

PLPROC

A Parameter List Processor

John Doherty

**Watermark Numerical Computing
and
National Centre for Groundwater Research and Training, Australia
May, 2020**

Preface

General

This document constitutes a manual for PLPROC, a “parameter list processor”. PLPROC is meant to serve as a parameter preprocessor for complex numerical models. It allows a user to create and manipulate lists of parameters, as well as individual parameters, and then to write the outcomes of these manipulations to model input files of arbitrary complexity. In doing so, it is intended to provide a modeller with the capabilities necessary for formulation of the inverse problem of model calibration, as well as the problem of calibration-constrained uncertainty analysis, in a way that best serves his/her particular modelling context.

PLPROC reproduces some of the capabilities of the PEST Groundwater Data Utility suite, particularly those associated with spatial interpolation. However it provides greater flexibility in implementing these capabilities. Furthermore, whereas many of the Groundwater Data Utilities were written with a particular model suite in mind (mainly MODFLOW-compatible models), the functionality provided by PLPROC is intended to be more general. In addition to this, PLPROC is designed to be compatible with models which do not necessarily use a rectangular grid, but use an unstructured grid or mesh instead.

Flexibility is inevitably accompanied by increased difficulty of use. Indeed it would be fair to say that use of PLPROC is not as straightforward as use of many of the Groundwater Data Utilities - all of which ask the user a series of questions, mostly specific to a certain type of model. In contrast to this, the PLPROC user communicates with PLPROC in an entirely different way, namely through a “language”. The syntax of this language has much in common with programming languages such as Python and FORTRAN. However the capabilities which are offered are not nearly as broad as those offered by a complete programming language. Most of the processing that is performed by PLPROC is accessed through a series of function calls, with processing details being specified by function arguments. Some of these functions assign values to entire parameter lists, while some assign no values at all (and hence resemble subroutines more than functions). Some operate directly on parameter lists, and can be viewed as functions which belong to “list objects”. The syntax used to invoke these functions supports this concept.

It is anticipated that for many years to come PLPROC will be “work in progress”, and that its capabilities will expand in order provide a wider range of parameter processing options, while supporting a wider range of models. Meanwhile, it is not impossible that a user of PLPROC will encounter a bug. Should this occur, send a report to me, John Doherty, at the following email address:

johndoherty@ozemail.com.au

This Document

This document is divided into two parts. Chapters 1 to 7 describe PLPROC in general terms, providing an overview of its capabilities, of the entities that it employs, and the ways in which these entities can be manipulated. Chapter 1 introduces PLPROC. Chapter 2 describes the concept of “parameter lists”, while Chapter 3 discusses the details of PLPROC’s equation processing functionality. Chapter 4 describes PLPROC function protocols. Chapter 5 covers

ways in which PLPROC can write model input files using function-embedded template files. Chapter 6 discusses some basic script control devices available in PLPROC.

Individual PLPROC functions are documented in the second section of this document, with each function given a chapter of its own. Functions are listed in alphabetical order. No chapter numbers are provided, so that new functions can be added at their correct alphabetical locations without the need to alter a chapter numbering sequence as the size of this manual grows to reflect expanding PLPROC capabilities over time.

Acknowledgements

I would like to express my gratitude to the National Centre for Groundwater Research and Training (NCGRT), Flinders University, Australia for funding the initial development of this software.

Additionally, I would like to thank BHP and Rio Tinto. These companies fund the GMDSI project which has also contributed to this software. GMDSI stand for “Groundwater Model Decision Support Initiative”. This 3.5 year project, conducted under the auspice of NCGRT, commenced in late 2019. Its aim is to create greater awareness of issues associated with decision-support groundwater modelling, and to assist in addressing these issues through dialogue, education, worked examples, research and strategic software development.

Dr. John Doherty
Watermark Numerical Computing
Australia

List of PLPROC functions

The table below contains a list of PLPROC functions. Note that this table does not constitute an exhaustive description of PLPROC capabilities, for PLPROC supports complex mathematical manipulation of lists and scalars through its equation functionality.

Alphabetical listing of PLPROC functions.

Function	Purpose
alluv_boundary_interp	The user provides coordinates of lines depicting both sides of an alluvial system, and one or more alluvial tributaries. Bearings of boundary line segments, and optionally other data, are interpolated from those boundaries to PLIST elements. These bearings can be used with other PLPROC functions to provide a basis for spatial interpolation where anisotropy varies spatially.
assign_by_relation	Assigns values to elements of one PLIST based on the values of elements in another, unrelated, PLIST. Elements of each are matched through equating elements of one or a number of associated SLISTs, PLISTs and/or parent CLIST element coordinates.
assign_by_slist	Assigns values to elements of one PLIST based on the values of elements in another, unrelated, PLIST. Elements of each are matched through equating elements of associated SLISTs. Provides an easy-to-use subset of the functionality provided by <i>assign_by_relation()</i> .
assign_from_closest	Assigns values to elements of one SLIST or PLIST equal to values pertaining to closest elements of another SLIST or PLIST.
assign_if_colocated_2d	Allows transfer of data between PLISTS whose reference CLISTS are different on the basis of planar proximity of list elements.
build_covmat_from_vario	Builds a covariance matrix pertaining to elements of a two- or three-dimensional CLIST based on a stationary or non-stationary variogram.
calc_linear_interp_factors	Undertakes linear interpolation down segments of a SEGLIST from elements of a SEGLIST-linked CLIST to another, arbitrary CLIST.
calc_kriging_factors_2d	Calculates interpolation factors for subsequent 2D kriging from one PLIST to another.
calc_kriging_factors_auto_2d	Calculates interpolation factors for subsequent 2D kriging from one PLIST to another; accommodates spatially varying anisotropy and requires minimal user input.
calc_kriging_factors_3d	Calculates interpolation factors for subsequent 3D kriging from one PLIST to another.

calc_rbf_factors_2d	Calculates information employed for subsequent interpolation from one PLIST to another using radial basis functions.
create_clist_from_seglist	Creates a CLIST which is automatically linked to the SEGLIST from which it is created.
find_cells_in_lists	Reads a model input file containing tables of model cell indices and corresponding cell properties. Identifies cells which are cited in these tables and populates a model-compatible SLIST accordingly.
gen_ran_plists_basic	Creates a suite of PLISTs and fills them with random values generated according to a variety of specifications.
gen_ran_plists_cond	Creates a suite of PLISTs and fills them with random variables conditioned by sampled variables with which they are statistically correlated.
goto	Redirects PLPROC script processing to another part of the script. Re-direction can depend on the value of a scalar variable.
interp_using_file	Assigns values to a PLIST based on interpolation factors recorded by a <i>calc_kriging_factors*()</i> functions, or by function <i>calc_linear_interp_factors()</i> .
inform_from_gplane	Parameterizes elements of a model-based PLIST into which a GPLANE is inserted.
ivd_interpolate_2d	Performs PLIST to PLIST interpolation using the inverse-power-of-distance methodology.
ivd_sda_interpolate_2d	Performs PLIST to PLIST interpolation using the inverse-power-of-distance methodology where anisotropy is spatially variable.
krige_using_file	Implements kriging based on factors computed by pertinent kriging factor generation functions.
link_seglist_to_clist	Facilitates pilot point parameterization of linear segmented features by linking elements of a CLIST to individual segments of a SEGLIST.
mremove	Removes an MLIST, and optionally associated PLISTs or SLISTS, from PLPROC memory.
new_gplane_clist	Creates a gridded GPLANE together with a CLIST based on local GPLANE coordinates. Optionally imports SLISTS and/or PLISTs associated with this GPLANE.
new_gridded_clist	Creates a CLIST based on a regular or semi-regular 2D or 3D grid.
new_mlist	Creates a new MLIST and associated PLISTs or SLISTS.

<code>new_plist</code>	Creates a new PLIST based on specifications of an existing reference CLIST.
<code>new_slist</code>	Creates a new SLIST based on specifications of an existing reference CLIST or of an existing PLIST.
<code>parameter_delimiter</code>	An embedded template file function; designates the parameter delimiter employed in definition of parameter spaces.
<code>partition_by_clist</code>	Creates a new SLIST or PLIST through excision from an existing SLIST or PLIST, with excision specifications being determined by an existing CLIST.
<code>partition_by_eqn</code>	Creates a new CLIST through excision from a larger CLIST with excision specifications being based on a user-specified logical equation.
<code>partition_by_file</code>	Creates a new CLIST through excision from a larger CLIST, with excision specifications being based on CLIST elements identified in a user-supplied file.
<code>rbf_interpolate_2d</code>	Interpolates from one PLIST to another using two-dimensional radial basis functions.
<code>rbf_interpolate_3d</code>	Interpolates from one PLIST to another using three-dimensional radial basis functions.
<code>rbf_sda_interpolate_2d</code>	Performs PLIST to PLIST interpolation using radial basis functions where anisotropy is spatially variable.
<code>rbf_using_file</code>	Interpolates from one PLIST to another using radial basis functions employing information previously computed by function <code>calc_rbf_factors_2d()</code> .
<code>read_column_data_file</code>	Reads tabular data associated with one or a number of SLISTs PLISTs or MLISTs from a file in which data is arranged in columns.
<code>read_list_as_array</code>	Reads data stored in a file (possibly in array format) into a single SLIST or PLIST.
<code>read_list_file</code>	Reads tabular data associated with multiple PLISTS and SLISTS from a file. Optionally reads CLIST specifications as well.
<code>read_matrix_from_file</code>	Reads a MATRIX from a text or binary file.
<code>read_mf_grid_specs</code>	Creates a 2D CLIST based on the contents of a MODFLOW grid specification file.
<code>read_mf_int_array</code>	Populates SLIST elements with the contents of a MODFLOW-compatible integer array read from a text file.
<code>read_mf_real_array</code>	Populates PLIST elements with the contexts of a MODFLOW-compatible real array read from a text file.

read_mf_usg_grid_specs	Creates a 3D CLIST and node layer SLIST through reading a MODFLOW-USG grid specification file.
read_mf6_grid_specs	Reads a binary grid file written by MODFLOW6. This file contains specifications for a MODFLOW6 grid, as well as other information such as the model's IDOMAIN array.
read_multiple_array_file	Reads data required for the population of one or a number of SLISTs and PLISTs from a file that holds this data in either list or array format. Individual lists and arrays stored within this file are preceded by text headers.
read_restart_data	Reads a binary file recorded using the <code>save_restart_data()</code> function.
read_scalar_file	Reads a set of scalar parameters and string variables from a tabular data file.
read_segfile	Reads a SEGLIST from a segmentation file (i.e. a "segfile").
remove	Removes an SLIST, PLIST, CLIST, MATRIX of GPLANE from PLPROC memory.
replace_cells_in_lists	Reads a model input file containing tables of model cell values identified by cell index. Replaces cell properties in these tables by elements of model-compatible PLISTs.
replace_column	An embedded template file function; replaces a column of data on an existing model input file by elements of a PLIST.
report_all_entities	Records specifications of all currently stored PLPROC entities in a user-nominated file.
report_dependent_lists	Records in tabular format in a user-specified file, the values for all elements of all SLISTS and PLISTS that are dependent on a user-specified CLIST.
save_restart_data	Writes a binary file in which all PLPROC variables are recorded. PLPROC processing can be re-commenced by reading this file on a subsequent PLPROC run.
stat	Calculates some statistics based on the contents of a PLIST.
stop	Terminates PLPROC processing at current command line.
upscale_by_averaging	Assigns values to elements of a coarse model PLIST by averaging those pertaining to elements of a fine model PLIST. Associations between coarse and fine model elements are read from a "node association file".
write_column_data_file	Exports data contained in SLISTs, PLISTs and MLISTs in columnar data format, ready for importation into spreadsheets and other software packages.

write_in_sequence	An embedded template file function; writes sequential elements of all or part of a PLIST to a file.
write_matrix_to_file	Records a MATRIX in a text or binary file.
write_model_input_file	Writes PLIST or SCALAR data to a model input file using a template of that file.

Functions grouped by nature of task performed.

Task	Relevant functions
Creation of a CLIST	new_gplane_clist new_gridded_clist read_list_file partition_by_eqn partition_by_file read_mf_grid_specs read_mf_usg_grid_specs read_mf6_grid_specs create_clist_from_seglist
Creation of an SLIST or PLIST	read_list_file new_mlist new_slist new_plist partition_by_clist read_mf_usg_grid_specs read_mf6_grid_specs find_cells_in_lists PLPROC equations
Importing PLIST/SLIST element values	read_column_data_file read_list_file read_mf_int_array read_mf_real_array read_mf_usg_grid_specs read_multiple_array_file read_list_as_array
Importing scalar and string data	read_scalar_file
Exporting PLIST/SLIST element values	write_column_data_file replace_cells_in_lists

Interpolation and PLIST-to-PLIST or SLIST-to-SLIST assignment where reference CLISTs differ	alluv_boundary_interp assign_from_closest assign_by_relation assign_by_slist calc_kriging_factors_2d calc_kriging_factors_auto_2d calc_kriging_factors_3d calc_linear_interp_factors inform_from_gplane interp_using_file ivd_interpolate_2d ivd_sda_interpolate_2d krige_using_file assign_if_colocated_2d calc_rbf_factors_2d rbf_interpolate_2d rbf_interpolate_3d rbf_sda_interpolate_2d rbf_using_file upscale_by_averaging
Processing the contents of a PLIST	stat
Creation and management of MLISTS	new_mlist read_column_data_file gen_ran_plists_basic gen_ran_plists_cond mremove
Creation and management of MATRIXs	build_covmat_from_vario read_matrix_from_file write_matrix_to_file remove
Creation and management of GPLANEs	new_gplane_clist inform_from_gplane remove
Creation and use of SEGLISTS	link_seglist_to_clist create_clist_from_seglist read_segfile find_cells_in_lists replace_cells_in_lists
Random number generation	build_covmat_from_vario gen_ran_plists_basic gen_ran_plists_cond
Reporting	report_all_entities report_dependent_lists
Writing a model input file	write_model_input_file find_cells_in_lists replace_cells_in_lists
Embedded template file functions	parameter_delimiter replace_column write_in_sequence

Script control	goto stop save_restart_data read_restart_data
Tidying up	remove mremove

Table of Contents

1. Introduction.....	1
1.1 General.....	1
1.2 PLPROC Input	3
1.3 Running PLPROC	4
2. Lists, Gplanes, Scalars, Strings and Matrices.....	5
2.1 General.....	5
2.2 CLISTS, SLISTS and PLISTS.....	5
2.3 CLIST Options	6
2.3.1 Dimensionality	6
2.3.2 Element referencing.....	6
2.3.3 Dependent SLISTS and PLISTS.....	7
2.3.4 Defining Lists	8
2.3.5 Gridded CLISTS	8
2.3.6 CLIST to Model Associations	11
2.4 SEGLISTS	11
2.5 GPLANES.....	12
2.5.1 GPLANE concepts.....	12
2.5.2 GPLANES and model parameterization.....	16
2.5.3 Some considerations when using a GPLANE.....	19
2.6 MLISTS	20
2.7 Scalars.....	21
2.8 Strings	21
2.9 Matrices	21
2.10 Naming Conventions	21
3. Equations	23
3.1 General.....	23
3.2 Operators and Mathematical Functions	23
3.2.1 Arithmetic operators	23
3.2.2 Logical operators	24
3.2.3 Mathematical functions.....	25
3.3 Assignment Equations	25
3.3.1 Equations involving scalars.....	26
3.3.2 Equations involving PLISTS	26
3.3.3 Using SLISTS in an equation.....	27
3.3.4 Using CLIST properties in an equation	27
3.3.5 List-derived scalar variables.....	28
3.4 Selection Equations	29
3.5 String Equations	30
4. Functions	32
4.1 General.....	32
4.2 Function Protocol	32
4.3 Arguments and Subarguments	33
4.4 Selection Equations	33
4.5 Using Variables as Function Arguments	34
5. Writing Model Input Files	36
5.1 General.....	36

5.2 Embedded Functions	36
5.3 Transferring Scalars and PLIST Elements.....	38
5.3.1 User-supplied element referencing	38
5.3.2 File-based element referencing	39
5.4 Writing Entire Lists or Sublists.....	40
5.5 Parameter Delimiter - Alternative Definition	41
5.6 A Cautionary Note.....	41
5.7 Other Ways to Write Model Input Files	42
6. Loops	43
6.1.1 General.....	43
6.1.2 String Equations	43
6.1.3 Function Arguments as Scalar and String Variables	43
6.1.4 \$scalar\$ Substitution.....	46
6.1.5 The <i>goto()</i> Function	47
6.1.6 Command Line Variable Definition.....	48
7. Conclusions.....	50
8. References.....	51
alluv_boundary_interp().....	53
General	53
Function Specifications	53
Function value.....	53
Arguments and subarguments.....	53
Discussion.....	54
The alluvial boundary file.....	54
Interpolation to target PLIST elements	56
Interpolated data types.....	58
Use of the function	58
Examples	59
Example 1	59
Example 2	59
Example 3	59
assign_by_relation().....	60
General	60
Function Specifications	60
Function value.....	60
Object associations	60
Arguments.....	60
Discussion.....	62
Function algorithmic details	62
Some function uses.....	62
Examples	63
Example 1	63
Example 2	63
Example 3	63
assign_by_slist()	64
General	64
Function Specifications	64
Function value.....	64

Object associations	64
Arguments.....	64
Discussion.....	64
Examples	65
Example 1	65
Example 2	65
assign_from_closest()	66
General	66
Function Specifications	66
Function value.....	66
Object associations	66
Arguments and subarguments.....	66
Discussion.....	67
Examples	68
Example 1	68
Example 2	68
assign_if_colocated_2d().....	69
General	69
Function Specifications	69
Function value.....	69
Object associations	69
Arguments.....	69
Discussion.....	69
Examples	70
Example 1	70
Example 2	70
build_covmat_from_vario().....	71
General	71
Function Specifications	71
Function value.....	71
Object associations	71
Arguments and subarguments.....	71
Discussion.....	74
Variogram dimensions.....	74
Variogram properties.....	74
Spatially varying variograms	74
Examples	75
Example 1	75
Example 2	75
Example 3	75
Example 4	75
Example 5	75
calc_kriging_factors_2d().....	77
General	77
Function Specifications	77
Function value.....	77
Arguments and subarguments.....	77
Discussion.....	79

Variograms.....	79
Simple and ordinary kriging	79
Examples	79
Example 1	79
Example 2	80
Example 3	80
<code>calc_kriging_factors_auto_2d()</code>	81
General	81
Function Specifications	81
Function value.....	81
Arguments and subarguments.....	81
Discussion.....	82
Interpolation and implementation details	82
Simple and ordinary kriging	84
Strong enforcement of anisotropy	84
Examples	84
Example 1	84
Example 2	84
Example 3	84
Example 4	85
<code>calc_kriging_factors_3d()</code>	86
General	86
Function Specifications	86
Function value.....	86
Arguments and subarguments.....	86
Discussion.....	88
Variograms.....	88
Simple and ordinary kriging	88
Examples	88
Example 1	88
Example 2	89
<code>calc_linear_interp_factors()</code>	90
General	90
Function Specifications	90
Function value.....	90
Arguments and subarguments.....	90
Discussion.....	91
Linkage of a SEGLIST to a CLIST.....	91
Examples	94
Example 1	94
Example 1	94
<code>calc_rbf_factors_2d()</code>	95
General	95
Function Specifications	95
Function value.....	95
Arguments and subarguments.....	95
Discussion.....	96
Examples	97

Example 1	97
Example 2	97
Example 3	98
Example 4	98
find_cells_in_lists()	101
General	101
Function Specifications	101
Function value	101
Arguments	101
Discussion	102
Model input files	102
Model type	104
Blocks to read	104
Starting and ending strings	105
Comments	105
Examples	105
Example 1	105
Example 2	105
Example 3	106
Example 4	106
gen_ran_plists_basic()	107
General	107
Function Specifications	107
Function value	107
Object associations	107
Arguments and subarguments	107
Discussion	110
Legal argument values	110
Global values or PLISTs	110
Covariance matrix	110
Staged filling of PLISTs	111
Examples	111
Example 1	111
Example 2	112
Example 3	112
gen_ran_plists_cond()	114
General	114
Function Specifications	114
Function value	114
Object associations	114
Arguments and subarguments	114
Discussion	117
Theory	117
Conditioning values	118
Selection equations	119
Limits	120
A final warning	120
Examples	120

Example 1	120
Example 2	121
Example 3	121
goto()	123
General	123
Function Specifications	123
Function value	123
Arguments and subarguments	123
Discussion	123
Example	123
Example 1	123
inform_from_gplane()	125
General	125
Function Specifications	125
Function value	125
Object associations	125
Arguments and subarguments	125
Discussion	126
Example	126
interp_using_file()	128
General	128
ivd_interpolate_2d()	129
General	129
Function Specifications	129
Function value	129
Object associations	129
Arguments and subarguments	129
Discussion	130
Example	130
ivd_sda_interpolate_2d()	132
General	132
Function Specifications	132
Function value	132
Object associations	132
Arguments and subarguments	132
Discussion	135
General	135
Anisotropy interpolation	135
Interpolation from source to target PLISTs	136
Examples	137
Example 1	137
Example 2	138
Example 3	138
Example 4	139
krige_using_file()	140
General	140
Function Specifications	140
Function value	140

Object associations	140
Arguments and subarguments	140
Discussion	141
General	141
Element selection	141
Transform	142
Simple and ordinary kriging	142
Examples	142
Example 1	142
Example 2	142
Example 3	142
link_seglist_to_clist()	144
General	144
Function Specifications	144
Arguments and subarguments	144
Discussion	145
How linkages are made	145
Number of linkages	145
Reporting	145
An Example	146
mremove()	147
General	147
Function Specifications	147
Function value	147
Object associations	147
Arguments	147
Examples	147
Example 1	147
Example 2	147
new_gridded_clist()	148
General	148
Function Specifications	148
Function value	148
Arguments and subarguments	148
Discussion	150
Examples	151
Example 1	151
Example 2	151
Example 2	152
new_mlist()	153
General	153
Function Specifications	153
Function value	153
Object associations	153
Arguments and subarguments	153
Discussion	153
Example	154
new_gplane_clist()	155

General	155
Function value	155
Arguments	155
Discussion	157
GPLANE specification file	157
Location of the GPLANE	159
Within-GPLANE interpolation	160
Reporting	160
Examples	161
Example 1	161
Example 2	161
Example 3	161
new_plist()	163
General	163
Function Specifications	163
Function value	163
Arguments	163
Example	163
new_slist()	164
General	164
Function Specifications	164
Function value	164
Arguments	164
Discussion	164
Examples	165
Example 1	165
Example 2	165
parameter_delimiter()	166
General	166
Function Specifications	166
Function value	166
Arguments	166
Examples	166
Example 1	166
Example 2	167
partition_by_clist()	168
General	168
Function Specifications	168
Function value	168
Object associations	168
Arguments	168
Example	168
partition_by_eqn()	170
General	170
Function Specifications	170
Function value	170
Object associations	170
Arguments	170

Discussion.....	170
Example.....	171
partition_by_file()	172
General	172
Function Specifications	172
Function value.....	172
Object associations	172
Arguments.....	172
Discussion.....	173
Example.....	173
rbf_interpolate_2d()	174
General	174
Function Specifications	174
Function value.....	174
Object associations	174
Arguments and subarguments.....	174
Discussion.....	176
Radial basis functions.....	176
Choices and consequences.....	178
Epsilon and epsminsepfac.....	179
Examples	179
Example 1	179
Example 2	179
Example 3	180
Example 4	180
rbf_interpolate_3d()	181
General	181
Function Specifications	181
Function value.....	181
Object associations	181
Arguments and subarguments.....	181
Discussion.....	183
Examples	183
Example 1	183
Example 2	183
Example 3	184
rbf_sda_interpolate_2d()	185
General	185
Function Specifications	185
Function value.....	185
Object associations	185
Arguments and subarguments.....	185
Discussion.....	188
General.....	188
Anisotropy interpolation.....	188
Interpolation from source to target PLISTs	189
Examples	190
Example 1	190

Example 2	191
Example 3	191
Example 4	192
rbf_using_file()	193
General	193
Function Specifications	193
Function value	193
Object associations	193
Arguments and subarguments	193
Discussion	194
General	194
Element selection	195
Transform	195
Examples	195
Example 1	195
Example 2	195
read_column_data_file()	196
General	196
Function Specifications	196
Function value	196
Arguments and subarguments	196
Discussion	197
Examples	198
Example 1	198
Example 2	198
read_list_as_array()	199
General	199
Function Specifications	199
Function value	199
Object associations	199
Arguments	199
Discussion	199
Examples	200
Example 1	200
Example 2	200
read_list_file()	201
General	201
A List File	201
Function Specifications	202
Function value	202
Arguments and subarguments	202
Examples	204
Example 1	204
Example 2	204
Example 3	205
Additional Notes	205
read_matrix_from_file()	207
General	207

Function Specifications	207
Function value	207
Arguments and subarguments	207
Discussion	207
Examples	207
Example 1	207
Example 2	208
read_mf_grid_specs()	209
General	209
MODFLOW Grid Specification File	209
Function Specifications	209
Function value	209
Arguments	209
Discussion	210
Example	210
read_mf_int_array()	211
General	211
Integer Array File	211
Function Specifications	211
Function value	211
Arguments	211
Discussion	211
Examples	212
Example 1	212
Example 2	212
read_mf_real_array()	213
General	213
Real Array File	213
Function Specifications	213
Function value	213
Arguments	213
Discussion	213
Examples	214
Example 1	214
Example 2	214
read_mf_usg_grid_specs()	215
General	215
MODFLOW-USG Grid Specification File	215
Function Specifications	216
Function value	216
Arguments	217
Examples	217
Example 1	217
Example 2	217
read_mf6_grid_specs()	218
General	218
Function Specifications	218
Function value	218

Arguments.....	218
Discussion.....	219
Examples	220
Example 1	220
Example 2	220
read_multiple_array_file().....	221
General	221
Multiple Array File	221
Function Specifications	222
Function value.....	222
Arguments and subarguments.....	222
Discussion.....	223
Examples	223
Example 1	223
Example 2	224
read_restart_data()	225
General	225
Function Specifications	225
Function value.....	225
Arguments.....	225
Discussion.....	225
Example.....	225
read_scalar_file()	226
General	226
Scalar File.....	226
Function Specifications	226
Function value.....	226
Arguments.....	226
Discussion.....	227
Example.....	227
read_segfile()	228
General	228
Segment File	228
Function Specifications	229
Function value.....	229
Arguments.....	229
Discussion.....	229
Example.....	230
remove()	231
General	231
Function Specifications	231
Function value.....	231
Object associations	231
Arguments.....	231
Example.....	231
replace_cells_in_lists	232
General	232
Function Specifications	232

Function value	232
Arguments	233
Discussion	235
Examples	237
Example 1	237
Example 2	237
Example 3	237
replace_column()	239
General	239
Function Specifications	239
Function value	239
Arguments and subarguments	239
Discussion	240
Examples	242
Example 1	242
Example 2	243
Example 3	243
report_all_entities()	244
General	244
Function Specifications	244
Function value	244
Arguments	244
Example	244
report_dependent_lists()	245
General	245
Function Specifications	245
Function value	245
Object associations	245
Arguments	245
Example	245
save_restart_data()	246
General	246
Function Specifications	246
Function value	246
Arguments	246
Example	246
stat()	247
General	247
Function Specifications	247
Function value	247
Object associations	247
Arguments	247
Example	247
Example 1	247
Example 2	247
stop()	248
General	248
Example	248

upscale_by_averaging().....	249
General	249
Function Specifications	250
Function value.....	250
Object associations	250
Arguments and subarguments.....	250
Discussion.....	251
Examples	253
Example 1	253
Example 2	254
write_column_data_file()	255
General	255
Function Specifications	255
Function value.....	255
Arguments and subarguments.....	255
Discussion.....	256
Examples	256
Example 1	256
Example 2	256
write_in_sequence()	257
General	257
Function Specifications	257
Function value.....	257
Object associations	257
Arguments.....	257
Examples	258
Example 1	258
Example 2	258
Example 3	258
Example 4	258
write_matrix_to_file()	259
General	259
Function Specifications	259
Function value.....	259
Arguments and subarguments.....	259
Examples	259
Example 1	259
Example 2	259
write_model_input_file().....	260
General	260
Function Specifications	260
Function value.....	260
Arguments.....	260
Example.....	260

1. Introduction

1.1 General

“PLPROC” stands for “Parameter List PROCessor”. It was written to expedite parameterization of numerical models for which system properties must be specified at many grid or mesh nodes, elements or cells. For brevity this document will use the word “grid” instead of “grid or mesh”, and “node” to encompass “node”, “cell” and “element”. Values of system properties, and/or system inputs assigned to nodes, will be referred to as “parameters”. Parameters often require alteration during model calibration and uncertainty analysis.

The number of parameter values required by a numerical model can number in the millions for, conceptually, a different set of system properties and/or system stresses may be assigned to every model node. Where nodes are arranged in a regular grid or mesh, parameter values are often supplied in array format. However where a grid is irregular, parameter values are often supplied as lists of values. Sublists may then be employed for the assignment of parameter values to subsets of the total number of nodes within the entire model domain. These subsets may pertain to individual layers, or to those parts of a model domain to which certain boundary conditions and/or source terms are applied.

The huge number of parameter values that are supplied to the nodes of a numerical model are often calculated from a far smaller number of “adjustable parameters”. When a model is being calibrated, it is this latter set of parameters whose values are altered, either manually or by software such as PEST, in an attempt to match model outputs to historical observations of system state. A model preprocessor must then calculate grid-based parameter values from the values of adjustable parameters, writing input files for the numerical model as it does so. PLPROC is meant to serve such a preprocessing role.

Despite the fact that adjustable parameters may be far fewer in number than grid-based parameters, they may still number in the thousands. This will indeed be the case if calibration of a model is undertaken using the highly parameterized methodologies available through the PEST suite. These parameters too may be considered as lists of numbers. However members of adjustable parameter lists may pertain to entities such as pilot points, coefficients of spatial or temporal basis functions, and to zones of piecewise spatial or temporal constancy.

PLPROC is designed to be used in partnership with PEST, and with other model-independent software which employs PEST-like protocols for communicating with models. PLPROC employs the template file concept to write grid-based parameter values to model input files. However in order to reduce the work required of a modeller to write such template files, given the large numbers of parameters that they may cite, PLPROC employs an expanded template file protocol. Unlike PEST template files, PLPROC template files can include statements (referred to as “embedded functions”) which cite entire lists or sublists of parameters rather than individual parameters. Furthermore, where individual parameter list entities are referenced, list element designation can employ numbers which are already recorded in a template file. Use of these protocols not only facilitates construction of large template files which can govern the writing of model input files that cite innumerable grid-based parameters; they also contribute to the model-independent nature of the parameter value transfer process in general, and of PLPROC’s contribution to that process in particular.

PLPROC provides a number of means through which parameters comprising one list (for example the list of parameters pertaining to a model grid) can be calculated from those comprising another (for example the list of parameters associated with a family of pilot points). These means can be considered to comprise the “computational engine” of PLPROC. At the time of writing, these include most of those provided by the PEST Groundwater Data Utilities suite. These options continue to expand over time; hence PLPROC, rather than the Groundwater Data Utility suite, constitutes the software embodiment of parameter preprocessing functionality necessary for enabling optimal PEST usage with complex spatial models.

Further flexibility in adjustable and grid-based parameter definition is provided by PLPROC’s ability to perform arbitrary mathematical manipulation between compatible parameter lists based on user-supplied equations. A single equation can be applied to all list elements, or to a subset of lists elements based on a so-called “selection equation” whose outcome must be of the logical type. As well as parameter lists, PLPROC equations can employ scalar parameters, both as computational elements and as entities to which equation outcomes are assigned. The latter can then be written to model input files using templates of the latter, along with parameter lists. PLPROC can therefore be used in place of the parameter preprocessor PAR2PAR supplied with PEST.

A PLPROC user invokes various aspects of PLPROC functionality by providing it with a text input file containing a set of statements or commands which tell it what to do next. Each of these statements calls a function or embodies an equation; some processing control functionality is also available. If a function is called, the specifics of the action invoked by the function are conveyed to PLPROC through the function’s arguments. The syntax employed for function calls resembles that of programming languages with which many PLPROC users will be familiar. A sequence of PLPROC commands provided in a single such input file can thus be considered to be a “program” or “script”; the latter terminology will be employed herein. If a particular command within a script does not constitute an equation, then it usually invokes a function - thereby commanding PLPROC to undertake some manner of parameter list processing. Such processing can include (but is not limited to):

- definition of a new parameter list;
- reading of scalar or list values from an input file;
- creation of a sublist from an entire list;
- one-, two- or three-dimensional interpolation from one list to another;
- writing of list values to a model input file;
- reporting on the status of current PLPROC entities.

In summary, PLPROC is a parameter-list processor designed specifically to facilitate calibration of large numerical models by serving as a model-independent preprocessor for such models. Through a combination of:

- bulk manipulation of parameters contained in user-defined parameter lists;
- assignment of values to members of one list based on values of non-congruent lists;
- an ability to read parameter values from files that employ a number of different formats; and
- an ability to write parameter values to model input files of arbitrary construction using template files that include embedded functions or references to the elements of model cells that are linked to parameter lists;

PLPROC provides a modeller with the ability to create and manipulate parameters that are defined in a manner that makes them suitable for calibration-constrained manipulation, as these in turn inform the node-based parameters of a numerical model. In doing this PLPROC supports PEST in facilitating model-based decision-support based on the premise that a model should encapsulate what we know while quantifying what we don't.

1.2 PLPROC Input

As stated above, PLPROC receives instructions through commands recorded in a text file, this being referred to herein as a “script file”. Collectively the set of commands within a script file is referred to as a “PLPROC script”. Figure 1.1 shows an example of a PLPROC script. The syntactical similarities to programming languages such as Python and FORTRAN are obvious.

```
# -- The grid spec file is read.

cl_m = read_mf_usg_grid_specs(file="model_usg.spc")

# -- List details are recorded so that we can relate layer numbers to elements.

cl_m.report_dependent_lists(file='temp.dat')

# -- A PLIST and SLIST are created for the model grid.

pm_1=new_plist(reference_clist=cl_m,value=1.01e30)

# -- Model domain zonation is determined by reading multiple integer array files.

tmp=new_plist(reference_clist=cl_m,value=1.0)
tmp(select=(layer==7)|| (layer==8)|| (layer==9))=2.0
sm_1=new_slist(plist=tmp)
tmp.remove

# -- The pilot points file is now read. This also creates a PLIST and an SLIST.

cl_pp = read_list_file(dimensions=3,                                &
                      slist='sp_1';column=5,                      &
                      plist='pp_1';column=6,                      &
                      id_type='character',file='pp3d_sl.dat')

# -- Interpolation factors are now calculated for zone 1.

calc_kriging_factors_3d(target_clist=cl_m;select=(sm_1==1),        &
                        source_clist=cl_pp;select=(sp_1==1),      &
                        file='fac1.dat';form='text',              &
                        variogram='exponential',                  &
                        nugget=0.1,sill=0.3,                      &
                        a_hmax=50000,a_hmin=20000,a_vert=500,     &
                        ang1=45,ang2=20,ang3=0.0,                 &
                        kriging='ordinary',                        &
                        search_rad_max_hdir=1e20,                 &
                        search_rad_min_hdir=1e20,                 &
                        search_rad_vert=1e20,                     &
                        min_points=1,max_points=50)

# -- Interpolation factors are now calculated for zone 2.

calc_kriging_factors_3d(target_clist=cl_m;select=(sm_1==2),        &
                        source_clist=cl_pp;select=(sp_1==2),      &
                        file='fac2.dat';form='text',              &
                        variogram='exponential',                  &
                        a_hmax=200000,a_hmin=50000,a_vert=100,    &
```

```

        ang1=90,ang2=0,ang3=0,                &
        mean=1.0,                             &
        kriging='simple',                       &
        search_rad_max_hdir=1e20,search_rad_min_hdir=1.0e20, &
        search_rad_vert=1e20,                  &
        min_points=1,max_points=50)

# -- Interpolation is carried out.

pm_1=pp_1.krige_using_file(file='fac1.dat';form='text',transform='log')
pm_1=pp_1.krige_using_file(file='fac2.dat';form='text',transform='none')

# -- List details are written out so we can relate layer numbers to elements.

cl_m.report_dependent_lists(file='temp.dat')

# -- Write a model input file using a template file.

write_model_input_file(template_file='model.tpl',                &
                        model_input_file='model.dat')
```

Figure 1.1 Example of a PLPROC script.

The above example demonstrates a number of aspects of PLPROC script syntax. Note, in particular, the following.

- Blank lines are ignored.
- Spaces on any line are ignored unless they are situated between quotes.
- Any characters following a “#” character on any line are ignored. Thus a user can insert comments on any line, or devote the entirety of a line to a comment.
- The “&” character signifies continuation of a command if placed at the end of a line or is the last active character before a “#” character. It informs PLPROC that the following line is not a new PLPROC command, but is in fact a continuation of the same command.

Note also that all PLPROC script input (except for filenames) is case-insensitive; variable and object names are also case-insensitive. All text (except for filenames) is converted to lower case as it is read.

The *stop()* command can be used to terminate PLPROC’s processing of its command sequence immediately, regardless of the presence of any further script commands. The *goto()* command can alter PLPROC’s processing sequence from that provided in a script.

1.3 Running PLPROC

PLPROC is run using the command:

```
plproc filename
```

where *filename* is the name of a script file. Alternatively, if invoked without a command line argument, PLPROC prompts for the name of its script file as follows:

```
Enter name of PLPROC script file:
```

As it runs, PLPROC writes to the screen the first 80 characters of the command that it is currently processing. If an error condition arises, that error is detected. PLPROC then ceases execution, writing an appropriate message to the screen as it does so.

2. Lists, Gplanes, Scalars, Strings and Matrices

2.1 General

As the name implies, a “parameter list” is a list of numbers. The numbers comprising a list generally bear some relationship to each other, thereby comprising a “type” or a “family”. For example they can represent the values of a particular hydraulic property at different places within a model domain, or in a single layer of a model domain. Integer lists can represent the distribution of a category, for example rock type or soil type, over a model domain. Parameters within lists are generally associated with entities that have a position in space. These entities may be, for example, model nodes, groups of model nodes comprising a zone, or model-independent pilot points.

2.2 CLISTs, SLISTs and PLISTs

PLPROC supports four types of lists. SEGLISTS are discussed in the next section; these are not intimately associated with other list types as are CLISTs, SLISTs and PLISTs.

Of all the list types, only PLISTs, actually contain parameter values. Within a PLIST these are stored as a one-dimensional array of double precision numbers. On the other hand an SLIST (“s” stands for “selection”) is a one-dimensional array of integers. The zonation that may be defined on the basis of these integers can be used as a basis for selective processing of PLIST elements when using PLPROC equations and some of its functions.

Neither a PLIST nor an SLIST can exist on its own. Each must possess a “reference CLIST” (where “c” stands for “coordinates”). A CLIST provides the spatial coordinates associated with each element of its dependent PLISTs and SLISTs. It also provides the indexing through which individual list elements can be accessed.

A CLIST can have multiple dependent PLISTS and SLISTS. The relationship is shown schematically in Figure 2.1.

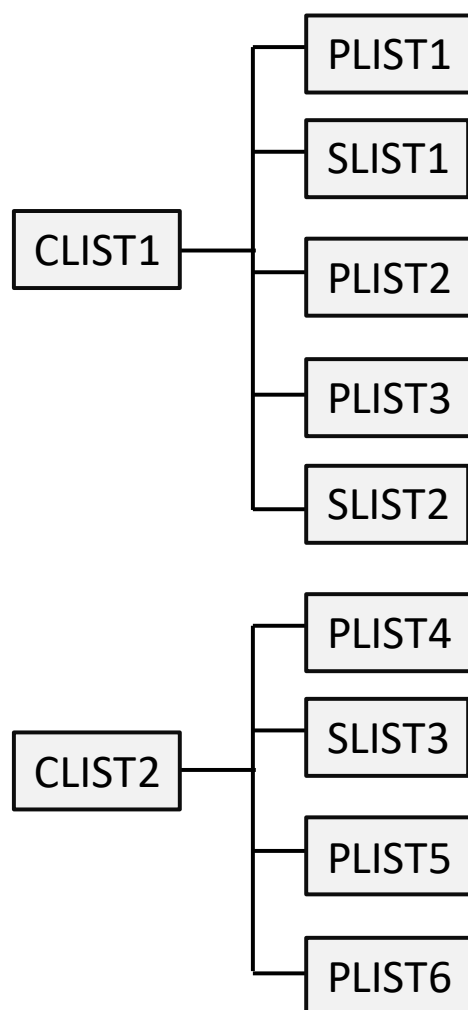


Figure 2.1 Two CLISTs and their dependent SLISTs and PLISTs.

Every PLIST, SLIST and CLIST must be given a name of 20 characters or less in length (with no spaces). The name must be unique, and must not clash with that of any PLPROC function or standard mathematical function (such as “cos” or “tan”). In addition to this, certain characters are banned from PLPROC entity names, these being those associated with mathematical operations, such as “*”, “/” etc.

2.3 CLIST Options

2.3.1 Dimensionality

When defined, each CLIST must be declared as either three dimensional or two dimensional. In the former case an x , y and z coordinate must be provided for each element. In the latter case only x and y coordinates are required for each element.

2.3.2 Element referencing

Internally, the elements of a CLIST are referenced by index number. Index numbers are sequential integers. The first index is arbitrary and is subject to user definition; if no such definition is made, the first index is 1. Externally (i.e. in PLPROC scripts) individual elements of a CLIST (and its dependent SLISTs and PLISTs) can be referenced in one of

three different ways, this depending on the CLIST's user-specified element identification type (or "id_type" for short).

The three element identification options are "indexed", "integer" and "character". If a CLIST's id-type is "integer", a user may refer to individual list elements through a user-specified integer that is associated with each such element. This integer must be unique within the CLIST. Alternatively, if the CLIST id-type is "indexed" the integer that is used to refer to a specific list element is the actual index of that element. Use of this id-type reduces CLIST storage requirements; it also reduces access times to individual CLIST (and dependent SLIST and PLIST) elements where these lists are large. The final option for list element referencing is "character". In this case each CLIST element can be provided with a unique (to each CLIST) character label of 20 characters or less in length. This label can then be used to identify an individual CLIST element. (Note that in many instances of list processing the element number can be used for individual element referencing if desired, regardless of a CLIST's id-type.)

There are advantages and disadvantages associated with each of the above element identification protocols. An advantage of character and integer identification is that PLPROC can retain the same element referencing as a user may employ for representation of entities outside of PLPROC. Thus if, for example, pilot points are endowed with names such as "113-32-3" where "113" is the model row number, "32" is the model column number and "3" is the model layer number at which a pilot point has been emplaced, this character string can be read during definition of a pilot-point-based CLIST, and then maintained within PLPROC as a mechanism for referencing that element. A disadvantage of character referencing, however, is that it cannot be used as a basis for element selection using selection equations. In contrast, an integer identifier can be cited in PLPROC selection equations. Hence if the above pilot point was associated with the integer "235313", this perhaps indicating the node number with which the pilot point's location coincides, that integer can be used in any equation that governs selection or processing of parameters which reside in PLISTs which are dependent on the pilot-point CLIST. It is important to note that user-specified integer identifiers do not need to be sequential. All that is required is uniqueness of integers within any particular CLIST.

As stated above, use of indexed element identification accrues savings in PLPROC memory requirements, and possibly very large savings in element access times (should access to individual elements be required in any of the commands comprising a script). Increased element access efficiency is an outcome of the fact that the location of each element is known in advance through the sequencing that is implicit in indexed element identification; a user-specified element does not therefore need to be found through conducting a search of the CLIST. Use of certain functions can also become very much faster if indexed element identification is employed. For example if the *read_list_file()* function is used to add extra dependent PLISTs and/or SLISTs to a parent CLIST, the savings in processing time for large CLISTs can be considerable.

2.3.3 Dependent SLISTs and PLISTs

SLISTs and PLISTs inherit their dimensionality, coordinates and element referencing from their parent CLISTs. Thus a child SLIST or PLIST of a two-dimensional CLIST has no *z* coordinates associated with its elements; the *x* and *y* coordinates associated with each of its elements are those of its reference CLIST. If individual elements of a CLIST are identified by

character string, then so too are individual elements of its dependent SLISTs and PLISTs; the same character string cites the same corresponding element of each.

2.3.4 Defining Lists

The various types of lists used by PLPROC can come into existence in a variety of ways. Some of these are now discussed.

The definition of a CLIST must be accompanied by the assignment of coordinates to all of its elements. One means through which the creation of a CLIST can be accomplished is through reading the coordinates of its elements from a file. Functions such as *read_list_file()* provide this capability. Using this function element identifiers are read at the same time; optionally, data pertaining to one or a number of dependent SLISTs and PLISTs can also be read.

A new CLIST can be defined through extraction from a larger CLIST using the *partition_by_eqn()* or *partition_by_file()* functions. In these cases the new CLIST simply inherits its characteristics from the existing one. However some nuances of the inheritance process require careful consideration if the original CLIST employs indexed element referencing and the extraction process does not maintain juxtaposition of previously sequential elements; see documentation of these functions for further details.

Once a CLIST has been created, PLISTs and SLISTs which reference that CLIST can be readily defined using the *new_slist()* and *new_plist()* functions. Through use of the *new_plist()* function all elements of the newly-created PLIST are provided with a single value. The *new_slist()* function also provides this option; alternatively SLIST element values can be defined as the nearest integers to corresponding elements of an existing PLIST. Once a PLIST has been defined through the *new_plist()* function, its element values can be subsequently altered using pertinent PLPROC functions and equations. SLISTs and PLISTs may also be created, and values for all of their elements read from a file, using the *read_list_file()* function.

PLPROC equation functionality provides another means of PLIST creation. If the outcome of an equation is obviously a PLIST and the pertinent PLIST does not yet exist, then evaluation of the equation brings the PLIST into existence. However, as will be described in the next chapter, this depends on certain conditions being met, notably that no selection equations restrict evaluation of the PLIST assignment equation to a limited number of its elements.

CLISTs, SLISTs and PLISTs can be removed from PLPROC memory through use of the *remove()* function.

2.3.5 Gridded CLISTs

If a CLIST represents the nodes of a node-centred regular or semi-regular grid, then it is useful to recognize and denote it as such, as this can make certain processing options easier and can facilitate data exchange with grid-based external programs and simulators.

Element identification for CLISTs which are specifically denoted as gridded is indexed. The index of the first element is 1 while the index of the last element is equal to the number of elements comprising the CLIST. The latter is equal to the product of the number of grid rows and the number of grid columns where a CLIST is two-dimensional, or the product of the number of grid rows, the number of grid columns and the number of grid layers where a grid is three-dimensional.

PLPROC uses a single integer to represent the gridded (or otherwise) status of a CLIST; this is referred to as its *grid_type* designator. *grid_type* is an integer which can be positive or negative, its sign indicating the ordering of CLIST elements; it is zero if the CLIST does not pertain to a grid. If the sign of *grid_type* is negative, then grid element (1,1) for a two dimensional grid or (1,1,1) for a three-dimensional grid is situated at the bottom, left, front corner of the grid. Element numbering then proceeds along the grid, with most rapid variation being in the *x* direction, then the *y* direction, and then the *z* direction (if the grid is three-dimensional). For this type of grid, it is convenient to denote the number of columns as *nx*, the number of rows as *ny* and the number of layers as *nz* where *x*, *y* and *z* follow the normal directions of a right-handed coordinate system. Suppose that an element of this grid is identified as (*ix*, *iy*, *iz*). Then the number of the corresponding gridded CLIST element is calculated as $ix + (iy-1) \times nx + (iz-1) \times nx \times ny$.

If the sign of *grid_type* is positive, then cell (1,1,1) of the grid is assumed to be situated at its top left corner. Rather than using *nx*, *ny*, and *nz* to designate grid dimensions, dimensions are instead specified using *ncol*, *nrow* and *nlay* (like a MODFLOW grid). Layer numbers increase downward while row numbers increase from back to front. For grid element (*icol*, *irrow*, *ilay*), the index of the corresponding CLIST element is calculated as $icol + (irrow-1) \times ncol + (ilay-1) \times ncol \times nrow$.

The nature of a grid is such that real-world coordinates need to be provided for only one node or vertex of the grid; real-world coordinates for other nodes and vertices can then be readily calculated from grid dimensions. Where *grid_type* is negative, real-world coordinates (*x0*, *y0*, *z0*) are provided for the bottom, left, front corner of the grid, i.e. for the bottom, left, front vertex of the cell whose (*ix*, *iy*, *iz*) coordinates are (1,1,1). Where *grid_type* is positive, real-world coordinates are provided for the top, left, back corner of the grid (i.e. the top, left back vertex of the cell whose (*icol*, *irrow*, *ilay*) coordinates are (1,1,1). In addition to this, a *rotation* angle must be provided; this is the angle between the direction of increasing *ix* or *icol* index, and east. Protocols for the two grid types are illustrated in Figures 2.2a and 2.2b.

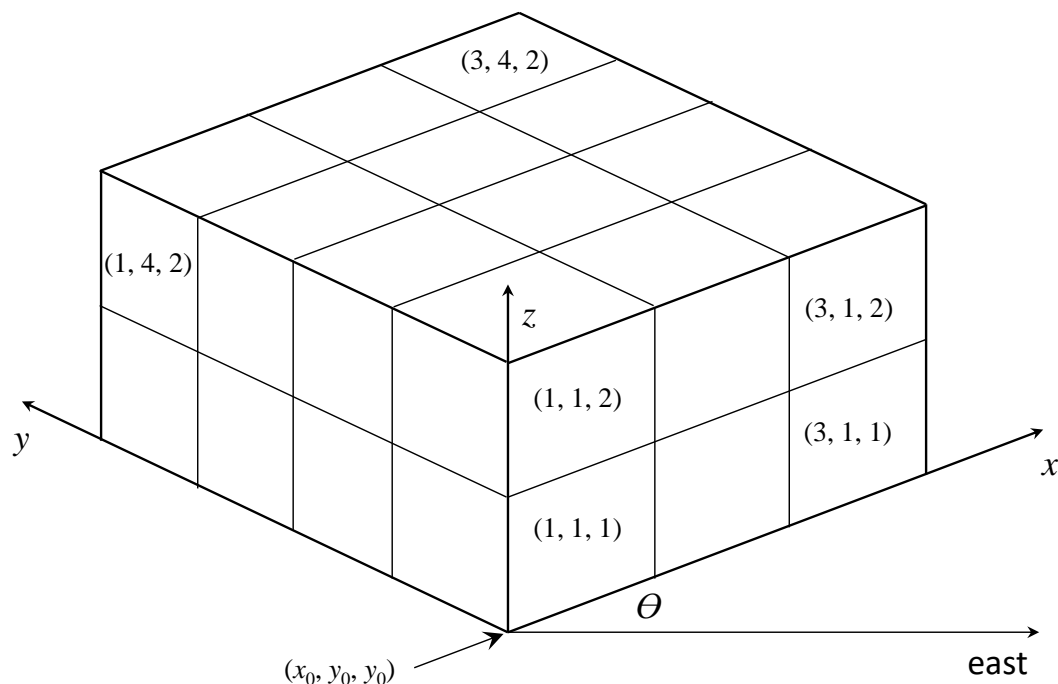


Figure 2.2a. A three-dimensional grid with negative *grid_type* index. θ indicates the grid rotation. (ix, iy, iz) coordinates for some grid cells are shown.

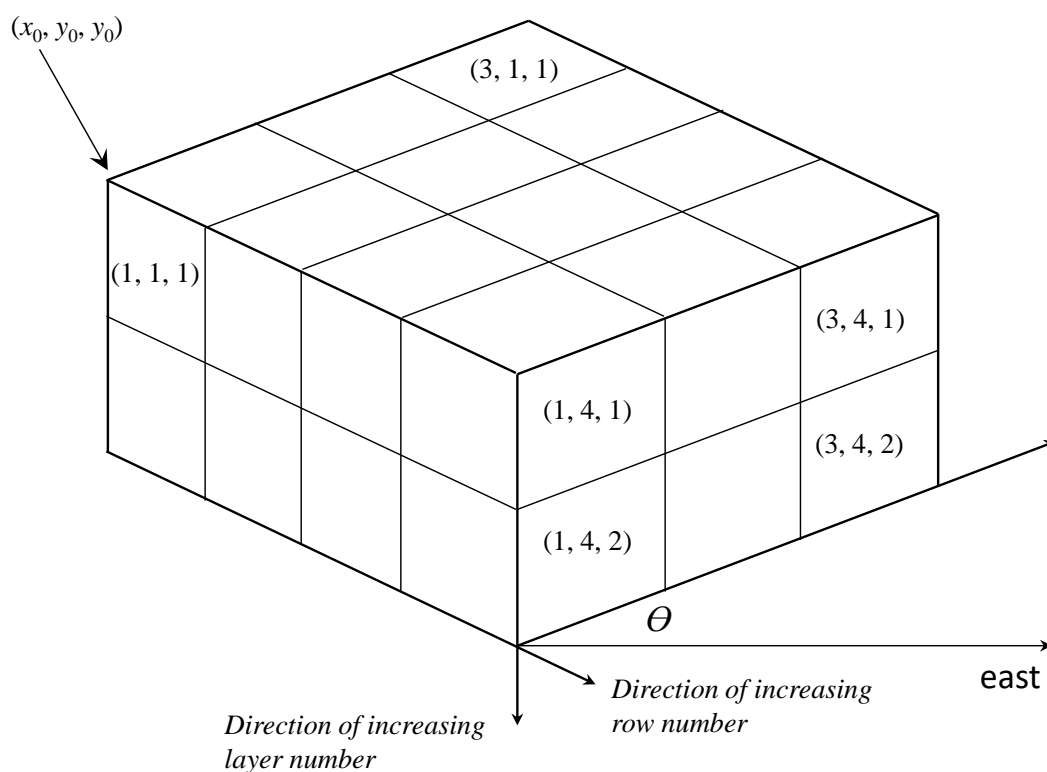


Figure 2.2b. A three-dimensional grid with positive *grid_type* index. θ indicates the grid rotation. $(icol, irow, ilay)$ coordinates for some grid cells are shown.

If a grid is two-dimensional then its *grid_type* index has two digits; in contrast, three digits are used to specify a three-dimensional grid. The first digit specifies whether cell widths in the x direction (for negative *grid_type*) or the column direction (for positive *grid_type*) are

uniform or variable; 1 is used to indicate the first condition and 2 is used to indicate the latter condition. The same protocol applies to the second digit. The third digit is only present if the grid is three dimensional. The value of this third digit can be 1, 2 or 3. A value of 1 indicates uniform cell widths in the z or layer direction while a value of 2 indicates non-uniform cell widths in this direction. A value of 3 indicates that grid node elevations are irregular and must be supplied for every cell in the grid as a three-dimensional array of elevations.

Here are some examples.

- A *grid_type* of -111 indicates a three-dimensional grid whose cells are specified using the (ix, iy, iz) convention; delta- x , delta- y and delta- z (i.e. cell widths in the x , y and z directions) are independent of ix , iy and iz .
- A *grid_type* of -12 indicates a two-dimensional grid whose cells are specified using the (ix, iy) convention. Cell widths in the x direction are constant. However cell widths in the y direction are a function of iy .
- A *grid_type* of 122 indicates a three-dimensional grid whose cells are specified using the $(icol, irow, ilay)$ convention. Cell widths in the direction of increasing column index (these corresponding to the contents of the MODFLOW *delr* or “row direction cell width” array) are uniform, cell widths in the direction of increasing row index (this corresponding to contents of the MODFLOW *delc* or “column direction cell width” array) are row-number-specific, while cell widths in the direction of increasing layer index are layer-specific.
- A *grid_type* of 113 indicates a three-dimensional grid whose cells are specified using the $(icol, irow, ilay)$ convention. Cell widths in the column and row directions are independent of column and row number. However the elevation of each cell centre is non-uniform, this depending on all of *icol*, *irow* and *ilay*.

2.3.6 CLIST to Model Associations

PLPROC provides functions that allow automatic creation of CLISTS and associated PLISTS and SLISTS for some specific model types. These model types include MODFLOW, MODFLOW-USG and MODFLOW6 (for both DIS and DISV grids). CLISTS created in this way are tagged as such. Use of certain PLPROC functions is made easier when a CLIST is recognized as pertaining to a certain model type. This applies particularly to functions such as *find_cells_in_lists()* and *replace_cells_in_lists()* where the model-to-CLIST relationship is exploited for identification of CLIST elements using model cell indices.

2.4 SEGLISTS

A SEGLIST is a set of segments. Each segment is a polyline defined by a set of vertices. As presently coded, only two-dimensional coordinates can be ascribed to these vertices. A segment begins at the first vertex of the vertex list that is attributed to a segment, and ends at the last vertex in the list. Each segment within a SEGLIST has a name (of 20 characters or less in length).

Segments within a SEGLIST are considered to be independent entities. In practice they may collectively define a geographical feature such as a stream or other polylinear feature that is of relevance to a model. Hence the end of one segment may coincide with the beginnings of one or more other segments, and the beginning of one segment may coincide with the ends of one or more other segments.

SEGLIST vertex coordinates and names can be read from a segment file, called a “segfile” for short. An example of such a file is provided in documentation of function *read_segfile()*. This file can be easily prepared with the help of any program with digitizing capabilities, such as a GIS or SURFER.

Segments within a SEGLIST can be linked to elements of a CLIST. PLPROC provides two mechanism for SEGLIST-to-CLIST linkage. The first option is to link each SEGLIST segment to a single CLIST element; ideally, for easy recognition of this linkage, each CLIST element should lie somewhere near the centre of the segment to which it is linked. Alternatively, each SEGLIST segment can be linked to two CLIST elements; ideally these two elements should reside close to opposite ends of the segment to which they are linked. SEGLIST-to-CLIST linkages can be made using function *link_seglist_to_clist()*. Alternatively, CLISTs can be created specifically for the purpose of SEGLIST linkage using function *create_seglist_from_clist()*.

Once SEGLIST-to-CLIST linkages have been made, the stage is set for pilot point parameterization of polylinear model entities. Where each SEGLIST segment is linked to two CLIST elements, interpolation from PLISTs that belong to this CLIST is linear down the segment. A model-based PLIST then inherits interpolated values from the segment point to which it is closest; this point can lie between segment-defining vertices. Where each segment is linked to a single CLIST element, interpolation from a CLIST-owned PLIST to the SEGLIST, and then to closest elements of a model-representative CLIST, is piecewise constant. In either case, interpolation is implemented using the *calc_linear_interp_factors()* function.

2.5 GPLANES

2.5.1 GPLANE concepts

A GPLANE is a plane which hosts a two-dimensional CLIST. This CLIST is a gridded CLIST whose *grid_type* is -11. CLIST element locations are expressed using a local plane-specific coordinate system that is defined as part of GPLANE specifications.

The orientation of a GPLANE is defined by three points. These are labelled (x_{pi}, y_{pi}, z_{pi}) for $i=1$ to 3 in Figure 2.3a. The coordinates ascribed to these points must employ the same “global” coordinate system as that used to ascribe locations to all other non-GPLANE CLISTs defined in other parts of a PLPROC script. These points thus anchor the plane to the “real world”.

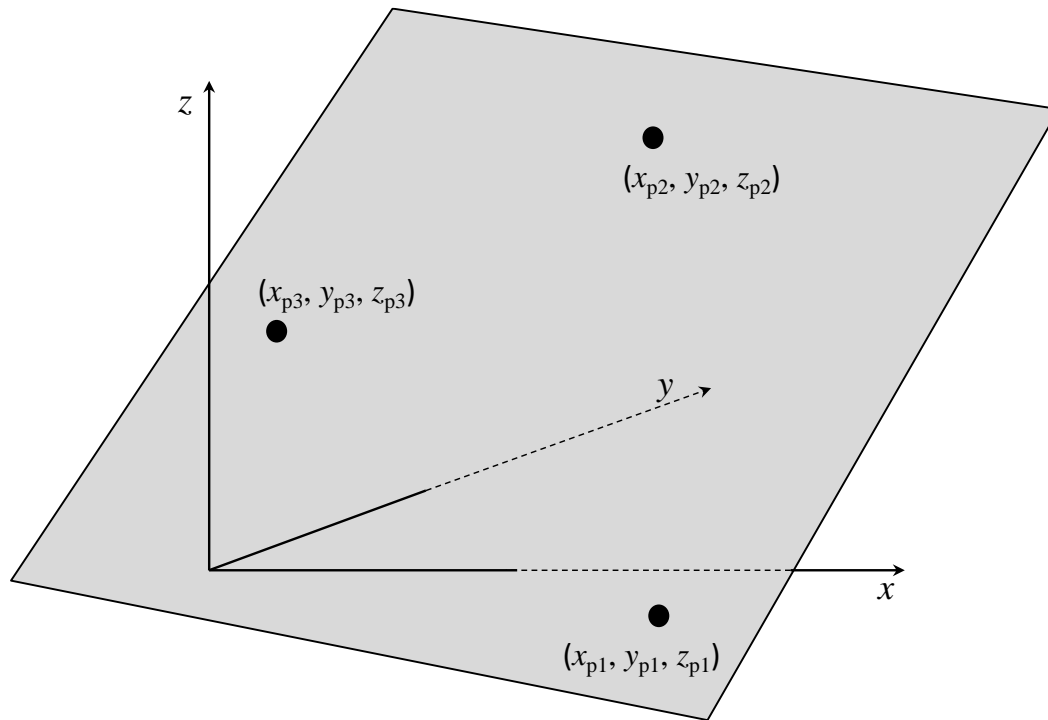


Figure 2.3a A GPLANE with its plane-defining points. The axes depicted in this figure pertain to the real-world coordinate system.

A GPLANE has its own local, specific, planar coordinate system. Like the GPLANE itself, this coordinate system must be related to that of the real world. The user thus specifies the origin of the GPLANE coordinate system using real world coordinates. However only two real-world coordinates must be supplied, these being (x_{g0}, y_{g0}) , (y_{g0}, z_{g0}) or (x_{g0}, z_{g0}) . The choice of the most appropriate pair of x , y and z coordinates to use depends on the orientation of the plane; the missing coordinate is evaluated automatically using the fact that the point must lie in the plane. The user indicates which pair of coordinates that he/she will use to specify these and other points of interest on the GPLANE (see below); PLPROC will complain if this choice is not appropriate.

To complete definition of the local GPLANE coordinate system, a further two points must be supplied using the same protocol. The first is (x_{g1}, y_{g1}) , (y_{g1}, z_{g1}) or (x_{g1}, z_{g1}) . The line between (x_{g0}, y_{g0}) , (y_{g0}, z_{g0}) or (x_{g0}, z_{g0}) and (x_{g1}, y_{g1}) , (y_{g1}, z_{g1}) or (x_{g1}, z_{g1}) defines the direction of the local GPLANE x axis. The second point is (x_{g2}, y_{g2}) , (y_{g2}, z_{g2}) or (x_{g2}, z_{g2}) . A line perpendicular to the just-defined GPLANE x axis in the approximate direction of this second point defines the local GPLANE y axis. The location of this second point does not need to be exact. Its purpose is to indicate which of the two in-plane directions that are perpendicular to the local GPLANE x axis constitutes the direction of the GPLANE y axis. See Figure 2.3b. In this figure the user employs real-world (x, z) coordinates to locate axis-defining points on the plane; meanwhile the GPLANE coordinate system is illustrated using primed x and y coordinates, i.e. (x', y') . Note that PLPROC does not insist that the usual geometric relationship between the direction of increasing x' axis and the direction of increasing y' axis applies. These directions are entirely up to the user to set.

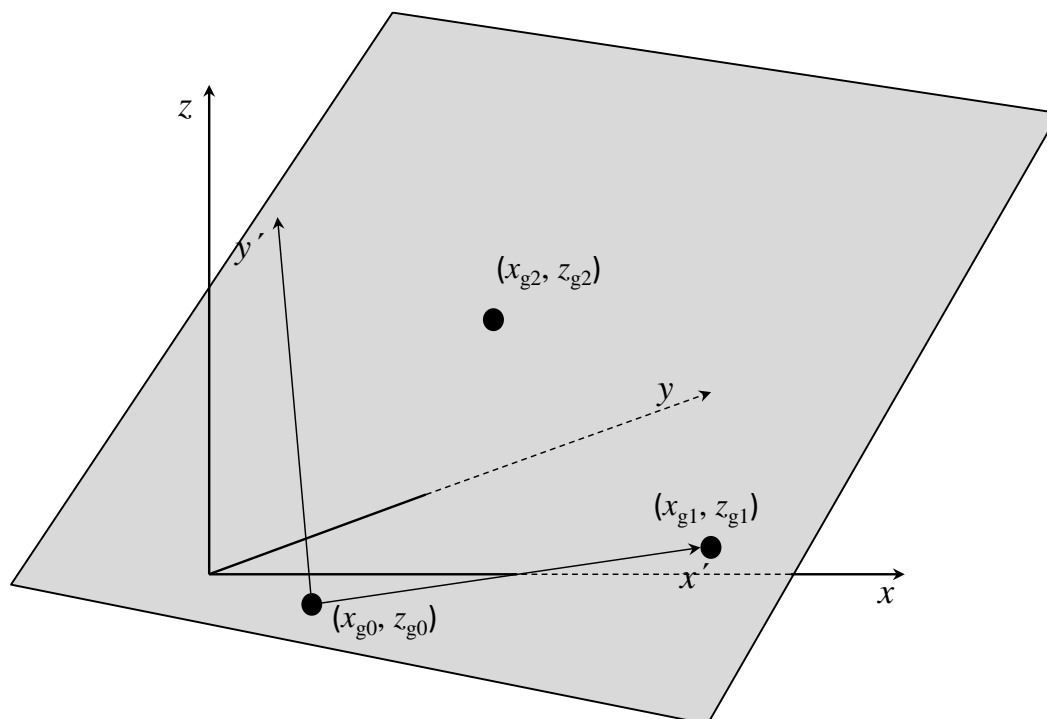


Figure 2.3b. The real-world coordinate system and the local GPLANE coordinate system. Coordinates of the latter, two-dimensional, coordinate system are expressed as (x', y') .

