const char *delim`
: the inter-string delimiter, may be `NULL`

`size_t n`
: the number of strings

`...`
: the strings

Description

`str_join` concatenates a list of strings, separated by the given delimiter. If any of the passed pointers is `NULL`, it is ignored.

Returns

It returns a pointer to a newly allocated buffer containing the the concatenated string, `NULL` if it couldn't allocate the requisite memory. The caller is responsible for freeing the buffer.

4.3 str_prune

remove leading and trailing white space from a string

Synopsis

```
#include <suplib/str.h>

char *str_prune(char *str);
```

Parameters

`char *str`
: the string to modify

Description

`str_prune` removes leading and trailing white space from a string. It truncates the string at the first trailing white space, then moves the string to the left, starting at the first non-white space character. The input string is modified.

Returns

It returns the pointer which it was passed.

Author

Diab Jerius

4.4 str_rep

replaces all occurances of a character in a string with another

Synopsis

```
#include <suplib/str.h>

char *str_rep(
  char *str,
  int c_old,
  int c_new
);
```

Parameters

`char *str` the string to scan

`int c_old` the character to be replaced

`int c_new` the character to be inserted

Description

`str_rep` scans a string, replacing every occurance of a the specified character with another.

Returns

It returns the pointer to the string which was passed.

Author

Diab Jerius

4.5 str_tokcnt

count the number of tokens in a string.

Synopsis

```
#include <suplib/str.h>

size_t str_tokcnt(
  const char *string,
  const char *delim
);
```

Parameters

`const char *string`
: string to parse

`const char *delim`
: the delimiters to use

Description

`str_tokcnt` determines the number of tokens which would be returned in the course of calling the `strtok` function through completion. It does not change the passed string.

Returns

it returns the number of tokens in the string.

Author

Diab Jerius

4.6 str_tokenize

tokenize a string at specified delimiters

Synopsis

```
#include <suplib/str.h>

int str_tokenize(
  char *string,
  const char *delim,
  char ***string_argv,
  int skip,
  int restore
);
```

Parameters

`char *string`
: string to parse

`const char *delim`
: the delimiters to use

`char ***string_argv`
: to store the parsed tokens

`int skip`
: true if consecutive delimiters are treated as a single delimiter

`int restore`
: true if `str_tokq` should restore the delimiter character from the previous invocation

Description

This routine tokenizes the string using `str_tokq`. The invoking routine calls `str_tokenize` once to parse all the tokens for a given delimiter. The tokens are stored in *string_argv*.

if the `restore` argument is false, `str_tokenize` will allocate memory for each of the tokens; otherwise it will return pointers to the locations in the original string.

The routine `str_tokenize_free` is provided to free the memory allocated by `str_tokenize`

Returns

The number of tokens. The third argument, `string_argv`, contains the parsed tokens.

Errors

On error, `str_tokenize` returns `0` and sets `errno` accordingly. The following errors are recognized:

`EINVAL` There were unbalanced quotes or the last character in the string was an escape character (i.e., no character to escape).

`ENOMEM` A memory allocation failed

Author

Dan Nguyen

Diab Jerius

4.7 str_tokenize_free

free memory allocated by `str_tokenize`

Synopsis

```
#include <suplib/str.h>

void str_tokenize_free(
  char **string_argv,
  int restore
);
```

Parameters

`char **string_argv`
: storage for the parsed tokens

`int restore`
: same as passed to str_tokenize

Description

This routine frees the memory allocated by the `str_tokenize`. The user must call `str_tokenize_free` with the same value for restore as was used when `str_tokenize` was invoked.

Returns

void

Author

Dan Nguyen

Diab Jerius

4.8 str_tokq

split a string at specified delimiters

Synopsis

```
#include <suplib/str.h>

char *str_tokq(
  char *string,
  const char *delim,
  int skip,
  int restore,
  char **ptr,
  char *dchar
);
```

Parameters

`char *string`
: the string to parse

`const char *delim`
: the set of characters that delimit tokens

`int skip`
: true if consecutive delimiters are treated as a single delimiter

`int restore`
: true if `str_tokq` should restore the delimiter character from the previous invocation

`char **ptr`
: state information needed by `str_tokq`

`char *dchar`
: state information needed by `str_tokq`

Description

This routine works like `strtok`, except that it understands escaped characters (with '\' as the escape character) and quoted strings. It also can be told *not* to skip over multiple consecutive delimiter characters (e.g., if the delimiter is ',', the string `",,a"` will return three tokens, the first two empty strings). Additionally `str_tokq` can be instructed to restore the parsed string to its initial state (normally it sticks '`\0`' where the delimiter characters are).

The invoking routine calls `str_tokq` multiple times, once per token. The first call should be made with `string` set to the address of the string to parse. On subsequent calls it should be set to `NULL`. The delimiters may be changed on subsequent calls.

Each invocation of `str_tokq` returns a pointer to the next token. If no more tokens are available, it returns `NULL`. If skip mode is on (the parameter `skip` is non-zero), an empty string is considered to have no tokens. If skip mode is off, an empty string is considered to have a single token.

If the parameter `restore` is non-zero, `str_tokq` will restore the delimiter character which was replaced by a '`\0`' on the *previous* call. Thus, to fully restore the string to its original state, `str_tokq` *must* be invoked repeatedly until it returns a `NULL`.

Escaped characters are recognized anywhere in a string. Quotes need not be the first character in a token (e.g., '`foo" "bar`') will be treated as one token, and can be either single (both forward and backwards) or double. To embed quotes in like-quoted strings, escape them (e.g., '`foo"\""bar`').

Unlike `strtok`, `str_tokq` is set up to allow concurrent use on multiple strings. To accomplish this, all internal state information is kept in two user supplied locations. The first (parameter `ptr`), is used by `strtok` to keep track of where it is in the string. The second, `dchar`, contains the delimiter character that ended the token. Please note that the invoking function must pass the *addresses* of these locations to `str_tokq`. Here's some sample code:

```
char *ptr, dchar, *tok;
tok = str_tokq(string, ",", 0, 1, &ptr, &dchar);
while (tok)
{
... process tokens ...
tok = str_tokq(NULL, ",", 0, 1, &ptr, &dchar);
}
```

Errors

On error, `str_tokq` returns `NULL` and sets `errno` accordingly. The following errors are recognized:

`EINVAL` There were unbalanced quotes or the last character in the string was an escape character (i.e., no character to escape).

4.9 str_tokq_restore

restore string passed to str_tokq

Synopsis

```
#include <suplib/str.h>

char *str_tokq_restore(
  char *ptr,
  char dchar
);
```

Parameters

`char *ptr` state information returned by `str_tokq`

`char dchar`
state information returned by `str_tokq`

Description

`str_tokq` changes the scanned string as it processes it. If the `restore` flag passed to `str_tokq` is set then the string will be restored if the `str_tokq` is called to completion. However, sometimes it's necessary to interrupt the parsing, and repeatedly calling the `str_tokq` to restore the string is ridiculous.

This function restores the string.

Returns

It returns a pointer to the next character that `str_tokq` would have returned, except in the case where the end of string was reached, in which case it will return a pointer to the end of string character. This allows one to pass the pointer to another independent invocation of `str_tokq`, if, for example, continued parsing is to be done in another context.

Author

Diab Jerius

4.10 str_tokqcnt

count the number of tokens in a string

Synopsis

```
#include <suplib/str.h>

int str_tokqcnt(
  char *string,
```

```
  const char *delim,
  int skip
);
```

Parameters

`char *string`
: the string to parse

`const char *delim`
: the set of characters that delimit tokens

`int skip`
: true if consecutive delimiters are treated as a single delimiter

Description

This routine counts the number of tokens in a string as would be returned by `str_tokq`. It *does* change the input string, but restores it to its initial state upon completion. A constant string (e.g., a statically declared string) should not be passed.

