# Introduction

## Function Philosophy

### Passing Text

Text is exchanged with the functions of the *model\_interface* module using one-dimensional character arrays. These are converted to FORTRAN CHARACTER variables internally. Presumably these character arrays are passed by address. Hence the receiving program does not know the length of the array. So it assumes a conservative length. The length does not matter unless a *model\_interface* module function writes to the array. Instances of this have been kept to a minimum. In programming that has been undertaken so far, this only occurs in the *retrieve\_error\_message()* function. Documentation of this function specifies the maximum number of elements that will be written, and hence the recommended size of the user-supplied receiving character array.

It is important that a user terminate any character array that it supplies to *model\_interface* functions using an easily recognized character. In accordance with C convention, this character is “\0” (i.e. ACHAR(0)). in accordance with C convention. It is easy to employ another termination code if this is preferable.

### Avoidance of Passing Text

Because passing text is painful, integer flags are used instead of text flags where this is possible, for example to indicate file type, precision type, etc. (Note that the FORTRAN standard allows for C-interoperability of FORTRAN CHARACTER variables – up to a point. However this functionality is not employed in *model\_interface* functions in order to avoid any possibility of error in argument passing.)

## Function Return Value

All functions return an integer value. This value is zero unless an error condition arises. An error message can then be retrieved using function *retrieve\_error\_message()*. This function call must be made immediately after the function call in which the error condition was encountered.

# calc\_mf6\_interp\_factors

## Description

Function *calc\_mf6\_interp\_factors()* calculates interpolation factors from a MODFLOW 6 DIS or DISV grid to a set of user-supplied points. It records these factors in a text or binary interpolation factor file. This file can be used by other *model\_interface* module functions such as *interp\_from\_mf6\_depvar\_file()*.

## Function Call

|  |
| --- |
| integer function calc\_mf6\_interp\_factors( &  gridname, &  npts,ecoord,ncoord,layer, &  factorfile, factorfiletype, &  blnfile, &  interp\_success)  character (len=1), intent(in) :: gridname(LENGRIDNAME)  integer, intent(in) :: npts  double precision, intent(in) :: ecoord(npts),ncoord(npts)  integer, intent(in) :: layer(npts)  character (len=1), intent(in) :: factorfile(LENFILENAME)  integer, intent(in) :: factorfiletype  character (len=1), intent(in) :: blnfile(LENFILENAME)  integer, intent(out) :: interp\_success(npts) |

## Return Value

The return value is zero unless an error condition is encountered, in which case the return value is 1. An error message can then be retrieved using function *retrieve\_error\_message()*.

## Function Arguments

|  |  |
| --- | --- |
| **Argument** | **Role** |
| gridname | The user-supplied name of an installed MODFLOW 6 grid specification. |
| npts | The number of points for which interpolation factors must be calculated. |
| ecoord, ncoord | The eastings and northings of points to which interpolation must take place. |
| layer | The model layer numbers to which interpolation target points belong. |
| factorfile | The file in which interpolation factors must be stored. |
| factorfiletype | Provide as 0 for a binary interpolation factor file, and 1 for an ASCII (i.e. text) interpolation factor file. |
| blnfile | If this is provided with a nonblank value, then *calc\_mf6\_interp\_factors()* writes a SURFER BLN file denoting polygons that contain interpolation target points. This polygon is a triangle for DISV grids, or a rectangle (for DIS grids). |
| interp\_success | This is returned as 1 for points to which interpolation can take place, and as 0 for points to which interpolation cannot take place. |

## Notes

### Interpolation Methodology

For a DIS grid, function *calc\_mf6\_interp\_factors()* implements bilinear interpolation from the four cell centres that surround each interpolation point (normally points at which system state observations have been made). If an observation point is not surrounded by four cell centres (this occurs if it lies outside of the model domain, or is too close to the edge of the model domain), then *calc\_mf6\_interp\_factors()* reports its inability to interpolate to that point through the pertinent element of the *interp\_success* array.

For a DISV grid, *calc\_mf6\_interp\_factors()* attempts to construct a triangle of connected cell centres to enclose each point to which spatial interpolation is required (normally points at which system state observations have been made). If it cannot build a triangle, it tries to construct a quadrilateral of connected cell centres. If it cannot build a quadrilateral of enclosing, connected cell centres around a particular point, then *calc\_mf6\_interp\_factors()* declares that interpolation cannot take place to that point. If a triangle can be found, interpolation is barycentric. If a quadrilateral must be used, then *calc\_mf6\_interp\_factors()* performs barycentric interpolation from the two quadrilateral-forming triangles that enclose the observation point and averages the result.

It is important to note that all interpolation is two-dimensional. Interpolation to a particular point takes place only from cell centres which belong to the same model layer as that to which the point is assigned.

### Coordinates

The geographical coordinates that are assigned to points to which spatial interpolation must take place should be consistent with those of the model. These can be real world coordinates, as MODFLOW 6 grid specifications include numbers which link a model grid to the real world. (XORIGIN, YORIGIN and ANGROT). These are applied when calculating interpolation factors.

### The Interpolation Factor File

No point names are provided to subroutine *calc\_mf6\_interp\_factors()*. Hence no point names are stored in the interpolation factor file that is written by this function. It is assumed that other functions of the *model\_interface* module that use this file are aware of the ordering of points.

As a quality-assurance measure, point details (i.e. the eastings, northings and layers of points) are recorded before interpolation factors in the interpolation factor file that is written by function *calc\_mf6\_interp\_factors()*. All user-supplied points are cited in this table, regardless of whether interpolation can take place to any particular point or not. The following table of interpolation factors also features all user-supplied points. However the integer which indicates the number of cell centres from which interpolation takes place to any point is assigned a value of zero for points to which spatial interpolation is impossible; no interpolation factors are associated with such points.

Entries on each line in the interpolation factor table are as follows:

1. The number of cell centres from which interpolation takes place to the point;
2. (Integer, real number) pairs comprised of the index of a cell centre and the interpolation factor associated with that cell centre.

# extract\_flows\_from\_cbc\_file

## Description

Function *extract\_flows\_from\_cbc\_file()* reads a cell-by-cell flow term file written by any version of MODFLOW (including MODFLOW-USG). It accumulates flows (normally pertaining to boundary conditions) over user-specified zones. Flows are stored in arrays that can be used by function *interp\_to\_obstime()* to conduct time-interpolation to observation times.

## Function Call

|  |
| --- |
| integer function extract\_flows\_from\_cbc\_file( &  cbcfile,flowtype,isim,iprec, &  ncell,izone,nzone, &  numzone,zonenumber, &  ntime,nproctime, &  timestep,stressperiod,simtime,simflow)  character (len=1), intent(in) :: cbcfile(LENGRIDNAME)  character (len=1), intent(in) :: flowtype(17)  integer, intent(in) :: isim  integer, intent(in) :: iprec  integer, intent(in) :: ncell  integer, intent(in) :: izone(ncell)  integer, intent(in) :: nzone  integer, intent(out) :: numzone  integer, intent(out) :: zonenumber(nzone)  integer, intent(in) :: ntime  integer, intent(out) :: nproctime  integer, intent(out) :: timestep(ntime)  integer, intent(out) :: stressperiod(ntime)  double precision, intent(out) :: simtime(ntime)  double precision, intent(out) :: simflow(ntime,nzone) |

## Return Value

Function *extract\_flows\_from\_cbc\_file()* returns a value of zero unless an error condition is encountered, in which case it returns a value is 1. An error message can then be retrieved using function *retrieve\_error\_message()*.