Optionally, a GPLANE may be laterally bounded (as is the plane shown in Figures 2.3a and 2.3b). If this is the case, coordinates of the GPLANE-bounding polygon must also be supplied. To make this job easier for the user, they are supplied as pairs of real-world coordinates, i.e. (x, y) , (y, z) or (x, z) . The same pair of x , y , or z coordinates must be used to specify points along this boundary as those which were employed for specification of the GPLANE coordinate system origin and axis directions; see Figure 2.3c. The bounding polygon can have up to 10 vertices. They must be supplied in order of being encountered during a traverse along the boundary plane. None of the lines connecting any two neighbouring boundary points must cross any of the other lines. (PLPROC will cease execution with an appropriate error message if this occurs.)

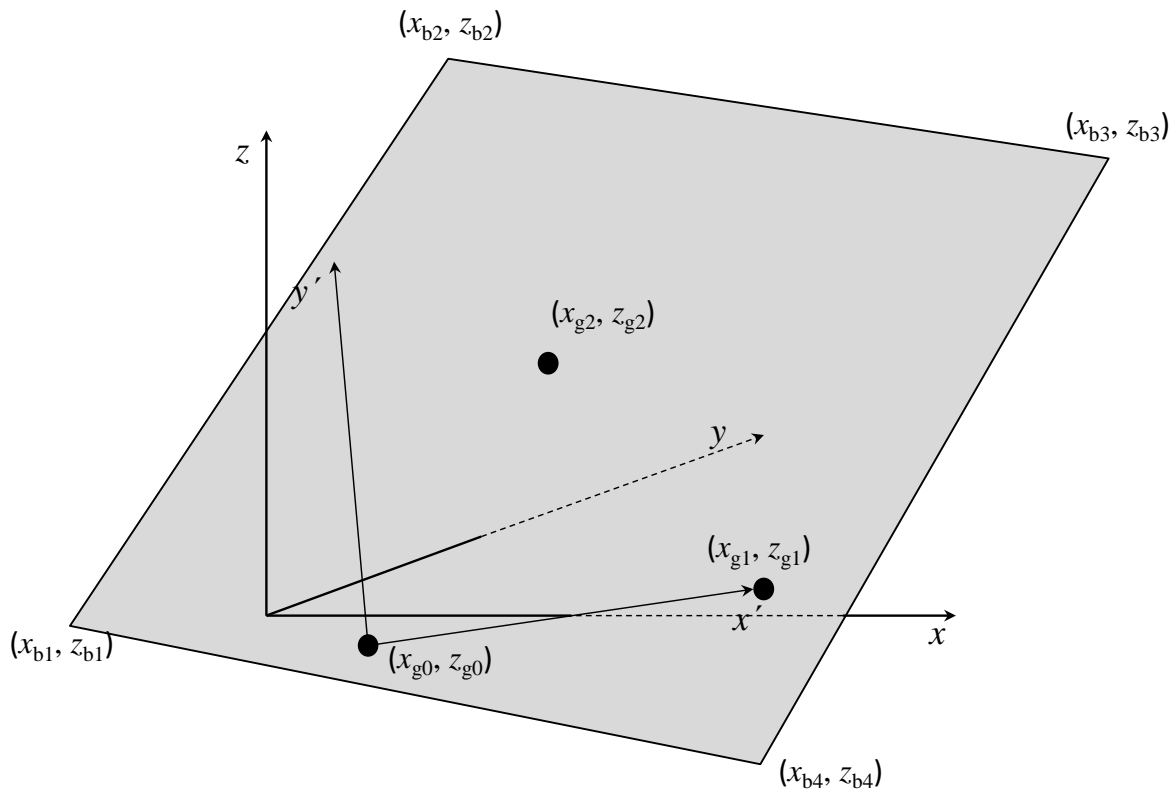


Figure 2.3c. A GPLANE with vertices of its optional bounding polygon specified in global coordinates.

Every GPLANE hosts a gridded CLIST. The origin of the GPLANE coordinate system defines the origin of the grid on which the CLIST is based. The *grid_type* of the grid is -11 (see previous section). The bottom left corner of grid cell (1,1) lies at the origin of the local GPLANE coordinate system, i.e. at the point $(x', y') = (0, 0)$. As is typical of a grid of *grid_type* value -11, CLIST element identification is indicial, and increases with the *ix* and then *iy* cell indices. The directions of increasing *ix* and *iy* index coincide with the directions of increasing x' and y' (which may not accord with that of a right hand coordinate system as mentioned above). CLIST elements are located at the centres of cells which define the grid. Coordinates ascribed to these elements are expressed using the local GPLANE coordinate system (i.e. the $x' y'$ coordinate system). See Figure 2.3d.

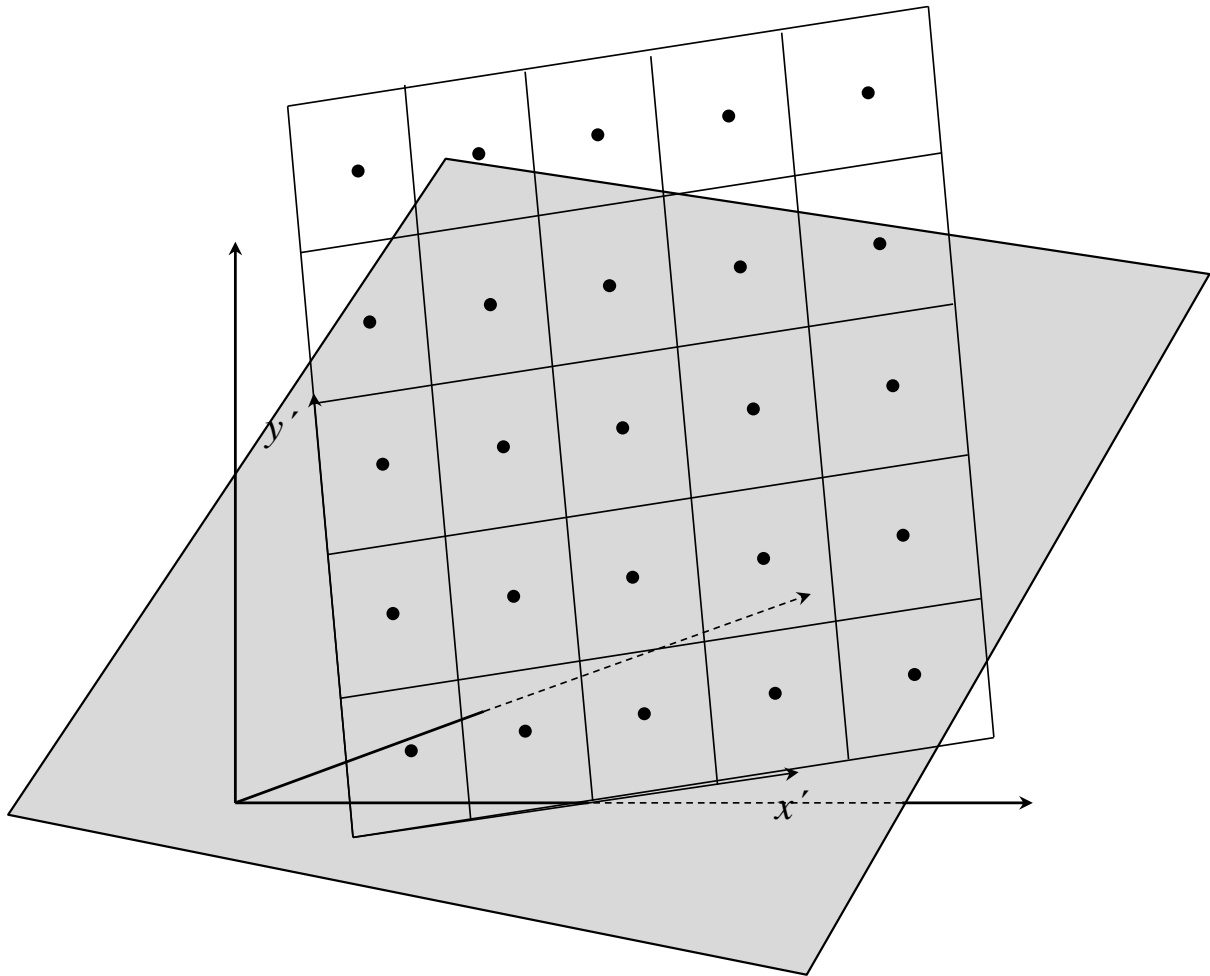


Figure 2.3d. Local GPLANE grid. Grid cell centres which define CLIST element locations are also shown. The CLIST is two-dimensional; CLIST coordinates are specified in the local GPLANE coordinate system. Cells are identified by x' -direction and y' -direction index, i.e. (ix, iy) . The directions of increasing ix and iy coincide with those of increasing x' and y' .

The GPLANE grid is regular; however the cell spacing in the x' direction does not need to equal that in the y' direction. If a GPLANE has a bounding polygon, PLPROC will not object if the GPLANE grid extends beyond the GPLANE's bounds (as occurs in Figure 2.3d).

2.5.2 GPLANES and model parameterization

In parameterization of a model, a GPLANE can be used to represent a planar feature such as a fault. However because its associated CLIST employs a local coordinate system, the reference points for SLISTs and PLISTs that are associated with this CLIST are independent of the model grid into which the fault is inserted. To use a GPLANE in model parameterization, properties must first be assigned to elements of a local GPLANE PLIST. These properties can then be used to inform PLISTS which are dependent on model grid CLISTs, the latter presumably employing real-world coordinates.

To inform an element of a model-based PLIST using a GPLANE-based PLIST, the orthogonal projection of the model-based PLIST element onto the GPLANE's surface is first determined. Within-GPLANE interpolation (using the inverse-power-of-distance method)

then takes place from the GPLANE PLIST to this projection point. The model-based PLIST is then awarded a value based on its orthogonal distance to the GPLANE. Suppose this distance is d . A factor f_1 is first determined using the formula:

$$f_1 = 1 \quad \text{if } (d \leq w/2) \quad (2.5.1a)$$

$$f_1 = e^{\left(\frac{d-w/2}{a}\right)^2} \quad \text{if } (d > w/2) \quad (2.5.1b)$$

where w is a user-specified “constancy width”, and a is a user-specified decay factor; see Figure 2.3e. The GPLANE-informed value of the model PLIST element is then determined in one of four ways. Suppose also that the GPLANE PLIST value interpolated to the point of projection on the GPLANE is v_g . Suppose also that the existing value of the model PLIST element is v_{m1} . The new v_m value for this model PLIST element (i.e. v_{m2}) is calculated using one of the four user-selectable equations:

$$v_{m2} = v_{m1} + f_1 v_g \quad \text{if } \textit{add} \quad (2.5.2a)$$

$$v_{m2} = v_{m1} + f_1 (v_g - v_{m1}) \quad \text{if } \textit{blend} \quad (2.5.2b)$$

$$v_{m2} = \min(v_{m1} + f_1 (v_g - v_{m1}), v_{m1}) \quad \text{if } \textit{blend_never_higher} \quad (2.5.2c)$$

$$v_{m2} = \max(v_{m1} + f_1 (v_g - v_{m1}), v_{m1}) \quad \text{if } \textit{blend_never_lower} \quad (2.5.2d)$$

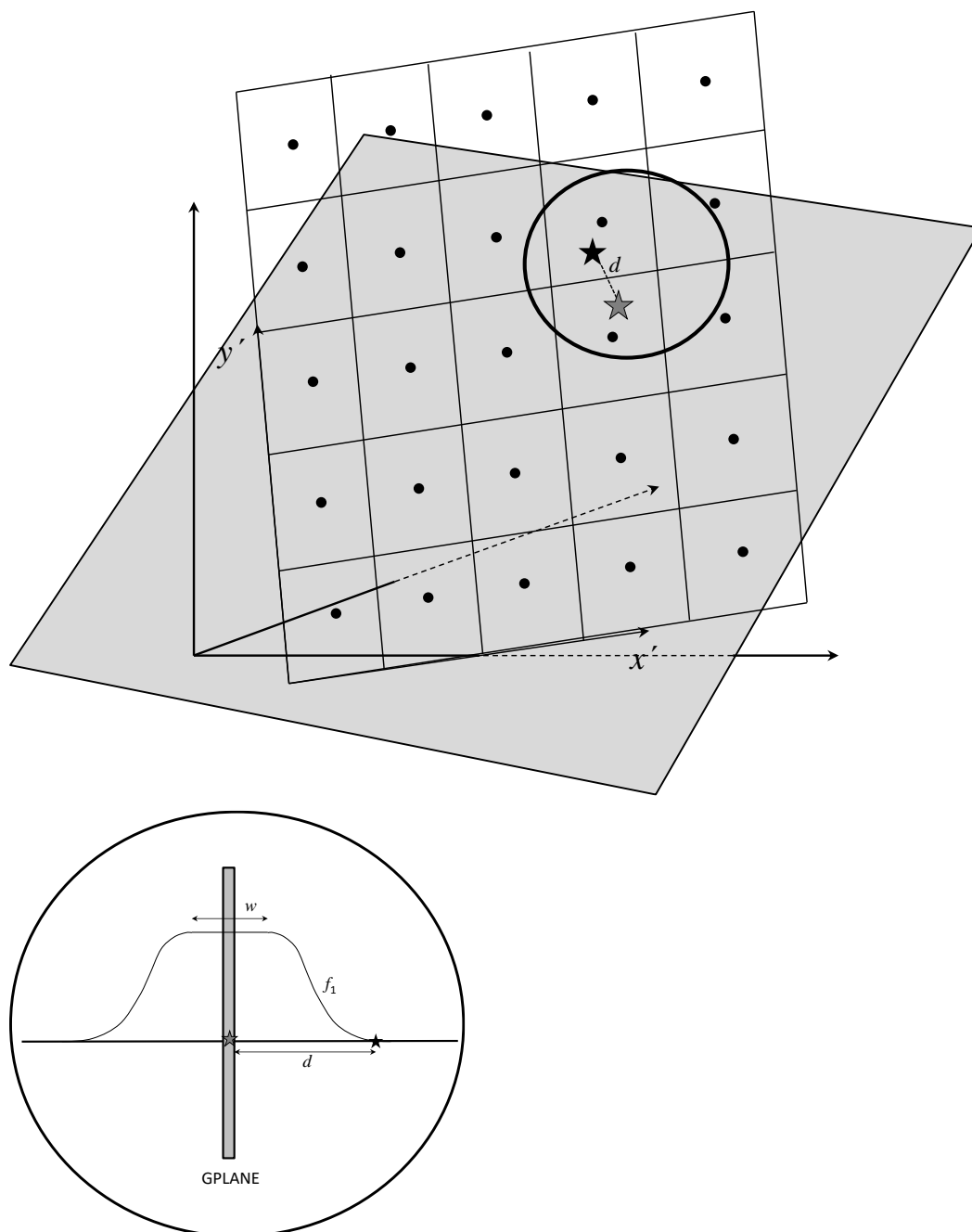


Figure 2.3e. This diagram schematically depicts the way in which a model-based PLIST element is informed by GPLANE-based PLIST elements. First GPLANE PLIST element values are interpolated within the GPLANE to the point of projection (illustrated by the grey star shadow of the black star where the PLIST element is located). The interpolated value at this point then influences model-based PLIST element as a function of the distance of the latter element from the GPLANE surface.

It is apparent that the GPLANE can influence model PLIST elements which are at some distance from the GPLANE itself. Furthermore, this influence decays continuously with distance of these elements from the GPLANE. Such continuity of influence results in continuity of model outputs with respect to parameter values if some of these parameters are coordinates of the three points which define the disposition of the GPLANE. The GPLANE can thus be moved under PEST's influence.

If a GPLANE has lateral boundaries, and if the projection of a model PLIST element onto the GPLANE is outside of these boundaries, then the manner in which GPLANE PLIST elements inform model PLIST elements is altered to accommodate this. As discussed above, the projection of a model PLIST element into the GPLANE is first calculated. If the projection does not lie within GPLANE boundaries, then the distance from this point of projection to the nearest boundary is calculated. GPLANE PLIST element values are interpolated to this boundary point (using inverse-power-of-distance interpolation one again). The f_1 factor featured in equations (2.5.2) is then modified through multiplication by another factor, this being named f_2 . This factor is calculated using equation (2.5.1) where, in this case, d is the distance from the model PLIST projection point outside the boundary to the nearest point on the boundary. Thus the manner in which the GPLANE PLIST informs a model PLIST diminishes not just perpendicular to the GPLANE, but also perpendicular to GPLANE boundaries in the direction of the extension of the GPLANE outside of these boundaries.

2.5.3 Some considerations when using a GPLANE

GPLANE usage in model parameterization can be numerically intensive. It requires that the point of projection of all (selected) model PLIST elements onto the GPLANE be determined, and then that inverse-power-of-distance interpolation take place within the GPLANE to all of these points. If the GPLANE has boundaries, further numerically intensive calculations are required to determine whether a projection point is within the boundary and, if it is not, how close the projection point lies to the nearest boundary. The use of GPLANE boundaries can thus greatly increase the numerical burden of GPLANE usage. (Note that restriction of a GPLANE's influence to only part of a wider model's grid can be implemented in an efficient manner through judicious use of source and target selection equations.)

Within-GPLANE interpolation from GPLANE PLIST elements to model GPLANE projection points can be numerically intensive if a GPLANE grid is dense; the fewer elements that the GPLANE grid possesses, the quicker will be this interpolation.

Note that, optionally, within-GPLANE interpolation can be anisotropic. In this case the "bearing" of the major axis of GPLANE anisotropy (normally the direction of greater continuity of GPLANE properties) is measured with respect to the GPLANE y' axis. The direction of positive rotation is towards the positive direction of the x' axis, regardless of whether this axis points in the conventional positive direction or not (see above).

The element values of a GPLANE PLIST may be estimated by PEST. In this case they represent a regular array of cell-centred pilot points. If the user prefers that interpolation from pilot points takes place through a method other than inverse-power-of-distance, this can be easily accommodated through taking the following steps.

- Notwithstanding the numerical cost, define a dense GPLANE grid; the closer are grid elements together, the less does the interpolation method to points of external PLIST projection on the GPLANE matter.
- Define an independent set of pilot points using a CLIST that employs local GPLANE coordinates. Interpolate from a pilot-points PLIST based on this CLIST to a PLIST based on the primary GPLANE CLIST using any of the two-dimensional pilot point interpolation procedures supported by PLPROC.

If desired, zonation can be introduced to GPLANES using SLISTs whose parent CLIST is the primary GPLANE CLIST. If the user desires, interpolation to off-plane PLIST projection

points can then take place only from GPLANE PLIST elements which are in user-specified zones. As usual, this can be achieved using a selection equation (see the next chapter).

As is described in the documentation of pertinent GPLANE-related functions, GPLANE definition takes place through an external file. A PEST-compatible template can be made of this file. PEST can then be used to adjust, for example, elevations assigned to GPLANE definition points. In this way the location and orientation of the GPLANE can be estimated through the model calibration process.

A user may introduce more than one GPLANE to a model domain. In the pertinent PLPROC script GPLANEs inform model-based PLISTs in a sequential manner. Where a GPLANE crosses another GPLANE, the interaction between the two is easily accommodated using one of the appropriate “blend” options described above.

2.6 MLISTS

An MLIST is a collection of PLISTs or SLISTs which have common CLIST parentage and which share a common root name. This collection is “informal”, in that the PLISTs or SLISTs which are collected into an MLIST may or may not be formally associated with an MLIST when they are created. Furthermore, individual PLISTs or SLISTs which feature in an MLIST can be used independently of other members of the MLIST. The concept of an MLIST simply provides a convenient way of defining, populating and processing a collection of related PLISTs and SLISTs in similar ways, especially where that processing has stochastic overtones.

Any particular MLIST can hold either PLISTs or SLISTs, but not both.

Like other PLPROC entities, an MLIST has a name of 20 characters or less. However the name must include one “*” character. (An MLIST is the only type of PLPROC entity whose name can include the “*” character). In relating this name to the PLISTs or SLISTs which comprise the MLIST, this character is replaced by a sequence of integers. Thus, for example, an MLIST named *pk*dat* may collect PLISTs named *pk10dat*, *pk11dat*, etc. into a single group. The integers which replace the “*” character in referring to PLISTs/SLISTs which comprise the MLIST are provided by the user when the MLIST is defined. The user supplies the upper and lower index; it is presumed that all integers between and including these upper and lower indices denote PLISTs/SLISTs which belong to the MLIST. These integers must be no smaller than 1, and no greater than the maximum number of PLISTs/SLISTs which PLPROC is programmed to store.

Because the link between an MLIST and the PLISTs/SLISTs to which it refers is only loose, this link can be broken by some PLPROC activities. For example a PLIST belonging to an MLIST may be deleted, and then re-defined as linked to another CLIST. This invalidates its membership of that particular MLIST. Hence caution is required in using MLISTs and associated PLISTs. PLPROC will warn the user if an operation is attempted on an MLIST whose linkage to component PLISTs/SLISTs has been corrupted.

PLISTs or SLISTs can be defined, filled and processed in certain ways through a single function call involving an MLIST. Alternatively they can be defined individually through multiple PLPROC function calls first, and collected into an MLIST later for ease of subsequent PLPROC processing using MLIST-specific function calls.

Because of the way in which component PLISTs/SLISTs are named, individual processing of PLISTs/SLISTs comprising an MLIST is easily accomplished using PLPROC looping

functionality; in this case the loop variable simply replaces the “*” character in references to individual PLISTs/SLISTs made through pertinent function calls.

2.7 Scalars

Scalars are independent parameter values. Scalars have no coordinates, and are not referenced by index - only by name. The length of this name must be 20 characters or less.

Scalars can be defined through having their names and values read from a file, or by direct assignment. They can also be defined by assigning the outcome of an equation which cites no PLISTs or SLISTs to a variable that has not yet been defined and is thus obviously a scalar.

Once a scalar has been defined in any of the above ways it cannot be undefined. However it can be assigned a different value.

2.8 Strings

Like scalars, strings have no coordinates. They are referenced only by name. As for scalars, this name must be 20 characters or less in length. The length of a string can be 256 characters or less.

Strings are defined through having their names and values read from a file. Once a string has been defined it cannot be undefined. However it can be assigned a different value.

2.9 Matrices

A MATRIX is a two dimensional array of real numbers. Like other PLPROC entities it has a name which must be 20 characters or less in length.

2.10 Naming Conventions

A user is given considerable freedom in naming the CLISTs, SLISTs, PLISTs, SEGLISTs, scalars and strings that he/she creates through pertinent PLPROC commands. However the following restrictions apply.

- The name of any entity must be 20 characters or less in length.
- An entity name cannot contain a space.
- An entity name cannot include any of the characters listed in Table 2.1.
- An entity name cannot be the same as a PLPROC function name.
- An entity name cannot be the same as a mathematical function name; see Table 3.3.
- The names shown in Table 2.2 are proscribed as they are used for other purposes within PLPROC.

If the user attempts to name an entity in a way that violates any of the above rules, PLPROC will cease execution with an appropriate error message.

Illegal characters
= * + - ^ / ; : \$
() [] { }

Table 2.1 Characters which cannot feature in a PLPROC entity name.

Illegal names
default select
int_id char_id index
skiplines
clist slist plist
file column dimensions valuecolumn namecolumn
x y z
xxxx

Table 2.2 Illegal PLPROC entity names.

3. Equations

3.1 General

Equations can occur in a number of different contexts within a PLPROC script; they can comprise their own command, or can occur as arguments or subarguments to PLPROC functions. PLPROC supports three types of equations - assignment equations, selection equations and string equations. The outcome of an assignment equation is a real (actually double precision) number. The outcome of a selection equation is a logical value (i.e. *TRUE* or *FALSE*). As the name implies, the latter are employed for differentiating those elements of a list that are included in the processing undertaken by a PLPROC assignment equation or function from those elements which are not.

String equations involve only concatenation operations. These are discussed in the final section of this chapter. In the meantime the discussion focusses only on assignment and selection equations.

Entities cited within an assignment or selection equation can include PLISTs, SLISTs, scalars, individual PLIST and SLIST elements, CLIST coordinates, CLIST integer element identifiers and ordinary numbers. Depending on the entities featured within it, the equation may be evaluated only once, or it may be evaluated over all (selected) elements of the lists which are featured in it. For the latter process to work correctly the following fundamental rule must be obeyed in formulating an assignment or selection equation. All SLISTs and PLISTs cited in the same equation must employ the same reference CLIST. This ensures that they employ common indexing and have common coordinates.

Because it is a real number, the outcome of a PLPROC equation cannot be assigned to an SLIST, in spite of the fact that SLISTs can feature in the body of an equation. In the latter case the integers which it encapsulates are converted to double precision numbers before being used in any calculation. If you would like to assign the outcomes of an equation to an SLIST, then you must assign its outcomes to a PLIST first; the *new_slist()* function can then be used to create an SLIST through nearest-integer rounding of corresponding PLIST elements. (Note however that assignment of SLIST values to a PLIST is an easier operation, requiring just a simple assignment statement. Such a statement is in fact an equation, with an SLIST being the only entity featured on its right hand side.)

3.2 Operators and Mathematical Functions

3.2.1 Arithmetic operators

Table 3.1 shows a list of arithmetic operators that can appear in assignment and selection equations. Note that the usual rules of operator precedence apply. If you are in doubt about precedence, use brackets to enforce your own order; inner bracketed expressions are always evaluated first.

Operator symbol	Operation
** or ^	<u>Power</u> . $a^{**}b$ or a^b is interpreted as “a raised to the power b”.

/	<u>Division</u> . a/b is interpreted as “ a divided by b ”.
*	<u>Multiplication</u> . $a*b$ is interpreted as “ a multiplied by b ”.
-	<u>Subtraction</u> . This can be a unary or binary operator. $a-b$ is interpreted as “ a minus b ”; $-a$ is interpreted as “negative a ”.
+	<u>Addition</u> . This can be a unary or binary operator. $a+b$ is interpreted as “ a plus b ”; $+a$ is interpreted as “positive a ”.
()	<u>Brackets</u> . Terms within brackets are evaluated first. For example $5 + 4 * 3$ is evaluated as 17. However $(5 + 4) * 3$ is evaluated as 27.

Table 3.1 Arithmetic operators supported by PLPROC.

3.2.2 Logical operators

Logical operations produce an outcome that is *TRUE* or *FALSE*. Logical operators supported by PLPROC are listed in Table 3.2.

Operator symbol	Operation
.lt. or <	<u>Less than</u> . $a.lt.b$ or $a < b$ is <i>TRUE</i> if a is less than b .
.le. or <=	<u>Less than or equal to</u> . $a.le.b$ or $a <= b$ is <i>TRUE</i> if a is less than or equal to b .
.eq. or ==	<u>Equal to</u> . $a.eq.b$ or $a == b$ is <i>TRUE</i> if a equals b .
.gt. or >	<u>Greater than</u> . $a.gt.b$ or $a > b$ is <i>TRUE</i> if a is greater than b .
.ge. or >=	<u>Greater than or equal to</u> . $a.ge.b$ or $a >= b$ is <i>TRUE</i> if a is greater than or equal to b .
.ne. or !=	<u>Not equal to</u> . $a.ne.b$ or $a != b$ is <i>TRUE</i> if a does not equal b .
.and. or &&	<u>And</u> . $a.and.b$ or $a \&\& b$ is <i>TRUE</i> if both a and b are true; for example $(1.lt.10).and.(6.lt.7)$ is <i>TRUE</i>
.or. or	<u>Or</u> . $a.or.b$ or $a b$ is <i>TRUE</i> if a is <i>TRUE</i> or b is <i>TRUE</i> or both are <i>TRUE</i> ; for example $(1.lt.10).or.(1.gt.0)$ is <i>TRUE</i>
!	<u>Not</u> . $!(a.lt.b)$ is <i>FALSE</i> if a is less than b .

Table 3.2 Logical operators supported by PLPROC.

Note that PLPROC takes special care in comparing numbers for equality. Good numerical programming dictates that comparison of real numbers for equality should be avoided. PLPROC declares two real numbers to be equal if they are separated from each other by less than five quanta of the numerical granularity associated with computer storage of these numbers.

3.2.3 Mathematical functions

PLPROC assignment and selection equations support the following mathematical functions. The outcomes of all of these equations are real numbers (though evaluated and stored internally as double precision numbers).

Function	Definition
abs()	<u>Absolute value</u> . Argument can be any real number.
cos()	<u>Cosine</u> . Argument can be any real number supplied in radians.
acos()	<u>Inverse cosine</u> . Absolute value of argument must be less than or equal to one. Value is returned in radians.
sin()	<u>Sine</u> . Argument can be any real number supplied in radians.
asin()	<u>Inverse sine</u> . Absolute value of argument must be less than or equal to one. Value is returned in radians.
tan()	<u>Tan</u> . Argument can be any real number supplied in radians.
atan()	<u>Inverse tan</u> . Argument can be any real number. Value is returned in radians.
cosh()	<u>Hyperbolic cosine</u> . Argument can be any real number.
sinh()	<u>Hyperbolic sine</u> . Argument can be any real number.
tanh()	<u>Hyperbolic tan</u> . Argument can be any real number.
exp()	<u>Exponential</u> . Argument can be any real number.
log()	<u>Log to base e</u> . Argument must be a positive real number.
log10()	<u>Log to base 10</u> . Argument must be a positive real number.
sqrt()	<u>Square root</u> . Argument must be non-negative.
min(, ,)	<u>Minimum of a series of numbers</u> . Arguments can be any real numbers.
max(, ,)	<u>Maximum of a series of numbers</u> . Arguments can be any real numbers.
mod(,)	<u>Remainder</u> . mod(a,b) is the remainder after a is divided by b .

Table 3.3 Mathematical functions supported by PLPROC.

If an equation asks something impossible of PLPROC (for example if it asks PLPROC to calculate the square root of a negative number) PLPROC will cease its attempts to evaluate the equation, and then terminate execution with an appropriate error message.

3.3 Assignment Equations

The rules governing use of equations in PLPROC scripts will now be demonstrated by example, starting with simple equations and evolving to more complex ones. First we will examine assignment equations; these are equations that calculate values for a sequence of

PLIST elements, or for a scalar. Next the use of selection equations will be demonstrated; these can be used to restrict the operational range of an assignment equation to a subset of PLIST elements.

3.3.1 Equations involving scalars

Figure 3.1 shows a sequence of equations which involve only scalars. (Note that the direct assignment of a value to a scalar can be considered to be an equation.) It is assumed that values have been assigned to scalars *s1* and *s2* through previous PLPROC commands.

```
.  
s3=4.49234  
s4=s3*(s1^2+s2^2)  
s2=s4  
.
```

Figure 3.1 Portion of a PLPROC script showing equations involving scalars.

PLPROC recognizes that a command is in fact an equation by the absence of a function call in the command, and by the fact that the command involves an assignment. Once recognized as such, PLPROC then reads the equation, looking for the variables that it cites. If any of these variables is a list entity, then the outcome of the equation must also be a list entity. If no list entities are cited within the equation, and the variable to which the assignment is made is not a declared PLIST, PLPROC determines that its outcome is a scalar. The outcomes of the three cited equations depicted in Figure 3.1 are the scalars *s3*, *s4* and *s2*.

Scalars to which values are assigned by an equation may actually be created by evaluation of that equation. Thus if the first of the equations in Figure 3.1 constitutes the first mention of *s3*, then the *s3* scalar is brought into existence by the equation. The same applies to *s4* in the second of the equations depicted in Figure 3.1. On the other hand, if the scalar variable to which the outcome of an equation is assigned already exists, then its existing value is replaced by the value calculated by the equation.

3.3.2 Equations involving PLISTs

Now consider the PLPROC script fragment reproduced in Figure 3.2. It is assumed that *p1* and *p2* are PLISTs which were defined in earlier script commands.

```
.  
s1=6322.11  
new_p = p1/p2*4.321 + s1  
p1=new_p+p1  
p3=2.0  
.
```

Figure 3.2 Portion of a PLPROC script showing equations involving scalars and PLISTs.

In the first equation of Figure 3.2, a value is assigned to the scalar *s1*. This scalar is then featured in the right side of the following equation, along with the *p1* and *p2* PLISTs.

As is demonstrated in Figure 3.2 an equation can cite more than one PLIST. However PLPROC will cease execution with an error message if all PLISTs cited in the same equation do not have the same reference CLIST. This ensures that all PLISTs featured in the equation have the same number of elements, possess the same geographical coordinates, and are indexed in the same way.

If an equation cites one or more PLISTs on its right hand side, then the outcome of that equation must be assigned to another PLIST. If the target PLIST already exists, PLPROC checks that it, too, belongs to the same CLIST family as those that are cited on the equation's right side; existing target PLIST element values are overwritten by the outcomes of the equation. If the left hand PLIST does not exist, then it is created by the equation (as is the *new_p* PLIST in Figure 3.2). In creating the new PLIST, PLPROC associates it with the same reference CLIST as that on which PLISTs cited on the right side of the equation depend.

The fourth of the equations shown in Figure 3.2 demonstrates that the same PLIST can occur as both a target PLIST and as an evaluation PLIST. As for any target PLIST, its values are overwritten in evaluating the equation.

When an equation involves PLISTs, it is evaluated on an element-by-element basis for all PLIST elements in succession. Thus, for example, the second of the equations of Figure 3.2 is evaluated N times, where N is the number of elements in each of the PLISTs. On the first occasion of its evaluation, the value of the first element of each cited PLIST is substituted into the equation; the outcome of equation evaluation is then assigned to the first element of the target PLIST. The equation is then re-evaluated for the second element of each PLIST cited on the equation's right side, with the outcome being assigned to the second element of the target PLIST, etc.

The last equation depicted in Figure 3.2 will have a different outcome depending on the nature of the $p3$ entity. If $p3$ has not been previously defined, it will be declared as a scalar and assigned the value shown (i.e. 2.0). However if it has been previously declared as a PLIST, then all elements of $p3$ will be assigned a value of 2.0.

3.3.3 Using SLISTS in an equation

An SLIST can be used on the right side of an equation as if it were a PLIST. The integers which are its element values are converted to real numbers before being used in the equation. Of course it must have the same reference CLIST as any PLISTs, and any other SLISTs, that are featured in the equation.

An SLIST cannot be the target of a PLPROC equation.

3.3.4 Using CLIST properties in an equation

Because of the CLIST parentage of SLISTs and PLISTs, each element of a PLIST and SLIST has an x , y and possibly a z coordinate associated with it. If the parent CLIST employs indexed or integer element identifiers, it also has a unique integer associated with it. For an indexed CLIST this is the actual element index while for a CLIST with integer element identification, it is the value of the integer identifier. All of these can be used in PLPROC equations. Figure 3.3 shows some examples. (Note that CLIST property variables such as these will often find more use in selection equations than in assignment equations; selection equations will be discussed shortly.)

```
.
x0=143232.43
y0=423423.45
plist3 = plist2 + sqrt((x-x0)^2 + (y-y0)^2)
plist4 = plist3 + int_id
.
```

Figure 3.3 Examples of equations involving list properties.

In Figure 3.3 it is assumed that *plist2* is an SLIST or PLIST which has been defined through previous PLPROC commands. This then defines the reference CLIST for the third of the equations featured in Figure 3.3. If *plist3* has already been defined, it must have the same reference CLIST as *plist2*. Alternatively, if it has not been previously defined, it is defined through this equation. *x0* and *y0* are scalars, defined through the first two of the equations featured in Figure 3.3. *x* and *y* featured in the third equation are the coordinates associated with the reference CLIST (and hence dependent PLIST) elements. Like the PLIST elements themselves, these assume different values during each evaluation of the equation as the latter is repeatedly evaluated on an element-by-element basis. As is apparent from this description, the names *x*, *y* and *z* are reserved names; where employed in an equation they specify the coordinates associated with list elements.

The fourth of the equations depicted in Figure 3.3 features the *int_id* CLIST property; *int_id* is also a reserved name with a special meaning. *int_id* is equal to a CLIST's current element index in the equation evaluation sequence if the CLIST uses indexed element identification. Alternatively, it is equal to the user-supplied integer element identifier if the CLIST employs integer element identification. If a CLIST employs character identifiers then use of the *int_id* CLIST property in an equation is illegal as the identifier is not a number.

If it has not already been defined, the *plist4* PLIST appearing on the left of the last of the equations shown in Figure 3.3 is defined through the equation. Repeated evaluation of this equation assigns values to all *plist4* elements. The value of the *int_id* variable on the right hand side of this equation will, of course, be different for each such evaluation.

3.3.5 List-derived scalar variables

Sometimes it may be necessary to include the value of an individual SLIST or PLIST element, or a property of a specific CLIST element, in a PLPROC equation. Figure 3.4 shows some examples. Explanations follow.

```
.
s3 = p1[5]
s3 = p1{5}
s3 = p2{"abc"}
p1 = p2 + p2{"abc"} + p1[5] + p4{3}
p1 = p3 + c13[13].x * c13[13].x + c13[13].y * c13[13].y
p2 = p3 + c15{"abc"}.z + c16{"def"}.z
.
```

Figure 3.4 Equations which use list-derived scalars.

In the first of the equations featured in Figure 3.4 the scalar *s3* is assigned a value equal to that of the element of PLIST *p1* whose index is 5. Square brackets are used to indicate a list element index. Regardless of whether an equation's underlying CLIST employs indexed, integer or character element identification, list elements can be accessed through citation of their indices in this way. As discussed in the previous chapter and in documentation to pertinent PLPROC functions, element indices can begin at 1, or they can begin at a user-specified number, this depending on reference CLIST specifications. If you are unsure of the indexing of a CLIST (which, of course is inherited by its dependent SLISTs and PLISTs) you can gather the information that is required for linking indices to elements through use of the *report_dependent_lists()* function.

In the second of the equations featured in Figure 3.4 reference is again made to a specific element of the *p1* PLIST. However on this occasion curly rather than square brackets are employed. This means that element referencing takes place through the integer identifier or the character identifier of the list's reference CLIST; the chosen method depends, of course, on the element identification type that the parent CLIST employs. If a CLIST's element identification type is "indexed", then an element's integer identifier is defined as being equal to its index. If its element identification type is "integer", then the integer identifier pertaining to each element will have been provided when the CLIST was defined. The same applies to character identifiers. In the third equation of Figure 3.4, the *p2* PLIST is assumed to employ character element identification; an individual element of the *p2* PLIST is referenced through the "abc" identifier.

The fifth and sixth equations of Figure 3.4 demonstrate the only mechanism through which explicit mention can be made of an equation's underlying CLIST in a PLPROC equation. This is done to gain access to a property of a specific CLIST element where that property is to be used as a scalar in the equation; as such, its value does not change during sequential re-evaluation of the equation wherein values are assigned to successive elements of the target PLIST. The specific CLIST element whose property is required can be referenced through element index as in the fifth equation of Figure 3.4, or using its integer or character identifier as in the sixth equation of Figure 3.4. CLIST element properties which can be referenced in this way are the element's *x*, *y* and *z* coordinates.

As was stated above, all PLISTs and SLISTs used in an equation must have the same reference CLIST. This does NOT apply to list-derived scalars. A list-derived scalar can pertain to any PLIST, SLIST or CLIST currently employed by PLPROC, regardless of the equation's underlying CLIST.

3.4 Selection Equations

Selection equations can be used in conjunction with assignment equations. They can also be used in a number of PLPROC functions. In both of these contexts their role is the same: they restrict evaluation of the equation or function to a subset of list elements. In the case of an equation, elements of the target PLIST for which values are not evaluated remain unchanged. It follows that a selection equation can only be employed within an assignment equation when the following conditions are met:

- The target of the equation is a PLIST (and not a scalar);
- The equation's target PLIST has been pre-defined and is therefore not defined through the equation itself.

Some examples of selection equation usage are provided in Figure 3.5.

```
.
p13(select=(x>3000)&&(y<2000)&&(z>6))=p1*p2
p13(select=int_id!=3)=p2
p13(select=(layer==1))=p12
p13(select=(layer==1))=1.0
.
```

Figure 3.5 Use of selection equations in conjunction with assignment equations.

As is apparent from Figure 3.5, if a selection equation is present it must follow the string "select=" on the left hand side of the assignment equation. Both the "select=" string and the

selection equation are enclosed in the one set of brackets. For clarity, it is a good idea to enclose the selection equation itself in brackets. When doing this, however, ensure overall balancing of brackets on the left side of the equation. (PLPROC will soon inform you if brackets are not balanced.)

The same rules apply in formulation of a selection equation as apply in formulation of an assignment equation. A selection equation can cite SLISTs, PLISTs, CLIST properties, scalars and list-derived scalars. However it must not be completely comprised of scalars, for an equation that is comprised entirely of scalars cannot provide a means of discriminating between elements of a PLIST, and hence cannot constitute a selection equation. The following rules also apply.

- The outcome of a selection equation must be either *TRUE* or *FALSE*. If it is *FALSE* for a particular element, the assignment equation is not evaluated for that element and the pertinent target PLIST element remains unchanged.
- The underlying CLIST of a selection equation must not differ from that of the assignment equation whose operation it governs.

3.5 String Equations

The only operation that can be performed by a string equation is concatenation. The only variables that can feature in a string equation are string variables and scalar variables (the latter only after conversion to integers – see below). The outcome of a string equation is assigned to a string variable.

Some examples of string equations are provided in Figure 3.6.

```
.
file_base = "filename"
file_ext = ".dat"
output_file = file_base//"1"//file_ext
scalar = 2.9
output_file=file_base//"$scalar$"//file_ext
alt_output_file=output_file
.
```

Figure 3.6. Examples of string equations.

In the first of the equations featured in Figure 3.6, the string “filename” is assigned to the string variable *file_base*. If this variable already exists then the string which it holds is overwritten by this action; if it does not already exist then it is brought into existence by this assignment operation. In the second equation the new or existing string variable *file_ext* is assigned the string “.dat”.

The third equation featured in Figure 3.6 exemplifies the only operation that can be undertaken between string variables, namely that of concatenation. The string associated with the *output_file* string variable after this operation is “filename1.dat”. If the *output_file* string variable did not exist prior to this operation, it is brought into existence by it.

The fourth operation shown in Figure 3.6 assigns the value 2.9 to the scalar variable *scalar*. This variable then features in the fifth equation surrounded by “\$” characters. As is described elsewhere in this manual, wherever a scalar variable is surrounded by “\$” characters, it is rounded to the nearest integer and then converted to text. As such, it can form part of the concatenation operation performed by the equation in which it is featured, after having been

placed between quotes. The final value of the string variable *output_file* after evaluation of this equation is “filename3.dat”.

In the sixth of the above equations the *alt_output_file* string variable is given the same value as the *output_file* string variable, namely “filename3.dat”.

The following rules apply to string equations.

- The outcome of a string equation must be assigned to an existing string variable or to an uncreated variable; in the latter case a new string variable is created.
- Only string variables, strings enclosed in single or double quotes, and scalar variables surrounded by “\$” characters can appear in string equations; in the latter case quotes must surround the \$-delimited scalar variable.
- Only the // operation (signifying concatenation) can appear in the string equation.
- A string surrounded by “\$” characters must denote a scalar variable. If it is desired that a list-derived scalar be converted to a string, then it should first be assigned to a normal scalar variable in an assignment equation before the latter is placed between “\$” characters.
- A number cannot be placed between “\$” characters. For example “\$60.2\$” is not allowed. Use the string “60” instead.
- In concatenation operations, leading and trailing blanks in strings are removed.
- A string enclosed in quotes cannot be blank.

If any of the above rules are violated, PLPROC will cease execution with an appropriate error message.

4. Functions

4.1 General

Specifications and usage protocols for all PLPROC functions are provided in the second part of this manual. The present chapter outlines some general features of their use.

4.2 Function Protocol

Figure 4.1 provides some examples of PLPROC function calls. Their protocols have much in common. However, there are some differences too, these arising from differences in the nature of the tasks that they perform.

```

c11 = read_list_file(skiplines=1,dimensions=3,      &
                    plist='p1';column=5,           &
                    plist='p2';column=6,           &
                    slist='s1';column=7,           &
                    slist='s2';column=8,           &
                    slist='s3';column=9,           &
                    id_type='integer',file='list.dat')

c11.report_dependent_lists(file='templa.dat')

calc_kriging_factors_2d(target_clist=modflow_grid;select=(ib1==1),      &
                        source_clist=cl_pp;select=z1==1,                &
                        file='fac1.dat';form='text',                    &
                        variogram='exponential',                        &
                        a=45000,                                         &
                        kriging='simple',                                 &
                        mean=1,nugget=0.2,sill=0.1,                     &
                        anis_bearing=30,anis_ratio=5.0,                 &
                        min_points=1,max_points=12,search_radius=1e20)

kx=new_plist(reference_clist=musg_grid,value=1.01e30)

sm1=s1.krige_using_file_2d(file='fac1.dat';form='text',      &
                           transform='log')

pp2(select=(z.lt.0))=p1.assign_if_colocated_2d,      &
    (acceptance_threshold=1.0,                        &
    select=(y==300))

```

Figure 4.1 Examples of PLPROC function calls.

As Figure 4.1 demonstrates, a function is invoked by writing its name, followed by an opening round bracket, followed by function arguments, followed by a closing round bracket. As is usual PLPROC protocol, continuation of a function call statement to the next line is enabled through the “&” character. This gives the user the ability to format function calls (especially complicated function calls) in a way that helps a reader to understand them.

Some functions assign values to a target list. In that case the name of the target entity is placed on the left side of the command which invokes the function; the name of the function then follows after an “=” symbol. Where a function makes no such assignment, the name of the function leads the PLPROC command which invokes it.

Some functions are formulated as “object functions” using a similar protocol to that employed in object-oriented languages such as C++ and Python. See the fifth example in Figure 4.1. In this case the name of the function follows the name of the entity on which it operates (normally a PLIST), with a dot separating the two.

4.3 Arguments and Subarguments

As stated above, function arguments are enclosed in round brackets following the name of the function itself. The protocol used for assignment of values to function arguments is as follows:

```
arg_name = arg_value
```

where *arg_name* is the name of a function argument and *arg_value* is its value. This value can be real, integer or character, depending on the nature of the function argument. Sometimes abbreviations can be used in the argument name; where this is permissible it is indicated in the documentation of individual functions provided in the second part of this manual. Function arguments can be supplied in any order. Non-essential arguments can be omitted if desired, in which case default values for these arguments are provided by PLPROC itself. Within an argument list, argument value assignments are demarcated by commas.

Some arguments require values for one or more subarguments, as well as for the main argument itself. Where subarguments are employed, some subarguments may be mandatory while others may be optional. The subargument protocol is shown below:

```
arg_name = arg_value; subarg_name=subarg_value; subarg_name=subarg_value,
```

As is apparent, subarguments of the same argument are separated by semicolons. A comma after the final subargument then separates the total argument-plus-subargument list from that of the ensuing argument. For most functions, subarguments within an argument can be listed in any order. (Exceptions are the *find_cells_in_lists()* and *replace_cells_in_lists()* functions.) See Figure 4.1 for some examples of subargument usage.

When supplying a value for a function argument or subargument which requires a text string, the user may optionally surround this string in single or double quotes. While not mandatory, this is recommended. The only occasion on which the use of quotes is mandatory is if the argument is the name of a file which contains a blank.

4.4 Selection Equations

Some PLPROC function arguments and subarguments require a selection equation as their value. The syntax of a selection equation must follow the same rules as those described previously for equations in general. In all cases the selection equation text must follow a “select=” string, for in all cases “select” is the pertinent argument or subargument name. Naturally, the CLIST implied by SLISTs, PLISTs and CLIST properties cited in a selection equation must comply with the reference CLIST employed by other function arguments. Evaluation of the function then takes place only for those list elements for which the selection equation is evaluated as *TRUE*.

The sixth command in Figure 4.1 illustrates use of a selection equation on both sides of an equality sign. The *assign_if_collated_2d()* function is one of a group of functions that calculate values for the elements of one PLIST (i.e. a target PLIST) from those of another PLIST (i.e. a source PLIST), where the target and source PLISTs have different reference CLISTs. In a case such as this the selection equation on the right governs selection of source

PLIST elements whereas the selection equation on the left governs selection of target PLIST elements. Naturally, calculations undertaken by the different selection equations must be compatible with the reference CLISTs pertinent to the different sides of the overall assignment equation.

Where selection equations are employed to restrict the elements for which target PLISTs receive values, non-selected target PLIST elements retain their existing values.

4.5 Using Variables as Function Arguments

Function arguments and subarguments can be numeric or character; their value is supplied following an “=” symbol. Instead of providing a number, a scalar variable can be provided. Instead of providing a character string, a string variable can be provided. PLPROC makes the conversion itself before evaluating the function.

In the following example a function is called. The values of scalar and string variables are then assigned through a set of equations. (Note that scalar and string variable values can also be read from a scalar file; the former are thus easily adjusted using PEST.) The same function is then called using scalar and string variables for some function arguments. Function outcomes are identical.

```
calc_kriging_factors_2d(target_clist=modflow_grid;select=(ib1==1), &
                        source_clist=cl_pp;select=z1==1, &
                        file='fac1.dat';form='text', &
                        variogram='exponential', &
                        a=45000, &
                        kriging='simple', &
                        mean=1,nugget=0.2,sill=0.1, &
                        anis_bearing=30,anis_ratio=5.0, &
                        min_points=1,max_points=12,search_radius=1e20)

# Values are assigned to some scalar and string variables.

one = 1.0
twelve = 12.0
range=45000.0
factorfile='fac1.dat'

# These variables are now used as function arguments.

calc_kriging_factors_2d(target_clist=modflow_grid;select=(ib1==one), &
                        source_clist=cl_pp;select=z1==1, &
                        file=factorfile;form='text', &
                        variogram='exponential', &
                        a=range, &
                        kriging='simple', &
                        mean=1,nugget=0.2,sill=0.1, &
                        anis_bearing=30,anis_ratio=5.0, &
                        min_points=1,max_points=$twelve$,search_radius=1e20)
```

Figure 4.2 A PLPROC function call in which scalar and string variables are used as arguments.

When using a scalar variable as a function arguments or subargument, the name of the variable can optionally be surrounded by “\$” characters. In this case the value of the scalar variable is converted to the nearest integer before being provided to the function. Integer conversion is essential for function arguments and subarguments which must be integer. “\$” symbols cannot enclose a non-scalar variable.

5. Writing Model Input Files

5.1 General

Like PEST, PLPROC uses template files for writing model input files. However considerable modifications to the PEST protocol have been introduced in order to handle parameter lists. These modifications accommodate the indexed nature of these lists, as well as the possibly large volume of data that may need to be written to a model input file.

5.2 Embedded Functions

A template file is usually made from a model input file by manual editing of the latter. In many cases comparatively minor changes are required. Hence when PLPROC or PEST processes a template file in order to thereby write a model input file, most of the contents of the template file are transferred directly to the model input file. In undertaking the editing necessary to create a template file from a model input file, it is therefore the user's responsibility to ensure that the integrity of the template file as an only slightly altered replica of a model input file is safeguarded.

PEST template files rely exclusively on the concept of "parameter spaces" for transferring parameter values to model input files. A parameter space is a sequence of characters that is introduced to a template file to mark the place where the value of a parameter is to be written. The beginning and end of this character sequence is denoted by a special character, named a "parameter delimiter", that occurs nowhere else in the file. Between these delimiters is the name of a parameter. Before it transfers the line of the template file on which the parameter space resides to the model input file which it is writing, PEST (and PLPROC) replaces characters between and including the parameter delimiter with the current value of the named parameter. In doing so the number representing the parameter's current value is written with as much precision as the parameter space will allow, up to a limit of seven significant figures - this limit being in accordance with storage of single precision numbers in a computer's memory. Note that, with the inclusion of a possible negative sign, decimal point and exponent, up to 13 characters may be required for the writing of a number with this level of precision, this depending on the size of the number. (Note also that while PEST provides an option for the use of as many significant figures as double precision storage allows, the present version of PLPROC does not.)

PLPROC supplements parameter space functionality with that of embedded functions. One such function can be used to define the parameter delimiter. The delimiter can then be re-defined within the same template file using subsequent embedded function calls. In fact (unless the parameter delimiter is defined using the older PEST protocol at the top of the template file - see below) PLPROC will not recognise parameter spaces until it encounters the embedded function which defines the parameter delimiter. Hence where parameter spaces are employed for parameter value transfer, it is a good idea to place the parameter delimiter definition function near the start of a template file.

Other embedded functions perform far more complex tasks. These include the transfer of lists or sublists of parameter values to a model input file in a variety of formats, including as arrays or as tabulated lists of numbers. The ordering of elements within these lists can be determined either by PLPROC, or by data that already resides in the model input file.

An embedded function is recognized by the character string “\$#p” in a template file. If, on reading a template file, PLPROC encounters a line which begins with this string it will not transfer the contents of that line to a model input file. Instead it will remove the string and attempt to interpret the remainder of the line as a PLPROC function.

As for other PLPROC commands, continuation characters can be used to spread a command over many lines for easier writing and reading of that command by the user. Consider, for example, the fragment of a template file depicted in Figure 5.1.

```
.
.
$#p # Following is a PLPROC embedded function
$#p  replace_column(plist=pp1;col=3,    &
$#p                      plist=p2;col=4,    &
$#p                      startindex=1,endindex=2)
.
.
```

Figure 5.1 Part of a PLPROC template file showing an embedded function.

Each line of the template file fragment depicted in Figure 5.1 begins with the character string “\$#p”. Hence PLPROC knows that it must not transfer these lines to the model input file which it is writing on the basis of the template file. Instead, once it has stripped these leading characters from each of these lines, it must interpret the rest of each line as if it were part of a standard PLPROC script.

On the first of the above lines PLPROC immediately encounters a “#” character following the “\$#p” character sequence. In accordance with standard PLPROC script protocol, this tells PLPROC to ignore the rest of the line.

On reading the next line of the above template file fragment, PLPROC encounters the leading “\$#p” character sequence again; it therefore knows that this line too is to be treated as a command. Following the leading “\$#p” character string it finds the *replace_column()* function. However at the end of the line it encounters the “&” continuation character. This informs PLPROC that the command is not finished, and that it must read the next line of the template file for more of the command. Implied in this instruction is the necessity for the next line to begin with the “\$#p” character string. So PLPROC proceeds to read the next line of the template file in expectation of finding this character string, followed by more of the embedded command. (If the leading “\$#p” string is absent, PLPROC will cease execution with an appropriate error message.) It then strips away these leading characters and (unless it encounters a “#” character) appends the remainder of the line to the previous partially-read command. Because, in the above example, this line also is terminated with a “&” character, this process continues until all lines of the function have been read. The embedded function is then implemented.

Figure 5.2 shows another fragment of a template file, this fragment comprised of just one line. The function on this line informs PLPROC that, until further notice, the parameter delimiter is the “\$” character. Note how a comment has been added to the line after the function. Note also that the parameter delimiter can be changed later using another *parameter_delimiter()* function.

```
$#p parameter_delimiter = "$" # Definition of parameter delimiter
```

Figure 5.2. Part of PLPROC template file showing a call to the *parameter_delimiter()* function.

5.3 Transferring Scalars and PLIST Elements

5.3.1 User-supplied element referencing

Only scalars and PLISTs can be written to a model input file. The values of PLIST elements can be transferred on an element-by-element basis, or in a more efficient way using functions that enable entire PLISTs, or sublists of PLISTs, to be transferred using a single command. Element-by-element transfer is considered first.

The protocol for transferring the values of scalars and individual PLIST elements to a model input file is similar to that used by PEST, relying on the concept of the parameter space. An example is provided in Figure 5.3.

```
.
.
$#p parameter_delimiter = "#"
...other data .... #scalar1      # ...other data...
...other data .... #scalar2      # ...other data...
...other data .... #plist1[12]   # ...other data...
...other data .... #plist1{1342} # ...other data...
...other data .... #plist2{abcd} # ...other data...
.
.
```

Figure 5.3 Part of a PLPROC template file.

The first statement shown in the template file fragment of Figure 5.3 defines the parameter delimiter as “#”. (Note that while it is not necessary to surround the delimiter by quotes when defining it using the *parameter_delimiter()* function, it is a good idea to use quotes if defining the “#” character as the parameter delimiter. Recall that the “#” character has another role, namely that of defining the start of comment lines and strings within PLPROC scripts and embedded function lines. Surrounding it by quotes when defining it as the parameter delimiter prevents PLPROC from removing the end of the line, thereby leaving the parameter delimiter undefined. The same applies when defining the “&” character as the parameter delimiter, as this character can also serve as a line continuation indicator.)

Parameter spaces residing on the following two lines of Figure 5.3 inform PLPROC that the values of two scalars named *scalar1* and *scalar2* must be written to the model input file corresponding to the above template file. As is the usual practice, these numbers are written to the space between and including the first and last parameter delimiters surrounding the string which provides the name of the entity whose value is to be recorded. If the number representing the entity’s value does not completely fill the space, it is right-justified.

The next line instructs PLPROC to record the current value of element 12 of PLIST *plist1* on the model input file. The square brackets following the name of the PLIST denote an element index. If you are unsure of a PLIST’s indicial details, use the *report_dependent_lists()* function to gain the necessary information.

The following line requests that the value of another element of the *plist1* PLIST be recorded. However on this occasion the element is referenced by its element identifier, this being

indicated by the use of curly rather than square brackets. If the PLIST employs indexed element identification, then the “1342” between the brackets refers to the element’s actual index. If integer element identification is employed, then “1342” denotes the user-supplied integer that is associated with the element whose value must be recorded. On the other hand, if the *plist1* PLIST employs character element identification, then “1342” is treated as a character string, and the value of the element whose identifier comprises this string is written to the model input file. The *plist2* PLIST which features on the following line of the template file fragment of Figure 5.3 employs the character element identification protocol. The parameter space appearing on this line instructs PLPROC to report the value of the element with character identifier “abcd” to the model input file. (Note that quotes could optionally be placed around the “abcd” string within the curly brackets.)

5.3.2 File-based element referencing

There are times when the PLIST element whose value must be written to a model input file is identified by information recorded in the model input file itself, and therefore in the template of this file (which is largely a copy of the model input file). Where data must be written to a model input file in columns, and where one of these columns contains the element identifier or index for data which must be recorded in another column, this is easily accommodated. See the next example.

```
.
.
$#p parameter_delimiter = ~
12      456.32      ~ p1[c=1]      ~
13      494.31      ~ p1[c=1]      ~
18      489.32      ~ p1[c=1]      ~
19      453.32      ~ p1[p=1:9]    ~
.
.
```

Figure 5.4 Part of a PLPROC template file demonstrating file-based PLIST referencing.

In the template file fragment depicted in Figure 5.4, values of elements of the *p1* PLIST must be written to the third column of a model input file. The square brackets following the name of this PLIST indicate that the bracket-enclosed string references PLIST elements by index rather than by identifier. However the index must be read from another part of the same line of the template file (which of course is transferred directly to the model input file). The “[c=1]” string indicates that the index of the element whose value is to be written to the model input file should be read from column 1 on the same line. Hence the values of *p1* elements whose indices are 12, 13 and 18 must be recorded in the third column of the first three lines of the table represented in Figure 5.4. For the final table entry, square brackets once again indicate that the PLIST element is referenced by index. However in this case the element’s index should be read from between (and including) character positions 1 and 9 on the same line of the template file (this corresponding to the first column again); the element index in this case is 19.

Figure 5.5 is similar to Figure 5.4; however in this case PLIST elements are referenced by identifier rather than by index, this being indicated by the use of curly brackets. Once again, the integer or character string that references the element is sought from the model input file (and hence template file) itself, from a different place on the same line.

```

.
.
$#p parameter_delimiter = ~
12      456.32      ~ p1{c=1}      ~
13      494.31      ~ p1{c=1}      ~
abc      489.32      ~ p2{c=1}      ~
def      453.32      ~ p2{p=1:9}    ~
.
.

```

Figure 5.5 Part of a PLPROC template file demonstrating file-based PLIST referencing.

In the example of Figure 5.5 the *p1* PLIST may employ indexed or integer identification. In either case the integer that is used to identify the element whose value must be written to the model input file must be read from column 1 (on the same line as that on which the element is cited for output). The *p2* PLIST obviously employs character element identification. The “{c=1}” string on the second last line of the illustrated table indicates that this identifier must be read from column 1; it is thus “abc”. The “{p=1:9}” string on the last line of the table indicates that the identifier must be read from between character positions 1 and 9 (inclusive), this coinciding with column 1 in the present case; the character identifier of the PLIST element requiring output is thus “def”.

5.4 Writing Entire Lists or Sublists

The *write_in_sequence()* and *replace_column()* functions can be included as embedded functions in a PLPROC template file. Details of their usage are provided in the second part of this manual. Meanwhile two examples of their use will be briefly discussed.

```

.
.
$#p hk.write_in_sequence(new_line_interval=143,number_per_line=8)
.
.

```

Figure 5.6 Part of a PLPROC template file showing an embedded *write_in_sequence()* function.

Figure 5.6 illustrates a simple application of the *write_in_sequence()* function. In this case PLPROC is instructed to write all of the elements of the *hk* PLIST to a model input file, starting at the line on which the embedded function appears. Element values must be recorded eight to a line. However a new line must be commenced after every 143 element values have been recorded. This output protocol can be useful where a PLIST holds an array of values corresponding, for example, to one layer of a structured grid. In this case the structured grid has 143 columns; the input file protocol for the structured grid model is such that element value wrapping within a single row of input is allowed.

In the example of Figure 5.7 the *replace_column()* function directs PLPROC to write elements 1 to 100 of the *pp1* PLIST, and elements 1 to 100 of the *pp2* PLIST, to columns 3 and 4 of the table that immediately follows this function. In doing so it replaces existing numbers within the table by new numbers representing PLIST element values.


```

.
.
$#p  replace_column(plist=pp1;col=3,          &
$#p                plist=pp2;col=4,          &
$#p                startindex=1,endindex=100)
2      11      23.4000      1.00000E-03      23.40000
2      12      13.4000      2.00000E-03      13.40000
3      13      3.40000     3.00000E-03      3.400000
4      14      83.4000     4.00000E-03      83.40000
5      15      23.4000     5.00000E-03      23.40000
.
.

```

Figure 5.7 Part of a PLPROC template file showing an embedded *replace_column()* function.

5.5 Parameter Delimiter - Alternative Definition

As was described above, the character employed as the parameter delimiter can be defined using the *parameter_delimiter()* embedded function. For convenience, it can also be defined using the traditional template file protocol employed by PEST. See Figure 5.8.

```

ptf $
.
.
12      456.32      $ p1{c=1}      $
13      494.31      $ p1{c=1}      $
abd      489.32      $ p2{c=1}      $
def      453.32      $ p2{p=1:9}    $
.
.

```

Figure 5.8 Traditional parameter delimiter definition.

The traditional PEST template file header is comprised of the string “ptf” followed by a space, followed by a single character denoting the parameter delimiter. If the first line of a PLPROC template file follows this protocol exactly, then PLPROC will assume that it is a traditional template file header. PLPROC will define the parameter delimiter accordingly, and will not transfer the first line of the template file to the model input file. However if any breaches to traditional template file header protocol are encountered in this header line, PLPROC will assume that the first line of a template file is not in fact a traditional header. It will then transfer the contents of this line to the model input file under the assumption that is a required part of that file.

5.6 A Cautionary Note

As is explained at length in the PEST manual, PEST computes the derivatives of model outputs with respect to adjustable parameters using the method of finite parameter differences. In implementing this methodology PEST makes a small alteration to each parameter that it estimates, runs the model, reads the model-generated counterparts to observations, and divides model output increments incurred in this fashion by the parameter increment. Variations of this theme include the use of three point and five point finite difference stencils, as well as the use of various methods of interpolating between the three or five points.

When PEST writes a number to a model input file, it adjusts its internal representation of that number so that it accords with the number which it has just written. Thus no loss in the integrity of numerical derivatives is suffered if this space is insufficiently wide to hold all of the significant figures that are required to represent this number with the same precision as that employed by single (or double) precision computer arithmetic. In addition to this, PEST also goes to great lengths to squeeze as many significant figures as it can into a parameter space of limited width if this is required, dispensing with exponents and unneeded zeroes if this can be done while maintaining the integrity of the number.

Where PLPROC is being used as a model preprocessor during the model calibration process, one or a number of “model input files” as far as PEST is concerned will actually be PLPROC input files. PLPROC processes the numbers that PEST writes to these input files in accordance with commands provided in its input script file, and then writes another set of files - files which are actually read by the numerical model. While PLPROC, like PEST, writes numbers to these files using maximum possible precision, it does not adjust its internal representation of numbers that it writes to accord with their external representations. Hence any loss of precision incurred in writing these numbers cannot be recovered. In fact, even if PLPROC did attempt to recover lost precision in this way this would achieve no purpose, for there is no way that the new value of an adjusted number could be transmitted back to PEST.

With this in mind, the user should take all measures possible to ensure that the numbers which PLPROC writes to model input files are written with maximum possible precision. This can be facilitated by making parameter spaces on template files as wide as possible, while still adhering to the formatting conventions that are required of these files by the model. The user should always bear in mind the fact that any loss of significance incurred in transferring numbers between PLPROC and the model, and hence between PEST and the model, could result in a serious deterioration in PEST’s performance.