Returns

Upon success it returns the number of tokens in the string. On error it returns '`-1`'.

Author

Diab Jerius

4.11 str_tokbqenize

tokenize a string at specified delimiters

Synopsis

```
#include <suplib/str.h>

int str_tokbqenize(
  char *string,
  char ***string_argv,
  const char *delim,
  const char *open_quote,
  const char *close_quote,
  char escape_char,
  int actions,
  struct str_tokbqdata *tbqd,
  int *error
);
```

Parameters

`char *string`
: the string to parse

`char ***string_argv`
: to store the parsed tokens

`const char *delim`
: the set of characters that delimit tokens

`const char *open_quote`
: the set of characters that delimit opening quotes. e.g., '`"'{[`'

`const char *close_quote`
: the set of matching characters that delimit closing quotes. These must be in the same order as the opening characters in `open_quote`, e.g., '`"'}]`'

`char escape_char`
: the escape character

`int actions`
: skip, restore, and escape modes

`struct str_tokbqdata *tbqd`
: state information

`int *error`
: returned error code

Description

This routine tokenizes the string using `str_tokbq`. The invoking routine calls `str_tokbqenize` once to parse all the tokens for a given delimiter. The tokens are stored in *string_argv*.

if the `restore` argument is false, `str_tokbqenize` will allocate memory for each of the tokens; otherwise it will return pointers to the locations in the original string.

The routine `str_tokbqenize_free` is provided to free the memory allocated by `str_tokbqenize`

Returns

The number of tokens. The third argument, `string_argv`, contains the parsed tokens.

Errors

On error, `str_tokbqenize` returns `0` and sets `errno` accordingly. The following errors are recognized:

`EINVAL` There were unbalanced quotes or the last character in the string was an escape character (i.e., no character to escape).

`ENOMEM` A memory allocation failed

Author

Diab Jerius Dan Nguyen

4.12 str_tokbqenize_free

free memory allocated by `str_tokbqenize`

Synopsis

```
#include <suplib/str.h>

void str_tokbqenize_free(
  char **string_argv,
  int action
);
```

Parameters

`char **string_argv`
: storage for the parsed tokens

`int action`
: same as passed to str_tokbqenize

Description

This routine frees the memory allocated by the `str_tokbqenize`. The user must call `str_tokbqenize_free` with the same value for restore as was used when `str_tokbqenize` was invoked.

Returns

void

Author

Diab Jerius Dan Nguyen

4.13 str_tokbq

split a quoted string at specified delimiters

Synopsis

```
#include <suplib/str.h>

char *str_tokbq(
```

```
  char *string,
  const char *delim,
  const char *open_quote,
  const char *close_quote,
  char escape_char,
  int actions,
  struct str_tokbqdata *tbqd,
  int *error
);
```

Parameters

`char *string`
: the string to parse

`const char *delim`
: the set of characters that delimit tokens

`const char *open_quote`
: the set of characters that delimit opening quotes. e.g., '`"'{[`'

`const char *close_quote`
: the set of matching characters that delimit closing quotes. These must be in the same order as the opening characters in `open_quote`, e.g., '`"')}]`'

`char escape_char`
: the escape character

`int actions`
: skip, restore, and escape modes

`struct str_tokbqdata *tbqd`
: state information

`int *error`
: returned error code

Description

This routine works like `strtok`, except that it understands escaped characters and quoted strings. It handles balanced quote characters (i.e. '`[`' and '`]`'). It does not handle nested quotes.

It can collapse consecutive delimiter characters, or generate empty tokens. For example, if the delimiter is '`,`', parsing of the string `",,a"` would return three tokens, the first two empty strings. Additionally `str_tokbq` can be instructed to restore the parsed string to its initial state (normally it sticks '`\0`' where the delimiter characters are).

`str_tokbq` is invoked multiple times, once per token. Unlike `str_tok`, `str_tokbq` stores all of its state information externally, in a caller provided structure. This must be initialized

with `str_tokbq_init` before the first call to `str_tokbq`. The contents are unique to each parsed string.

Delimiters, quote characters, and the escape character may be changed between calls.

Each invocation of `str_tokbq` returns a pointer to the next token. If no more tokens are available, it returns `NULL`. If skip mode is on an empty string is considered to have no tokens. If skip mode is off, an empty string is considered to have a single token.

The `actions` flag is used to specify how to handle consecutive delimiters, string restoration, and whether to recognize escape sequence. The following flags (which may be combined with `|` are available:

`STRTOK_ESCAPE`
: Recognize escape character sequences before quotes and delimiters. They are ignored elsewhere. The escape character is given by the `escape_char` parameter.

`STRTOK_RESTORE str_tokbq replaces delimiter characters`
: in the passed string with '`\0`' as it parses it. This flag directs `str_tokbq` to restore the character from the previous call. To fully restore the string to its original state, the caller must either repeatedly invoke `str_tokbq` until it returns a `NULL` or invoke the `str_tokbq_restore` function.

`STRTOK_SKIP`
: If set, this indicates that consecutive delimiters are collapsed ("skipped").

Quotes need not be the first character in a token (e.g., '`foo" "bar`') will be treated as one token, and can be either single (both forward and backwards) or double. To embed quotes in like-quoted strings, escape them (e.g., '`foo"\""bar`').

Errors

On error, `str_tokbq` returns `NULL` and stores an error code in `*error`. The following errors are recognized:

`EINVAL` There were unbalanced quotes.

4.14 str_tokbq_init

initialize str_tokbq state information

Synopsis

```
#include <suplib/str.h>

struct str_tokbqdata *str_tokbq_init(struct str_tokbqdata *tbqd);
```

Parameters

`struct str_tokbqdata *tbqd`
Not Documented.

Description

`str_tokbq_init` is called before the first call to `str_tokbq` to initialize its state information.

Returns

It returns the pased structure pointer

Author

Diab Jerius

4.15 str_tokbq_free

free str_tokbq state information

Synopsis

```
#include <suplib/str.h>

void str_tokbq_free(struct str_tokbqdata *tbqd);
```

Parameters

`struct str_tokbqdata *tbqd`
Not Documented.

Description

`str_tokbq_free` should be called if the full set of calls to `str_tokbq` (e.g. until it returns `NULL` is interrupted and will not be completed.

Author

Diab Jerius

4.16 str_tokbq_restore

restore string passed to str_tokbq

Synopsis

```
#include <suplib/str.h>

char *str_tokbq_restore(struct str_tokbqdata *tqbd);
```

Parameters

`struct str_tokbqdata *tqbd`
: Not Documented.

Description

`str_tokbq` changes the scanned string as it processes it. If the `restore` flag passed to `str_tokbq` is set then the string will be restored if the `str_tokbq` is called to completion. However, sometimes it's necessary to interrupt the parsing, and repeatedly calling the `str_tokbq` to restore the string is ridiculous.

This function restores the string.

Returns

It returns a pointer to the next character that `str_tokbq` would have returned, except in the case where the end of string was reached, in which case it will return a pointer to the end of string character. This allows one to pass the pointer to another independent invocation of `str_tokbq`, if, for example, continued parsing is to be done in another context.

Author

Diab Jerius

4.17 str_tokbqcnt

count the number of tokens in a string

Synopsis

```
#include <suplib/str.h>

int str_tokbqcnt(
  char *string,
  const char *delim,
  const char *open_quote,
  const char *close_quote,
  char escape_char,
  int actions,
  struct str_tokbqdata *tbqd,
  int *error
);
```

Parameters

`char *string`
: the string to parse

`const char *delim`
: the set of characters that delimit tokens

`const char *open_quote`
: the set of characters that delimit opening quotes. e.g., '`"'{[`'

`const char *close_quote`
: the set of matching characters that delimit closing quotes. These must be in the same order as the opening characters in `open_quote`, e.g., '`"')}]`'

`char escape_char`
: the escape character

`int actions`
: skip, restore, and escape modes

`struct str_tokbqdata *tbqd`
: state information

`int *error`
: returned error code

Description

This routine counts the number of tokens in a string as would be returned by `str_tokbq`. It *does* change the input string, but restores it to its initial state upon completion. A constant string (e.g., a statically declared string) should not be passed.