## Function Arguments

|  |  |
| --- | --- |
| **Argument** | **Role** |
| cbcfile | The name of a cell-by-cell flow term file produced by any version of MODFLOW. |
| flowtype | Text that identifies a flow type of interest. This variable is used in a case-insensitive manner. It should uniquely identify a 16-character text string that precedes flow-type-pertinent tables or arrays in *cbcfile*. |
| isim | An integer code which informs *extract\_flows\_from\_cbc\_file()* of the simulator that generated the binary file:  1 traditional MODFLOW;  21 MODFLOW-USG with structured grid;  22 MODFLOW-USG with unstructured grid;  31 MODFLOW 6 with DIS grid;  32 MODFLOW 6 with DISV grid;  33 MODFLOW 6 with DISU grid. |
| iprec | Indicates the precision with which real variables are stored in *cbcfile*. A value of 1 indicates single precision while a value of 2 indicates double precision. |
| ncell | The number of cells in the model grid. For a structured grid model this is ncol ᵡ nrow ᵡ nlay. For a MODFLOW 6 DISV model this is NCPL ᵡ NLAY. |
| izone | An integer zonation array. It associates an integer with each cell of the model grid. Cells with the same integer belong to the same zone. Cells that are assigned an integer value of zero do not belong to any zone. |
| nzone | A user-supplied positive integer which must equal or exceed the number of different zones which are featured in the *izone* array, that is the number of different non-zero-valued integers that are featured in this array. |
| numzone | The number of different zones in the *izone* array as determined by function *extract\_flows\_from\_cbc\_file()*. If *numzone* is found to exceed *nzone*, then *extract\_flows\_from\_cbc\_file()* will report an error condition. |
| zonenumber | An array which lists *izone* zone numbers in order of increasing value. Note that zero is not a zone number. |
| ntime | A user-supplied number which should equal or exceed the number of simulation times for which cell-by-cell flow terms are recorded in *cbcfile*. This is also the size of the *timestep*, *stressperiod*, and *simtime* arrays provided to function *extract\_flows\_from\_cbc\_file()*. It is also the leading dimension of the *simflow* array. All of these arrays are filled by function *extract\_flows\_from\_cbc\_file()*. |
| nproctime | The number of processed times. This is less than *ntime* if fewer than *ntime* simulation times are featured in *cbcfile*. If there are more than *ntime* simulation times featured in *cbcfile*, then *nproctime* is limited to *ntime*. |
| timestep | The simulation time steps for which *flowtype* flows are recorded in *cbcfile*, for up to *nproctime* output times. |
| stressperiod | The simulation stress periods for which *flowtype* flows are recorded in *cbcfile*, for up to *nproctime* output times. |
| simtime | Simulation times for which *flowtype* flows are recorded in *cbcfile*, for up to *nproctime* output times. |
| simflow | The flow in a particular zone at a particular output time. |

## Notes

### Zones

Function *extract\_flows\_from\_cbc\_file()* accumulates flows over zones. Zones are defined in the *izone* integer array. *extract\_flows\_from\_cbc\_file()* identifies all of the different non-zero integers that occur in this array and counts them. The total count is equal to *numzone*. It then sorts these integers in order of increasing value and returns them in the *zonenumber* array. Zones can then be referenced by array index in functions such as *interp\_to\_obstime()*. The *zonenumber* (and some other) arrays are given a dimension of *nzone* by the user. If *extract\_flows\_from\_cbc\_file()* establishes that *nzone* does not equal or exceed *numzone* it reports this as an error condition.

### Contents of a Cell-by-Cell Flow Term File

The headers to all arrays and tables that appear in a cell-by-cell flow term file can be ascertained using function *inquire\_modflow\_binary\_file\_specs()*. Each array header includes a 16 character text identifier for each flow type at each output time. A user should provide one of these headers as the *flowtype* input variable (or enough of this header to allow its unique identification). Note, however, that *extract\_flows\_from\_cbc\_file()* will not extract flows between one cell and another internally to a model grid; it reports an error message if asked to do so. It is used to accumulate flows over part or all of a boundary condition type; that is, it accumulates flows of a certain type in or out of a model domain.

### Simulation Times

The headers to flow arrays in cell-by-cell flow term files produced by newer versions of MODFLOW include the simulation time to which each array pertains. This is not the case for earlier versions of MODFLOW. Hence *extract\_flows\_from\_cbc\_file()* reports time steps and stress periods pertaining to accumulated flows in addition to the model simulation time that is associated (or not) with these accumulated flows. Where model simulation times are not provided in a cell-by-cell flow term file, a dummy simulation time of ‑1.0 is recorded by *extract\_flows\_from\_cbc\_file()*. It is the user’s responsibility to fill this array with correct simulation times prior to providing it to a function such as *interp\_to\_obstime()* that undertake time-interpolation of model outputs to user-supplied times of interest.

### The *simflow* Output Array

The *simflow* array is dimensioned by the user. It is filled by function *extract\_flows\_from\_cbc\_file()*. If *ntime* exceeds the number of simulation times for which flow terms are reported in *cbcfile*, and/or *nzone* exceeds the number of zones that are present in the *izone* array, then pertinent elements of *simflow* cannot be filled by *extract\_flows\_from\_cbc\_file()*. These elements are provided with values of zero.

The second dimension of the *simflow* array links flows to zones. Linkage is by order. Thus the contents of *simflow(i,j)* are the accumulated flows in zone *j* at time *i*. The zone pertaining to index *j* is the *j*’th element of the *zonenumber* array. The time associated with index *i* is the *i’*th element of the *simtime* array. Note that a user of function *extract\_flows\_from\_cbc\_file()* does not need to be aware of its internal array indexing conventions. However, this does not apply to function *interp\_to\_obstime()* which uses this array. When calling the latter function, a user can assume that zonal indexing begins at zero.

# free\_all\_memory

## Description

This function deallocates all arrays that are used by the *model\_interface* module for persistent storage.

## Function Call

|  |
| --- |
| integer function free\_all\_memory() |

## Return Value

Function *free\_all\_memory()* returns a value of zero. In the unlikely event that arrays cannot be deallocated, it returns a value of 1. An error message to this effect can then be accessed using the *retrieve\_error\_message()* function.

## Function Arguments

Function *free\_all\_memory()* has no arguments.

# inquire\_modflow\_binary\_file\_specs

## Description

Function *inquire\_modflow\_binary\_file\_specs()* reads a binary, dependent-variable or budget file that is written by MODFLOW, MODFLOW-USG or MODFLOW 6. It informs the user of the following file specifications:

* whether the file is recorded in single or double precision;
* the number of separate arrays that are recorded in the file;
* the number of output times that are represented in the file. (This may be required for dimensioning arrays prior to making calls to functions such as *interp\_from\_structured\_grid()*.)

Optionally, *inquire\_modflow\_binary\_file\_specs()* will write a tabular data file that lists all elements of the header to each array that it finds in the simulator output file.

## Function Call

|  |
| --- |
| integer function inquire\_modflow\_binary\_file\_specs &  filein, fileout, isim, itype &  iprec, narray, ntime)  character (len=1), intent(in) :: filein(LENFILENAME)  character (len=1), intent(in) :: fileout(LENFILENAME)  integer, intent(in) :: isim  integer, intent(in) :: itype  integer, intent(out) :: iprec  integer, intent(out) :: narray  integer, intent(out) :: ntime |

## Return Value

The return value is zero unless an error condition is encountered, in which case it is 1. An error message can then be retrieved using function *retrieve\_error\_message()*.

## Function Arguments

|  |  |
| --- | --- |
| **Argument** | **Role** |
| filein | The name of the MODFLOW-generated binary file that must be read. The length of the filename should be 256 characters or less. |
| fileout | The name of a file to which function *inquire\_modflow\_binary\_file\_specs()* will write a table of array headers. (Supply this as blank for no output file.) The length of the filename should be 256 characters or less. |
| isim | An integer code which informs the function of the simulator that generated the binary file:  1 traditional MODFLOW;  21 MODFLOW-USG with structured grid;  22 MODFLOW-USG with unstructured grid;  31 MODFLOW 6 with DIS grid;  32 MODFLOW 6 with DISV grid;  33 MODFLOW 6 with DISU grid. |
| itype | Type of file to read:  1 dependent variable;  2 cell-by-cell flow term. |
| iprec | Precision used for real numbers:  1 single precision;  2 double precision. |
| narray | Number of arrays/tables recorded in the binary MODFLOW output file. Each array or table pertains to a flow or boundary condition type at a particular time. Each array or table is preceded by a header. |
| ntime | The number of different times for which dependent variables or flow terms are recorded. |

## Notes

As stated above, a termination character (“\0”, i.e. ACHAR(0)) must be provided with string input arguments. For function *inquire\_modflow\_binary\_file\_specs()* these are filenames. The character arrays through which input and output filenames are provided should contain no greater than 256 elements. However they can have fewer elements than this if desired.

# install\_mf6\_grid\_from\_file

## Description

Function *install\_mf6\_grid\_from\_file()* reads a binary grid file (i.e. a GRB file) written by MODFLOW 6. This file contains all specifications of the grid of a MODFLOW 6 model, including its geographical reference point and rotation. *install\_mf6\_grid\_from\_file()* stores specifications for this grid for the future use of other *model\_interface* functions such as *calc\_mf6\_interp\_factors()*.

Multiple instances of MODFLOW 6 grid specifications can be installed; as presently programmed, an upper limit of five is imposed.