Other precautions that should be taken to ensure the integrity of finite-difference derivatives calculation are discussed in the PEST manual. In particular, model solver convergence criteria should be set tight; and the model should write numbers to its own output files with maximum precision. See the PEST manual for further details.

5.7 Other Ways to Write Model Input Files

PLPROC provides a number of alternative ways to write PLISTS, or parts of PLISTS, to model input files that do not involve the use of template files. For example, depending on its input file expectations, a model may be able to read PLIST values recorded in tabular format using function *write_data_in_columns()*.

More specialized writing of model input files is performed by function *replace_cells_in_lists()*. This function reads one model input file and writes another in which the values of certain model inputs that are cited in tables on that model input file are replaced by the values ascribed to elements of one or more PLISTS. PLIST elements can be denoted by model cell indices within these tables. The linkage between the parent CLIST of the exported PLISTS and the model grid is made through function arguments.

6. Loops

6.1.1 General

PLPROC scripts can implement primitive looping functionality. This allows certain sequences of commands to be implemented repetitively, with different values being employed for *LIST*s, filenames and other variables during each iteration of the loop. This provides the basis for implementation of stochastic functionality through which different realizations of random-valued *LIST*'s can be read and/or generated, and then used to populate model input files.

As well as being undertaken by PLPROC itself, looping can be implemented by a program which repetitively calls PLPROC. In this case the iteration counter can be supplied either through a scalar file which is written by the calling program and which PLPROC reads, or through the command line which the calling program uses to run PLPROC.

Aspects of PLPROC functionality which assist in looping will now be described. Some of these PLPROC specifications can be useful in other PLPROC implementation contexts as well. Some of these aspects of PLPROC functionality are described elsewhere in this manual, but are grouped together in the present section to demonstrate their use in internal or external PLPROC looping.

6.1.2 String Equations

String equations allow concatenation of string fragments. Iteration-specific filenames can be formed where string equations are combined with *\$scalar\$* substitution; *\$scalar\$* substitution is explained below. In the meantime, some examples of string equations are provided in the following figures.

```
root='temp'
number='23'
extension='.dat'
afile=root//number/extension
```

Figure 6.1 Examples of string equations.

In Figure 6.1 the *afile* string variable is assigned a value of “temp23.dat”. The same occurs in Figure 6.2; however in this case the string “23” is not housed in a string variable before being concatenated with other strings to comprise the value of the *afile* string variable.

```
root='temp'
extension='.dat'
afile=root//'23'/extension
```

Figure 6.2 More examples of string equations.

6.1.3 Function Arguments as Scalar and String Variables

Any argument or subargument of a PLPROC function that must be supplied to PLPROC as a number can also be supplied as a scalar variable. By way of example, the function calls demonstrated in Figures 6.3 and 6.4 are equivalent.

```

calc_kriging_factors_2d(target_clist=modflow_grid;select=(ib1==1),    &
                        source_clist=c1_pp;select=z1==1,              &
                        file='fac1.dat';form='text',                  &
                        variogram='exponential',                      &
                        a=4500.0,                                     &
                        kriging='simple',                              &
                        nugget=0.2,sill=0.1,                         &
                        anis_bearing=30,anis_ratio=5.0,              &
                        min_points=1,max_points=20,                  &
                        search_radius=1e20)

```

Figure 6.3 An example of a call to the *calc_kriging_factors_2d()* function.

```

range=4500.0
calc_kriging_factors_2d(target_clist=modflow_grid;select=(ib1==1),    &
                        source_clist=c1_pp;select=z1==1,              &
                        file='fac1.dat';form='text',                  &
                        variogram='exponential',                      &
                        a=range,                                     &
                        kriging='simple',                              &
                        nugget=0.2,sill=0.1,                         &
                        anis_bearing=30,anis_ratio=5.0,              &
                        min_points=1,max_points=20,                  &
                        search_radius=1e20)

```

Figure 6.4 The *range* scalar variable is used to provide a value for the *a* subroutine argument.

In the second of the above script fragments (i.e. Figure 6.4) the *calc_kriging_factors_2d()* function uses the scalar variable *range* to inform the *a* argument; this is highlighted in Figure 6.4. A value for *range* can be provided by direct assignment through a PLPROC equation in the manner indicated in Figure 6.4. Alternatively, it can be read from a scalar file, or provided through the PLPROC command line. If provided through a scalar file, this file can be written by an external program (such as PEST) which then runs PLPROC to undertake spatial interpolation.

Where a PLPROC function requires an integer as the value for one of its arguments or subarguments, then a scalar value must be converted to an integer before being provided to the function. This can be done through \$scalar\$ substitution, the details of which are discussed below. Through this mechanism the *min_points* and *max_points* arguments of the *calc_kriging_factors_2d()* function are assigned values of 1 and 20 respectively in Figure 6.5. The scalar variables *minp* and *maxp* are first assigned values of 1.0 and 20.0 (scalar variables can only hold real numbers). Integer conversion takes place through \$scalar\$ substitution.

```

minp=1.0
maxp=20.0
range=4500.0
calc_kriging_factors_2d(target_clist=modflow_grid;select=(ib1==1),    &
                        source_clist=cl_pp;select=z1==1,              &
                        file='fac1.dat';form='text',                  &
                        variogram='exponential',                      &
                        a=range,                                       &
                        kriging='simple',                               &
                        nugget=0.2,sill=0.1,                          &
                        anis_bearing=30,anis_ratio=5.0,               &
                        min_points=$minp$,max_points=$maxp$,          &
                        search_radius=1e20)

```

Figure 6.5 Integer values are supplied to the *min_points* and *max_points* subroutine arguments through \$scalar\$ substitution.

String variables can also be employed to supply values for function arguments. As for scalar variables, values of string variables can be provided through direct assignment within the PLPROC script file. Alternatively they can be read from a scalar file, or provided through the PLPROC command line. This allows external programs to alter subroutine arguments on each occasion that they run PLPROC.

The following fragment of a PLPROC script achieves the same as the previous fragments. However the *variogram* and *kriging* arguments are provided through string variables *vartype* and *krigtype*.

```

vartype='exponential'
krigtype='simple'
minp=1.0
maxp=20.0
range=4500.0
calc_kriging_factors_2d(target_clist=modflow_grid;select=(ib1==1),    &
                        source_clist=cl_pp;select=z1==1,              &
                        file='fac1.dat';form='text',                  &
                        variogram=vartype,                             &
                        a=range,                                       &
                        kriging=krigtype,                             &
                        nugget=0.2,sill=0.1,                          &
                        anis_bearing=30,anis_ratio=5.0,               &
                        min_points=$minp$,max_points=$maxp$,          &
                        search_radius=1e20)

```

Figure 6.6 The string variables *vartype* and *krigtype* are used to provide values for the *variogram* and *kriging* function arguments.

In the following example, the value of the *file* argument is provided as a string variable. *afile* is set to “Temp23.dat” and then provided as the value of the *file* argument. Note that PLPROC preserves the case of string variables. It also preserves whitespace between characters that occur within a string. However it does not preserve whitespace at the beginning and end of strings.

```

root='Temp'
extension='.dat'
afile=root/'23'/extension
calc_kriging_factors_2d(target_clist=modflow_grid;select=(ib1==1),    &
                        source_clist=cl_pp;select=z1==1,              &
                        file=afile;form='text',                      &
                        variogram='exponential',                     &
                        a=45000.0,                                   &
                        kriging='simple',                             &
                        nugget=0.2,sill=0.1,                        &
                        anis_bearing=30,anis_ratio=5.0,              &
                        min_points=1,max_points=20,                 &
                        search_radius=1e20)

```

Figure 6.7 The string variable *afile* is employed to assign a value of “Temp23.dat” to the *file* argument of the *calc_kriging_factors_2d()* function.

6.1.4 \$scalar\$ Substitution

\$scalar\$ substitution inserts the value of a scalar variable into text which defines a PLPROC function or equation. It does this before the PLPROC function or equation is parsed. Hence if substitution results in improper function or equation syntax, this is not detected until PLPROC actually parses the function or equation, and then attempts to evaluate it. (Note that \$scalar\$ substitution cannot be undertaken into a PLPROC line label; line labels are described below.)

Suppose that *svar* is a scalar variable. Its value may have been assigned through direct assignment, through an equation, through reading of a scalar file, or through the PLPROC command line. (Scalar value initiation through command line arguments is described below.) Suppose also that the string “\$svar\$” comprises part of a function or equation that resides in a PLPROC script, and that PLPROC must execute this function sometime after the value of *svar* has been assigned. Then before parsing the function or equation in which this string is emplaced, PLPROC replaces the \$svar\$ string with the integer-converted value of the *svar* scalar variable. Note that integer conversion takes place to the nearest integer. Note also that an error condition will arise (which PLPROC will report) if the value of *svar* is too large or too small to be converted to an integer.

Some examples of \$scalar\$ substitution follow.

```

root="temp"
extension=".dat"
inum = 22.9
filename = root/"$inum$"/extension

```

Figure 6.8 An example of \$scalar\$ substitution in forming the name of a file.

An important feature of string concatenation when undertaken in conjunction with \$scalar\$ substitution is demonstrated in the last line of Figure 6.8. *inum* is real number (as are all scalar variables). The string directly “23” replaces “\$inum\$” in the last line of this figure as \$scalar\$ substitution involves conversion to the nearest integer. After \$scalar\$ substitution, the string equation which defines *filename* then interprets “23” as a string; as such, it must be surrounded by quotes. If, in the last line of Figure 6.8, the \$inum\$ string had not been surrounded by single or double quotes, then an error condition would have occurred as 23 is not the name of a string variable.

While `$scalar$` substitution as a pre-processor to PLPROC equation and function parsing may seem a little cumbersome, it is also powerful. In making `$scalar$` substitution, PLPROC pays no attention to the text into which the substitution is actually made. The names of scalar and string variables, as well as the names of PLISTS, SLISTS and CLISTS can thus be altered. This allows indexing of their names according to loop iteration number.

In the following example it is assumed that PLISTS `kx` and `vert_anis` already exist, and that they share common CLIST parentage. Elements of the `kx1prop` PLIST are assigned values that are products of respective elements of the `kx` and `vert_anis` PLISTS. Elements of the `kx2prop` PLIST are multiplied by a further factor of 3.452. Elements of PLIST `kx3prop` are equal to corresponding elements of `kx`. As is usual protocol with PLPROC equations, the `kx1prop`, `kx2prop` and `kx3prop` PLISTS may have already been defined, or may have been brought into existence through this series of assignment operations.

```
number=1
kx$number$prop = kx*vert_anis
number=2
kx$number$prop=kx*vert_anis*3.452
number=3
kx$number$prop=kx
```

Figure 6.9 An example of `$scalar$` substitution in forming the names of PLISTS.

Through the `$scalar$` substitution mechanism, the names of PLISTS and SLISTS that are manipulated during loop traversals can be linked to the loop variable. This effectively allows definition of arrays of PLISTS and SLISTS.

6.1.5 The `goto()` Function

The `goto()` function informs PLPROC that it must interrupt sequential script execution, and that it must jump to a labelled line and proceed with script execution from that line.

A label is a non-executable statement in a PLPROC script. A line containing a label must begin with a colon (":") character. The text following the colon is the label. Like other PLPROC variables, label names must be 20 characters or less in length and contain no spaces.

The jump in execution point enabled by a `goto()` function can be conditional on the results of a selection equation that is supplied as part of the `goto()` function. This equation should contain only scalar variables, of which one is presumably the loop variable. The following example (the same as used in documentation of the `goto()` function later in this document), illustrates use of the `goto()` function in specifying a loop.

```
i=0
:start_loop
  i=i+1
  storcoeff=stor$i$coeff
  infile='storage'/'$i$'/'ref'
  write_model_input_file(template_file='storage.tpl', &
                        model_input_file=infile)
  goto(label=':end_loop';select=(i.eq.3))
  goto(label=:start_loop)
:end_loop
```

Figure 6.10 A loop under the control of `goto()` functions.

It is presumed that PLPROC script preceding that shown in Figure 6.10 has defined and filled three PLISTs named *stor1coeff*, *stor2coeff* and *stor3coeff*. It is also presumed that the template file *storage.tpl* provides the means for writing the contents of a PLIST named *storcoeff* to a file.

The iteration variable in the loop of Figure 6.10 is the scalar *i*. This variable is defined and initiated through the first line of script shown in the above PLPROC script fragment. The *start_loop* label defines the beginning of the loop while the *end_loop* variable defines the end of the loop.

The first action undertaken within the loop is incrementation of the loop variable *i*. Then the *storcoeff* PLIST is assigned the contents of *stor1coeff*, *stor2coeff* or *stor3coeff*, this depending on the loop count. PLIST selection is done through \$scalar\$ substitution of the loop counter *i* to formulate the name of the pertinent PLIST. At the same time, the file to which the contents of this PLIST are to be written is, in similar fashion, denoted as *storage1.ref*, *storage2.ref* or *storage3.ref*, this also depending on the number of times that the loop has been traversed. In the fourth statement following the *start_loop* label, this file is written. It is always written using the same template file *storage.tpl*. The name of the PLIST cited in the template file cannot be altered through \$scalar\$ substitution; this is why the contents of the *stor*coeff*, PLISTs must be copied to the *storcoeff* PLIST before they are written to the respective *storage*.ref* file.

After the pertinent file is written, the *goto()* function directs PLPROC execution to the *end_loop* label, but only if the scalar variable *i* is equal to 3. If this is not the case, then the next scripting line is executed. This line of script contains another *goto()* function, but without a *select* subargument of the *label* argument. Hence re-direction to the start of the loop is unconditional. Another passage through the loop then takes place.

6.1.6 Command Line Variable Definition

Where PLPROC is run as part of a model through an iterative procedure that is controlled by an external program, iteration-specific values for PLPROC scalar and string variables can be provided through the PLPROC command line. These variables can then be used to govern internal PLPROC operation. For example they can specify the names of files from which information is retrieved or to which it is stored by the PLPROC script.

Figure 6.11 illustrates the command used to run PLPROC wherein PLPROC “wakes up” to find a value of 3.0 assigned to the scalar variable *iteration* and a value of “report.dat” assigned to the string variable *afile*. At the same time, PLPROC is told to read the script file *plproc.in* and execute functions and equations that are recorded in this file. Operations in this file may cause values assigned to *afile* and *iteration* to be over-ridden.

```
plproc plproc.in afile='report.dat' iteration=3
```

Figure 6.11 A PLPROC command line with scalar and string value assignment.

The following rules apply when assigning values to scalar or string variables through the PLPROC command line.

- The name of the scalar or string variable to which a value is assigned must be followed by the “=” symbol; this symbol must be followed by the value of the variable.

-
- The values assigned to string variables must be surrounded by single quotes. It is through the presence of quotes that PLPROC recognizes a variable as a string variable.
 - A number must be assigned to a scalar variable. It is through its status as a number that PLPROC recognizes the variable to which the value is assigned as a scalar variable.
 - If the value of a variable is neither surrounded by single quotes, nor is supplied as a number, PLPROC ceases execution with an error message.
 - If the name of a PLPROC script file is supplied on the command line then it must not be followed by an “=” symbol. If a variable on the PLPROC command line is not followed by an “=” symbol, then it is presumed to be the name of a PLPROC script file. If two variables are unaccompanied by “=” symbols, PLPROC ceases execution with an error message.

7. Conclusions

If numerical models are to be used as a basis for environmental management, they cannot be used on their own. Instead they must be used in partnership with software such as PEST in modes of deployment that extract as much information from historical system datasets as possible, while allowing quantification of the uncertainty associated with model predictions after this information has been extracted. Quantification of predictive uncertainty is essential for quantification of risk. Environmental decision-making can balance risks against benefits to be accrued through implementation of alternative management options only after predictive uncertainty has been quantified, after having been reduced as much as possible through extraction of all available information from existing data.

Increases in model complexity, and in the availability of high quality data, demands commensurate increases in the complexity of model parameterization, and in the sophistication of model-based data processing that is required to inform model parameters. It also demands flexibility in parameter definition, estimation and uncertainty analysis. This allows a modeller to readily adapt the modelling process to each new modelling context that he/she encounters.

Conceptually, a modeller can gain unlimited flexibility in defining and adjusting model parameters by writing context-specific utility software that encapsulates the most appropriate mechanism for PEST-model linkage for use in a particular modelling context. Unfortunately the unlimited flexibility provided by this approach comes at a high cost. In the first place, not all modellers are programmers. Secondly, the programming that may be required to optimize the parameterization of complex models, particularly those that employ an unstructured grid or mesh, is a speciality in itself, one that not all modellers necessarily need to acquire.

An alternative approach is for a modeller to rely entirely on the functionality provided by his/her favourite model graphical user interface. This may be short-sighted however, for the “parameterization prescriptions” which many interfaces provide are not flexible enough to respond to the demands of every modelling context. Innovation thus becomes difficult or impossible.

PLPROC is meant to provide something of a middle ground between the above two extremes. It allows a modeller to build a model, and design some of its parameterization, using a graphical user interface. However it relieves him/her of the burden of having to write his/her own software where he/she finds it necessary to employ innovative parameterization schemes, and to tailor the design of the inverse problem of model calibration to meet the demands of a particular study site in ways that a graphical user interface may not provide.

To be sure, use of PLPROC constitutes a type of programming language in itself. Access to its functionality is not menu-driven, nor can its options be selected through the click of a mouse. It is the author’s hope that its use will eventually be made easier through provision of access to its capabilities by graphical user interfaces, thereby removing this impediment to its use. However, regardless of whether or not this occurs, a modeller would be well-advised to learn PLPROC usage details at the command-line level, for it is anticipated that its development will continue into the indefinite future; hence functionality that is available through direct use of PLPROC will always exceed that which is available through any interface that supports it.

8. References

Deutsch, C and Journel, A., 1998. GSLIB Geostatistical Software Library and User's Guide. Second Edition. Oxford University Press.

Documentation of Specific Functions

alluv_boundary_interp()

General

The *alluv_boundary_interp()* function has been designed to complement PLPROC functions which undertake interpolation using spatially varying anisotropic variograms where it is desired that local anisotropy follow the direction of a meandering river. The user provides an “alluvial boundary file” which defines a minimum of two lines, these enclosing an alluvial system; other pairs of lines can demarcate tributaries to this system. The *alluv_boundary_interp()* function calculates the bearings of all segments of all of these lines. It interpolates these bearings to the elements of PLISTs (probably pilot points) which lie within, or adjacent to, these alluvial channel systems; optionally, other boundary-assigned data can be interpolated to PLIST elements at the same time.

Once target PLIST elements have been informed of the local directions of the alluvial system in this manner, these directions can be further interpolated (using, for example, the *calc_kriging_factors_auto_2d()* function) to the cells of a model grid. This same function can then be used for interpolation of other pilot point data (for example hydraulic properties) to the cells of that grid, with interpolation anisotropy being defined at the model cell level for the use of the *calc_kriging_factors_auto_2d()* function if desired.

Function Specifications

Function value

The *alluv_boundary_interp()* function makes no assignment to any PLPROC entity. Its name must lead the PLPROC command through which it is invoked.

Arguments and subarguments

alluv_boundary_file The name of the alluvial boundary file which is read by the *alluv_boundary_interp()* function. This argument can be contracted to *file* if desired. Specifications of an alluvial boundary file are provided below.

plist This is the name of a PLIST whose elements will be informed through interpolation from alluvial boundaries. Up to five *plist* arguments can be provided in a single *alluv_boundary_interp()* function call, each citing the name of a different PLIST. If more than one *plist* argument is provided, all cited PLISTs must have the same parent CLIST. Any PLIST featured as the value of a *plist* argument must have been defined in previous PLPROC function calls.

plist; datum A *datum* subargument must be provided with each *plist* argument. This is a data type number from the alluvial boundary file for which interpolation must take place to the specified PLIST. A *datum* number of zero instructs the *alluv_boundary_interp()* function to interpolate the bearings of alluvial boundary segments. Values for *datum* can range from 0 to 4.

<i>plist; transform</i>	A <i>transform</i> argument specifies whether interpolation from the alluvial boundary value file to PLIST elements will be logarithmic or not. The value of the argument must be supplied as “yes” or “no”. If <i>datum</i> is supplied as zero for a particular <i>plist</i> argument, then <i>transform</i> must be “no”; alternatively, for a <i>datum</i> value of zero, the <i>transform</i> subargument can be omitted.
<i>select</i>	This argument is optional. A target selection equation can be provided if desired. Interpolation will only take place to those PLIST elements which are selected through this equation.

Discussion

The alluvial boundary file

The figure below illustrates an alluvial boundary file.

```
# This is an alluvial boundary file
3 2 # Number of alluvial systems; number of data types
# The main channel
123 # Number of points in first boundary of first system
457453.23 2344334.2 34.5 134.2 # easting, northing, data1, data2
457445.19 2344298.2 42.2 231.3
.
82 # Number of points in second boundary of first system
456432.87 2338326.6 45.7 344.4 # easting, northing, data1, data2
456574.87 2331452.8 32.4 401.2
.
# First tributary
56 # Number of points in first boundary of second system
4267842.87 223557.6 23.0 233.3
4267041.49 223981.1 19.4 532.3
etc.
```

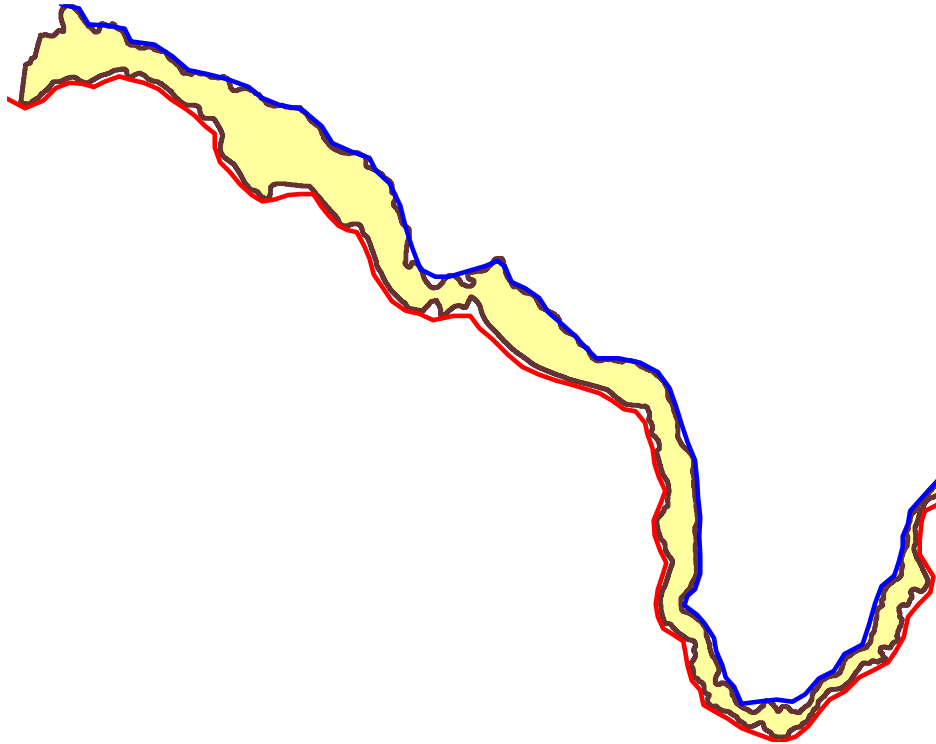
Example of an alluvial boundary file.

Lines on an alluvial boundary file which begin with the “#” character are treated as comments; they are therefore ignored. Comments can appear anywhere within an alluvial boundary file. So too can blank lines, which are also ignored.

The first data line in an alluvial boundary file must contain two integers. The first is the number of alluvial systems that are represented in the file; the second is the number of data types which are associated with alluvial system boundaries. Actually, there is a data type which is associated with alluvial boundaries by default. This is data type zero; it is the bearing (i.e. the angle with respect to north) of each segment of each boundary.

Next follow the specifications of each alluvial system. Each alluvial system dataset is defined by two boundary lines. The first such line must run along one side of the system, while the second boundary line must run along the other side of the system. They must run in the same direction (for example downstream). They must not join. Each segment of a system boundary line is defined by two points; neighbouring segments share a common point; collectively adjacent segments comprise a boundary line. As stated above, the bearings (i.e. directions) of these segments can be interpolated to target PLIST elements.

The figure below shows an alluvial system; this has a yellow interior and brown boundaries. The blue and red lines can be used to define system boundaries appearing in an alluvial boundary file. Points along the boundary can be readily digitized using software such as SURFER. These boundary lines do not need to be exact; in fact, they will probably be smoother than the true alluvial boundaries. The purpose of these lines is to define approximate local alluvial system directions for the purpose of providing location-specific anisotropies for the use of interpolation functions supported by PLPROC.



An alluvial system (yellow with brown boundary), together with two lines used to bound the system as supplied in an alluvial boundary file.

As stated above, two lines must be provided for each alluvial system. In the alluvial boundary file these are represented by a sequence of points; lines joining these points (i.e. segments) form the system boundary line. Boundary lines must run along opposite sides of the system in the same direction. The easting and northing of each point (i.e. vertex) along each of these lines must be provided in an ordered list. The record preceding each such list in the alluvial boundary file must contain a single integer, this being the number of points in the following list.

If the number of data types on the first data line of the alluvial boundary file is specified as zero, then each line (i.e. record) of the alluvial boundary file which provides vertex coordinates needs to provide only two numbers, these being the easting and northing of the respective vertex. However if the number of data types is specified as N , then N items of data must follow these two coordinates - except for the last record of each line for which no data values need be provided. Data items are, in fact, ascribed to the midpoint of each line segment by the *alluv_boundary_interp()* function; however they appear in the alluvial boundary file alongside the vertex that defines the beginning of each segment.

More than one alluvial system (each defined by two boundary lines), can be represented in a single alluvial boundary file. However the order in which they are provided is important.

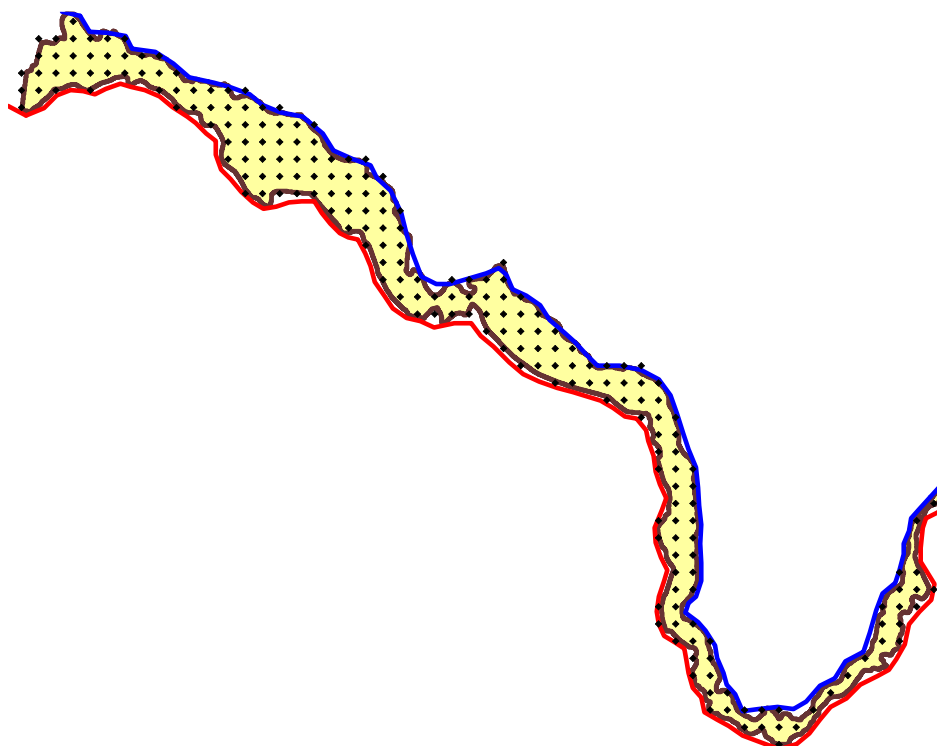
When interpolating from system boundaries to the locations of PLIST elements, the *alluv_boundary_interp()* function associates a target PLIST element with an alluvial system by trying to enclose it in that alluvial system's boundaries. The *alluv_boundary_interp()* function forms a polygon for each supplied system by joining the starting points of the two boundary lines to each other, and by joining the finishing points of the two boundary lines to each other, using a single straight line in each case. Where a main channel and one or more tributary channels are represented in the same alluvial boundary file, alluvial system polygons formed in this manner may overlap. Where this is the case, the order of precedence as far as polygon containment of a target PLIST element is concerned is the same as the order in which alluvial systems are defined in the alluvial boundary file; hence a target PLIST element belongs to the first alluvial system which encloses it (which should normally be the main alluvial channel). If a target PLIST element is not enclosed by any alluvial polygon, then interpolation from alluvial boundaries to the point accommodates this. Interpolation details are provided below.

When defining alluvial boundaries, the following rules should be observed.

- Maintain rough equality of boundary segment lengths (i.e. separation of boundary vertices) in areas where there is little or no boundary curvature; however feel free to make segment lengths smaller where boundaries become complex. In general, boundary segment lengths should be about a fifth to a half of the separation between opposite boundaries. As is discussed below, the manner in which the *alluv_boundary_interp()* function interpolates from alluvial boundaries to target PLIST elements is simplistic. Furthermore, interpolation actually takes place from segment centre-points. Artificialities in interpolation can arise if boundary segments are too long, and hence if interpolation source points are too widely spaced.
- Extend the alluvial system boundaries supplied in the alluvial boundary file past the last points to which interpolation must take place in both the upstream and downstream directions of an alluvial system. As stated above, an alluvial system polygon is formed by joining the ends of each of the two boundaries which define the opposite sides of the alluvial system with a straight line. Ideally the polygon that is formed in this fashion should enclose all points to which interpolation must take place. (The north western end of the alluvial system depicted in the following figure shows an example of where this rule is broken.)
- For the same reason, alluvial boundaries provided in an alluvial boundary file may need to be somewhat further apart than the actual boundaries of an alluvial system. This helps to ensure that all pilot points that are associated with a particular alluvial system are enclosed by the polygon which defines the system.

Interpolation to target PLIST elements

The figure below shows a set of pilot points that comprise the target PLIST elements to which interpolation from the alluvial boundaries which were depicted in the previous figure takes place. It is apparent that most of these points lie within the polygon defined from alluvial boundaries supplied in the alluvial boundary file. However some target elements lie outside of it. While this is not ideal, the interpolation algorithm provided by the *alluv_boundary_interp()* function, while simplistic, can accommodate this, as long as the target points do not lie too far from a boundary.



Target PLIST elements to which alluvial boundary interpolation takes place.

Before undertaking interpolation to target PLIST elements, the *alluv_boundary_interp()* function calculates the midpoint of each of the many line segments which collectively define an alluvial boundary. As mentioned above, these midpoints are the locations from which data interpolation takes place. Interpolation to an arbitrary target PLIST element lying within an alluvial polygon then proceeds as follows.

1. The alluvial polygon in which the point lies is first determined. Recall the order of polygon precedence discussed above.
2. The closest boundary segment midpoint to the target point on each side of the alluvial boundary is evaluated.
3. An “interpolated value” is assigned to the target element twice, first using data associated with one side of the boundary, and then using data associated with the other side of the boundary. (This “data” can be either the bearings of boundary segments, and/or numbers assigned to boundary line segments in the alluvial boundary file.) This is done through taking a weighted average of data assigned to the closest segment, and to the segment on either side of it; weighting is in accordance with inverse squared distance from the target point to these segments.
4. The final interpolated value at the target point is then determined through weighted averaging of these two values; weights are the inverse squared distances from the target point to the closest segment on either boundary.

While this the interpolation scheme is simplistic, it “does the job” where it is employed to assign alluvial system directions to pilot points whose main purpose is to hold data of other types for eventual interpolation to a model grid.

If a target point does not lie within an alluvial polygon, then a search of all alluvial boundaries is made in order to determine the closest boundary segment midpoint to it. The “interpolated” value at the target element is then calculated as the weighted average of values

ascribed to that boundary segment, and to those on either side of it. Weighting is, once again, according to inverse squared distance. It is readily apparent that this “interpolation” scheme is useable only where a target point does not lie too far from an alluvial boundary.

Interpolated data types

The data type interpolated to target PLIST elements depends on the *datnum* subargument associated with the respective *plist* argument. If *datnum* is zero, then boundary segment bearings are interpolated to target PLIST elements. Otherwise data associated with alluvial boundary segments supplied in the alluvial boundary file are interpolated to target PLIST elements. If interpolation is logarithmic, then all such data must be positive in value.

Use of the *alluv_boundary_interp()* function requires that the following rules be obeyed. If they are not obeyed, an error condition will be encountered and an appropriate error message issued.

- The number of data types associated with boundary line segments in an alluvial boundary file cannot exceed 4.
- The number of *plist* arguments appearing in an *alluv_boundary_interp()* function cannot exceed 5.
- Two PLISTS appearing in *alluv_boundary_interp()* function arguments cannot be assigned the same *datnum*.

If the *datnum* subargument for a particular *plist* argument is supplied as zero, then segment bearings are interpolated to elements of the target PLIST in the manner described above. However special considerations apply when interpolating bearings because of the inherent discontinuity in bearings that must occur at some point of the compass. (For example, slightly west of south can be designated as -179 degrees while slightly east of south can be designated as 179 degrees.). While this discontinuity has no effect on calculated anisotropies, it can have disastrous consequences if the target PLIST elements are pilot points and further interpolation of a discontinuous dataset is then undertaken to the elements of a model grid. (The average of -179 degrees and 179 degrees is not zero degrees; it is either 180 degrees or -180 degrees.) The *alluv_boundary_interp()* function does its best to avoid this problem by altering bearings calculated for target PLIST elements so that this discontinuity is not encountered between any bearings which are ascribed to these elements. This may result in some interpolated bearings being smaller than -180 degrees or greater than 360 degrees where alluvial systems have complex shapes. This should make no difference to calculated anisotropies.

Use of the function

Function *alluv_boundary_interp()* was written to assist usage of the *calc_kriging_factors_2d_auto()* function in alluvial contexts. That function allows hydraulic property anisotropy to be supplied for all cells in a model domain to which interpolation must take place. While the *alluv_boundary_interp()* function can be used to assign anisotropy values directly to model cells, smoother variation of model-cell-based anisotropy is generally achieved if interpolation to model cells is undertaken in two steps. Firstly, interpolation of alluvial boundary directions takes place to pilot points (perhaps the same set of pilot points as that from which hydraulic properties will later be interpolated using the *calc_kriging_factors_2d_auto()* function); these bearings can then be interpolated to the model grid using, for example, the *calc_kriging_factors_2d_auto()* function again. The second stage of interpolation removes any artificial discontinuities that may arise through

direct interpolation from alluvial boundaries to the model grid through use of the interpolation scheme discussed above. While that scheme allows bearings to closely follow nearby alluvial boundaries, the cost of such close boundary tracking can be the creation of discontinuities as boundary segments move in and out of the interpolation process in accordance with whether, or not, they are the closest segments to an interpolation target.

Examples

Example 1

```
alluv_boundary_interp(alluv_boundary_file="alluv.dat",      &
                     plist=bearing;datnum=0)              &
```

This is the simplest (and probably most common) example of function *alluv_boundary_interp()* usage. Bearings of alluvial boundary segments defined in file *alluv.dat* are interpolated to elements of a PLIST named *bearing*.

Example 2

```
alluv_boundary_interp(alluv_boundary_file="alluv.dat",      &
                     plist=bearing;datnum=0,                &
                     select=(zone.ne.0))                    &
```

This example is similar to the preceding example except for the fact that interpolation takes place only to elements of the *bearing* PLIST for which elements of a *zone* SLIST are non-zero. Note that *zone* and *bearing* must have the same parent CLIST.

Example 3

```
alluv_boundary_interp(alluv_boundary_file="alluv.dat",      &
                     plist=bearing;datnum=0,                &
                     plist=other_data;datnum=2;transform=none; &
                     select=(zone.ne.0))                    &
```

This is slightly more complicated than the previous example in that the alluvial boundary file ascribes at least two data types to each alluvial boundary segment. The second of these is interpolated to the *other_data* PLIST. Interpolation takes place in exactly the same way as for segment bearing data; however the logarithms of data values are subject to the inverse squared distance weighting procedure described above; final values are back-transformed to the domain of natural numbers before being stored as PLIST elements.

assign_by_relation()

General

The *assign_by_relation()* function assigns values to elements of one PLIST based on those of another PLIST. The “assign-to” (i.e. target) and “assign-from” (i.e. source) PLISTs can actually be the same PLIST if desired, as long as target and source PLIST elements do not overlap (as can be ensured through use of appropriate target and source selection equations).

Elements in the target PLIST to which values are transferred from the source PLIST are identified through commonality of values of associated SLIST elements, similarity of values of associated PLIST elements, and/or similarity of x , y , and/or z coordinates of target and source CLIST elements. Up to five (as presently programmed) such associations can be used to define element value transfer; all association relations must be simultaneously met for PLIST data transfer to take place.

Function Specifications

Function value

The *assign_by_relation()* function assigns values to elements of an existing PLIST (referred to as the “target PLIST”). The name of this PLIST constitutes the output of the function. This name must appear before the function in a command which invokes the function. The name should be separated from the function by a “=” symbol. Optionally a selection equation can accompany the name of the target PLIST.

Object associations

Function *assign_by_relation()* operates on an existing PLIST (referred to as the “source PLIST”). A call to this function must follow the name of this source PLIST, with a dot between the two.

Arguments

<i>select</i>	Source PLIST selection equation.
<i>relate</i>	The type of relationship that must prevail between source and target PLIST elements if PLIST element values are to be transferred from the former to the latter. Up to five <i>relate</i> arguments can be provided together with appropriate subarguments; the nature of the subarguments depends on the nature of the relationship defined as the value of the <i>relate</i> argument. Allowed values of the <i>relate</i> argument are “slist”, “plist”, “x”, “y” and “z”. At least one <i>relate</i> argument must be provided in the <i>assign_by_relation()</i> function.
<i>relate;source</i>	If the value of the <i>relate</i> argument is “slist” or “plist” then the name of a source relational SLIST or PLIST respectively must be provided as a subargument. The parent CLIST for this source relational SLIST or PLIST must be the same as that of the source PLIST of the <i>assign_by_relation()</i> function.

<i>relate;target</i>	If the value of the <i>relate</i> argument is “slist” or “plist” then a target relational SLIST or PLIST respectively must be provided as a subargument. The parent CLIST for this target relational SLIST or PLIST must be the same as that of the target PLIST of the <i>assign_by_relation()</i> function. Data transfer from a source PLIST element to a target PLIST element will only take place if pertinent relational source and target SLIST elements are identical, or if pertinent relational source and target PLIST elements are identical to within a user-defined tolerance. (Note that values of relational list elements are not altered.)
<i>relate;tol</i>	If the value of the <i>relate</i> argument is “plist”, “x”, “y” or “z” then the value assigned to the mandatory <i>tol</i> subargument specifies how close the numerical values of these quantities must be for the relationship to exist and for data transfer to thereby take place.
<i>num_target_match</i>	Data transfer can take place from a single source PLIST element to up to <i>num_target_match</i> target PLIST elements which satisfy all relationships defined through <i>relate</i> function arguments. Once this number is exceeded the <i>assign_by_relation()</i> function searches for no further relational matches between a source PLIST element and various target PLIST elements. Instead it moves on to the next source PLIST element; this can result in significant computational savings if, for example, it is known that relationships are one-to-one. The value of this argument must be set to 1 or greater. If this argument is omitted, it is set internally to a very high number.
<i>dual_assign_error</i>	If an attempt is made to assign a value to a target PLIST element by transferring values to this element from more than one source PLIST element, a flag is set. This will result in an error condition if the value of <i>dual_assign_error</i> is set to “yes”. Alternatively, if <i>dual_assign_error</i> is set to “no”, then multiple values assigned to the same target PLIST element are averaged. If this argument is omitted, a value of “no” is assumed.
<i>no_match_value</i>	If no value is assigned to a PLIST element that is not excluded from the target PLIST element selection range through a target selection equation, then a user-specified value can be assigned to this element. This value is supplied as the value of the <i>no_match_value</i> argument. (This will occur if no relational specifications between source and target PLIST elements are met for this target PLIST element.) If this argument is omitted, the value of the unmatched PLIST element is left unchanged from its existing value.

Discussion

Function algorithmic details

The *assign_by_relation()* function denotes a source and target PLIST. The function considers each element of the source PLIST in turn; the elements which it considers may be restricted through use of a source selection equation. For each source PLIST element it must decide what target PLIST elements it must transfer the value of the source PLIST element to. Hence it visits all elements of the target PLIST that are selected through the target PLIST selection equation. Up to five relational conditions must be met for data transfer to take place; if any of these are violated then data transfer does not take place. These relations are specified using up to five *relate* arguments. If all are satisfied, data transfer takes place and the *assign_by_relation()* function then moves on to the next target PLIST element to see if this element too can be related to the particular source PLIST element that is under consideration. However if the value of the *num_target_match* argument is set to 1, the *assign_by_relation()* function ceases exploration of the target PLIST and moves on to the next source PLIST element. Alternatively if the *num_target_match* argument is set to *N*, then *N* data transfers will take place before the search of the target PLIST for further matches is abandoned, and the *assign_by_relation()* function moves on to the next source PLIST element. Setting *num_target_match* to a low value can considerably hasten execution speed. However it can result in omissions of data transfer if a particular source PLIST element can be related to a large number of target PLIST elements.

If a single target PLIST element can be related to multiple source PLIST elements, this may signify poor design of the relationships which promote source-to-target PLIST data transfer. An error condition can then be triggered if the value of the *dual_assign_error* argument is set to “yes”. Alternatively, multiple target PLIST element assignments are averaged by the *assign_by_relation()* function.

If a target PLIST element in the range defined by a target PLIST selection equation is unmatched to any source PLIST elements, then it can optionally be assigned a value specified through the *no_match_value* argument.

Some function uses

Following are two uses to which the *assign_by_relation()* function can be put.

- Values of hydraulic properties can be transferred from one layer of a subsurface reservoir or groundwater model to one or a number of underlying layers. In this case, similarity of parent CLIST element *x* and *y* coordinates, together with selection equations based on a *layer_number* SLIST can be used to direct data transfer.
- Pilot points could be used to parameterize a two-dimensional (horizontal) PLIST. The interpolated dataset could then be transferred to a dipping fault of arbitrary thickness and disposition where the relationship between the horizontal PLIST and the fault PLIST is provided by corresponding SLISTS which define a one-to-one or one-to-many template for data transfer.

Other uses of this function, based on other means by which relationships can be defined between elements of the same or different PLISTS, can be readily established.

Examples

Example 1

```
p1(select=(layer.eq.2)) = p1.assign_by_relation(      &
      select=(layer.eq.1),                          &
      relate=x;tol=0.001,                            &
      relate=y;tol=0.001,                            &
      num_target_match=1)
```

In this example values are transferred from one part of the *p1* PLIST to another part of the *p1* PLIST. Source elements of *p1* are identified as elements for which values of the corresponding *layer* SLIST are 1. The target area of *p1* is identified as elements for which values of the *layer* SLIST are 2. Transfer takes place if source and target PLIST elements have parent CLIST *x* and *y* values which are equal to within a tolerance of 0.001. As soon as a target PLIST element is matched to a source PLIST element, the *assign_by_relation()* function moves to the next source PLIST element, checking for matches with no further target PLIST elements. This reduces the search time.

Example 2

```
hk_fault = hk_plane.assign_by_relation(      &
      relate=slist;source=splane;target=sfault,    &
      relate=slist;source=szone;target=szone)
```

In this example it is assumed that the *hk_fault* and *hk_plane* PLISTs have different parent CLISTs. The parent CLIST of the *sfault* SLIST is the same as that of the *hk_fault* PLIST; the parent CLIST of the *splane* SLIST is the same as that of the *hk_plane* PLIST. A value pertaining to an element of the *hk_plane* PLIST is transferred to all elements of the *hk_fault* PLIST which have the same *sfault* SLIST value as the *splane* value pertaining to the source PLIST element in question. However this will only occur if both the source and target PLIST elements belong to the same zone, as defined through the *zsource* and *zfault* SLISTS; the parent CLISTs for these two SLISTS are the same as those of the *hk_source* and *hk_fault* PLISTs respectively.

Example 3

```
hk_fault = hk_plane.assign_by_relation(      &
      relate=plist;source=dsource;target=dtarget,tol=1.0  &
      relate=slist;source=szone;target=szone)
```

This example is similar to the previous example. However data transfer takes place between the source *hk_plane* and target *hk_fault* PLISTs only if corresponding elements of the *dsource* and *dtarget* PLISTs are the same to within a tolerance of 1.0.

assign_by_slist()

General

The *assign_by_slist()* function can be used to distribute element values of a smaller PLIST over elements of a larger PLIST. Distribution of these values is accomplished through use of SLISTs – one of which shares the same CLIST parent as the smaller PLIST and one of which shares the same CLIST parent as the larger PLIST. Element value transfer takes place on the basis of equality of integer elements of these SLISTs. The functionality provided by *assign_by_slist()* is a subset of that provided by *assign_by_relation()*.

Function Specifications

Function value

The *assign_by_slist()* function assigns values to elements of an existing PLIST (referred to as the “target PLIST”). The name of this PLIST constitutes the output of the function. This name must appear before the function in a command which invokes the function. The name should be separated from the function by a “=” symbol. Optionally a selection equation can accompany the name of the target PLIST.

Object associations

Function *assign_by_slist()* operates on an existing PLIST (referred to as the “source PLIST”). A call to this function must follow the name of this source PLIST, with a dot between the two.

Arguments

<i>target_slist</i>	The name of an SLIST whose CLIST parent is the same as that of the target PLIST.
<i>source_slist</i>	The name of an SLIST whose CLIST parent is the same as that of the source PLIST.

Discussion

Let *sp* designate a small PLIST where “*sp*” stands for “source PLIST”. Let the values of the elements of this PLIST be designated as sp_i . Suppose that an associated SLIST is named *ss* and that its element values are designated as ss_i .

Let *tp* designate a larger PLIST, where “*tp*” stands for “target PLIST”. Let tp_i designate its element values. Let ts_i designate element values for an associated SLIST.

Use of the *assign_by_slist()* function is predicated on the assumption that the elements of *ss* have unique integer values. Thus each integer value provides a unique designation of its corresponding element, and thereby of the corresponding element of any other SLIST or PLIST which shares the same parent CLIST as *ss*. In contrast, element values of *ts* do not need to be unique.

Suppose that the value of the i 'th element of ts is the integer n . In assigning a real value to the i 'th element of tp , the `allocate_by_slist()` function searches the elements of ss to find a value of n . Suppose that element j of ss has this value. Suppose that the corresponding element j of sp has the real value r . Then `allocate_by_slist()` assigns the value r to tp_i . Alternatively, if no element of ss has the value n , then the value of tp_i remains unchanged.

Examples

Example 1

```
p2=p1.assign_by_slist(target_slist=s2,source_slist=s1)
```

Element values are transferred from the source PLIST $p1$ to the target PLIST $p2$ according to correspondence of integer values between the SLIST $s1$ which shares common CLIST parentage with the $p1$ PLIST, and the SLIST $s2$ which shares common CLIST parentage with the $p2$ PLIST.

Example 2

```
p2(select=(s2a==1))=p1.assign_by_slist(target_slist=s2,source_slist=s1)
```

This is identical to the previous example except for the fact that element values of the $p2$ PLIST are altered only if values of the associated $s2a$ SLIST are equal to 1.

assign_from_closest()

General

The *assign_from_closest()* function assigns values to elements of one SLIST or PLIST based on element values pertaining to another SLIST or PLIST. The source and target LISTS will probably have different parent CLISTs. For any particular target LIST element, the closest source LIST element is located. The value of this closest source LIST element is then assigned to the target LIST element.

Where source and target LISTS are large, the process of finding the closest element of the source LIST to all elements in turn of the target LIST can be a time-consuming matter. The numerical burden can be reduced through use of source and target selection equations, as well as through use of other function arguments which prevent the *assign_from_closest()* function from undertaking unnecessary distance calculations.

Functionality provided by the *assign_from_closest()* function encompasses and exceeds that provided by the *assign_if_colocated()* function. One manner in which it is superior to the latter function is that it can be used for the assignment of values to SLIST elements. In doing this, it can prove useful in the building of relational SLISTs for use of the *assign_by_relation()* function.

Function Specifications

Function value

The *assign_from_closest()* function calculates values for some or all elements of an existing SLIST or PLIST; this is referred to as the “target LIST”. The name of the target SLIST or PLIST comprises the output of the function. This name must appear before the function in a PLPROC command which uses the function, optionally accompanied by a selection equation. This name should be separated from the function by a “=” symbol.

Object associations

Function *assign_from_closest()* assigns the values of source SLIST or PLIST elements to elements of a target SLIST or PLIST. The name of this function must follow the name of the source LIST on which it operates; a dot must separate these two names.

Arguments and subarguments

<i>select</i>	The source LIST selection equation. This argument is optional. Note that a target LIST selection equation can also be supplied if desired. As stated above, the latter must be placed adjacent to the name of the target LIST on the left side of the assignment operator; see the example below.
<i>search_dim</i>	The value supplied for this mandatory argument must be “2” or “3”. This specifies whether distance is calculated only in the horizontal direction (whereby <i>search_dim</i> should be set to “2”) or in three dimensions (whereby <i>search_dim</i> should be set to “3”). The latter option is not allowed unless the parent CLISTs of both

of the source and target LISTs are three-dimensional.

<i>search_dist_hor</i>	The search distance in the horizontal direction. Elements of the source LIST which are greater than this horizontal distance from a target LIST element are not considered when assigning a value to the latter element.
<i>search_dist_vert</i>	The search distance in the vertical direction. Elements of the source LIST which are greater than this vertical distance from a target LIST element are not considered when assigning a value to the latter element. This argument is mandatory if <i>search_dim</i> is 3.
<i>closest_hor_tol</i>	This optional argument can be used to reduce search time. If the horizontal distance between a source and target LIST element is less than or equal to the value of this argument, the elements are presumed to be matched and no further searching of the source LIST for a closer element is undertaken. Note that for a three-dimensional search the <i>closest_vert_tol</i> distance must also be respected before a match is declared.
<i>closest_vert_tol</i>	This optional argument can be used to reduce search time. If the horizontal distance between a source and target LIST element is less than or equal to <i>closest_hor_tol</i> , and the vertical distance between a source and target LIST element is less than or equal to <i>closest_vert_tol</i> , the elements are presumed to be matched and no further searching of the source list for a closer element to the target element is undertaken.
<i>too_far_value</i>	If no selected element of the source LIST is within a horizontal distance of <i>search_dist_hor</i> and a vertical distance of <i>search_dist_vert</i> of a target LIST element, a value of <i>too_far_value</i> as assigned to the latter element.

Discussion

Operation of the *assign_from_closest()* function is straightforward. Calculation of the closest source LIST element to a target LIST element can take place in the horizontal dimension (i.e. in two dimensions) or in three dimensions. Separate search distances can be specified for these two directions.

Finding the closest source LIST element to a particular target LIST element can be time-consuming. This process can be greatly expedited if the user knows that if a source element is closer than a certain distance to a target element, then no other source element will be closer. Consequently, if a source element is found that lies within this distance, then source-to-target LIST element value assignment takes place immediately and the no further searching is undertaken for a closer source LIST element. Where a search is two-dimensional the *closest_hor_tol* argument provides the “close enough” distance in the horizontal direction. Where a search is three-dimensional the *closest_vert_tol* argument must also be supplied; the distance to a source LIST element must be smaller than both these tolerances in both directions for a status of “close enough” to be declared.

Further gains in efficiency can be made through use of selection equations. For example where LISTS are three-dimensional but elements pertain to different layers of a particular model, then undertaking two-dimensional assignment on a layer-by-layer basis may prove far more efficient than one large three-dimensional search-and-assignment operation.

Examples

Example 1

```
pal=p1.assign_from_closest(           &
                                search_dim=3,           &
                                search_dist_hor=1e20,    &
                                search_dist_vert=1e20,   &
                                too_far_value=999)
```

The above call to *assign_from_closest()* provides a particularly simply example of its usage. Every element of the *pal* SLIST/PLIST is assigned a value equal to the closest element of the *p1* SLIST/PLIST. Given the search radii, it is highly unlikely that any element of *pal* will be assigned a value of 999.

Example 2

```
sal(select=(layer_t==2))=s1.assign_from_closest(           &
                                select=(layer_s==2),       &
                                search_dim=2,               &
                                search_dist_hor=1000.0,     &
                                closest_hor_tol=10,         &
                                too_far_value=999)
```

Values associated with elements of the *s1* LIST are transferred to elements of the *sal* LIST. However this takes place only for elements of *sal* for which elements of the *layer_t* SLIST are equal to 2. Of course, *sal* and *layer_t* must share a common parent CLIST. When finding a particular element of *s1* that is closest to a particular element of *sal* the search is restricted to elements of *s1* for which values of the *layer_s* SLIST are equal to 2; *s1* and *layer_s* must share the same parent CLIST. If any element of *sal* is further than a distance of 1000.0 from the nearest *s1* element, then it is assigned a value of 999. If any element of *s1* is closer than 10.0 to an element of *sal*, the search for closer *s1* elements is terminated as the closest element to the respective *sal* element is deemed to have been found. Only horizontal distances are considered when undertaking the search.

assign_if_colocated_2d()

General

The *assign_if_colocated_2d()* function provides a means of transferring values from one PLIST to another, notwithstanding the fact that the PLISTs are dependent on different reference CLISTs. Elements of a target PLIST are assigned values equal to the elements of a source PLIST where the two have equal, or nearly equal, x and y coordinates; differences in z coordinates (if one or both of the PLISTs possess reference CLISTs which are three dimensional) are not taken into account.

If required, a selection equation can be employed to limit the source PLIST elements that are employed in PLIST-to-PLIST data transfer. Another, entirely different, selection equation can be used to limit target PLIST elements involved in this process. The source selection equation can dictate element selection on the basis of source PLIST, source parent CLIST and source sibling SLIST/PLIST element values and properties. The target selection equation can dictate element selection on the basis of target PLIST, target parent CLIST and target sibling SLIST/PLIST element values and properties.

Function Specifications

Function value

The *assign_if_colocated_2d()* function assigns values to elements of an existing PLIST (referred to as the “target PLIST”). The name of this PLIST constitutes the output of the function. This name must appear before the function in a command which invokes the function. The name should be separated from the function by a “=” symbol. Optionally a selection equation can accompany the name of the target PLIST.

Object associations

Function *assign_if_colocated_2d()* operates on an existing PLIST (referred to as the “source PLIST”). A call to this function must follow the name of this source PLIST, with a dot between the two.

Arguments

acceptance_threshold The distance in the horizontal direction between two points below which they are deemed to be co-located.

select Source PLIST selection equation.

Discussion

As stated above, an element of a target PLIST is assigned a value equal to an element of a source PLIST if the two elements are co-located to within a user-specified distance threshold. Suppose that this threshold is T distance units. If more than one element of the source PLIST is within a distance T of a particular element of the target PLIST, PLPROC undertakes element value assignment on the basis of the first source PLIST element that it encounters. Where PLISTs are large it would take too long to then look for any other elements of the

source PLIST which may also be co-located with a particular element of the target PLIST. The user should keep this in mind when selecting the co-location threshold distance.)

If, for any element of the target PLIST, no element of the source PLIST can be found which is within a co-location threshold distance of it, the value of that particular target PLIST element remains unchanged.

Examples

Example 1

```
hk_broad=hk_fine.assign_if_colocated_2d(acceptance_threshold=1.0)
```

Use of the above command is predicated on the assumption that two PLISTs, one named *hk_broad* and the other named *hk_fine*, have been defined. The two PLIST's are effectively overlain. If any element of *hk_fine* is within a horizontal distance of 1.0 length unit from an element of *hk_broad*, then the latter element is assigned a value equal to that of the former.

Example 2

```
hk_broad(select=(layer_broad==1))=hk_fine.assign_if_colocated_2d(      &
                                acceptance_threshold=1.0,              &
                                select=(layer_fine<10))
```

This invocation of the *assign_if_colocated_2d()* function will assign *hk_fine* element values to elements of the *hk_broad* PLIST only if members of the *layer_fine* SLIST are less than 10, and members of the *layer_broad* SLIST are equal to 1. It is assumed that the *layer_fine* SLIST has the same parent CLIST as the *hk_fine* PLIST, and that the *layer_broad* SLIST has the same reference CLIST as the *hk_broad* PLIST.

build_covmat_from_vario()

General

Function *build_covmat_from_vario()* builds a covariance matrix that pertains to selected elements of a two- or three-dimensional CLIST. The matrix is built from a two- or three-dimensional variogram. Alternatively, its construction can be based on non-stationary two- or three-dimensional variograms whose properties can vary from CLIST element to CLIST element. In all cases the covariance matrix constructed by the *build_covmat_from_vario()* function is, square, symmetric and positive definite.

Function Specifications

Function value

The *build_covmat_from_vario()* function creates and populates a MATRIX. The name of this new MATRIX is the target of the function. This name must appear before the function in a PLPROC command which uses the function. This name should be separated from the function by a “=” symbol.

Object associations

Function *build_covmat_from_vario()* operates on an existing two- or three-dimensional CLIST (referred to as the “source CLIST”). A call to this function must follow the name of this source CLIST, with a dot between the CLIST name and the function name.

Arguments and subarguments

<i>Select</i>	A selection equation which can be used to restrict the number of CLIST elements for which covariance matrix elements are generated.
<i>variogram</i>	The name of the variogram employed in calculation of the covariance matrix. Options are “spherical”, “exponential”, “gaussian” and “power”.
<i>Nugget</i>	The geostatistical nugget value. If this argument is omitted the nugget is assumed to be zero.
<i>nugget_plist</i>	This argument can be employed instead of the <i>nugget</i> argument if the <i>nugget</i> can differ from element to element of the source CLIST. The PLIST supplied as the value of this argument must then provide the value of the <i>nugget</i> on an element-by-element basis. The parent CLIST of this PLIST must be the source CLIST of the function.
<i>Sill</i>	The sill of the variogram. This argument, or the <i>sill_plist</i> argument, are mandatory.
<i>sill_plist</i>	This argument can be employed instead of the <i>sill</i> argument if the <i>sill</i> can differ from element to element of the source CLIST. In this case it provides the value of the <i>sill</i> on an element-by-element basis. The

	parent CLIST of the PLIST supplied as the value of this argument must be the source CLIST of the function.
<i>a</i>	The “a” value pertaining to the variogram. This argument can be used only if the source CLIST is two-dimensional, in which case either this argument, or the <i>a_plist</i> argument, is mandatory.
<i>a_plist</i>	This argument can be employed instead of the <i>a</i> argument if the variogram <i>a</i> value can differ from element to element of the source CLIST. It provides the value of <i>a</i> on an element-by-element basis. The parent CLIST of the PLIST supplied as the value of this argument must be the source CLIST of the function.
<i>hanis</i>	The horizontal anisotropy of a two-dimensional variogram. This is the ratio of the variogram range in the <i>bearing</i> direction to its range at right angles to this direction. If the <i>hanis</i> argument is omitted, then the horizontal anisotropy is assumed to be 1.0. If it is included, then a <i>bearing</i> or <i>bearing_plist</i> argument must also be provided. If the source CLIST is three-dimensional then a <i>hanis</i> argument must not be supplied.
<i>hanis_plist</i>	This argument can be employed instead of the <i>hanis</i> argument if the horizontal anisotropy can differ from element to element of the source CLIST. It provides the value of <i>hanis</i> on an element-by-element basis. The parent CLIST of the PLIST supplied as the value of this argument must be the source CLIST of the function.
<i>bearing</i>	The bearing (in degrees clockwise from north) of the major axis of anisotropy of a two-dimensional variogram; this is the direction of maximum range. This argument must be supplied if a <i>hanis</i> or <i>hanis_plist</i> argument is supplied. It must not be supplied otherwise.
<i>bearing_plist</i>	This argument can be employed instead of the <i>bearing</i> argument if the direction of the major axis of variogram anisotropy can vary from element to element of the source CLIST. It provides the value of <i>bearing</i> on an element-by-element basis. The parent CLIST of the PLIST supplied as the value of this argument must be the source CLIST of the function.
<i>a_hmax</i>	This argument must be supplied if the source CLIST is three-dimensional. It specifies the maximum horizontal “a” value of a three-dimensional variogram, this being its value in the direction of <i>ang1</i> (see below).
<i>a_hmax_plist</i>	This argument can be employed instead of the <i>a_hmax</i> argument if <i>a_hmax</i> can vary from element to element of the source CLIST. It provides the value of <i>a_hmax</i> on an element-by-element basis. The parent CLIST of the PLIST supplied as the value of this argument must be the source CLIST of the function.

<i>a_hmin</i>	This argument must be supplied if the source CLIST is three-dimensional. It specifies the minimum horizontal “a” value of a three-dimensional variogram, this being its value in the direction at right angles to <i>ang1</i> .
<i>a_hmin_plist</i>	This argument can be employed instead of the <i>a_hmin</i> argument if <i>a_hmin</i> can vary from element to element of the source CLIST. It provides the value of <i>a_hmin</i> on an element-by-element basis. The parent CLIST of the PLIST supplied as the value of this argument must be the source CLIST of the function.
<i>a_vert</i>	This argument must be supplied if the source CLIST is three-dimensional. It specifies the “a” value of a three-dimensional variogram in the vertical direction (see below).
<i>a_vert_plist</i>	This argument can be employed instead of the <i>a_vert</i> argument if <i>a_vert</i> can vary from element to element of the source CLIST. It provides the value of <i>a_vert</i> on an element-by-element basis. The parent CLIST of the PLIST supplied as the value of this argument must be the source CLIST of the function.
<i>ang1</i>	This argument must be supplied if the source CLIST is three-dimensional. It is the bearing (in degrees with respect to north) pertaining to <i>a_hmax</i> .
<i>ang1_plist</i>	This argument can be employed instead of the <i>ang1</i> argument if <i>ang1</i> can vary from element to element of the source CLIST. It provides the value of <i>ang1</i> on an element-by-element basis. The parent CLIST of the PLIST supplied as the value of this argument must be the source CLIST of the function.
<i>ang2</i>	This argument must be supplied if the source CLIST is three-dimensional. It is the plunge of the direction pertaining to <i>a_hmax</i> .
<i>ang2_plist</i>	This argument can be employed instead of the <i>ang2</i> argument if <i>ang2</i> can vary from element to element of the source CLIST. It provides the value of <i>ang2</i> on an element-by-element basis. The parent CLIST of the PLIST supplied as the value of this argument must be the source CLIST of the function.
<i>ang3</i>	This argument must be supplied if the source CLIST is three-dimensional. It is normally zero. For further details see below.
<i>ang3_plist</i>	This argument can be employed instead of the <i>ang3</i> argument if <i>ang3</i> can vary from element to element of the source CLIST. It provides the value of <i>ang3</i> on an element-by-element basis. The parent CLIST of the PLIST supplied as the value of this argument must be the source CLIST of the function.

Discussion

Variogram dimensions

If the source CLIST of the *build_covmat_from_vario()* function is two-dimensional, then arguments of this function must specify parameters of a two-dimensional variogram. Required arguments are *variogram*, *sill* and *a* (or the PLIST counterparts to *sill* and *a*). Optionally, *nugget*, *hanis* and *bearing* (or their PLIST counterparts) can also be provided.

On the other hand, if the source CLIST of the *build_covmat_from_vario()* function is three-dimensional, then arguments of this function must specify parameters of a three-dimensional variogram. Required arguments are *variogram* and *sill* (or the PLIST counterpart to the latter), and all of the *a_hmax*, *a_hmin*, *a_vert*, *ang1*, *ang2* and *ang3* arguments (or their PLIST counterparts). The *nugget* argument (or its PLIST counterpart) is optional.

Variogram properties

Properties and equations for two-dimensional variograms are provided with documentation of the *calc_kriging_factors_2d()* function.

A three-dimensional variogram requires specification of nine variables. These are supplied through the *variogram*, *nugget*, *sill*, *a_hmax*, *a_hmin*, *a_vert*, *ang1*, *ang2* and *ang3* arguments. Optionally the *nugget* argument can be omitted, in which case it is assumed to be zero.

Similarly to a two-dimensional variogram, a three-dimensional variogram can be spherical, exponential, Gaussian or power. *a_hmax*, *a_hmin* and *a_vert* provide variogram “a” values in the directions of maximum horizontal range, minimum horizontal range and roughly vertically respectively. The angles *ang1*, *ang2* and *ang3* define directions of geometric anisotropy of the three-dimensional variogram. Their roles are as follows.

- *ang1* defines the angle between north and the direction of maximum horizontal anisotropy in degrees clockwise;
- *ang2* defines the plunge of the direction of maximum anisotropy, that is the angle (positive downwards) between horizontal (in the direction defined by *ang1*) and the direction of actual maximum anisotropy;
- To quote Deutsch and Journel (1998) - “The third rotation angle *ang3* leaves the principal direction, defined by *ang1* and *ang2*, unchanged. The two directions orthogonal to that principal direction are rotated clockwise relative to the principal direction when looking toward the origin.” In the vast majority of cases *ang3* should be set to zero.