Returns

Upon success it returns the number of tokens in the string. On error it returns '`-1`'.

Author

Diab Jerius

4.18 str_trim

removes white space from front of a string

Synopsis

```
#include <suplib/str.h>

char *str_trim(char *str);
```

Parameters

`char *str` the string to trim

Description

`str_strim` removes white space from the front of a string by shifting the string to the left, starting at the first non-white space character. It works in place, altering the passed string.

Returns

It returns the pointer which it was passed

Author

Diab Jerius

4.19 str_trunc

truncate white space at the end of a string

Synopsis

```
#include <suplib/str.h>

char *str_trunc(char *string);
```

Parameters

`char *string`
: the string to truncate

Description

`str_trunc` truncates white space at the end of a string by writing a '`\0`' at the appropriate position in the string. It alters the passed string.

Returns

It returns the pointer which it was passed.

Author

Diab Jerius

4.20 str_dtokcnt

count the number of tokens in a string.

Synopsis

```
#include <suplib/str.h>

size_t str_dtokcnt(
```

```
  const char *string,
  char *delim
);
```

Parameters

`const char *string`
: string to parse

`char *delim`
: the delimiters to use

Description

`str_dtokcnt` parses a string in the same manner as `str_dtoksplit` and returns the number of tokens. It does not alter the string.

Returns

It returns the number of tokens in the string.

Author

Diab Jerius

4.21 str_dtoksplit

split a string into tokens.

Synopsis

```
#include <suplib/str.h>

size_t str_dtoksplit(
  char *str,
  char *tok[],
  const char *delim,
  int ntok
);
```

Parameters

`char *str` the string to split up

`char *tok[]`
: the array to stick

`const char *delim`
: the delimiters to split on

`int ntok` the maximum number of tokens to read

Description

This routine splits a string into a series of tokens. Unlike the system library function `strtok`, each instance of a delimiter implies a token. In `strtok`, sequential delimiters are collapsed into one. This function allows one to have empty tokens. It fills a caller provided array with pointers to the tokens. The caller may specify a maximum number of tokens to read. Note that an empty string corresponds to a single, empty, token.

The passed string is modified (end of string characters are inserted where necessary).

Returns

It returns the number of tokens read.

Author

Diab Jerius

4.22 str_varscan

scan a string for $VAR or ${VAR}

Synopsis

```
#include <suplib/str.h>

int str_varscan(
  const char *string,
  void (*callback)(const char *,const char *,const char *,const char *,void *),
  void *udata,
  const char **errptr
);
```

Parameters

`const char *string`
: the string to process

`void (*callback)(const char *,const char *,const char *,const char *,void *)`
: callback

`void *udata`
: extra data to pass to the callback routine

`const char **errptr`
: place to store a pointer to a bad spot in the string

Description

`str_varscan` scans a strings for substrings of the form `${VAR}` or `$VAR`. `VAR`'s first character must be in the set `[a-zA-Z_]`, with the remaining characters being in the set `[a-zA-Z0-9_]`.

When such a string is found, the passed callback function is called with the addresses of the first and last characters of the entire substring, as well as the first and last characters in `VAR`.

Returns

Upon success, it returns `0`; Upon error, *errptr is set to point the the character that evoked the error, and one of the following is returned:

`EDOM` There was an unbalanced '`{`'

Author

Diab Jerius

4.23 str_interp

interpolates environmental variables into a string

Synopsis

```
#include <suplib/str.h>

char *str_interp(
  const char *text,
  int keep_undef,
  const char **ustart,
  const char **uend,
  int *error
);
```

Parameters

`const char *text`
: the text to be interpolated into

`int keep_undef`
: if true, undefined variables are left in the text. if false, they are removed

`const char **ustart`
: if not NULL, set to the start of the undefined variable name

`const char **uend`
: if not NULL, set to the end of the undefined variable name

`int *error`
: error code

Description

This routine scans a string for embedded environmental variables and returns a pointer to a copy of the string with the interpolated values. The user is responsible for freeing the new string.

The input string is scanned left to right for substrings of the form `${VAR}` or `$VAR`. `VAR` must begin with an alphabet character or the '`_`' character. The remaining characters must be alphabetic, numeric, or the '`_`' character.

Returns

Upon success a pointer to the interpolated string is returned and `*error` is set to `0`. The user is responsible for freeing the memory pointed to by the returned string. If `keep_undef` was set and there was an undefined variable, then `*errp` is set to `EFAULT` and `*ustart` and `*uend` point at the first and last characters in the variable name. If multiple variables are undefined, they will point to the first undefined variable name. Upon error, `NULL` is returned and `*errp` is set to one of the following

`EDOM` `There was an unbalanced '{'. *ustart is set`
: to point the the character that evoked the error. `*uend` is undefined.

`ENOMEM`
: A memory allocation failed.

Author

Diab Jerius

4.24 tokmatch

match a string to a list of tokens

Synopsis

```
#include <suplib/str.h>

int tokmatch(
  const char *token,
  const TokList *toklist
);
```

Parameters

`const char *token`
: Not Documented.

`const TokList *toklist`
: Not Documented.

Description

`tokmatch` compares a string to a list of tokens in a user provided `TokList`. The `TokList` should be sorted alphabetically by name.

Returns

It returns the `id` of the matched token, '`-1`' otherwise. Note that this restricts the values of `id`.

Author

Diab Jerius

4.25 tokcmp

compare two tokens by name

Synopsis

```
#include <suplib/str.h>

int tokcmp(
  const void *tok1,
  const void *tok2
);
```

Parameters

`const void *tok1`
: Not Documented.

`const void *tok2`
: Not Documented.

Description

`tokcmp` compares two `TokListToken`'s by name. It is suitable for use by `bsearch` or `qsort`.

Returns

It returns '`-1`' if the first token's name is alphabetically first, '`0`' if they have the same name, and '`1`' if the first token's name follows the second, alphabetically.

Author

Diab Jerius

4.26 tokqsplit

split a string into tokens

Synopsis

```
#include <suplib/str.h>

int tokqsplit(
  char *str,
  char *tok[],
  const char *delim,
  int ntok,
  int split
);
```

Parameters

`char *str` the string to split up

`char *tok[]`
: the array that will receive the tokens

`const char *delim`
: the delimiters to split on

`int ntok` the maximum number of tokens to read

`int split` the `strtokq` `split` argument

Description

`tokqsplit` splits a string into a series of tokens using `str_tokq`. It fills a caller provided array with pointers to the tokens. The caller should specify the maximum number of tokens to read.

Returns

It returns the actual number of tokens in the string, which may differ (either greater and lesser) than the number requested. Upon error, it returns '`-1`'.

Author

Diab Jerius

4.27 toksplit

split a string into tokens

Synopsis

```
#include <suplib/str.h>

int toksplit(
  char *str,
  char *tok[],
  const char *delim,
  int ntok
);
```

Parameters

`char *str` the string to split up

`char *tok[]`
: the array to stick

`const char *delim`
: the delimiters to split on

`int ntok` the maximum number of tokens to read

Description

`toksplit` splits a string into a series of tokens using `strtok`. It fills a caller provided array with pointers to the tokens. The caller should specify the maximum number of tokens to read.

Returns

It returns the actual number of tokens in the string, which may differ (either greater and lesser) than the number requested.