## Function Call

|  |
| --- |
| integer function install\_mf6\_grid\_from\_file( &  gridname, grbfile, &  idis, ncells, ndim1, ndim2, ndim3)  character (len=1), intent(in) :: gridname(LENGRIDNAME)  character (len=1), intent(in) :: grbfile(LENFILENAME)  integer, intent(out) :: idis  integer, intent(out) :: ncells  integer, intent(out) :: ndim1,ndim2,ndim3 |

## Return Value

The function returns a value of zero unless an error condition is encountered, in which case it returns a value of 1. In the latter case, an error message can be retrieved using function *retrieve\_error\_message()*.

## Function Arguments

|  |  |
| --- | --- |
| **Argument** | **Role** |
| gridname | The user-supplied name for an installed set of MODFLOW 6 grid specification. As presently coded, this name must be 200 characters or less in length. |
| grbfile | The binary grid file, recorded by MODFLOW 6, from which grid specifications are read. |
| idis | This argument reports whether the MODFLOW 6 grid is of the DIS type (returned value of 1) or DISV type (returned value of 2). As presently coded, *install\_mf6\_grid\_from file()* will object with an error message if the MODFLOW 6 grid is of the DISU type. |
| ncells | The number of cells in the MODFLOW 6 grid. |
| ndim1 | Returned as the number of model columns (*ncol*) for a DIS grid or the number of model cells per layer (*ncpl*) for a DISV grid. |
| ndim1 | Returned as the number of model rows (*nrow*) for a DIS grid, or as 1 for a DISV grid. |
| ndim3 | Returned as the number of model layers (*nlay*) for both DIS and DISV grids. |

## Notes

Every instance of a MODFLOW 6 grid specification must be provided with a unique name. As presently programmed, this name must be 200 characters or less in length. It is case-insensitive.

As presently programmed, functions of the *model\_interface* module cannot accommodate a DISU grid. If the binary grid file which *install\_mf6\_grid\_from\_file()* is asked to read is written by a DISU-gridded model, an error condition arises. An error message can then be retrieved using the *retrieve\_error\_message()* function.

The geographical coordinates of a MODFLOW 6 grid are read from the binary grid file. These are provided by the modeller as the XORIGIN, YORIGIN and ANGROT variables in the model’s discretization input file. It is the user’s responsibility to ensure that the values of these variables are correct, and that they are therefore compatible with the coordinates of points for which interpolation factors will be calculated by other functions of the *model\_interface* module.

For large models, storage of MODFLOW 6 grid specifications requires a lot of memory. It is recommended that a set of MODFLOW 6 specifications be uninstalled using function *uninstall\_mf6\_grid()* when no longer required.

# install\_structured\_grid

## Description

Function *install\_structured\_grid()* installs the specifications of a structured grid into the persistent memory of the *model\_interface* module. These specifications are comprised of the following:

* the number of rows, columns and layers in the model grid;
* real world coordinates of either its top left or bottom left corner;
* the anticlockwise rotation of the grid row direction from east;
* the model grid DELR and DELC arrays.

Once grid specifications are installed, they can be used by functions such as *interp\_from\_structured\_grid()*. The latter function can read binary dependent-variable files produced by structured MODFLOW, MT3D, MODFLOW-USG and MODFLOW 6 models.

Multiple instances of structured grid specifications can be installed; as presently programmed, an upper limit of five is imposed.

## Function Call

|  |
| --- |
| integer function install\_structured\_grid &  (gridname, &  ncol,nrow,nlay, &  icorner,e0,n0,rotation, &  delr,delc)  character (len=1), intent(in) :: gridname(LENGRIDNAME)  integer, intent(in) :: ncol,nrow,nlay  integer, intent(in) :: icorner  double precision, intent(in) :: e0,n0,rotation  double precision, intent(in) :: delr(ncol),delc(nrow) |

## Return Value

The function returns a value of zero unless an error condition is encountered, in which case it returns a value of 1. In the latter case, an error message can then be retrieved using function *retrieve\_error\_message()*.

## Function Arguments

|  |  |
| --- | --- |
| **Argument** | **Role** |
| gridname | The user-supplied name of an installed structured grid specification. As presently coded, this name must be 200 characters or less in length. |
| ncol | The number of columns in the structured model grid. |
| nrow | The number of rows in the structured model grid. |
| nlay | The number of layers in the structured model grid. |
| icorner | Supply as 1 or 2. If supplied as 1, *e0* and *n0* pertain to the top left corner of the structured model grid. If supplied as 2, *e0* and *n0* pertain to the bottom left corner of the structured model grid. |
| e0 | Real-world eastern coordinate of model grid corner. |
| n0 | Real-world northern coordinate of model grid corner. |
| rotation | Counter-clockwise rotation of row direction of model grid from east. |
| delr | Row widths of model cells |
| delc | Column widths of model cells. |

## Notes

Every instance of an installed structured grid specification must be provided with a unique name. As presently programmed, this name must be 200 characters or less in length. It is case-insensitive.

Specifications of a structured model grid are also provided in a “grid specification file” that is supported by the PEST Groundwater Utilities. In this file, *e0* and *n0* pertain to the top left corner of the grid (i.e. *icorner* option 1). Also, a grid specification file does not record the number of layers in a model grid.

# interp\_from\_mf6\_depvar\_file

## Description

Function *interp\_from\_mf6\_depvar\_file()* reads a dependent-variable file written by MODFLOW 6. For every output time that is recorded in this file, it undertakes spatial interpolation of model-calculated dependent variables to a set of points. Spatial interpolation is implemented using factors that were previously calculated by function *calc\_mf6\_interp\_factors()*.

Arrays used by *interp\_from\_mf6\_depvar\_file()* to store spatial interpolation outcomes are useable by function *interp\_to\_obstime()*. Hence temporal interpolation to observation times can follow spatial interpolation to observation sites.

## Function Call

|  |
| --- |
| integer function interp\_from\_mf6\_depvar\_file( &  depvarfile,factorfile,factorfiletype, &  ntime,vartype,interpthresh,reapportion,nointerpval, &  npts,nproctime,simtime,simstate)  character (len=1), intent(in) :: depvarfile(LENFILENAME)  character (len=1), intent(in) :: factorfile(LENFILENAME)  integer, intent(in) :: factorfiletype  integer, intent(in) :: ntime  character (len=1), intent(in) :: vartype(17)  double precision, intent(in) :: interpthresh  integer, intent(in) :: reapportion  double precision, intent(in) :: nointerpval  integer, intent(in) :: npts  integer, intent(out) :: nproctime  double precision, intent(out) :: simtime(ntime)  double precision, intent(out) :: simstate(ntime,npts) |

## Return Value

Function *interp\_from\_mf6\_depvar\_file()* returns a value of zero unless an error condition is encountered. In that case it returns a value of 1. An error message can then be retrieved using function *retrieve\_error\_message()*.

## Function Arguments

|  |  |
| --- | --- |
| **Argument** | **Role** |
| depvarfile | The name of a binary dependent-variable file written by MODFLOW 6. |
| factorfile | The name of an interpolation factor file written by function *calc\_mf6\_interp\_factors()*. The factors that are recorded in this file must pertain to the same MODFLOW 6 grid as that to which the dependent variable file pertains. |
| factorfiletype | 0 for binary, or 1 for ASCII. |
| ntime | The first dimension of the *simtime* and *simstate* arrays. Generally, this should equal or exceed the number of simulation times for which model outputs are available in *depvarfile*. However if *ntime* is less than this, only dependent variables corresponding to the first *ntime* times that are recorded in the dependent variable file are processed. |
| vartype | Each array in a dependent variable file contains a 16 character text header. *vartype* provides (case-insensitive) text that allows unique identification of a header of interest. Only variables of this type will be processed by function *interp\_from\_mf6\_depvar\_file()*. |
| interpthresh | If the absolute value of a model-calculated dependent variable is equal to, or greater than *interpthresh*, its value is treated as invalid. This may, for example, indicate a dry or inactive model cell. |
| reapportion | If a cell from which spatial interpolation takes place is endowed with a dependent variable value which exceeds *interpthresh* then this can be handled in one of two ways. If *reapportion* is provided as 0, then the point-interpolated value is also declared to be invalid; hence it is assigned a value of *nointerpval*. However if reapportion is supplied as 1, then interpolation to the point may still take place. Other factors which feature in the interpolation process to that point are then adjusted so that they add to 1.0. However if all cells from which interpolation takes place are endowed with dependent variable values which are greater than *interpthresh*, then spatial interpolation to that point becomes impossible; the point is then awarded an interpolated value of *nointerpval*. |
| nointerpval | Function *interp\_from\_mf6\_depvar\_file()* assigns a value of *nointerpval* to any point for which spatial interpolation is impossible. This can occur if the point lies outside the MODFLOW 6 grid. It can also occur if surrounding model cells are dry or inactive. |
| npts | The number of points to which interpolation takes place. This is the second dimension of the *simstate* array. This same value should have been supplied to function *calc\_mf6\_interp\_factors()* when it calculated point interpolation factors. Its value is recorded at the top of the interpolation factor file which the latter function writes. If the value of *npts* supplied to function *interp\_from\_mf6\_depvar\_file()* does not agree with this value, an appropriate error message is generated. |
| nproctime | Function *interp\_from\_mf6\_depvar\_file()* informs a calling program of the number of separate simulation times for which it undertakes spatial interpolation through this variable. Note that the value assigned to *nproctime* will not exceed the user-supplied value of *ntime,* as *interp\_from\_mf6\_depvar\_file()* does not process simulator-generated data pertaining to more than *ntime* simulation times. |
| simtime | Simulation times for which values of dependent variables have been calculated are recorded in this array. Time values beyond *nproctime* are returned as *nointerpval*. |
| simstate | This two-dimensional array contains spatially-interpolated values of dependent variables. The ordering of points is the same as in the *ecoord* and *ncoord* arrays that were provided to function *calc\_mf6\_interp\_factors()* when it calculated interpolation factors. The ordering of times is the same as that encountered in the dependent variable output file that was written by MODFLOW 6. |