Spatially varying variograms

As is apparent from the above description of its arguments, the *build_covmat_from_vario()* function allows variogram properties to vary on a point by point basis. The manner in which the *build_covmat_from_vario()* function handles spatially varying variograms is very simple. The total variance assigned to any CLIST element is the sum of the nugget and the sill that are ascribed to that element. The covariance between them is calculated using the variogram characteristics ascribed to both CLIST elements; the lesser of these covariances is then adopted. Positive definiteness of the resulting matrix is then attained by subjecting it to

singular value decomposition and rebuilding it after equating the \mathbf{V} matrix to the \mathbf{U} matrix. These matrices emerge from singular value decomposition of the covariance matrix according to the following equation:

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^t$$

Examples

Example 1

```
covmat=cl_pp.build_covmat_from_vario(variogram=exp,          &
                                     a=40.0,                 &
                                     sill=1.0)
```

In this example a MATRIX named *covmat* pertaining to elements of the two-dimensional CLIST *cl_pp* is constructed using an exponential variogram with an *a* value of 40 and a *sill* of 1.0.

Example 2

```
covmat=cl_pp.build_covmat_from_vario(variogram=exp,      &
                                     a=40.0,             &
                                     sill=1.0,          &
                                     select=(pp zone==1))
```

This example is similar to the previous example. However in this case the CLIST elements for which the covariance matrix is constructed are limited to those for which elements of the *pp_zone* SLIST have a value of 1. The parent CLIST of the *pp_zone* SLIST must be *cl_pp*.

Example 3

```
covmat=cl_pp.build_covmat_from_vario(variogram=exp,      &
                                     a=40.0,              &
                                     sill=1.0,           &
                                     nugget=0.2,         &
                                     hanis=2.0,          &
                                     bearing=52)
```

This example is similar to Example 1. However in this example it is specified that a nugget of 0.2 be employed and that the horizontal anisotropy is 2.0. The bearing of the direction of maximum elongation of the variogram is 52 degrees (clockwise from north).

Example 4

[illegible]

In this example all variogram properties are supplied on an element-by-element basis using PLISTS. All cited PLISTs must have the same parent CLIST, namely *cl_pp*.

Example 5

```
covmat=cl_pp_3d.build_covmat_from_vario(                                &
                                variogram=exp,                          &
                                nugget plist=pp nugget,                 &
```

```
sill_plist=pp_sill,           &
a_hmax_plist = pp_a_hmax,     &
a_hmin_plist = pp_a_hmin,     &
a_vert = 10.0,                &
ang1_plist=pp_ang1,           &
ang2=0.0,                     &
ang3=0.0)
```

In this three-dimensional example some variogram parameters are provided on an element-by-element basis while others are supplied as a single value.

calc_kriging_factors_2d()

General

The *calc_kriging_factor_2d()* function writes a text or binary file which links elements of a source CLIST (and hence PLISTs for which it is the reference) to elements of a target CLIST (and hence PLISTs for which it is the reference). Linkage between the source and the target takes place through two-dimensional spatial interpolation implemented using kriging.

The file written by *calc_kriging_factor_2d()* records the factors by which element values of a source PLIST are multiplied before being added together to form element values of a target PLIST. Calculation of these kriging factors is based on *x* and *y* coordinates stored in respective reference CLISTs, together with variogram properties and other interpolation variables supplied through the *calc_kriging_factors_2d()* argument list.

The actual spatial interpolation process through which element values of one PLIST are calculated from those of another PLIST is carried out by the *krige_using_file()* function. This function can be re-used many times to undertake PLIST-to-PLIST interpolation based on the same set of kriging factors, provided that source and target PLISTs cited in each call to function *krige_using_file()* employ the same source and target CLISTs as those employed in calculation of kriging factors by the *calc_kriging_factors_2d()* function.

calc_kriging_factors_2d() employs a modified version of GSLIB subroutine *kb2d()* to calculate kriging factors. Much of the nomenclature used in the following description follows protocols employed in GSLIB documentation. See Deutsch and Journel (1998) for further details.

Note that the capabilities of the *calc_kriging_factors_2d* function are very similar to those of the PPK2FAC utility of the PEST Groundwater Data Utility suite.

Function Specifications

Function value

The *calc_kriging_factors_2d()* function makes no assignment to any PLPROC entity. Its name must lead the PLPROC command through which it is invoked.

Arguments and subarguments

<i>file</i>	The name of the file in which kriging factors are recorded.
<i>file; format</i>	<p>The <i>format</i> subargument of the <i>file</i> argument must be supplied as “formatted” or “binary”. In the former case an ASCII (i.e. text) file is written, whereas binary storage is employed in the latter case. (The user can employ “text” or “ascii” instead of “formatted” if he/she wishes, and “unformatted” instead of “binary” if he/she so desires.)</p> <p>If this subargument is omitted it is assumed to be “formatted”.</p>
<i>source_clist</i>	The name of the CLIST from which interpolation takes place.

<i>source_clist; select</i>	A <i>select</i> subargument of the <i>source_clist</i> argument restricts calculation of kriging factors to a subset of source CLIST elements based on a selection equation.
<i>target_clist</i>	The name of the CLIST to which interpolation takes place.
<i>target_clist; select</i>	A <i>select</i> subargument of the <i>target_clist</i> argument restricts calculation of kriging factors to a subset of target CLIST elements based on a selection equation.
<i>variogram</i>	The name of the variogram employed in calculation of kriging factors. Options are “spherical”, “exponential”, “gaussian” and “power”. The first two are recommended.
<i>a</i>	The “a” value of the variogram equation. This is closely related to the variogram range; see below.
<i>kriging</i>	The type of kriging undertaken. Options are “simple” and “ordinary”.
<i>anis_ratio</i>	The anisotropy ratio. If not supplied, this is assumed to be 1.0. This is the ratio of the variogram range in the direction given by <i>anis_bearing</i> to the variogram range in the direction at right angles to it. If the <i>anis_ratio</i> argument is supplied then the <i>anis_bearing</i> argument must also be supplied.
<i>anis_bearing</i>	The bearing in degrees (with north being zero degrees and clockwise rotation being positive) of the anisotropy axis. It is normal protocol for this to be the direction in which the variogram range is greatest and for <i>anis_ratio</i> to thereby be greater than 1.0.
<i>nugget</i>	The nugget associated with the variogram. If omitted (as is usually done) the nugget is assumed to be zero.
<i>sill</i>	The variogram sill. This argument is not required unless a nugget argument is supplied as, with a nugget of zero, kriging factors are independent of the sill of the variogram.
<i>search_radius</i>	In searching for elements of the source CLIST for which interpolation takes place to a particular element of the target CLIST, a source CLIST element will NOT be used if it is situated further from the target CLIST element than this distance. Provide a very large number (e.g. 10^{15}) for an effectively infinite search radius.
<i>max_points</i>	Regardless of the search radius, interpolation factors will be calculated for no more than the <i>max_points</i> closest source CLIST elements to any target CLIST element.
<i>min_points</i>	This is the minimum number of source CLIST elements which must be included within the <i>search_radius</i> centred on any target CLIST

```

        variogram=exponential,           &
        a=45000,                         &
        kriging=ordinary,                 &
        min_points=1,max_points=12,search_radius=1e15)

```

In this example kriging factors are calculated for interpolation from the *pilot_points* CLIST to the *modflow_grid* CLIST. Kriging employs an exponential variogram with a range of 45000 length units. For each target CLIST element, the closest 12 source CLIST elements must be used as a basis for interpolation. Calculated kriging factors are stored in text format in file *fac1.dat*.

Omission of the *anis_ratio* argument implies an anisotropy of 1.0. The *anis_bearing* argument is therefore not required. As no *nugget* argument is supplied, no *sill* argument is required.

Example 2

```

calc_kriging_factors_2d(target_clist=modflow_grid,           &
                        source_clist=pilot_points,           &
                        file=fac1.dat;form=binary,            &
                        variogram=spherical,                   &
                        a=45000,                               &
                        kriging=simple,                         &
                        nugget=0.2,sill=0.1,                   &
                        anis_bearing=30,anis_ratio=5.0,        &
                        min_points=1,max_points=12,search_radius=1e15)

```

In this example simple kriging is carried out. As the anisotropy ratio (*anis_ratio*) is not equal to unity, an *anis_bearing* argument is supplied.

Example 3

```

calc_kriging_factors_2d(
    target_clist=modflow_grid; select=(layer<=3),           &
    source_clist=pilot_points; select=(zone==2),             &
    file=fac1.dat,                                           &
    variogram=exponential,                                   &
    a=45000,                                                 &
    kriging=ordinary,                                         &
    min_points=1,max_points=12,search_radius=1e15)

```

This is a modification of example 1. In this case only source CLIST elements for which *zone* SLIST elements are 2 are employed in the interpolation process; *zone*'s reference CLIST must, of course, be *pilot_points*. Furthermore interpolation takes place only to elements of the *modflow_grid* CLIST for which values of the *layer* SLIST are less than or equal to 3. The reference CLIST for *layer* must, of course, be *modflow_grid*.

calc_kriging_factors_auto_2d()

General

The *calc_kriging_factors_auto_2d()* function is similar to the *calc_kriging_factors_2d()* function in that it calculates and records kriging factors for use of the *krige_using_file()* function. However it differs from *calc_kriging_factors_2d()* in a number of important respects, these being as follows:

- Variogram and variogram parameter selection are undertaken automatically;
- The parameters of the variogram used for interpolation vary spatially in response to density of points from which interpolation takes place, and are thus locally optimal;
- Anisotropy can vary spatially.

Because of these features, *calc_kriging_factors_auto_2d()* is easier to use than *calc_kriging_factors_2d()*. Also, its performance is normally better. See the “discussion” section below for more details.

Function Specifications

Function value

The *calc_kriging_factors_auto_2d()* function makes no assignment to any PLPROC entity. Its name must lead the PLPROC command through which it is invoked.

Arguments and subarguments

<i>file</i>	The name of the file in which kriging factors are recorded.
<i>file; format</i>	<p>The <i>format</i> subargument of the <i>file</i> argument must be supplied as “formatted” or “binary”. In the former case an ASCII (i.e. text) file is written, whereas binary storage is employed in the latter case. (The user can employ “text” or “ascii” instead of “formatted” if he/she wishes, and “unformatted” instead of “binary” if he/she so desires.)</p> <p>If this subargument is omitted it is assumed to be “formatted”.</p>
<i>source_clist</i>	The name of the CLIST from which interpolation takes place.
<i>source_clist; select</i>	A <i>select</i> subargument of the <i>source_clist</i> argument restricts calculation of kriging factors to a subset of source CLIST elements based on a selection equation.
<i>target_clist</i>	The name of the CLIST to which interpolation takes place.
<i>target_clist; select</i>	A <i>select</i> subargument of the <i>target_clist</i> argument restricts calculation of kriging factors to a subset of target CLIST elements based on a selection equation.
<i>kriging</i>	The type of kriging undertaken. Options are “simple” and “ordinary”.

<i>anis_ratio</i>	The anisotropy ratio. If not supplied, this is assumed to be 1.0. This is the ratio of the variogram range in the direction given by <i>anis_bearing</i> to the variogram range in the direction at right angles to it. If the <i>anis_ratio</i> argument is supplied then the <i>anis_bearing</i> argument must also be supplied.
<i>anis_bearing</i>	The bearing in degrees (with north being zero degrees and clockwise rotation being positive) of the anisotropy axis. It is normal protocol for this to be the direction in which the variogram range is greatest and for <i>anis_ratio</i> to thereby be greater than 1.0. <i>anis_bearing</i> can range between -360 degrees and 360 degrees.
<i>anis_ratio_plist</i>	The name of a PLIST whose parent CLIST is the <i>target_clist</i> . Elements of this PLIST denote the anisotropy ratio on an element-by-element basis. If the <i>anis_ratio_plist</i> argument is supplied then the <i>anis_bearing_plist</i> argument must also be supplied.
<i>anis_bearing_plist</i>	The name of a PLIST whose parent CLIST is the <i>target_clist</i> . Elements of this PLIST denote the anisotropy bearing on an element-by-element basis. If the <i>anis_bearing_plist</i> argument is supplied then the <i>anis_ratio_plist</i> argument must also be supplied.
<i>enforce_strong_anis</i>	This optional argument must have a value of “yes” or “no”; the default is “no”. If supplied as “yes”, the effect of anisotropy (global or local) on the interpolation process is stronger; coordinate transformation prior to searching for nearby data points ensures that search directions are also anisotropic. Execution can take longer with this option if anisotropy is spatially variable.

Discussion

Interpolation and implementation details

The first important difference between *calc_kriging_factors_auto_2d()* and *calc_kriging_factors_2d()* is that the former function requires much less input from the user. Hence *calc_kriging_factors_auto_2d()* has fewer arguments than *calc_kriging_factors_2d()*. The user does not supply a variogram; nor do any variogram properties need to be supplied (except perhaps anisotropy). *calc_kriging_factors_auto_2d()* automatically chooses the exponential variogram. Unless instructed otherwise, ordinary (rather than simple) kriging is undertaken.

The second major difference between *calc_kriging_factors_auto_2d()* and *calc_kriging_factors_2d()* is that the range of the variogram used by *calc_kriging_factors_auto_2d()* actually varies with location within the model domain. It is potentially different for each target CLIST element (normally model cells) to which interpolation must take place. It is smaller where elements of the source CLIST from which interpolation takes place (normally pilot points) are closer together, and larger where they are further apart. It also increases as the distance between the target point and the closest point from which interpolation takes place increases. This adaptation of interpolation parameters to

variations in source point density allows a user to place source (pilot) points close together where observation wells are close together, and further apart where observation well density is low. While this could be done even with a spatially uniform variogram, spatial interpolation may be suboptimal under these conditions as the spatial scale of variability that must be expressed through interpolation is often non-uniform.

Problems that accompany interpolation from pilot points with very different spatial densities within different parts of a model domain are compounded where a single search radius and uniform number of source points to use in the interpolation process must be selected. As source points move in and out of the search radius as interpolation shifts from target point to target point, discontinuities in the interpolated field may be encountered if the spatial correlation implied by the variogram is not very low between the target cells in question, and source points that move in and out of the search radius. This can create an ugly “paintbrush effect” in the interpolated field. Where the search radius and number of source interpolation points must accommodate vastly different source point spatial densities in different parts of a model domain, this can be very difficult to avoid.

calc_kriging_factors_auto_2d() attempts to overcome these difficulties. It commences the interpolation process by computing, for each source point, the distance to the nearest neighbouring source point. This provides a measure of source point density in different parts of the model domain. At each target point to which interpolation must take place it then computes the weighted average of these nearest neighbour source point separations, with the weighting function being the inverse distance squared from the target point to each source point. The “*a*” value for the exponential variogram is then computed as this spatially weighted nearest neighbour distance value, or as the distance from the target cell to the nearest source point, whichever is the larger. The local search radius is then set to 4 times the “*a*” value.

If no variogram anisotropy ratio is supplied, this is assumed to be 1.0. Alternatively, a uniform anisotropy ratio and corresponding compass bearing of maximum variogram length can be supplied as *calc_kriging_factors_auto_2d()* subroutine arguments. Another option provided by the *calc_kriging_factors_auto_2d()* function is for the user to supply PLISTS of anisotropy range and bearing. The parent CLIST for these PLISTS must be the target CLIST to which interpolation takes place. These PLISTS may be the outcomes of previous interpolation exercises, possibly conducted with *calc_kriging_factors_auto_2d()* in conjunction with *krige_using_file()*. Hence anisotropy can vary throughout a model domain in order to reflect variations in the strike of structural features which are known to affect model properties. The direction and magnitude of anisotropy may even be an estimable parameter, with a specific family of pilot points dedicated to their representation.

Calculations undertaken by the *calc_kriging_factors_auto_2d()* function are more complex than those undertaken by the *calc_kriging_factors_2d()* function. Hence execution times may be a little longer for this function. Also, the factor file written by *calc_kriging_factors_auto_2d()* may be larger than that written by *calc_kriging_factors_2d()*. However experience to date suggests that the costs (if any) incurred through extra computation time and file size are worth the trouble because of the smoother fields that are forthcoming from the interpolation process, and because of the ability of these fields to respond flexibly to variations in pilot point density and/or variations in local anisotropy throughout a model domain.

Simple and ordinary kriging

Simple kriging requires that a mean value for the interpolated variable be supplied. As kriging factors are independent of this variable, function *calc_kriging_factors_auto_2d()* does not require it. Instead it is provided to function *krige_using_file()* which uses kriging factors computed by *calc_kriging_factors_auto_2d()* together with source PLIST data to calculate interpolated values for target PLIST elements.

It should be noted however, that simple kriging should be used with caution in the *calc_kriging_factors_auto_2d()* function, as automatic calculation of the spatially-varying variogram range may lead to a tendency for interpolated parameter fields to approach the user-supplied mean value in too many places.

Strong enforcement of anisotropy

If the optional *enforce_strong_anis* argument of the *calc_kriging_factors_auto_2d()* function is set to “yes” (the default value is “no”), then the effect of anisotropy on the spatial interpolation process becomes more pronounced. Not only are covariances between points a function of anisotropy, but the selection of points used in the interpolation process also becomes a function of anisotropy. Prior to selection of local data points for interpolation to a field point, coordinates are transformed in accordance with local anisotropy. A data point from which interpolation must take place becomes “closer” to a field point if the line between the two corresponds with the major axis of anisotropy. In some circumstances (for example in interpolation down narrow alluvial valleys where anisotropy can change with the direction of the river), this can have a favourable impact on the interpolation process. However where anisotropy is spatially variable, the numerical burden can become high as regular re-computing of transformed coordinates is then required.

Examples

Example 1

```
calc_kriging_factors_auto_2d(target_clist=musg_grid,      &
                             source_clist=cl_pp,         &
                             file=factors.dat)
```

This example shows use of function *calc_kriging_factors_auto_2d()* with a minimal number of arguments.

Example 2

```
calc_kriging_factors_auto_2d(
    target_clist=musg_grid;select=(kx_node_zones==201),      &
    source_clist=cl_pp;select=(kx_pp_zones==201),             &
    file=factors.dat,                                          &
    anis_ratio=2.0,anis_bearing=315)
```

This example adds complexity to the previous one by including source and target CLIST selection equations, as well as a constant anisotropy ratio and bearing.

Example 3

```
calc_kriging_factors_auto_2d(
    target_clist=musg_grid;select=(kx_node_zones==201),      &
    source_clist=cl_pp;select=(kx_pp_zones==201),             &
    file=factors201.dat,                                       &
```

```
anis_ratio_plist=anis_ratio, &  
anis_bearing_plist=anis_bearing)
```

This example is identical to the above example except for the fact that anisotropy ratio and direction are spatially varying.

Example 4

```
calc_kriging_factors_auto_2d(  
  target_clist=musg_grid;select=(kx_node_zones==201), &  
  source_clist=cl_pp;select=(kx_pp_zones==201), &  
  file=factors201.dat, &  
  anis_ratio_plist=anis_ratio, &  
  anis_bearing_plist=anis_bearing, &  
  enforce_strong_anis='yes')
```

In this example the effect of (spatially varying) anisotropy on the interpolation process is made more pronounced as not just spatial correlations, but also search specifications, become a function of anisotropy. This is effected through use of the *enforce_strong_anis* argument.

calc_kriging_factors_3d()

General

The *calc_kriging_factor_3d()* function writes a text or binary file which links elements of a source CLIST (and hence PLISTs for which it is the reference) to elements of a target CLIST (and hence PLISTs for which it is the reference). Linkage between the source and the target is through three-dimensional spatial interpolation implemented using kriging. Both the source and target CLISTs employed by *calc_kriging_factors_3d()* must therefore be three-dimensional CLISTs.

The file written by *calc_kriging_factor_3d()* records the factors by which element values of a source PLIST are multiplied before being added together to form element values of a target PLIST. Calculation of these kriging factors is based on *x*, *y* and *z* coordinates attributed to respective reference CLIST elements, together with variogram properties and other interpolation variables supplied through the *calc_kriging_factors_3d()* argument list.

The actual spatial interpolation process through which element values of one PLIST are calculated from those of another PLIST is carried out by the *krige_using_file()* function. This function can be re-used many times to undertake PLIST-to-PLIST interpolation based on the same set of kriging factors, provided that source and target PLISTs cited in each call to function *krige_using_file()* employ the same source and target CLISTs as those employed in calculation of kriging factors by the *calc_kriging_factors_3d()* function.

calc_kriging_factors_3d() employs a modified version of GSLIB subroutine *kt3d()* to calculate kriging factors. Much of the nomenclature used in the following description follows protocols employed in GSLIB documentation. See Deutsch and Journel (1998) for details.

Note that the capabilities of the *calc_kriging_factors_3d* function are very similar to those of the PPK2FAC3D utility of the PEST Groundwater Data Utility suite.

Function Specifications

Function value

The *calc_kriging_factors_3d()* function assigns a value to no PLPROC entity. Its name must lead the PLPROC command from which it is invoked.

Arguments and subarguments

<i>file</i>	The name of the file to which kriging factors are recorded.
<i>file; format</i>	<p>The <i>format</i> subargument of the <i>file</i> argument must be supplied as “formatted” or “binary”. In the former case an ASCII (i.e. text) file is written, whereas binary storage is employed in the latter case. (The user can employ “text” or “ascii” instead of “formatted” if he/she wishes, and “unformatted” instead of “binary” if he/she so desires.)</p> <p>If the <i>format</i> subargument is omitted, it is assumed to be “formatted”.</p>

<i>source_clist</i>	The name of the CLIST from which interpolation takes place.
<i>source_clist; select</i>	A <i>select</i> subargument of the <i>source_clist</i> argument restricts calculation of kriging factors to a subset of source CLIST elements based on a selection equation.
<i>target_clist</i>	The name of the CLIST to which interpolation takes place.
<i>target_clist; select</i>	A <i>select</i> subargument of the <i>target_clist</i> argument restricts calculation of kriging factors to a subset of target CLIST elements based on a selection equation.
<i>variogram</i>	The name of the variogram employed in calculation of kriging factors. Options are “spherical”, “exponential”, “gaussian” and “power”. The first two are recommended.
<i>kriging</i>	The type of kriging undertaken. Options are “simple” and “ordinary”.
<i>nugget</i>	The nugget associated with the variogram. If omitted (as is usually done) it is assumed to be zero.
<i>sill</i>	The variogram sill. This is not required unless a <i>nugget</i> argument is employed, as kriging factors are independent of the variogram sill where the nugget is zero.
<i>a_hmax</i>	Value of variogram <i>a</i> parameter in direction of maximum horizontal <i>a</i> .
<i>a_hmin</i>	Value of variogram <i>a</i> parameter in direction of minimum horizontal <i>a</i> .
<i>a_vert</i>	Value of variogram <i>a</i> parameter in vertical direction.
<i>ang1</i>	Defines the angle between north and the direction of maximum horizontal anisotropy in degrees clockwise.
<i>ang2</i>	Defines the negative of the plunge of the direction of maximum anisotropy, that is the angle (positive upwards) between horizontal (in the direction defined by <i>ang1</i>) and the direction of actual maximum anisotropy.
<i>ang3</i>	To quote Deutsch and Journel (1998) - “ <i>ang3</i> leaves the principal direction defined by <i>ang1</i> and <i>ang2</i> unchanged. The two directions orthogonal to that principal direction are rotated clockwise relative to the principal direction when looking toward the origin.” In the vast majority of cases <i>ang3</i> should be set to zero. This is its default value if this argument is omitted.

<i>search_rad_max_hdir</i>	Search radius in direction of maximum horizontal elongation.
<i>search_rad_min_hdir</i>	Search radius in direction of minimum horizontal elongation.
<i>search_rad_vert</i>	Search radius in vertical direction.
<i>max_points</i>	Regardless of the search radius, interpolation factors associated with a particular target CLIST element will be calculated for no more than the <i>max_points</i> closest source CLIST elements.
<i>min_points</i>	The minimum number of source CLIST elements which must be included in the search volume (defined by the three search radii) centred on any target CLIST element. If the search volume is not large enough to include this number of source CLIST elements, PLPROC will report an error condition and then cease execution.

Discussion

Variograms

Equations for the four types of variogram supported by the *calc_kriging_factors_3d()* function are provided in documentation of the *calc_kriging_factors_2d()* function.

Simple and ordinary kriging

Simple kriging requires that a mean value for the interpolated variable be supplied. As kriging factors are independent of this variable, function *calc_kriging_factors_3d()* does not require it. Instead it is provided to function *krige_using_file()* which uses kriging factors computed by *calc_kriging_factors_3d()* together with source PLIST data to calculate interpolated values for target PLIST elements.

Examples

Example 1

```
calc_kriging_factors_3d(target_clist=model_nodes,           &
                        source_clist=pilot_points,         &
                        file='fac1.dat',                   &
                        variogram=exponential,              &
                        a_hmax=50000,a_hmin=20000,a_vert=500, &
                        ang1=45,ang2=20,                   &
                        kriging=ordinary,                   &
                        search_rad_max_hdir=1e15,           &
                        search_rad_min_hdir=1e15,           &
                        search_rad_vert=1e15,               &
                        min_points=1,max_points=50)
```

Kriging factors computed through the above *calc_kriging_factors_3d()* function call are recorded in a text file named *fac1.dat*. Note that this file may be very large if the model grid is large because of the *max_points=50* argument.

Example 2

```
calc_kriging_factors_3d(  
    target_clist=model_nodes;select=(layer==4),      &  
    source_clist=pilot_points;select=(zone==1),      &  
    file='fac1.dat';format='binary',                 &  
    variogram=exponential,                            &  
    a_hmax=50000,a_hmin=20000,a_vert=500,            &  
    angl=45,ang2=20,                                  &  
    kriging=ordinary,                                  &  
    search_rad_max_hdir=1e15,                          &  
    search_rad_min_hdir=1e15,                          &  
    search_rad_vert=1e15,                              &  
    min_points=1,max_points=50)
```

In this example calculation of kriging factors is restricted to those elements of the target CLIST for which element values of the *layer* SLIST are equal to 4, and to those elements of the source CLIST for which element values of the *zone* SLIST are equal to 1. It is presumed that *model_nodes* is the parent CLIST of the *layer* SLIST and that *pilot_points* is the parent CLIST of the *zone* SLIST.

calc_linear_interp_factors()

General

Function *calc_linear_interp_factors()* calculates interpolation factors that implement piecewise-constant or linear interpolation from a PLIST whose parent CLIST is linked to a SEGLIST. The PLIST to which interpolation takes place is arbitrary. However those elements of its parent CLIST to which interpolation should take place should lie in close proximity to segments of the SEGLIST.

The interpolaton methodology is described in the “discussion” section below.

Once interpolation factors have been calculated, they can be used by either of functions *interp_using_file()* or *krige_using_file()* to carry out PLIST-to-PLIST interpolation. (These functions are identical.)

As presently programmed, the CLIST from which interpolation takes place (i.e. the source SEGLIST-linked CLIST) must be a 2D CLIST. The CLIST to which interpolation takes place (i.e. the target CLIST) can be either a 2D CLIST or a 3D CLIST.

Function Specifications

Function value

The *calc_linear_interp_factors()* function makes no assignment to any PLPROC entity. Its name must lead the PLPROC command through which it is invoked.

Arguments and subarguments

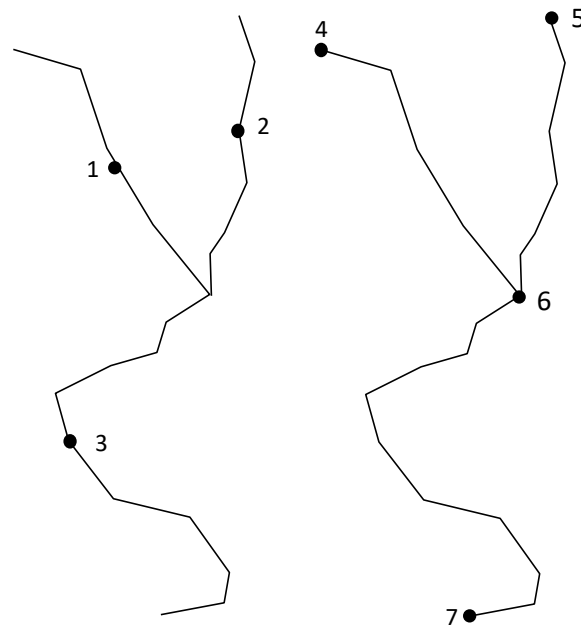
<i>file</i>	The name of the file in which interpolation factors calculated by <i>calc_linear_interp_factors()</i> are recorded.
<i>file; format</i>	<p>The <i>format</i> subargument of the <i>file</i> argument must be supplied as either “formatted” or “binary”. In the former case an ASCII (i.e. text) file is written, whereas binary storage is employed in the latter case. (The user can employ “text” or “ascii” instead of “formatted” if he/she wishes, and “unformatted” instead of “binary” if he/she so desires.)</p> <p>If this subargument is omitted, it is assumed to be “formatted”.</p>
<i>source_clist</i>	The name of the CLIST from which interpolation takes place. Note that it is illegal for a selection equation subargument to accompany the <i>source_clist</i> argument, for this would interfere with SEGLIST-to-CLIST linkages which form the basis for piecewise-constant or linear interpolation undertaken by this function. Note also that PLPROC will cease execution with an error message if the source CLIST is not linked to a SEGLIST.
<i>target_clist</i>	The name of the CLIST to which interpolation takes place.
<i>target_clist; select</i>	A <i>select</i> subargument of the <i>target_clist</i> argument restricts calculation

	of interpolation factors to a subset of target CLIST elements based on a selection equation.
<i>dimensions</i>	The value of this argument must be either 2 or 3. It specifies whether distance between a target CLIST element location and a SEGLIST segment is calculated as the horizontal distance or actual distance (for the case of a 3D target CLIST). As presently programmed, the value of this argument must be 2. If it is omitted, then a value of 2 is assumed.
<i>search_radius</i>	If the distance between a target CLIST element and the closest segment of the SEGLIST to which the source CLIST is linked is greater than this distance, PLPROC ceases execution with an error message. If omitted, the value of this argument is set to a very high number (effectively infinity).

Discussion

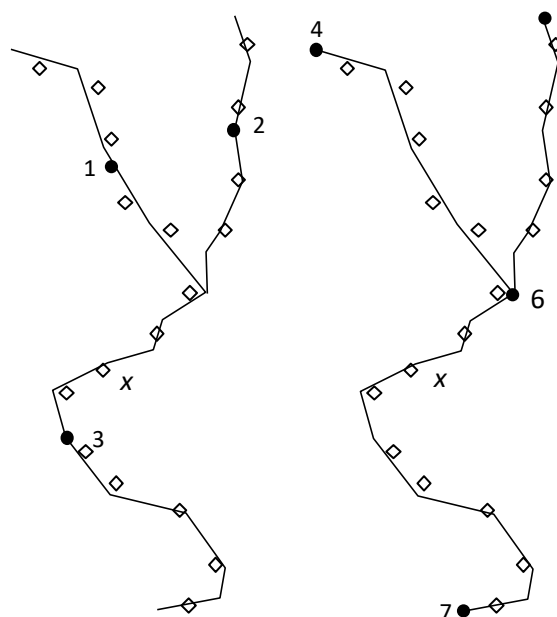
Linkage of a SEGLIST to a CLIST

As is described elsewhere in this manual, functions *link_seglist_to_clist()* and *create_clist_from_seglist()* link a SEGLIST to a CLIST in one of two ways. The first option is to link each segment of the SEGLIST to a single element of the CLIST; presumably the latter point lies somewhere near the midpoint of the segment. (Note that a segment may be comprised of multiple linear sectors.) The second option is to link each segment of the SEGLIST to two elements of the CLIST; presumably each of the points corresponding to these elements is located near opposite ends of the segment. In the latter case, one or both of these CLIST elements can also be linked to an adjacent segment of the SEGLIST. These two options are depicted in the following figure. In this figure, the locations of CLIST elements are depicted by round dots. In the discussion that follows, those on the left of the figure will be ascribed to *clist1* while those on the right of the figure will be ascribed to *clist2*.



A SEGLIST linked to elements of two CLISTS in two different ways. The first linkage type (on the left) employs the “midpoint” option, while the second linkage type (on the right) employs the “endpoints” option. SEGLIST-to-CLIST linkage is made using the `link_seglist_to_clist()` and `create_clist_from_seglist()` functions.

Consider now a series of points belonging to another CLIST to which interpolation must take place; we shall identify this CLIST as *clist3*. The *clist3* elements to which interpolation must take place are depicted by open diamonds in the figure below. These points may lie at the centres of certain cells of a model grid. They may have been identified using the PLPROC function `find_cells_in_lists()`. As such, they may represent cells to which a MODFLOW *drain*, *ghb*, *river* or similar type of boundary condition has been ascribed. Meanwhile, the elements of *clist1* and *clist2* may be visualized as the locations of pilot points from which piecewise constant or linear interpolation down SEGLIST segments to the centres of these model cells must take place. The quantities that are interpolated in this manner may be boundary condition properties such as elevation and conductance; for the purposes of model parameterization (and estimation of model parameters) the values of these properties may be ascribed to PLISTs associated with the *clist1* and *clist2* CLISTS.



The open diamonds indicate points to which SEGMENT-based piecewise constant or linear interpolation must take place.

Consider first interpolation from a PLIST whose parent CLIST is *clist1* to a PLIST whose parent CLIST is *clist3*. Function *calc_linear_interp_factors()* finds the closest sector of the closest SEGLIST segment to each element of *clist3* to which interpolation must take place. It then assigns the *clist3* PLIST element a value that is equal to that of the *clist1* PLIST element to which the segment is linked. Thus where segment-to-CLIST linkage is of the “midpoint” type, interpolation from the SEGMENT-linked source CLIST to the target CLIST is piecewise constant. So, for example, the PLIST element corresponding to the *clist3* element located at position *x* in the above figure is assigned the value of the PLIST element corresponding to *clist1* element number 3.

Now consider interpolation from a PLIST whose parent CLIST is *clist2* to a PLIST whose parent CLIST is *clist3*. In similar fashion to the above, PLPROC first establishes the SEGLIST segment to which each target PLIST element is closest. Interpolation from the *clist2* PLIST to the *clist3* PLIST is then linear along the segment. In this case the value assigned to the PLIST element corresponding to *clist3* element *x* in the above example is calculated from PLIST elements associated with *clist2* elements 6 and 7. Interpolation is linear in terms of distance along the length of the segment to the point that is closest to *x*, taking the length of each sector comprising the segment into account.

In both of the above cases, PLPROC commences the interpolation process by finding the SEGLIST segment to which a target interpolation point is closest. The distance between an arbitrary point and a SEGLIST is the length of a line that is perpendicular to the nearest sector of the nearest segment of the SEGLIST, or that joins the arbitrary point to a vertex of the SEGLIST, whichever is closer. If this distance exceeds the user-specified search radius, then PLPROC ceases execution with an appropriate error message.

Examples

Example 1

```
calc_linear_interp_factors(source_clist=cl_ep,           &
                           target_clist=cl_pi,         &
                           file="factors_ep.dat")
```

This example shows use of function *calc_linear_interp_factors()* with a minimal number of arguments. Presumably the *cl_ep* CLIST has been previously linked to a SEGLIST using function *link_seglist_to_clist()*, or has been created specifically for linkage to this SEGLIST using the *create_clist_from_seglist()* function.

Example 2

```
calc_linear_interp_factors(source_clist=cl_ep,           &
                           target_clist=cl_pi;select=(selghb.eq.1), &
                           file="factors_ep.dat";form=binary,      &
                           search_radius=500)
```

In the above example, *selghb* is an SLIST whose parent CLIST is *cl_pi*. Interpolation is restricted to target CLIST elements for which corresponding *selghb* SLIST elements are one. An error condition will arise if any selected target CLIST element is more than 500 units from the nearest segment of the SEGLIST to which the *cl_ep* CLIST is linked. Interpolation factors are stored in a binary file named *factors_ep.dat*.

calc_rbf_factors_2d()

General

Function *calc_rbf_factors_2d()* implements the first stage of two-part radial basis function interpolation. Using the information provided by this function, interpolation from many source PLISTs to many target PLISTs can be undertaken with smaller numerical burden than would otherwise be the case, provided all such source and target PLISTs share a common CLIST. One call to the *calc_rbf_factors_2d()* function for calculation of the numerical prerequisites for interpolation can then be followed by many calls to the *rbf_using_file()* function for carrying out the actual interpolation.

Because interpolation is based on two-dimensional radial basis functions, the *z* coordinates associated with both source and target CLIST elements are ignored in calculating interpolation information.

Function Specifications

Function value

The *calc_rbf_factors_2d()* function makes no assignment to any PLPROC entity. Its name must lead the PLPROC command through which it is invoked.

Arguments and subarguments

<i>file</i>	The name of the file in which information calculated by <i>calc_rbf_factors_2d()</i> is recorded.
<i>file; format</i>	The <i>format</i> subargument of the <i>file</i> argument must be supplied as either “formatted” or “binary”. In the former case an ASCII (i.e. text) file is written, whereas binary storage is employed in the latter case. (The user can employ “text” or “ascii” instead of “formatted” if he/she wishes, and “unformatted” instead of “binary” if he/she so desires.) If this subargument is omitted it is assumed to be “formatted”.
<i>source_clist</i>	The name of the CLIST from which interpolation takes place.
<i>source_clist; select</i>	A <i>select</i> subargument of the <i>source_clist</i> argument restricts calculation of interpolation information to a subset of source CLIST elements based on a selection equation.
<i>target_clist</i>	The name of the CLIST to which interpolation takes place.
<i>target_clist; select</i>	A <i>select</i> subargument of the <i>target_clist</i> argument restricts calculation of interpolation information to a subset of target CLIST elements based on a selection equation.
<i>rbf</i>	The name of the radial basis function type used for spatial interpolation. The value supplied for this argument must be one of “ga”, “iq”, “imq”, “mq”, “lin”, “cub” or “tps”. See the discussion below and in

	documentation of the <i>rbf_interpolate_2d()</i> function.
<i>rbf; epsilon</i>	The value of the “epsilon” parameter employed by some radial basis function types. Choice of a suitable value for the <i>epsilon</i> subargument of the <i>rbf</i> argument may be critical to the success of radial basis function interpolation. The “epsilon” and “epsminsepfac” subarguments are mutually exclusive.
<i>rbf; epsminsepfac</i>	The factor by which the distance to the closest neighbouring source PLIST member is multiplied in determining the reciprocal of the value of the “epsilon” parameter employed by some radial basis function types. The “epsilon” and “epsminsepfac” subarguments are mutually exclusive.
<i>constant_term</i>	The value for this optional argument must be supplied as “yes” or “no”. This determines whether the radial basis function will be augmented with a single, estimated, domain-wide additive term. If this argument is omitted, its value is assumed to be “no”.
<i>linear_term</i>	The value for this optional argument must be supplied as “yes” or “no”. This determines whether the radial basis function will be augmented with linear-with-distance additive terms applied in orthogonal directions. If this argument is omitted, it is assumed to be “no”.
<i>anis_ratio</i>	The anisotropy ratio. If not supplied, this is assumed to be 1.0. This is the ratio of system property elongation in the direction given by <i>anis_bearing</i> to that in the direction at right angles to it. If an <i>anis_ratio</i> argument is supplied then an <i>anis_bearing</i> argument must also be supplied.
<i>anis_bearing</i>	The bearing (with north being zero degrees) of the anisotropy axis. It is normal protocol for this to be the direction in which system property elongation is greatest and for <i>anis_ratio</i> to thereby be greater than 1.0.

Discussion

Refer to documentation of the *rbf_interpolate_2d()* function for specifications of the equations used in two-dimensional radial basis function interpolation. Refer to *rbf_interpolate_2d()* documentation also for a discussion of many of the arguments of the *calc_rbf_factors_2d()* function; arguments of the same name serve the same purpose in both of these functions.

No PLISTs feature as *calc_rbf_factors_2d()* arguments. Instead, only CLISTs are cited. The information calculated and recorded by this function is independent of the values which must be interpolated. The actual interpolation is carried out by the *rbf_using_file()* function. It is therefore incumbent on the user to ensure that the source PLIST cited in a subsequent call to the *rbf_using_file()* function employs the source CLIST cited in a prior *calc_rbf_factors_2d()* function call as its reference CLIST. Similar considerations apply to the target PLIST.

Where interpolation must be undertaken from a suite of source PLISTs with a common reference CLIST to a suite of target PLISTs with another common reference CLIST many calculations must be repeated. In order to avoid such repetition, the *calc_rbf_factors_2d()* function can be used to undertake calculations which these multiple interpolation processes have in common, thus reducing the overhead of repetitive PLIST-to-PLIST interpolation. Unfortunately, however, numerical overhead is replaced by file writing and reading overhead. Hence while two-step radial basis function interpolation involving a single call to function *calc_rbf_factors_2d()* followed by multiple calls to function *rbf_using_file()* may promote efficiency in some situations, it may not do so in all situations

The *calc_rbf_factors_2d()* function undertakes the following aspects of radial basis function computation. See documentation of *rbf_interpolate_2d()* for details.

- Filling of the **A** matrix;
- Factorization of the **A** matrix so that calculation of any **c** from any **f** is readily accomplished with only a small numerical burden;
- Calculation of the radial basis function value at all target points arising from a unit coefficient value at all source points, with anisotropy taken into account. Hence subsequent PLIST-to-PLIST interpolation requires only the multiplication of these pre-evaluated functions by pertinent coefficients. (Unfortunately there is no option but to compute these coefficients during the second stage of radial basis function interpolation as they are dependent on sampled data values.)

The outcomes of *calc_rbf_factors_2d()* calculations are stored in a file, the name of which is provided by the user as a function argument. This file can be large. Considerable time can be save in storing and retrieving data from this file if the “binary” rather than “formatted” option is chosen for its type.

Note that, unlike the *rbf_interpolate_2d()* function, the *calc_rbf_factors_2d()* function does not allow recording of information related to calculation of radial basis function coefficients in a report file.

Examples

Example 1

```
calc_rbf_factors_2d(target_clist=modflow_grid,           &
                   source_clist=cl_pp,                 &
                   file='fac1.dat';form=binary,        &
                   rbf=mq;epsilon=5e-5)
```

In this example the *calc_rbf_factors_2d()* function is asked to undertake the numerical prerequisites for interpolation from PLISTs whose reference CLIST is *cl_pp* to PLISTs whose reference CLIST is *modflow_grid*. The multiquadratic radial basis function is employed. Outcomes of *calc_rbf_factors_2d()* calculations are stored in a binary file named *fac1.dat* for later use by the *rbf_using_file()* function.

Example 2

```
calc_rbf_factors_2d(target_clist=modflow_grid,           &
                   source_clist=cl_pp,                 &
                   file='fac1.dat';form='binary',       &
                   anis_ratio=3,anis_bearing=60,       &
                   rbf=mq;epsilon=5e-5,               &
```

```

        constant_term='yes',
        linear_term='yes')

```

In this call to function *calc_rbf_factors_2d()* the multiquadratic radial basis function is augmented with a constant term and linear (with distance) drift. Also, anisotropy of the interpolated system property or state is assumed to exist, with the direction of principle anisotropy having a bearing of 60 degrees clockwise from north; the anisotropy ratio is 3.0.

Example 3

```

calc_rbf_factors_2d(target_clist=modflow_grid;select=(layer==4), &
                    source_clist=cl_pp;select=(zone==1),      &
                    file='fac1.dat';form='binary',             &
                    anis_ratio=3,anis_bearing=60,              &
                    rbf=mq;epsilon=5e-5,                       &
                    constant_term='yes',                        &
                    linear_term='yes')

```

This is identical to the previous example except for the fact that interpolation from source CLIST elements is restricted to those for which the value of corresponding *zone* SLIST elements is 1. Meanwhile interpolation to target CLIST elements is restricted to those for which the value of the *layer* SLIST is 4.

Example 4

```

calc_rbf_factors_2d(target_clist=modflow_grid;select=(layer==4), &
                    source_clist=cl_pp;select=(zone==1),      &
                    file='fac1.dat';form='binary',             &
                    anis_ratio=3,anis_bearing=60,              &
                    rbf=mq;epsilon=5e-5,                       &
                    constant_term='yes',                        &
                    linear_term='yes')

```

This example is identical to the previous example except that ε is calculated on a source PLIST element by source PLIST element basis using the *epsminsepfac* subargument of the *rbf* argument. ε for each source PLIST element is calculated as the reciprocal of 0.8 times the distance between that element and its nearest neighbour.

create_clist_from_seglist()

General

Linkage of a CLIST to a SEGLIST allows pilot points to be employed for parameterization of linear model features such as boundary conditions that represent rivers and streams. A CLIST can be linked to a SEGLIST in either of two ways. Each element of the CLIST can be linked to each segment of the SEGLIST on a one-to-one basis. Normally, for ease of recognition, each CLIST element should lie close to the midpoint of its partnered SEGLIST segment, in close proximity to the latter's trace. Alternatively, each SEGLIST segment may be linked to two CLIST elements; presumably, these two CLIST elements should lie near either end of the SEGLIST segment to which they are linked. If SEGLIST segments meet, a single CLIST element may thus be linked to more than one SEGLIST segment.

Function *create_clist_from_seglist()* designs and creates a CLIST specifically for the purpose of linking it to a SEGLIST. Where SEGLIST-to-CLIST linkage is one-to-one (this is the "midpoint" option discussed below), each element of the new CLIST is placed at the midpoint of the trace of the SEGLIST segment to which it is linked. Where SEGLIST-to-SEGLIST linkage is of the alternative type (this is the "endpoints" option discussed below), then an element of the new CLIST is placed at the end of each segment of the SEGLIST. If the ends of two or more segments are separated by less than a user-nominated distance, only a single CLIST element is created, this being linked to all SEGLIST segments which terminate at, or close to, that point.

Function Specifications

Function Value

As its name implies, the *create_clist_from_seglist()* function creates a new CLIST. The name of the new CLIST is the target of the function. This name must appear before the function in a PLPROC command which uses the function. This name should be separated from the function by a "=" symbol.

Arguments and subarguments

<i>seglist</i>	Provide the name of an existing SEGLIST. A SEGLIST can be linked to a maximum of two CLISTS; hence the nominated SEGLIST can be linked to, at most, one SEGLIST before a call is made to function <i>create_clist_from_seglist()</i> .
<i>linkage_type</i>	The value of this argument must be supplied as either "endpoints" (or "ends" for short) or "midpoint" (or "mid" for short). It specifies the nature of SEGLIST-to-CLIST linkage. The argument name can be specified as <i>linkage</i> for short in a <i>create_clist_from_seglist()</i> function call.
<i>dist_thresh</i>	If the specified <i>linkage_type</i> is "endpoints", a <i>dist_thresh</i> argument must be supplied; alternatively, if the <i>linkage_type</i> is "midpoint" then it is ignored if it is supplied. If the beginning or end of one SEGLIST segment is closer than <i>dist_thresh</i> from the beginning or end of another

	SEGLIST segment, then only a single CLIST element is introduced to link with both of these. It is placed at the beginning/end of one of the converging SEGLIST segments.
<i>dimensions</i>	As presently programmed, if this optional argument is not set to 2, PLPROC will cease execution with an error message.
<i>reportfile</i>	PLPROC can record a file that tabulates the outcomes of the CLIST creation process and the SEGLIST-to-CLIST linkage process that immediately follows it. If this file is required, its name should be provided as the value of the optional <i>reportfile</i> argument.

Discussion

For more details of SEGLIST-to-CLIST linkage, and of the use of this linkage in interpolation from pilot points down linear model elements, see documentation of functions *link_seglist_to_clist()*, *interp_using_file()*, *find_cells_in_lists()* and *replace_cells_in_lists()*.

Coordinates of the newly-created CLIST can be obtained using the *report_dependent_lists()* function. The file that is written by this function can be used to construct a file in which extra columns provide values for PLIST elements. These can pertain to pilot points co-located with elements of the newly-created CLIST. A template of this file can then be made for the use of PEST. PLIST values recorded in this way can be read using PLPROC functions such as *read_list_file()*.

Examples

Example 1

```
cl_ep1=create_clist_from_seglist(seglist=rivers,           &
                                linkage_type=endpoints,   &
                                reportfile="report3.dat",  &
                                dist_thresh=10.0)
```

A CLIST named “cl_ep1” is created. An element of the new CLIST is placed at the beginning and end of each segment of the “rivers” SEGLIST. However if the beginning/end of one SEGLIST segment is closer than 10m to the beginning/end of another SEGLIST segment, then only a single CLIST element is created. Once it has been created, the CLIST is then linked to the SEGLIST using the “endpoints” protocol. A report of this process is provided in file “report3.dat”.

Example 2

```
cl_mp1=create_clist_from_seglist(seglist=rivers,           &
                                linkage_type=midpoints)
```

A CLIST named “cl_mp1” is created. This CLIST has as many elements as there are segments in the “rivers” SEGLIST. Each element of the newly-created CLIST lies at the midpoint of a segment of the “rivers” SEGLIST.

find_cells_in_lists()

General

Function *find_cells_in_lists()* is designed to be used in conjunction with other functions such as *read_mf6_grid_specs()*, *calc_linear_interp_factors()* and *replace_cells_in_lists()*. It reads a model input file in which cell properties are provided in one or more lists. The MODFLOW family of models uses this protocol extensively. Property lists may be repeated many times within a single model input file - once for each stress period. It is assumed that PLPROC already holds a CLIST in its memory, the elements of which represent a model grid. *find_cells_in_lists()* builds an SLIST belonging to this CLIST in which elements corresponding to cells cited in property lists are awarded a value of 1; other cells are awarded a value of zero. Calculation of interpolation factors by functions such as *calc_linear_interp_factors()* and *calc_kriging_factors_auto_2d()* can then be restricted to these cells.

Optionally, a selection equation can be used to prevent elements of the new SLIST from being assigned values of 1 if they do not satisfy certain criteria.

Function Specifications

Function value

The *find_cells_in_lists()* function creates a new SLIST. This SLIST is the output of the function. Its name must be placed prior to the name of the function in front of an “=” symbol. Optionally, a selection equation can be placed between the function and the “=” symbol.

Following the “=” symbol must be the name of a CLIST. This name must be followed by a dot and then by the name of the function, and then by its arguments. The elements of the CLIST must pertain to the cells of a model.

Arguments

<i>file</i>	Provide the name of a model input file which includes one or a number of tables containing lists of cells and accompanying property values.
<i>keytext_start</i>	It is assumed that lists of cells in a model input file are collected into blocks (i.e. tables). These blocks are identified by an easily recognized text string on the line preceding the block in the model input file. At least part of that text string (enough for unique recognition of the text string) should be provided as the value of the <i>keytext_start</i> argument. If it contains a space, then the text string should be surrounded by quotes. PLPROC will object if the text string is blank. The search for this string in a model input file is case-insensitive.
<i>keytext_end</i>	This text string marks the end of a property list table within a model input file. Optionally, it can be blank.
<i>blocks_to_read</i>	A model input file may be very lengthy. Cell property tables within that file may be repeated many times. Alternatively, a cell property table may

	occur just once, near the beginning of the file. Or the first table may contain all cells that are cited in all other tables contained in the file. In both of these latter cases, it is not necessary for PLPROC to search the rest of the file for another table. Provide an integer value greater than zero for this argument. Suppose that this integer is N . Then the <i>find_cells_in_lists()</i> function will discontinue its reading of the model input file after it has encountered N property list tables. The <i>blocks_to_read</i> argument is optional; if it is not supplied, a very high number is assumed.
<i>list_col_start</i>	This argument can be supplied as <i>list_col</i> for short. This is the column number within each cell property table at which the listing of model cell indices begins. Often, but not always, this is column 1. As is described below, model cell indices may also occupy the following one or two columns.
<i>model_type</i>	The value of this argument is a text string. Depending on its value, up to three accompanying subarguments must be supplied. The value of the <i>model_type</i> argument must be supplied as “mf6_dis”, “mf6_disv” or “undefined” (“undef” for short). Only in the “undefined” case are subarguments required.
<i>model_type;node</i>	Where <i>model_type</i> is provided as “undefined”, then a <i>node</i> subargument may follow. This informs the <i>find_cells_in_lists()</i> function of the number of nodes in the model. This must be the same as the number of elements in the CLIST to which this function pertains.
<i>model_type;nlay</i> <i>model_type;ncpl</i>	Where <i>model_type</i> is provided as “undefined”, then an <i>nlay</i> subargument may follow. Where a model is of the DISV type, then an <i>ncpl</i> (number of cells per layer) subargument must follow the <i>nlay</i> subargument. These are both integers. The product of <i>nlay</i> and <i>ncpl</i> must equal the number of elements in the CLIST to which this function pertains.
<i>model_type;nlay</i> <i>model_type;nrow</i> <i>model_type;ncol</i>	Where <i>model_type</i> is provided as “undefined”, then an <i>nlay</i> subargument may follow. Where the model is of the structured grid (i.e. DIS) type, then a <i>nrow</i> (number of rows) subargument must follow that; this must then be followed by a <i>ncol</i> (number of columns) subargument. The product of <i>nlay</i> , <i>nrow</i> and <i>ncol</i> must equal the number of elements in the CLIST to which this function pertains. Meanwhile <i>nlay</i> , <i>nrow</i> and <i>ncol</i> must equal the number of layers, rows and columns in the model to which the CLIST pertains.

Discussion

Model input files

The MODFLOW family of models read some of their boundary condition specifications from lists (i.e. tables). Each line of such a table pertains to one model cell. The cell is identified by

up to three indices. The nature of these indices depends on the nature of the model grid. A new cell property list may be provided for each new stress period.

Depending on the type of boundary condition, one or a number of property columns may follow the cell index columns. For example, extraction rates follow cell indices in a MODFLOW WEL package input file. Elevations and conductances follow cell indices in a MODFLOW GHB package input file; elevations, conductances and bottom elevations follow cell indices in a MODFLOW RIV package input file. One or more of these boundary condition properties may be adjustable by PEST. SEGLIST-linked pilot points, or ordinary pilot points, may be employed for boundary condition parameterization.

In a MODFLOW-USG or MODFLOW6 DISU model, a cell is depicted by its node number. Hence only one column of cell indices is required in boundary condition specification tables. For a MODFLOW6 DISV model, cell index specification requires two columns; the first column supplies cell layer numbers while the second column provides so-called “cell 2d numbers” (i.e. the cell index within each model layer). For a MODFLOW6 DIS model, or a structured grid MODFLOW model, three columns are required for cell specifications; these provide cell layer, row and column numbers (in that order).

The following figure shows part of a MODFLOW6 WEL package input file. The model to which it pertains employs a structured grid. Cells are depicted by their layer, row and column numbers. Each block in which properties are listed is bracketed by a “begin period” (i.e. *keytext_start*) text string and an “end period” (i.e. *keytext_end*) text string. Note that PLPROC’s use of these text strings is case-insensitive.

```
BEGIN PERIOD 1
  1  6  4    0.0000000
  1  6  5    0.0000000
  1  7  4    0.0000000
  1  7  5    0.0000000
  1  8  4    0.0000000
  1  8  5    0.0000000
  1  9  4    0.0000000
  1  9  5    0.0000000
  1 10  4    0.0000000
  1 10  5    0.0000000
END PERIOD

BEGIN PERIOD 3
  1  6  4   -0.85
  1  6  5   -0.85
  1  7  4   -0.85
  1  7  5   -0.85
  1  8  4   -0.85
  1  8  5   -0.85
  1  9  4   -0.85
  1  9  5   -0.85
  1 10  4   -0.85
  1 10  5   -0.85
END PERIOD
etc
```

Part of a WEL package input file for a structured grid MODFLOW model.

Model type

As presently programmed, three values can be supplied for the *model_type* argument. These are “mf6_dis”, “mf6_disv” and “undefined”. For the first two options, PLPROC checks whether the reference CLIST was defined using a *read_mf6_grid_specs()* function call. If it was not defined using this function, or if the type of grid (DIS or DISV) to which the CLIST pertains does not correspond with that provided through the *model_type* argument, then the *find_cells_in_lists()* function generates an error message. However, if there is agreement, then the *find_cells_in_lists()* function knows whether it must read layer and cell 2d numbers, or layer, row and column numbers from cell property lists in order to identify each cell.

Alternatively, a user can denote the *model_type* associated with the function’s reference CLIST as “undefined”. In this case the manner of model cell identification must be provided through subarguments of the *model_type* argument. As is discussed above, cells within a model can be identified using a single index (i.e. the node number of the cell), two indices (layer number and cell 2d number), or three indices (layer, row and column numbers). The nature of the user-provided subarguments indicates the manner of cell identification. The values of these arguments provides the number of nodes in the model, or its number of layers together with the number of cells in each layer, or the number of layers, rows and columns comprising the structured model grid. The *find_cells_in_lists()* function verifies that the total number of model cells calculated from these dimensions is equal to the number of cells in the reference CLIST.

Blocks to read

If it encounters a call to the *find_cells_in_lists()* function, PLPROC reads the user-specified model input file, looking for blocks containing cell property lists. For any model cell that is cited in any such block, the corresponding element of the new SLIST generated by the *find_cells_in_lists()* function is awarded a value of 1. Remaining cells are awarded values of 0.

There may be some occasions on which the *find_cells_in_lists()* function does not need to read the entirety of a model input file. See, for example, the following “connection data” block from the MAW (multi-aquifer well) package input file of a MODFLOW6 DIS model. (PLPROC may be used as an agent for parameterization of skin hydraulic conductivities and/or well radii.)

```
BEGIN CONNECTIONDATA
#   conn  l  r  c  stop  sbot      K      rskin
1     1  2 16 14    -5   -65 2982.54    0.1
1     2  3 16 14    -5   -65 2982.54    0.1
1     3  4 16 14    -5   -65 2982.54    0.1
1     4  5 16 14    -5   -65 2982.54    0.1
1     5  6 16 14    -5   -65 2982.54    0.1
1     6  7 16 14    -5   -65 2982.54    0.1
1     7  8 16 14    -5   -65 2982.54    0.1
1     8  9 16 14    -5   -65 2982.54    0.1
1     9 10 16 14    -5   -65 2982.54    0.1
1    10 11 16 14    -5   -65 2982.54    0.1
1    11 12 16 14    -5   -65 2982.54    0.1
1    12 13 16 14    -5   -65 2982.54    0.1
END CONNECTIONDATA
```

The “connection data” block of a MAW package input file for a MODFLOW6 DIS model.

The “connection data” block is provided only once in a MAW package input file. Hence there is no need for the *find_cells_in_lists()* function to continue reading this file once it has read the contents of this block. The value of the *blocks_to_read* argument can therefore be set to 1. In the present example, the MODFLOW6 model is of the DIS type; this requires that model cells be identified by layer, row and column numbers. When reading this block, the value of the *list_col_start* argument of function *find_cells_in_lists()* should be set to 3 because this is the column in which layer numbers are provided; row and column numbers are recorded in ensuing columns.

Starting and ending strings

A PLPROC user must supply a text string as the value of the *keytext_start* argument of the *find_cells_in_lists()* function. This function scans all lines of the user-nominated model input file in search of this text string. The search is case-insensitive. If the text is found, then PLPROC assumes that a cell properties block (i.e. table) begins on the next line. It is important to note that if this text string appears anywhere within a line of text on a MODFLOW input file, it denotes the start of a block. It can be accompanied by other text.

Similar considerations apply to the *keytext_end* argument. However there is an important difference. *keytext_end* can be blank, whereas this is not permitted for *keytext_start*. If *keytext_end* is not set to a blank string, then PLPROC ignores blank lines within a cell property block; furthermore, it does not declare the block to be closed unless it actually encounters the *keytext_end* character string. However if a blank string is assigned to the *keytext_end* argument, then a cell property block is declared to be closed if a blank line is encountered while processing its contents.

Comments

In accordance with MODFLOW protocol, the *find_cells_in_lists()* function ignores any lines in a model input file that begin with the “#” character, for this line is treated as a comment.

Examples

Example 1

```
sl_ghb = cl_mf6.find_cells_in_lists(file=ci.ghb,      &
    model_type=mf6_disv,                             &
    list_col_start=1,                                  &
    keytext_start='begin period',                      &
    keytext_end='end period')
```

This example assumes that a CLIST named “cl_mf6” had been previously created using a *read_mf6_model_specs()* function call. The MODFLOW6 model grid is of the DISV type. A model input file named *ci.ghb* is read. Within each block of this file that is subtended by lines that include the “begin period” and “end period” strings (these may or may not be capitalized in file *ci.ghb*), cell layer, row and column numbers are read from columns 1, 2 and 3. A new SLIST named “sl_ghb” is created. For any cell that is cited in the model input file, the corresponding element of the SLIST is awarded a value of 1. Other cells are assigned an SLIST value of 0.

Example 2

```
sl_ghb (select = (mf6_layer.gt.1)) =
```

```

cl_mf6.find_cells_in_lists(file=ci.ghb,      &
model_type=mf6_disv,                        &
list_col_start=1,                          &
keytext_start='begin period',              &
keytext_end='end period')

```

This example is the same as example 1 except for the presence of a selection equation on the left side of the “=” symbol. No member of the new *sl_ghb* SLIST is awarded a value of 1 unless the corresponding value of the *mf6_layer* SLIST is greater than 1.

Example 3

```

sl_ghb (select = (mf6_layer.gt.1)) =      &
cl_mf6.find_cells_in_lists(file=ci.ghb,  &
model_type=undefined;nlay=2;ncpl=5240,   &
list_col_start=1,                        &
keytext_start='begin period',            &
keytext_end='end period')

```

This example achieves a similar outcome to the preceding example, but using a different (and more cumbersome) protocol. The *model_type* is declared as “undefined”. Subarguments of this argument indicate that the model is of the MODFLOW6 DISV type. Subargument values provide the dimensions of the model. In doing so, they implicitly provide the number of elements in the *cl_mf6* CLIST; PLPROC verifies this number against the actual number of elements in the CLIST.

Example 4

```

sl_riv = cl_mf6 .find_cells_in_lists(file="case.riv",  &
model_type=undefined; nodes=84321,                    &
list_col_start=1,                                     &
keytext_start='list_follows',                         &
keytext_end=' ',                                     &
blocks_to_read=9)

```

This example pertains to a MODFLOW-USG model with an unstructured grid. Within this type of grid, each model cell is specified by a single number, this being its node number. According to the value of the *keytext_start* argument, the onset of a block containing a cell property list is denoted by the presence of the string “list_follows” on the preceding line. This is not standard MODFLOW-USG protocol; presumably these text strings were added to the MODFLOW-USG input file either by the modeller, or by the graphical user interface that wrote the model input file. The latter file is named *case.riv*. The end of a cell property block is recognized by a blank line. The value of 9 ascribed to the *blocks_to_read* argument informs function *find_cells_in_lists()* to read only the first 9 blocks from the model input file. Subsequent blocks are ignored. (Perhaps all cells that are ever awarded a “river” status in any model stress period are already cited in one or more of these first nine blocks.)

gen_ran_plists_basic()

General

Function *gen_ran_plists_basic()* creates or alters PLISTs cited in an MLIST; if necessary, it creates the MLIST itself. As described elsewhere in this document, an MLIST is a collection of PLISTs which share common CLIST parentage, and which share a common root name.

Some or all of the PLISTs which comprise the MLIST that the *gen_ran_plists_basic()* function targets may already be in existence. If this is the case, their parent CLIST must be that with which the new MLIST is associated; otherwise PLPROC will cease execution with an appropriate error message. Those PLISTs which do not exist are brought into existence by the *gen_ran_plists_basic()* function.

Many options are provided for random number generation. The governing distribution can be (log)uniform or (log)normal; in the latter case individual PLIST elements can be statistically independent or linked by a user-supplied covariance matrix. Where a covariance matrix is employed, the PLIST should not be too large - never more than two to three thousand elements. This is because the user-supplied covariance matrix must be subjected to singular value decomposition before random number generation can take place. Where generation of correlated random numbers is required for larger PLISTs, then alternative methods for filling these PLISTs (for example sequential multiGaussian) field generation must be found.

Random number generation can be applied to all or part of a PLIST; in the latter case elements which are to be filled by random numbers are denoted through a selection equation.

Function Specifications

Function value

The *gen_ran_plists_basic()* function populates (and creates if necessary) a sequence of PLISTs that are wholly or partially filled with random values. These are collected into an MLIST (which is created if necessary). The name of this MLIST is the target of the function. This name must appear before the function in a PLPROC command which uses the function. This name should be separated from the function by a “=” symbol. Optionally a target selection equation can be provided with the MLIST name, this restricting random number generation to a subset of elements of the PLISTs represented by the collective MLIST.

Object associations

All PLISTs which are populated by the *gen_ran_plists_basic()* function must be associated with a common CLIST. In PLPROC commands which employ this function, the name of the function must follow the name of the CLIST on which it operates; a dot must separate these two names.

