Naval Research Laboratory

Monterey, CA 93943-5502

NRL/MR/7531--98-7239

Binning and Sorting Software Package

TOM LOUGHEED

Atmospheric Dynamics & Prediction Branch
Marine Meteorology Division

December 1998

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. Agency Use Only *(Leave Blank)*.	**2. Report Date.** December 1998	**3. Report Type and Dates Covered.** Final

4. Title and Subtitle.
Binning and Sorting Software Pacakge

5. Funding Numbers.
PE 0603207N
PN X2342
AN DN153-161

6. Author(s).
Tom Lougheed

7. Performing Organization Name(s) and Address(es).
Naval Research Laboratory
Marine Meteorology Division
Monterey, CA 93943-5502

8. Performing Organization Reporting Number.
NRL/MR/7531-- 98 - 7239

8. Sponsoring/Monitoring Agency Name(s) and Address(es).
Space and Naval Warfare Systems Command (PMW 185)
4301 Pacific Highway, San Diego CA 92110-3127

10. Sponsoring/Monitoring Agency Report Number.

11. Supplementary Notes.

12a. Distribution /Availability Statement.

Approved for public release; distribution unlimited

12b. Distribution Code.

13. Abstract *(Maximum 200 words)*.

The iterative matrix solution used in NRL's three-dimensional variational analysis required that the observations be sorted into tiles with a prespecified maximum and minimum number in each. This is the software documentation for code that does the required sorting.

14. Subject Terms.
Binning and Sorting Software Package

15. Number of Pages.
24

16. Price Code.

17. Security Classification of Report. UNCLASSIFIED	**18. Security Classification of This Page.** UNCLASSIFIED	**19. Security Classification of Abstract.** UNCLASSIFIED	**20. Limitation of abstract.** Same as report

ACKNOWLEDGEMENT

This work is sponsored by the Oceanographer of the Navy through the Space and Naval Warfare Systems Command, Program Office PMW-185, under Program Element 0603207N.

Binning and Sorting Software Package

1 Introduction

This is the documentation for the Binning and Sorting algorithm implemented in the subroutines 'bin_sort' and 'latlon_sequencer2', versions of 5 September 1998.

The software documented here performs a function called "binning": input data are grouped or binned into smaller collections of similar data, which are small enough in number to be accommodated by subsequent analyses. In this case the similarity is that the data are near each other on the globe.

The subroutine 'latlon_sequencer2' is a front-end for the subroutine 'bin_sort.' It mimics the interface and function of the preexisting subroutine 'latlon_sequencer' in the program 'ob_sort', version of 31 August 1998. It receives as input arrays of observations and returns the same arrays sorted, along with a code number for each observation. The sorted observations are grouped into clusters of observations that are adjacent on the globe; all of the observations in one group have the same code number. The groups are organized so that the number of observations in each group falls below a count hard-coded in 'latlon_sequencer2' (but the value is a variable in every subroutine below it) if possible.

There are several constraints observed when observations are grouped into bins.

- Observations from the same site are always kept together, never split between two bins.
- The number of sites in any bin is never allowed to fall below an input minimum number. The size of a bin is never allowed to fall below an input minimum size. These two numbers and the desired maximum number of observations in a bin are hard-coded in the "glue" subroutine 'latlon_sequencer2' but are variables in the driver subroutine 'bin_sort' and every subroutine below it.
- The approximate diameter of the border of the region that surrounds every group of observations is not allowed to fall below an input size (in degrees). The minimum allowed size is also hard-coded in 'latlon_sequencer2' but is variable in every subroutine below it.

(The minimum number of sites per bin is currently set at 1, which effectively eliminates that as a constraint.)

For the purposes of these subroutines, a site is identified by a sequence of input observations that were consecutive in the input order, and all had the same (or very nearly the same) latitude and longitude. If two subsequences of observations have identical latitude and longitude but aren't consecutive, the two subsequences will be identified and treated as different observation sites. It is possible, although very unlikely, that two such observations would even be placed into different bins on output.

2 Overview of the Binning and Sorting Algorithm ('bin_sort')

The subroutine 'bin_sort' is the driver for several subroutines, which together divides a list of observations covering all or part of the earth into triangular regions, called "tiles" in the software, which then become bins. It is intended to be the main interface to the sorting software, and has both a Fortran-90 style interface, which requires a Fortran-90 "use" statement and Fortran-90 data structures ("type"), and a reduced Fortran-77 callable subroutine. It determines how to bin the input observations by covering the globe with tiles and then splitting the tiles. When the splitting is finished all observations that lie within the same tile are marked in the same bin, and their sort order is adjusted so that they will all be grouped together.

This section describes in general the input and output of 'bin_sort' and. briefly goes over the three stages of the algorithm in subroutine 'bin_sort', and identifies the subroutines that 'bin_sort' calls to accomplish each stage.

In addition to the subroutines named below, 'bin_sort' also calls several Fortran-90 "contained subroutines" in the first stage, but they are just a way of breaking up long subroutines into short ones. The contained subroutines are found in the same file as their container and they have in common all of the variables in their container. Their code could be cut and pasted without change into the location they are called in the containing routine (with the possible omission of variable declarations in the contained subroutines). None of them is mentioned below.

A brief note on vocabulary: The bins begin as the triangles of an icosahedron (20-sided regular solid) or an octahedron (8-sided regular solid). (The choice is currently hard-coded in subroutine 'bin_sort'.) The process of covering a surface with regularly shapes is called "tesselation" by mathematicians, and the shapes are called "tiles". "Tessera" is the Latin (Greek?) word for a tile. In the subroutines below the triangles are called "tiles," and the set of all the triangles needed to cover the globe is called a "tesselation".