## Notes

### Complementary Functions

Use of function *interp\_from\_mf6\_depvar\_file()* complements that of function *calc\_mf6\_interp\_factors()*. Spatial interpolation factors that were calculated by the latter function can be used to interpolate many sets of dependent variables that reside in one or a number of binary, MODFLOW 6 dependent variable files using function *interp\_from\_mf6\_depvar\_file()*. However this implies that spatial interpolation of all of these dependent variables takes place to the same set of points.

Use of function *interp\_from\_mf6\_depvar\_file()* also complements that of function *interp\_to\_obstime()*. The latter function undertakes temporal interpolation, presumably to the times at which observations of system state were made. This function requires use of *nproctime*, as well as the *simtime* and *simstate* arrays that are filled by function *interp\_from\_mf6\_depvar\_file()*.

### Ordering of Points

The eastings and northings of points to which spatial interpolation must take place were supplied to function *calc\_mf6\_interp\_factors()* through the *ecoord* and *ncoord* arrays that are required by that function. These arrays host coordinates for *npts* points, where *npts* is another *calc\_mf6\_interp\_factors()* input variable.

Function *interp\_from\_mf6\_depvar\_file()* also requires a value for *npts*. This must be the same as that which was previously supplied to *calc\_mf6\_interp\_factors()*. The ordering of points in the *simstate* array (these pertain to the second dimension of that array) is the same as that in the *ecoord* and *ncoord* arrays that were previously supplied to *calc\_mf6\_interp\_factors()*. Meanwhile, the ordering of times in the *simtime* and *simstate* arrays is the same as that for which system states are recorded in the MODFLOW 6 dependent variable file.

### Inability to Interpolate

As is described in documentation to function *calc\_mf6\_interp\_factors()*, spatial interpolation to user-specified points is barycentric for a DISV grid, and bilinear for a DIS grid. However, as is also explained in its documentation, function *calc\_mf6\_interp\_factors()* does not actually undertake spatial interpolation; it simply calculates the factors which enable spatial interpolation to take place.

Spatial interpolation of model-calculated dependent variables to a particular point in space requires that the values of dependent variables that are calculated for model cells be multiplied by these factors and summed. For any interpolation point, these factors sum to 1.0. Hence the spatially interpolated value of a dependent variable is the weighted average of dependent variable values ascribed to the centres of model cells that surround it.

If the dependent variable value ascribed to a particular model cell equals or exceeds *interpthresh,* then this value cannot be used in the spatial interpolation process. Function *interp\_from\_mf6\_depvar\_file()* provides two options for handling this situation. The appropriate option can be selected through its *reapportion* input argument. A *reapportion* value of 0 instructs *interp\_from\_mf6\_depvar\_file()* to endow any point for which spatial interpolation requires use of an above-*interpthresh* dependent variable value with an interpolated value of *no\_interpval*. Alternatively, if *reapportion* is supplied as 1, then the above-*interpthresh* value is omitted from the interpolation process; other spatial interpolation factors pertaining to the point of interest are then adjusted so that they sum to 1.0.

### Dependent Variable Types

On most occasions of MODFLOW 6 usage, a single dependent variable type is stored in each dependent variable file. The type of variable is recognized by a 16 character descriptor that is provided in the header to each array that is stored in the file.

There may be circumstances, however, where more than one dependent variable type is stored in a single dependent variable file. Function *interp\_from\_mf6\_depvar\_file()* allows processing of only one dependent variable at a time. This variable must be denoted by the user using the *vartype* function argument. Supply a set of characters that allows unique identification of the variable type of interest from its header. For example “he” or even “h” can be used to identify heads arrays, if heads are the only type of dependent variable that is stored in a dependent variable file. Array identifiers for different dependent variable types are listed in the MODFLOW 6 manual. Those that are featured in a particular binary dependent variable file can be ascertained using function *inquire\_modflow\_binary\_file\_specs()*.

### Array Precision

Function *interp\_from\_mf6\_depvar\_file()* assumes that all real numbers are stored in double precision in a dependent variable file. At the time of writing, this is a MODFLOW 6 specification.

# interp\_from\_structured\_grid

## Description

Function *interp\_from\_structured\_grid()* undertakes layer-specific (i.e. two-dimensional) spatial interpolation from dependent-variable arrays that are recorded in binary simulator output files. Interpolation takes place to user-specified points. The simulator can be MODFLOW, MT3D, MODFLOW-USG or MODFLOW 6. An important requirement for the latter two simulators is that the model employs a structured grid.

## Function Call

|  |
| --- |
| integer function interp\_from\_structured\_grid( &  gridname,depvarfile,isim,iprec,ntime, &  vartype,interpthresh,nointerpval, &  npts,ecoord,ncoord,layer, &  nproctime,simtime,simstate)  character (len=1), intent(in) :: gridname(LENGRIDNAME)  character (len=1), intent(in) :: depvarfile(LENFILENAME)  integer, intent(in) :: isim  integer, intent(in) :: iprec  integer, intent(in) :: ntime  character (len=1), intent(in) :: vartype(17)  double precision, intent(in) :: interpthresh  double precision, intent(in) :: nointerpval  integer, intent(in) :: npts  double precision, intent(in) :: ecoord(npts),ncoord(npts)  integer, intent(in) :: layer(npts)  integer, intent(out) :: nproctime  double precision, intent(out) :: simtime(ntime)  double precision, intent(out) :: simstate(ntime,npts) |

## Return Value

Function *interp\_from\_structured\_grid()* returns a value of zero unless an error condition is encountered, in which case it returns a value of 1. In the latter case, an error message can then be retrieved using function *retrieve\_error\_message()*.

## Function Arguments

|  |  |
| --- | --- |
| **Argument** | **Role** |
| gridname | The name of an installed structured grid. |
| depvarfile | The name of a binary dependent-variable file. This file may have been written by MODFLOW, MT3D, MODFLOW-USG or MODFLOW 6. It must pertain to a structured grid. |
| isim | This argument must be supplied as 1 or -1. Supply a value of 1 for a file produced by any version of MODFLOW. Supply a value of -1 for a file written by MT3D. |
| iprec | Provide a value of 1 or 2. 1 indicates single precision while 2 indicated double precision. |
| ntime | This is used for dimensioning arrays that *interp\_from\_structured\_grid()* will fill with interpolated values of model-calculated dependent variables. If *ntime* is greater than the number of times for which model outputs are recorded in *depvarfile*, then elements pertaining to non-represented simulation times are assigned a value of *nointerpval*. If *ntime* is less than the number of model simulation times that are featured in *depvarfile*, spatially-interpolated model outputs pertaining only to the first *ntime* simulation times are represented in *interp\_from\_structured\_grid()* output arrays. |
| vartype | It is possible that a dependent-variable output file contains calculated values for more than a single dependent variable. Dependent variable types are identified by array headers. Provide the dependent variable type for which interpolation to user-supplied points is required using the *vartype* argument. |
| interpthresh | If the absolute value of a dependent variable assigned to a model cell is above this threshold, then interpolation does not take place from that cell. The cell is assumed to be inactive or dry. |
| nointerpval | This value is provided to elements of *interp\_from\_structured\_grid()* output arrays where spatial interpolation is impossible. |
| npts | The number of points to which spatial interpolation is required. |
| ecoord | Point east coordinates. |
| ncoord | Point north coordinates. |
| layer | The model layers to which points belong. |
| nproctime | Function *interp\_from\_structured\_grid()* informs a calling program of the number of separate simulation times for which it undertakes spatial interpolation through this variable. Note that the value assigned to *nproctime* will not exceed the user-supplied value of *ntime* as *interp\_from\_structured\_grid()* does not process simulator-generated data pertaining to more than *ntime* simulation times. |
| simtime | Simulation times are recorded in this array. Time values beyond *nproctime* are returned as *nointerpval*. |
| simstate | This two-dimensional array contains spatially-interpolated values of dependent variables. The ordering of points is the same as in the user-supplied *ecoord* and *ncoord* arrays. The ordering of times is the same as that encountered in the dependent variable output file. |

## Notes

### Interpolation Scheme

Function *interp\_from\_structured\_grid()* performs within-layer, bilinear interpolation from the centres of model cells to points of user interest. This interpolation scheme accommodates dry and inactive cells. By setting pertinent simulator input variables (for example HNOFLO and HDRY) to appropriate values before running a model, a user must ensure that such cells are easily recognizable.