Author

Diab Jerius

4.28 unescape

replace escaped characters with their true values

Synopsis

```
#include <suplib/str.h>

char *unescape(char *string);
```

Parameters

`char *string`
: the string to unescape

Description

`unescape` replaces escaped characters with their true values. The escape prefix is the character '\'. It recognizes the following special characters: '`\t`', '`\n`'. All other escaped characters are replaced by the character (i.e., '`\g`' is turned into '`g`'). It makes the changes in-place.

Returns

It returns a pointer to the original string upon success, `NULL` if the escape prefix occurred without a character to escape.

Author

Diab Jerius

4.29 unquote

remove quotes from a string

Synopsis

```
#include <suplib/str.h>

char *unquote(char *string);
```

Parameters

`char *string`
: the string to unquote

Description

`unquote` removes pairs of quotes from a string. The first quote need not be at the beginning of the string. It makes the changes in-place. For example, '`foo" "bar`' is turned into '`foo bar`', just like in a UNIX shell. It can deal with either single (forward or backward) or double quotes, and understands escaped characters (with '\' as the escape prefix).

Returns

It returns a pointer to the original string upon success, `NULL` if the quotes were unbalanced or an escape prefix occurred without a character to escape.

Author

Diab Jerius

5 Parsing Keyword_value pairs

5.1 keyval_st

parse a string with keyword-value pairs

Synopsis

```
#include <suplib/keyval.h>

KeyValErr keyval_st(
  const char *kv_spec,
  KeyVal *map,
  size_t nkey,
  void *data,
  long *where
);
```

Parameters

`const char *kv_spec`
: keyword - value specification string to parse

`KeyVal *map`
: keyword value type and offset map

`size_t nkey`
: number of keywords in map

`void *data`
: where the data are to be stored

`long *where`
: position in string where error occurred

Description

`keyval_st` parses a string which contains multiple keyword-value pairs, or boolean switches. Pairs or switches should be separated by semi-colons, commas, spaces, or tabs. Keywords and values are separated by an equals sign. Values which are strings with embedded spaces should be surrounded by qutoes. To embed a quote, escape it with '\'. Boolean values may be specified either as keyword-value pairs, or just as the keyword (indicating *true*) with an optional '`!`' (indicating *false*). In the former case, the value may be any of '`yes`', '`no`', '`0`', '`1`', '`on`', '`off`' (case is not significant).

The resultant values are stored in a user supplied structure. Offsets into the structure are encoded in an array of `KeyVal` structures, which my be constructed by the caller. The structures have the following definition:

```
typedef struct KeyVal
{
```

```
char *key;        keyword name
KeyType type;    keyword type
size_t offset;    offset of value
int set;        set to true if keyword read
int (*xfrm)( char *in, void *out );
} KeyVal;
```

The *type* field is one of `Key_String`, `Key_Integer`, `Key_Float`, `Key_Double`, or `Key_Boolean`. The *offset* is the byte offset into the user supplied structure where the value is to be stored. The *set* field is set by `keyval_st` if that keyword was present in the passed specification string.

The *xfrm* field points to an optional function which takes the value string (*in*) and converts it to the correct type. It should store the converted value in the address pointed to by the *out* argument. For string types, it should store a freshly allocated string. It should return zero upon success, non-zero upon failure. *xfrm* should be set to `NULL` if not used. It is not used for `Key_Boolean` types.

Returns

It returns a code indicating whether an error ocurred. (See `suplib/keyval.h` for the possible errors, and codes. It also returns the location of the error in the passed specification via the passed `where` variable. The `keyval_perror` routine will print a human understandable form of a `KeyValErr`.

Possible values for a `KeyValErr` are as follows:

`KeyValErr_OK`
: no error

`KeyValErr_NOMEM`
: out of memory

`KeyValErr_NOKEY`
: no such keyword

`KeyValErr_UNBAL`
: unbalanced quote or escape character

`KeyValErr_INVAL`
: invalid value specification

`KeyValErr_RANGE`
: the value is out of range

`KeyValErr_MAXERR`

The Keyword Types And Their Storage Requirements Are As Follows

`Key_String`
: `keyval` will allocate space for a string; in this case, the field in the structure must be a `char *` so as to accept a pointer to the string. If the contents of that pointer are not `NULL`, `keyval` will free that memory first.

`Key_Integer`
: the field must be an `int`;

`Key_Float`
: the field must be a `float`;

`Key_Double`
: the field must be a `double`;

`Key_Boolean`
: the field must be an `int`;

Construction of this table is eased by use of the `KeyValStEntry` preprocessor macro:

```
where to stick the values
typedef struct Data
{
int    n_flies;
double fly_lifetime;
char *food;
} Data;

KeyVal vals[] =
{
KeyValStEntry( fly_lifetime, Key_Double, Data ),
KeyValStEntry( food, Key_String, Data ),
KeyValStEntry( n_flies, Key_Integer, Data ),
};
#define NFIELDS ( sizeof(vals) / sizeof(KeyVal ) )
```

The `KeyValStEntry` macro assumes that the name of the keyword is the same as the name of the field in the structure which is to receive the data. You can get around this restriction by specifying the values to the `KeyVal` structure directly.

The entries in the `KeyVal` array *must* be in alphabetical order by keyword name. (See `keyval_cmp`.)

Author

Diab Jerius

5.2 keyval_perror

output an error message associated with a `KeyValErr`

Synopsis

```
#include <suplib/keyval.h>

void keyval_perror(
  FILE *fout,
  KeyValErr error,
  const char *spec,
  long where
);
```

Parameters

`FILE *fout`
: where to print the error message

`KeyValErr error`
: which error to print

  Possible values for a `KeyValErr` are as follows:

  `KeyValErr_OK`
  : no error

  `KeyValErr_NOMEM`
  : out of memory

  `KeyValErr_NOKEY`
  : no such keyword

  `KeyValErr_UNBAL`
  : unbalanced quote or escape character

  `KeyValErr_INVAL`
  : invalid value specification

  `KeyValErr_RANGE`
  : the value is out of range

  `KeyValErr_MAXERR`

`const char *spec`
: the specification which caused the error

`long where`
: where in the spec the error ocurred

Description

Error messages are returned by the `keyval` routines via an integer code of type `KeyValErr`. This routine acts much like the system `perror` routine. It takes as input the output stream, the error code, the keyval specification passed to the keyval parsing routines, and the *where* argument returned by those routines. It prints an error message to the passed output stream.

Author

Diab Jerius

5.3 keyval_cmp

compare `KeyVal` structures, alphabetically

Synopsis

```
#include <suplib/keyval.h>

int keyval_cmp(
  const void *v1,
  const void *v2
);
```

Parameters

`const void *v1`
: pointer to first `KeyVal` structure

`const void *v2`
: pointer to second `KeyVal` structure

Description

`keyval_cmp` is an alphabetical comparison routine for `KeyVal` structures intended for use by `bsearch` or `qsort`.

Returns

It returns '`-1`' if the first structure is lexically earlier than the second, '`0`' if they are the same, and '`1`' if the first is lexically after the second.

Author

Diab Jerius

6 Timing processes

6.1 clearTime

initialize a `Time` structure to zero.

Synopsis

```
#include <suplib/times.h>

Time *clearTime(Time *t);
```

Parameters

`Time *t` the Time to clear

Description

`clearTime` clears the passed `Time` structure.

Returns

It returns a pointer to the passed `Time` structure.

Author

Diab Jerius

6.2 startTime

initialize a `Time` structure to the current time.

Synopsis

```
#include <suplib/times.h>

Time *startTime(Time *start);
```

Parameters

`Time *start`
the structure to fill. may be `NULL`

Description

`startTime` initializes an internal `Time` structure with the current user, system, and clock times for the process. If the `start` parameter is not `NULL`, the same data are copied to the user specified `Time` structure. The internal `Time` structure will be overwritten by each call to `startTime`.