Arguments and subarguments

<i>distribution</i>	The type of probability distribution from which random numbers are drawn. This must be supplied as either “uniform” or “normal”. This argument is mandatory.
---------------------	--

<i>transform</i>	This indicates whether the probability distribution supplied as the value of the <i>distribution</i> argument pertains to native or log (to base 10) transformed PLIST elements. Legal values are “none” and “log”. This argument is mandatory.
<i>nonselect</i>	This is the value that is supplied to PLIST elements for which random values are not generated because of the existence of a selection equation. If new PLISTs are created then either this argument, or the <i>nonselect_plist</i> argument, are mandatory. If no new PLISTs are created then this argument is optional. If it is not supplied, then non-selected PLIST elements retain their existing values.
<i>nonselect_plist</i>	This has the same role as the <i>nonselect</i> argument. However it allows PLIST elements that remain unselected by the selection equation to be filled with values from another PLIST. That PLIST must already exist and have the same parent CLIST as that to which the function pertains.
<i>upper_limit</i>	The <i>upper_limit</i> argument is mandatory if the distribution is <i>uniform</i> and optional otherwise. In the former case it defines the upper end of the uniform probability interval. In the latter case normally-distributed random values are clipped if they exceed the <i>upper_limit</i> . If the transform is “log”, then this value is log-transformed before being used in random number generation or clipping.
<i>upper_limit_plist</i>	This argument serves the same role as the <i>upper_limit</i> argument, but allows the <i>upper_limit</i> to be provided on an element-by-element basis using an existing CLIST-compatible PLIST whose name is provided as the value of this argument.
<i>lower_limit</i>	The <i>lower_limit</i> argument is mandatory if the distribution is <i>uniform</i> and optional otherwise. In the former case it defines the lower end of the uniform probability interval. In the latter case normally-distributed random values are clipped if they are less than the <i>lower_limit</i> . If the transform is “log”, then this value is log-transformed before being used in random number generation or clipping.
<i>lower_limit_plist</i>	This argument serves the same role as the <i>lower_limit</i> argument, but allows the <i>lower_limit</i> to be provided on an element-by-element basis using an existing CLIST-compatible PLIST whose name is provided as the value of this argument.
<i>mean</i>	The mean argument is mandatory if the <i>distribution</i> type is “normal”. It is forbidden otherwise. If the <i>transform</i> type is “log” then the value supplied as the argument to this function is log (to base 10) transformed before random number generation commences.
<i>mean_plist</i>	This argument serves the same purpose as the <i>mean</i> argument, but allows PLIST element mean values to be supplied on an element-by-element basis using a CLIST-compatible PLIST whose name is

	provided as the argument to this function.
<i>sd</i>	This is the standard deviation if the <i>distribution</i> is “normal” and if the <i>transform</i> is “none”. This or the <i>sd_plist</i> argument is mandatory under these conditions; otherwise the <i>sd</i> argument is illegal.
<i>sd_plist</i>	This argument serves the same purpose as the <i>sd</i> argument, but allows PLIST element standard deviation values to be supplied on an element-by-element basis using a CLIST-compatible PLIST whose name is provided as the value of this argument.
<i>sd_fac</i>	“sd_fac” stands for “standard deviation factor”. This argument must be supplied if the <i>distribution</i> is “normal” and the <i>transform</i> is “log”. The log (to base 10) of this value is the standard deviation of the normal distribution used for random number generation in log-space. In untransformed space it thus comprises a factor applied to the mean.
<i>sd_fac_plist</i>	This argument serves the same purpose as the <i>sd_fac</i> argument, but allows PLIST element standard deviation factors values to be supplied on an element-by-element basis using a CLIST-compatible PLIST whose name is provided as the value of this argument.
<i>covmat</i>	The name of a PLPROC-stored matrix which serves as the covariance matrix where the <i>distribution</i> is “normal”. Where the <i>transform</i> is “none”, variances/covariances supplied in this matrix must refer to native PLIST elements. Where the <i>transform</i> is “log”, variances/covariances supplied in this matrix must refer to log (to base 10) of PLIST elements.
<i>covmat; covmult</i>	The entire covariance matrix is multiplied by the value of the <i>covmult</i> subargument. This must be greater than zero. If it is omitted a value of 1.0 is assumed.
<i>covmat_file</i>	The name of the file in which the covariance matrix resides. This must not be supplied if a <i>covmat</i> argument is supplied. Where the <i>transform</i> is “none”, variances/covariances supplied in this matrix must refer to native PLIST elements. Where the <i>transform</i> is “log”, variances/covariances supplied in this matrix must refer to log (to base 10) of PLIST elements.
<i>covmat_file; format</i>	The <i>format</i> subargument of the <i>covmat_file</i> argument must be supplied as either “formatted” or “binary”. In the former case an ASCII (i.e. text) file is read, whereas binary storage is assumed in the latter case. (The user can employ “text” or “ascii” instead of “formatted” if he/she wishes, and “unformatted” instead of “binary” if he/she so desires.) If this subargument is omitted it is assumed to be “formatted”.
<i>covmat_file; covmult</i>	The entire covariance matrix is multiplied by the value of the <i>covmult</i> subargument. This must be greater than zero. If it is omitted a value of

	1.0 is assumed.
<i>mlist_start_counter</i>	This argument is mandatory. It is the initial PLIST index in the sequence of PLISTs to which the target MLIST refers. This must be no less than 1.
<i>mlist_end_counter</i>	This argument is mandatory. It is the final PLIST index in the sequence of PLISTs to which the target MLIST refers.
<i>seed</i>	This argument is mandatory. It is the random number generator seed. It must be an integer greater than zero.

Discussion

Legal argument values

If the following protocols are not observed, PLPROC will cease execution with an error message.

- If *transform* is log, then values supplied for the global *mean* and (optionally) the global *upper_limit* and *lower_limit* arguments must be greater than zero.
- The same applies if values are provided on an element-by-element basis through PLISTs whose names are provided to the *mean_plist*, *upper_limit_plist* and *lower_limit_plist* arguments.
- *sd* must be greater than zero. The same applies to elements of the *sd_plist* PLIST if this is supplied instead.
- *sd_fac* must be greater than 1.0. The same applies to elements of the *sd_fac_plist* PLIST if this is supplied instead.

Global values or PLISTs

For many parameters governing random number generation the *gen_ran_plists_basic()* function provides two alternatives. Either a single value can be provided, or the name of a PLIST can be provided. An error condition will occur if both of these are provided. If a PLIST is provided then it must have been created and filled before calling the *gen_ran_plists_basic()* function. The parent CLIST of the any such PLIST must be the same as that named in the *gen_ran_plists_basic()* function call.

Covariance matrix

Caution must be employed in using a covariance matrix for random number generation. The following should be noted carefully.

- The covariance matrix must be square, symmetric and positive definite. (This applies to all covariance matrices; it is fundamental to random number generation.)
- The number of elements in the covariance matrix must match the number of PLIST elements for which random numbers are to be generated (with the existence of a target selection equation taken into account).
- The ordering of rows/columns of the covariance matrix must be the same as that of PLIST elements for which random numbers are being generated. This is the same

order as elements are represented in the PLIST itself. It is the user's responsibility to ensure that this is the case.

If the covariance matrix is read from an ASCII external file, the specifications of this file are similar to those employed by PEST-suite utilities.

The first line of a covariance matrix file must contain three integers, these being *nrow*, *ncol* and *icode* where *nrow* and *ncol* denote the number of rows and columns in the following matrix. (These, of course, must be equal.) If *icode* is 1 or 2, then all elements of the matrix must follow, these being provided a row at a time. The first row is provided first, followed by the second row, etc. In a text file, elements of a row can wrap around onto the next line. However if *icode* is provided as -1, the matrix is thereby denoted as diagonal. A set of *nrow* numbers is then read, these being the diagonal elements of the matrix; off-diagonal elements are assumed to be zero.

An example of the first part of a small matrix file is shown below.

8	8	1					
0.3000000		9.3800500E-02	9.1346554E-02	7.6806664E-02	7.2156183E-02	8.7985337E-02	
8.6666018E-02		8.3441921E-02					
9.3800500E-02	0.3000000	9.3800500E-02	7.7414200E-02	7.2614901E-02	9.3800500E-02		
9.1346554E-02	8.6666018E-02						
Etc							

First part of a matrix file.

If the matrix is read from a binary file then the contents of the file are slightly different. A binary matrix file contains the same three-integer header as an ASCII matrix file. Integers in this header have the same meaning. However the matrix is stored in FORTRAN matrix element order, that is with row number cycling fastest and then column number.

Staged filling of PLISTS

As is discussed elsewhere in this document, the name of an MLIST must include a "*" character. This together with the *mlist_start_counter* and *mlist_end_counter* define a range of PLISTS (see the examples below). If these PLIST's do not exist, then the *gen_ran_plists_basic()* function creates them and fills them with random values. If a selection equation is supplied with the name of the target MLIST, then unselected elements of these PLISTS are filled with the number supplied as the *nonselect* argument, or with the values of corresponding elements of a PLIST whose name is provided as the value of the *nonselect_plist* argument.

If desired, subsequent calls to the *gen_ran_plist_basic()* function can be used to fill previously non-selected PLIST elements with random values generated according to different random number generation specifications. In that case the selection equation must exclude elements for which random numbers have already been generated. In this case neither the *nonselect* nor *nonselect_plist* argument must be provided in the call to the *gen_ran_plist_basic()* function. As the PLISTS which are collected by the target MLIST already exist, they do not need to be created. Elements which are not selected by the target selection equation remain untouched while selected elements are filled with random numbers.

Examples

Example 1

```
pu* = cl_pp.gen_ran_plists_basic(distribution='uniform', &
```

```

transform='none',          &
upper_limit=10.0,         &
lower_limit=5.0,          &
mlist_start_counter=15,   &
mlist_end_counter=20,     &
seed=10)

```

In the above example the target MLIST is named *pu**. Because *mlist_start_counter* is 15 and *mlist_end_counter* is 20, PLISTs *pu15*, *pu16*, *pu17*, *pu18*, *pu19* and *pu20* will be filled with random numbers. If these PLISTs do not exist, they will be created. If they exist, they will be overwritten. However if one of them does, in fact, exist and belongs to a CLIST other than *cl_pp*, PLPROC will record an error message and cease execution.

If the MLIST *pu** does not exist, it will be created. If it was previously created with *mlist_start_counter* and *mlist_end_counter* defining a range that is narrower than the range indicated above, then the MLIST counter range will be expanded to accommodate the new associated PLISTs. If the existing MLIST counter range is broader than that indicated above, then only those PLISTs between the *mlist_start_counter* and *mlist_end_counter* subroutines argument values will have their elements filled with random numbers.

Note that the individual PLISTs which collectively comprise the MLIST can be used in PLIST-specific function calls in later PLPROC processing. That is, they can be used in isolation from the MLIST which cites them. The fact that an index is built into their name makes them easy targets for loop-directed sequential processing. See the *goto()* function.

Example 2

```

pn*(select=(zone==2)) = cl_pp.gen_ran_plists_basic(          &
    distribution='normal',                                   &
    transform='log',                                       &
    mean_plist=mp,                                         &
    sd_fac=1.5,                                           &
    mlist_start_counter=15,                               &
    mlist_end_counter=17,                                 &
    seed=20)

```

In this example the only those elements of the *pn15*, *pn16*, and *pn17* PLISTs for which elements of a CLIST-compatible SLIST called *zone* are equal to 2 are filled with random numbers. It is presumed that these PLISTs already exist; if this is not the case a *nonselect* argument would need to be supplied with the *gen_ran_plists_basic()* function. Random numbers are generated using a log normal distribution. The mean of this distribution is different for every PLIST element; it is log (to base 10) of the value of pertinent elements in a PLIST named *mp*. The standard deviation is the same for all elements however; it is the log (to base 10) of 1.5. Note that random numbers are back-transformed to natural numbers before filling of the *pn15*, *pn16* and *pn17* PLISTs.

Example 3

```

pm*(select==(zone==2)) = cl_pp.gen_ran_plists_basic(          &
    distribution='normal',                                   &
    transform='log',                                       &
    mean=10.0,                                           &
    covmat_file='covmat.dat',                             &
    mlist_start_counter=15,                               &
    mlist_end_counter=20,                                 &
    seed=30)

```

A covariance matrix read from a file named *covmat.dat* is used as a basis for random number generation. The matrix housed in *covmat.dat* should be square, with the number of rows and columns equal to the number of elements for which SLIST *zone* values are 2. The variances and covariances in that matrix must pertain to log-transformed variables as the *transform* type used in the *gen_ran_plists_basic()* function call is “log”.

gen_ran_plists_cond()

General

The *gen_ran_plists_cond()* function is similar to the *gen_ran_plists_basic()* function in that it generates random numbers which fill a suite of PLISTs which collectively comprise an MLIST. However the following is assumed.

- Random variables are statistically correlated. Hence a covariance matrix must be supplied as a basis for random number generation.
- Some PLIST values have been sampled. Hence the covariance matrix and mean values pertaining to non-sampled PLIST elements are modified in accordance with conditioning imposed through the sampling process.

The *gen_ran_plists_cond()* function offers many options. All, or selected parts of, the suite of target PLISTs can be filled with random numbers; the probability distribution on which random number generation is based can be normal or log-normal; limits can be imposed on generated values; sampled values on which random number conditioning is based can be noisy or noise-free.

This function should not be used to fill large PLISTs. Where PLISTs have many elements (more than two to three thousand), sequential multiGaussian simulation should be used instead.

Function Specifications

Function value

The *gen_ran_plists_cond()* function populates (and creates if necessary) a sequence of PLISTs that are wholly or partially filled with random numbers. These PLISTs are collected into an MLIST (which is created if necessary); the name of this MLIST is the target of the function. This name must appear before the function in a PLPROC command which uses the function. The MLIST name should be separated from the function by a “=” symbol. Optionally, a target selection equation can be provided with the MLIST name, this restricting random number generation to a subset of elements of the PLISTs represented by the collective MLIST.

Object associations

All PLISTs which are populated by the *gen_ran_plists_cond()* function must be associated with a common CLIST. In PLPROC commands which employ this function, the name of the function must follow the name of the CLIST on which it operates; a dot must separate these two names.

Arguments and subarguments

transform

The probability distribution on which random number generation is based is assumed to be normal (see the discussion below). However this normal distribution can pertain to PLIST elements directly (if the *transform* is designated as “none”), or to their logs (if the *transform* is designated as “log”). This argument is mandatory.

<i>nonselect</i>	This is the value that is supplied to PLIST elements for which random values are not generated because of the existence of a target selection equation. If new PLISTs are created to fill the MLIST, then either this argument, or the <i>nonselect_plist</i> argument, is mandatory. If no new PLISTs are created (but are overwritten instead) then this argument is optional; if it is not supplied, non-selected PLIST elements retain their existing values.
<i>nonselect_plist</i>	This has the same role as the <i>nonselect</i> argument. However it allows PLIST elements that remain unselected by the target selection equation to be filled with values from another PLIST. That PLIST must already exist and have the same parent CLIST as that to which the function pertains.
<i>upper_limit</i>	This argument is arbitrary. If supplied, random values are clipped if they are greater than this value.
<i>upper_limit_plist</i>	This argument serves the same role as the <i>upper_limit</i> argument, but allows the <i>upper_limit</i> to be provided on an element-by-element basis using an existing CLIST-compatible PLIST whose name is provided as the value of this argument.
<i>lower_limit</i>	This argument is arbitrary. If supplied, random values are clipped if they are less than this value.
<i>lower_limit_plist</i>	This argument serves the same role as the <i>lower_limit</i> argument, but allows the <i>lower_limit</i> to be provided on an element-by-element basis using an existing CLIST-compatible PLIST whose name is provided as the value of this argument.
<i>mean</i>	The <i>mean</i> argument is mandatory. It is the prior (i.e. non-conditioned) mean ascribed to randomly generated PLIST element values; see the discussion below. If the <i>transform</i> type is “log” then the value supplied as the argument to this function is log (to base 10) transformed before conditioning and random number generation commences.
<i>mean_plist</i>	This argument serves the same purpose as the <i>mean</i> argument, but allows PLIST element prior mean values to be supplied on an element-by-element basis using a CLIST-compatible PLIST whose name is provided as the argument to this function.
<i>cond_value</i>	If all PLIST sample values used to condition random number generation are equal, then supply this single sample value through the <i>cond_value</i> argument. Either the <i>cond_value</i> or the <i>cond_value_plist</i> argument must be featured in the <i>gen_ran_plists_cond()</i> function, but not both.
<i>cond_value; select</i>	A selection equation (referred to as the “conditioning selection equation”) through which PLIST elements used for conditioning are

	designated. The CLIST which governs the conditioning selection equation should be the same as that governing the <i>gen_ran_plists_cond()</i> function. This subargument is mandatory.
<i>cond_value_plist</i>	This serves the same role as the <i>cond_value</i> argument. However it allows values used for conditioning to be provided on an element-by-element basis. The PLIST supplied as the value of this argument should have the same parent CLIST as that governing the <i>gen_ran_plists_cond()</i> function.
<i>cond_value_plist;</i> <i>select</i>	A selection equation (referred to as the “conditioning selection equation”) through which PLIST elements used for conditioning are designated. The CLIST which governs the selection equation should be the same as that governing the <i>gen_ran_plists_cond()</i> function. This subargument is mandatory.
<i>cond_mean_plist</i>	Provide the name of an existing CLIST-compatible PLIST in which the conditional mean of randomly-generated PLISTs will be recorded; see the discussion below. This argument is optional.
<i>sd_cond</i>	Values used for conditioning can have randomness associated with them. If they all have the same standard deviation, then this standard deviation can be supplied through the <i>sd_cond</i> argument. This argument is optional. If the <i>transform</i> is “log” then the <i>sd_cond_fac</i> argument should be used instead.
<i>sd_cond_plist</i>	This serves the same role as the <i>sd_cond</i> argument, but allows conditioning standard deviations to be supplied on an element-by-element basis using a CLIST-compatible PLIST whose name is provided as the value of this argument.
<i>sd_cond_fac</i>	If <i>transform</i> is “log” then the log (to base 10) of the value supplied for this argument provides the standard deviation of the logs of randomly-generated conditioning values. This argument is optional.
<i>sd_cond_fac_plist</i>	This serves the same role as the <i>sd_cond_fac</i> argument, but allows conditioning standard deviations of the logs of conditioning variables to be supplied on an element-by-element basis using a CLIST-compatible PLIST whose name is provided as the value of this argument.
<i>covmat</i>	The name of a PLPROC-stored matrix which serves as the pre-conditioning covariance matrix on which random number generation is based. Where the <i>transform</i> is “none”, variances/covariances supplied in this matrix must refer to native PLIST elements. Where the <i>transform</i> is “log”, variances/covariances supplied in this matrix must refer to the log (to base 10) of PLIST elements.
<i>covmat; covmult</i>	The entire covariance matrix is multiplied by the value of the <i>covmult</i> subargument. This must be greater than zero. If it is omitted a value of

	1.0 is assumed.
<i>covmat_file</i>	The name of the file in which the pre-conditioning covariance matrix resides. This must not be supplied if a <i>covmat</i> argument is supplied. Where the <i>transform</i> is “none”, variances/covariances supplied in this matrix must refer to native PLIST elements. Where the <i>transform</i> is “log”, variances/covariances supplied in this matrix must refer to the log (to base 10) of PLIST elements.
<i>covmat_file</i> ; <i>format</i>	The <i>format</i> subargument of the <i>covmat_file</i> argument must be supplied as either “formatted” or “binary”. In the former case an ASCII (i.e. text) file is read, whereas binary storage is assumed in the latter case. (The user can employ “text” or “ascii” instead of “formatted” if he/she wishes, and “unformatted” instead of “binary” if he/she so desires.) If this subargument is omitted it is assumed to be “formatted”.
<i>covmat_file</i> ; <i>covmult</i>	The entire covariance matrix is multiplied by the value of the <i>covmult</i> subargument. This must be greater than zero. If it is omitted a value of 1.0 is assumed.
<i>mlist_start_counter</i>	This argument is mandatory. It is the initial PLIST index in the sequence of PLISTs to which the target MLIST refers. This must be no less than 1.
<i>mlist_end_counter</i>	This argument is mandatory. It is the final PLIST index in the sequence of PLISTs to which the target MLIST refers.
<i>seed</i>	This argument is mandatory. It is the random number generator seed. It must be an integer greater than zero.

Discussion

Theory

Let \mathbf{x} be a vector of random variables and let $C(\mathbf{x})$ be its covariance matrix. Let the vector $\boldsymbol{\mu}$ denote its mean. Suppose that we partition \mathbf{x} into two vectors \mathbf{x}_1 and \mathbf{x}_2 as follows:

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} \quad (1)$$

so that its mean is likewise partitioned as

$$\boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{bmatrix} \quad (2)$$

Let the $C(\mathbf{x})$ covariance matrix be correspondingly partitioned as:

$$C(\mathbf{x}) = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \quad (3)$$

Now suppose that values have been obtained for the elements of \mathbf{x}_2 through direct sampling of these values. If randomness of \mathbf{x} is described by a normal distribution, then the following formulas can be used to compute the conditional mean $\boldsymbol{\mu}_1'$ of \mathbf{x}_1 and the conditional covariance matrix \mathbf{C}'_{11} of \mathbf{x}_1 that prevails after values for \mathbf{x}_2 have been obtained through sampling.

$$\boldsymbol{\mu}_1' = \boldsymbol{\mu}_1 - \mathbf{C}_{12} \mathbf{C}_{22}^{-1} (\mathbf{x}_2 - \boldsymbol{\mu}_2) \quad (4)$$

$$\mathbf{C}'_{11} = \mathbf{C}_{11} - \mathbf{C}_{12} \mathbf{C}_{22}^{-1} \mathbf{C}_{21} \quad (5)$$

Once $\boldsymbol{\mu}_1'$ and \mathbf{C}'_{11} have been determined, random values can be generated for the elements of \mathbf{x}_1 subject to conditioning by \mathbf{x}_2 . These random values are centred on $\boldsymbol{\mu}_1'$ and employ \mathbf{C}'_{11} as a covariance matrix.

In the spatial setting, it can be shown that equation (4) above is equivalent to simple kriging.

Conditioning values

All operations performed by the *gen_ran_plists_cond()* function assume a common CLIST. The name of this CLIST must precede the function name, followed by a dot, in PLPROC scripts which use this function. PLISTs comprising the MLIST produced or modified by the *gen_ran_plists_cond()* function have the same parent CLIST. Any PLIST or SLIST cited in a *gen_ran_plists_cond()* argument, or in a *gen_ran_plists_cond()* selection equation, must be a child of this same CLIST.

Operation of *gen_ran_plists_cond()* is predicated on the assumption that random values are to be generated for elements of target PLISTs based on a covariance matrix that provides statistical characterisation of PLIST elements. It is also assumed that values have already been acquired for some of these elements. The acquisition of this data modifies the mean and covariance matrix pertaining to unsampled elements. The *gen_ran_plists_cond()* function first computes the conditional mean and covariance of unsampled elements, and then generates random values for non-sampled PLIST elements on that basis. Optionally, the conditional mean can be recorded in a PLIST identified through the *cond_mean_plist* argument. Random numbers are written to PLISTs comprising the MLIST which is the target of the *gen_ran_plists_cond()* function. See the introduction to this manual, and the *gen_ran_plists_basic()* function, for more discussion of MLISTs.

In writing random values to target PLISTs, values provided to PLIST elements that correspond to conditioning points are simply the conditioning values themselves. If these conditioning values are all the same, they can be supplied on an element-by-element basis through the *cond_value* argument. Alternatively, they can be supplied through a PLIST named as the value of the *cond_value_plist* argument. In either case a selection equation must be supplied; PLIST elements used for conditioning purposes are identified through this equation. Evaluation of this equation must therefore result in the selection of between 1 and $N-1$ elements as conditioning elements, where N is the number of target PLIST elements that must be filled with random numbers. N can be equal to the total number of elements which comprise each of these PLISTs; alternatively a target selection equation can instruct the *gen_ran_plists_cond()* function to fill only a subset of elements of target PLISTs with random numbers.

If desired, some stochasticity can be associated with sampled values. This simulates the fact that the sampling process may be accompanied by measurement noise. Hence the single

conditioning value supplied with the *cond_value* argument, or the element-specific sample values supplied in the PLIST cited as the *cond_value_plist* argument, can be perturbed on each occasion that random numbers are generated for target PLISTs. Random generation of these sample values is centred on respective *cond_value* or *cond_value_plist* sample values. If the *transform* is “none” then the standard deviation associated with perturbation can be supplied as either a single value through the *sd_cond* argument, or as element-specific values through the *sd_cond_plist* argument. Alternatively, if the *transform* is log, then the stochasticity of factors by which *cond_value* or *cond_value_plist* values are randomly multiplied can be supplied through the *sd_cond_fac* argument or the *sd_cond_fac_plist* argument. These factors are log-transformed before random number generation. The log-transformed values of these factors are the standard deviations used for generation of random numbers (centred on zero) that are added to the logs of sample values supplied through the *cond_value* or *cond_value_plist* arguments.

Note that random perturbation of conditioning values is optional. It does not take place at all if no values are supplied for the *cond_sd*, *cond_sd_plist*, *cond_sd_fac* or *cond_sd_fac_plist* arguments.

Caution should be exercised in randomly perturbing conditioning values. Presumably, the standard deviation of their perturbation should be much smaller than the pre-conditioned standard deviations of these elements as supplied through the covariance matrix. (The latter is provided through the *covmat* or *covmat_file* subroutine arguments.) If this is not the case, then direct sampling of values is useless.

Note that the conditional mean used for random number generation changes as conditioning values are randomly perturbed. However conditional means recorded in the *cond_mean_plist* PLIST do not. In this PLIST, values ascribed to sampled PLIST elements are those provided through the *cond_value* or *cond_value_plist* arguments. Values calculated for other elements assume these conditioning values.

Selection equations

The *gen_ran_plists_cond()* function accepts two selection equations, an optional target selection equation and a mandatory conditioning selection equation.

The target selection equation restricts generation of random values for PLIST elements to those selected through that equation. This allows complex PLISTs to be built in sequence. See the description of the *gen_ran_plists_basic()* function for a description of how this works, and for the role of the optional *nonselect* and *nonselect_plist* arguments.

While the target selection equation is optional, the conditioning selection equation supplied with the *cond_value* or *cond_value_plist* arguments is not. This identifies those PLIST elements which are to be used for conditioning purposes. Presumably, the conditioning selection equation that is supplied as the subargument to either of these arguments cites an SLIST which identifies a handful of PLIST elements through use of an appropriate integer; values that are actually used for conditioning purposes are then obtained through the *cond_value* or *cond_value_plist* arguments. If a target selection equation is present, all elements selected by the conditioning selection equation must lie within the range of elements selected for random number generation by the target selection equation. If this is not the case, PLPROC will cease execution with an appropriate error message.

Limits

Limits can be placed on values assigned to target PLISTs through use of the *lower_limit*, *lower_limit_plist*, *upper_limit* and *upper_limit_plist* arguments. No random or conditioned value which is outside of these limits will be assigned to elements of PLISTs comprising the target MLIST. The same does not apply to the conditional mean PLIST however (i.e. the PLIST whose name is supplied through the *cond_mean_plist* argument). If filled, the values assigned to elements of this PLIST are exactly as calculated using equation (4) above. If limits on elements of this PLIST are desired, these can be applied in later PLPROC processing through the *min()* and *max()* arguments of a PLIST equation.

Care should be taken when using limits. Limits imposed on random values alter the properties of the probability distribution used to generate them. This is an important consideration for the *gen_ran_plists_cond()* function, as a normal distribution is assumed in derivation of equations (4) and (5) above.

A final warning

The *gen_ran_plists_cond()* function should not be used on large PLISTs. Covariance matrix conditioning requires matrix inversion; see equations (4) and (5). Generation of correlated random numbers takes place through singular value decomposition. Random number generation will thus become very slow where selected PLIST elements are greater than about two thousand in number.

Examples

Example 1

```
pc* (select = (z1==2)) =
      cl_pp.gen_ran_plists_cond(           &
      transform='log',                   &
      nonselect=9999.99,                  &
      mean=10,                            &
      covmat_file='covmat2.dat';covmult=2.0, &
      mlist_start_counter=15,             &
      mlist_end_counter=20,              &
      seed=20,                            &
      cond_value=11; select=(c1==1),      &
      cond_mean_plist=pcm)
```

In this example PLISTs named *pc15*, *pc16*, *pc17*, *pc18*, *pc19*, and *pc20* are written or altered. Only part of each of these PLISTs is filled with conditioned random numbers however, for a target selection equation restricts random-number-filled elements to those for which values in the *z1* SLIST are equal to 2. This SLIST must have the same parent CLIST as that governing the *gen_ran_plists_cond()* function, namely *cl_pp*. All target PLIST elements which are not selected through this equation are given a value of 9999.99.

A log-normal probability distribution is assumed. The pre-conditioned (i.e. prior) mean of all random numbers is uniformly log (to base 10) of 10. The pre-conditioned (i.e. prior) covariance matrix is supplied in a file named *covmat2.dat*. However all entries in this file must be multiplied by 2.0 to achieve the covariance matrix required for the random number generation process.

A number of PLIST elements have already been sampled and are thus used for conditioning. These are those for which values in a concordant SLIST named *c1* are equal to 1. (As stated above, all of these elements must lie within the selection interval specified by the target selection equation.) All sampled values are equal to 11.0. This value is transferred directly to pertinent elements of PLISTs comprising the target MLIST.

As well as writing random values to PLISTs comprising the *pc** MLIST, the *gen_ran_plists_cond()* function fills values of a pre-existing PLIST named *pcm* with conditional mean values (back-transformed to the domain of natural numbers). However, as for randomly-generated values, filling of elements is restricted to those identified through the target selection equation. Elements that are not provided with random values are left unchanged, even if a value is supplied for the *nonselect* or *nonselect_plist* arguments. This leaves the user free to process this PLIST in ways that are different from randomly generated PLISTs.

Example 2

```
pc* (select = (z1==2)) =
      cl_pp.gen_ran_plists_cond(           &
      transform='log',                     &
      nonselect=9999.99,                   &
      mean=10,                             &
      covmat_file='covmat2.dat';covmult=2.0, &
      mlist_start_counter=15,              &
      mlist_end_counter=20,                &
      seed=20,                             &
      cond_value_plist=pcc; select=(c1==1), &
      cond_sd_fac=1.1,                     &
      cond_mean_plist=pcm)
```

This example is almost equivalent to the preceding example. However there are two important differences. The first is that values used for conditioning are read from a PLIST named *pcc*; the elements of this PLIST which are actually used for conditioning are selected through the (*c1==1*) selection equation supplied as a subargument of the *cond_value_plist* argument.

The second difference is that conditioning values are subject to randomness. They are multiplied by a random number with a mean of 1.0 and with randomness governed by the *cond_sd_fac* argument. In the log domain this is equivalent to addition of a random number with a mean of zero and a standard deviation equal to the log (to base 10) of 1.1. Hence the actual numbers used for conditioning will be slightly larger or slightly smaller than the values supplied in the *pcc* PLIST on each occasion of conditional random number generation.

Example 3

```
pc* (select = (z1==2)) =
      cl_pp.gen_ran_plists_cond(           &
      transform='none',                    &
      nonselect=9999.99,                   &
      mean=10,                             &
      covmat_file='covmat2.dat';covmult=2.0, &
      mlist_start_counter=15,              &
      mlist_end_counter=20,                &
      seed=20,                             &
      cond_value_plist=pcc; select=(c1==1), &
      cond_sd_plist = csd,
```

```
cond_mean_plist=pcm)
```

This example is very similar to the previous example. However a normal rather than log-normal distribution is assumed. As a result, *transform* is set to “none”. Also the *cond_sd_fac* argument cannot be used; the *cond_sd_plist* argument is used instead.

In this example the *gen_ran_plists_cond()* function is directed to the *csd* PLIST to find standard deviations that govern randomness assigned to conditioning elements. Hence a random variable with a mean of zero is added to each conditioning value supplied in the *pcc* PLIST before conditioning is undertaken. The standard deviation associated with these random values is provided in respective elements of the *csd* PLIST.

goto()

General

The *goto()* function directs execution of a PLPROC script to another part of the same PLPROC script which is denoted by a label. A label is a non-executable line of PLPROC script which begins with a colon.

Re-direction can be conditional. The *goto()* function includes an optional selection equation in which this condition can be denoted. The selection equation employed in a *goto()* function must cite only scalar or list-derived scalar variables; it cannot cite PLISTs or SLISTs as the equation is not evaluated for each element in these LISTs. Normally the scalar variable included in a *goto()* function selection equation will be a counter. Conditional redirection of execution in this manner provides the basis for looping within a PLPROC script.

Function Specifications

Function value

The *goto()* function makes no assignment to any PLPROC entity. Its name must lead the PLPROC command through which it is invoked.

Arguments and subarguments

<i>label</i>	The name of a label within the current PLPROC script file. A label must be 20 characters or less in length and contain no spaces.
<i>label; select</i>	A <i>select</i> subargument of the <i>label</i> argument makes action of the <i>goto()</i> function conditional on certain specifications being met. This condition will normally pertain to a scalar loop variable. Its outcome must be logical, that is .TRUE. or .FALSE.

Discussion

As is shown in the example below, the *goto()* function and its associated selection equation, together with \$scalar\$ substitution, implements sequential execution of groups of commands, with loop-specific entities being included in these commands.

Example

Example 1

```
i=0
:start_loop
  i=i+1
  storcoeff=stor$i$coeff
  infile='storage'/'$i$'/'ref'
  write_model_input_file(template_file='storage.tpl', &
                        model_input_file=infile)
  goto(label=':end_loop';select=(i.eq.3))
  goto(label=:start_loop)
:end_loop
```

It is presumed that PLPROC script preceding that shown in the above code fragment has defined and filled three PLISTs, and that these are named *stor1coeff*, *stor2coeff* and *stor3coeff*. It is also presumed that the template file *storage.tpl* provides the means for writing the contents of a PLIST named *storcoeff* to a file.

The iteration variable in the above loop is the scalar *i*. It is defined and initiated through the first line of script shown in the above PLPROC script fragment. The *start_loop* label defines the beginning of the loop while the *end_loop* variable defines the end of the loop.

The first action undertaken within the loop is incrementation of the loop variable *i*. Then the *storcoeff* PLIST is assigned the contents of *stor1coeff*, *stor2coeff* or *stor3coeff*, this depending on the loop count. PLIST selection is done through \$scalar\$ substitution of the loop counter to formulate the name of the pertinent PLIST. At the same time the file to which the contents of this PLIST are to be written is, in similar fashion, denoted as *storage1.ref*, *storage2.ref* or *storage3.ref*, this also depending on the number of times that the loop has been traversed. In the fourth statement following the *start_loop* label, this file is written. It is always written using the same template file *storage.tpl*. The name of the PLIST cited in the template file cannot be altered through \$scalar\$ substitution; this is why the contents of the *stor*coeff*, PLISTs must be copied to the *storcoeff* PLIST before they are written to the respective *storage*.ref* file.

After the pertinent file is written, the *goto()* function directs PLPROC execution to the *end_loop* label, but only if the scalar variable *i* is equal to 3. If this is not the case, then the next scripting line is executed. This line of script contains another *goto()* function, but without a *select* subargument of the *label* argument. Hence re-direction to the start of the loop is unconditional. Another passage through the loop then takes place.

inform_from_gplane()

General

The *inform_from_gplane()* function assigns values to non-GPLANE PLIST elements which are close enough to a GPLANE to be influenced by its presence. The values of PLIST elements associated with the GPLANE's gridded CLIST are first interpolated to points of projection of external PLIST elements onto the GPLANE. Unless the projection point of a particular PLIST element lies outside the polygonal boundaries of the GPLANE, the influence of the GPLANE on the external point is calculated using the perpendicular distance of the point to its projection on the GPLANE surface. If the off-GPLANE point projects onto the GPLANE's plane at a point that is external to its boundaries, further diminution of the GPLANE's influence on that point occurs in the manner discussed in Section 2.4.

Function Specifications

Function value

The *inform_from_gplane()* function modifies values of some or all elements of an existing PLIST. This off-GPLANE PLIST is referred to as the "target PLIST". The name of the target PLIST comprises the output of the function. This name must appear before the function in a PLPROC command which uses the function, optionally accompanied by a target selection equation. This name should be separated from the function by a "=" symbol.

Object associations

Function *inform_from_gplane()* employs in-GPLANE source PLIST element values to calculate values for elements of a target, off-GPLANE PLIST. The name of the function must follow the name of the source, in-GPLANE PLIST on which it operates; a dot must separate these two names.

Arguments and subarguments

<i>select</i>	The source PLIST selection equation. This argument is optional. Note that a target PLIST selection equation can also be supplied if desired. The latter must be placed adjacent to the name of the target PLIST on the left side of the assignment operator; see the example below.
<i>conwidth</i>	The "constancy width" of the GPLANE influence function. This is w of Figure 2.3a and equation (2.4.1). This argument is mandatory.
<i>a</i>	This is a of equation (2.4.1). It specifies how quickly the influence of the GPLANE decays with distance from the GPLANE surface. This argument is mandatory.
<i>action</i>	The value of this mandatory argument must be "add", "blend", "blend_never_higher" or "blend_never_lower".

action;transform The value supplied for this mandatory subargument of the *action* argument must be “log” or “none”. If it is supplied as “log”, the log of off-GPLANE PLIST elements are influenced by the log of GPLANE PLIST element values interpolated to the point of off-GPLANE projection. If supplied as “none” the influence is calculated using the native value of GPLANE-interpolated PLIST element values at the point of external PLIST element projection.

Discussion

The mechanism through which a GPLANE influences the values of PLIST elements that are close to it is discussed in Section 2.4, and is not repeated here. However a number of points are worthy of mention.

The text string which must be supplied as the value of the *action* argument of function *inform_from_gplane()* pertain to the options presented in equations 2.4.2(a-d). As stated above, the *conwidth* and *a* arguments are *w* and *a* of equations 2.4.1(a-b).

Equations (2.4.1) and (2.4.2) can be applied to the logarithms of the numbers on which they operate, or to the native values of these numbers. If the “log” option is selected for the *transform* subargument of the *action* argument of function *inform_from_gplane()*, then the way in which GPLANE PLIST elements influence off-GPLANE PLIST elements proceeds as follows. First interpolation takes place from elements of the GPLANE PLIST to the point of projection of the off-GPLANE PLIST element onto the GPLANE itself. This inverse-power-of-distance interpolation can optionally be based on the logged values of GPLANE PLIST elements, this being determined by the value of the *interp_transform* argument of the *new_gplane_clist()* function. Then, independently of whether or not in-GPLANE inverse-power-of-distance is applied to log or native GPLANE PLIST element values, the influence of the GPLANE PLIST values interpolated to this projection point on the value of an external PLIST element is calculated using equations (2.4.1) and (2.4.2) as applied to the logarithms of quantities represented in these equations, or to the native values of these quantities; the user chooses between these two options using the *transform* subargument of the *action* argument. If the *transform* value is “log”, the GPLANE value interpolated to the point of external PLIST projection is log-transformed before influence is calculated. Once its influence on the external PLIST element has been calculated, back-transformation to the latter’s natural value takes place before its new value is stored.

Example

```
pl_g(select=(sl_g==1))=                                &
    pl_p.inform_from_gplane(                             &
        select=(sl_p==1),                                &
        action='blend_never_lower';transform='log',      &
        conwidth=50.0,                                   &
        a=100.0)
```

Elements of the off-GPLANE PLIST *pl_g* for which elements of a corresponding SLIST named *sl_g* have values of 1 are informed by the GPLANE PLIST *pl_p*, but only those for which elements of a corresponding GPLANE SLIST have values of 1. *w* and *a* for the GPLANE influence function are 50.0 and 100.0 respectively. Equation 2.4.2c is used to inform selected values of the off-GPLANE *pl_g* PLIST; influence on the log of off-GPLANE

PLIST values is calculated using the logs of GPLANE values as interpolated to the respective points of external PLIST projection on the GPLANE. Back-transformation is then undertaken before native off-plane PLIST values are stored.

interp_using_file()

General

Function *interp_using_file()* is identical to function *krige_using_file()*. See documentation of that function for full usage details.

ivd_interpolate_2d()

General

Function *ivd_interpolate_2d()* undertakes spatial interpolation from one PLIST (normally representing a set of pilot points) to another PLIST (normally representing the cell centres or nodes of a model grid or mesh) using the inverse power of distance algorithm. This is not a very sophisticated interpolation algorithm. However it is fast. Furthermore where points from and to which interpolation takes place are collinear, and where points from which interpolation takes place are limited to two at a time, and where the inverse power of distance is 1.0, then interpolation is piecewise linear.

Function Specifications

Function value

The *ivd_interpolate_2d()* function calculates values for some or all elements of an existing PLIST (referred to as the “target PLIST”). The name of the target PLIST comprises the output of the function. This name must appear before the function in a PLPROC command which uses the function. This name should be separated from the function by a “=” symbol. An optional selection equation can be provided with the name of the target PLIST.

Object associations

Function *ivd_interpolate_2d()* employs source PLIST element values in calculating values for elements of a target PLIST. The name of this function must follow the name of the source PLIST on which it operates; a dot must separate these two names.

Arguments and subarguments

<i>select</i>	The source PLIST selection equation. This argument is optional. Note that a target PLIST selection equation can also be supplied if desired. The latter must be placed adjacent to the name of the target PLIST on the left side of the assignment operator; see the example below. This argument is optional.
<i>search_radius</i>	In searching for elements of the source PLIST from which interpolation takes place to a particular element of the target PLIST, a source PLIST element will NOT be used if it is situated further from the target PLIST element than this distance. Provide a very large number (e.g. 10^{15}) for an effectively infinite search radius. This argument is mandatory.
<i>max_points</i>	Regardless of the search radius, interpolation factors will be calculated for no more than the <i>max_points</i> closest source PLIST elements to any target PLIST element. This argument is mandatory.
<i>min_points</i>	This is the minimum number of source PLIST elements which must be included within the <i>search_radius</i> centred on any target PLIST element. If the search radius is not large enough to include this number of source PLIST elements, PLPROC reports the error, and then ceases

This example demonstrates use of all arguments of the *ivd_interpolate_2d()* function. PLIST elements to which interpolation takes place, and those from which interpolation takes place, are both governed by a selection equation. Elements are log transformed prior to interpolation and then back-transformed after interpolation. Limits are imposed on interpolated values. In calculating powers of distance, distances are effectively lengthened by a factor of three in directions of 135 degrees and -45 degrees with respect to north. A maximum of 20 source elements is employed in interpolating to any target element. If less than 2 source elements can be found within a radius of 2000.0 from a target element, PLPROC will cease execution with an appropriate error message.

ivd_sda_interpolate_2d()

General

“SDA” stands for “spatially dependent anisotropy”.

Function *ivd_sda_interpolate_2d()* undertakes spatial interpolation from one PLIST (normally representing a set of pilot points) to another PLIST (normally representing the cell centres or nodes of a model grid or mesh). However three interpolation exercises are actually carried out. First the ratio and bearing of anisotropy are interpolated from two PLISTs which are assumed to have a common parent CLIST. This is done using radial basis functions. Next interpolation is carried out from the actual source PLIST to the actual target PLIST. This interpolation is carried out using the inverse-power-of-distance methodology. As part of this interpolation stage, distance calculation between target and source PLIST elements is modified in accordance with the spatially varying ratio and direction of anisotropy as interpolated from the anisotropy ratio and bearing PLISTs.

Function Specifications

Function value

The *ivd_sda_interpolate_2d()* function calculates values for some or all elements of an existing PLIST (referred to as the “target PLIST”). The name of the target PLIST comprises the output of the function. This name must appear before the function in a PLPROC command which uses the function. This name should be separated from the function by a “=” symbol.

Object associations

Function *ivd_sda_interpolate_2d()* employs source PLIST element values in calculating values for elements of a target PLIST. The name of this function must follow the name of the source PLIST on which it operates; a dot must separate these two names.

Arguments and subarguments

<i>select</i>	The source PLIST selection equation. This argument is optional. Note that a target PLIST selection equation can also be supplied if desired. The latter must be placed adjacent to the name of the target PLIST on the left side of the assignment operator; see the examples below. This argument is optional.
<i>max_pts</i>	The maximum number of source PLIST elements used to interpolate to any one target PLIST element. Before eliminating source PLIST elements when interpolating to any one target PLIST element, source PLIST elements are sorted in increasing order of distance from the current target PLIST element. Hence the closest <i>max_pts</i> source elements are selected for interpolation to the target PLIST element. This argument is mandatory.
<i>max_dist</i>	The search radius (in real distance units). No source PLIST

	element which is greater than this distance from a target PLIST element is used in interpolation to that element. This argument is mandatory.
<i>inverse_power</i>	Inverse of power to which distance is raised in determining interpolation weights from the source PLIST to the target PLIST. This must be greater than zero. This is a mandatory argument.
<i>transform</i>	This must be supplied as “log” or “none”. If supplied as “log” then spatial interpolation is applied to the logs of source PLIST element values rather than to their native values. Target PLIST native element values are then back-calculated from their logs before storage. This argument is mandatory.
<i>lower_limit</i>	If an interpolated value for any element of the target PLIST is less than the value supplied for this argument, it is assigned a value equal to this <i>lower_limit</i> . If this argument is omitted, the lower interpolation limit is assumed to be a very large negative number.
<i>upper_limit</i>	If an interpolated value for any element of the target PLIST is greater than the value supplied for this argument, it is assigned a value equal to this <i>upper_limit</i> . If this argument is omitted, the upper interpolation limit is assumed to be a very large positive number.
<i>sda_report_rcond</i>	“yes” or “no”. If set to “yes” the condition number is reported to the screen when radial basis function coefficients are determined for use in spatial interpolation of anisotropy ratio and bearing data prior to inverse-power-of-distance interpolation from the source PLIST to the target PLIST. If this argument is omitted, it is assumed to be “no”.
<i>sda_bearing_plist</i>	The name of the PLIST containing anisotropy bearings. This is a mandatory argument.
<i>sda_ratio_plist</i>	The name of the PLIST containing anisotropy ratios. This is a mandatory argument.
<i>sda_rbf</i>	The name of the radial basis function type used for spatial interpolation of anisotropy data. The value supplied for this argument must be one of “ga”, “iq”, “imq”, “mq”, “lin”, “cub” or “tps”. This is a mandatory argument.
<i>sda_rbf; epsilon</i>	The value of the “epsilon” parameter employed for anisotropy ratio and bearing interpolation using radial basis functions. This subargument of the <i>sda_rbf</i> argument is mandatory.
<i>sda_anis_bearing</i>	A real number, this being the anisotropy bearing used in interpolation of anisotropy bearing and ratio data. This argument must be supplied if a value is supplied for the <i>sda_anis_ratio</i>

	argument. If omitted it is assumed to be zero. It must lie between -360 and 360 degrees.
<i>sda_anis_ratio</i>	A real number, this being the anisotropy ratio used in interpolation of anisotropy bearing and ratio data. This argument must be supplied if a value is supplied for the <i>sda_anis_bearing</i> argument. If omitted it is assumed to be 1.0. If supplied, it must be greater than zero.
<i>sda_constant_term</i>	“yes” or “no”. This determines whether a constant term is included in radial basis function interpolation of anisotropy data. If omitted it is assumed to be “no”.
<i>sda_delta</i>	The line joining any target PLIST element and any source PLIST element is divided into increments of this length in order to integrate along the line in order to determine the “geostatistical distance” between those points. This argument is mandatory. It must be a real number greater than zero.
<i>sda_target_ratio_plist</i>	If this argument is supplied, elements of the nominated PLIST will be assigned rbf-interpolated values of the spatially varying anisotropy ratio. If the PLIST already exists, it must have the same parent CLIST as that of the interpolation target PLIST. If it does not exist, it is created as such. Dummy values are provided for elements which correspond to target PLIST elements to which interpolation does not take place because of non-selection through a target element selection equation. This argument is optional.
<i>sda_target_bearing_plist</i>	If this argument is supplied, elements of the nominated PLIST will be assigned rbf-interpolated values of the spatially varying anisotropy bearing. If the PLIST already exists, it must have the same parent CLIST as that of the interpolation target PLIST. If it does not exist, it is created as such. Dummy values are provided for elements which correspond to target PLIST elements to which interpolation does not take place because of non-selection through a target element selection equation. This argument is optional.
<i>sda_bearing_upper_limit</i>	The upper limit of interpolated anisotropy bearing. This argument is mandatory. The difference between <i>sda_bearing_upper_limit</i> and <i>sda_bearing_lower_limit</i> must be less than 180 degrees.
<i>sda_bearing_lower_limit</i>	The lower limit of interpolated anisotropy bearing. This argument is mandatory. The difference between <i>sda_bearing_upper_limit</i> and <i>sda_bearing_lower_limit</i> must be less than 180 degrees.
<i>sda_ratio_upper_limit</i>	The upper limit of interpolated anisotropy ratio. This argument is mandatory. It must be a real number greater than zero.
<i>sda_ratio_lower_limit</i>	The lower limit of interpolated anisotropy ratio. This argument is

mandatory. It must be a real number greater than zero.

Discussion

General

Function *ivd_sda_interpolate_2d()* has much in common with function *rbf_sda_interpolate_2d()*. The difference between the two is that interpolation from a source to target PLIST through a domain characterized by spatially variable anisotropy is undertaken using the inverse-power-of-distance methodology rather than radial basis functions when the *ivd_sda_interpolate_2d()* function is invoked. Use of the inverse-power-of-distance method is much faster. Hence where target and/or source PLISTs are large, function *ivd_sda_interpolate_2d()* may be useable while function *rbf_sda_interpolate_2d()* may not be useable in undertaking spatially variable anisotropy interpolation. In both cases, however, interpolation of anisotropy ratio and bearing prior to source-to-target interpolation is implemented using radial basis functions.

As discussed above, use of function *ivd_sda_interpolate_2d()* actually implies three spatial interpolation exercises. In the first of these exercises the anisotropy ratio is notionally interpolated from a user-provided set of PLIST elements to all points throughout a model domain. (Actually a set of radial basis function coefficients is determined so that such interpolation can take place on an as-needed basis when calculating geostatistical distances between source and target PLIST elements in later inverse-power-of-distance interpolation). In the second of these exercises the same is done for anisotropy bearing. Finally interpolation between source and target PLIST elements is undertaken through the spatially varying anisotropy field determined by the first two interpolation exercises using inverse-power-of-distance interpolation.

Anisotropy interpolation

PLISTs containing anisotropy ratio and bearing data must be supplied to function *ivd_sda_interpolate_2d()* through its *sda_ratio_plist* and *sda_bearing_plist* arguments. These PLISTs must have a common parent CLIST (and hence pertain to the same set of pilot points). No selection equation can operate on these PLISTs; hence all elements of these PLISTs are employed in determination of a spatially varying anisotropy field.

Upper and lower bounds for interpolated anisotropy ratios and bearings must also be provided through the *sda_ratio_upper_limit*, *sda_ratio_lower_limit*, *sda_bearing_upper_limit* and *sda_bearing_lower_limit* arguments. Each of these must be provided as a single real number. Anisotropy ratio limits must be positive. The difference between the upper and lower anisotropy bearing limits must be less than 180 degrees. Recall from other sections of this manual that the anisotropy bearing is normally the direction of maximum continuity of hydraulic properties; this occurs if the anisotropy ratio is greater than 1.0. If this is the case, then hydraulic property continuity in the direction perpendicular to this bearing is correspondingly smaller than in the direction parallel to this bearing. This condition is achieved by notionally “stretching” distance in the direction perpendicular to the anisotropy bearing so that points which are aligned in this direction are further apart (and hence less “connected” when undertaking spatial interpolation).

Spatial interpolation of anisotropy ratio and bearing data is undertaken using radial basis functions. Anisotropy ratio is actually log-interpolated; however this is done behind the

scenes. The use of radial basis functions for this stage of interpolation is numerically efficient, for its implementation requires only that a set of coefficients be first computed. These are then applied on an as-needed basis later in the overall interpolation process as “geostatistical distances” between source and target PLIST elements are determined through integration along lines which separate these elements.

Use of radial basis functions as an interpolation device demands that a radial basis function type and corresponding epsilon (for those radial basis functions which require it) be supplied. These are provided through the *sda_rbf* argument and the *epsilon* subargument of the *sda_rbf* argument respectively. Optionally, radial basis function interpolation of anisotropy data can itself incorporate (spatially uniform) anisotropy. If this is required, the uniform anisotropy ratio and bearing employed for radial basis function interpolation of PLIST-supplied anisotropy ratio and bearing data can be provided through the *sda_anis_ratio* and *sda_anis_bearing* function arguments.

Optionally, radial basis function interpolation can include an accompanying constant function, this being activated through assigning the *sda_constant_term* argument a value of “yes” (for activation) or “no” (for omission). Absence of this argument signifies a default value of “no”. Linear terms are not allowed in the spatial interpolation process however.

Once coefficients have been determined for the radial basis functions, as well as for the optional constant term, interpolation can then take place to any point. If a user wishes to have interpolation of spatially-dependent anisotropy ratio and/or bearing carried out to all elements of a special target PLIST, this can be achieved through supplying values for the optional *sda_target_ratio_plist* and/or *sda_target_bearing_plist* arguments. The name of a PLIST must be supplied in both cases. If the PLIST already exists, it must have the same parent CLIST as the interpolation target PLIST, this being the PLIST whose name appears on the left of the “=” symbol in the call to function *ivd_sda_interpolate_2d()*. If the PLIST does not exist it will be created (with the target CLIST as its parent).

Interpolation from source to target PLISTs

Interpolation from the source PLIST to the target PLIST takes place using the inverse-power-of-distance methodology. Let s_j represent the value of the j^{th} element of the source PLIST. Let t_i represent the value of the i^{th} element of the target PLIST, this value being determined through interpolation from the source PLIST. Using inverse-power-of-distance interpolation, t_i is calculated as:

$$t_i = \frac{\sum_j \frac{s_j}{r_{ij}^\alpha}}{\sum_j \frac{1}{r_{ij}^\alpha}}$$

where r_{ij} is the “geostatistical distance” between target PLIST element j and source PLIST element i . This distance is lengthened (or shortened if the anisotropy ratio is less than unity) in accordance with its projection onto the direction perpendicular to the direction of principal anisotropy.

Where anisotropy is spatially variable, the geostatistical distance r between two points must be determined through integration along the line joining the two points. Integration is done numerically using the rectangle rule with an interval of *sda_delta*; hence this line is

subdivided into increments of length *sda_delta*. The geostatistical length of each increment is determined on the basis of the anisotropy ratio and bearing that prevails at its midpoint. Geostatistical increment lengths are then summed to determine the actual geostatistical separation between points at either end of the line.

The length of time required to implement interpolation between a source and target PLIST will depend heavily on the value supplied for *sda_delta*. The smaller is this incremental distance, the greater is the number of required incidences of spatial interpolation of anisotropy ratio and bearing from the user-supplied PLISTs which hold this data.

A user determines the value for α in the above inverse-power-of-distance interpolation equation through the value that he/she supplies for the *inverse_power* argument. Values less than 2 can result in pronounced “bulls eyes” in the interpolated data field. The number of source PLIST elements from which interpolation takes place to any particular target PLIST element can be limited through use of a source selection equation. It can also be limited through use of the *max_pts* and/or *max_dist* function arguments. Regardless of the number of source PLIST elements selected through the selection equation, interpolation will take place only from the closest *max_pts* selected source PLIST elements to any target PLIST element. Furthermore, if any source PLIST element is further from a target PLIST element than a distance of *max_dist* (expressed as a NON-geostatistical distance), then interpolation will not take place from that element.

Optionally, spatial interpolation can be applied to the logs of source PLIST elements. Interpolated values are back-transformed to native values before being assigned to target PLIST elements. Limits can be placed on interpolated values through use of the *upper_limit* and *lower_limit* function arguments.

The following should also be noted.

- The source PLIST can have a different parent CLIST from that of the PLISTs used to express anisotropy ratio and bearing. So too, of course, can the target PLIST.
- If desired, a selection equation can be supplied with the target PLIST; interpolation will then take place only to elements of that PLIST that are identified through that equation.
- A selection equation can also be applied to the source PLIST. However it cannot be supplied to the anisotropy ratio and bearing PLISTs.
- The target PLIST must already have been defined through previous PLPROC functions.

Examples

Example 1

```
grid_hk=pp_hk.ivd_sda_interpolate_2d(                                &
    upper_limit=1e20,                                              &
    lower_limit=1e-20,                                           &
    transform='log',                                             &
    inverse_power=2,                                             &
    max_pts=12,                                                  &
    max_dist=1.0e20,                                             &
    sda_bearing_plist=pp_b_anis,                                  &
    sda_ratio_plist=pp_r_anis,                                    &
    sda_delta=10,                                                &
    sda_bearing_upper_limit=180,                                  &
```

```

sda_bearing_lower_limit= 90,      &
sda_ratio_upper_limit=100,      &
sda_ratio_lower_limit=.1,      &
sda_rbf=mq;epsilon=0.01)

```

Using this command spatial interpolation is undertaken from the *pp_hk* PLIST to the *grid_hk* PLIST using inverse-power-of-distance interpolation with an inverse power of 2.0. The interpolation radius is effectively infinite, but only the closest 12 points must be used in interpolating to any target PLIST element.

In evaluating the geostatistical distance between source and target PLIST elements, the line increment is 10. Anisotropy ratio and bearing data are housed in the *pp_r_anis* and *pp_b_anis* PLISTs respectively. Radial basis function interpolation takes place from the elements of these PLISTs using the multiquadratic basis function with an epsilon value of 0.01 (see the description of the *rbf_interpolate_2d()* function for an explanation of this variable). In undertaking anisotropy interpolation, the anisotropy ratio is limited at its upper end by a value of 100 and at its lower end by a value of 0.1; the anisotropy bearing must never be greater than 180 degrees or less than 90 degrees.

Example 2

```

grid_hk=pp_hk.ivd_sda_interpolate_2d(      &
    upper_limit=1e20,                      &
    lower_limit=1e-20,                    &
    transform='log',                      &
    inverse_power=2,                      &
    max_pts=12,                          &
    max_dist=1.0e20,                     &
    sda_report_rcond='yes',              &
    sda_bearing_plist=pp_b_anis,          &
    sda_ratio_plist=pp_r_anis,            &
    sda_delta=10,                        &
    sda_bearing_upper_limit=180,          &
    sda_bearing_lower_limit= 90,          &
    sda_ratio_upper_limit=100,            &
    sda_ratio_lower_limit=.1,             &
    sda_rbf=mq;epsilon=0.01,             &
    sda_target_bearing_plist=grid_b_anis, &
    sda_target_ratio_plist=grid_r_anis)

```

Implementation of this function has the same outcomes as implementation of the function of example 1. However the condition number of the matrix that must be inverted in determination of radial basis function coefficients is reported to the screen. (This can be important information – see the description of the *rbf_interpolate_2d()* function.) As well as this, spatially interpolated anisotropy ratios and bearings are reported to the *grid_r_anis* and *grid_b_anis* PLISTs respectively.

Example 3

```

grid_hk select=(model_zone==3))=      &
    pp_hk.ivd_sda_interpolate_2d(      &
    upper_limit=1e20,                  &
    lower_limit=1e-20,                &
    transform='log',                  &
    inverse_power=2,                  &
    max_pts=12,                      &
    max_dist=1.0e20,                  &

```

```

sda_report_rcond='yes',           &
sda_bearing_plist=pp_b_anis,      &
sda_ratio_plist=pp_r_anis,        &
sda_delta=10,                     &
sda_bearing_upper_limit=180,      &
sda_bearing_lower_limit= 90,      &
sda_ratio_upper_limit=100,        &
sda_ratio_lower_limit=.1,         &
sda_rbf=mq;epsilon=0.01,          &
sda_target_bearing_plist=grid_b_anis, &
sda_target_ratio_plist=grid_r_anis, &
sda_anis_ratio=2.0,               &
sda_anis_bearing=30.0)

```

This example is similar to the above example. However a selection equation is employed with the target PLIST so that interpolation takes place only to those of its elements which lie in model zone 3. At the same time, radial basis function interpolation of anisotropy ratio and bearing is itself based on a uniform anisotropy ratio of 2.0 with a bearing of 30 degrees.

Example 4

```

grid_hk=pp_hk.ivd_sda_interpolate_2d(           &
    select=((pp_zone==2) || (pp_zone==3)),        &
    upper_limit=1e20,                             &
    lower_limit=1e-20,                             &
    transform='log',                               &
    inverse_power=2,                               &
    max_pts=12,                                    &
    max_dist=1.0e20,                               &
    sda_bearing_plist=pp_b_anis,                   &
    sda_ratio_plist=pp_r_anis,                     &
    sda_delta=10,                                  &
    sda_bearing_upper_limit=180,                   &
    sda_bearing_lower_limit= 90,                   &
    sda_ratio_upper_limit=100,                     &
    sda_ratio_lower_limit=.1,                      &
    sda_rbf=mq;epsilon=0.01,                       &
    sda_constant_term='yes')

```

This example is identical to example 1 except for the fact that a target source selection equation is employed. Also a constant term is used as part of the radial basis function interpolation process. The coefficient of this term is determined as part of the coefficient solution process.

krige_using_file()

General

The *krige_using_file()* function employs CLIST-to-CLIST interpolation factors computed by functions *calc_kriging_factors_2d()*, *calc_kriging_factors_auto_2d()* and *calc_kriging_factors_3d()* to undertake spatial interpolation from the elements of a source PLIST to those of a target PLIST. The reference CLIST for the target PLIST must have been the target CLIST used in calculation of the kriging factors. The reference CLIST for the source PLIST must have been the source CLIST employed in calculation of the kriging factors.

Where elements of the target PLIST to which interpolation actually takes place have been restricted through use of selection equations in the kriging factor calculation functions, the excluded values of the target PLIST are left unaltered by the *krige_using_file()* function.

Function Specifications

Function value

The *krige_using_file()* function calculates values for some or all elements of an existing PLIST (referred to as the “target PLIST”). The name of the target PLIST comprises the output of the function. This name must appear before the function in a PLPROC command which uses the function. This name should be separated from the function by a “=” symbol.

Object associations

Function *krige_using_file()* employs source PLIST element values in calculating values for elements of a target PLIST. The name of this function must follow the name of the source PLIST on which it operates; a dot must separate these two names.

Arguments and subarguments

<i>file</i>	The name of the file in which kriging factors are recorded.
<i>file; format</i>	<p>The <i>format</i> subargument of the <i>file</i> argument must be supplied as “formatted” or “binary”. In the former case an ASCII file is expected, whereas binary storage is expected in the latter case. (The user can employ “text” or “ascii” instead of “formatted” if he/she wishes, and “unformatted” instead of “binary” if he/she so desires.)</p> <p>If this subargument is omitted the kriging factor file is assumed to be formatted.</p>
<i>transform</i>	This must be supplied as “log” or “none”. If supplied as “log” then the interpolation factors are applied to the logs of source PLIST elements to calculate the logs of target PLIST elements; the latter are then back-transformed to native element values before being stored in the target PLIST.

<i>mean</i>	Expected value of the interpolated variable. This is required only if simple kriging is undertaken. See the discussion below concerning the value that a <i>mean</i> argument should take when that of the <i>transform</i> argument is “log”.
<i>lower_limit</i>	If a calculated value for any element of the target PLIST is less than the value supplied for this argument it is assigned this <i>lower_limit</i> value instead. If this argument is omitted, the lower interpolation limit is assumed to be a very large negative number.
<i>upper_limit</i>	If a calculated value for any element of the target PLIST is greater than the value supplied for this argument, it is assigned this <i>upper_limit</i> value instead. If this argument is omitted, the upper interpolation limit is assumed to be a very large positive number.

Discussion

General

Where PLISTs are large calculation of kriging factors can take a long time. However because these factors are stored in a file they need only be calculated once. These factors can then be used, and re-used, for interpolation of one or many PLISTS on the same or subsequent PLPROC runs. In undertaking spatial interpolation using the *krige_using_file()* function however, PLPROC will always check that the reference CLIST for the source PLIST specified in this function call is the CLIST on which basis kriging factors were previously calculated, and that the same applies for the target CLIST. (The name of the target and source CLISTS used in calculation of kriging factors are stored in the kriging factor file.)

From the above considerations it follows that the PLPROC script file that implements spatial interpolation using the *krige_using_file()* function may be different from that employed to calculate the interpolation factors, as the former may omit the *calc_kriging_factors_2d()*, *calc_kriging_factors_auto_2d()* and/or *calc_kriging_factors_3d()* function calls necessary for interpolation factor calculation. However the two different PLPROC scripts cannot be VERY different, in that they must reference the same CLISTS, which must have the same properties in both of these scripts. The script which employs the *krige_using_file()* function is most easily built from that which calculates the kriging factors (and probably also employs the *krige_using_file()* function to verify the integrity of the interpolation process) simply by commenting out the kriging factor construction commands while leaving other parts of the script intact. The faster-running script file thereby produced could then be employed when using PLPROC as a parameter pre-processor for a model that is being calibrated by PEST (and hence being run repetitively by PEST).

Element selection

The *krige_using_file()* function does not allow use of source or target selection equations. It is assumed that list element selection has already been carried out when calculating the interpolation factors using the selection function capabilities provided by the factor calculation functions. If further target element restriction is required when undertaking actual PLIST-to-PLIST interpolation, a temporary PLIST can first be filled; its values can then be

transferred to the final target PLIST using PLPROC's equation functionality in conjunction with an appropriate selection equation.

Transform

As stated above, interpolation can be applied to log or native PLIST values. If the logarithmic option is selected, all element values within the source PLIST must be positive. If this is not the case PLPROC will write an appropriate error message to the screen, and then cease execution.

Simple and ordinary kriging

Interpolation factors through which simple kriging is implemented are different from those which implement ordinary kriging. When calculating kriging factors, the user informs the *calc_kriging_factors_2d()*, *calc_kriging_factors_auto_2d()* and *calc_kriging_factors_3d()* functions whether simple or ordinary kriging is to be carried out. However it does not provide an element mean value to these functions (as simple kriging requires), as the mean element value is only required when interpolation actually takes place. This allows the same set of kriging factors to be used in interpolation of different kinds of data, with possibly very different mean values.

When reading the kriging factor file written by either of the *calc_kriging_factors_2d()*, *calc_kriging_factors_auto_2d()* or *calc_kriging_factors_3d()* functions, the *krige_using_file()* function determines whether the kriging factors contained in these files implement simple or ordinary kriging. If they implement simple kriging then it expects a value for the *mean* argument. It is important to note that where interpolation is carried out in the log domain (i.e. if the value of the *transform* argument is "log"), the user-supplied mean value required for implementation of simple kriging must nevertheless pertain to native element values. Logarithmic transformation of the user-supplied mean is undertaken by the *krige_using_file()* function prior to applying the simple-kriging-based interpolation factors.