A number with a high absolute value must be used to denote inactive and dry cells. This absolute value value must be higher than that of *interpthresh*. That is, a “calculated” system state whose absolute value is greater than *interpthresh* is assumed to be dry or inactive.

Interpolation will not take place to a user-supplied point under the following circumstances:

* The point lies outside the model grid;
* The point is surrounded by the centres of four dry or inactive model cells.

Interpolation can take place to a point which lies inside the model grid, but is beyond the first/last row/column of (active) model cells. Obviously, interpolation to such a point can only be approximate. To prevent interpolation to such points, ensure they are not included in the set of points that is supplied to function *interp\_from\_structured\_grid()* through its *ecoord* and *ncoord* arguments.

Where interpolation cannot take place to a point, its interpolated value is set to the user-supplied value of *nointerpval.*

### Knowledge of File Contents

Some of the argument values required by function *interp\_from\_structured\_grid()* require knowledge of the contents of the binary file which is read by this function. In particular, a user must know:

* array text headers;
* the precision with which real numbers are recorded;
* the number of times for which simulator outputs are recorded.

For all but MT3D output files, this information can be obtained using function *inquire\_modflow\_binary\_file\_specs()*.

### Model Output Times

Ideally the value of the *ntime* argument should be the same as the number of simulation times for which calculated system states are recorded in the binary dependent-variable file *depvarfile*. If *ntime* is greater than this, then elements of the *simstate* array that pertain to unpresented model output times are filled with dummy spatially-interpolated system state values of *nointerpval*. If *ntime* is provided with a value that is less than the number of times for which simulation outputs are provided in file *depvarfile*, then spatial interpolation takes place for only the first *ntime* simulation times.

### Dependent Variable Types

A MODFLOW-compatible dependent-variable file often provides values for only a single type of dependent variable (i.e. for a single system state). This state type is listed as a 16 character text string in the header to each array that is recorded in *depvarfile*. Examples are “HEAD”, “DRAWDOWN” and “SUBSIDENCE”. On most occasions of model usage, different types of model-computed state arrays are recorded in different binary output files.

However, there may be some occasions on which this protocol is not followed. A common example is where concentrations of different chemical species are recorded in a single MODFLOW-USG binary output file.

A user specifies what type of system state data are to be processed using the *vartype* argument of function *interp\_from\_structured\_grid()*. Enough characters must be provided in this variable to uniquely identify the requested dependent-variable type. For example “d” can identify HEAD arrays in a file that contains only head arrays. However it cannot uniquely identify HEAD arrays in a binary output file that contains both HEAD and DRAWDOWN arrays. Where a dependent variable type is not uniquely identifiable, interpolated values pertaining to one dependent variable type will be overwritten by those pertaining to another dependent variable type whenever both of them are recorded at the same simulation type. Obviously, this is an undesirable state of affairs. It can be avoided by supplying the full name of a dependent variable through the *vartype* argument. Function *inquire\_modflow\_binary\_file\_specs()* can be used to ascertain these full names from array text headers.

### Grid Real World Coordinates

The real world location of a structured model grid is recorded in its specifications. These are stored using function *install\_structured\_grid()*.

A binary grid file which is written by MODFLOW 6 also contains information that links a model grid to real world coordinates. However function *interp\_from\_structured\_grid()* does not read this file. Hence if the real-world location of a MODFLOW 6 grid as recorded in its binary grid file differs from that which is installed in its structured grid specifications, then *interp\_from\_structured\_grid()* has no way of knowing about this conflict.

Similarly, the real-world coordinates of a structured-grid MODFLOW-USG model may be recorded in its (rather extensive) grid specification file. *Interp\_from\_structured\_grid()* makes no reference to this file. It relies purely on geographical information that is installed with pertinent structured grid specifications using function *install\_structured\_grid()*.

# interp\_to\_obstime()

## Description

Function *interp\_to\_obstime()* undertakes time-interpolation of quantities that were previously read from simulator output files, and possibly spatially interpolated to a set of user-supplied points. The times to which time-interpolation takes place are normally the times at which field observations were made.

Arrays that store the outcomes of spatial interpolation or boundary flow extraction are useable by function *interp\_to\_obstime()*.

## Function Call

|  |
| --- |
| integer function interp\_to\_obstime( &  nsimtime,nproctime,npts, &  simtime,simval, &  interpthresh,how\_extrap,time\_extrap,nointerpval, &  nobs,obspoint,obstime,obssimval)  integer, intent(in) :: nsimtime  integer, intent(in) :: nproctime  integer, intent(in) :: npts  double precision, intent(in) :: simtime(nsimtime)  double precision, intent(in) :: simval(nsimtime,npts)  double precision, intent(in) :: interpthresh  character (len=1) :: how\_extrap  double precision, intent(in) :: time\_extrap  double precision, intent(in) :: nointerpval  integer, intent(in) :: nobs  integer, intent(in) :: obspoint(nobs)  double precision, intent(in) :: obstime(nobs)  double precision, intent(out) :: obssimval(nobs) |

## Return Value

Function *interp\_to\_obstime()* returns a value of 0. However if an error condition is encountered, it returns a value of 1. An error message can then be retrieved using the *retrieve\_error\_message()* function.

## Function Arguments

|  |  |
| --- | --- |
| **Argument** | **Role** |
| nsimtime | First dimension of arrays *simtime* and *simval*. |
| nproctime | Number of times featured in arrays *simtime* and *simval*. *nproctime* must not exceed *nsimtime*. |
| npts | Second dimension of array *simval*. This is the number of points or zones for which temporal interpolation must take place. |
| simtime | This array contains simulation times for which model outputs are available. The *simtime* array will normally have been filled by functions such as *interp\_from\_structured\_grid()*, *interp\_from\_mf6\_depvar\_file()* or *extract\_flows\_from\_cbc\_file()*. |
| simval | Like the *simtime* array, this array will normally have been filled by functions such as *interp\_from\_structured\_grid()*, *interp\_from\_mf6\_depvar\_file()* and *extract\_flows\_from\_cbc\_file()*. In the first two cases, it contains spatially-interpolated, model-simulated values at times for which these have been read from a binary model dependent-variable output file In the second case it contains zone-accumulated flows extracted from a model-generated cell-by-cell flow term file. In both cases, these values are available at *nproctime* times. Functions *interp\_from\_structrured\_grid()*, *interp\_from\_mf6\_depvar\_file()* and *extract\_flows\_from\_cbc\_file()* assign a value to *nproctime*. |
| *interpthresh* | If the absolute value of a quantity in the *simval* array exceeds *interpthresh*, then its value is treated with suspicion by *interp\_to\_obstime()*. See following notes. |
| *how\_extrap* | Supply this as “L” or “C”. “L” signifies linear while “C” signifies constant. |
| *time\_extrap* | The time over which time-extrapolation (in contrast to interpolation) is allowed. This must be supplied as 0.0 or greater. |
| *noninterpval* | If time-interpolation cannot take place to a certain point or zone at a certain time, then a dummy time-interpolated value of *noninterpval* is provided. |
| *nobs* | The number of times (normally observation times) to which time-interpolation must be undertaken. |
| *obspoint* | The indices of points or zones to which time-interpolation is undertaken. These pertain to the *simval* *npts* array dimension, starting at 0. If a point or zone is not referenced in the *simval* array, provide an index of -1. No temporal interpolation is therefore possible to this point or zone. |
| *obstime* | Times to which time-interpolation must take place, presumably times at which observations were made. The time units and reference time must be the same as for elements of the *simtime* array; presumably these are model simulation times. |
| *obssimval* | Time interpolated values. Times correspond to times provided in *obstime*. Points or zones correspond to indices provided in *obspoint*. |

## Notes

### General Usage

As stated above, normally function *interp\_to\_obstime()* is called following a call to a function such as *interp\_from\_structured\_grid()*, *interp\_from\_mf6\_depvar\_file()* and *extract\_flows\_from\_cbc\_file()*. Function *interp\_from\_structured\_grid()* and *interp\_from\_mf6\_depvar\_file()* undertake spatial interpolation of system states that are extracted from a binary MODFLOW-generated dependent variable file. Function *extract\_flows\_from\_cbc\_file()* does not undertake spatial interpolation; however it accumulates flow rates over use-specified zones.