2.1 Input and Output

The principle output returned from the 'bin_sort' subroutine is a pair of indirect index arrays, which convert from input to sorted order, and from sorted to input order.

Other information is returned as well: An array of data structures, each structure of which contains information about the size and shape of the corresponding tile (this is not available from the Fortran-77 interface to 'bin_sort'). An array that relates each observation number to an observation site number, and another array that gives the bin number for each observation. A pair of arrays that give the sines and cosines of each observation site, numbered by site number (hence the need to lookup site number for an observation).

2.2 Algorithm in Three Stages

The 'bin_sort' software runs in three stages, all managed inside the subroutine 'bin_sort'. The first two stages are binning and the third stage is sorting.

2.2.1 First Stage

In the first stage the world is covered with a few tiles (triangles) in a regular pattern. So far, the pattern is an icosahedron or octahedron. Each of the input observations is assigned to a site (normally many observations are made at any one site) and then each site is assigned to the tile whose edges contain the datum's coordinates. A "site" is identified as a sequence of consecutively indexed observations, all of which have the same latitude and longitude.

The subroutines called for the first stage are 'init_tess' to create the desired initial tesselation and 'init_asgn' to put the sites into the tiles of the tesselation.

2.2.2 Second Stage

In the second stage each of the tiles is iteratively cut into two or more smaller tiles that cover the same area, until either (a) the number of observations made within the tile is sufficiently small, (b) the tile is too small, or (c) the number of sites within the tile is too small. Input parameters to 'bin_sort' set the exact criteria for (a), (b), and (c), and these are the "tuning" parameters for the algorithm. Once none of the tiles can be cut any longer, the second stage is finished.

The subroutine called in the second stage is 'split_all'.

2.2.3 Third Stage

In the third stage a somewhat arbitrary sort order is assigned to each observation, with care being taken to ensure that observations at the same site will be indexed consecutively in sorted order the same as they were on input, and all of the observations at sites in the same tile are indexed consecutively as well. There is some small effort made in the ordering to make the observations from at least some of the tiles that are close together in space to be close together in index, but unlike the other desired results of the sort, it is not assured. (In general, it is impossible to make a 1-dimensional ordering of 2-dimensional data which keeps all locations that are close by in 2 dimensions also close by in 1 dimension.)

The third stage, creating the sort order output arrays, is performed inside 'bin_sort', only contained subroutines are called.

3 Notes

The following are notes that concern the entire program

3.1 Multiprocessing

Test-trials of the binning and sorting algorithm shows that it is very fast. This appears in the most part to be due to the fact that after the first stage, only sites' numbers are tracked and divided between tiles, and for any iteration of the slicing, only the site numbers that are listed inside a tile need to be searched through, not all the observations in the entire globe. If further speedup is desired, the algorithm can easily be multiprocessed in the first and second stage. The scheme is simple: a single process sets up or splits a single tile. The third stage, the so-called sorting stage, appears to be a poor candidate for parallelizing.

To multiprocess the first stage, the subroutines 'init_tess' (in file "bin_init_tess.f90"), 'init_conv' (in file "bin_init_conv.f90"), and 'init_assign' (in file "bin_init_asgn.f90") must all be slightly modified. Currently, both subroutines contain a simple, outer loop, which cycles over all of the input/output arrays of tiles. These "do" loops can be immediately parallelised by substituting a Fortran-95 "forall" loop.

Similarly, in the second stage inside subroutine 'split_all' (in file "bin_split_all.f90") there is a do loop that cycles through a linked list of tiles. Since this is not an array, a simple "do" becomes "forall" substitution is not possible, but the same idea applies: 'split_all' becomes a dispatching subroutine run in the "master" processor, and the subroutines it calls, 'split_one_a', 'split_one_b', and 'split_one_c', are executed in multiple "slave" processors. The master processor proceeds through the linked list as slave processors become available, and resumes splitting at the beginning of the loop after all of the child processes have finished.

The master processor should not wait at the end of its pass through the linked list for all of the dispatched slave processors to finish.

Rather a new logical variable imbedded in the linked list data structure 'tile_list_t' should be passed in as an argument to the subroutine performing the splitting: 'split_one_a', 'split_one_b', or 'split_one_c'. These subroutines, which are executed by the slave processors, will change the value of the flag for each of the tiles just before returning. Until the flag comes clear, the master will ignore the corresponding tiles' entries in the linked list, similar to but not the same as the variable 'finished' already imbedded in that structure. Note that different splitting subroutines would require different numbers of flags to be used. Subroutine 'split_one_b' for example operates on 4 tiles at a time, all 4 of which must be kept on hold while it is working, whereas method 'split_one_a' only operates on 2 tiles at once.

I expect that it will not significantly improve performance, but even more efficient multiprocessing is possible if the outer loop in all of the four "dispatching" or master processor subroutines 'init_tess', 'init_conv', 'init_assign', and 'split_all', is raised out of these subroutines and up into subroutine 'bin_sort'. Multiprocessing identical to that described for the second stage can then be performed, effectively combining the first and second stages. None of the algorithms in the subroutines 'split_one_a', 'split_one_b', and 'split_one_c' requires work in the first stage to have been completed for any tile other than the one passed in to the subroutine. Hence tiles whose initialization (first stage) is completed can proceed directly to be split (second stage) before all the other tiles have finished their first stage, if processors are available. It will however require considerable rewriting of the code to bring the entire first and second stages are managed at the same software level in subroutine 'bin_sort'.