Examples

Example 1

```
sm1=s1.krige_using_file(file='fac1.dat';form='binary',transform='log')
```

Using this command spatial interpolation is undertaken from the *s1* PLIST to the *sm1* PLIST using interpolation factors stored in the binary file *fac1.dat*. Each element of *sm1* is obtained by taking the antilog of a number that is calculated through multiplying pertinent logged elements of *s1* by respective interpolation factors, and then adding.

Example 2

```
sm1=s1.krige_using_file(file='fac1.dat',transform='none',
                        upper_limit=0.3, lower_limit=0.001) &
```

This is similar to example 1, except for the following:

- interpolation does not employ logged values;
- upper and lower bounds are imposed on target PLIST elements.

Example 3

```
sm1=s1.krige_using_file(file='fac1.dat',transform='none',
                        &
```

```
mean=14.5,  
upper_limit=0.3, lower_limit=0.001)
```

&

This is similar to example 2. However in this case a value is supplied for the *mean* argument in recognition of the fact that simple, rather than ordinary, kriging is implemented.

link_seglist_to_clist()

General

Elements of an existing two-dimensional CLIST can be linked to segments of a two-dimensional SEGLIST. (As presently programmed, all PLPROC SEGLISTS are two-dimensional.) Each element of a CLIST is a point in space. Often, it represents a pilot point. Values associated with pilot points are ascribed to PLISTs that are associated with the CLIST. Through SEGLIST-to-CLIST linkage, pilot points can be used to parameterize a model boundary condition of which the SEGLIST is a trace. A program such as PEST can then be used to estimate values for these pilot point parameters. Functions provided by PLPROC can then undertake linear or piecewise-constant interpolation of these pilot point values to segments comprising the SEGLIST. These interpolated values can then be transferred to model cells or elements that lie beside this trace on the basis of proximity of these cells to the segments of the SEGLIST.

The *link_seglist_to_clist()* function supports two types of SEGLIST-to-CLIST linkages. These are denoted as “midpoint” and “endpoints”.

If the “midpoint” option is selected, then a single element of a CLIST is linked to a single segment of a SEGLIST. If the “endpoints” option is selected, then two elements of a CLIST are linked to each SEGLIST segment, one to each end. These linkages are a precursor to spatial interpolation from pilot points to polylinear model boundary conditions. For the “midpoint” option, the parameter value ascribed to a pilot point is transferred to the whole segment to which it is linked (and thence to proximal model cells). For the “endpoints” option, interpolation of values ascribed to pilot points that are linked to either end of the segment is linear along a segment between its endpoints; model cells are then ascribed values on the basis of proximity to the segment. These matters are further discussed below.

Function Specifications

Function *link_seglist_to_clist()* makes no assignment. Its name must lead the PLPROC command through which it is invoked.

Arguments and subarguments

<i>seglist</i>	The name of the SEGLIST for which linkage to a CLIST is sought.
<i>clist</i>	The name of the CLIST for which linkage to a SEGLIST is sought.
<i>linkage_type</i>	The value of this argument must be supplied as “endpoints” (or “ends” for short) or “midpoint” (or “mid” for short). It specifies the nature of SEGLIST-to-CLIST linkage. The argument name can be specified as <i>linkage</i> for short in the <i>link_seglist_to_clist()</i> function call.
<i>max_dist</i>	As is described below, linkages between elements of a CLIST and segments of a SEGLIST is based on proximity of CLIST elements to either the endpoints of SEGLIST segments, or to the segments themselves (i.e. any point along the segment). The search radius that defines proximity can be restricted to a distance of <i>max_dist</i> if desired.

The *max_dist* argument is optional.

reportfile

PLPROC can record a file that tabulates the outcomes of the SEGLIST-to-CLIST linkage process. If this file is required, its name should be provided as the value of the optional *reportfile* argument.

Discussion

How linkages are made

Suppose that the SEGLIST-to-CLIST linkage type is specified as “midpoint”. Then PLPROC attempts to match individual segments comprising a SEGLIST to individual elements comprising a CLIST on a one-to-one basis. The match is unique; hence the number of segments comprising the SEGLIST must be the same as the number of elements comprising the CLIST. (An error message will be generated if this is not the case.) Each segment of the SEGLIST is linked to the closest element of the CLIST. The latter does not necessarily need to lie near the midpoint of the segment; however this may be convenient from a visual point of view. If a value is supplied for the *max_dist* argument, then a CLIST element must lie within this distance of a SEGLIST segment for the linkage to be made. If a SEGLIST segment cannot be linked to a CLIST element, or if a CLIST element cannot be linked to a SEGLIST segment, an error is reported.

Suppose that the linkage type is specified as “endpoints”. Then PLPROC links the end of each segment of the SEGLIST to the closest element of the CLIST, provided the distance between the segment endpoint and the CLIST element is less than *max_dist*. (If a value is not supplied for *max_dist*, then it is effectively set to infinity.) The same CLIST element may be linked to the ends of more than one SEGLIST segment; this happens if the end of one segment coincides with the beginning of another. However, when all linkages have been made, an error condition will arise if any CLIST element is not linked to the end of at least one SEGLIST segment. An error condition will also arise if the same CLIST element is linked to both ends of the same SEGLIST segment.

Number of linkages

PLPROC allows a single SEGLIST to be linked to two different CLISTs. These linkages must be made through two different *link_seglist_to_clist()* function calls. This can be useful in parameter estimation settings where more than one parameter type is associated with a single polylinear model feature. Thus, for example, a user may desire that linear interpolation take place along each segment when estimating the elevation of a general head boundary. However he/she may desire that the conductance of this boundary be constant along each segment. Hence the SEGLIST which traces the boundary may be endpoint-linked to one CLIST and midpoint-linked to another CLIST. Elevations are then assigned to a pilot point PLIST associated with the former CLIST while conductances are assigned to a pilot point PLIST associated with the latter CLIST.

Reporting

If a value is supplied for the *reportfile* argument, then a linkage report file is written, the name of the file being the value of the argument. An example of such a file follows.

Report of linkage between SEGLIST "rivers" and CLIST "cl_ep".

Each segment in the SEGLIST is linked to two CLIST nodes.
These nodes are the closest to either end of the segment.

Segment_name	Segment_point	Closest_CLIST_point	Distance
-----	-----	-----	-----
seg1	first	3	3.8308
seg1	last	2	7.0637
seg2	first	1	10.4210
seg2	last	2	3.8308
seg3	first	2	3.8308
seg3	last	6	2.4228
seg4	first	5	8.7356
seg4	last	6	3.8308
seg5	first	4	3.8308
seg5	last	6	3.8308
seg6	first	6	3.8308
seg6	last	7	1.7132

Example of a linkage report file.

As is apparent from the above example, information within the linkage report file makes the details of SEGLIST-to-CLIST linkage explicit. The distance between each element of the CLIST and either the endpoint of the SEGLIST segment to which it is linked (where *linkage_type* is "endpoints") or the segment itself (where *linkage_type* is "midpoint") is also listed.

Example

```
link_seglist_to_clist(seglist=rivers,           &
                     clist=cl_ep,              &
                     linkage='endpoints',      &
                     max_dist=15,              &
                     reportfile=r1.dat)
```

In the above call to function *link_seglist_to_clist()*, a SEGLIST named "rivers" is linked to a CLIST named "cl_ep". The linkage type is "endpoints". However, in seeking linkages between SEGLIST segments and CLIST elements, an error condition will arise if a CLIST element is not located within a distance of 15.0 from the end of any segment. After linkages have been made, linkage outcomes are reported to file *r1.dat*.

mremove()

General

The *mremove()* function removes an MLIST from PLPROC's memory. Optionally, it also removes all SLISTs and SLISTs which comprise the MLIST.

Function Specifications

Function value

The *mremove()* function makes no assignments. Its name must lead the PLPROC command through which it is invoked.

Object associations

Function *mremove()* operates on an existing MLIST. Its name must follow the name of the MLIST whose removal it activates; a dot must separate the MLIST name from the function name.

Arguments

<i>remove_lists</i>	The value of this argument must be "yes" or "no". If it is supplied as "yes" then all SLISTs or PLISTs (depending on the MLIST's type) which comprise the MLIST will also be removed from PLPROC's memory. This argument is optional; its default value is "no".
---------------------	--

Examples

Example 1

```
pm*.mremove()
```

If *pm** is an MLIST, it is removed from PLPROC's memory. However none of the SLISTs or PLISTs which comprise the *pm** MLIST are removed from memory. Hence they can still be used individually in later PLPROC processing.

Example 2

```
pm*.mremove(remove_lists="yes")
```

The *pm** MLIST, it is removed from PLPROC's memory. So too are all of the SLISTs or PLISTs which it cites.

new_gridded_clist()

General

As the name suggests, the *new_gridded_clist()* function creates a CLIST based on a grid. The grid can be two- or three-dimensional. Grid specifications are provided through subroutine arguments. The grid can be regular or semi-regular. It can be of the (*nx*, *ny*, *nz*) type or the (*ncol*, *nrow*, *nlay*) type. Optionally, companion SLISTs can be created which provide (*ix*, *iy*, *iz*) or (*icol*, *irow*, *ilay*) coordinates for each element of the created CLIST. See the discussion of gridded CLISTs in Section 2.3.5 of this document. As is explained in that section, grid design specifications can be summarized using the *grid_type* variable. This is a mandatory argument in the *create_gridded_clist()* function.

Function Specifications

Function value

The *new_gridded_clist()* function creates a CLIST. The name of this CLIST constitutes its output value. A call to this function must thus follow the user-specified name of the CLIST which it creates; the latter must be followed by a “=” symbol.

Arguments and subarguments

<i>grid_type</i>	This is a positive or negative integer which specifies the type of grid to which the created CLIST will pertain. A two- or three-dimensional grid can be specified. Allowed values for <i>grid_type</i> are 11, 12, 21, 111, 211, 121, 112, 212, 122, 113, 213, 123 and the negatives of all of these. This argument is mandatory.
<i>nx</i> , <i>ny</i>	The number of cells in the <i>x</i> and <i>y</i> directions. These must be supplied if <i>grid_type</i> is negative.
<i>nz</i>	The number of cells in the <i>z</i> direction. This must be supplied if <i>grid_type</i> is negative and the grid (and hence created CLIST) is three-dimensional.
<i>ncol</i> , <i>nrow</i>	The number of cells in the directions of increasing column and row index. These must be supplied if <i>grid_type</i> is positive.
<i>nlay</i>	The number of cells in the direction of increasing layer number (i.e. downwards). This must be supplied if <i>grid_type</i> is positive and the grid (and hence created CLIST) is three-dimensional.
<i>x0</i> , <i>y0</i>	The easting and northing of the (bottom, left, front) or (top, left, back) of the grid if <i>grid_type</i> is respectively negative or positive. These arguments are mandatory.
<i>z0</i>	The elevation of the (bottom, left, front) or (top, left, back) of the grid if <i>grid_type</i> is negative or positive respectively. This argument is

	mandatory if the grid is three-dimensional.
<i>rotation</i>	The angle between the direction of increasing <i>ix</i> or <i>icol</i> index and east. This argument is mandatory. It must be between -180 degrees and 180 degrees.
<i>dx</i> or <i>dx_file</i>	If <i>grid_type</i> is negative, supply a single number for the uniform grid cell width in the <i>x</i> direction if the first digit of <i>grid_type</i> is 1, or the name of a file containing <i>dx</i> values (in order of increasing <i>ix</i>) if the first digit of <i>grid_type</i> is 2. One of these must be provided, but not both.
<i>dy</i> or <i>dy_file</i>	If <i>grid_type</i> is negative, supply a single number for the uniform grid cell width in the <i>y</i> direction if the second digit of <i>grid_type</i> is 1, or the name of a file containing <i>dy</i> values (in order of increasing <i>iy</i>) if the second digit of <i>grid_type</i> is 2. One of these must be provided, but not both.
<i>dz</i> , <i>dz_file</i> or <i>elev_file</i>	If <i>grid_type</i> is negative and the grid is three-dimensional, supply a single number for the uniform grid cell width in the <i>z</i> direction if the third digit of <i>grid_type</i> is 1, the name of a file containing <i>dz</i> values (in order of increasing <i>iz</i>) if the third digit of <i>grid_type</i> is 2, or the name of a file containing a three-dimensional array of grid node elevations (read in <i>ix</i> , <i>iy</i> , <i>iz</i> order), if <i>grid_type</i> is 3. One of these must be provided, but not more than one of them.
<i>dcol</i> or <i>dcol_file</i>	If <i>grid_type</i> is positive, supply a single number for the uniform grid cell width in the direction of increasing <i>icol</i> index if the first digit of <i>grid_type</i> is 1, or the name of a file containing <i>dcol</i> values (in order of increasing <i>icol</i>) if the first digit of <i>grid_type</i> is 2. One of these must be provided, but not both.
<i>drow</i> or <i>drow_file</i>	If <i>grid_type</i> is positive, supply a single number for the uniform grid cell width in the direction of increasing <i>irow</i> index if the second digit of <i>grid_type</i> is 1, or the name of a file containing <i>drow</i> values (in order of increasing <i>irow</i>) if the second digit of <i>grid_type</i> is 2. One of these must be provided, but not both.
<i>dlay</i> , <i>dlay_file</i> or <i>elev_file</i>	If <i>grid_type</i> is positive and the grid is three-dimensional, supply a single number for the uniform grid cell width in the direction of increasing layer index if the third digit of <i>grid_type</i> is 1, the name of a file containing <i>dlay</i> values (in order of increasing <i>ilay</i>) if the third digit of <i>grid_type</i> is 2, or the name of a file containing a three-dimensional array of grid node elevations (read in <i>icol</i> , <i>irow</i> , <i>ilay</i> order), if <i>grid_type</i> is 3. One of these must be provided, but not more than one of them.
<i>ix_slist</i> , <i>iy_slist</i> , <i>iz_slist</i>	These are optional arguments. Provide the name of a new SLIST with each. The parent CLIST of the new SLIST will be that created by the

new_gridded_clist() function. The new SLIST will contain the *ix*, *iy* or *iz* grid coordinate of each CLIST element. None of these arguments should be provided if *grid_type* is positive.

icol_slist, *irow_slist*, *ilay_slist* These are optional arguments. Provide the name of a new SLIST with each. The parent CLIST of the new SLIST will be that created by the *new_gridded_clist()* function. The new SLIST will contain the *icol*, *irow* or *ilay* grid coordinate of each CLIST element. None of these arguments should be provided if *grid_type* is negative.

Discussion

When creating a gridded CLIST a user must first decide:

- Whether the grid is two- or three-dimensional. This will determine the number of digits in the *grid_type* grid specification variable.
- Whether grid elements will be indexed using the *ix*, *iy*, *iz* or the *icol*, *irow*, *ilay* convention. This will determine the sign of *grid_type*. See Section 2.3.5 of this document for repercussions of these choices.

To position the grid in space, *x0*, *y0* and *z0* (the latter only if the grid is three-dimensional) function arguments must then be established, together with the value of the *grid rotation* argument.

Next the user must decide on values for grid cell widths. If *nx*, *ny*, *nz* grid specifications are employed, then cell widths must be provided using *dx* or *dx_file*, *dy* or *dy_file*, *dz* or *dz_file* arguments. A value of 1 or 2 for the respective *grid_type* digit determines which is to be used; a value of 1 indicates uniform gridding in the pertinent direction, while a value of 2 indicates that cell widths in this direction must be read from a file. If the file option is chosen then cell widths are read in order of increasing *ix*, *iy* or *iz* index using free-field formatting. Thus these widths can be provided one to a line, all on one line, or wrapped. Similar considerations apply if *ncol*, *nrow*, *nlay* grid specifications are employed.

If the grid is three-dimensional and the third digit of *grid_type* is 3, this informs the *new_gridded_clist()* function that cell node elevations are to be supplied on a cell-by-cell basis through a file. This file is also read using the free-field format convention. Suppose that the name of the three-dimensional array holding cell elevations is *elev*. If (*nx*, *ny*, *nz*) grid specifications are employed, the array is read as:

```
((elev(ix, iy, iz), ix=1, nx), iy=1, ny), iz=1, nz)
```

However if *ncol*, *nrow*, *nlay* grid specifications are employed, the array is read as:

```
((elev(icol, irow, ilay), icol=1, ncol), irow=1, nrow), ilay=1, nlay)
```

This is an important distinction as *iy* increases toward the north while *irow* increases towards the south. Likewise *iz* increases upwards while *ilay* increases downwards.

It is important to note that the *new_gridded_clist()* function does not check that the elevations supplied in an external file result in positive vertical widths of every cell. It is the user's responsibility to ensure this. Hence if a model pre-processor does not calculate positive widths for inactive cells, this can be tolerated. Presumably an activity array can be read as an SLIST in later PLPROC processing.

Finally a user must decide whether he/she would like to create up to three SLISTS along with the CLIST which is created by the *new_gridded_clist()* function. These SLISTS are named by the user, the name of each of these being supplied as the value of the pertinent *ix_slist*, *iy_slist*, *iz_slist*, *icol_slist*, *irow_slist* or *ilay_slist* function argument. These SLISTS contain the pertinent grid coordinate of each CLIST element.

Examples

Example 1

```
cl3d=new_gridded_clist(                                &
                    grid_type=-111,                    &
                    nx=30,                             &
                    ny=40,                             &
                    nz=10,                             &
                    x0=0.0,                             &
                    y0=0.0,                             &
                    z0=0.0,                             &
                    rotation=0.0,                      &
                    ix_slist='ix',                     &
                    iy_slist='iy',                     &
                    iz_slist='iz',                     &
                    dx=25,                             &
                    dy=30,                             &
                    dz=15)
```

The above call to function *new_gridded_clist()* creates a three-dimensional, completely regular grid. The indexing protocol is (*ix*, *iy*, *iz*). The equivalent grid, but with (*icol*, *irow*, *ilay*) indexing protocol, is created using the following call to function *new_gridded_clist()*. Note the difference in *x0*, *y0* and *z0* coordinates to account for the different reference point that is employed for these two grid indexing conventions.

```
cl3d=new_gridded_clist(                                &
                    grid_type=111,                    &
                    ncol=30,                          &
                    nrow=40,                          &
                    nlay=10,                          &
                    x0=0.0,                             &
                    y0=1200.0,                        &
                    z0=150.0,                         &
                    rotation=0.0,                    &
                    icol_slist='icol',                &
                    irow_slist='irow',                &
                    ilay_slist='ilay',                &
                    dcol=25,                          &
                    drow=30,                          &
                    dlay=15)
```

Example 2

```
cl3d=new_gridded_clist(                                &
                    grid_type=-112,                    &
                    nx=30,                             &
                    ny=40,                             &
                    nz=10,                             &
                    x0=0.0,                             &
                    y0=0.0,                             &
                    z0=0.0,                             &
```

```

rotation=0.0,      &
dx=25,             &
dy=30,             &
dz_file='dz.dat')

```

This example is similar to the first example presented above. However the grid is not uniform in the z direction. Cell widths in the z direction are read from file *dz.dat*. These widths must be arranged in order of increasing iz index (i.e. in order of increasing elevation). No cell index SLISTs are requested in this case. A counterpart *new_gridded_clist()* function call for (*icol*, *irow*, *ilay*) grid indexing is shown below.

```

cl3d=new_gridded_clist(      &
    grid_type=111,           &
    ncol=30,                 &
    nrow=40,                 &
    nlay=10,                 &
    x0=0.0,                  &
    y0=1200.0,               &
    z0=150.0,                &
    rotation=0.0,            &
    dcol=25,                  &
    drow=30,                  &
    dlay_file='dlay.dat')

```

In this case vertical cell widths are supplied in file *dlay.dat*. They must be supplied in increasing *ilay* order. This is in order of decreasing (rather than increasing) elevation.

Example 2

```

cl3d=new_gridded_clist(      &
    grid_type=-113,          &
    nx=30,                   &
    ny=40,                   &
    nz=10,                   &
    x0=0.0,                  &
    y0=0.0,                  &
    z0=0.0,                  &
    rotation=0.0,            &
    dx=25,                   &
    dy=30,                   &
    elev_file='elev.dat')

```

This is similar to the above example, except that elevations for all cell nodes are read from file *elev.dat*.

new_mlist()

General

Function *new_mlist()* creates an MLIST comprised of SLISTs or PLISTs which share a common CLIST parentage and which share MLIST-compatible names. If they do not exist already, all elements of the SLISTs and PLISTs which are brought into existence by the *new_mlist()* function are provided with a single value.

Function Specifications

Function value

The *new_mlist()* function creates and/or collects a series of SLISTs or PLISTs into a single MLIST, which is created by the function. The name of the new MLIST is the target of the function. This name must appear before the function in a PLPROC command which uses the function. This name should be separated from the function by a “=” symbol.

Object associations

All SLISTs or PLISTs which are created and/or collected by the *new_mlist()* function must be associated with a common CLIST. The name of this CLIST must lead the *new_mlist()* function name in a call to that function; a dot must separate these two names.

Arguments and subarguments

<i>mlist_type</i>	Either of the text strings “slist” or “plist” must be provided as the value of this mandatory function argument.
<i>mlist_start_counter</i>	This argument is mandatory. It is the initial SLIST or PLIST index in the sequence of SLISTs or PLISTs which the target MLIST assimilates. This must be no less than 1.
<i>mlist_end_counter</i>	This argument is mandatory. It is the final SLIST or PLIST index in the sequence of SLISTs or PLISTs which the target MLIST assimilates.
<i>value</i>	This is the value that is provided to all elements of all SLISTs and PLISTs that are created by the <i>new_mlist()</i> function. This argument is mandatory except for cases where no new SLISTs or PLISTs are created because they already exist. (Elements of SLISTs or PLISTs which already exist are not altered by the <i>new_mlist()</i> function.)

Discussion

MLISTs are discussed in section 2.4. The name of an MLIST must include a single “*” character. SLISTs or PLISTs which are collected into a single MLIST are named the same as the MLIST, but with the “*” character replaced by an integer. The starting and ending values for the sequence of SLISTs or PLISTs which are assimilated into a common MLIST are provided through the *mlist_start_counter* and *mlist_end_counter* arguments.

The *new_mlist()* function creates an MLIST which collects under its embrace appropriately-named SLISTs or PLISTs. Those LISTs which already exist must have the same parent CLIST as the CLIST with which the function is associated. (If not, an error condition will occur, and will be reported.) If SLISTs or PLISTs must be created to span the indicial interval between *mlist_start_counter* and *mlist_end_counter* then the *new_mlist()* function will create these LISTs. All elements of these LISTs will be filled with a user-specified value.

Example

```
pp*=cl_pp.new_mlist(mlist_type='plist',          &
                    mlist_start_counter=2,       &
                    mlist_end_counter=5,         &
                    value=-8.888)
```

The above call to function *new_mlist()* collects PLISTs name *pp2*, *pp3*, *pp4*, and *pp5* into a single MLIST named *pp**. If any of these PLISTs already exist they will be collected into the *pp** MLIST, but only if their parent CLIST is *cl_pp*. If any of these PLISTs do not exist, they will be created. All elements of created PLISTs will be given a value of -8.8888.

new_gplane_clist()

General

The *new_gplane_clist()* function creates a GPLANE and an associated CLIST, the latter pertaining to a regular grid of *grid_type* -11 defined on the GPLANE. Optionally, the function writes a report file in which the equation of the plane which supports the GPLANE, as well as local and global coordinates of the GPLANE CLIST, and possibly GPLANE boundaries, are recorded. If requested, the *new_gplane_clist()* function also writes a so-called “legacy VTK file”. The latter can be imported into a platform such as PARAVIEW so that the user can view the GPLANE grid, GPLANE grid cell centres (these corresponding to CLIST element locations) and, if applicable, GPLANE boundaries – all using real-world coordinates. Hence the GPLANE can be visualized in the context of the model whose parameterization it informs.

The GPLANE concept is fully described in Section 2.4 of this manual.

Function value

The name of the CLIST created by the *new_gplane_clist()* function is the output of the function. This name must appear before the function in the command which invokes it. The name should be separated from the function name by a “=” symbol.

Arguments

<i>gplane</i>	The name of the GPLANE created by the function. This argument is mandatory.
<i>gplane_file</i>	The name of a file from which GPLANE specifications are read. This argument is mandatory.
<i>grid_interp</i>	This argument is optional. This determines the methodology used to interpolate from a GPLANE-based PLIST to points of projection of non-GPLANE PLISTs onto the GPLANE. As presently programmed, it must be set to “ivd” (for “inverse power of distance”) if it is provided.
<i>grid_interp;</i> <i>inverse_power</i>	If the <i>grid_interp</i> argument is provided, then so too must the <i>inverse_power</i> subargument. If the <i>grid_interp</i> argument is omitted, an <i>inverse_power</i> of 2.0 is assumed.
<i>interp_transform</i>	This mandatory argument must be set to “log” or “none”. If set to “log” then log-transformed, rather than native, PLIST values are interpolated to points of non-GPLANE PLIST projection onto the GPLANE. After interpolation has taken place, these values are back-transformed to native values.
<i>interp_anis_ratio</i>	In interpolating from a GPLANE PLIST to the point of projection of an external PLIST onto the GPLANE, an assumption can be invoked of greater continuity of GPLANE properties in one direction than in an

	orthogonal direction. If this is the case, the value of the <i>interp_anis_ratio</i> argument must provide the ratio of continuity distance in one direction to that in the perpendicular direction. If this argument is omitted, its value is assumed to be 1.0.
<i>interp_anis_bearing</i>	This argument must be provided in a call to function <i>new_gplane_clist()</i> if the <i>interp_anis_ratio</i> argument is provided. The bearing is taken with respect to the local GPLANE y axis; it is positive for rotation towards the local GPLANE positive x axis (even if the GPLANE coordinate system is not right-handed). In carrying out interpolation from GPLANE PLIST locations to external PLIST projection points, distances are multiplied by <i>interp_anis_ratio</i> in the direction perpendicular to this bearing.
<i>tolerance</i>	This optional argument is used to define a separation below which two points are declared to coincide. It can be used to accommodate loss of precision in coordinates supplied in the GPLANE specification file. If this argument is omitted, then <i>tolerance</i> is assumed to be a very small value. The name of this argument can be supplied as <i>tol</i> for short.
<i>slist</i>	Up to five <i>slist</i> arguments can be included in a call to function <i>new_gplane_clist()</i> . All are optional. The value supplied for each argument must be the name of a new SLIST which the function will create. Values of SLIST elements are read from the GPLANE specification file; see below.
<i>plist</i>	Up to five <i>plist</i> arguments can be included in a call to function <i>new_gplane_clist()</i> . All are optional. The value supplied with each argument must be the name of a new PLIST which the function will create. Values of PLIST elements are read from the GPLANE specification file; see below.
<i>report_file</i>	Optionally, the <i>new_gplane_clist()</i> function writes a report file in which it lists some GPLANE construction details. If this argument is omitted, no report is generated.
<i>report_file;</i> <i>position</i>	If this optional subargument of the <i>report_file</i> argument is supplied as “append”, then the GPLANE report is written at the end of an existing file, namely that provided as the value of the <i>report_file</i> argument - if this file exists; if the file does not exist it is created. Alternatively, if the value of this subargument is supplied as “rewind”, an existing file is over-written. If this subargument is not included with the <i>report_file</i> argument, its value is assumed to be “rewind”.
<i>vtk_file</i>	Optionally, the <i>new_gplane_clist()</i> function writes a “legacy VTK” file which, once imported into appropriate visualization software, allows visualization of the GPLANE grid, GPLANE grid cell centres, and (if bounded) the GPLANE bounding polygon.

<i>x1, x2, x3</i>	Optionally, the user can provide the <i>x</i> coordinate of one or more of the three GPLANE-defining points (x_{p1} , x_{p2} , x_{p3} in the discussion below) as the value of one of these <i>new_gplane_clist()</i> function arguments. Coordinates supplied in this way override those supplied in the GPLANE specification file.
<i>y1, y2, y3</i>	Optionally, the user can provide the <i>y</i> coordinate of one or more of the three GPLANE-defining points (y_{p1} , y_{p2} , y_{p3} in the discussion below) as the value of one of these <i>new_gplane_clist()</i> arguments. Coordinates supplied in this way override those supplied in the GPLANE specification file.
<i>z1, z2, z3</i>	Optionally, the user can provide the <i>z</i> coordinate of one or more of the three GPLANE-defining points (z_{p1} , z_{p2} , z_{p3} in the discussion below) as the value of one of these <i>new_gplane_clist()</i> arguments. Coordinates supplied in this way override those supplied in the GPLANE specification file.

Discussion

GPLANE specification file

GPLANES are discussed in detail in Section 2.4 of this manual. The *new_gplane_clist()* function reads specifications for the new GPLANE, and for the new CLIST which is associated with this GPLANE, from a file whose name is provided as the value of the *gplane_file* argument. The following figure provides specifications for this file. The figure after that provides an example of such a file.

```
* gplane definition
xp1 yp1 zp1
xp2 yp2 zp2
xp3 yp3 zp3
* gplane reference coordinates
x y
or
x z
or
y z
* gplane limits
N
xb1 yb1 or xb1 zb1 or yb1 zb1
xb2 yb2 or xb2 zb2 or yb2 zb2
..
xbN ybN or xbN zbN or ybN zbN
* gplane grid
xg0 yg0 or xg0 zg0 or yg0 zg0
xg1 yg1 or xg1 zg1 or yg1 zg1
xg2 yg2 or xg2 zg2 or yg2 zg2
nx dx
ny dy
* plist
val1, val2, etc.
* slist
val1, val2, etc.
```

Specifications of a GPLANE specification file.

```
* gplane definition
10000.0  2000.0  530.0
10000.0  42000.0  530.0
20000.0  2000.0   0.0
* gplane reference coordinates
x  y
* gplane limits
4
10000.0  10000.0
10000.0  40000.0
30000.0  40000.0
30000.0  10000.0
* gplane grid
10000.0  2000.0
20000.0  2000.0
10000.0  42000.0
10  1000
20  2000
```

Example of a GPLANE specification file.

A GPLANE specification file is divided into sections. Each section contains a header; the header text begins with the “*” character followed by a space followed by the pertinent header name. Headers must be as shown in the above figures and must be listed in the same order. All sections shown above, except for the “* slist” and “* plist” sections, must be present within any GPLANE specification file; the “* slist” and “* plist” sections are optional.

The “* gplane definition” section must contain three lines. Each of these lines must contain three entries, these comprising the (x , y , z) coordinates of a point on the plane. The three points that are provided in this section of the GPLANE specification file define the plane; the plane passes through all of these points.

The next section is labelled “* gplane reference coordinates”. This section contains one line of data. It specifies the two coordinate types that will be used in all further GPLANE specifications provided in the GPLANE specification file. Entries in the single line comprising this section must be “x y”, “x z” or “y z”. As only two coordinates are required to specify a point on a plane, the third can be calculated automatically from the other two. Real world coordinates must be provided in all cases.

The “* gplane limits” section follows. The first line within this section must contain an integer which specifies the number of lines to follow. This number can range between 0 and 10 (as presently programmed). If the number is 0, then the GPLANE has no boundary-limiting polygon; it effectively extends to infinity in all directions. However suppose that this entry is N . Then N data lines must follow. Each line must contain two entries, these being the (x , y) (x , z) or (y , z) coordinates of the vertex of a polygon that defines the GPLANE’s limits. The polygon must not close; the *new_gplane_clist()* function closes the polygon itself by joining the last point to the first. Segments must not cross; if they do cross, an error condition will occur and PLPROC will cease execution with an appropriate message. If more than zero points are provided, a minimum of three bounding points must be provided.

Next follows the “* gplane grid” section. This section must contain 5 lines of data. The first is the real-world (x_0, y_0) , (x_0, z_0) or (y_0, z_0) coordinates of the origin of the GPLANE local coordinate system. This is also the origin of the GPLANE grid (i.e. the coordinates of the lower left corner of cell (1,1) of the grid). The second line of the “* gplane grid” section must provide the real-world (x_1, y_1) , (x_1, z_1) or (y_1, z_1) coordinates of a second point. The line from the origin to this point defines the direction of the local GPLANE coordinate system x axis (which is also the x -axis of the GPLANE grid). The third line of the “* gplane grid” section specifies the real-world coordinates of a third point, these being (x_2, y_2) , (x_2, z_2) or (y_2, z_2) . The location of this point does not have to be exact. It should specify the approximate direction of the GPLANE grid y axis from the GPLANE coordinate system origin. The actual direction of this axis is perpendicular to the GPLANE grid x axis, within the GPLANE itself. The location of this user-provided point only needs to differentiate between the two possibilities of GPLANE axis directions which are perpendicular to the GPLANE x axis.

The last two lines of the “* gplane grid” section of the GPLANE specification file specify the number of GPLANE grid cells in the x -direction and the x -direction width of those cells (first ensuing line), and the number of GPLANE grid cells in the y -direction and the y -direction width of those cells (second ensuing line).

Optionally, up to 5 PLISTs and up to 5 SLISTs can be defined using a GPLANE file. The parent CLIST of all such LISTs is that pertaining to the GPLANE grid. Hence the number of elements associated with any of these LISTs is the number of rows of the GPLANE grid multiplied by the number of columns of the GPLANE grid.

Providing the values of PLIST and/or SLIST elements within a GPLANE specification file is optional. If they are, in fact provided, the values of an individual PLIST must be placed in an individual “* plist” section of the GPLANE specification file. Similarly the values of an individual SLIST must be placed in an individual “* slist” section of the GPLANE specification file. The values of all elements are read using list-directed input. Hence they must be placed side by side and separated from each other by commas or whitespace, wrapping onto successive lines as necessary, until all have been provided. As their parent CLIST is a grid of *grid_type* -11, their ordering should be *ix* followed by *iy*.

Just because a PLIST or SLIST is recorded on a GPLANE specification file, it does not follow that it must actually be read by the *new_gplane_clist()* function. Reading of an SLIST or PLIST takes place only through *slist* and *plist* arguments associated with this function. It is through these arguments that the LISTs are named. They are read from the GPLANE specification file in the same order as that in which respective arguments appear in the *new_gplane_clist()* function. Hence, for example, the third occurrence of a *plist* argument reads the third occurrence of a PLIST from the GPLANE specification file. Intervening SLISTs are ignored unless pertinent *slist* arguments require that they also be read. If a third PLIST is not encountered in a GPLANE specification file, an error condition is precipitated. If more than three PLISTS are recorded in a GPLANE specification file, the *new_gplane_clist()* function is unconcerned.

Location of the GPLANE

As is described above, a GPLANE is located in space using the three points whose coordinates are provided in the “* plane definition” section of the GPLANE specification file. Optionally, any or all of these coordinates can be over-ridden through use of an *x1*, *x2*, *x3*, *y1*, *y2*, *y3*, *z1*, *z2*, or *z3* argument in a call to function *new_gplane_clist()*. The numerals

associated with these arguments pertain to the order in which these points are provided in the GPLANE specification file. Thus $x1$, $y1$ and $z1$ refer to the first provided point, $x2$, $y2$ and $z2$ refer to the second provided point, and so on.

Within-GPLANE interpolation

As is described in Section 2.4 of this manual, when a PLIST element pertaining to an off-GPLANE point is informed by a GPLANE PLIST, that point is first orthogonally projected onto the GPLANE. If the projection lies outside the polygonal boundary of the GPLANE, then a further within-plane projection is undertaken onto the closest GPLANE boundary. Interpolation from the PLIST associated with the GPLANE grid then takes place to the projection point. A stretched Gaussian function centred at that point is then employed to inform the external PLIST value. As presently programmed, within-GPLANE interpolation is accomplished using the inverse-power-of-distance methodology. Optionally, the user can supply this power as a subargument of the *grid_interp* argument.

The mandatory *interp_transform* argument of the *new_gplane_clist()* function informs this function whether inverse-power-of-distance interpolation is applied to PLIST elements themselves, or to the logarithms of these elements. In the latter case, back-transformation to a native non-logarithmic value is then undertaken at the point of external PLIST projection.

Optionally inverse-power-of-distance interpolation can be anisotropic. A bearing for the axis of anisotropy must be supplied through the *interp_anis_bearing* argument; “north” for the purposes of supplying this bearing is the local y axis of the GPLANE coordinate system. The anisotropy bearing normally defines the direction of maximum elongation of GPLANE property heterogeneity. Hence distances perpendicular to this direction are effectively lengthened in carrying out anisotropic inverse-power-of-distance interpolation; they are lengthened by a factor equal to the value of the *interp_anis_ratio* argument.

Reporting

Optionally the *new_gplane_clist()* function records GPLANE data in a report file. The user can thus check that GPLANE data has been correctly imported from the GPLANE specification file, and that these specifications suit his/her requirements. Information recorded in the report file includes the following:

- the equation for the plane in which the GPLANE lies (in global coordinates);
- global coordinates for the origin of the local, two-dimensional, GPLANE coordinate system;
- global and local coordinates of GPLANE CLIST elements (these pertaining to the centres of GPLANE grid cells);
- global and local coordinates of the polygon forming the boundary of the GPLANE (if the GPLANE does, indeed, possess a boundary); note that local GPLANE coordinates must be used in all future references to the GPLANE CLIST.

Optionally, the *new_gplane_clist()* function will also write a so-called “legacy VTK file”; this file can be imported into a visualization package such as PARAVIEW. Using this file, the following items can be viewed:

- the GPLANE plane, but only if it has a boundary;
- lines forming the GPLANE grid;
- GPLANE grid cell centres, these defining the locations of GPLANE CLIST elements.

To allow switching-on and switching-off of visualization of different displayed GPLANE elements, a number of VTK “scalars” are recorded in the VTK file; these scalars are associated with different GPLANE elements. Associations are as listed in the following table. All scalars are integers.

Name of scalar	Values
Object_type_index	3 for points describing CLIST element locations 2 for the GPLANE grid lines 1 for the GPLANE itself (if it is bounded)
ix	ix coordinates of gridded CLIST elements 0 for GPLANE grid lines 0 for the GPLANE itself
iy	iy coordinates of gridded CLIST elements 0 for GPLANE grid lines 0 for the GPLANE itself

Data values associated with objects recorded in a VTK file written by the *new_gplane_clist()* function.

Examples

Example 1

```

clist=new_gplane_clist(                                     &
    gplane='gp1',                                          &
    gplane_file='gplane.dat',                             &
    interp_transform='log')

```

In this example a new gridded CLIST named *clist* is created. This CLIST is associated with a GPLANE named *gp1* which is created at the same time. GPLANE specifications are read from a file named *gplane.dat*. Inverse-power-of-distance interpolation (with a default inverse power of 2) is applied to the log of PLIST values associated with the *clist* GPLANE CLIST to projection points of external PLISTs on the GPLANE.

Example 2

```

clist=new_gplane_clist(                                     &
    vtk_file='plane.vtk',                                  &
    gplane='gp1',                                          &
    gplane_file='gplane.dat',                             &
    interp_transform='log',                               &
    interp_anis_ratio=2.0,                                 &
    interp_anis_bearing=30.0,                             &
    report_file='report.dat',                             &
    tol=.01,                                              &
    grid_interp=ivd;inverse_power=2.5)

```

This is similar to the previous example. However in this case a legacy VTK file named *plane.vtk* is written, together with a report file named *report.dat*. The inverse power of distance is altered to 2.5. In setting up the GPLANE coordinate system, as well as the GPLANE grid and GPLANE boundaries, two user-supplied points are decreed to be coincident if their separation is calculated to be less than 0.1.

Example 3

```

clist=new_gplane_clist(                                     &
    vtk_file='plane.vtk',                                  &

```

```

gplane='gp1',                                &
gplane_file='gplane.dat',                    &
interp_transform='log',                      &
interp_anis_ratio=2.0,                      &
interp_anis_bearing=30.0,                   &
report_file='report.dat',                   &
plist=pl1,                                  &
slist=sl1,                                  &
slist=sl2,                                  &
z1=23.0,                                    &
z2=36.5,                                    &
z3=5.5,                                    &
tol=10.0,                                   &
grid_interp=ivd;inverse_power=2.5)

```

This example expands on the previous example. In-GPLANE interpolation assumes an anisotropy of 2.0, with the direction of maximum elongation of the anisotropy ellipse being 30 degrees rotated from the GPLANE y axis towards the GPLANE x axis.

The z coordinates of the three GPLANE defining points are modified from those supplied in the GPLANE specification file to values of 23.0, 36.5 and 5.5; the ordering of “1”, “2” and “3” as applied to the $z1$, $z2$ and $z3$ function arguments corresponds to that in which these coordinates are supplied in the GPLANE specification file. In addition to GPLANE specification data, a PLIST and two SLISTs are read from file *gplane.dat*. These are named *pl1*, *sl1*, and *sl2* respectively.

new_plist()

General

As the name implies, the *new_plist()* function creates a new PLIST. The reference CLIST for the new PLIST must already be stored in PLPROC memory. All elements of the new PLIST are assigned a user-specified value. These values can, of course, be wholly or partially overwritten in further PLPROC processing.

Function Specifications

Function value

The name of the PLIST created by the *new_plist()* function is the output of the function. This name must appear before the function in the command which calls it. The name should be separated from the function name by a “=” symbol.

Arguments

reference_clist The name of the reference CLIST for the new PLIST.

value The value assigned to all elements of the newly-created PLIST.

Example

```
hk=new_plist(reference_clist=model_grid,value=1.0e-3)
```

In the above example a new PLIST named *hk* is created. The reference CLIST for this new PLIST is *model_grid*; this CLIST must already be in PLPROC's memory. All elements of the new *hk* PLIST are assigned a value of 1.0e-3.

new_slist()

General

As the name implies, the *new_slist()* function creates a new SLIST. It provides two mechanisms for achieving this. One option is the same as that offered by the *new_plist()* function, namely provision of a reference CLIST (which must already reside in PLPROC's memory) together with a value to be assigned to all elements of the new SLIST. The other option is to provide the name of an existing PLIST. The *new_slist()* function then creates an SLIST whose reference CLIST is the same as that of the user-provided PLIST. Each element of the new SLIST is assigned a value calculated as the nearest integer to the value of the corresponding PLIST element.

Function Specifications

Function value

The name of the SLIST created by the *new_slist()* function is the output of the function. This name must appear before the function in the command which calls it. The name should be separated from the function by a “=” symbol.

Arguments

<i>plist</i>	The name of an existing PLIST. Element values of the new SLIST are the integers nearest to values of corresponding elements of this PLIST.
<i>reference_clist</i>	The name of the reference CLIST for the new SLIST. This argument cannot be used if the <i>plist</i> argument is used.
<i>value</i>	The integer value assigned to all elements of the newly-created SLIST. Alternatively the text string “index” or “int_id” can be provided as the value of this argument; see the discussion below. This argument must be provided if the <i>reference_clist</i> argument is used; it must not be provided if the <i>plist</i> argument is employed.

Discussion

As stated in the above table of *new_slist()* function arguments, the value of the *value* argument can be an integer, or the text “index”, or the text “int_id”. If an integer is supplied, then all elements of the newly created CLIST are provided with a value equal to that integer. In contrast, use of the “index” text informs PLPROC that each element of the SLIST must be given a value equal to the index of the respective LIST element. For the special case where elements can also be identified using a user-supplied integer (see CLIST specifications earlier in this manual), SLIST elements can be given values equal to element integer identifier values; this occurs if the string “int_id” is provided as the value of the *value* argument.

Examples

Example 1

```
zonation1=new_slist(reference_clist=model_grid,value=1)
```

This command creates a new SLIST named *zonation1*. The reference CLIST for this new SLIST is *model_grid*; this CLIST must already reside in PLPROC's memory. All elements of the new *zonation1* SLIST are assigned a value of 1.

Example 2

```
zonation2=new_slist(plist=temp)
```

In this example a new SLIST named *zonation2* is created. Its reference CLIST is the same as that of the existing PLIST *temp*. Values for elements of the new *zonation2* SLIST are calculated from those of the *temp* PLIST using the "nearest-integer" function.

parameter_delimiter()

General

The *parameter_delimiter()* function is not so much a function as an assignment statement. This statement can only appear on a template of a model input file where it takes the role of an embedded function. Hence the leading characters on any line on which it is featured in that file must be “\$#p”.

Through the *parameter_delimiter* statement a value is assigned to the character which is used to define the beginning and end of a parameter space. See Chapter 5 of this manual for further details.

Function Specifications

Function value

The *parameter_delimiter()* function makes no assignment (except to itself).

Arguments

The *parameter_delimiter()* function has no arguments.

Examples

Example 1

The following fragment of a template file illustrates a number of features of the *parameter_delimiter()* function. Note in particular the following.

- The character used as the parameter delimiter can be changed throughout the processing of a template file through making successive calls to this function.
- The character denoted as the parameter delimiter may, or may not, be surrounded by single or double quotes when making the assignment.
- Even though they have other roles, the “#” and “&” characters can be used as the parameter delimiter. However when denoting them as such it is good practice to surround them by quotes so that PLPROC does not misinterpret their role when it is processing the *parameter_delimiter* function.
- Note the spelling of *delimiter*. The correct spelling is NOT “delimeter”.

```
.
.
Table of input data
$#p parameter_delimiter = $
3.4532          $   par1      $      48.5323
23.423          $   par2      $      34.4223
$#p parameter_delimiter = "&"
34.232          &   par3      &      38.3234
2.3344          &   par4      &      48.3249
.
.
```

Part of a template file showing use of the *parameter_delimiter()* function.

Example 2

The following example demonstrates the traditional way of defining a parameter delimiter in a template file, namely by placing it after the “ptf” header string at the top of the template file with a space between “ptf” and the parameter delimiter character. PLPROC supports this method. While a PLPROC template file does not require a “ptf” header string, PLPROC will detect its presence, and that of the ensuing parameter delimiter, if it is provided. Note, however, that the parameter delimiter defined in this way can be altered through a subsequent *parameter_delimiter()* call.

```
ptf $  
$ p1[1]      $ $ p2[1]      $  
$ p1[2]      $ $ p2[2]      $  
$ p1[3]      $ $ p2[3]      $  
$#p parameter_delimiter = ~  
~ p1[4]      ~ ~ p2[4]      ~  
~ p1[5]      ~ ~ p2[5]      ~  
~ p1[6]      ~ ~ p2[6]      ~  
.  
.
```

Part of a template file showing two ways to denote a parameter delimiter.

partition_by_clist()

General

The *partition_by_clist()* function creates a new SLIST or PLIST through excision of a sublist from a larger, pre-existing SLIST or PLIST.

It is assumed that at least two CLISTS are stored within PLPROC's memory, namely that associated with an existing SLIST or PLIST from which partitioning is required, and that which is to be associated with the new SLIST or PLIST. The two CLISTS must employ compatible element identification protocols, with the smaller one containing a subset of elements of the larger one. The SLIST or PLIST created through the *partition_by_clist()* function adopts the smaller CLIST as its parent CLIST.

Function Specifications

Function value

The *partition_by_clist()* function defines and constructs a new SLIST or PLIST. The name of the new SLIST or PLIST is the output of the function. This name must appear before the function in the command which calls it. The name should be separated from that of the function by a "=" symbol.

Object associations

Function *partition_by_clist()* operates on an existing SLIST or PLIST. A call to this function must follow the name of the SLIST or PLIST whose partitioning it governs; a dot must separate the two.

Arguments

<i>clist</i>	The name of the CLIST on which partitioning of the existing SLIST or PLIST is based.
--------------	--

Example

```
hk_lay2 =hk_all.partition_by_clist(clist=layer2)
```

Prior to implementing the above command a CLIST named *layer2* must have been created. Furthermore, it must be a sublist of the reference CLIST of the *hk_all* PLIST. This latter CLIST is not mentioned by name in the *partition_by_clist()* command; however it is, of course, known to PLPROC. The *layer2* CLIST may have been created from the *hk_all* parent CLIST using the *partition_by_eqn()* or *partition_by_file()* functions. Or it may have been created by the user through other means (in which case caution should be exercised to ensure element identification compatibility between the two CLISTS).

Because *hk_all* is a PLIST, PLPROC knows that the new entity which it creates (i.e. *hk_lay2*) must also be a PLIST. If *hk_all* had instead been an SLIST, PLPROC would have created an SLIST.

In undertaking partitioning of the existing *hk_all* PLIST to create the new *hk_lay2* PLIST, PLPROC examines the identifier of each element of the *layer2* CLIST cited as the

partition_by_clist() argument. On the assumption that the parent CLIST of *hk_all* uses the same element identification protocol as that of *layer2* (which can be indexed, integer or character), PLPROC then links every element of the newly-created *hk_lay2* PLIST to an element of the *hk_all* PLIST on the basis of parent CLIST element identification equivalence. Element values of *hk_all* are transferred to their *hk_lay2* element counterparts on this basis.

If any element identifiers featured in the *layer2* CLIST are missing from the *hk_all* parent CLIST, PLPROC ceases execution with an appropriate error message.

The *partition_by_clist()* function makes special provision for sub-CLISTs whose element identification protocol is integer, which were created from parent CLISTs whose element identification protocol is indexed. This situation can sometimes be an outcome of CLIST partitioning using the *partition_by_eqn()* or *partition_by_file()* functions. As element identification for both CLISTs ultimately relies on integers, identification compatibility is maintained.

partition_by_eqn()

General

Through use of the *partition_by_eqn()* function, a new CLIST is created as a sublist of an existing CLIST. The new CLIST inherits all of the properties of the existing CLIST. In particular, it inherits its dimensionality and its coordinates. If the element identifier type of the existing CLIST is integer or character, the integer and character identifiers of the existing CLIST are also transferred to the new CLIST. However if the element identifier type of the existing CLIST is indexed, then the new CLIST's element identification protocol is indexed only if the selection equation creates the new CLIST from a contiguous block of existing CLIST elements, thereby maintaining a continuous indexing sequence. If this is not the case, the new CLIST's identification type is declared as integer. Indexing of the new CLIST then begins at 1; however integer identifiers for elements of the new CLIST are defined (and stored) as the indices of respective elements of the CLIST from which they were extracted.

Function Specifications

Function value

The *partition_by_eqn()* function defines and constructs a new CLIST. The name of the new CLIST is the output of the function. Its name must appear before that of the function in the command which calls the function. The name of the new CLIST should be separated from the function by a "=" symbol.

Object associations

Function *partition_by_eqn()* operates on an existing CLIST. The name of the function must follow the name of the CLIST whose partitioning it governs; a dot must separate the two.

Arguments

select The value of the *select* argument is the equation that is employed for CLIST partitioning. This must have a logical outcome.

Discussion

When a new CLIST is formed through partitioning of an existing CLIST, no dependent SLISTs or PLISTs are defined through partitioning of SLISTs or PLISTs for which the original CLIST is the reference CLIST; only the new "childless" CLIST is created. Partitioning of existing SLISTs and PLISTs in a similar manner to that which was accomplished to create the new CLIST, to thereby obtain new SLISTs and PLISTs for which the new CLIST is their reference CLIST, can be accomplished using the *partition_by_clist()* function.

As discussed above, it is possible that the element identification protocol of the new CLIST is different from that of the old CLIST. To check its indexing and identifier status, use the *report_dependent_lists()* function.

If partitioning according to the user-supplied selection equation results in an empty CLIST, PLPROC will cease execution with an appropriate error message.

Example

```
cll_sub = cll.partition_by_eqn(select=((z>5*scalar1)&&(int_id>20)))
```

In the above example the *cll* CLIST is partitioned using the depicted selection equation. Selection equation syntax, and the types of variables that may appear in a selection equation, are discussed in Chapter 4 of this document. These variables include CLIST coordinates and CLIST integer identifiers. The outcome of the selection equation must be logical.

The new CLIST is named *cll_sub*.

partition_by_file()

General

Through use of the *partition_by_file()* function a new CLIST is created as a sublist of an existing CLIST. The new CLIST inherits all of the properties of the existing CLIST. In particular, it inherits its dimensionality and its coordinates. If the element identifier type of the existing CLIST is integer or character, the integer and character identifiers of the existing CLIST are transferred to the new CLIST. However if the element identifier type of the existing CLIST is indexed, then the new CLIST's element identification protocol is indexed only if file-based partitioning creates the new CLIST from a contiguous block of existing CLIST elements, thereby maintaining a continuous indexing sequence. If this is not the case, the new CLIST's identification type is declared as integer. Indexing of the new CLIST then begins at 1; however integer identifiers for elements of the new CLIST are defined (and stored) as the indices of respective elements of the CLIST from which they were extracted.

Where CLIST partitioning is implemented using the *partition_by_file()* function, elements of the existing CLIST that are retained in creation of the new CLIST are read from a user-nominated file.

Function Specifications

Function value

The *partition_by_file()* function defines and constructs a new CLIST. The name of the new CLIST is the output of the function. Its name must appear before the function in the command which calls the function. The name of the new CLIST should be separated from the function name by a “=” symbol.

Object associations

Function *partition_by_file()* operates on an existing CLIST. Its name must follow the name of the CLIST whose partitioning it governs. A dot must separate the two names.

Arguments

<i>file</i>	The name of the tabular data file containing element identifiers which are used as a basis for CLIST partitioning.
<i>idcolumn</i>	The column of the table recorded in the file which contains CLIST element identifiers.
<i>skiplines</i>	The number of lines which should be skipped at the top of the file before encountering the column of element identifiers which will be used as a basis for CLIST partitioning. If this argument is omitted, a value of 0 is assumed.

Discussion

The file which the *partition_by_file()* function uses as a basis for CLIST partitioning must contain a list of element identifiers. If desired, these can comprise part of a multi-column table. Alternatively the list can exist on its own (in which case it effectively comprises the first column of a single-column table). If the CLIST to be partitioned employs the indexed or integer element identification protocol, then the entities appearing in this column must be integers. Alternatively, if the CLIST to be partitioned employs character identifiers, the entities appearing in this column will be interpreted as character strings. Any element whose identifier appears in this identifier column is retained in the new CLIST. If an integer or character string appearing in the identifier column does not identify an element of the user-specified CLIST, PLPROC will cease execution with an appropriate error message.

If the table or single column providing element identifiers is preceded by a header, the *skiplines* argument must be employed to skip header lines.

When a new CLIST is formed through partitioning of an existing CLIST, no dependent SLISTs or PLISTs are defined through partitioning of SLISTs or PLISTs for which the original CLIST is the reference CLIST; only the new “childless” CLIST is created. Partitioning of existing SLISTs and PLISTs in a similar manner to that which was accomplished to create the new CLIST, to thereby obtain new SLISTs and PLISTs for which the new CLIST is their reference CLIST, can be accomplished using the *partition_by_clist()* function.

As discussed above, it is possible that the element identifier type of the new CLIST may be different from that of the old CLIST. To check its indexing and/or identifier status, use the *report_dependent_lists()* function.

Example

```
cl2_sub=cl1.partition_by_file(file=select.dat,idcolumn=2,skiplines=3)
```

In the above example a new CLIST is excised from the existing *cl1* CLIST using element identifiers contained in the second column of a table that is resident in file *select.dat*. The table begins on the fourth line of the file.

The new CLIST is named *cl2_sub*.

rbf_interpolate_2d()

General

Function *rbf_interpolate_2d()* calculates values for the elements of one PLIST from those of another using two-dimensional radial basis functions as a device for performing spatial interpolation. In many cases of model usage, the source PLIST comprises a set of pilot points while the target PLIST comprises the nodes of a numerical model grid or mesh. The interpolation process can be confined to a subset of either list using source and/or target selection equations.

Because interpolation is based on two-dimensional basis functions, the z coordinates associated with both source and target PLIST elements are ignored in carrying out the interpolation.

Function Specifications

Function value

The name of the PLIST to which interpolation takes place (i.e. the target PLIST) comprises the output of function *rbf_interpolate_2d()*. This name must appear before the function name in a PLPROC command which invokes the function; it should be separated from the function name by a “=” symbol. Optionally, a target selection equation can be supplied with the name of the target PLIST. This PLIST should have been created in previous PLPROC processing as it is not created by the *rbf_interpolate_2d()* function itself. Target PLIST elements which are not assigned values because of the presence of a target selection equation retain their existing values.

Object associations

Function *rbf_interpolate_2d()* calculates values for elements of a target PLIST from those of a source PLIST. A call to this function must follow the name of the source PLIST on which it operates; a dot must separate the PLIST name from the function name. Source PLIST element selection can be undertaken using a selection equation supplied as an argument of the *rbf_interpolate_2d()* function.

Arguments and subarguments

<i>select</i>	The source PLIST selection equation. This argument is optional. Note that a target PLIST selection equation can also be supplied if desired. The latter must be placed adjacent to the name of the target PLIST on the left side of the assignment operator; see the examples below.
<i>rbf</i>	The name of the radial basis function type used for spatial interpolation. The value supplied for this argument must be one of “ga”, “iq”, “imq”, “mq”, “lin”, “cub” or “tps”. See the discussion below.
<i>rbf; epsilon</i>	The value of the “epsilon” parameter employed by some radial basis function types. Choice of a suitable value for the <i>epsilon</i> subargument

	of the <i>rbf</i> argument may be critical to the success of radial basis function interpolation. The “epsilon” and “epsminsepfac” subarguments are mutually exclusive.
<i>rbf; epsminsepfac</i>	The factor by which the distance to the closest neighbouring source PLIST member is multiplied in determining the reciprocal of the value of the “epsilon” parameter employed by some radial basis function types. The “epsilon” and “epsminsepfac” subarguments are mutually exclusive.
<i>constant_term</i>	The value for this optional argument must be supplied as “yes” or “no”. This determines whether the radial basis function is augmented with a single, estimated, domain-wide additive term. If this argument is omitted, its value is assumed to be “no”.
<i>linear_term</i>	The value for this optional argument must be supplied as “yes” or “no”. This determines whether the radial basis function is augmented with linear-with-distance additive terms applied in orthogonal directions. If this argument is omitted, it is assumed to be “no”.
<i>anis_bearing</i>	The bearing (with respect to north with clockwise positive) of the direction of principle anisotropy, normally the direction of greatest continuity of the variable being interpolated. If this argument is omitted, isotropy is assumed.
<i>anis_ratio</i>	If supplied, the value of this argument is normally greater than 1. This is the ratio of system property elongation in the <i>anis_bearing</i> direction to that in the orthogonal direction. If omitted, a value of 1 is assumed, this comprising specification of system property isotropy.
<i>transform</i>	This must be supplied as “log” or “none”. If supplied as “log” then spatial interpolation is applied to the logs of source PLIST element values rather than to their native values. Target PLIST native element values are then back-calculated from their logs before storage.
<i>lower_limit</i>	If an interpolated value for any element of the target PLIST is less than the value supplied for this argument, it is assigned a value equal to this <i>lower_limit</i> . If this argument is omitted, the lower interpolation limit is assumed to be a very large negative number.
<i>upper_limit</i>	If an interpolated value for any element of the target PLIST is greater than the value supplied for this argument, it is assigned a value equal to this <i>upper_limit</i> . If this argument is omitted, the upper interpolation limit is assumed to be a very large positive number.
<i>report_file</i>	The name of a file to which details of the radial basis function coefficient solution process are recorded. If omitted, no such report is made.

report_file; position The value of the *position* subargument should be “a” or “append” on the one hand, or “r” or “rewind” on the other hand. If it is supplied as “append” the report is added to the end of an existing file. If this subargument is omitted, a default value of “rewind” is assumed.

Discussion

Radial basis functions

Suppose that the value of a system property is sampled at n points p_i . In interpolating between these sample points in order to approximate unknown inter-point property values, a continuous function g is employed. The function g is, in fact, a sum of n individual functions. Each of these individual functions possesses radial symmetry about one of the sample points.

Suppose that the radial function centred on point p_i is denoted as ϕ_i . Then:

$$g(x, y) = \sum_{i=1}^n c_i \phi_i(x, y) \quad (1)$$

where c_i is a multiplier (i.e. coefficient) applied to the function ϕ_i . Using vector notation let \mathbf{c} denote the vector of coefficients c_i appearing in equation (1). These can be determined through insisting that g replicate sampled values at the sample points p_i . Let these sampled values be stored in the vector \mathbf{f} . Let $\phi_{i,j}$ represent the value of ϕ_i at sample point j . Then the coefficients c_i are determined through solution of the symmetric matrix equation:

$$\mathbf{A}\mathbf{c} = \mathbf{f} \quad (2a)$$

so that:

$$\mathbf{c} = \mathbf{A}^{-1}\mathbf{f} \quad (2b)$$

where the matrix \mathbf{A} is given by:

$$\mathbf{A} = \begin{bmatrix} \phi_{1,1} & \phi_{1,2} & \phi_{1,3} & \cdot & \phi_{1,n} \\ \phi_{2,1} & \phi_{2,2} & \phi_{2,3} & \cdot & \phi_{2,n} \\ \phi_{3,1} & \phi_{3,2} & \phi_{3,3} & \cdot & \phi_{3,n} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \phi_{n,1} & \phi_{n,2} & \phi_{n,3} & \cdot & \phi_{n,n} \end{bmatrix} \quad (3)$$

Optionally g of equation (1) can be augmented by a constant term and/or a linear drift term, so that it becomes:

$$g(x, y) = \sum_{i=1}^n c_i \phi_i(x, y) + c_{n+1} + c_{n+2}x + c_{n+3}y \quad (4)$$

where three extra coefficients c_{n+1} , c_{n+2} and c_{n+3} must now be determined. If these are added to the end of the vector \mathbf{c} the matrix \mathbf{A} of equation (2) becomes:

$$\mathbf{A} = \begin{bmatrix} \phi_{1,1} & \phi_{1,2} & \phi_{1,3} & \cdot & \phi_{1,5} & 1 & x_1 & y_1 \\ \phi_{2,1} & \phi_{2,2} & \phi_{2,3} & \cdot & \phi_{2,5} & 1 & x_2 & y_2 \\ \phi_{3,1} & \phi_{3,2} & \phi_{3,3} & \cdot & \phi_{3,5} & 1 & x_3 & y_3 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \phi_{n,1} & \phi_{n,2} & \phi_{n,3} & \cdot & \phi_{n,5} & 1 & x_n & y_n \\ 1 & 1 & 1 & \cdot & 1 & 0 & 0 & 0 \\ x_1 & x_2 & x_3 & \cdot & x_n & 0 & 0 & 0 \\ y_1 & y_2 & y_3 & \cdot & y_n & 0 & 0 & 0 \end{bmatrix} \quad (5)$$

while the three extra elements of \mathbf{f} are zero.

As presently programmed, PLPROC allows use of any of the following radial basis functions. This choice may expand over time. In the following table r denotes distance from the central point of each function. That is:

$$r = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

where (x_i, y_i) denotes the centre point of the function (this being a sample point) while (x_j, y_j) denotes a field point.

Function	Equation
Gaussian (GA)	$e^{-(\epsilon r)^2}$
Inverse quadratic (IQ)	$\frac{1}{1 + (\epsilon r)^2}$
Inverse multiquadratic (IMQ)	$\frac{1}{\sqrt{1 + (\epsilon r)^2}}$
Multiquadratic (MQ)	$\sqrt{1 + (\epsilon r)^2}$
Linear (LIN)	r
Cubic (CUB)	r^3
Thin plate spline (TPS)	$r^2 \log(r)$

Radial basis functions presently supported by PLPROC.

Optionally, the user may designate that anisotropy exists within the model domain, this implying greater continuity of the sampled system property in one direction than in the direction at right angles to it. In this case the same radial basis functions as tabulated above are employed for spatial interpolation. However the coordinate system is stretched in the direction of greatest continuity. The coordinates of all sample points, and points to which interpolation must take place, are first expressed in a rotated coordinate system in which the

x -axis is aligned with the direction of principle anisotropy. Then, after “stretching” in this direction is applied, the above formulae are employed.

If the log transformation option is chosen, radial basis function interpolation is applied to the logs of source PLIST element values rather than to their native values. Interpolated log-transformed values are back-transformed to native values before being stored in the target PLIST.

Choices and consequences

When using radial basis function interpolation in a particular modelling context (for example as a basis for interpolation to a model grid from a set of pilot points) the user is faced with a number of choices. These include the following:

- number of pilot points to use;
- direction and magnitude of anisotropy;
- the type of radial basis function to employ;
- the value of ε for those functions which use it;
- whether to augment the radial basis function with a constant and drift terms as in equation (4);
- whether to interpolate log or native values.

Before allowing PEST to estimate pilot point parameters in calibrating a particular model, the user is urged to do a little experimentation first in order to assure him/herself that node-based parameter fields are reasonable when interpolation takes place according to his/her chosen settings. A few aspects of the use of radial basis functions which require consideration are now discussed.

The solution to the problem described by equations (2a) and (2b) must be stable. Hence the condition number of the \mathbf{A} matrix must not be too high. The reader will recall that the higher is the condition number of a matrix, the closer does that matrix approach singularity (i.e. noninvertibility). Furthermore, numerical noise that is associated with the matrix and with sampled values is amplified in proportion to the condition number of \mathbf{A} as the matrix equation is solved. PLPROC’s internal use of double precision arithmetic alleviates this to some degree. Nevertheless, well-posedness of the problem of calculating \mathbf{c} from \mathbf{f} using equation (2b) must still be maintained.

The literature on radial basis functions reports that the condition number of \mathbf{A} can rise as the number of radial basis functions (i.e. the number of points within the source PLIST) increases. It can also rise if ε (required by some equations) is assigned a very high or very low value. The *rbf_interpolate_2d()* function assists the user in choosing appropriate settings by providing him/her with the option of receiving a report on some numerical aspects of the radial basis function coefficient solution process; see the *report_file* argument. The report that is recorded in the report file includes the condition number of the \mathbf{A} matrix. While it is difficult to provide a value that should never be exceeded for this number, the user is advised that values over 10^{10} are definitely worrisome; ideally values should be lower than 10000.