Where spatial interpolation has preceded a call to *interp\_to\_obstime()*, the geographical coordinates of points to which spatial interpolation took place were supplied to respective spatial interpolation functions. *Interp\_to\_obstime()* undertakes temporal interpolation for these same points. These points are referenced by their *simval* array *npts* index. These indices are presumed to start at 0. The ordering of points is the same as that supplied to the spatial interpolation function.

Following a call to *extract\_flows\_from\_cbc\_file()* the *simval* array may contain zone-accumulated flows at model output times. Zone numbers for which flow was accumulated are referenced by index in the second dimension of the *simval* array. Hence in this case, *npts* actually refers to the number of zones for which flow was accumulated rather than to the number of points to which system states had previously been spatially interpolated. Zone indices start at zero. The zone with an index of zero is that with the lowest zone number, that with an index of 1 is that with the next highest zone number etc. Zone numbers are extracted from a model-specific inter zonation array by function *extract\_flows\_from\_cbc\_files()*. They are listed (after sorting) in the *zonenumber()* array that is produced by this function. The zone number index refers to this array.

Times from which interpolation takes place are recorded in the *simtime* array. These are generally model simulation times. *interp\_to\_obstime()* checks that these times are provided in increasing order. Efficiency of its time-interpolation algorithm depends on this.

In contrast, times that are provided in the *obstime* array do not need to be in increasing order. In fact, this is unlikely, as the times at which field measurements of system state or flux were made are normally different for different points or zones.

As stated above, temporal interpolation of model-originating values that are recorded in the *simval* array to observation times that are recorded in the *obstime* array requires that points to which spatial interpolation has already taken place or zones over which flow accumulation has already taken place be identified. These are identified by their *simval* *npts* index, starting at 0. If a user-supplied point or zone does not correspond to a *simval*-indexed point or zone, then the *obspoint* index for this point should be supplied as -1. “Interpolated” values for this point or zone are then assigned values of *nointerpval* at all observation times that are associated with it.

### Interpolation Scheme

Time-interpolation is linear if an observation time lies between two simulation times.

### Extrapolation

If an observation time lies before the first simulation time or after the last simulation time, then temporal extrapolation (rather than interpolation) is required. Three extrapolation options are provided.

The first option is to assign a value at the extrapolated time that is equal to that of the first or last simulation time. The second is to undertake linear extrapolation from the first two or last two simulation times. The third is to forbid extrapolation. The last option is implemented by setting *time\_extrap* to 0.0. *time\_extrap* is the time over which extrapolation is allowed. If an observation time precedes the first simulation time by more than this amount, or postdates the last simulation time by more than this amount, then its “extrapolated” value is *nointerpval*.

### Dummy Simulated Values

In functions such as *interp\_from\_structured\_grid()* and *interp\_from\_mf6\_depvar\_file()*,a dummy model-generated value is assigned at points and times at which spatial interpolation is impossible. This may occur if, for example, a point is outside the model grid, or because a point is temporarily or permanently surrounded by dry or inactive cells. These dummy values must be recognizable as being larger in absolute value than *interpthresh*.

If temporal interpolation of model-generated values to observation times requires use of a *simval* element whose absolute value is greater than *interpthresh*, then the time-interpolated value ascribed to the pertinent *obssimval* element is assigned a value of *nointerpval* as interpolation/extrapolation is considered to be unreliable under these circumstances.

# retrieve\_error\_message

## Description

Function *retrieve\_error\_message()* returns an error message. This is a non-blank string if the previous *model\_inferface* module function call returned an error condition.

## Function Call

|  |
| --- |
| integer function retrieve\_error\_message(errorstring)  character (len=1), intent(out) :: errorstring(LEMMESSAGE) |

## Return Value

Function *retrieve\_error\_message()* returns a value of 0. However if the error message string is not empty, it returns a value of 1.

## Function Arguments

|  |  |
| --- | --- |
| **Argument** | **Role** |
| errorstring | A character array of length 1500 containing an error message. This message is terminated by the string termination character “\0” (i.e. ACHAR(0)). |

## Notes

If a *model\_interface* module function experiences an error condition, then an error message string is available until the next function call. During this subsequent call, it is assigned a blank value, or is overwritten by another error message.

The user-supplied character array to which this error message is written should be at least 1500 characters in length. If this is not the case, there is a risk of inadvertent memory over-write.

# uninstall\_mf6\_grid

## Description

This function acts as a complement to *install\_mf6\_grid\_from\_file()*. As the name implies, it uninstalls previously-installed specifications of a MODFLOW 6 grid. In doing this, it deallocates the considerable amount of memory that may be required to store the complete specifications of this grid.

## Function Call

|  |
| --- |
| integer function uninstall\_mf6\_grid (gridname)  character (len=1), intent(in) :: gridname(LENGRIDNAME) |

## Return Value

Function *uninstall\_mf6\_grid()* returns a value of zero unless an error condition arises. This will occur if the MODFLOW 6 grid nominated through the *gridname* argument is not actually installed, or if some problem is encountered in deallocating memory.

## Function Arguments

|  |  |
| --- | --- |
| **Argument** | **Role** |
| gridname | The name of a MODFLOW 6 grid which requires removal. As presently programmed, this name can be a maximum of 200 characters in length. |

# uninstall\_structured\_grid

## Description

This function acts as a complement to *install\_structured\_grid()*. As the name implies, it uninstalls previously-installed specifications of a structured grid. In doing this, it deallocates memory that is used to house the DELR and DELC arrays pertaining to this grid.

## Function Call

|  |
| --- |
| integer function uninstall\_structured\_grid(gridname)  character (len=1), intent(in) :: gridname(LENGRIDNAME) |

## Return Value

Function *uninstall\_structured\_grid()* returns a value of zero unless an error condition arises. This will occur if the structured grid nominated through the *gridname* argument is not actually installed, or if some problem is encountered in deallocating memory.

## Function Arguments

|  |  |
| --- | --- |
| **Argument** | **Role** |
| gridname | The name of an installed structured grid. As presently programmed, this name can be a maximum of 200 characters in length. |

# Driver Programs

## Introduction

A number of “driver programs” have been written to test the integrity of *model\_interface* module functions. The functionality of these driver programs is primitive; they are written in an ad-hoc manner, and accomplish very little. Nevertheless, their workings are explained herein in case they prove useful to someone other than the author.

Note the following.

1. Because these driver programs are written in FORTRAN, they do not test C or Python interoperability of *model\_interface* functions. However they allow testing of *model\_interface* function algorithms, as well as their responses to error conditions. With small modifications to these driver programs, the range and nature of this testing can be easily extended.
2. It is expected that small modifications will be made to these driver programs on a regular basis. Furthermore, more of them will be written as more *model\_interface* module functions are developed. Hence documentation provided in this chapter is likely to become quickly outdated.

## DRIVER1

DRIVER1 calls the following *model\_interface* module functions:

1. *inquire\_modflow\_binary\_file\_specs()*
2. *retrieve\_error\_message()*

Typical prompts and responses are as follows:

A simulation code is required.

(1=mf; 21=mfusg\_s; 22=mfusg\_us; 31=mf6\_dis; 32=mfusg\_disv; 33=mfusg\_disu)

Enter simulation code: ***22***

Enter name of MODFLOW binary output file (<Enter> if no more): ***GWHv4TR011.hds***

Enter file type code (1=state; 2=flow): ***1***

Enter name for file details file (<Enter> if none): ***temp1.dat***

Calling function inquire\_modflow\_binary\_file\_specs...

IFAIL = 0

FILE TYPE = dependent-variable

PRECISION = single

Number of arrays = 1820

Number of output times = 91

Enter name of MODFLOW binary output file (<Enter> if no more): ***GWHv4TR011.cbb***

Enter file type code (1=state; 2=flow): ***2***

Enter name for file details file (<Enter> if none): ***temp2.dat***

Calling function inquire\_modflow\_binary\_file\_specs...

IFAIL = 0

FILE TYPE = cell-by-cell flow term

PRECISION = single

Number of arrays = 727

Number of output times = 91

Enter name of MODFLOW binary output file (<Enter> if no more): ***<Enter>***

If an error condition is encountered, DRIVER1 reports the error message to the screen. It then prompts for another MODFLOW binary output file to read. If no error condition is encountered, it records some details of the file that it has just read to the screen. See above.