Returns

It returns a pointer to the internal `Time` structure.

Author

Diab Jerius

6.3 elapsedTime

return the time elapsed

Synopsis

```
#include <suplib/times.h>

Time *elapsedTime(Time *start);
```

Parameters

`Time *start`
: the start time from which to determine the elapsed time

Description

`elapsedTime` determines the difference in time between the current user, system and clock times and either a passed `Time` structure, or the internal `Time` structure initialized by `startTime` (if the passed pointer `start` is `NULL`).

Returns

It returns a pointer to an internal `Time` structure containing the difference in times. This structure is private to `elapsedTime` and will be overwritten by subsequent calls to it. It returns NULL upon error.

Author

Diab Jerius

6.4 currentTime

return the current process times

Synopsis

```
#include <suplib/times.h>

Time *currentTime(Time *time);
```

Parameters

`Time *time`
: the destination `Time` structure

Description

`currentTime` fills a specified `Time` structure with the current user, system and clock times for the process.

Returns

It returns the passed pointer.

Author

Diab Jerius

6.5 incTime

add some Time to some other Time

Synopsis

```
#include <suplib/times.h>

Time *incTime(
  Time *dest,
  Time *inc
);
```

Parameters

`Time *dest`
: The Time which will be incremented

`Time *inc`
: the Time increment

Description

`incTime` increments a `Time` structure by the amount in a another `Time` structure

Returns

It returns a pointer to the destination `Time` structure, `NULL` if either the destination or the increment was a `NULL` pointer.

Author

Diab Jerius

6.6 diffTime

return the difference in times between two `Time` structures.

Synopsis

```
#include <suplib/times.h>

Time *diffTime(
  Time *diff,
  Time *end,
  Time *start
);
```

Parameters

`Time *diff`
: where to copy the time differences

`Time *end`
: the ending time

`Time *start`
: the starting time

Description

`diffTime` fills an internal `Time` structure with the difference in times between two passed `Time` structures (`end` - `start`). The internal structure will be overwritten by the next call to `diffTime`. If the `diff` argument is not `NULL`, the difference data are copied there as well.

Returns

It returns a pointer to the internal `Time` structure upon success. If either of `start` or `end` are `NULL`, it returns `NULL`.

Author

Diab Jerius

6.7 stringTime

return a string representation the passed `Time` structure.

Synopsis

```
#include <suplib/times.h>

char *stringTime(Time *time);
```

Parameters

`Time *time`
: Not Documented.

Description

`stringTime` generates a printable string representing the passed `Time` structure. The string is rendered into internal storage which will be overwritten by then next call to `stringTime`.

Returns

It returns a pointer to a printable string.

Author

Diab Jerius

7 I/O handling

7.1 fget_rec_new

Create a `fget_rec` buffer structure.

Synopsis

```
#include <suplib/io.h>

void *fget_rec_new(
  size_t buf_size,
  size_t extend
);
```

Parameters

`size_t buf_size`
: the initial size of the buffer

`size_t extend`
: the amount to extend the buffer each time it is too small

Description

This routine creates a buffer structure that can be used by `fget_rec`. The buffer structure should be destroyed with `fget_rec_delete`.

The calling procedure must provide the initial size of the buffer (including the trailing '\0' which seals off the string), as well as the amount by which the buffer should be extended if the buffer is shorter than the input record. `fget_rec` repeatedly extends the buffer until a record fits, so this value needn't be that large. Note that the `buf_size` argument should be greater than '`1`'.

Returns

It returns a pointer to a buffer structure (*not* the actual buffer) or `NULL` if it was unable to allocate it.

Author

Diab Jerius

7.2 fget_rec_delete

Delete a `fget_rec` buffer structure.

Synopsis

```
#include <suplib/io.h>

void fget_rec_delete(void *rec);
```

Parameters

`void *rec` an `fget_rec` buffer structure to delete

Description

This routine deletes a buffer structure previously created by `fget_rec_new`. It also deletes the accompanying buffer.

Author

Diab Jerius

7.3 fget_rec_read

Read a record of arbitrary length.

Synopsis

```
#include <suplib/io.h>

char *fget_rec_read(
  FILE *fin,
  void *recbuf,
  size_t start
);
```

Parameters

`FILE *fin` the input stream from which to read the data

`void *recbuf`
: an `fget_rec` buffer structure, created by `fget_rec_new`

`size_t start`
: the zero-based index into the buffer where the incoming record should be stored

Description

`fget_rec_read` reads data from the passed stream until either an end of line character or the end of file is reached. It removes any end of line character from the string. It writes the data into a user provided buffer, and extends the buffer as necessary to hold a complete

input record. The end of the buffer is terminated with a '\0'. The buffer structure should created by `fget_rec_new` and deleted with `fget_rec_delete`. The data is written into the buffer at the position specified by the `start` parameter. If the starting position is beyond the end of the buffer the buffer is extended and the string currently in the buffer is extended with spaces to have length `start`. This happens even if there is no new data available in the input stream.

Returns

It normally returns a pointer to the data that has been read. Upon end of file, it returns `NULL`. If it cannot extend the record, it returns `NULL` and sets `errno` to `ENOMEM`. If there was a read error on the stream, it returns `NULL` and the stream's error flag is set (use `ferror` to check it).

Author

Diab Jerius

7.4 fget_rec

Read a record of arbitrary length.

Synopsis

```
#include <suplib/io.h>

char *fget_rec(
  FILE *fin,
  void *recbuf
);
```

Parameters

`FILE *fin` the input stream from which to read the data

`void *recbuf`
: an `fget_rec` buffer structure, created by `fget_rec_new`

Description

`fget_rec` reads data from the passed stream until either an end of line character or the end of file is reached. It removes any end of line character from the string. It writes the data into a user provided buffer, and extends the buffer as necessary to hold a complete input record. The end of the buffer is terminated with a '\0'. The buffer structure should created by `fget_rec_new` and deleted with `fget_rec_delete`.

This is essentially a wrapper around `fget_rec_read`, with `start = 0`.

Returns

It normally returns a pointer to the data that has been read. Upon end of file, it returns `NULL`. If it cannot extend the record, it returns `NULL` and sets `errno` to `ENOMEM`. If there was a read error on the stream, it returns `NULL` and the stream's error flag is set (use `ferror` to check it).

Author

Diab Jerius

7.5 fget_rec_append

Append a record of arbitrary length.

Synopsis

```
#include <suplib/io.h>

char *fget_rec_append(
  FILE *fin,
  void *recbuf
);
```

Parameters

`FILE *fin` the input stream from which to read the data

`void *recbuf`
: an `fget_rec` buffer structure, created by `fget_rec_new`

Description

`fget_rec_append` reads data from the passed stream until either an end of line character or the end of file is reached. It removes any end of line character from the string. It appends the data to the end of the data in the user provided buffer, and extends the buffer as necessary to hold a complete input record. The end of the buffer is terminated with a '\0'. The buffer structure should created by `fget_rec_new` and deleted with `fget_rec_delete`. Note that it appends the new data to the end of the

This is essentially a wrapper around `fget_rec_read`, with `start = strlen(buf)`.

Returns

It normally returns a pointer to the data that has been read. Upon end of file, it returns `NULL`. If it cannot extend the record, it returns `NULL` and sets `errno` to `ENOMEM`. If there was a read error on the stream, it returns `NULL` and the stream's error flag is set (use `ferror` to check it).

Author

Diab Jerius

8 File/Directory processing

8.1 base_name

Remove the prefix and optionally a suffix of a string.