The inverse of the variable ε featured in some of the radial basis function equations plays a similar role to that of the variogram range where interpolation is implemented using kriging. Lower values of ε lead to broader, smoother basis functions. When calibrating a groundwater model this is often a desirable feature of the calibrated parameter field. However the literature warns against making ε too small as it may lead to a rise in the condition number of \mathbf{A} . As

stated above, the same can also occur if it is too large. Nevertheless a moderate ε may sometimes be required where local, small-scale heterogeneity of hydraulic properties is anticipated and spatial interpolation between pilot points should therefore be more local. A good starting point for ε is the inverse of the average inter-pilot-point distance.

Other choices required in implementing radial basis function interpolation may be problem-specific. If sample points are not distributed out to the edges of a model domain, then it may be a good idea to employ functions which diminish rather than grow with distance in order to prevent the interpolation process from providing disagreeably high or low interpolated values near the model domain boundaries. Under these circumstances the use of a linear drift is also questionable.

If system property values represented by PLIST element values can vary over orders of magnitude, and can never become negative, then log interpolation becomes almost mandatory. Conductivity/permeability properties fall into this category. Log transformation of other properties such as storage coefficient and porosity should also be considered.

Epsilon and epsminsepfac

As stated above, a good starting point for ε may be the inverse of the average distance between pilot points from which interpolation is taking place. However where the separation between pilot points varies greatly throughout the interpolation domain, then adoption of such a strategy may prove problematical, as the same ε value may then not be appropriate for all pilot-point-specific radial basis function instances. The *rbf_interpolate_2d()* function actually allows ε to vary from pilot point to pilot point (for those radial basis function types which employ ε). It can, in fact, be assigned a value equal to the reciprocal of a factor times the distance between any pilot point and its nearest neighbour. This factor is supplied as the *epsminsepfac* subargument of the *rbf* argument (*epsminsepfac* stands for “epsilon as a factor of minimum separation”). Try a value of between 0.5 and 2.0 (start with 0.8). Note that either *epsilon* or *epsminsepfac* can be specified, but not both.

Examples

Example 1

```
sm1=s1.rbf_interpolate_2d(transform='log',          &
                           rbf=mq; epsilon=5e-5,    &
                           report_file='report.dat')
```

In this example, radial basis function interpolation takes place from all members of the *s1* PLIST to all members of the *sm1* PLIST. The multiquadratic function is employed with an ε value of 5e-5. Interpolation acts on the logs of PLIST values. A report, which includes the condition number of the **A** matrix of equation 2, as well as the coefficients of the individual basis functions, is written to file *report.dat*.

Example 2

```
sm1=s1.rbf_interpolate_2d(transform=none,          &
                           anis_ratio=3, anis_bearing=60, &
                           rbf=iq; epsilon=5e-5,      &
                           constant_term='yes', linear_term='yes', &
                           upper_limit=5, lower_limit=1, &
                           report_file='report.dat')
```

In this example radial basis function interpolation is based on the inverse quadratic function, and is applied to native PLIST values rather than to their logs. The axis of principle anisotropy is oriented at a bearing of 60 degrees with respect to north; continuity of system properties in this direction is assumed to be 3 times greater than in the direction orthogonal to it. In carrying out spatial interpolation the inverse quadratic radial basis function is augmented by a constant term and a linear drift so that spatial interpolation is based on equation (4) rather than on equation (1).

Example 3

```
sm1(select=(model_zone==3))=s1.rbf_interpolate_2d(           &
    select=((pp_zone==2)|| (pp_zone==3)),                   &
    transform='log',                                         &
    rbf=mq; epsilon=5e-5,                                    &
    report_file='report.dat')
```

This is the same as example 1 except that selection equations are now employed to select a subset of the *sm1* target PLIST to which interpolation takes place, and a subset of the *s1* source PLIST from which values are interpolated. In the former case, interpolated values are assigned only to those elements of *sm1* for which values of the *model_zone* SLIST are 3. (As is discussed in Chapter 3 of this manual, the *sm1* PLIST and the *model_zone* SLIST must share the same reference CLIST.) In the latter case interpolation takes place only from those elements of the *s1* PLIST for which *pp_zone* SLIST values are 2 or 3.

Elements of the *sm1* PLIST which are not assigned values through radial basis function interpolation because of the presence of a target selection equation retain their original values.

Example 4

```
sm1(select=(model_zone==3))=s1.rbf_interpolate_2d(           &
    select=((pp_zone==2)|| (pp_zone==3)),                   &
    transform='log',                                         &
    rbf=mq; epsminsepfac=0.8,                                &
    report_file='report.dat')
```

This example is identical to the previous example except that ϵ is calculated on a source PLIST element by source PLIST element basis using the *epsminsepfac* subargument of the *rbf* argument. ϵ for each source PLIST element is calculated as the reciprocal of 0.8 times the distance between that element and its nearest neighbour.

rbf_interpolate_3d()

General

Function *rbf_interpolate_3d()* calculates values for the elements of one PLIST from those of another using three-dimensional radial basis functions as a device for performing spatial interpolation. In many cases where this function is employed, the source PLIST will comprise a set of pilot points while the target PLIST will comprise the nodes of a numerical model grid or mesh. The interpolation process can be confined to a subset of either list using source and/or target selection equations.

Because interpolation employs three-dimensional basis functions, the reference CLIST for each of the source and target PLISTs cited in a call to function *rbf_interpolate_3d()* must be three-dimensional.

Function Specifications

Function value

The name of the PLIST to which interpolation takes place (i.e. the target PLIST) comprises the output of function *rbf_interpolate_3d()*. This name must appear before the function name in a PLPROC command which invokes the function; it should be separated from the function name by a “=” symbol. Optionally, a target selection equation can be supplied with the name of the target PLIST. The target PLIST should have been created in previous PLPROC processing as it is not created by the *rbf_interpolate_3d()* function itself. Target PLIST elements which are not assigned values because of the presence of a target selection equation retain their existing values.

Object associations

Function *rbf_interpolate_3d()* calculates values for elements of a target PLIST from those of a source PLIST. A call to this function must follow the name of the source PLIST on which it operates; a dot must separate the source PLIST name from the function name. Source PLIST element selection can be undertaken using a selection equation supplied as an argument of the *rbf_interpolate_3d()* function.

Arguments and subarguments

<i>select</i>	The source PLIST selection equation. This argument is optional. Note that a target PLIST selection equation can also be supplied if desired. The latter must be placed adjacent to the name of the target PLIST on the left side of the assignment operator; see the example below.
<i>rbf</i>	The name of the radial basis function type used for spatial interpolation. The value supplied for this argument must be one of “ga”, “iq”, “imq”, “mq”, “lin”, “cub” or “tps”. See the discussion below.
<i>rbf; epsilon</i>	The value of the “epsilon” parameter employed by some radial basis function types. Choice of a suitable value for the <i>epsilon</i> subargument

	of the <i>rbf</i> argument may be crucial to the success of radial basis function interpolation.
<i>constant_term</i>	The value for this optional argument must be supplied as “yes” or “no”. This determines whether the radial basis function is augmented with a single, estimated, domain-wide additive term. If this argument is omitted, its value is assumed to be “no”.
<i>linear_term</i>	The value for this optional argument must be supplied as “yes” or “no”. This determines whether the radial basis function is augmented with linear-with-distance additive terms applied in three orthogonal directions, these directions coinciding with the axes of the ellipse of anisotropy. If this argument is omitted, it is assumed to be “no”.
<i>transform</i>	This must be supplied as “log” or “none”. If supplied as “log” then spatial interpolation is applied to the logs of source PLIST element values rather than to their native values. Target PLIST native element values are then back-calculated from their spatially interpolated logs before storage.
<i>ang1</i>	Defines the angle between north and the direction of maximum horizontal anisotropy in degrees clockwise.
<i>ang2</i>	Defines the negative of the plunge of the direction of maximum anisotropy, that is the angle (positive upwards) between horizontal (in the direction defined by <i>ang1</i>) and the direction of actual maximum anisotropy.
<i>ang3</i>	To quote Deutsch and Journal (1998) - “ <i>ang3</i> leaves the principal direction defined by <i>ang1</i> and <i>ang2</i> unchanged. The two directions orthogonal to that principal direction are rotated clockwise relative to the principal direction when looking toward the origin.” In the vast majority of cases <i>ang3</i> should be set to zero. This is its default value if this argument is omitted.
<i>hor_anis</i>	The ratio of system property elongation in the direction of <i>ang1</i> to that in the direction of <i>ang2</i> .
<i>vert_anis</i>	The ratio of system property elongation in the direction of <i>ang1</i> to that in the direction of <i>ang3</i> .
<i>lower_limit</i>	If an interpolated value for any element of the target PLIST is less than the value supplied for this argument, it is assigned a value equal to this <i>lower_limit</i> . If this argument is omitted the lower interpolation limit is assumed to be a very large negative number.
<i>upper_limit</i>	If an interpolated value for any element of the target PLIST is greater than the value supplied for this argument, it is assigned a value equal to this <i>upper_limit</i> . If this argument is omitted the upper interpolation

	limit is assumed to be a very large positive number.
<i>report_file</i>	The name of a file to which details of the radial basis function coefficient solution process are recorded. If omitted, no such report is made.
<i>report_file; position</i>	The value of the <i>position</i> subargument should be “a” or “append” on the one hand, or “r” or “rewind” on the other hand. If it is supplied as “append” the report is added to the end of an existing file. If this subargument is omitted, a default value of “rewind” is assumed.

Discussion

See the discussion of radial basis functions in the documentation of function *rbf_interpolate_2d()*. The mathematics and implementation of three-dimensional radial basis functions are identical to those of two-dimensional radial basis functions except that calculation of the distance between two points, and hence of r in the equation for each of the supported radial basis function types, includes the difference of their z coordinates in addition to that of their x and y coordinates. Where there is vertical anisotropy, appropriate z coordinate scaling takes place in formulating this difference.

Because of the likelihood of at least vertical anisotropy, the *hor_anis*, *vert_anis*, *ang1* and *ang2* arguments of the *rbf_interpolate_3d()* function are mandatory. (Recall that, in contrast, the *anis_bearing* and *anis_ratio* specifiers of two-dimensional anisotropy are optional in calls to function *rbf_interpolate_2d()*.) On the other hand, if *ang3* is omitted it is assumed to be zero.

Examples

Example 1

```
pl_mod=pl_d.rbf_interpolate_3d(           &
    transform='log',                       &
    rbf=iq; epsilon=0.05,                 &
    hor_anis=1,vert_anis=10,              &
    ang1=0,ang2=0)                        &
```

In this example, radial basis function interpolation takes place from all members of the *pl_d* PLIST to all members of the *pl_mod* PLIST. The inverse quadratic function is employed with an ϵ value of 0.05. Interpolation acts on the logs of PLIST values. A vertical anisotropy of 10 is employed in the interpolation process; that is, the ratio of elongation in the horizontal direction to that in the vertical direction is 10.

Example 2

```
pl_mod=pl_d.rbf_interpolate_3d(           &
    transform='log',                       &
    rbf=iq; epsilon=0.05,                 &
    constant_term='yes',linear_term='yes' &
    hor_anis=1,vert_anis=10,              &
    ang1=0,ang2=0,                        &
    report_file='report.dat';position='a') &
```

This is identical to example 1 except that the radial function support base is now augmented with a constant and linear terms. This requires calculation of four extra interpolation parameters, namely the constant term, and the coefficient of distance in each of the three principle directions of anisotropy. Additionally, a report on the radial basis function solution process is appended to the end of file *report.dat*.

Example 3

```
pm_1(select=(sm_1==1)) = pp_1.rbf_interpolate_3d(           &
    select=sp_1==1,                                         &
    rbf='iq'; epsilon=5e-5,                                 &
    hor_anis=3,vert_anis=10,                                &
    ang1=60,ang2=20,ang3=0.0,                               &
    transform='none',                                       &
    upper_limit=10.0,lower_limit=1.0,                       &
    report_file='report.dat')
```

This example demonstrates the use of source and target PLIST selection equations. Interpolation takes place only from elements of the *pp_1* source PLIST for which members of the *sp_1* SLIST or PLIST are equal to 1. It takes place only to elements of the *pm_1* target PLIST for which elements of the *sm_1* SLIST or PLIST are equal to 1. Naturally *pp_1* and *sp_1* should share the same reference CLIST; similarly *pm_1* and *sm_1* should share the same reference CLIST.

If any interpolated value is less than 1.0, it is assigned a value of 1.0; while if any interpolated value is greater than 10.0 it is assigned a value of 10.0.

rbf_sda_interpolate_2d()

General

“SDA” stands for “spatially dependent anisotropy”.

Function `rbf_sda_interpolate_2d()` undertakes spatial interpolation from one PLIST (normally representing a set of pilot points) to another PLIST (normally representing the cell centres or nodes of a model grid or mesh). However three interpolation exercises are actually carried out. First the ratio and bearing of anisotropy are interpolated from two PLISTs which are assumed to have a common parent CLIST. This is done using radial basis functions. Next interpolation is carried out from the actual source PLIST to the actual target PLIST. This interpolation is also carried out using radial basis functions. However in implementing this final interpolation stage distance calculation between target and source PLIST elements is modified in accordance with the spatially varying ratio and direction of anisotropy as spatially interpolated from the anisotropy ratio and bearing PLISTs.

Function Specifications

Function value

The `rbf_sda_interpolate_2d()` function calculates values for some or all elements of an existing PLIST (referred to as the “target PLIST”). The name of the target PLIST comprises the output of the function. This name must appear before the function in a PLPROC command which uses the function. This name should be separated from the function by a “=” symbol.

Object associations

Function `rbf_sda_interpolate_2d()` employs source PLIST element values in calculating values for elements of a target PLIST. The name of this function must follow the name of the source PLIST on which it operates; a dot must separate these two names.

Arguments and subarguments

<i>select</i>	The source PLIST selection equation. This argument is optional. Note that a target PLIST selection equation can also be supplied if desired. The latter must be placed adjacent to the name of the target PLIST on the left side of the assignment operator; see the examples below. This argument is optional.
<i>rbf</i>	The name of the radial basis function type used for spatial interpolation. The value supplied for this argument must be one of “ga”, “iq”, “imq”, “mq”, “lin”, “cub” or “tps”. See the discussion below. This argument is mandatory.
<i>rbf; epsilon</i>	The value of the “epsilon” parameter employed by some radial basis function types. Choice of a suitable value for the epsilon subargument of the <i>rbf</i> argument may be critical to the success of

	radial basis function interpolation.
<i>constant_term</i>	The value for this optional argument must be supplied as “yes” or “no”. This determines whether the radial basis function is augmented with a single, estimated, domain-wide additive term. If this argument is omitted, its value is assumed to be “no”.
<i>transform</i>	This must be supplied as “log” or “none”. If supplied as “log” then spatial interpolation is applied to the logs of source PLIST element values rather than to their native values. Target PLIST native element values are then back-calculated from their logs before storage. This argument is mandatory.
<i>lower_limit</i>	If an interpolated value for any element of the target PLIST is less than the value supplied for this argument, it is assigned a value equal to this <i>lower_limit</i> . If this argument is omitted, the lower interpolation limit is assumed to be a very large negative number.
<i>upper_limit</i>	If an interpolated value for any element of the target PLIST is greater than the value supplied for this argument, it is assigned a value equal to this <i>upper_limit</i> . If this argument is omitted, the upper interpolation limit is assumed to be a very large positive number.
<i>report_rcond</i>	“yes” or “no”. If set to “yes” the condition number is reported to the screen when radial basis function coefficients are determined for use in both spatial interpolation of anisotropy ratio and bearing data, and in source-to-target PLIST interpolation. If this argument is omitted, it is assumed to be “no”.
<i>sda_bearing_plist</i>	The name of the PLIST containing anisotropy bearings. This is a mandatory argument.
<i>sda_ratio_plist</i>	The name of the PLIST containing anisotropy ratios. This is a mandatory argument.
<i>sda_rbf</i>	The name of the radial basis function type used for spatial interpolation of anisotropy data. The value supplied for this argument must be one of “ga”, “iq”, “imq”, “mq”, “lin”, “cub” or “tps”. If this argument is omitted the same radial basis function type is used for anisotropy interpolation as is used for source-to-target PLIST interpolation.
<i>sda_rbf; epsilon</i>	The value of the “epsilon” parameter employed for anisotropy ratio and bearing interpolation using radial basis functions. This subargument of the <i>sda_rbf</i> argument is mandatory (if the <i>sda_rbf</i> argument is supplied).
<i>sda_anis_bearing</i>	A real number, this being the anisotropy bearing used in interpolation of anisotropy bearing and ratio data. This argument

	must be supplied if a value is supplied for the <i>sda_anis_ratio</i> argument. If omitted it is assumed to be zero. It must lie between -360 and 360 degrees.
<i>sda_anis_ratio</i>	A real number, this being the anisotropy ratio used in interpolation of anisotropy bearing and ratio data. This argument must be supplied if a value is supplied for the <i>sda_anis_bearing</i> argument. If omitted it is assumed to be 1.0. If supplied, it must be greater than zero.
<i>sda_constant_term</i>	“yes” or “no”. This determines whether a constant term is included in radial basis function interpolation of anisotropy data. If omitted it is assumed to be “no”.
<i>sda_delta</i>	The line joining any target PLIST element and any source PLIST element is divided into increments of this length in order to integrate along the line in order to determine the “geostatistical distance” between those points. This argument is mandatory. It must be a real number greater than zero.
<i>sda_target_ratio_plist</i>	If this argument is supplied, elements of the nominated PLIST will be assigned rbf-interpolated values of the spatially varying anisotropy ratio. If the PLIST already exists, it must have the same parent CLIST as that of the interpolation target PLIST. If it does not exist, it is created as such. Dummy values are provided for elements which correspond to target PLIST elements to which interpolation does not take place because of non-selection through a target element selection equation. This argument is optional.
<i>sda_target_bearing_plist</i>	If this argument is supplied, elements of the nominated PLIST will be assigned rbf-interpolated values of the spatially varying anisotropy bearing. If the PLIST already exists, it must have the same parent CLIST as that of the interpolation target PLIST. If it does not exist, it is created as such. Dummy values are provided for elements which correspond to target PLIST elements to which interpolation does not take place because of non-selection through a target element selection equation. This argument is optional.
<i>sda_bearing_upper_limit</i>	The upper limit of interpolated anisotropy bearing. This argument is mandatory. The difference between <i>sda_bearing_upper_limit</i> and <i>sda_bearing_lower_limit</i> must be less than 180 degrees.
<i>sda_bearing_lower_limit</i>	The lower limit of interpolated anisotropy bearing. This argument is mandatory. The difference between <i>sda_bearing_upper_limit</i> and <i>sda_bearing_lower_limit</i> must be less than 180 degrees.
<i>sda_ratio_upper_limit</i>	The upper limit of interpolated anisotropy ratio. This argument is mandatory. It must be a real number greater than zero.

sda_ratio_lower_limit The lower limit of interpolated anisotropy ratio. This argument is mandatory. It must be a real number greater than zero.

Discussion

General

Function *rbf_sda_interpolate_2d()* has much in common with function *ivd_sda_interpolate_2d()*. The difference between the two is that interpolation from a source to target PLIST through a domain characterized by spatially variable anisotropy is undertaken using distance-modified radial basis functions rather than a distance-modified inverse-power-of-distance methodology when the *rbf_sda_interpolate_2d()* function is invoked. Unfortunately, use of distance modified radial basis functions is slower than use of the distance-modified inverse-power-of-distance methodology. Hence where target and/or source PLISTs are large, function *ivd_sda_interpolate_2d()* may be useable while function *rbf_sda_interpolate_2d()* may not be useable in undertaking spatially variable anisotropy interpolation. For both of these functions, however, interpolation of anisotropy ratio and bearing prior to source-to-target interpolation is implemented using radial basis functions.

As discussed above, use of function *rbf_sda_interpolate_2d()* actually implies three spatial interpolation exercises. In the first of these exercises the anisotropy ratio is notionally interpolated from a user-provided set of PLIST elements to all points throughout a model domain. (Actually a set of radial basis function coefficients is determined so that such interpolation can take place on an as-needed basis when calculating geostatistical distances between source and target PLIST elements in later radial basis function interpolation). In the second of these exercises the same is done for anisotropy bearing. Finally interpolation between source and target PLIST elements is undertaken through the spatially varying anisotropy field determined by the first two interpolation exercises using the radial basis function methodology once again. However the type of radial basis function employed in this final step, and the epsilon value provided for its deployment, do not need to be the same as those employed in the preceding anisotropy interpolation.

Anisotropy interpolation

PLISTs containing anisotropy ratio and bearing data must be supplied to function *rbf_sda_interpolate_2d()* through its *sda_ratio_plist* and *sda_bearing_plist* arguments. These PLISTs must have a common parent CLIST (and hence pertain to the same set of pilot points). No selection equation can operate on these PLISTs; hence all elements of these PLISTs are employed in determination of a spatially varying anisotropy field.

Upper and lower bounds for interpolated anisotropy ratios and bearings must also be provided through the *sda_ratio_upper_limit*, *sda_ratio_lower_limit*, *sda_bearing_upper_limit* and *sda_bearing_lower_limit* arguments. Each of these must be provided as a single real number. Anisotropy ratio limits must be positive. The difference between upper and lower anisotropy bearing limits must be less than 180 degrees. Recall from other sections of this manual that the anisotropy bearing is normally the direction of maximum continuity of hydraulic properties; this occurs if the anisotropy ratio is greater than 1.0. If this is the case, then hydraulic property continuity in the direction perpendicular to this bearing is correspondingly smaller than in the direction parallel to this bearing. This condition is achieved by notionally “stretching” distance in the direction perpendicular to the

anisotropy bearing so that points which are aligned in this direction are further apart (and hence less “connected” when undertaking spatial interpolation).

Spatial interpolation of anisotropy ratio and bearing data is undertaken using radial basis functions. Anisotropy ratio is actually log-interpolated; however this is done behind the scenes. The use of radial basis functions for this stage of interpolation is numerically efficient, for its implementation requires only that a set of coefficients be first computed. These are then applied on an as-needed basis later in the overall interpolation process as “geostatistical distances” between source and target PLIST elements are determined through integration along lines which separate these elements.

Use of radial basis functions as an interpolation device demands that a radial basis function type and corresponding epsilon (for those radial basis functions which require it) be supplied. These are provided through the *sda_rbf* argument and the *epsilon* subargument of the *sda_rbf* argument respectively. Optionally, radial basis function interpolation of anisotropy data can itself incorporate (spatially uniform) anisotropy. If this is required, the uniform anisotropy ratio and bearing employed for radial basis function interpolation of PLIST-supplied anisotropy ratio and bearing data can be provided through the *sda_anis_ratio* and *sda_anis_bearing* function arguments.

Optionally, radial basis function interpolation can include an accompanying constant function, this being activated through assigning the *sda_constant_term* argument a value of “yes” (for activation) or “no” (for omission). Absence of this argument signifies a default value of “no”. Linear terms are not allowed in the spatial interpolation process however.

Once coefficients have been determined for the radial basis functions, as well as for the optional constant term, interpolation can then take place to any point. If a user wishes to have interpolation of spatially-dependent anisotropy ratio and/or bearing carried out to all elements of a special target PLIST, this can be achieved through supplying values for the optional *sda_target_ratio_plist* and/or *sda_target_bearing_plist* arguments. The name of a PLIST must be supplied in both cases. If the PLIST already exists, it must have the same parent CLIST as the interpolation target PLIST, this being the PLIST whose name appears on the left of the “=” symbol in the call to function *rbf_sda_interpolate_2d()*. If the PLIST does not exist it will be created (with the target CLIST as its parent).

Interpolation from source to target PLISTs

Interpolation from the source PLIST to the target PLIST also takes place using the radial basis function methodology. The radial basis function for this stage of interpolation is selected using the *rbf* function argument, whereas the epsilon value required by this function is denoted using the *epsilon* subargument of the *rbf* argument (see the description of the *rbf_interpolate_2d()* function for an explanation of this important variable). An optional constant term (whose coefficient is determined along with the coefficient of radial basis functions) can be employed in the interpolation process through provision of a *constant_term* argument value of “yes”. Note that the use of linear terms is not allowed.

In implementing the radial basis function methodology in a domain of spatially varying anisotropy, distances perpendicular to the direction of local anisotropy bearing are modified in accordance with the local value of anisotropy ratio. Distance expansion takes place if the anisotropy ratio is positive whereas distance contraction takes place if it is negative. Because anisotropy is spatially-dependent, the total “geostatistical distance” between any two points is determined by summing (i.e. integrating) incremental geostatistical distances along the line.

Subroutine *rbf_sda_interpolate_2d()* performs this integration numerically using the rectangle rule applied to increments of length *sda_delta* (supplied by the user). The smaller is *sda_delta* the more work is required in carrying out integration along a line to determine the geostatistical distance between its end points. Hence the value supplied for this argument can have a considerable impact on the speed of the interpolation process.

The modified geostatistical distance is used in determining the distance of each target PLIST element from the focal point of each radial basis function (these being the locations of source PLIST elements). As all radial basis functions are employed in interpolating to all target PLIST elements, the overall time required for radial basis function interpolation under conditions of spatially varying anisotropy can be considerable. (Where this is interpolation is carried out using the inverse-power-of-distance methodology using subroutine *ivd_sda_interpolate_2d()* the number of source points used for interpolation to any target point can be limited to a relatively small number, this resulting in considerable computational savings.)

Optionally, spatial interpolation can be applied to the logs of source PLIST elements. Interpolated values are back-transformed to native values before being assigned to target PLIST elements. Limits can be placed on interpolated values through use of the *upper_limit* and *lower_limit* function arguments.

The following should also be noted.

- The source PLIST can have a different parent CLIST from that of the PLISTs used to express anisotropy ratio and bearing. So too, of course, can the target PLIST.
- If desired, a selection equation can be supplied with the target PLIST; interpolation will then take place only to elements of that PLIST that are identified through that equation.
- A selection equation can also be applied to the source PLIST. However it cannot be supplied for the anisotropy ratio and bearing PLISTs.
- The target PLIST must already have been defined through previous PLPROC functions.

Examples

Example 1

```
grid_hk=pp_hk.rbf_sda_interpolate_2d(                                &
    upper_limit=1e20,                                              &
    lower_limit=1e-20,                                             &
    transform='log',                                              &
    rbf=ga; epsilon=0.005,                                         &
    constant_term='yes',                                          &
    sda_bearing_plist=pp_b_anis,                                   &
    sda_ratio_plist=pp_r_anis,                                     &
    sda_delta=10,                                                 &
    sda_bearing_upper_limit=180,                                   &
    sda_bearing_lower_limit= 90,                                   &
    sda_ratio_upper_limit=100,                                     &
    sda_ratio_lower_limit=.1,                                     &
    sda_rbf=mq;epsilon=0.01)
```

Using this command spatial interpolation is undertaken from the *pp_hk* PLIST to the *grid_hk* PLIST using a Gaussian radial basis function with an epsilon value of 0.005; a constant term

is also employed in the interpolation process. In evaluating the geostatistical distance between source and target PLIST elements the line increment is 10. Anisotropy ratio and bearing data are housed in the *pp_r_anis* and *pp_b_anis* PLISTs respectively. Radial basis function interpolation takes place from the elements of these PLISTs using the multiquadratic basis function with an epsilon value of 0.01 (see the description of the *rbf_interpolate_2d()* function for an explanation of this variable). In undertaking anisotropy interpolation, the anisotropy ratio is limited at its upper end by a value of 100 and at its lower end by a value of 0.1; the anisotropy bearing must never be greater than 180 degrees or less than 90 degrees.

Example 2

```
grid_hk=pp_hk.rbf_sda_interpolate_2d(                                &
    upper_limit=1e20,                                              &
    lower_limit=1e-20,                                             &
    transform='log',                                              &
    rbf=ga; epsilon=0.005,                                         &
    constant_term='yes',                                          &
    report_rcond='yes',                                           &
    sda_bearing_plist=pp_b_anis,                                   &
    sda_ratio_plist=pp_r_anis,                                     &
    sda_delta=10,                                                 &
    sda_bearing_upper_limit=180,                                   &
    sda_bearing_lower_limit= 90,                                   &
    sda_ratio_upper_limit=100,                                     &
    sda_ratio_lower_limit=.1,                                     &
    sda_rbf=mq;epsilon=0.01,                                       &
    sda_target_bearing_plist=grid_b_anis,                         &
    sda_target_ratio_plist=grid_r_anis)
```

Implementation of this function has the same outcomes as implementation of the function of example 1. However the condition numbers of the matrices that must be inverted in determination of radial basis function coefficients for anisotropy interpolation on the one hand and source-to-target PLIST interpolation on the other hand, are reported to the screen. (This can be important information – see the description of the *rbf_interpolate_2d()* function.) As well as this, spatially interpolated anisotropy ratios and bearings are reported to the *grid_r_anis* and *grid_b_anis* PLISTs respectively.

Example 3

```
grid_hk(select=(model_zone==3))=                                  &
    pp_hk.rbf_sda_interpolate_2d(                                &
    upper_limit=1e20,                                              &
    lower_limit=1e-20,                                             &
    transform='log',                                              &
    rbf=ga; epsilon=0.005,                                         &
    constant_term='yes',                                          &
    report_rcond='yes',                                           &
    sda_bearing_plist=pp_b_anis,                                   &
    sda_ratio_plist=pp_r_anis,                                     &
    sda_delta=10,                                                 &
    sda_bearing_upper_limit=180,                                   &
    sda_bearing_lower_limit= 90,                                   &
    sda_ratio_upper_limit=100,                                     &
    sda_ratio_lower_limit=.1,                                     &
    sda_rbf=mq;epsilon=0.01,                                       &
    sda_target_bearing_plist=grid_b_anis,                         &
    sda_target_ratio_plist=grid_r_anis,                           &
```

```

sda_anis_ratio=2.0,
sda_anis_bearing=30.0)
&

```

This example is similar to the above example. However a selection equation is employed with the target PLIST so that interpolation takes place only to those of its elements which lie in model zone 3. At the same time, radial basis function interpolation of anisotropy ratio and bearing is itself based on a uniform anisotropy ratio of 2.0 with a bearing of 30 degrees.

Example 4

```

grid_hk=pp_hk.rbf_sda_interpolate_2d(
    select=((pp_zone==2) || (pp_zone==3)),
    upper_limit=1e20,
    lower_limit=1e-20,
    transform='log',
    rbf=ga; epsilon=0.005,
    constant_term='yes',
    sda_bearing_plist=pp_b_anis,
    sda_ratio_plist=pp_r_anis,
    sda_delta=10,
    sda_bearing_upper_limit=180,
    sda_bearing_lower_limit= 90,
    sda_ratio_upper_limit=100,
    sda_ratio_lower_limit=.1,
    sda_rbf=mq;epsilon=0.01,
    sda_constant_term='yes')
&

```

This example is identical to example 1 except for the fact that a target source selection equation is employed. Also a constant term is used as part of the anisotropy radial basis function interpolation process. (The coefficient of this term is determined as part of the coefficient solution process.)

rbf_using_file()

General

The *rbf_using_file()* function employs CLIST-to-CLIST interpolation information computed by function *calc_rbf_factors_2d()* to undertake spatial interpolation from the elements of a source PLIST to those of a target PLIST. The reference CLIST for the target PLIST must have been the target CLIST used in the prior call to the *calc_rbf_factors_2d()* function. The reference CLIST for the source PLIST must have been the source CLIST employed in the same prior call to the *calc_rbf_factors_2d()* function.

Where elements of the target PLIST to which interpolation takes place have been restricted through use of a target selection equation in the prior call to the *calc_rbf_factors_2d()* function, the excluded values of the target PLIST are left unaltered by the *rbf_using_file()* function.

Function Specifications

Function value

The *rbf_using_file()* function calculates values for some or all elements of an existing PLIST (referred to as the “target PLIST”). The name of the target PLIST comprises the output of the function. This name must appear before the function in a PLPROC command which uses the function. This name should be separated from that of the function by a “=” symbol.

Object associations

Function *rbf_using_file()* employs source PLIST element values in calculating values for elements of a target PLIST. A call to this function must follow the name of the source PLIST on which it operates; a dot must separate the name of the function from the name of the source PLIST.

Arguments and subarguments

<i>file</i>	The name of the file in which pre-calculated radial basis function interpolation information is recorded. This file will have been written by the <i>calc_rbf_factors_2d()</i> function.
<i>file; format</i>	<p>The <i>format</i> subargument of the <i>file</i> argument must be supplied as “formatted” or “binary”. In the former case an ASCII file is expected, whereas binary storage is expected in the latter case. (The user can employ “text” or “ascii” instead of “formatted” if he/she wishes, and “unformatted” instead of “binary” if he/she so desires.)</p> <p>If this subargument is omitted, the file is assumed to be formatted.</p>
<i>transform</i>	This must be supplied as “log” or “none”. If supplied as “log” then interpolation is applied to the logs of source PLIST elements to calculate the logs of target PLIST elements; the latter are then back-transformed to native element values before being stored in the target

PLIST.

<i>lower_limit</i>	If the calculated value for any element of the target PLIST is less than the value supplied for this argument, it is assigned this <i>lower_limit</i> value instead. If this argument is omitted, the lower interpolation limit is assumed to be a very large negative number.
<i>upper_limit</i>	If the calculated value for any element of the target PLIST is greater than the value supplied for this argument, it is assigned this <i>upper_limit</i> value instead. If this argument is omitted, the upper interpolation limit is assumed to be a very large positive number.

Discussion

General

The use of radial basis functions as a means of two-dimensional interpolation is explained in documentation of function *rbf_interpolate_2d()*. The benefits that can sometimes be accrued through undertaking radial basis function interpolation in two separate steps is explained in documentation of function *calc_rbf_factors_2d()*. Realization of these benefits normally requires that interpolation be undertaken from many source PLISTs with a single reference CLIST to many target PLISTs which also employ a single reference CLIST. Alternatively they can be realized when interpolation from the same source PLIST to the same target PLIST is undertaken many times on successive model runs, as may happen during a PEST calibration process wherein a model batch file comprising both the simulator and its pre/post-processors is run repetitively by PEST.

Where PLISTs are large, the computations required for radial basis function interpolation can take a long time. However where the *calc_rbf_factors_2d()* function is employed prior to undertaking the actual PLIST-to-PLIST interpolation, the time required for the latter can be significantly reduced. Because the outcomes of *calc_rbf_factors_2d()* pre-interpolation calculations are stored in a file, they need only be calculated once. This information can then be used, and re-used, for interpolation of the same or different PLISTS, on the same or subsequent PLPROC runs.

In undertaking spatial interpolation using the *rbf_using_file()* function, PLPROC always checks that the reference CLIST for the source PLIST specified in this function call is the CLIST on which basis calculations were made during the previous call to function *calc_rbf_factors_2d()* which was used to write the radial basis function pre-interpolation information file. The same applies to the target PLIST. (The names of the target and source CLISTS used in calculation of radial basis function pre-interpolation information are stored in this file.)

From the above considerations it follows that the PLPROC script file that implements spatial interpolation using one or more *rbf_using_file()* functions may be different from that employed to undertake the first part of the two-part radial basis function interpolation process, as the former may omit *calc_rbf_factors_2d()* function calls. However the two different PLPROC scripts cannot be VERY different, in that they must reference the same CLISTS and PLISTS, which must have the same properties in both of these scripts. The script which employs the *rbf_using_file()* function(s) is most easily built from that which calculates first-stage radial basis function information (and probably also employs the *rbf_using_file()*

function to verify the integrity of the two-part interpolation process) simply by commenting out commands which call the *calc_rbf_factors_2d()* function, while leaving other parts of the script intact. The faster-running script file thereby produced could then be employed when using PLPROC as a parameter pre-processor for a model that is being calibrated by PEST (and hence being run repetitively by PEST).

Element selection

The *rbf_using_file()* function does not allow use of source or target selection equations. It is assumed that list element selection has already been carried out when undertaking pre-interpolation calculations using the selection function capabilities provided by the *calc_rbf_factor_2d()* function. If further target element restriction is required when undertaking actual PLIST-to-PLIST interpolation then a temporary PLIST can be filled; its values can then be transferred to the final target PLIST using PLPROC's equation functionality in conjunction with an appropriate selection equation.

Transform

Interpolation can be applied to log or native PLIST values, this depending on the value supplied for the *transform* argument of the *rbf_using_file()* function. If the logarithmic option is selected then all element values within the source PLIST which are used in the interpolation process must be positive. If this is not the case PLPROC will write an appropriate error message to the screen, and then cease execution.

Examples

Example 1

```
sm1=s1.rbf_using_file(file='fac1.dat';form='binary')
```

Using this command spatial interpolation is undertaken from the *s1* PLIST to the *sm1* PLIST using information stored in the binary file *fac1.dat*.

Example 2

```
sm1=s1.rbf_using_file(file='fac1.dat';form='binary',           &
                      transform='log',                         &
                      upper_limit=5,lower_limit=1)
```

This is similar to example 1, except for the following:

- interpolation is applied to the logs of source PLIST values; back-transformation to native values then takes place before storage of data in the target PLIST.
- upper and lower bounds are imposed on target PLIST elements.

read_column_data_file()

General

As the name suggests, function *read_column_data_file()* reads the values of elements pertaining to one or a number of SLISTs, PLISTs or MLISTs from a file in which data is arranged in columns. Correct operation of this function is predicated on the assumption that the ordering of data in each column is the same as that within pertinent LISTs. Hence the LIST element number to which an imported data value is assigned is implied in the position of the data element in the column from which it is read. List element ordering can be readily obtained using the *report_dependent_lists()* function.

All LISTs for which data is read using *read_column_data_file()* function must have already been initiated in pertinent PLPROC function calls prior to the calling of this function. If multiple lists are cited in a *read_column_data_file()* function, they must all have the same parent CLIST.

Function Specifications

Function value

Function *read_column_data_file()* makes no assignment. Its name must lead the PLPROC command through which it is invoked.

Arguments and subarguments

<i>file</i>	The name of the file from which LIST data must be read. Optionally the name of the file can be surrounded by quotes; this is mandatory if the name of the file contains a space.
<i>select</i>	Optionally, a selection equation can be provided. Data that is read from the file is directed only to LIST elements which are selected by this equation. See further discussion below.
<i>skiplines</i>	An integer value equal to the number of lines that must be skipped at the top of the file before reading data from file columns. If omitted, its value is assumed to be zero. This argument name can be specified as <i>skip</i> for short.
<i>slist</i>	The name of an SLIST for which integer data must be read from the external file. Data for more than one SLIST can be read by repeating this argument and its associated <i>column</i> subargument.
<i>slist; column</i>	This mandatory subargument specifies the column number (starting from the left) from which SLIST element values must be read. “ <i>column</i> ” can be reduced to “ <i>col</i> ” for short.
<i>plist</i>	The name of a PLIST for which real data must be read from the external file. Data for more than one PLIST can be read by repeating

	this argument and its associated <i>column</i> subargument.
<i>plist; column</i>	This mandatory subargument specifies the column number (starting from the left) from which PLIST element values must be read. “ <i>column</i> ” can be reduced to “ <i>col</i> ” for short.
<i>mlist</i>	The name of an MLIST. Data for all SLISTs or PLISTs associated with the MLIST will be read from the file. Data for SLISTs and PLISTs associated with more than one MLIST can be read by repeating this argument and its associated <i>startcol</i> subargument.
<i>mlist; startcol</i>	The column from which the first SLIST or PLIST associated with the MLIST is read. Data for other LISTs associated with the MLIST are read from subsequent columns in the same order in which they are featured in the MLIST.
<i>slist_ignore_thresh</i>	Any elements of any imported SLISTs (whether these be stand-alone SLISTs or part of an MLIST) whose absolute values are at or above this threshold are ignored; respective existing SLIST elements retain their original values. This argument name can be supplied as <i>slist_ignore</i> for short.
<i>plist_ignore_thresh</i>	Any elements of any imported PLISTs (whether these be stand-alone PLISTs or part of an MLIST) whose absolute values are equal to or above this threshold are ignored; respective existing PLIST elements retain their original values. This argument name can be supplied as <i>plist_ignore</i> for short.

Discussion

The file from which LIST data is read can be space, tab or comma-delimited. Data columns are counted from the left, starting at 1. Unlike the file which is read by the *read_list_file()* function, no coordinate or element identification columns need be present in a file read by the *read_column_data_file()* function. Numbers in each column are transferred directly to elements of pertinent SLISTs or PLISTs. Ordering of numbers in columns is assumed to be the same as ordering of numbers in SLIST/PLIST arrays, regardless of the element indexing convention associated with the parent CLIST of these LISTs.

Optionally, numbers within a data column of the external file can be ignored. This will occur if their absolute values are equal to, or greater than, thresholds supplied as values for the *slist_ignore_thresh* and *plist_ignore_thresh* arguments. In this case, existing SLIST or PLIST elements whose values would have been over-written by imported data retain their original values.

Further filtering of data transfer from the external file to pertinent SLISTs/PLISTs is offered through use of an optional selection equation. However this option must be used only with great caution. Data within columns of the input file are read from the top of the column with no gaps in reading this data, regardless of the presence of a selection equation in the *read_column_data_file()* function. However there will be gaps in the LIST elements to which these numbers are transferred if a selection equation is featured in this function. Any number

read from the input file is directed to the next LIST element that is selected by the selected equation; meanwhile the values of non-selected elements remain unchanged. Hence if a selection equation is employed in the *read_column_data_file()* function, the number of elements featured in each column of this file (i.e. the number of rows in the columns) can be less than the number of elements in the LISTS into which these numbers are read, as the importation process leaves data-transfer gaps spanning non-selected LIST elements.

Examples

Example 1

```
read_column_data_file(file='import.dat',      &
                      skiplines=4,           &
                      slist=z2;col=6,        &
                      plist=pp1;column=7,    &
                      plist=pp2;column=1,    &
                      mlist=pu*;startcol=9)
```

File *import.dat* is read to obtain data for the *z2* SLIST, the *pp1* and *pp2* PLISTs and all SLISTs or PLISTs which comprise the *pu** MLIST. Data for the latter starts in column 9 and progresses for as many columns as there are SLISTs/PLISTs associated with the *pu** MLIST. The reading of numbers begins on the fifth line of file *import.dat*.

Example 2

```
read_column_data_file(file='import.dat',      &
                      skiplines=4,           &
                      slist=z2;col=6,        &
                      plist=pp1;column=7,    &
                      plist=pp2;column=1,    &
                      mlist=pu*;startcol=9,  &
                      plist_ignore_thresh=1.1e20)
```

Assume that the *pu** MLIST is comprised of collected PLISTs rather than SLISTs. Then this example is the same as the previous example except for the fact that any numbers that are greater than 1.1e20 are not transferred to pertinent elements of the *pp1*, *pp2* and *pu** PLISTs. Instead these PLIST elements retain their existing values.

read_list_as_array()

General

Function *read_list_as_array()* reads a sequence of integer or real numbers from an existing file (which will probably be a model input file) into part or all of an existing SLIST or PLIST. The numbers can be recorded one-to-a-line or many-to-a-line in the file from which they are read; they can represent either a one-dimensional vector or a two-dimensional array in this file. In either case they can be space- or comma-delimited. The user identifies the location on the existing file where the numbers of interest lie by providing the file line number where the numbers begin.

Function Specifications

Function value

The *read_list_as_array()* function makes no assignments.

Object associations

Function *read_list_as_array()* operates on an existing SLIST or PLIST. Its call must follow the name of the SLIST or PLIST whose elements it reads; a dot must separate the name of the PLIST from the name of the function.

Arguments

<i>file</i>	The name of the file from which the sequence of numbers (possibly formatted as an array) is read.
<i>line</i>	The line number of the file at which the sequence of numbers begins. If this argument is omitted, the sequence of numbers is assumed to begin at the first line of the file.
<i>start_index</i>	The starting index of the SLIST or PLIST at which the sequence of numbers that is read from the file is stored. If this is omitted, it is assumed that <i>start_index</i> is the first element of the SLIST or PLIST.
<i>end_index</i>	The end index of the SLIST or PLIST at which the sequence of numbers read from the file is stored. If this is omitted, it is assumed that <i>end_index</i> is the final element of the SLIST or PLIST.

Discussion

With the exception of the *file* argument, all arguments of the *read_list_as_array()* function are optional. If the *line* argument is omitted the sequence of numbers to be read from the file is assumed to commence at the first line of the file.

If either one of the *start_index* or *end_index* arguments is provided in a call to function *read_list_as_array()*, then both of these arguments must be provided. If these arguments are omitted, then all elements of the SLIST or PLIST are read.

Function `read_list_as_array()` can be used as an alternative to function `read_multiple_array_file()`. However where many arrays or data lists are read from the same file, the `read_list_as_array()` function must be called many times whereas the `read_multiple_array_file()` function may need to be called only once. Nevertheless, there is no option but to employ the `read_list_as_array()` function where arrays of interest are not preceded by a unique text header in the file from which they are read.

Examples

Example 1

```
tt2.read_list_as_array(file=hk1.dat,           &
                       start_index=2275,end_index=3411, &
                       line=471)
```

The above call to the `read_list_as_array()` function reads data for the pre-defined *tt2* SLIST or PLIST from file *hk1.dat*. The list of numbers that it reads begins on line 471 of this file. Nothing other than these numbers is assumed to occupy this and ensuing lines of the file until all numbers are read. If these numbers are real numbers, then *tt2* must be a PLIST. If they are integers, *tt2* can be either an SLIST or a PLIST. Numbers which are read from *hk1.dat* are placed into elements of the *tt2* list starting at the element whose index is 2275 and ending at the element whose index 3411.

Example 2

```
tt2.read_list_as_array(file=hk1.dat)
```

In this example a series of numbers is read into the *tt2* SLIST or PLIST. As many numbers are read from file *hk1.dat* as there are elements in this list. Numbers read from file *hk1.dat* are assumed to begin at the first line of this file.

read_list_file()

General

Function *read_list_file()* reads CLIST, PLIST and/or SLIST data from a text file. In doing so it creates new lists of the nominated types. Only one CLIST can be created on a single reading of the file. However multiple PLISTs and SLISTs can be created during a single reading of the file. These can be linked to the CLIST that is optionally created on the same reading of the file, or to an existing CLIST that is already resident in PLPROC's memory.

A List File

A list file is a tabular data file. Optionally, headers can be provided to its columns. If this is the case, these headers are ignored.

Part of a list file follows.

pp	x	y	z	kh	kz	z1	z2	z3
11	345.234	274.234	53.1	23.4	0.001	1	111	111
12	346.234	377.234	25.1	13.4	0.002	1	121	111
13	347.234	372.234	35.1	3.4	0.003	2	132	112
14	348.234	302.234	55.1	83.4	0.004	2	142	112
15	545.234	574.234	21.1	23.4	0.005	1	151	111
16	546.234	577.234	21.1	13.4	0.006	1	161	111
17	447.234	572.234	51.1	3.4	0.007	2	172	112
18	448.234	302.234	51.1	83.4	0.008	9	182	112
19	447.234	572.234	51.1	3.4	0.090	2	172	112
20	448.234	402.234	51.1	83.4	0.010	9	182	112
21	345.234	374.234	53.1	23.4	0.001	1	111	111
.								
.								

Part of a list file.

While the specifications of a list file are flexible, certain aspects of its construction are mandatory. Most of these mandatory specifications pertain to the first column of the file. This first column must contain a list of CLIST element identifiers. If a CLIST is created through reading this file, the elements of this first column become the element identifiers of the created CLIST. If only PLIST or SLIST information is read from a list file, the information in this column links data entries on each line of the table to existing CLIST elements, this precluding the need for SLIST and PLIST data to be supplied in order of element storage in their reference CLIST. (Note that where lists are long, this is not a recommended procedure, as sequential access is much faster.)

As is discussed in the first part of this document, CLIST element identification can follow three different protocols, namely indexed, integer or character. Where a CLIST employs indexed element identification, the numbers in the first column of a list file must be sequential integers. Where a CLIST employs integer identification, they must be integers, but do not need to be sequential; however they must be unique. Where a CLIST uses character identification for its elements, the first column of a list file must contain character strings of 20 characters or less in length with no gaps. Each character element identifier must be unique.

The contents of the second, third and fourth columns of a list file depend on what information PLPROC is seeking from the file, this being defined through *read_list_file()* function arguments. If PLPROC's intention is to read and create a two-dimensional CLIST then it requires that x and y coordinates reside in the second and third columns of the list file; for a three-dimensional CLIST the fourth column must contain z coordinates. If CLIST data is not being read from a list file, coordinate columns are not required.

Columns from which SLIST data are read should contain integers. Either integers or real numbers can comprise the entries of columns from which PLIST elements are read.

Columns within a list file must be space or tab delimited. No data gaps are allowed.

Function Specifications

Function value

If the *read_list_file()* function is employed to define and read a CLIST, then optionally its outcome can be assigned to the name of the new CLIST that is defined through its use. If the name of the new CLIST is not provided in this way, then it must be provided as the value of the *clist* function argument.

If the *read_list_file()* function is employed to define and read new PLISTs and CLISTs whose reference CLIST has already been defined, then it makes no assignments.

Arguments and subarguments

<i>file</i>	The name of the list file from which CLIST, SLIST and/or PLIST data must be read. Optionally the name of the file can be surrounded by quotes; this is mandatory if its name contains a space.
<i>clist</i>	<p>If the <i>read_list_file()</i> function is employed to define a new CLIST then the value of this argument provides the name of the new CLIST. Alternatively, the name of the new CLIST can be provided through function assignment. Either of these protocols can be used, but not both.</p> <p>If a <i>clist</i> argument is provided, then a <i>reference_clist</i> argument must NOT be provided, as provision of the latter argument infers that the PLISTs and CLISTs which are being defined and read through the <i>read_list_file()</i> function call will be assigned to an existing CLIST, and that a new CLIST is therefore not being defined.</p>
<i>reference_clist</i>	<p>This argument must be used only if the <i>read_list_file()</i> function is employed to define and read SLISTs and/or PLISTs whose reference CLIST already resides within PLPROC's memory. The value supplied for this argument must be the name of the existing reference CLIST.</p> <p>If a call to function <i>read_list_file()</i> contains a <i>reference_clist</i> argument, then it must not contain a <i>clist</i> argument. Nor can its output be ascribed to a CLIST, for both of these conventions indicate that the <i>read_list_file()</i> function is being used to define a new CLIST, and that any new SLISTs and PLISTs read from the list file are assigned to that new CLIST so that</p>

the later becomes their reference CLIST. In contrast, the presence of a *reference_clist* argument indicates that an existing CLIST is serving as the reference CLIST for new SLISTs and PLISTs read from the list file.

dimensions

If the *read_list_file()* function is employed to define a new CLIST a *dimensions* argument must be provided. Its value must be supplied as “2” or “3”. The former informs PLPROC that the new CLIST defined through use of the *read_list_file()* function is to be a two-dimensional CLIST (requiring only *x* and *y* coordinates). In contrast, a *dimensions* value of “3” informs PLPROC that the new CLIST is three-dimensional. *x*, *y*, and *z* coordinates will then be read from the list file.

If the *read_list_file()* function is being used to define one or more SLISTs and/or PLISTs for an existing CLIST, then the *dimensions* argument need not be supplied. If it is supplied, then it must agree with the dimensions of the CLIST whose name is supplied as the value of the *reference_clist* argument.

id_type

If the *read_list_file()* function is employed to define a new CLIST this argument must be provided. Its value must be supplied as “indexed”, “integer” or “character”. The CLIST element identifiers will be read from the first column of the list file. If the *id_type* is “indexed” then the first column must be comprised of integers in sequence; these integers do not need to begin at 1. If the *id_type* is “integer” the first column must be comprised of integers which are unique, but not necessarily in sequence. If the *id_type* is “character” then string variables of up to 20 characters in length must be supplied in the first column; each character string must be unique, and contain no spaces.

If the *read_list_file()* function is used to define one or more SLISTs and/or PLISTs which refer to an existing CLIST, then the *id_type* argument need not be supplied. If it is supplied however, then it must agree with the identifier type of the CLIST whose name is supplied as the value of the *reference_clist* argument.

slist

The *slist* argument is optional. It is required only if an SLIST is to be read from the list file. Its value is the name of the new SLIST. This name must not clash with that of any existing PLPROC entities and must be 20 characters or less in length.

slist; column

If supplied, the *slist* argument requires a mandatory *column* subargument (which can be written as *col* for short). The value assigned to this subargument provides the number of the file column from which values for the new SLIST are to be read. Columns are numbered from left to right starting at 1. Data in the nominated column must be comprised of integers only.

plist

The *plist* argument is optional. It is required only if a PLIST is to be read from the list file. Its value is the name of the new PLIST. This name must not clash with that of any existing PLPROC entity and must be 20

characters or less in length.

- plist; column* If supplied, the *plist* argument requires a mandatory *column* subargument (which can be written as *col* for short). The value assigned to this subargument provides the number of the column from which values for the new PLIST are to be read. Columns are numbered from left to right starting at 1. The column from which PLIST data are read must contain only numbers (and no characters).
- skiplines* An integer value equal to the number of lines that PLPROC must skip at the top of the file before attempting to read data columns. This argument can be denoted as *skip* for short. If omitted its value is assumed to be zero.

Examples

Example 1

```
c11 = read_list_file(skiplines=1,dimensions=3,    &
                    plist='p1';column=5,        &
                    plist='p2';column=6,        &
                    slist=s1;column=7,          &
                    slist=s2;column=8,          &
                    slist=s3;column=9,          &
                    id_type='integer',file='list.dat')
```

In the above example PLPROC is asked to read file *list.dat*. Information in columns 1 to 4 of that file is used to define a new CLIST named *c11*. Integers occupy column 1 of file *list.dat*; these are used to identify each element of the CLIST, as well as SLISTs and PLISTs that depend on it. Because the CLIST is not declared as having an indexed identifier PLPROC will internally commence CLIST element indexing at 1. Depending on the context, either the index or the integer identifier can be used to refer to individual elements of *c11* or its dependent SLISTs/PLISTs in later processing. The *report_dependent_lists()* function can be used to see both element indices and integer element identifiers listed together.

The *c11* CLIST is three-dimensional. Hence the *read_list_file()* function reads *z* coordinates from the fourth column of file *list.dat*.

As well as reading and defining a new CLIST, the *read_list_file()* function reads and defines two new PLISTs (to which it gives the names *p1* and *p2*), and three new SLISTs (to which it gives the names *s1*, *s2* and *s3*). Data required to fill these lists are read from columns 5, 6, 7, 8 and 9 respectively of file *list.dat*.

Example 2

```
c11 = read_list_file(skiplines=1,dimensions=3,    &
                    plist='p1';column=5,        &
                    plist='p2';column=6,        &
                    slist=s1;column=7,          &
                    slist=s2;column=8,          &
                    slist=s3;column=9,          &
                    id_type='indexed',file='list.dat')
```

This is identical to the above example except for the fact that the *id_type* for the CLIST has been provided as “indexed”. Suppose that the list file from which data is read is that shown at

the beginning of this section. For this file the “indexed” *id_type* option is legal because the first column of the file is indeed comprised of sequential integers. In this case PLPROC commences its internal indexing of CLIST elements at 11 (because that is the first number in the first column of the file), and continues this indexing sequentially to as high a number as is necessary to read all CLIST elements. It also refrains from storing any integer or character identifiers for the CLIST, as this wastes memory and is unnecessary as indices are used as de facto integer identifiers for any CLIST whose element identifier protocol is “indexed”. Hence if an integer identifier is cited for *c11* or any of its dependent SLISTs/PLISTs in future PLPROC processing, PLPROC will use the element index as the integer identifier.

As already stated, use of indexed referencing allows faster access to elements of a CLIST, and to elements of SLISTs and PLISTs which employ it as their reference CLIST.

Example 3

```
read_list_file(reference_clist='c11',skiplines=1,      &
               plist='p1';column=5,                 &
               plist='p2';column=6,                 &
               slist=s1;column=7,                   &
               slist=s2;column=8,                   &
               slist=s3;column=9,                   &
               file='list.dat')
```

The function call illustrated in this example is not very different from that of the two previous examples. However those differences are vital. In particular, on this occasion the *read_list_file()* function does not define a CLIST. Instead it creates new SLISTs and PLISTs which employ an existing CLIST (named *c11*) as their reference CLIST. Because a new CLIST is not defined through this particular *read_list_file()* function call, there is no need to supply a *dimensions* or *id_type* argument. Furthermore the *read_list_file()* function does not need to read *x*, *y* or *z* coordinates from the list file; hence columns containing these coordinates are not required in that file.

In accordance with list file protocol however, the initial column of file *list.dat* must contain element identifiers. If the *id_type* of the existing *c11* CLIST is indexed, these element identifiers must be the same as element indices; they must therefore be sequential and begin and end at the correct indices (an error condition will be reported if this is not the case). Alternatively, if the CLIST employs integer or character element identification, the identifiers comprising the first column of the list file do not need to be supplied in the same order as that in which they were supplied when the CLIST was defined. In that case the *read_list_file()* function reorders the data as it reads it in order to concur with its internal ordering of *c11* CLIST element storage. However if file *list.dat* fails to provide values corresponding to all existing CLIST elements an error condition is reported. An error is also reported if an unrecognized element identifier is encountered.

Additional Notes

Calls to the *read_list_file()* function often occur early in a PLPROC script. Pilot point data can be stored in list file format, and then read into PLPROC using this function. Pilot point identifiers can be indexed, integer or character. Because pilot point lists are not in general large, the loss in storage and access efficiency incurred by using integer or character element identifiers is not very large; meanwhile the convenience of using integer or character identifiers could be considerable. On the other hand, indexed element identification is preferable for model-based CLISTs and their dependent SLISTs and PLISTs. In this case

each list element may correspond to a node of a model grid. Nodes may number in the hundreds of thousands, or maybe even millions. Element access times may be considerably increased through the use of integer or character identifiers.

read_matrix_from_file()

General

Function *read_matrix_from_file* reads a MATRIX from a text or binary file.

Function Specifications

Function value

Function *read_matrix_from_file()* fills a PLPROC MATRIX. The name of this MATRIX must precede the function call. The MATRIX must be new; that is, the matrix must not have been declared and filled earlier in a PLPROC script. An “=” sign must follow the name of the new matrix; the name of the function must follow that.

Arguments and subarguments

<i>file</i>	The name of the file from which the MATRIX is read.
<i>file; format</i>	<p>The <i>format</i> subargument of the <i>file</i> argument must be supplied as “formatted” or “binary”. In the former case an ASCII (i.e. text) file is read, whereas binary storage is assumed in the latter case. (The user can employ “text” or “ascii” instead of “formatted” if he/she wishes, and “unformatted” instead of “binary” if he/she so desires.)</p> <p>If this subargument is omitted it is assumed to be “formatted”.</p>

Discussion

The PLPROC matrix storage protocol allows text or binary file storage. In either case the stored matrix must be preceded by a header line that contains three integers, these being (in order) *nrow*, *ncol* and *icode*. *nrow* and *ncol* are the number of rows and columns in the matrix. *icode* must be -1 or 1. It can only be supplied as -1 if (a) the matrix is square, and (b) its only non-zero elements are along its diagonal. In that case only the diagonal elements are stored in the matrix file.

A matrix stored in a text file is recorded row by row, with element values along each row possibly wrapping onto successive lines of the file. Each row has *ncol* elements; *nrow* rows are stored in this way. On the other hand, if a matrix is stored in a binary file, then its elements are ordered in the manner in which they are most efficiently read by a FORTRAN program. They are read using the code:

```
read(iunit) ((array(irow,icol),irow=1,nrow),icol=1,ncol)
```

Examples

Example 1

```
covmat=read_matrix_from_file(file=covmat.dat)
```

This function instructs PLPROC to read the matrix COVMAT from the text file *covmat.dat*.

Example 2

```
covmat=read_matrix_from_file(file=covmat.dat;form=binary)
```

This example is identical to the above except that *covmat.dat* is a binary file.

read_mf_grid_specs()

General

Function `read_mf_grid_specs()` reads a MODFLOW grid specification file of the type employed by the PEST Groundwater Data Utilities. In doing so, it creates a two-dimensional CLIST whose coordinates pertain to the centres of the cells of the MODFLOW grid.

MODFLOW Grid Specification File

The figure below shows part of a grid specification file. Some specification details are discussed below that. See documentation of the PEST Groundwater Data Utility suite for further information.

```
120  149
234342  7247000  23.0
500.0  500.0  500.0  450.0  450.0  450.0  450.0  400.0  400.0  400.0
400.0  400.0  400.0  400.0  400.0  400.0  400.0  400.0  400.0  400.0
.
.
5*500.0 5*450 100*400 5*450 5*500
```

Part of a MODFLOW grid specification file.

The first line of a MODFLOW grid specification file contains two integers, these being *nrow* and *ncol*, the number of rows and columns respectively in the finite difference grid.

The next line of a grid specification file contains the east and then north coordinate of the top left corner of the MODFLOW grid. This is followed by another number, this defining the angle that the grid row direction makes with east; a positive angle signifies anti-clockwise rotation. An angle of zero signals alignment of the MODFLOW grid row direction with the easterly direction.

Then follow two one-dimensional arrays. The first is the MODFLOW *delr* (i.e. cell row-direction width) array, while the second is the MODFLOW *delc* (i.e. cell column direction width) array. The first has *ncol* elements while the second has *nrow* elements. These are read in list-directed format, and hence can wrap onto successive lines. They can also be recorded in list-directed shorthand, as depicted in the representation of the *delc* array in the above example.

Function Specifications

Function value

The `read_mf_grid_specs()` function creates a CLIST. The name of this CLIST constitutes its output value. A call to this function must thus follow the user-specified name of the CLIST which it creates; this must be followed by a “=” symbol.

Arguments

file The name of the grid specification file from which MODFLOW grid data is read.

Discussion

The CLIST created by the *read_mf_grid_specs()* function has *nrow* times *ncol* elements. These are ordered by increasing column number and then row number, with column number comprising the inner loop and row number comprising the outer loop. (This is the same protocol as that employed by MODFLOW). The new CLIST is endowed with an element identifier type of “indexed”. Element indexing starts at 1 and finishes at *nrow* times *ncol*.

Example

```
modflow_grid = read_mf_grid_specs(file="model.spc")
```

In the above call to function *read_mf_grid_specs()* MODFLOW grid specifications are read from file *model.spc*. A new CLIST named *modflow_grid* is created. If desired, specifications for this new CLIST can be obtained using the *report_all_entities()* function. Element coordinates can be obtained using the *report_dependent_lists()* function.

read_mf_int_array()

General

Function *read_mf_int_array()* reads an integer array from a MODFLOW-compatible integer array file, creating an SLIST in the process. It is assumed that a two-dimensional CLIST which holds the *x* and *y* coordinates of MODFLOW grid cell centres has already been created by PLPROC, this probably having been achieved using the *read_mf_grid_specs()* function. This becomes the reference CLIST for the new SLIST.

Integer Array File

This file type is discussed extensively in documentation of the PEST Groundwater Data Utilities. It consists of an *nrow* by *ncol* array of integers, where *nrow* and *ncol* specify the number of rows and columns respectively within a MODFLOW finite difference grid. Integers comprising each row of the array are read using list-directed formatting, and can wrap onto successive lines. However integers associated with each new row of the array are assumed to commence on a new line.

Optionally the integer array file can contain a header. This header is assumed to contain just two integers, these being the number of columns and number of rows respectively comprising the ensuing array.

Function Specifications

Function value

The *read_mf_int_array()* function creates an SLIST. The name of this SLIST constitutes its output value. The call to this function must thus follow the user-specified name of the SLIST which it creates; this must be followed by a “=” symbol.

Arguments

<i>file</i>	The name of the integer array file from which the MODFLOW-compatible integer array is read.
<i>reference_clist</i>	The name of a two-dimensional CLIST, already stored within PLPROC's memory, which holds the coordinates of the centres of the finite difference grid. This will become the reference CLIST for the new SLIST.
<i>colrow_header</i>	This argument is used to specify whether the integer array file contains a number-of-columns, number-of-rows header. Its value should be “yes” if the header is present and “no” if it is not. If this optional argument is omitted, its value is assumed to be “no”. This argument name can be written as <i>colrow</i> for short.

Discussion

The optional *colrow_header* argument indicates whether a number-of-columns, number-of-rows header is present in the integer array file which is read by PLPROC. If the integer array

file contains such a header, PLPROC checks that the number of rows and columns specified in this header is consistent with the number of elements in the reference CLIST whose name is supplied as the value of the *reference_clist* argument of the *read_mf_int_array()* function. An inconsistency between the two will precipitate an error message.

Examples

Example 1

```
ibound1=read_mf_int_array(reference_clist='modflow_grid',      &
                           file='ibound1.inf',               &
                           colrow_header='yes')
```

An integer array is read from a file named *ibound1.inf*. This file contains a number-of-columns, number-of-rows header. Integer array elements are assigned to a new SLIST named *ibound1*. The reference CLIST for the new SLIST is *modflow_grid*.

Example 2

```
ibound1=read_mf_int_array(reference_clist='modflow_grid',      &
                           file=ibound1.inf)
```

This command has the same outcome as that provided in the above example except that file *ibound1.inf* does not possess a number-of-columns, number-of-rows header.

read_mf_real_array()

General

Function *read_mf_real_array()* reads a real array from a MODFLOW-compatible real array file, creating a PLIST in the process. It is assumed that a two-dimensional CLIST which holds the *x* and *y* coordinates of MODFLOW grid cell centres has already been read by PLPROC, probably using the *read_mf_grid_specs()* function. This becomes the reference CLIST for the new PLIST.