The third stage, the so-called sorting stage, appears to be a poor candidate for any kind of parallelizing or multiprocessing. Unlike the second stage, for any given tile in the linked list it requires the prior stage (splitting) to have finished for all tiles listed before the given tile. Although that is not a complete bar to converting the third stage to multiprocessing, there probably would be little gain from doing so. The third step is just a single do-loop that scans once through the entire linked list of tiles, and assigns a number in sequence to each of the observations at each of the sites within the tile's "contents" array. Because this serial process is very quick - it's just sequential numbering, after all - it is very doubtful that parallelizing it would provide any great processing speed-up.

If it really is necessary to convert the third stage to multiprocessing, proceed as follows: In the second stage, retain the number of observations at all of the sites in a tile are summed, in order to determine if splitting has finished, stored in an integer variable the tile. While the master processor loops through the linked list, it will keep a running sum 'm' of the number of observations in all of the previously scanned tiles (finished splitting or not). For any tile that is finished splitting, the master processor hands off the running sum 'm' of observations in all preceding tiles, along with the finished tile, to the next available slave processor. The slave processor then enumerates the 'n' observations within the tile from 'm+1' to 'm+n'. This will enable the observations in the "finished splitting" tiles to be enumerated before the tiles before them have finished. This only works because the tiles, when split, are kept in the same order in the linked list as the parent tile from which they were split out. No such arrangement is possible if the tiles are tracked using an ordered array.

3.2 Improvements for Subroutine 'init_assign'

There is another improvement that can be made to speed up 'init_asgn'. Initially I planned to use a single formula to determine what tile a site belonged in. The formula is immediate and obvious for an octahedron. The same expression is used at NRL-Monterey, for determining which octant of the globe a site lies in.

For an icosahedron, there is no formula that works every time, but the icosohedron's triangles are organized into three bands: north cap, equatorial, and south cap. Except near the edges, the band is determined by latitude. Once the band is known, a simple formula on longitude determines either the one tile that a site belongs to, in the case of the north and south caps, in the case of the equatorial band, longitude determines which of two tiles a site belongs to, one must then be tested. Sites lying near the boundaries of the bands must be tested against several (or all) of the tiles in the bands separated by the boundary.

Neither of these was done, and the formulas become more complicated if the octahedron or icosahedron is tilted slightly to make points near the north and south the poles lie clearly in a single triangle, as was requested initially.

3.3 Notation and Coding Practices Used

The source code makes use of several new Fortran-90 features that will not be obvious to a reader only familiar with Fortran-77. One of the least obvious to the casual reader is array assignment. In Fortran-90, if "X" and "Y" are arrays of the same length and same length, "N", and data type, then the statement "`X = Y`" is equivalent to "`DO I=1, N; X(I) = Y(I); END DO`".

Everywhere I used array assignment and array addition in the code, I strived to append the unnecessary symbol "(:)" to arrays. (Doing so would trigger a compiler error if an array expression were invalid.) Doing so is not required by the Fortran-90 language, but because array assignment and array arithmetic is new to F90, I wanted to make the places in the code where it is used stand out.

A note on Fortran-90 modules: A "module" is a Fortran-90 language addition that acts as a container for common variables, subroutines and functions, and interfaces and data types. (Common variables, subroutines, and functions were part of Fortran-77, but interfaces and data types are new in Fortran-90.) The idea of modules is present in one way or another in most other languages (including 'C'). In Fortran-90, a module is intended to package together related variables and their data type definitions and the functions and subroutines that operate on them. When a module name appears in a "use" statement, the F90 compiler is then able look inside the previously compiled module and check to ensure that any subroutines or functions that are in the module are called in the using subroutine with the correct number and type of variables. Common, public variables at the beginning of a module are a direct substitute for named common blocks.

The "use" statement also provides the opportunity to rename the variable or subroutine being used, but that was only done in the dummy Fortran-77 interface subroutine 'bin_sort' (in file "bin_sort_f77.f90") which renames the original 'bin_sort' (from module 'bin_sort_m' in file "bin_sort.f90") and for the data type 'tile_t', which is renamed to 'bin_t' in the interface to the version 'bin_sort' from module 'bin_sort_m'. The Fortran-77 interface to 'bin_sort' omits the data type as an output argument because its declaration cannot be done using the Fortran-77 language.

3.4 Vocabulary

The word "bin" is not an abbreviation for anything (like "binary"); it is the English word for a small, box-like container.

The "bins" are tiles. It is common in mathematical and algorithmic texts to read the "tessellation" referring to all the tiles together. The word "tessellation" means "tiling" in Latin.

The comments that follow refer to a "site". It is a location at which a number of observations were made, which observations are to be moved around as an intact group. It's essentially equivalent to a reporting station. An "input observation" is a single measurement, which is the input to this subroutine. Variables dealing with sites generally begin with "s"; variables dealing with observations generally begin with "o".

3.5 Summary Tables

The following tables provide a roadmap to the source code.