## DRIVER2

DRIVER2 calls the following *model\_interface* module functions:

* *install\_structured\_grid()*
* *uninstall\_structured\_grid()*
* *retrieve\_error\_message()*
* *free\_all\_memory()*
* *interp\_from\_structured\_grid()*

DRIVER2 commences execution by testing installation of structured grid specifications. It asks:

Enter name of a GW Utils grid spec file (<Enter> if no more):

Provide the name of a grid specification file in response to this prompt. See Part A of the manual for the PEST groundwater utilities for specifications of this file type.

Once DRIVER2 has read the file it asks:

How many layers in this grid?

Supply a name for this grid:

After having obtained this information, DRIVER2 installs the grid specifications. Any error messages are reported to the screen.

A response of <Enter> to the first of the above prompts causes DRIVER2 to move into the second phase of its operations. During this phase it uninstalls user-specified structured grid specifications. It asks:

Enter name of grid to uninstall (<Enter> if no more):

Provide the name of previously installed grid specifications. DRIVER2 then calls function *uninstall\_structured\_grid()* in order to uninstall the pertinent structured grid specifications. It then informs the user of its success or failure in doing this. An error message accompanies failure.

If the response to the above prompt is <Enter>, DRIVER2 provides the opportunity to return to phase 1 of its operations, that is the installation of more grids. It asks:

Install more grids? [y/n]:

A response of “n” takes the user into phase 3 of DRIVER2’s operations. This is where it tests function *interp\_from\_structured\_grid()*. It asks the following series of questions.

Enter name of point coordinates file (<Enter> if no more):

Enter name of MODFLOW binary dep. var. output file:

Enter grid name to which it pertains:

Enter number of output times in this file:

Enter precision (1=single; 2=double):

Enter HDRY/HNOFLO threshold:

Enter header text of interest:

Enter name for output file:

A “points coordinates file” contains four columns of data, with no header. The first column contains point identifiers. These must be 20 spaces or less in length. (Note that this implies no name length restrictions for other programs which call *model\_interface* model functions.) The next two columns contain real-world eastings and northings, while the last column contains point layer numbers.

Once it has enough information to call *interp\_from\_structured\_grid()*, DRIVER2 calls this function. If the function call was unsuccessful, this is reported to the screen, together with an error message. Otherwise, DRIVER2 records the values of spatially-interpolated dependent variables in the user-nominated output file and reports to the screen the number of separate simulation times for which interpolation took place.

The spatial interpolation cycle is then re-commenced. DRIVER2 repeats the first of the above prompts. If the response is <Enter> (implying that no more testing of spatial interpolation functionality is required), DRIVER2 tidies up. It does this by calling function *free\_all\_memory()*. It then ceases execution.

## DRIVER3

DRIVER3 calls the following *model\_interface* module functions:

* *install\_structured\_grid()*
* *retrieve\_error\_message()*
* *free\_all\_memory()*
* *interp\_from\_structured\_grid()*
* *interp\_to\_obstime()*

Upon commencement of execution, DRIVER3 installs the specifications of a structured MODFLOW grid. Its prompts are:

Enter name of a GW Utils grid spec file:

How many layers in this grid?

Supply a name for this grid:

Next DRIVER3 reads point data from a “point data file”. As explained in documentation of DRIVER2, this file contains four columns of data, with no header. The first column must contain point identifiers. These must be 20 characters or less in length. (Note that this implies no name length restrictions for programs which call *model\_interface* model functions.) The next two columns must contain real-world eastings and northings, while the last column must contain model layer numbers with which points are associated. The prompt is:

Enter name of point data file:

DRIVER3’s next task is to read a MODFLOW-generated binary dependent-variable file. So it prompts for the name of this file, as well as for a few details pertaining to this file (see documentation of DRIVER2 for further details). The prompts are:

Enter name of MODFLOW binary dep. var. output file:

Enter grid name to which it pertains:

Enter number of output times in this file:

Enter precision (1=single; 2=double):

Enter HDRY/HNOFLO threshold:

Enter header text of interest:

DRIVER3 next calls function *interp\_from\_structured\_grid()* to undertake spatial interpolation of the contents of this file to the points whose coordinates were provided in the point data file.

DRIVER3 then prompts for the name of a “post-spatial-interpolation” output file. This is the same file that is written by DRIVER2. It contains model-generated values of dependent variables that have undergone spatial interpolation to the locations of user-supplied points. The prompt is:

Enter name for post-spatial interpolation output file:

Next DRIVER3 prompts for the name of an “observation time file”. This file should contain two data columns with no headers. The first column should contain point names (20 characters or less in length). These should correspond to point names in the previously-supplied point data file. However these points do not need to be supplied in the same order. Points can be omitted if desired. Furthermore, points that are not featured in the point data file can be supplied in the observation time file if desired; this tests *interp\_to\_obstime()*’s ability to ignore these points. The second column should associate an arbitrary time with each point. These times can be in arbitrary order.

DRIVER3 next asks for time-extrapolation options:

Enter time-extrapolation option (L/C):

Enter extrapolation time limit:

Finally DRIVER3 asks for the name of another file that it will write:

Enter name for post-time-interpolation output file:

This file contains time-interpolated values to point observation times.

## DRIVER4

DRIVER4 tests the following *model\_interface* module functions:

* *install\_mf6\_grid\_from\_file()*
* *uninstall\_mf6\_grid()*
* *calc\_mf6\_interp\_factors()*
* *interp\_from\_mf6\_depvar\_file()*
* *retrieve\_error\_message()*
* *interp\_to\_obstime()*
* *free\_all\_memory()*

Upon commencement of execution, DRIVER4 provides the user with a suite of choices. Its screen display is as follows:

What do you want to do?

To end this program - enter 0

To install a set of MF6 grid specs - enter 1

To uninstall a set of MF6 grid specs - enter 2

To read a set of point coordinates - enter 3

To write an interpolation factor file - enter 4

To interpolate from an interp factor file - enter 5

To time-interpolate after spatial interpolation - enter 6

Enter your choice:

Some of the tasks that are performed by DRIVER4 are the same as those performed by DRIVER3. However, in contrast to DRIVER3, DRIVER4 uses *model\_interface* module functionality to perform spatial interpolation from MODFLOW 6 grids to a set of points. Spatial interpolation is achieved through a two-step process. First, interpolation factors that are specific to a particular MODFLOW 6 DIS or DISV grid are calculated by function *calc\_mf6\_interp\_factors()*. These factors are then applied to the contents of a binary dependent-variable file that is written by MODFLOW 6 using function *interp\_from\_mf6\_depvar\_file()*. If desired, temporal interpolation to user-specified times that are unique to each observation can also be carried out. (This mimics interpolation to the times at which measurements are made.)

Some of the details of DRIVER4 functionality are now described.

When implementing any of the following options, DRIVER4 reports its activities to the screen. If an error condition is encountered during any call to a *model\_interface* function, DRIVER4 calls function *retrieve\_error\_message()* and then reports the error message to the screen. Hence DRIVER4 can be used to test *model\_interface* function error recognition and reporting at the same time as it tests its functionality.

### Option 1

If the first of the above options is selected, DRIVER4 asks:

Enter name of MF6 GRB file:

Enter name for grid:

Once it has received answers to these questions, DRIVER4 calls function *install\_mf6\_grid\_from\_file()* to install a set of MODFLOW 6 grid specifications. It associates this set of specifications with a user-supplied name.

### Option 2

DRIVER4 prompts for the name of the set of MODFLOW 6 grid specifications that it must uninstall.

Enter name of MF6 grid to uninstall:

Once the name of a previously installed set of grid specifications has been provided, DRIVER4 uninstalls this set of specifications by calling function *uninstall\_mf6\_grid()*.

### Option 3

Invocation of this option causes MODFLOW 6 to read a set of point coordinates. These are points to which spatial interpolation will take place through invocation of other DRIVER4 options.

DRIVER4 asks:

Enter name of points coordinate file:

The format of a points coordinates file is straightforward. The following figure shows the first part of one such file. The file must have four columns, these being space- or comma-delimited. The first column must contain point names (20 characters or less in length). Other columns contain point easting, northing and layer numbers respectively. (Note that functions of the *model\_interface* module do not store point names. Hence the 20 character limit does not apply to them.)

|  |
| --- |
| BH13,711867,7459940,1  BH14a,719593,7461005,1  BH14b,719586,7461003,1  BH15,722145,7461300,1  BH16d,724055,7462377,1  BH16s,724058,7462378,1  BH17d,725698.126,7463790.821,1  BH17s,725698.822,7463790.494,1  BH18d,725308,7463169,1  BH18s,725309,7463167,1  etc |

If option 3 is invoked again, then any previously-read set of points is emptied from DRIVER4’s memory and over-written by a new set of points.