Synopsis

```
#include <suplib/file.h>

char *base_name(
  char *string,
  char *suffix
);
```

Parameters

`char *string`
: the string to process

`char *suffix`
: an optional suffix to remove. Set it to `NULL` to do nothing

Description

`base_name` removes all but the last level in a path (the file name) and, optionally, a suffix. The former is accomplished by returning a pointer to the first character in the filename. The latter is accomplished by writing an end of string character into the string.

Returns

It returns a pointer to the beginning of the filename.

Author

Diab Jerius

8.2 searchpath

Search a set of paths for a file.

Synopsis

```
#include <suplib/file.h>

char *searchpath(
  char *file,
  char *default_path,
```

```
  char *path_spec
);
```

Parameters

`char *file`
: the file to look for

`char *default_path`
: the default path

`char *path_spec`
: the list of paths

Description

`searchpath` scans a list of paths for a file. The paths are specified in a string, separated by the `PATHSEP` character (':' for UNIX). The paths are searched in the order that they occur in the string. If an empty path is found in the specification, a user specified default path (or the current directory, should this not be specified) will be searched at that point. If the filename begins with `DIRSEP` ('/' for UNIX), it is used directly.

Returns

It returns a pointer to a dynamically allocated string holding the complete path, or `NULL` if it couldn't find the file or it ran out of memory. The calling procedure must free this string.

Author

Diab Jerius

9 Routines helpful for Debugging

9.1 debug_init

parse debug flags and initialize internals

Synopsis

```
#include <suplib/debug.h>

int debug_init(const char *debug_flags);
```

Parameters

`const char *debug_flags`
: a comma delimited list of debug flags

Description

`debug_init` and `dbflag` comprise a simple system for allowing users to specify debug flags to a program in a human understandable fashion.

`debug_init` parses a string for debug flags. The flags are comma separated tokens of any characters (preferrably alphanumeric). It builds an internal list of the specified flags, which can be tested with the `dbflag` function.

There is but a single list kept per process, and subsequent calls to `debug_init` will overwrite the list.

To free the memory associated with the internal list, `debug_init` should be called with *debug_flags* set to `NULL`.

Returns

It returns zero upon success, nonzero if it ran out of memory

Author

Diab Jerius

9.2 dbflag

test for a debug flag

Synopsis

```
#include <suplib/debug.h>

int dbflag(const char *dbf);
```

Parameters

`const char *dbf`
: the debug flag to test

Description

`dbflag` tests if the passed string was in the list of debug flags previously parsed by `debug_init`.

Returns

It returns zero if the flag was not specified, non-zero if it was.

Author

Diab Jerius

9.3 die

print a formatted message and exit with error

Synopsis

```
#include <suplib/debug.h>

void die(
  char *format,
  ...
);
```

Parameters

`char *format`
: the `printf` style format string for the message

`...`
: the arguments that make up the message

Description

`die` is a quick and dirty method of printing an error message and then exiting a program. It's just a combination of `fprintf` and `exit`, sending the output to `stderr` and exiting with an exit value of '`1`'

Author

Diab Jerius

9.4 hexdump

print out memory as hexidecimals

Synopsis

```
#include <suplib/debug.h>

void hexdump(
  FILE *stream,
  void *buf,
  size_t length
);
```

Parameters

`FILE *stream`
: where to print

`void *buf`
: the start of memory to dump

`size_t length`
: the number of bytes to dump

Description

`hexdump` will pretty print out a section of memory in base 16.

Author

Diab Jerius

10 Handling units

It's often handy for users to input numbers with unit specifications, so that they can work in units appropriate to them, rather than to the program. This subpackage helps parse such specifications, as well as convert between units.

Each type of unit is assigned an id, which is a non-negative integer. The id serves as an index into an array of `UnitConvert` structures, which contain multiplicative conversion factors from the given unit to a fiducial one. For example, length may be specified in kilometers, meters, or millimeters. If millimeters is the fiducial unit, than the conversion factors would be 1000, 1, .001, respectively. `UnitConvert` structures are bundled into a `UnitConvertList` for ease of access.

Names are mapped to units via a `TokList` structure, with the units' ids specified in the `TokListToken` id field. There may be more than one name for a given unit (e.g. '`kilometer`', '`km`').

These two data structures are bundled together via the `UnitsList` structure.

This subpackage provides a set of linear, angular, temporal and energy units via `UnitsLinear_def`, `UnitsAngular_def`, `UnitsTime_def`, and `UnitsEnergy_def`. Their definitions are available in `suplib/units_linear.h`, `suplib/units_angular.h`, `suplib/units_time.h`, and `suplib/units_energy.h`, respectively

10.1 Setting up the structures

The first thing to do is to determine the set of units, and to declare an `enum` to hold the id's. Note that since the id's are indices into the conversion list, they should be zero-based:

```
enum
{
  UNIT_parsec, /* 'parsec' */
  UNIT_meter, /* 'm', 'meter' */
  UNIT_decimeter, /* 'decimeter', 'dm' */
  UNIT_centimeter, /* 'centimeter', 'cm' */
  UNIT_millimeter, /* 'millimeter', 'mm' */
  UNIT_micrometer, /* 'micrometer', 'micron', 'um' */
  UNIT_nanometer, /* 'nanometer', 'nm' */
  UNIT_Angstrom /* 'A', 'Angstrom', 'angstrom' */
};
```

Next, create a `TokList` structure which holds the names by which users can refer to the units. Note that this must be in alphabetical order, and there may be more than one name for a unit.

```
static TokListToken
map[] =
{
  { "A", UNIT_Angstrom },
  { "Angstrom", UNIT_Angstrom },
  { "angstrom", UNIT_Angstrom },
```

```
  { "centimeter", UNIT_centimeter },
  { "cm", UNIT_centimeter },
  { "decimeter", UNIT_decimeter },
  { "dm", UNIT_decimeter },
  { "m", UNIT_meter },
  { "meter", UNIT_meter },
  { "micrometer", UNIT_micrometer },
  { "micron", UNIT_micrometer },
  { "millimeter", UNIT_millimeter },
  { "mm", UNIT_millimeter },
  { "nanometer", UNIT_nanometer },
  { "nm", UNIT_nanometer },
  { "parsec", UNIT_parsec },
  { "um", UNIT_micrometer }
};
TokList map_list = GenTokList( map );
```

Then, construct a table of conversion factors. These *must* be in the same order as your `enum` table for the indices to work correctly.

```
static UnitConvert
convert[] =
{
  { UNIT_parsec, 3.086e19 },
  { UNIT_meter, 1000 },
  { UNIT_decimeter, 100 },
  { UNIT_centimeter, 10 },
  { UNIT_millimeter, 1 },
  { UNIT_micrometer, 1e-3 },
  { UNIT_nanometer, 1e-6 },
  { UNIT_Angstrom, 1e-7 },
};
UnitConvertList convert_list = genUnitConvertList( convert );
```

Finally, construct a `UnitList` structure:

```
UnitList MyUnits = { &convert_list, &map_list };
```

If you'd like to automate this, see `mk_units` and `default.units` in the `suplib/units` source directory. `mk_units` is a script which reads unit names and conversions from `default.units` and generates the header and `C` code for the structures.

10.2 units_parse

parse a string composed of a floating point number and a units specification.

Synopsis

```
#include <suplib/units.h>
```

```
int units_parse(
  const char *spec,
  UnitVal *uv,
  const UnitsList *list
);
```

Parameters

`const char *spec`
: the specification to parse

`UnitVal *uv`
: the resultant number and unit id

`const UnitsList *list`
: the list of units

Description

This routine parses a string composed of a floating point number and a units specification (e.g. '`1mm`', '`23 arcsec`'). White space between the number and the unit is ignored. The caller must ensure that there is no white space following the unit specification.