Table 3.5.1 – Binning and Sorting Source Code Files

In almost of these files, there is only one subroutine, whose name matches the file-name (with "bin_" left out). In even more cases, the subroutine, function, and/or data structures are/is contained within a Fortran-90 module, and the name of the module is the same as the name of the file (but with "_m" added).

file name	contains
bin_assert.f90	Error checking flags imitative of 'assert' in 'C'.
bin_divide_cont.f90	Divides the sites in a split triangle between the two halves.
bin_earth_rad.f90	Single constant: earth radius. Currently set to 1.
bin_init_asgn.f90	Assigns input data to initial tesselation of the globe.
bin_init_conv.f90	Converts observation sites' latitudes and longitudes into vectors.
bin_init_tess.f90	Creates the initial tesselation of the globe.
bin_precis.f90	Defines the Fortran-90 "kind" numbers for number types.
bin_print_tiles.f90	Prints out a description of the current tesselation.
bin_sort.f90	Main driver subroutine.
bin_sort_f77.f90	Permits Fortran-77 to call otherwise inaccessible 'bin_sort.'
bin_split_all.f90	Manages the splitting of the entire list of triangles.
bin_split_one_a.f90	Split 1 triangle using method "A" (cut through center of data).
bin_split_one_b.f90	Split 1 triangle using method "B" (cut into 4 equal triangles).
bin_split_one_c.f90	Split 1 triangle using method "C" ("best" cut thru. side midpoint).
bin_test_cont.f90	Tests a triangle to make sure that its observations belong to it.
bin_tile.f90	Data structures and utility subroutines for 'tile' (triangle).
bin_vector.f90	Subroutines and operators for 3-vectors.
latlon_sequencer2.f90	Front-end for 'bin_sort' to glue it into the program 'ob_sort'.

Table 3.5.2 - Testing Source Code Files

file name	contains
string.f90	Logical functions for character and string conversion & classification.
getarg_inf.f90	F90 interface declaration for Unix "Get Command Arguments" function.
getenv_inf.f90	F90 interface declaration for Unix "Get Environment String" function.
qsort_f.c	Subroutine 'qsort_f', in 'C', that can be called from Fortran that calls the Unix sort Quick Sort function 'qsort'.
qsort_inf.f90	Declares an interface to 'qsort_f' for the Fortran-90 compiler.
sgn.f90	Mathematical 'segnum' function, which produces importantly different results than the mathematicly non-standard Fortran standard function 'sign'.
pr_graph.f90	Writes out a simple printer graph.
pr_graph_test.f90	Tests 'pr_graph'.
qsort_f_test.f90	Tests the Fortran interface written in 'C' for Unix utility 'qsort'.
rd_fgge.f90	Reads latitudes and longitudes from a FGGE format t-file for testing.
test_anal.f90	Subroutine that analyses the results of a test run.
test_dr5.f90	Driver program to test 'bin_sort' directly.

Table 3.5.3 - Subroutine Name, Module, and File Cross Reference

The files that make up the 'bin_sort' software are contained in the directory
`barker:/daley/data4/users/lougheed/bin_sort_delivery/`
And these are listing in the table below, along with their contents. Most subroutines and functions in these files are packaged as Fortran-90 modules, each of which is listed along with the functions and subroutines it contains. In the case where the file does contain a Fortran-90 module, there is always only one module per file, and in many cases only one subroutine per module.

file name	module name	subroutine/function name
bin_assert.f90	bin_assert_m	–
bin_divide_cont.f90	bin_divide_cont_m	divide_cont
bin_earth_rad.f90	bin_earth_rad_m	–
bin_init_asgn.f90	bin_init_asgn_m	init_asgn
bin_init_conv.f90	bin_init_conv_m	init_conv
bin_init_tess.f90	bin_init_tess_m	init_tess
bin_precis.f90	bin_precis_m	–
bin_print_tiles.f90	bin_print_tiles_m	print_tiles
bin_sort.f90	bin_sort_m	bin_sort
bin_sort_f77.f90	–	bin_sort
bin_split_all.f90	bin_split_all_m	split_all
bin_split_one_a.f90	bin_split_one_a_m	split_one_a
bin_split_one_b.f90	bin_split_one_b_m	split_one_b
bin_split_one_c.f90	bin_split_one_c_m	split_one_c
bin_test_cont.f90	bin_test_cont_m	test_cont
bin_tile.f90	bin_tile_m	tile_t, tile_list_t, add_site_tile, center_tile, clear_tile, copy_tile, harmonize_tile, is_on_edge_tile, is_small_tile, new_listing_tile, test_is_inside_tile
bin_vector.f90	bin_vector_m	dot, cross, cart_to_spheric, spheric_to_cart, normalize, vmag
latlon_sequencer2.f90	–	latlon_sequencer2
pr_graph.f90	pr_graph_m	pr_graph
qsort_f.c	–	qsort_f
rd_fgge.f90	rd_fgge_m	rd_fgge
sgn.f90		sgn
string.f90	string_m	is_digits, is_letters, tolower, toupper

Table 3.5.4 - Interface Definitions for Unix

The three files named below do not contain any actual subroutines or functions. Instead they contain interface declarations for the subroutines named in parentheses. These declarations tell the Fortran-90 compiler (when the module is named in a "use" statement) the number and types of arguments used in the Unix system calls. Because these modules contain no executable statements or data structures, they are not documented in the expanded module descriptions that follow.

file name	**module name**	**subroutine/function name**
`getarg_inf.f90`	`getarg_interface`	`(getarg)`
`getenv_inf.f90`	`getenv_interface`	`(getenv)`
`qsort_inf.f90`	`qsort_interface`	`(qsort_f)`

The file "`latlon_sequencer2.f90`" contains the single subroutine '`latlon_sequencer2`', which takes the same arguments as 'latlon_sequencer', calls 'bin_sort', and then re-orders all of its input arrays according the '`bin_sort`' returned order to produce similar results. Subroutine '`latlon_sequencer2`' can be called directly from Fortran-77 without any "`use`" statement. It was tested with code compiled by the Fortran-77 compiler (but necessarily linked with the Fortran-90 compiler) on machine 'barker' (a Sun Workstation) and worked without any apparent problem.