### Option 4

This is where DRIVER4 calls *function calc\_mf6\_interp\_factors()* to calculate spatial interpolation factors. It asks:

Enter name of MF6 grid to interpolate from:

Enter name of factor file to write:

Is this a binary or text file? [0/1]:

Enter name of bln file to write:

DRIVER4 will accept a blank response to the last of the above prompts. In this case it does not request that function *calc\_mf6\_interp\_factors()* writes a SURFER BLN file whose vertices are the centres of model cells that surround interpolation points.

After it has called function *calc\_mf6\_interp\_factors()*, DRIVER 4 reports the “interpolation success rate” to the screen. Interpolation is unsuccessful if a point is not enclosed by a triangle or quadrilateral whose vertices are model cell centres. The latter are the points from which spatial interpolation takes place.

### Option 5

If this option is invoked, DRIVER4 conducts spatial interpolation from a MODFLOW 6 dependent-variable file to the latest set of points that it has read. It asks for the name of the interpolation factor file that it must write. Note that this file may have been written on a previous DRIVER4 run.

DRIVER4’s prompts are:

Enter name of MODFLOW 6 dependent variable file:

How many times are represented in this file?

Enter text array identifier:

Enter inactive threshold:

Reapportion interp factors for dry/inact cells? [n/y = 0/1]:

Enter value indicating no interpolation:

Enter name of interpolation factor file:

Is this a binary or text file? [0/1]:

These prompts are all easily related to *interp\_from\_mf6\_depvar\_file()* arguments.

If spatial interpolation is successful, DRIVER4 prompts for the name of a file in which it can record the outcomes of spatial dependent-variable interpolation. The prompt is:

Enter file to record interpolated values:

This file has three columns. The first lists point identifiers, the second lists model output times, and the third lists spatially interpolated dependent variable values. All of the model output times and interpolated values are listed for the first point, then for the second point, and so on.

### Option 6

If this option is invoked, DRIVER 4 undertakes temporal interpolation of values which have previously been spatially interpolated to the locations of user-supplied points. It calls model interface function function *interp\_to\_obstime()* to perform this task.

Its prompts are:

Enter name of observation time file:

Enter inactive threshold in spatially-interpolated data:

Enter time-extrapolation option (L/C):

Enter extrapolation time limit:

Enter value indicating no time interpolation:

The figure below shows an example of an “observation time file” that is requested through the first of the above prompts.

|  |
| --- |
| BH3 42594.5000  WB15HD1S002 42601.5000  WB15HD1S002 42608.5000  WB15HD1S002 42615.5000  WB15HD1S002 42622.5000  WB15HD1S002 42629.5000  WB15HD1S002 42636.5000  WB15HD1S002 42643.5000  BH5 34521.9  WB18HD1N0002 43315.5000  WB19HD1N0007 44428.5000  WB20H1SW0001 44358.5000  Etc |

An observation time file must possess two columns of data. The first column contains point names while the second column contains times. “Times” are rea numbers that are, in fact, elapsed times since a certain reference time. These elapsed times must have the same reference time, and employ the same units, as the simulator does. Points and times can be supplied in any order. If a point is not among those to which spatial interpolation previously took place, then the interpolated value of the dependent variable to that point will be the user-supplied value for “no time interpolation” (see the last of the above DRIVER4 prompts).

Once time-interpolation has taken place, and if an error condition is not encountered, DRIVER4 asks:

Enter name for post-time-interpolation output file:

DRIVER4 records a file in which points are listed in the order that they were supplied to it through option 3. The times to which temporal interpolation was requested for each point are recorded in the second column of this file. These are recorded for the first point, then for the second point, and so on. The final column contains time-interpolated dependent-variable values to points and times referenced in the first two columns.

## DRIVER5

DRIVER5 tests the following *model\_interface* module functions:

* *inquire\_modflow\_binary\_file\_specs()*
* *extract\_flows\_from\_cbc\_file()*
* *interp\_to\_obstime()*
* *free\_all\_memory()*
* *retrieve\_error\_message()*

DRIVER5 commences execution by prompting for the number of cells in the model grid pertaining to the model whose cell-by-cell flow term file it will shortly read. It asks:

Enter number of cells in model:

It then asks for the type of model that it is dealing with:

A simulator code is required.

(1=mf; 21=mfusg\_s; 22=mfusg\_us; 31=mf6\_dis; 32=mfusg\_disv; 33=mfusg\_disu)

Enter simulation code:

As is apparent from the above prompt, this is the value that DRIVER5 assigns to the *isim* argument of function *extract\_flows\_from\_cbc\_file()*.

DRIVER5 then asks for the name of a file that contains an integer zonation array. This is a simple file that must contain two columns of integers (with no header provided for each column). The first column must contain model node numbers, while the second column must contain the zone number that is associated with respective nodes. The order in which these pairs of values are supplied does not matter. Omitted cells are awarded zone numbers of zero.

DRIVER5 now asks for the *nzone* value that it must provide to function *extract\_flows\_from\_cbc\_file()*. The prompt is:

Enter a number that equals or exceeds number of separate zones:

Next it asks for the MODFLOW-generated cell-by-cell flow term file that it must read:

Enter name of model-generated cbc flow term file to read:

Before calling *extract\_flows\_from\_cbc\_file()* DRIVER5 calls function *inquire\_modflow\_binary\_file\_specs()*. It prompts for the name of the file in which the latter function should record the headers to arrays and tables that if finds in the cell-by-cell flow term file:

Enter name of file to record contents of this file:

DRIVER5 then instructs function *inquire\_modflow\_binary\_file\_specs()* to reads the cell-by-cell flow term file and report its findings to the user-nominated file. It also writes a short report to the screen. For example:

- calling function inquire\_modflow\_binary\_file\_specs()...

- function call successful

FILE TYPE = cell-by-cell flow term

PRECISION = single

Number of arrays in file = 10

Number of output times in file = 1

DRIVER5 now prepares to call function *extract\_flows\_from\_cbc\_file()*. It asks for the text identifier of the flow type that it must read. This is the *flowtype* argument of function *extract\_flows\_from\_cbc\_file()*. The prompt is:

Enter text that denotes flow type of interest:

It then calls function *extract\_flows\_from\_cbc\_file()* to accumulate flow terms of this type. It records its success (or otherwise) in doing this to the screen. If *extract\_flows\_from\_cbc\_file()* was successful, DRIVER5 also lists (in order) the zone numbers that *extract\_flows\_from\_cbc\_file()* encountered in the integer zonation array. For example:

- calling function extract\_flows\_from\_cbc\_file()...

- function call successful

NPROCTIME = 1

NUMZONE = 5

Zone numbers:-

2

4

6

8

10

DRIVER5 also prompts for the name of a file to which it can report zone-accumulated flows for each simulation time that *extract\_flows\_from\_cbc\_file()* encountered in the cell-by-cell flow term file that it read. The prompt is:

Enter name of flow-in-zone file to write:

DRIVER5 now asks whether the user would like these accumulated flows to undergo interpolation to times of his/her choosing. It asks:

Undertake time interpolation? (y/n):

Then, if the answer is “y”:

Enter name of observation time file:

The figure below shows an example of such a file:

|  |
| --- |
| zone1 100.0  zone1 250.0  zone1 429.0  zone2 50.0  zone3 90.0  etc |

The file should have two columns, neither of which should possess a header. The first column should list text identifiers for a number of zones. The first four letters of each such zone identifier should be “zone”; the zone number must follow. (Note that if this protocol is not followed for a particular zone, then DRIVER5 will define an unidentified zone. The same will occur if a zone number supplied in this file is not featured in the previously supplied integer zonation array.)

Simulation times should appear in the second column of the above file. Note the following:

* zones and times can be provided in any order;
* zones that feature in the previously-supplied integer zonation array can be omitted if desired;

DRIVER5 next asks a few questions pertaining to temporal extrapolation:

Enter time-extrapolation option (L/C):

In the above prompt “L” pertains to “linear” and “C” pertains to “constant”.

Enter extrapolation time limit:

and then:

Enter value indicating no time interpolation:

DRIVER5 then calls function *interp\_to\_obstime()* to undertake the necessary time-interpolation. If there are no problems, it prompts for the name of a file to which it can report the outcomes of the time-interpolation process:

Enter name for post-time-interpolation output file:

This file records time-interpolated values of zonal flows at user-nominated, zone-specific, interpolation times.