Returns

The parsed number and the unit id are written into the passed `UnitVal` structure. The `unit` element is only changed if a valid unit was found. `units_parse` returns a `UNITS_ParseErr` consistent with the results of the parse:

`UNITS_OK` no errors were encountered

`UNITS_BADNUM`
: there was an error while parsing the number

`UNITS_NOSPEC`
: no units specification was present

`UNITS_BADSPEC`
: the units specification wasn't recognized

Author

Diab Jerius

10.3 units_cvt

get the conversion factor from one unit to another

Synopsis

```
#include <suplib/units.h>
```

```
double units_cvt(
  const UnitsList *list,
  int from,
  int to
);
```

Parameters

`const UnitsList *list`
: the Units list

`int from`
: the unit id of the source unit

`int to`
: the unit id of the destination unit

Description

`units_cvt` returns a conversion factor from one unit to another. Both units must be in the passed `UnitConvertList`.

Returns

Upon success (i.e., the units are in the `UnitConvertList`), it returns the conversion factor. Upon error, it returns '`0.0`'.

Author

Diab Jerius

11 List manipulation

11.1 bnd_bsearch

binary search with bounding element return

Synopsis

```
#include <suplib/lists.h>

const void *bnd_bsearch(
  const void *key,
  const void *base,
  size_t n,
  size_t size,
  const void **lo_bnd,
  const void **hi_bnd,
  int (*cmp)(const void *keyval,const void *datum)
);
```

Parameters

`const void *key`
: a pointer to the comparison key

`const void *base`
: the list of objects

`size_t n`
: the number of objects in the list

`size_t size`
: the size of an object in bytes

`const void **lo_bnd`
: the address of a pointer which will be set to the address of the largest object lower than the key

`const void **hi_bnd`
: the address of a pointer which will be set to the address of the smallest object higher than the key

`int (*cmp)(const void *keyval,const void *datum)`
: a routine which compares the user supplied key to an object, returning <1, 0, or >1 if the key is respectively less than, equal to, or greater than the object

Description

`bnd_bsearch` performs a binary search upon a list of objects, returning the matching object or the bounding objects if the key is not found.

- If an object matches *key*, its address is returned and *lo_bnd* and *hi_bnd* are set to the address.
- If an object is not found, `NULL` is returned, and *hi_bnd* and *lo_bnd* are set as follows:
  - If *key* falls between two objects, *lo_bnd* and *hi_bnd* are set to the addresses of the bracketing objects.
  - If *key* is less than any object in the list, *lo_bnd* is set to `NULL` and *hi_bnd* is set to *base*.
  - If *key* is larger than any object in the list, *lo_bnd* is set to *base* + (*n*-1) and *hi_bnd* is set to `NULL`.

Warning

Note that while *hi_bnd* and *lo_bnd* are declared as `void *`, the calling routine *must* pass a pointer to a pointer!

Author

Diab Jerius

11.2 partition

partition a list about an object

Synopsis

```
#include <suplib/lists.h>

void *partition(
  void *obj,
  void *p_obj,
  void *work,
  size_t s_obj,
  unsigned long n_obj,
  int (*obj_comp)(const void *obj1,const void *obj2)
);
```

Parameters

`void *obj` list of objects to partition

`void *p_obj`
: object about which to partition list. need not be in the list

`void *work`
: work area with size 2 * `s_obj`

`size_t s_obj`
: size of an object in bytes

`unsigned long n_obj`
: number of objects in list

`int (*obj_comp)(const void *obj1,const void *obj2)`
: function which compares two objects and returns < 1, 0, > 1, depending if the first object is less than, equal to, or greater than the second

Description

partitions the list `obj[0 ... n]` into two sub-lists `obj[0 ... q]` and `obj[q+1 ... n]`, such that `obj[i] (0 <= i <= q) <= pobj <= obj[j] (q < j <= n)` where `pobj` is a supplied "pivot" point. returns the top object in the first sub-list, i.e. `obj[q]`. `pobj` need not be in the list. in this case, if no object is less than `pobj`, `NULL` is returned.

References

See "Introduction to Algorithms", T.H.Kormen, C.E.Leiserson, and R.L. Rivest sec. 8.1, p.154.

Author

Diab Jerius

12 1D and 2D Image manipulation

12.1 ave_dev_err

determine statistics for a list of objects.

Synopsis

```
#include <suplib/imagefcts.h>

void ave_dev_err(
  void *objs,
  size_t n_obj,
  size_t s_obj,
  double *x_ave,
  double *x_ave_err,
  double *x_dev,
  double *x_dev_err,
  double *tot_wt,
  void (*get_stuff)(const void *obj,double *x,double *x_err,double *w,double *w_err)
);
```

Parameters

`void *objs`
: pointer to list of objects to process

`size_t n_obj`
: number of objects in list

`size_t s_obj`
: size of an object, in bytes

`double *x_ave`
: weighted average of objects

`double *x_ave_err`
: uncertainty in weighted average

`double *x_dev`
: weighted deviation of objects

`double *x_dev_err`
: uncertainty in weighted deviation

`double *tot_wt`
: total weight of objects

`void (*get_stuff)(const void *obj,double *x,double *x_err,double *w,double *w_err)`
: pointer to function which returns the weight, position, and uncertainties in weight and position of an object

Description

`ave_dev_err` determines the total weight, weighted average, uncertainty in weighted average, weighted deviation and uncertainty in weighted deviation for a list of objects.

Author

Diab Jerius

12.2 center_variter

iteratively determine the center of a distribution

Synopsis

```
#include <suplib/imagefcts.h>

int center_variter(
  void *objs,
  void *wwork,
  unsigned long n_objs,
  double tot_wt,
  size_t s_obj,
  double dtol,
  double fvar,
  unsigned long max_iter,
  unsigned long max_clip,
  double *center,
  double *dev_used,
  size_t *n_used,
  double *wt_used,
  void **objs_used,
  double (*get_x)(const void *obj,double *x),
  void (*put_x)(void *obj,double x),
  int (*comp)(const void *obj1,const void *obj2)
);
```

Parameters

`void *objs`
: the list of objects to process

`void *wwork`
: a work space of size 3 * s_obj

`unsigned long n_objs`
: total number of objects to process

`double tot_wt`
: the total (summed) weight of all of the objects.

`size_t s_obj`
: the size of an object in bytes

`double dtol`
: smallest absolute (not percentage) difference between two distances so as to consider them distinct.

`double fvar`
: fraction of determined variance in distance above which to ignore objects in center determination.

`unsigned long max_iter`
: maximum number of iterations to perform.

`unsigned long max_clip`
: maximum number of clips per iteration to perform.

`double *center`
: the final determined center

`double *dev_used`
: the standard deviation of the objects remaining after the last round of clips

`size_t *n_used`
: the number of objects remaining after the last round of clips

`double *wt_used`
: the summed weights of the objects remaining after the last round of clips

`void **objs_used`
: a pointer to the objects used

`double (*get_x)(const void *obj,double *x)`
: routine which retrieves position of an object. returns weight of object

`void (*put_x)(void *obj,double x)`
: routine which stuffs a position into an object.

`int (*comp)(const void *obj1,const void *obj2)`
: routine which compares two objects based upon their squared distances (same setup as routines for *<qsort>*)

Description

`center_variter` determines the center of a distribution by iteratively rejecting objects whose deviation in distance from a determined center is greater than a given number of sample deviations from the center. After each iteration, changes in the determined center are measured; if the changes are less than a specified threshold, the algorithm is deemed to

have converged. Additionally, there is a limit on the number of iterations performed. The input list of objects is reordered.

Returns

It returns the following information about the set of objects which survived the clipping spree:

`center` the final determined center

`dev_used` the mean deviation from the center

`n_used` the number of objects

`wt_used` the weight of the objects

`objs_used`
: a pointer into the (reordered) input list where the final list of objects starts

The actual value returned by `center_variter` will be one of:

`CVR_OK` everything went well

`CVR_LESSTHANTWO`
: less than two objects were left after clipping. `fvar` was probably too low

`CVR_MAXITER`
: the iteration limit was reached.