4 Module Descriptions

The following are descriptions of all of the modules and "naked" subroutines contained in the Binning and Sorting delivery directory.

4.1 Module 'bin_sort_m'

The module 'bin_sort_m' is the container for the driver subroutine 'bin_sort' and a generic subroutine 'bin_reorder' which takes the sort order from 'bin_sort' and performs the actual re-arrangement of data in an array. The subroutine 'bin_sort' is described above in the overview. Subroutine 'bin_reorder' is a generic name used for several nearly identical one-line subroutines which reorder data according to an index array.

Subroutines that 'bin_reorder' is a generic name for 'bin_reorder_i', which reorders arrays of default integers, 'bin_reorder_r', which reorders default reals, 'bin_reorder_dp', which reorders double precision reals, and 'bin_reorder_ch', which reorders arrays of any length character strings.

To make use of 'bin_sort' from a Fortran-90 program, put the statement "use bin_sort_m" at the very beginning of the calling subroutine, even before any "implicit none" statement. Then call 'bin_sort' as usual. There is also a Fortran-77 interface, which omits several returned variables that cannot be declared in Fortran-77. Without the "use" statement, any call to 'bin_sort' will default to the old-style Fortran 77.

The data-type name 'tile_t' is changed in 'bin_sort' to be 'bin_t' to conceal from the caller the details of how tiles are made - and to keep the nomenclature simple. The data returned through the variable of that type describes the shape and contents of the bin.

Subroutine 'bin_sort' is the parent subroutine for the bin sorter. It checks input and invokes the various service subroutines which implement the three steps involved in subdividing observations into small, manageable clusters.

4.1.1 Input

Subroutine 'bin_sort' takes as input parallel arrays describing the locations of observations, and how those observations have "companions" from the same location, which must be kept together.

4.1.2 Process

The first operation of 'bin_sort' is to combine the observations into "sites" and distributes them onto tiles (which so far happen to be triangles, but don't have to be so). The initial tiling of the earth is an icosahedron or an octahedron.

The second operation, and the meat of the subroutine is to subdivide those tiles into smaller and smaller tiles, until each tile either (1) contains 'max_bin_obs' tiles or fewer, or (2) the number of sites (or stations) in the tile has fallen below 'min_bin_stn', or (3) the size of the tile is smaller than 'min_bin_size_deg'. The result is that the input observations have been "binned" with adjacent observations, into groups that are small enough to be convenient for later calculations.

4.1.3 Output

Output from 'bin_sort' always includes two parallel arrays: 'sorted_input' and 'obs_bin_num'. The array 'sorted_input', gives the sorted order to put the observations into, so that observations that have been put into the same tile (triangle) list consecutively. The other array, 'obs_bin_num' gives the bin number into which the corresponding unsorted observation has been placed. Be warned: the bin/tile numbers are not sequential - there are gaps in the numbers, and tiles that are adjacent are not numbered in any ordered pattern. The input array 'obs_cnt' formerly was corrected on output (to mark singleton observations with a 1 instead of a 0), but no longer. Also always put out is the scalar 'status', which gives a return code - zero is good, all other values signify an error.

Regarding the output variables, which are pointers, they may be null if an error occurred in 'bin_sort'. It is the responsibility of the caller to test all the output pointers to see if they are "associated" (null or not) regardless of the return code ('status'). It is also the obligation of the caller to reallocate the pointers, including imbedded arrays within the tiles.

If the pointer output is not required, the user may call instead the entry point to 'bin_sort' in the module 'bin_sort_f77', which has no pointer arguments and correctly deal locates all of the pointer arrays that would otherwise have been passed back.

4.1.4 Code Organization

Subroutine 'bin_sort' invokes three initialization subroutines. The first to set up a starting tessellation (tiling) of the earth, the second to convert all of the input latitudes and longitudes into Cartesian 3-vectors, and the third to put the site of observations' index numbers into lists contained in each of the initial triangles.

After the initial setup, which includes several allocable structures being created in this subroutine, the subroutine 'split_all' is called to perform the main job of chopping up the initial tiles into smaller tiles, until each tile contains at most the maximum number of observations (note that it's observations counted, not sites).

Once the subdivision is complete, the results are transcribed into the output array(s) and those items, which have been allocated, are released.

4.2 Module 'bin_assert_m'

The module 'bin_assert_m' is meant to mimic the 'assert' header file in 'C'. It contains several logical variables as compile-time constants that are to be used to turn on and off debugging code. The imbedded variables turn on and off both terse and verbose messages, and control testing for errors. The tests verify certain simplifying assumptions made about input, output, and the results of calculations.

The intention of using these variables as compile time constants (Fortran 'parameter') was that they would control 'if' blocks and so create "dead code" (that is never executed) which any decent Fortran optimizing compiler would recognize and remove, so that there would be no penalty to retaining the extra tests in the code. If the size of the 'bin_sort' code is of no concern, they can be left as they are. But if the code seems large and that is of concern, if the compiler appears to not be removing the dead code, then either the constants will have to be declared locally in every module (instead of bringing them in via "use" statements) to make them more obvious to the compiler, or if that fails, the blocks of code the variables turn off may have to be commented out by hand.

4.3 Module 'bin_print_tiles_m'

Subroutine 'print_tiles' is a helper routine for the bin sorter. It writes an ASCII file describing the current tessellation. The contents of the resulting ASCII file are columns that list tile id number, the latitude and longitude of each of the three vertices of the tile, and the number of contained sites and the number of observations at those sites. Other than the counts of sites and observations, there is no output regarding the contents of the tile.

