```
typedef struct {
    int nbucket;                        ///< number of buckets
    psHashBucket **buckets;             ///< the buckets themselves
} psHash;
```

where `nbucket` is the number of buckets defined for the hash functions, and `buckets` is an array of pointers to the individual buckets, each of which is defined by:

```
typedef struct psHashBucket {
    char *key;                          ///< key for this item of data
    void *data;                         ///< the data itself
    struct psHashBucket *next;          ///< list of other possible keys
} psHashBucket;
```

where each bucket contains the value of the `key`, a pointer to the `data`, and a pointer to the `next` list entry in the bucket (in the event that two or more keys have the same hash value).

A hash table is created with the following function:

```
psHash *psHashAlloc(void);
```

which allocates the space for the hash table and initializes all of the buckets.

The destructor for `psHash` must free all data associated with a complete hash table.

A data item may be added to the hash table with the function:

```
bool psHashAdd(psHash *table, const char *key, void *data);
```

In this function, the value of the string `key` is used to construct the hash value, find the appropriate bucket set, and add the new element `data` to the list. An existing element with the same value of `key` is destroyed using its registered destructor (`psMemBlock`). The return value of the function is a boolean defining the success or failure of the operation.

The data associated with a given key may be found with the function:

```
void *psHashLookup(psHash *table, const char *key);
```

which returns the data value pointed to by the element associated with `key`, or the value `NULL` if no match is found. Similarly, a specific key may be removed (deleted) with the function:

```
bool psHashRemove(psHash *table, const char *key);
```

The function returns a value of `true` if the operation was successfull, and `false` otherwise.

The function

```
psList *psHashKeyList(psHash *table);
```

returns the complete list of defined keys associated with the `psHash` table as a linked list.

```
psArray *psHashToArray(psHash *hash);
```

This function places the data in a `psHash` into a `psArray` container. This function does not free the elements or destroy the input collection. Rather, it increments the reference counter for each of the elements. The resulting array does not have any information about the has key values, and the order is not significant.

3.9 Lookup Tables

Lookup tables store a variety of values indexed on a certain column. An example is for storing the difference between UT1 and UTC, and the polar motion vector as a function of date.

One of the key functionalities of a lookup table is to read data from an ordinary text file into an array of vectors. This functionality is generally useful, and so we specify a separate function that may be called independently:

```
psArray *psVectorsReadFromFile(const char *filename, const char *format);
```

psVectorsReadFromFile shall return an array of psVectors, read from the specified filename. The file shall be plain text, consisting of an identical number of columns on each line, with the values separated by whitespace. Lines commencing with a comment character (the pound sign, #) and blank lines shall be ignored. The format is a scanf-like format which specifies the number of columns in the file, as well as their types. The following formats shall be defined: \%d for psS32, \%ld for psS64, \%f for psF32, and \%lf for psF64. A star (*) in the format shall indicate that the column is to be skipped.

```
typedef struct {
    const char *filename;               ///< File from which data is to be read
    const char *format;                 ///< scanf-like format string for file
    unsigned int indexCol;              ///< Column of the index vector (starting at zero)
    psVector *index;                    ///< Index values
    psArray *values;                    ///< Corresponding values: an array of vectors
    const psF64 validFrom, validTo;     ///< Range of validity
} psLookupTable;
```

filename shall specify the file from which the lookup table data is to be read. format shall contain a scanf-like format string specifying how the columns are to be interpreted (see psVectorsReadFromFile). indexCol shall specify the index of the column (with the first column having an index of zero) that will form the index values. index shall contain the index values, which shall be sorted in increasing order. The values shall consist of an array of vectors, each of the same length as the index vector. The vectors (including the index and all vectors in the values array) may be any numerical type except complex types. The validFrom and validTo shall specify the range of valid values for the index; in most cases, these will simply be the first and last indices.

The constructor shall be:

```
psLookupTable *psLookupTableAlloc(const char *filename, ///< File from which to read
                                  const char *format ///< scanf-like format string
  int indexCol ///< Column of the index vector (starting at zero)
                                  );
```

This function shall allocate a psLookupTable, and set the appropriate values, but it shall not read the lookup table. This is so that the lookup table can be specified at the initialisation of a program, but not read unless required.

The destructor shall free all the components.

```
psLookupTable *psLookupTableImport(psLookupTable *table, ///< Lookup table into which to import
                                   const psArray *vectors, ///< Array of vectors
                                   int indexCol ///< Index of the index vector in the array of vectors
                                   );
```

psLookupTableImport shall import an array of vectors into a table. If table is NULL, a new psLookupTable shall be allocated and returned. The array of vectors, which was likely generated by psVectorsReadFromFile, are imported by setting the table->index to the vector specified by the indexCol, and pointing the table->values array data to the remaining vectors in vectors. Reference counters for the vectors shall be incremented as appropriate. The validFrom and validTo members of the table shall be set to the first and last values in the index vector. If the index vector is not sorted in the file, the lookup table shall be sorted prior to the function returning.

```
int psLookupTableRead(psLookupTable *table);
```

psLookupTableRead combines psVectorsReadFromFile and psLookupTableImport to read the appropriate file and import the data into the extant table. If the input table has already been read from a file, the file shall be re-read, and the contents replaced. The function shall return the number of lines read (not including ignored lines).

Interpolation on a lookup table is performed by the following functions:

```
psF64 psLookupTableInterpolate(const psLookupTable *table, psF64 index, int column);
psVector *psLookupTableInterpolateAll(const psLookupTable *table, psF64 index);
```

Both functions shall interpolate the `table` at the provided `index`. For `psLookupTableInterpolate`, only the value in the specified `column` shall be calculated and returned. For `psLookupTableInterpolateAll`, all the values shall be calculated and returned as a `psVector`, the type of which shall be `PS_TYPE_F64`. If the `index` is beyond the range of the `table`, `psLookupTableInterpolate` shall return `NaN`, and `psLookupTableInterpolateAll` shall return `NULL` — that is, no attempt is made at extrapolation.

4 Data manipulation

We require a variety of basic data manipulation functions which will act upon data (in particular, arrays/vectors). We require the following capabilities:

- Bit masks;
- Vector and image arithmetic;
- Sorting;
- Statistics;
- Matrix operations and linear algebra;
- (Fast) Fourier Transforms;
- General functions; and
- Minimization and fitting routines.

4.1 BitSets

BitSets are required in order to turn options on and off. We require the capability to have a bitset of arbitrary length (i.e., not limited by the length of a `long`, say). The `psBitSet` structure is defined below. Note that the entry `bits` is an array of type `char` storing the bits as bits of each byte in the array, with 8 bits available for each byte in the array. Also note that the constructor is passed the number of required bits, which implies that `ceil(n/8)` bytes must be allocated. The bitset structure is define by:

```
typedef struct {
    int n;                              ///< Number of chars that form the bitset
    char *bits;                         ///< The bits
} psBitSet;
```

We also require the corresponding constructor and destructor:

```
psBitSet *psBitSetAlloc(int n);
```

where `n` is the requested number of bits.

Several basic operations on bitsets are required:

- Set a bit;