A More Detailed Description Of The Algorithm

1. a mean center is determined and used as the initial guess
2. distances to the determined center are calculated for all objects, and the variance (with respect to zero distance) is calculated.
3. objects with variances greater than the specified limit are rejected and the group variance recalculated after each rejection. this is repeated until either no more objects are rejected, the object count drops below a limit, or an iteration limit is reached.
4. a new mean center is calculated using the objects surviving the culling, and compared to the previous determination. if the distance between the two centers is less than the specified tolerance, the procedure ends. if the iteration limit has not been reached, the process continues with step 2.

Author

Diab Jerius

12.3 weightpos

determine the mean weighted position of a group of objects

Synopsis

```
#include <suplib/imagefcts.h>

double weightpos(
  void *objs,
  unsigned long n_tot,
  double tot_wt,
  size_t s_obj,
  double (*get_x)(const void *obj,double *x)
);
```

Parameters

`void *objs`
: list of objects whose mean position is to be determined

`unsigned long n_tot`
: total number of objects to process

`double tot_wt`
: total weight of objects. if zero, each object is assumed to have a weight of 1.

`size_t s_obj`
: size of an object

`double (*get_x)(const void *obj,double *x)`
: function which returns the position and weight of an object

Description

determine the mean weighted position of a group of objects. requires a user supplied function which extracts an object's position and weight. designed to be run over sub-groups of the list (for low memory situations). optimized for both weighted and unweighted (weight = 1) data.

Returns

It returns the mean weighted position of the group.

Author

Diab Jerius

12.4 wtvar

determine the unnormalized variance of objects' distance from a given point

Synopsis

```
#include <suplib/imagefcts.h>

double wtvar(
  double x,
  void *objs,
  unsigned long n_obj,
  double tot_wt,
  size_t s_obj,
  double (*get_x)(const void *obj,double *x)
);
```

Parameters

`double x` x coordinate of point from which to determine distance

`void *objs`
: pointer to list of objects to process

`unsigned long n_obj`
: number of objects in list

`double tot_wt`
: if non-zero, indicates that objects are weighted

`size_t s_obj`
: size of an object, in bytes

`double (*get_x)(const void *obj,double *x)`
: pointer to function which returns the coordinates of an object as well as its weight

Description

`wtvar` determines the unnormalized variance (from zero) of objects' distance from a given point. It requires a user supplied function which extracts an object's position.

Returns

It returns the variance.

13 Statistical Calculations

13.1 gsmirn

exact Smirnov Two-Sample tests for arbitrary distributions.

Synopsis

```
#include <suplib/stats.h>

int gsmirn(
  int nx,
  int ny,
  int kind,
  int *m,
  double dstat,
  double *q
);
```

Parameters

`int nx` The number of observations in the first sample

`int ny` The number of observations in the second sample

`int kind` The hypothesis tested

`int *m` The number of observations falling into each of K categories (with ascending order of category values)

`double dstat`
The statistic

`double *q` output: p-value

Description

`gsmirn` generates the P-value for the generalized two-sample Smirnov tests. It calculates the probability of the null hypothesis (that the two samples are the same) based upon one of three statistics

1. sup | X - Y |
2. sup (X - Y)
3. sup (Y - X)

The input parameter `kind` indicates which of these should be calculated. The input parameters `m` and `dstat` are calculated by the `stcalc` subroutine.

See Applied Statistics, Vol. 43, No. 1 (1994), 265-270.

Returns

0	no error
1	the nx < 1 or ny < 1
2	kind != 1, 2 or 3
3	q is not positive
4	m is inconsistent with nx and ny or has non-positive elements
5	allocation of workspace failed

Warning

This code returns incorrect values for large nx and ny for nx != ny for various ratios. See gsmirn2 for a slower version which does not suffer from these problems.

Author

Original by Andrei M. Nikiforov C transcription by Diab Jerius

13.2 gsmirn2

exact Smirnov Two-Sample tests for arbitrary distributions.

Synopsis

```
#include <suplib/stats.h>

int gsmirn2(
  int nx,
  int ny,
  int kind,
  int *m,
  double dstat,
  double *q
);
```

Parameters

int nx	The number of observations in the first sample
int ny	The number of observations in the second sample
int kind	The hypothesis tested
int *m	The number of observations falling into each of K categories (with ascending order of category values)
double dstat	The statistic

`double *q` output: p-value

Description

`gsmirn2` generates the P-value for the generalized two-sample Smirnov tests. It calculates the probability of the null hypothesis (that the two samples are the same) based upon one of three statistics

1. sup | X - Y |
2. sup (X - Y)
3. sup (Y - X)

The input parameter `kind` indicates which of these should be calculated. The input parameters `m` and `dstat` are calculated by the `stcalc` subroutine.

This version uses the method of Timonin and Chernomordik (Theor. Prob.Appl.,1985) to avoid large numbers for P instead of direct scaling adopted by published algorithm AS 288.

One-sided tests are coded separately to use the fast calculation scheme

See also `gsmirn`.

Returns

`0`	no error
`1`	the `nx` < 1 or `ny` < 1
`2`	`kind` != 1, 2 or 3
`3`	`q` is not positive
`4`	`m` is inconsistent with `nx` and `ny` or has non-positive elements
`5`	allocation of workspace failed

Author

Original by Andrei M. Nikiforov C transcription by Diab Jerius

Copyright

13.3 stcalc

calculate classical or weighted Smirnov statistic

Synopsis

```
#include <suplib/stats.h>
```

```
int stcalc(
  int icw,
  int nx,
  int ny,
  double *x,
  double *y,
  int *k,
  int **m,
  double *dstats
);
```

Parameters

`int icw` classical (1) or weighted (2) Smirnov statistic

`int nx` The number of observations in the first sample

`int ny` The number of observations in the second sample

`double *x` input: The first sample

`double *y` input: The second sample

`int *k` output: The number of categories in the pooled sample

`int **m` output: The number of observations in each category.

`double *dstats`
output: the calculated statistics

Description

`stcalc` calculates statistics, number of categories (i.e. unique values in the sample) in the pooled sample and numbers of observations falling into each category for the two-sample Smirnov tests for arbitrary distributions.

The following statistics are calculated.

1. sup | X - Y |
2. sup (X - Y)
3. sup (Y - X)

Returns

The number of categories is returned via the `k` argument. The address of an integer array with `*k` elements holding the number of observations per category is returned via `m`. `stcalc` allocates the array; the calling routine must free it. The statistics are written to the `dstats` parameter, which is a caller-allocated array of minimum length 3. The return value of the subroutine may have the following possible values.

`0` no error

`1` the `nx` < 1 or `ny` < 1

`2` allocation of workspace failed

Author

Original by Andrei M. Nikiforov C transcription by Diab Jerius

Copyright

13.4 kolmogorov

calculate the probability distribution of Kolmogorov's goodness-of-fit measure.

Synopsis

```
#include <suplib/stats.h>

int kolmogorov(
  int n,
  double d,
  double *p
);
```

Parameters

`int n`	The number of random variates
`double d`	The Kolmogorov statistic, D_n
`double *p`	Output: probability(D_n < d)

Description

`kolmogorov` calculates the probability distribution of Kolmogorov's goodness-of-fit measure, D_n, providing P(D_n < d).

This code is taken from G. Marsaglia, Wai Wan Tsang and Jingbo Wong, J.Stat.Software (http://www.jstatsoft.org/v08/i18/). The only changes were to support error returns.

Returns

`0`	no error
`ENOMEM`	a memory allocation failed.

Author

Original by G. Marsaglia, Wai Wan Tsang and Jingbo Wong.

Copyright