4.4 Module 'bin_precision_m'

The module 'bin_precision_m' is a source for compiler- and platform-specific "kind" codes for the precision used in the program 'bin_sort'. "Kind" numbers are integer codes new to Fortran-90 that are used to declare variables. According to the F90 standard, they are not uniform from compiler to compiler, or from machine to machine, and the use of symbolic names for the code numbers is advised.

The names used for the codes are abbreviations for old Fortran-77 data types, e.g. 'i4' for "INTEGER*4". The numeric codes for the Sun compiler are the same as the old IBM "star" codes, which gave the number of bytes used to hold the data type. Hence 'i4' has value 4, for the Sun F90 compiler. When the code is ported to a different compiler, the numbers will have to be changed to accommodate the new compiler's "kind" codes.

4.5 Module 'bin_init_tess_m'

Subroutine 'init_tess" creates the beginning tessellation ("tiling") of the globe. Currently it does so using either an octahedron or dodecahedron. Input is the already-allocated space for the data to reside (there must be either at least 8 or 24 allocated tiles, indexed starting at 1). On output, the space held by the input variables will have been filled with data.

A note on genealogical information for the starting tiles: Every tile contains optional information on the identification number of the parent is that it was split from, and the generation it is in. By arbitrary convention, this subroutine makes all of the initial tiles generation 0, so that the generation number is also the number of splits performed to produce the tile. The fake parent used for this generation has tile identification number '0', and each tile contains its own identification number for its founder.

In addition to monitoring the splitting process, these numbers were inserted into the tiles so that the final, split tiles can be sorted together, if desired, to more closely bunch the tiles. This happens anyway, in the current implementation because it uses pointers and linked lists, but if the code is converted to Fortran-77 the ordering will not happen automatically and these numbers will become necessary.

The ordering will keep close data close because each founder I.D. number will represent a region of the globe, and founder numbers that are adjacent represent adjacent regions (mostly).

The two internal subroutines are named 'icosahedron' and 'octahedron'. Each creates a tesselation matching its name, and the one called is selected based on the number of tiles that were passed in as an input array argument. If different shapes are desired for the initial tesselation, or the initial tesselation is required to be loaded in from a file, new subroutines can be added.

Also in the module are stubs for subroutines to rotate the tesselation away from the pole by a few degrees. This subroutine was considered, but never implemented. All it needs to do is create a 3x3 rotation matrix to rotate the tiles' vertices in the desired direction, and then multiply through every 3-vector in each tile by the matrix.

4.6 Module 'bin_init_conv_m'

Subroutine 'init_conv' converts observations from spherical coordinates to Cartesian coordinates. That is, it converts the observations' latitude and longitude (and altitude) into the Cartesian vectors that will be needed for the calculations. Note that although altitudes are nominally part of an observation's location, altitudes are not used in this calculation.

At the same time, the multiple input observations, numbered from 1 ... 'nobs', are consolidated into sites numbered 1 ... 'nsites', without any gaps in the numbering and without any sites with repeating locations. (Sites, which aren't adjacent, can duplicate.)

Note that this conversion will be made for many sites, so it should be encoded to be as efficient as possible. But it is only done once for each location, not for every observation at that location or "site."

4.7 Module 'bin_init_asgn_m'

Subroutine 'init_asgn' puts sites in the 'contents' list of the input tiles. This subroutine assigns its input observations' index numbers (not the whole site itself, just its identification number) to the contents lists of the given tiles. The content lists are dynamically re-sized as necessary.

4.8 Module 'bin_earth_rad_m'

Module 'bin_earth_rad_m' encapsulates the variable 'earth_radius'. This file contains a global compile-time constant used to establish the scale length for position vectors for sites. This is only useful for output. The current value of 1 for 'earth_radius' makes all position vectors unit vectors. This is not exploited anywhere in the rest of 'bin_sort'. The code, which uses this, is written so that the radius of the earth could be anything (like 6,378,140 m). However, it has never been tested with any other value.

Note that the 'bin_sort' software is not written to adapt to different values of 'earth_radius' at different positions (when using geodetic coordinates, for example).

4.9 Module 'bin_divide_cont_m'

Subroutine 'divide_cont' was intended to be a utility subroutine that determines which of two tiles a set of observations belongs to. Because it contains no code to identify the side along which the two input tiles adjoin, in the worst case it tests a sites position vector against every side of each tile, when it only really needs to test the adjoining side. For that reason it is not used everywhere. Subroutines with the name 'divide_cont' are specially coded for and contained in modules 'bin_split_one_b' and 'bin_split_one_c'.

Please note that this module also contains the subroutine 'divide_deallocate'. In order to split lists of unknown length, subroutine 'divide_cont' dynamically allocates storage space large enough to encompass any list of site numbers it has seen thus far. In order to avoid pointless reallocation and reallocation, the space is kept from call-to-call, and only reallocated if it needs to grow. At the end of execution of 'bin_sort' a call must be made to 'divide_deallocat' to cause that storage space to be released.

4.10 Module 'string_m'

The module 'string_m' contains several functions and subroutines for identifying character data and for conversion of character data from upper to lower case and vice-versa, like the 'C' string library. They are:

Name	Description
`is_digits`	True if character string is all digits ("0" - "9").
`is_digit`	True if single character is a digit.
`is_letters`	True if character string is all letters ("A" - "Z", "a" - "z" or "_").
`is_letter`	True if single character is a letter.
`toupper`	Converts all characters in a string to upper case.
`tolower`	Converts all characters in a string to lower case.

These are used in the driver programs that are used to test 'bin_sort', but not used in any of the "bin_sort" software itself.

4.11 Module 'bin_split_all_m'

This module is a container for subroutine 'split_all' which is the driver/organizer for the second stage of the binning and sorting algorithm. It chops up all tiles in the linked list into small enough pieces that they can serve as bins. In this context "small" means either there are few enough observations in the tile, or that the tile itself has shrunk in size to be so small that it ought not to be divided further, or that the tile contains the minimum number of observing sites (at least 1).

The work of splitting the tiles is divided into two parts: this part manages the traversing of the linked list (which begins at 'start_lnk') that keeps track of the allocated tiles and the allocation of new links and new tiles. It calls one of three subroutines 'split_one_a', 'split_one_b', or 'split_one_c' to take an old tile and an empty new tile and actually split the first up and put half of the observations into the second.

It is in 'split_all' that the linked list of tiles is managed, new tiles are created, and the decision is made to split a tile or not. A tile is split if (1) it has too many observations and (2) it contains more the minimum number of sites (which is at least 1 by default), and (3) it's big enough to split. If the current tile doesn't contain enough sites, or if it's too small, or if the number of observations is small enough, the tile will not be split and will be marked as finished (linked list entry has a logical variable named 'finished' and that is set to true). Once a tile is marked as "finished", it is not modified any further.

4.12 Module 'bin_split_one_a_m'

This module is a container for the subroutine 'split_one_a' which is a service subroutine for 'split_all'. It divides a single tile into two tiles. The algorithm is to make a cut from one of the three vertices through the "center of mass" of the data. The 3 different cuts through the three different vertices are then compared. The cut, which makes the smallest angle in both of the two tiles, combined as large as possible is chosen. (Without that criterion, the algorithm tends to make some tiles that are very long, skinny triangles.)

4.13 Module 'bin_split_one_b_m'

This module is a container for subroutine 'split_one_b' which is a service subroutine for 'split_all'. It divides a single tile into four tiles of identical shape, by making a cut through the original tile between the midpoints of every pair of sides in the triangular tile. It contains a specially coded version of the 'divide_contents' subroutine written to perform a minimum number of calculations required to determine which of the 4 tiles any site falls in. Unlike 'split_one_a' and 'split_one_c' it performs no trial cuts, which causes those subroutines to perform 2/3 of their calculations for cuts that are not actually made (not counting the division of sites between tiles). Perhaps for this reason, this subroutine is the one with the fastest performance.

4.14 Module 'bin_split_one_c_m'

This module is a container for subroutine 'split_one_c' which is a service subroutine for 'split_all'. It divides a single tile into two tiles. The algorithm is to make a cut from one of the three vertices to the midpoint of the opposing side. The 3 different cuts through the three different vertices/side pairs are then compared. The cut, which either (1) makes the most equal distribution of observations between the two tiles or (2) creates a completely empty tile, is chosen. An additional criterion is in place that rejects a cut if the resulting tile has too sharp a corner - the amount of sharpness is hard-coded (as the square of the sine of the half-angle).

4.14.1 Module 'bin_test_cont_m'

The module 'bin_test_cont_m' is a container for the subroutine 'test_contents' which tests a tile's listed contents and returns a 'good' status if contents do indeed belong in the tile. This subroutine is for error checking only. It will not be used in a thoroughly debugged program, which has been compiled with assertion testing turned off.

4.15 Module 'bin_tile_m'

Module 'bin_tile_m' is somewhat unlike the other modules in that it is a complete package of data types (structures) and the subroutines that manipulate them. The data types are 'tile_t' and 'tile_list_t'. The 'tile_t' data type is a data structure that holds the identification numbers, shape, location, genealogy (parent identification numbers), and contents (as sites) of a tile. Much of the information in the data structure is redundant to avoid duplicate calculations. The data structure 'tile_list_t' is a linked-list element that contains a 'tile_t' element and informational flags.

As the software is currently written, the tiles are triangular. Only the subroutines contained within module 'bin_tile_m' and subroutines 'split_one_a', 'split_one_b', and 'split_one_c', are coded with knowledge of this fact. If later on a different tile shape is desired, only those need be changed. All of the general code (not specific to a particular tile-splitting algorithm) is contained in module 'bin_tile_m'.

Note that the object called 'tile_t' here is renamed to 'bin_t' in the main subroutine 'bin_sort'.

Table 4.15.1 Contents of Module 'bin_tile_m'

The following data, data-types, and functions and subroutines are inside module 'bin_tile_m'.

What	Name	Description
constant	`null_site_num`	Value used for tiles' contents lists when they contain no sites.
type	`tile_t`	Data type containing information about the shape, position, and contents of a tile (currently a triangle).
type	`tile_list_t`	Data type for unidirectional linked list management with imbedded 'tile_t' data type.
subroutine	`new_tile_listing`	Allocates memory for a single variable of type 'tile_list_t', fills it with null data, and returns a pointer to its location.
subroutine	`clear_tile`	Fills a 'tile_t' variable with null values.
subroutine	`copy_tile`	Copies data from one 'tile_t' variable to another (deals witht the complication of an imbedded array pointer).
subroutine	`harmonize_tile`	Very important subroutine sets consistent values for all redundant variables within a 'tile_t' structure, based on the nominal verticies (also elevates vertices to exactly 1 earth-radius).
subroutine	`add_site`	Put a new site I.D. code into a 'tile_t' structure's 'contents' list, expanding the list if necessary.
function	`tile_too_small`	Returns 'true' if a tile tests as too small for further splitting.
subroutine	`tile_center`	Calculates latitude and longitude of the approximate center of a tile.
function	`is_on_edge`	Returns 'true' if a site lies on the edge of a tile.
subroutine	`test_if_inside`	Determines if a tile is clearly within a tile or near to one of its edges.

Please note that a considerable number of details are not documented here about the interpretation and use of internal variables in the structure 'tile_t' and found in the source file "bin_tile.f90"; they are arbitrary conventions and so do not affect an overview of the software, but are crucial to understand in order to modify the software. In particular the respective numbering of tiles' verticies and edges, the carrying and maintenance of tile identification numbers (including hidden tile genealogical data), and the exact formulas used to determine the "size" and the "center" of a tile.

4.16 Module 'bin_vector_m'

The module 'bin_vector_m' provides dot and cross products of 3-vectors (real arrays of size 3), and utility subroutines that operate on 3-vectors.

The cross and dot product are provided as Fortran-90 operators named '.cross.' and '.dot.', which are equivalent to calling the functions 'cross' and 'dot'.

There is also an interface declaration for the generic subroutine 'normalize' that allows 'normalize2' and 'normalize3' both to be called by the same name. They are subroutines that convert a 3-vector to a unit vector (if it isn't zero). The one with 3 arguments returns the length of the original 3-vector. The one with two arguments doesn't.

The function 'vmag' which returns the vector magnitude of its argument.

These two subroutines 'spheric_to_cart_r' and 'spheric_to_cart_d' convert latitude and longitude (spherical) into a 3-vector (Cartesian). The first, with suffix "_r" in its name takes its angular arguments in radians. The second, with suffix "_d" in its name takes its angular arguments in degrees. Similarly subroutines 'cart_to_spheric_r' and 'cart_to_spheric_d' convert a 3-vector (Cartesian) into latitude and longitude (spherical).

4.17 Module 'string_m'

The module 'string_m' contains several Fortran string testing functions inspired by the standard C libraries 'ctype.h' and 'string.h'.

Table 4.17.1 Contents of Module 'string_m'

The following items are contained within the module 'string_m'.

Name	Description
`is_digits`	Test string argument to see if it is only decimal digits.
`is_digit`	Test single character argument to see if it is a decimal digit.
`is_letters`	Test string argument to see if it is only letters.
`is_letter`	Test single character argument to see if it is only letters.
`toupper`	Convert string to all upper case letters.
`tolower`	Convert string to all lower case letters

This utility function 'is_digits' returns 'true' if all of the characters in its input string are decimal digits (or spaces).

The utility function 'is_letters' returns 'true' if all of the characters in its input string are alphabetic characters, or spaces, or the underscore (_), or the asterisk (*).

4.18 Subroutine 'pr_graph'

This subroutine prints a histogram. Normally it is used to show the frequency of values observed in a data set, which may be helpful in detecting decoding errors

4.19 Quick Sort

The test software sorts results in order to prepare histograms. The following are used to perform that sort. The implementation of Quick Sort actually used is the one provided for access via the 'C' programming language. On Unix systems it is usually supplied by the computer hardware manufacturer in the operating system object libraries, written in a form optimized for the computer it is running on.

4.19.1 Quick Sort: Module 'qsort_inf'

The module 'qsort_inf' is an interface declaration only that is used by the Fortran-90 compiler to determine the number and types of arguments to the subroutine 'qsort_f' that provides access to the system function 'qsort'.

Due to a design flaw in F90 language, this must be specially coded up every time one wants to use a different 'qsort_t' data type. The interface declaration for 'qsort_f' stays the same. In order to avoid that, this is limited to indexed arrays.

4.19.2 Quick Sort: Subroutine 'qsort_f'

Despite the "_f" in its name, subroutine 'qsort_f' in file "qsort_f.c" is written in the 'C' programming language. Its arguments are written so that it can be called from Fortran-77 or Fortran-90.

4.20 Module 'rd_fgge_m'

The module 'rd_fgge_m' is just a container for subroutine 'rd_fgge', which reads a FGGE format data file and creates fake data records with the same latitudes and longitudes of the original FGGE data. The number of observations created at any one site is arbitrarily set to the number of FGGE data records.

4.21 Module 'sgn_m'

Module 'sgn_m' is a container for function '`sgn`'. The function '`sgn`' provides the standard mathematical function '`sgn`', which is an abbreviation for Latin "segnum", not English "sign", and is not the same as the Fortran intrinsic function 'sign'. It is used for the Quick Sort calls in the test subroutines.

Regardless of the input scalar type to function '`sgn`', it returns a standard integer value of -1 if the input scalar argument is negative, +1 if its argument is positive, and 0 if its argument is zero.

The standard Fortran sign transfer function, 'sign(a,b)' is not the same, and cannot be substituted, because of its unsatisfactory handling of zero.

4.22 Undocumented Programs

The following programs are not documented. They are test programs used to check the 'bin_sort' subroutine and are included in the directory with its software. Note that they can be distinguished from the Binning and Sorting software itself since they do not start with "bin_".

• pr_graph.f90	writes out a simple printer graph.
• pr_graph_test.f90	Tests 'pr_graph'.
• qsort_f_test.f90	Tests the Fortran interface written in 'C' for Unix utility 'qsort'.
• rd_fgge.f90	Reads latitudes and longitudes from a FGGE format t-file for testing.
• test_anal.f90	Subroutine that analyses the results of a test run.
• test_dr5.f90	Driver program to test 'bin_sort' directly.