Real Array File

This file type is discussed extensively in documentation of the PEST Groundwater Data Utilities. It consists of an *nrow* by *ncol* array of real numbers, where *nrow* and *ncol* specify the number of rows and columns respectively within a MODFLOW finite difference grid. Numbers comprising each row of the array are read using list-directed formatting, and can wrap onto successive lines. However numbers associated with each new row of the array are assumed to commence on a new line.

Optionally the real array file can contain a header. This header is assumed to contain just two integers, these being the number of columns and number of rows respectively represented in the ensuing array.

Function Specifications

Function value

The *read_mf_real_array()* function creates a PLIST. The name of this PLIST constitutes its output value. A call to this function must thus follow the user-specified name of the PLIST which it creates; this must be followed by a “=” symbol.

Arguments

<i>file</i>	The name of the real array file from which the MODFLOW-compatible array is read.
<i>reference_clist</i>	The name of a two-dimensional CLIST, already stored within PLPROC's memory, which holds coordinates of the centres of the finite difference grid. This will become the reference CLIST for the new PLIST.
<i>colrow_header</i>	This argument is used to specify whether the real array file contains a number-of-columns, number-of-rows header. Its value should be “yes” if the header is present and “no” if it is not. If this optional argument is omitted, its value is assumed to be “no”. The argument name can be written as <i>colrow</i> for short.

Discussion

The optional *colrow_header* argument indicates whether a number-of-columns, number-of-rows header is present in the real array file which is read by PLPROC. If the real array file

contains such a header, PLPROC checks that the number of rows and columns specified in this header is consistent with the number of elements in the reference CLIST whose name is supplied as the value of the *reference_clist* argument of the *read_mf_real_array()* function. An inconsistency between the two will precipitate an error message.

Examples

Example 1

```
trans1=read_mf_real_array(reference_clist='modflow_grid',      &
                           file='t1.ref',                    &
                           colrow_header='yes')
```

A real array is read from a file named *t1.ref*. This file contains a number-of-columns, number-of-rows header. Real array elements are assigned to a PLIST named *trans1*. The reference CLIST for the new PLIST is named *modflow_grid*.

Example 2

```
trans1=read_mf_real_array(reference_clist='modflow_grid',      &
                           file='t1.ref')
```

This command has the same outcome as that in the above example except that file *t1.ref* does not possess a number-of-columns, number-of-rows header.

read_mf_usg_grid_specs()

General

Function `read_mf_usg_grid_specs()` reads a MODFLOW-USG grid specification file. In doing so it creates a three-dimensional CLIST whose coordinates pertain to the nodes of the unstructured MODFLOW-USG grid. It also creates a three-dimensional SLIST which lists the layer number of each node.

MODFLOW-USG Grid Specification File

The figure below provides specifications for a MODFLOW-USG grid specification file. Note that, at the time of writing, these specifications are provisional and may change with time.

Line 1:

"UNSTRUCTURED GWF"

Note: If the "GWF" specifier is omitted, then it is assumed. "CLN" is another option for this specifier, but is not yet supported by PLPROC.

Line 2:

nnode nlay [iz ic]

where:

nnode is the number of nodes in the grid

nlay is the number of layers in the model

iz is 1 if elevations of node and mesh element vertices are supplied; 0 otherwise

ic is 1 if the cell specifications associated with each node are supplied; 0 otherwise

Note. PLPROC requires that both *iz* and *ic* be 1. Options associated with values other than 1 are not presented below. If these are omitted from the grid specification file they are assumed to be 1.

A list of vertex coordinates is now provided. These vertices will be cited by index later in this file where mesh geometric details are provided. The indices of vertices are defined implicitly by their positions in the following list. Numbering is assumed to begin at 1.

Line 3:

nvertex is the number of element vertex definitions to follow

NVERTEX next lines:

x, y, z

where:

x, y and z are the coordinates of each vertex

NNODE next lines:

inode x y z lay m (ivertex(i), i=1,m)

where:

inode is a node number (these must be supplied in increasing order starting from 1)

x, y and z are node coordinates

lay is the layer number of the node

m is the number of vertices defining the three-dimensional element associated with the node

ivertex are node indices defined with reference to the node list provided above

Provisional specifications for a MODFLOW-USG grid specification file.

As presently programmed, PLPROC reads a MODFLOW-USG grid specification file only if it describes an unstructured grid, and only if the *iz* variable on line 2 is set to 1 (indicating that *z* coordinates of grid nodes are provided and that full three-dimensional definition of the vertices surrounding each node is provided). If these conditions are not met PLPROC will cease execution with an appropriate error message.

Function Specifications**Function value**

The *read_mf_usg_grid_specs()* function creates a CLIST. The name of this CLIST constitutes its output value. The name of this function must follow the user-specified name of the CLIST which it creates; this is followed by a "=" symbol. Note that

read_mf_usg_grid_specs() also creates an SLIST. The name of this SLIST is optionally provided as a function argument.

Arguments

<i>file</i>	The name of the MODFLOW-USG grid specification file from which unstructured grid data is read.
<i>slist</i>	The name of the new SLIST created by the <i>read_mf_usg_grid_specs()</i> function. This SLIST contains the layer number of each grid node (read from the grid specification file). If the <i>slist</i> argument is omitted, the SLIST receives the default name of “layer”.

Examples

Example 1

```
musg_grid = read_mf_usg_grid_specs(file=model.gsf)
```

Using the above command a MODFLOW-USG grid specification file named *model.gsf* is read. The CLIST containing the horizontal and vertical coordinates of all grid nodes is assigned the name *musc_grid*. Node layer numbers read from file *model.gsf* are assigned to an SLIST which receives the default name of *layer*.

Example 2

```
musg_grid = read_mf_usg_grid_specs(file=model.gsf,slist=layerspec)
```

This is identical to the command provided in example 1 except for the fact that node layer numbers are assigned to a user-specified SLIST named *layerspec*.

read_mf6_grid_specs()

General

Function *read_mf6_grid_specs()* reads a binary grid file written by MODFLOW6. This file contains specifications for a MODFLOW6 grid, as well as other information such as the model's IDOMAIN array. *read_mf6_grid_specs()* builds a CLIST based on model grid cell centres, and optionally associates one or a number of SLISTs and PLISTs with this CLIST. The CLIST can be three dimensional; in this case it represents all cells of the model. Alternatively, it can be two-dimensional; in this case, each element of the CLIST is ascribed only an *x* and *y* coordinate, these pertaining to all model layers.

Note, however, that while *read_mf6_grid_specs()* will accommodate MODFLOW6 models with both DIS and DISV grids, it is not presently programmed to build a CLIST based on a MODFLOW6 DISU grid.

Function Specifications

Function value

As stated above, the *read_mf6_grid_specs()* function creates a CLIST. The name of this CLIST constitutes its output value. The name of the function must follow the user-specified name of the CLIST which it creates; they are separated by a “=” symbol.

Note that the CLIST which is produced by *read_mf6_grid_specs()* has an *id_type* of “indexed”. The index accords with node numbers awarded to model cells by MODFLOW6.

Arguments

<i>file</i>	The name of the MODFLOW6 binary grid file which <i>read_mf6_grid_specs()</i> must read. The extension of this file is generally “.grb”.
<i>dimensions</i>	The dimensions of the CLIST produced by <i>read_mf6_grid_specs()</i> . The value of this argument must be either “2” or “3”.
<i>slist_idomain</i>	The value of this optional argument should be the name of an SLIST. This argument is allowed only if the CLIST is three-dimensional. The SLIST will contain the IDOMAIN value of every cell comprising the model grid.
<i>slist_layernum</i>	The value of this optional argument should be the name of an SLIST. This argument is allowed only if the CLIST is three-dimensional. The SLIST will contain the layer number of every cell comprising the model grid.
<i>plist_bottom</i>	The value of this optional argument should be the name of a PLIST. This argument is allowed only if the CLIST is three-dimensional. The PLIST will contain the bottom elevation of every cell comprising the model grid.

<i>plist_top</i>	The value of this optional argument should be the name of a PLIST. This argument is allowed only if the CLIST is two-dimensional, or if the CLIST is three-dimensional and the model has only a single layer. The PLIST will contain the elevations of the tops of cells in the first layer of the model.
<i>slist_layer_idomain</i>	The value of this optional argument should be the name of an SLIST. This argument is allowed only if the CLIST is two-dimensional. The SLIST will contain the IDOMAIN value of every cell in a specified model layer. The number of this layer should be supplied through the subargument of this argument. The <i>slist_layer_idomain</i> argument may be repeated up to 10 times.
<i>slist_layer_idomain;</i> <i>layer</i>	The number of the layer for which IDOMAIN is required.
<i>plist_layer_bottom</i>	The value of this optional argument should be the name of a PLIST. This argument is allowed only if the CLIST is two-dimensional. The PLIST will contain the bottom elevation of every cell in a specified model layer. The number of this layer should be supplied through the subargument of this argument. The <i>plist_layer_bottom</i> argument may be repeated up to 10 times.
<i>plist_layer_bottom;</i> <i>layer</i>	The number of the layer for which bottom elevations are required.

Discussion

It is apparent from the above table that most arguments of the *read_mf6_grid_specs()* function are optional. Furthermore, some are incompatible with others.

At a bare minimum, function *read_mf6_grid_specs()* generates a CLIST. Elements of this CLIST can pertain to a single layer, or to an entire model grid. In the former case, the CLIST is two-dimensional while in the latter case it is three-dimensional. For three-dimensional CLISTs, the *z* coordinate of each element (each element pertains to a model cell) is evaluated as the average of the cell's top and bottom elevations. If desired, the CLIST can be accompanied by an SLIST which records cell IDOMAIN values, and by a PLIST which records cell bottom elevations.

If the CLIST which is generated by function *read_mf6_grid_specs()* is two-dimensional, then it pertains to no particular model layer, so *z* coordinates are not required. Nevertheless, the bottom elevations of one or more model layers can be ascribed to separate PLISTs using repeated *plist_layer_bottom* arguments. Similarly, IDOMAIN values for one or more model layers can be ascribed to separate SLISTs using repeated *slist_layer_idomain* arguments.

The optional *plist_top* argument should provide the name of a PLIST in which model top elevations are to be stored. Because this is a two-dimensional object, it requires a two-dimensional CLIST parent. Hence the value of the *dimensions* argument must be 2. However the *read_mf6_grid_specs()* function will allow a user to request a PLIST of model top elevations if *dimensions* is set to 3, provided the model possesses only a single layer.

Examples

Example 1

```
cl_mf6 = read_mf6_grid_specs(file=ci.disv.grb,      &
                             dimensions=3,         &
                             slist_layernum = layer, &
                             slist_idomain  = idomain, &
                             plist_bottom   = bottom)
```

In this example, grid specifications are read from file *ci.disv.grb*. A three-dimensional CLIST is sought. Complementary *layer* and *idomain* SLISTs will contain model layer numbers and IDOMAIN values. The *bottom* PLIST will contain cell bottom elevations.

Example 2

```
cl_mf6 = read_mf6_grid_specs(file=ci.disv.grb,      &
                             dimensions=2,         &
                             slist_layer_idomain=id1; layer=1, &
                             slist_layer_idomain=id2; layer=2, &
                             slist_layer_idomain=id3; layer=3, &
                             plist_layer_bottom =bot1; layer=1, &
                             plist_layer_bottom =bot2; layer=2, &
                             plist_layer_bottom =bot3; layer=3, &
                             plist_top          = top)
```

In this example, a two-dimensional CLIST is created. IDOMAIN values for three model layers are stored in SLISTs named *id1*, *id2* and *id3*. Bottom elevations for three model layers are stored in PLISTs named *bot1*, *bot2* and *bot3*. Model top elevations are stored in a PLIST named *top*.

read_multiple_array_file()

General

Through a single call to function *read_multiple_array_file()* part or all of multiple SLISTs and/or multiple PLISTs can be read from a single file. It is presumed that SLIST/PLIST element values are recorded in a manner that allows easy access under list-directed format control. This includes use of array format for the recording of element values, whereby these are stored on each of multiple (possibly wrapped) lines.

It is assumed that each such series of element values is preceded by a character string that identifies the stored element sequence to follow. Such a protocol is common in MODFLOW and MODFLOW-USG input files (though not part of the official protocol of either). The text header normally indicates that the following sequence of values represents the hydraulic property associated with a particular layer, an aerielly-disposed stress pertaining to a particular stress period, or an integer indicator array pertaining to either. While MODFLOW may store such information in array format, PLPROC (and MODFLOW-USG) stores such data as part or all of a list.

Multiple Array File

The figure below shows fragments of a MODFLOW input file.

```
# MODFLOW2000 Layer Property Flow (LPF) Package
50 -1.000000e+030 0 1
1 1 1
0 0 0
1.000000e+000
1.000000e+000
1.000000e+000
0 0 0
0 0 0
INTERNAL 1.000000e+000 (FREE) -1 Kx Layer 1
2.000000e+02 2.000000e+02 1.250000e+02 1.250000e+02 ...
2.000000e+02 2.000000e+02 2.000000e+02 1.250000e+02 ...
2.000000e+02 2.000000e+02 2.000000e+02 1.250000e+02 ...
2.000000e+02 3.000000e+02 3.250000e+02 3.240000e+02 ...
1.000000e+02 1.000000e+02 1.000000e+02 1.000000e+02 ...
.
.
7.490000e+02 7.490000e+02 7.490000e+02 7.490000e+02 ...
7.490000e+02 7.490000e+02 7.490000e+02 7.490000e+02 ...
8.250000e+02 8.250000e+02 8.250000e+02 8.250000e+02 ...
INTERNAL 1.000000e+000 (FREE) -1 Kz Layer 2
5.210000e+01 5.210000e+01 5.210000e+01 5.210000e+01 ...
5.210000e+01 5.210000e+01 5.210000e+01 3.360000e+01 ...
4.350000e+01 4.350000e+01 1.000000e+01 3.360000e+01 ...
4.350000e+01 4.350000e+01 1.000000e+01 3.360000e+01 ...
.
.
```

Parts of a MODFLOW input file.

Multiple MODFLOW input arrays are recorded in the file depicted in the above figure. Each array is preceded by a header. Though not part of formal MODFLOW protocol, each header includes a character string indicating the nature of the array to follow. This protocol is

employed by a number of commercial MODFLOW graphical user interfaces when writing MODFLOW input datasets.

A similar protocol can also be used in writing input files for MODFLOW-USG. In this case, although the sequence of numbers that follows the header may be recorded in similar format to that indicated in the preceding example, individual numbers within the sequence correspond to nodes rather than to cells. MODFLOW-USG input protocol requires that nodal hydraulic property values be recorded on a layer-by-layer and/or stress-period-by-stress-period basis. As a consequence, headers to layer-specific lists of values will often indicate the layer or stress period to which the list pertains.

The *read_multiple_array_file()* function makes use of these text headers in locating sequences of numbers which are to be read into part or all of an SLIST or PLIST. Regardless of whether these numbers comprise an array of MODFLOW hydraulic property values, a layer-specific sequence of MODFLOW-USG node hydraulic property values, or numbers employed by another model altogether, they are read into PLPROC as sequential elements within a list.

Function Specifications

Function value

Function *read_multiple_array_file()* makes no assignment. Its name must lead the PLPROC command line which invokes it.

Arguments and subarguments

<i>file</i>	The name of the multi-array file from which SLIST or PLIST data must be read. Optionally the name of the file can be surrounded by quotes; this is mandatory if the name of the file contains a space.
<i>plist</i>	The name of an existing PLIST for which element values are to be read. This argument can be repeated multiple times.
<i>plist; prev_line_string</i>	The <i>prev_line_string</i> subargument to the <i>plist</i> argument must contain a text string of 100 characters or less that allows unique recognition of the header preceding the location on the file from which numbers are to be read into the PLIST. This subargument to the <i>plist</i> argument is mandatory. It is also case insensitive.
<i>plist; start_index</i>	The starting index within an existing PLIST where numbers read from the input file are to be stored. This subargument can be omitted; if it is indeed omitted, the PLIST starting index is assumed to be 1.
<i>plist; end_index</i>	The end index within an existing PLIST where numbers read from the input file are to be stored. This subargument can be omitted; if it is indeed omitted, the end index is assumed to be the final index of the PLIST.
<i>slist</i>	The name of an existing SLIST for which element values are to be

	read. This argument can be repeated multiple times.
<i>slist; prev_line_string</i>	The <i>prev_line_string</i> subargument to the <i>slist</i> argument must contain a text string of 100 characters or less that allows unique recognition of the header preceding the location on the file from which numbers are to be read into the SLIST. This subargument to the <i>slist</i> argument is mandatory. It is also case insensitive.
<i>slist; start_index</i>	The starting index within an existing SLIST where numbers read from the input file are to be stored. This subargument can be omitted; if it is indeed omitted, the SLIST starting index is assumed to be 1.
<i>slist; end_index</i>	The end index within an existing SLIST where numbers read from the input file are to be stored. This subargument can be omitted; if it is indeed omitted, the end index is assumed to be the final index of the SLIST.

Discussion

The following aspects of *read_multiple_array_file()* function protocol are worthy of note.

- There is no limit to the number of SLISTS and PLISTS which can be read from a single file.
- Only integers can be stored as SLISTS. Either real numbers or integers can be read into PLISTS; integers are converted internally to double precision real numbers.
- The text string supplied as the value of the *prev_line_string* subargument of an *slist* or *plist* argument need only be long enough to uniquely indentify a specific text header. If it identifies more than one header, only the first will be found. If it does not identify any header, PLPROC will cease execution with an appropriate error message.
- If the *start_index* subargument to an *slist* or *plist* argument is provided, then so too must the *end_index* subargument be provided (and vice versa). If these subarguments are omitted, PLPROC will attempt to read the entirety of the SLIST or PLIST whose name is specified through the *slist* or *plist* argument.
- Any SLIST or PLIST cited in an argument to the *read_multiple_array_file()* function must have been defined through previous PLPROC commands.
- The reading of SLISTS and PLISTS can be mixed within a single *read_multiple_array_file()* call.
- The *read_multiple_array_file()* function can be used to read only a single SLIST or PLIST if desired.

Examples

Example 1

```
read_multiple_array_file(file=gv65n.lpf,                                &
plist=kx;prev_line_string='Kx Layer 3';start_index=2275;end_index=3411, &
plist=kx;prev_line_string='Kx Layer 1';start_index=1;end_index=1137,    &
plist=kx;prev_line_string='Kx Layer 2';start_index=1138;end_index=2274)
```

The above call to function `read_multiple_array_file()` reads different parts of the one composite *kx* PLIST from different parts of the one MODFLOW input file. This file is named *gv65n.lpf*. Note that the different PLIST fragments do not need to be cited in the `read_multiple_array_file()` function argument list in the same order as that in which they are recorded in the file which this function reads.

Example 2

```
read_multiple_array_file(file=input.dat,                                &  
                        plist=trans;prev_line_string='transmissivity')
```

Through the above function call, the entire *trans* PLIST is read from a string of numbers immediately following a header which includes the string “transmissivity” which uniquely identifies this header.

read_restart_data()

General

The *read_restart_data()* function instructs PLPROC to read a binary file written by the *save_restart_data()* function. If a PLPROC run commences with this call, execution can commence from the point at which the call to the *save_restart_data()* function was made in the previous PLPROC run.

Function Specifications

Function value

The *read_restart_data()* function makes no assignment. Its name must lead the PLPROC command in which it is invoked.

Arguments

<i>file</i>	The name of the binary file in which values of all PLPROC variables have been recorded on a previous call to the <i>save_restart_data()</i> function. Quotes (single or double) are optional; however if the name of the file contains a space, quotes are mandatory.
-------------	---

Discussion

Normally a call to *read_restart_data()* is made at the commencement of PLPROC execution. Hence it is the first function invoked in a PLPROC script. However there is no reason why it cannot be called following other function calls. If this is done, however, all PLPROC memory is erased; memory is re-allocated, and the values of all variables are re-assigned, in accordance with the contents of the binary restart file.

Example

```
read_restart_data(file=restart.dat)
```

Using the above command the values of all variables stored in PLPROC's memory are erased. They are then replaced with the values of variables stored in file *restart.dat*.

read_scalar_file()

General

As the name implies, function *read_scalar_file()* reads data from an external file; that data is then stored as individual scalars within PLPROC. These scalars are then available for use in PLPROC equations, and for writing to model input files through the agency of template files.

String variables can also be read from a scalar file.

Scalar File

A scalar file is simply a file whose data is arranged in columns. One of those columns must contain the names of data items; these will become the names of PLPROC scalar variables. Another column must contain the values of those variables. An example is shown below.

Item	name	value
1	anisotropy	4.5
2	recharge	1.0e-4
3	rech_multiplier	1.23
4	zone_4_evt	3.4e-5
.		
344	stringvar1	"here is a string"
345	stringvar2	'here is another string'
.		

An example of a scalar file.

Function Specifications

Function value

Function *read_scalar_file()* makes no assignment. Its name must lead the PLPROC command through which it is invoked.

Arguments

<i>file</i>	The name of the file from which scalar data must be read. Optionally the name of the file can be surrounded by quotes; this is mandatory if the name of the file contains a space.
<i>valuecolumn</i>	The number of the data column (counting from the left) from which the values of scalar variables must be read. This can be written as <i>valuecol</i> for short.
<i>namecol</i>	The number of the data column (counting from the left) from which scalar variable names are to be read. Names must be 20 characters or less in length and contain no spaces.
<i>skiplines</i>	An integer value equal to the number of lines that must be skipped at the top of the file before reading data columns. If omitted its value is assumed to be zero. This argument name can be specified as <i>skip</i> for short.

overwrite If this argument is supplied as “yes” then an existing value for a scalar or string value is over-written by the new value read from a scalar file. If it is supplied as “no” then PLPROC will cease execution with an error message if an attempt is made to re-assign a value to an existing scalar or string variable. This argument is optional; if omitted, PLPROC assumes “yes”.

Discussion

The *read_scalar_file()* function can assign values to both scalar and string variables. In reading the file it is able to distinguish one from the other by the fact that the value of a string variable is surrounded by single or double quotes. If an item in the *valuecolumn* column of a scalar file is not surrounded by quotes, then it is assumed to be a scalar. If it cannot be read as a number an error condition occurs and is reported as such.

Leading and trailing blanks are stripped from strings as they are read. A string cannot be empty, nor contain a tab character.

Example

```
read_scalar_file(file="scalars.dat", valuecolumn=3, namecolumn=2)
```

This command reads the example scalar file depicted above. The file is named *scalars.dat*. Scalar and string names are read from column 2 while the values associated with these names are read from column 3 of the file.

read_segfile()

General

As the name suggests, function *read_segfile()* reads a segmentation file (referred to herein as a “segfile” for short). This file provides specifications of a set of segments; this set is collectively referred to as a SEGLIST. Each segment is defined using two or more vertices; the line that joins a pair of vertices is referred to as a “sector”. Hence a segment is comprised of one or more sectors, while a SEGLIST is comprised of one or more segments.

In the groundwater modelling context, a segment may follow part of the trace of a stream, river, drain, general head boundary or other polylinear feature that appears in a model. Multiple segments may be used to cover the entire length of such a feature; the first point in one segment may correspond to the last point in another. A user may link each segment of a SEGLIST to pilot points using the *link_seglist_to_clist()* or *create_clist_from_seglist()* functions. Pilot points may then be employed as a parameterization device for these polylinear features.

The *calc_linear_interp_factors()* function provided by PLPROC supports interpolation from SEGLIST-linked pilot points to any point along any segment of a SEGLIST. Properties associated with model cells can then be ascribed values following this linear interpolation process. First PLPROC finds the segment point to which the model cell centre is closest; the search for this closest point is made over all segments of a SEGLIST. PLPROC then transfers the interpolated value from the segment point to the property of the user-specified cell.

Segment File

A segfile is illustrated below.

```
# The first segment

START SEGMENT
  SEGMENT_ID  "segment_1"
  345567.235   78345293.325
  345353.948   78345201.321
  344932.002   78345189.209
  etc
END SEGMENT

# The second segment

START SEGMENT
  SEGMENT_ID  segment_1
  334092.235   78349832.987
  334195.002   78349911.821
  334225.012   78349998.732
  etc
END SEGMENT

# The third segment
etc
```

An example of a segfile.

As is apparent from the above example, a segfile is divided into blocks. Each block begins with START SEGMENT or BEGIN SEGMENT. It must end with END SEGMENT.

Each segment block contains a list of vertices. As presently programmed, only an x and y coordinate can be associated with each vertex. Somewhere in each block a SEGMENT_ID keyword must be provided, this being followed by the name of the segment. This name must be 20 characters or less in length, and can optionally be surrounded by quotes. It must contain no spaces. No two segments within a segfile can possess the same name.

Blank lines and comments can appear anywhere within a segfile. The latter must follow a # character.

If a user desires, other blocks with other names can also appear in a segfile. However, all SEGMENT blocks must be juxtaposed in this file; they cannot be separated from each other by blocks of other types. A segfile can contain only one SEGLIST.

Function Specifications

Function value

The *read_segfile()* function creates an internally-stored SEGLIST. Optionally, the name of this SEGLIST constitutes the output value of the function. If this is the case, the name of this function must follow the user-specified name of the SEGLIST which it creates; this is followed by a “=” symbol. Alternatively, the name of the newly-created SEGLIST can be provided as a function argument.

Arguments

<i>file</i>	The name of the segfile from which a SEGLIST is read. Optionally the name of this file can be surrounded by quotes; quotes are mandatory if the name of the file contains a space.
<i>dimensions</i>	As presently programmed, this argument is optional. It specifies the dimensionality of the SEGLIST. If supplied, the value of the <i>dimensions</i> argument must be 2. Hence an x and y coordinate must be provided for each vertex of every segment comprising a SEGLIST. If a z coordinate is supplied for any vertex, it is ignored.
<i>seglist</i>	The name of the new SEGLIST. This must be 20 characters or less in length and contain no spaces. However if the name of the new SEGLIST is supplied as the value of the function (preceding an “=” sign in front of the function name), it must not be supplied as a function argument. As for all other PLPROC entities, the name of the SEGLIST must be unique.

Discussion

PLPROC can store numerous SEGLISTs in its memory. However the specifications of each stored SEGLIST must be read from the segfile in which they are recorded through a unique *read_segfile()* function call. Once a SEGLIST is stored in memory, it can be linked to one or more CLISTs using the *link_seglist_to_clist()* function. Alternatively (or as well) CLISTs can

be created specifically for linkage with this SEGLIST using the *create_clist_from_seglist()* function.

Example

```
rivers = read_segfile(file="riverseg.dat")
```

This function reads a SEGLIST from file *riverseg.dat*. Internally to PLPROC, the SEGLIST is identified by the name “rivers”.

remove()

General

The *remove()* function removes an SLIST, PLIST, CLIST, MATRIX or GPLANE from PLPROC's memory. However it will not remove a CLIST unless all SLISTs and PLISTs which are dependent on the CLIST are first removed. Nor will it remove an SLIST or PLIST which belongs to an MLIST; use the *mremove()* function to remove MLISTs from PLPROC's memory.

If removing a GPLANE the *remove()* function will also remove the CLIST with which the GPLANE is associated. However it will not remove the GPLANE and associated CLIST if any PLISTs or MLISTs are dependent on this CLIST.

Function Specifications

Function value

The *remove()* function makes no assignments. Its name must lead the PLPROC command through which it is invoked.

Object associations

Function *remove()* operates on an existing SLIST, PLIST, CLIST, MATRIX or GPLANE. Its name must follow the name of the entity whose removal it activates; a dot must separate the entity name from the function name.

Arguments

The *remove()* function has no arguments.

Example

```
temp.remove()
```

Suppose that *temp* is an SLIST or a PLIST. The above command will result in removal of the *temp* list from PLPROC's memory. However if *temp* is a CLIST, PLPROC first checks that *temp* has no dependent SLISTs or PLISTs. If any such dependent lists exist, PLPROC ceases execution with an appropriate error message. If they do not exist, the *temp* CLIST is removed from PLPROC's memory.

The *temp* entity can be a MATRIX. It can also be a GLANE; in this case the CLIST associated with the GPLANE is also removed (except if it has dependent SLISTs or PLISTs).

replace_cells_in_lists

General

In many ways, operation of function *replace_cells_in_lists()* is similar to that of function *find_cells_in_lists()*. Documentation for the latter function should be read in conjunction with documentation for *replace_cells_in_lists()*.

Like *find_cells_in_lists()*, function *replace_cells_in_lists()* identifies model cells that are cited in tables on model input files, and relates them to elements of a model-representative CLISTs stored by PLPROC. However function *replace_cells_in_lists()* then replaces elements of these tables (normally specifications for model boundary conditions) by respective elements of model-representative PLISTs.

Use of the *replace_cells_in_lists()* function assumes that a model input file contains one or a number of tables (often one for each model stress period) that list calibration-adjustable model properties. Each line of each of these tables pertains to an individual model cell. The cell is identified by an appropriate set of model-specific indices. These indices can pertain to a structured or unstructured grid. For a structured grid, it is assumed that three sequential columns of the table provide layer, row and column numbers of model cells (in that order). Two options are available for an unstructured grid. For a MODFLOW6 DISV grid, it is assumed that two sequential columns are devoted to cell identification using layer numbers and layer cell indices (in that order). Alternatively, for a MODFLOW6 DISU model (or any model with an unstructured grid), it is assumed that a single column provides cell node numbers. See documentation of *find_cells_in_lists()* for more details. It is further assumed that PLPROC's memory holds a CLIST (normally three-dimensional) that represents cells in a model grid, and that PLPROC's memory also holds at least one corresponding PLIST containing values that pertain to the cells of this grid. These may have been obtained from an external file using the *read_list_file()* function; alternatively, they may have been obtained through interpolation from PLISTs associated with other CLISTs using functions such as *calc_linear_interp_factors()* and *interp_using_file()*.

It is assumed that one or a number of other entries on each line of these model input tables provide values for model properties which pertain to a model cell identified in the above way. The table columns which contain these property values are identified through arguments of the *replace_cells_in_lists()* function. PLPROC replaces these numbers with values residing in, or calculated from, PLISTs that are associated with the model CLIST. The element of the PLIST which replaces the number corresponding to a particular cell is identified through model-to-CLIST cell-to-element correspondence. The value of this PLIST element can replace the existing tabulated value, or can be added to, or subtracted from it; alternatively, it can multiply the existing model property value, or divide the existing value.

As it replaces values in an existing model input file, the *replace_cells_in_lists()* function writes a new model input file for the consumption of the model.

Function Specifications

Function value

Function *replace_cells_in_lists()* makes no assignment. Its name must lead the PLPROC command through which it is invoked.

Arguments

<i>old_file</i>	Provide the name of an existing model input file which includes one or a number of tables containing lists of cells and accompanying model property values. This input file will be replaced by another file in which property values contained in these tables are altered.
<i>new_file</i>	Provide the name of the file to be written by PLPROC. This will be identical to <i>old_file</i> except for replaced values of cell properties in one or a number of tables contained in this file.
<i>keytext_start</i>	It is assumed that lists of cells in a model input file are collected into blocks (i.e. tables). These blocks are identified by an easily recognized text string on the line preceding each block in the model input file. At least part of that text string (enough for unique recognition) should be provided as the value of the <i>keytext_start</i> argument. If it contains a space, then the text string should be surrounded by quotes. PLPROC will object if the text string is blank. The search for this string in a model input file is case-insensitive.
<i>keytext_end</i>	This text string marks the end of a table listing model cell property values within a model input file. Optionally, it can be blank.
<i>blocks_to_read</i>	A model input file may be very lengthy. Tables of cell properties within that file may be repeated many times. Alternatively, a cell property table may occur just once, near the beginning of the file. In the latter case, it is not necessary for PLPROC to search the rest of the file for another table. Provide an integer value greater than zero for this argument. Suppose that this integer is <i>N</i> . Then the <i>replace_cells_in_lists()</i> function will discontinue its reading of the model input file, and its replacement of pertinent cell property values, after it has encountered <i>N</i> tables. Then it will copy the rest of the old model input file to the new model input file with no alterations to copied text. The <i>blocks_to_read</i> argument is optional; if it is not supplied, a very high number is assumed.
<i>list_col_start</i>	This argument can be supplied as <i>list_col</i> for short. This is the column number within each cell property table at which the listing of model cell indices begins. Often, but not always, this is column 1. As is described below, depending on the model's grid type, model cell indices may also occupy the following one or two columns.
<i>model_type</i>	The value of this argument is a text string. Depending on its value, up to three accompanying subarguments must be supplied. The value of the <i>model_type</i> argument must be supplied as "mf6_dis", "mf6_disv" or "undefined" ("undef" for short). Only in the "undefined" case are subarguments required.
<i>model_type;node</i>	Where <i>model_type</i> is provided as "undefined", then a <i>node</i> subargument may follow. This informs the <i>replace_cells_in_lists()</i> function of the

	number of nodes in the model. This must be the same as the number of elements in the CLIST to which PLISTs cited in this function pertain. Where a <i>node</i> subargument is supplied for the <i>model_type</i> argument, cell elements in tables in the model input file are identified by a single column of node numbers.
<i>model_type;nlay</i> <i>model_type;ncpl</i>	Where <i>model_type</i> is provided as “undefined”, then a <i>nlay</i> subargument may follow. Where a model is of the DISV type, then a <i>ncpl</i> (number of cells per layer) subargument must follow the <i>nlay</i> subargument. These are both integers. The product of <i>nlay</i> and <i>ncpl</i> must equal the number of elements in the parent CLIST of PLISTs cited in the <i>replace_cell_in_lists()</i> function. Where a model type is of the unstructured DISV type, then cells in tables in the model input file are identified by layer number and layer index number.
<i>model_type;nlay</i> <i>model_type;nrow</i> <i>model_type;ncol</i>	Where <i>model_type</i> is provided as “undefined”, then an <i>nlay</i> subargument may follow. Where the model is of the structured grid (i.e. DIS) type, then a <i>nrow</i> (number of rows) subargument must follow that; this must then be followed by a <i>ncol</i> (number of columns) subargument. The product of <i>nlay</i> , <i>nrow</i> and <i>ncol</i> must equal the number of elements in the CLIST to which PLISTs cited in the <i>replace_cells_in_lists()</i> function pertain; they represent the number of model layers, rows and columns respectively in the structured model grid. Where a model grid is structured, cells in tables in a model input file are identified by layer, row and column number (in that order).
<i>plist</i>	This argument can be repeated up to five times. Supply the name of a PLIST that is stored in PLPROC’s memory. The parent CLIST to all PLISTs that are supplied in this way must be the same. It must pertain to a model whose cells can be identified by model cell indices in one of the ways described above. Two subarguments must be supplied with every PLIST argument.
<i>plist;column</i>	The <i>column</i> subargument of the <i>plist</i> argument must provide a column number in which cell properties are listed in tables that are contained in the model input file. The contents of this column will be replaced by those derived from the nominated PLIST. The element of the PLIST whose value replaces a number which occupies this column on a particular line of the model input file is that which is associated with the model cell which is identified on that same line. The value supplied for the <i>column</i> subargument must be an integer. The identified column must not be any column used for model cell identification, for these numbers cannot be replaced.
<i>plist;action</i>	The value of this subargument must be supplied as “replace”, “add”, “subtract”, “multiply” or “divide”. The number that replaces a particular property value resident in a table on the existing model input file is obtained by direct replacement of the corresponding PLIST value. Alternatively, it is obtained by adding the PLIST value to the existing

value, subtracting it from the existing value, through multiplication of the existing value or by dividing into the existing value. The action is specified through the value supplied for this *action* subargument.

Discussion

The following figure shows part of a GHB (i.e. general head boundary) input file for MODFLOW6. This model is of the DISV type. Hence cells are identified by their layer number and cell layer index number. As is usual for this type of file, a table of cell property values is provided for a number of stress periods. The first two columns of all lines within each of these tables identifies a cell by its layer number and cell layer index. The elevation and head associated with the cell follow in subsequent table columns on the same line. The beginning of each table can be identified by the character string “BEGIN PERIOD”; each table ends with the string “END PERIOD”. (Note that PLPROC’s identification of these character strings is case-insensitive.)

```
BEGIN OPTIONS
END OPTIONS

BEGIN DIMENSIONS
  MAXBOUND 3173
END DIMENSIONS

BEGIN PERIOD 1
  1 1 0.0 20.575
  1 2 0.0 20.688
  1 3 0.0 18.346
  1 4 0.0 33.421
  1 5 0.0 18.341
  1 6 0.0 40.544
  1 7 0.0 19.972
  .
  .
END PERIOD

BEGIN PERIOD 2
  1 11 0.0 21.777
  1 12 0.0 20.682
  1 13 0.0 43.240
  1 14 0.0 20.724
  1 15 0.0 20.136
  1 16 0.0 41.007
  1 17 0.0 20.141
  1 18 0.0 19.725
  1 19 0.0 39.507
  1 20 0.0 19.707
  1 21 0.0 19.568
END PERIOD

etc
```

Part of a GHB input file for a MODFLOW6 DISV model.

Often model features such as general head boundaries are disposed along linear segments. PLPROC allows pilot point parameters to inform the properties of these segmented

boundaries. This functionality is implemented through functions such as *read_segfile()*, *link_seglist_to_clist()*, *create_clist_from_seglist()*, *find_cells_in_lists()* and *calc_linear_interp_factors()*. Meanwhile a CLIST based on the MODFLOW6 model can be constructed using the *read_mf6_grid_specs()* function.

Perhaps, at a particular stage of its processing, PLPROC has already perused this GHB input file using its *find_cells_in_lists()* function; model cells to which linear interpolation from pilot points must take place may thereby have been defined. Perhaps pilot points are employed for parameterization of both elevation and conductance of GHB cells. The locations of these points may have been read using the *read_list_file()* function. Meanwhile, MODFLOW6 model-compatible PLISTS are easily defined once the *read_mf6_grid_specs()* function has been used to construct an appropriate, model-compatible CLIST. These model-compatible PLISTS can then be populated from pilot point PLISTS using functions *calc_linear_interp_factors()* and *interp_using_file()*; model-compatible interpolation can be restricted to GHB cells cited in property tables by employing a selection SLIST built during a previous call to function *find_cells_in_lists()*. Thus-populated PLIST elements can then be recorded in columns 4 and 5 of each table in the above file using function *replace_cells_in_lists()*; this requires the reading of the above file and the writing of another. This is done prior to each model run; the model may therefore be comprised of a batch or script file that runs PLPROC prior to running MODFLOW6.

It is important to note the following details of the operation of function *replace_cells_in_lists()*.

1. Prior to writing a number to a column of a table contained in a model input file, PLPROC tries to read the number that it must replace. Where writing of a new number requires addition, subtraction, multiplication or division of this number by the corresponding PLIST element, this action is obviously required in order to obtain its value. However, while pre-reading of this number is not required for direct PLIST element replacement, PLPROC does it anyway. If PLPROC cannot read this number, it tolerates this condition; furthermore, it does not write a number to this space. This allows the *replace_cells_in_lists()* function to tolerate the name of a time series in place of a number in accordance with MODFLOW6 data input protocols.
2. When writing a number to a model input file, function *replace_cells_in_lists()* uses 16 characters to express that number. Thus the number can be recorded with a high degree of numerical precision. Whitespace between column entries is preserved on any line on which a number is replaced. However numbers may be shifted along the line as replacement occurs. This does not matter if the model reads this file using the free format protocol. However a model error condition will arise if the model tries to read the number using the fixed format convention.
3. If required, a PLPROC user can prevent number replacement for cells of his/her choice by populating corresponding PLIST elements with numbers whose absolute value is 1.0E35 or greater. If the *replace_cells_in_lists()* function encounters a PLIST element value of this magnitude when it attempts to replace a number in a model input file, it leaves the latter number as it is; this number thus retains its original value in the model input file.
4. Elements from a single PLIST can replace numbers in more than one column of tables that are resident in a model input file. However the *replace_cells_in_lists()* function forbids the same column from being populated by more than one PLIST.

Example 1

PLPROC reads a file named *ci.ghb* and writes a new file named *ci_new.ghb* which is identical to the existing file except for the replacement of numbers in columns 3 and 4 of tables occurring in this file. These tables are preceded by a line which includes the text “begin period” and are terminated by a line that includes the text “end period”. Elements of the *elev* PLIST replace numbers in column 3 of these tables. Elements of the *cond* PLIST act as multipliers on existing numbers which are recorded in column 4 of these tables.

Example 2

PLPROC reads a file named *file1.in* and replaces it with a file named *file_new.in*. This file pertains to a model with a structured grid; this model has 3 layers, 122 rows and 240 columns.

Within the table of interest, layer, row and column numbers (in that order) reside in columns 3, 4 and 5. Numbers in columns 6 and 7 are replaced with elements of PLIST *param1* and *param2* in that order.

```
replace_cells_in_lists(old_file=file1.in,                                &
                       new_file=file_new.in,                          &
                       model_type=undefined;nodes=45320,              &
                       list_col_start=3,                              &
                       keytext_start='specifications',                &
                       keytext_end='end',                              &
```

```
plist=param1;column=6;action='divide',    &  
plist=param2;column=7;action='subtract',  &  
blocks_to_read=1)
```

In this example, function *replace_cells_in_lists()* performs similar tasks to those described in the preceding example. However in this case the model grid is unstructured, and cells are identified by node number. The model has 45320 nodes. Node numbers appear in column 3 of the table.

replace_column()

General

Function *replace_column()* is employed as an embedded function in a template file. It must thus be invoked from that file on a line that begins with the string “\$#p”. As was explained in the first part of this manual, no information appearing on any line beginning with this string is transferred to the model input file which PLPROC writes on the basis of a template file; instead, such a line is presumed to contain instructions for PLPROC itself.

Use of the *replace_column()* function is predicated on the assumption that the template file contains a data table, and that this table immediately follows the line on which the *replace_column()* command is recorded. This table must be transferred to the model input file which is being written on the basis of the template file. However the contents of one or more of the columns of the transferred table must be altered so that numbers comprising all or part of that column contain the current values of user-specified PLIST elements.

The PLIST elements whose values are transferred to the model input file may be specified through arguments of the *replace_column()* function. Alternatively, they may be obtained from other columns of the table that is transferred to the model input file.

Function Specifications

Function value

Function *replace_column()* makes no assignment. Its name must lead the PLPROC command in which it is invoked. This command, in turn, must follow the character sequence “\$#p” on a template file.

Arguments and subarguments

<i>plist</i>	The name of a PLIST whose values will feature in a column of the ensuing data table.
<i>plist; column</i>	The <i>column</i> subargument of the <i>plist</i> argument is optional. This is the data column to which values of the specified PLIST will be written.
<i>plist; startpos</i>	The <i>startpos</i> subargument of the <i>plist</i> argument is optional; however if present, it must be accompanied by an <i>endpos</i> subargument. <i>startpos</i> is the starting character position in the ensuing table to which PLIST element values must be written.
<i>plist; endpos</i>	The <i>endpos</i> subargument of the <i>plist</i> argument is optional; however if present, it must be accompanied by a <i>startpos</i> subargument. <i>endpos</i> is the ending character position in the ensuing table to which PLIST element values must be written.
<i>charidcolumn</i>	This is an optional argument; it can be written as <i>charidcol</i> for short. It specifies the column of the ensuing table from which PLIST element character identifiers can be obtained.

<i>indexcolumn</i>	This is an optional argument; it can be written as <i>indexcol</i> for short. It specifies the column of the ensuing table from which PLIST element indices can be obtained.
<i>intidcolumn</i>	This is an optional argument. It can be written as <i>intidcol</i> for short. It specifies the column of the ensuing table from which PLIST element integer identifiers can be obtained.
<i>tablelength</i>	This is an optional argument. It specifies the number of entries in the table to follow.
<i>startindex</i>	This is an optional argument; however if present an <i>endindex</i> argument must also be supplied. <i>startindex</i> is the starting index for recording of PLIST values in the ensuing table.
<i>endindex</i>	This is an optional argument; however if present a <i>startindex</i> argument must also be supplied. <i>endindex</i> is the ending index for recording PLIST values in the ensuing table.

Discussion

The *replace_column()* function is complex. Different aspects of its functionality can be activated through different arguments. Many of these arguments are optional, and some of them are mutually exclusive.

The discussion and examples to follow assume that the following table exists in a template file that is associated with a particular model input file. Presumably the template file is identical to the model input file, but for a few minor alterations - these denoting where and how current PLIST values are to be recorded on the model input file. Let us suppose that element values of a certain PLIST are to be recorded in column 3 of the following table, and that element values from another PLIST are to be recorded in column 4.

1	1-1a	23.40000	9.689000E-03	123.40000
2	1-2b	13.40000	2.866000E-03	213.40000
3	1-3a	3.400000	3.407000E-03	33.400000
4	1-4b	83.40000	4.684000E-03	183.40000
5	1-5c	23.40000	5.237000E-03	223.40000
6	1-6a	13.40000	6.848000E-03	313.40000
7	1-7d	5.467800	4.312000E-03	23.400000
8	1-8c	83.40000	8.534000E-03	183.40000
9	1-9a	23.40000	9.141000E-02	13.400000
10	2-0b	88.40000	1.453000E-02	283.40000
etc				

The first part of a table residing in a template file corresponding to a model input file.

In the template file, a call to the embedded *replace_column()* command would be placed on the line immediately preceding the table. This command would, of course, be preceded by the “\$#p” string, this string being the first item on the line; through this means PLPROC recognizes that this line does not need to be transferred to the model input file. Continuation lines (also starting with the “\$#p” string) can be used as appropriate to render the embedded function more readable.

Suppose that element values of PLIST *p1* must be recorded in column 3 of the above table, and that element values of PLIST *p2* must be recorded in column 4 of the above table. It is mandatory that these two PLISTs possess the same reference CLIST. Thus indexing and/or element identifier recognition is the same for both of them.

Two options are available for denoting the positions, on each line of the table, to which numbers representing element values of a given PLIST must be written. The first option is to provide the table column number. This is done through providing a *column* subargument in association with each pertinent *plist* argument. PLPROC then determines the location of each identified column in the data table and replaces the current number in the denoted column by the appropriate PLIST element value. In making this replacement it writes the new number using 14 characters so that the maximum number of single precision significant figures can be employed in representing the number, irrespective of its size. Columns to the right of the replaced column may therefore be moved further to the right (they will never be moved to the left). If the model reads the data table using the free field formatting protocol this does not matter. However if the model is more strict in its formatting requirements, then the alternative positioning option offered by the *replace_column()* function must be employed instead. This is to specify the actual character positions which define each column, and to which each number must thus be written. This is done using the *startpos* and *endpos* subarguments of the *plist* argument. Character positions are numbered from the left of each line, starting at 1. If you use this option, make the parameter space as wide as possible. As explained in PEST documentation and in the first part of this manual, the integrity of finite difference derivatives calculation depends on this.

The following rules apply.

- Either a *column* or *startpos/endpos* subargument must be supplied with every *plist* argument.
- The *column* and *startpos/endpos* subarguments are mutually exclusive for any one *plist* argument.
- If there are multiple *plist* arguments, then all of these must use either the *column* subargument or the *startpos/endpos* subarguments.
- If a *startpos* subargument is supplied for a *plist* argument, an *endpos* subargument must also be supplied.
- For any *plist* argument, the value of the *endpos* subargument must exceed that of the *startpos* subargument.
- Where there are multiple *plist* arguments, the values of *startpos* and *endpos* subarguments must not define overlapping character positions on the template file.

There are two ways in which PLIST elements are selected for appearance in the table. Where multiple *plists* are cited in the *replace_column()* argument list, the chosen method applies to all of them. The easiest method is to simply specify the PLIST starting and ending indices. This is done using the *startindex* and *endindex* function arguments. In this case it is assumed that PLIST elements must be recorded sequentially in the table. The length of the table (as far as PLIST output is concerned) is implicitly defined through this mechanism.

Alternatively, PLPROC can be directed to ascertain the PLIST element number to which each line of the table pertains by reading information from one of the other columns of the table. Three options are available for doing this, the chosen option being supplied through one of the three (mutually exclusive) *indexcolumn*, *intidcolumn* or *charidcolumn* arguments. Through one of these arguments a number must be supplied; this number must identify a

column of the table. As it encounters each line of the table while transferring it to the model input file, PLPROC tries to read the contents of the nominated column. For the *indexcolumn* and *intidcolumn* options it must find an integer in this column. In the *charidcolumn* case it will interpret whatever it finds in the identified column as a character string. PLPROC then determines which PLIST element the index/integer/character string identifies. This defines the element of any PLIST value that is then written to another column of the table.

The following rules apply.

- If a value is supplied for the *indexcolumn* argument, PLPROC interprets the integer that it reads from the nominated column as the index of the required element.
- If a value is supplied for the *intidcolumn* argument, PLPROC interprets the integer that it reads from the nominated column as the integer identifier of the required element. If the reference CLIST for the PLIST(s) whose value(s) must be written to other columns of the table employs indexed element identification, this is equivalent to the element index. If it is endowed with integer element identification, this is the value of the integer identifier of the required element. If the PLIST is endowed with character element identification an error condition will arise, in which case PLPROC will report the error and cease execution.
- If a value is supplied for the *charidcolumn* argument PLPROC interprets the character string that it encounters in the nominated column as the character identifier of the required element. The reference CLIST of the PLIST(s) whose value(s) must be written to the model input file must, of course, have been assigned character element identifier status.

If any of the *indexcolumn*, *intidcolumn* or *charidcolumn* options for identification of PLIST elements for table output are employed, then a value must be supplied for the *tablelength* argument - this indicating the number of rows in the table. This informs PLPROC when it should stop trying to read element information from table columns.

Regardless of the option chosen for identification of PLIST elements, if PLPROC encounters a blank line within a table before having transferred the requested number of PLIST element values to the model input file, it simply transfers the blank line to the model input file without incrementing the table row count. Further processing of the table then continues as if the blank line had not been encountered at all.

Finally, it should be noted that function *replace_column()* cannot accommodate a table with comma-delimited columns. The nuances of variable column width, including the possibility of zero column width where there are missing table entries, would become very difficult to handle. The use of tabs to delimit columns is also forbidden for similar reasons.

Examples

Example 1

```
$#p  replace_column(plist=p1;col=3,                                &
$#p                                plist=p2;col=4,startindex=1,endindex=10)
```

In this example, elements of the *p1* PLIST are written to column 3 while those of the *p2* PLIST are written to column 4. PLIST element values are written sequentially starting at 1 and finishing at 10.

Example 2

```
$#p replace_column(plist=p1;col=3,plist=p2;col=4,      &  
$#p                charidcol=2,tablelength=20)
```

Once again, values for the elements of the *p1* PLIST are written to column 3 while those of the *p2* PLIST are written to column 4. However PLPROC reads the second column of the table to obtain a character identifier for the PLIST elements which will be written to the same row of the table. It is presumed that the reference CLIST for both of the *p1* and *p2* PLISTS employs a character identifier protocol.

Example 3

```
$#p  replace_column(plist=p1;startpos=15;endpos=32,      &  
$#p                plist=p2;startpos=42;endpos=58,      &  
$#p                indexcol=1,tablelength=10)
```

The values of *p1* PLIST elements are written to a space on the model input file spanning character positions 25 to 32 (inclusive) while those of *p2* PLIST elements are written to a space spanning character positions 42 to 58 of the model input file. It is the user's responsibility to ensure that numbers in columns 3 and 4 of the existing table are thereby overwritten. Numbers will be written to these allotted character positions using as many significant figures as those character positions allow. Element indices are read from column 1. The table has 10 rows.

report_all_entities()

General

The *report_all_entities()* function can assist a PLPROC user to debug or refine his/her PLPROC script. It reports to an output file the following information:

- the names and design specifications of all declared CLISTs;
- the names of all declared SLISTs and PLISTs, together with the parent CLIST of each;
- the names of all MLISTs, together with the parent CLIST of each, the type of each (i.e. whether the MLIST collects PLISTs or SLISTs), and the starting and ending index of collected LISTs;
- the names and values of all declared scalars and strings.

Function Specifications

Function value

The *report_all_entities()* function makes no assignment. Its name must lead the PLPROC command in which it is invoked.

Arguments

<i>file</i>	The name of the file to which entity information is recorded. Quotes (single or double) are optional; however if the name of the file contains a space, then quotes are mandatory.
<i>position</i>	Values for this argument are “a” or “append” on the one hand, or “r” or “rewind” on the other hand. In the former case entity information is appended to information residing in an existing file. In the latter case a new file is written. If the <i>position</i> argument is omitted, “rewind” is assumed.

Example

```
report_all_entities(file=temp2.dat)
```

Using the above command, entity information is written to file *temp2.dat*. If file *temp2.dat* already exists, it is overwritten.

report_dependent_lists()

General

Through the *report_dependent_lists()* function the PLPROC user obtains a listing of all SLIST and PLIST data associated with a nominated CLIST, as well as all data associated with the CLIST itself. This can provide a useful record of PLPROC calculations. It also allows the user to link element numbers to list element identifiers, and to obtain other information that may be useful in PLPROC script debugging.

Function Specifications

Function value

Function *report_dependent_lists()* makes no assignment. Its name must lead the PLPROC command in which it is invoked.

Object associations

Function *report_dependent_lists()* operates on an existing CLIST. Its name must follow the name of the CLIST whose reporting it governs; this must be followed by a dot.

Arguments

<i>file</i>	The name of the file to which CLIST and dependent information is recorded. Quotes (single or double) are optional in supplying the name of the file. However if the name of the file contains a space, then quotes are mandatory.
<i>position</i>	Values for this argument are “a” or “append” on the one hand, or “r” or “rewind” on the other hand. In the former case entity information is appended to information already residing in an existing file. In the latter case a new file is written. If the <i>position</i> argument is omitted, then “rewind” is assumed.

Example

```
c11.report_dependent_lists(file='report.dat',position='append')
```

Using the above command, data pertaining to the *c11* CLIST, as well as data pertaining to all SLISTs and PLISTs for which *c11* is the reference CLIST, is appended to the end of file *report.dat*.

save_restart_data()

General

The *save_restart_data()* function instructs PLPROC to record all entities that it presently holds in its memory in a binary file. This file can be subsequently read using the *read_restart_data()* function on the same PLPROC run or, more usually, on a later PLPROC run. In the latter case, PLPROC execution can proceed from the point at which the *save_restart_data()* function was executed as if all processing up to that point had been undertaken on the current run.

Function Specifications

Function value

The *save_restart_data()* function makes no assignment. Its name must lead the PLPROC command in which it is invoked.

Arguments

<i>file</i>	The name of the binary file in which values of all PLPROC variables are to be recorded. Quotes (single or double) are optional; however if the name of the file contains a space, quotes are mandatory.
-------------	---

Example

```
save_restart_data(file=restart.dat)
```

Using the above command the values of all variables stored in PLPROC's memory are recorded in file *restart.dat*. If file *restart.dat* already exists, it is overwritten. This file can be subsequently read using the *read_restart_data()* function.

stat()

General

The *stat()* function calculates a statistic based on all or some elements of a PLIST. As presently programmed, this statistic can be the mean, maximum value, minimum value, value range (i.e. maximum minus minimum), standard deviation or geometric average of PLIST elements. The result is written to an existing or new scalar. In the latter case, the scalar is brought into existence through operation of the function. (Note that the geometric average option is allowed only if all elements of the PLIST involved in this calculation are greater than zero.)

Function Specifications

Function value

Function *stat()* assigns a value to a scalar. Hence the function must appear following an “=” symbol, which follows the name of a new or existing scalar.

Object associations

Function *stat()* operates on an existing PLIST. Its name must follow the name of the PLIST on which it operates; the function and PLIST names must be separated by a dot.

Arguments

<i>statistic</i>	This specifies the type of statistic to be calculated. It must be “mean”, “min”, “max”, “range”, “stdev” or “geomav”.
<i>select</i>	A PLIST selection equation which can be employed to restrict the range of PLIST elements over which the statistic is calculated.

Example

Example 1

```
sc2=hk_pp.stat(statistic=stdev)
```

Following this command, the scalar variable *sc2* contains the standard deviation of all elements comprising the *hk_pp* PLIST.

Example 2

```
sc2=hk_pp.stat(statistic=stdev,select=(zone==2))
```

This is similar to the above example. However in this case, calculation of the standard deviation is restricted to those elements of the *hk_pp* PLIST for which elements of the zone SLIST are equal to 2. Note that *zone* must have the same parent CLIST as *hk_pp*.

stop()

General

The *stop* command instructs PLPROC to cease execution immediately, regardless of the presence (or otherwise) of any ensuing command lines in the PLPROC script. It can be useful in debugging a PLPROC script.

Example

The following figure shows part of a complex PLPROC script.

```
calc_rbf_factors_2d(target_clist=modflow_grid;select=(ib1==1),      &
                    source_clist=cl_pp;select=z1==1,              &
                    file='fac1.dat';form='binary',                &
                    anis_ratio=3,anis_bearing=60,                 &
                    rbf=mq;epsilon=5e-5,                          &
                    constant_term='yes',                           &
                    linear_term='yes')

calc_rbf_factors_2d(target_clist=modflow_grid;select=(ib1==2),      &
                    source_clist=cl_pp;select=z1==2,              &
                    file=fac2.dat;form='binary',                  &
                    rbf=tps)

# -- Now undertake interpolation.

sm1=s1.rbf_using_file(file='fac1.dat';form='binary',                &
                      transform='log',upper_limit=5,               &
                      lower_limit=1)
modflow_grid.report_dependent_lists(file='temp.dat')
stop
sm1=s1.rbf_using_file(file='fac2.dat';form='binary',transform='none')

write_model_input_file(template_file='StoRage.tpl',
model_input_file='storage.ref')

# -- Further interpolation is undertaken

sm2(select=(ib1==1))=s1.rbf_interpolate_2d(transform='log',          &
      upper_limit=5,lower_limit=1,                                  &
      anis_ratio=3,anis_bearing=60,                                &
      rbf=mq;epsilon=5e-5,                                          &
      constant_term='yes',                                          &
      linear_term='yes',                                            &
      report_file='report.dat';position=rewind,                     &
      select=z1==1)
```

Part of a PLPROC script.

A *stop* command has been inserted between two *rbf_using file()* commands. Immediately preceding the *stop* command, a *report_dependent_lists()* function has been inserted. This strategy allows the user to inspect the contents of the *sm1* PLIST prior to subjecting that list to further processing using the following *rbf_using_file()* command. Any unintended consequences of the use of the first *rbf_using_file()* command are thereby more easily detected.

upscale_by_averaging()

General

Use of *upscale_by_averaging()* presumes the existence of two PLISTS, one pertaining to a coarsely-gridded model and one pertaining to a finely-gridded model. It is assumed that elements of these PLISTS represent property values assigned to nodes/cells which comprise the grid elements of these models. The *upscale_by_averaging()* function calculates property values for coarse model cells from those of fine model cells through averaging of the latter.

The *upscale_by_averaging()* function is informed of the relationships between fine and coarse model cells (and hence the PLIST elements which pertain to them) through reading a “node association file”. This file can be written by the USGDUALMODEL program from the Groundwater Data Utilities suite. Though this utility was written with MODFLOW-USG in mind, it could be used with other models as well, as long as an unstructured grid specification file (see documentation of function *read_mf_usg_grid_specs()* for a description of this file type) is used to define model node and vertex locations. Hence the processing undertaken by *upscale_by_averaging()* should not be considered to be MODFLOW-USG specific. It is important to note, however, that inherent in processing undertaken by this function is the notion that node numbers increase with layer number.

Because no spatial interpolation is performed, *upscale_by_averaging()* makes no use of the spatial coordinates associated with PLIST elements. However it does assume that element numbers coincide with model node numbers for both the coarse and fine models, as this provides the linkage between model node data provided in the node association file and element data provided in PLPROC PLISTS. Thus element numbering for parent CLISTS must begin at 1. There must be as many elements in each PLIST as there are nodes in the corresponding model. Furthermore, references to elements of the coarse and fine model parent CLISTS must be indexed rather than integer or character based. This is readily assured if both the coarse and fine model CLISTS are created through a call to the *read_mf_usg_grid_specs()* function; the unstructured grid specification file associated with the coarse and fine models is thereby read in both cases.

The specifications of a node association file are not provided herein; see documentation for the USGDUALMODEL program for contents of this file. It contains the following information, much (but not all) of which is used by the *upscale_by_averaging()* function:

- number of nodes (i.e. cells) and layers in each of the fine and coarse model grids;
- number of nodes in each layer of both of the fine and coarse model grids;
- area and (possibly) thickness of each coarse and fine model cell;
- coarse model cell in which each fine model cell is situated;
- fine models cells which lie within the confines of each coarse model cell.

The last of above quantities can take a while to compute where fine and coarse models contain many cells. PLPROC’s upscaling calculations as embodied in the *upscale_by_averaging()* function are made easier by the fact that much of the “heavy lifting” involved in associating fine model cells with coarse model cells has already been done for it. This allows it to better fulfil its role as a model pre-processor, for which small run times are important.

Function Specifications

Function value

The name of the PLIST to which interpolation takes place (i.e. the target PLIST) comprises the output of function *upscale_by_averaging()*. This name must appear before the function name in a PLPROC command which invokes the function; it should be separated from the function name by a “=” symbol. Optionally, a target selection equation can be supplied with the name of the target PLIST. This PLIST should have been created in previous PLPROC processing as it is not created by the *upscale_by_averaging()* function itself. Target PLIST elements which are not assigned values because of the presence of a target selection equation retain their existing values.

Object associations

Function *upscale_by_averaging ()* calculates values for elements of a target PLIST (this pertaining to a coarse model) from those of a source PLIST (this pertaining to a fine model). A call to this function must follow the name of the source PLIST on which it operates; a dot must separate the PLIST name from the function name. Source PLIST element selection can be undertaken using a selection equation supplied as an argument of the *upscale_by_averaging* function.

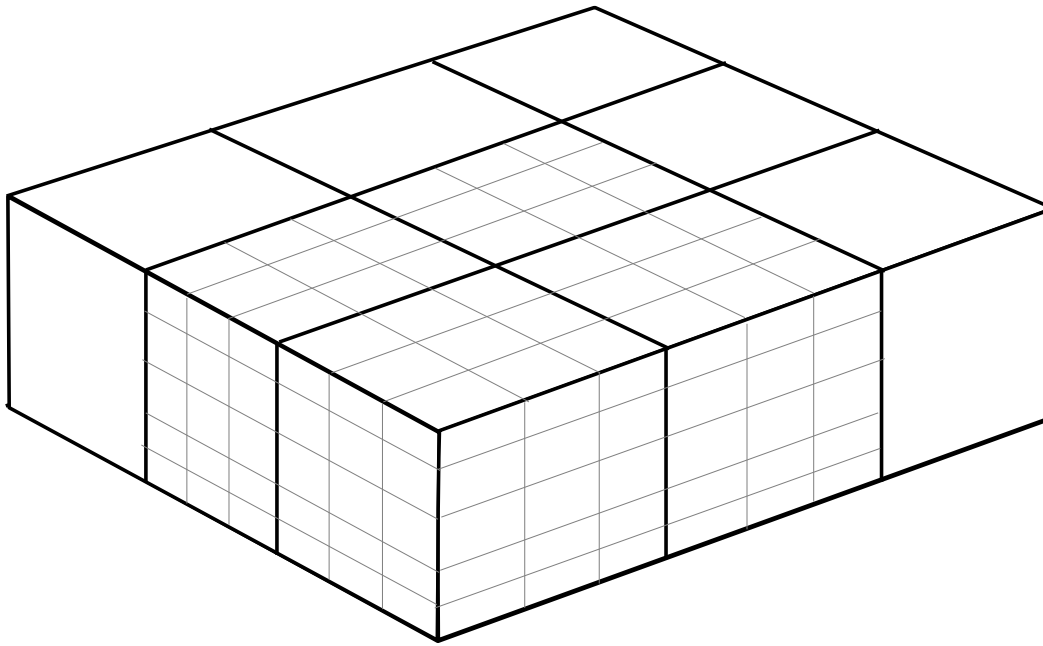
Arguments and subarguments

<i>select</i>	The source PLIST selection equation. This argument is optional. Note that a target PLIST selection equation can also be supplied if desired. The latter must be placed adjacent to the name of the target PLIST on the left side of the assignment operator; see the examples below. This argument is optional.
<i>node_assoc_file</i>	The name of the node association file that governs the upscaling process. This argument is mandatory. (The name of this argument can also be written as “node_association_file”.)
<i>stages</i>	This argument is optional. If it is not supplied a default value of 1 is assumed. See details of upscaling stages below.
<i>hor_averaging</i>	The type of averaging performed for stage 1 averaging, or for global averaging if there is only one stage. Allowed arguments are “geometric”, “arithmetic”, “harmonic”, “maximum” and “minimum”. This is a mandatory argument.
<i>hor_averaging;</i> <i>areaweight</i>	This is an optional subargument. A value of either “yes” or “no” is required. If set to “yes” (which is the default if this subargument is omitted), then the geometric, arithmetic or harmonic averaging process assigns weights to element values equal to the horizontal areas of fine model cells to which source PLIST elements pertain. (Cell areas are read from the node association file.) Note, however, that if <i>hor_averaging</i> is set to “maximum” or “minimum”, then this subargument must be

	omitted as weighting is irrelevant.
<i>hor_averaging;</i> <i>thicknessweight</i>	This is an optional subargument. A value of “yes” or “no” is required. If set to “yes”, then the geometric, arithmetic or harmonic averaging process assigns weights to element values equal to the thicknesses of fine model cells to which source PLIST elements pertain. (Thicknesses are read from the node association file.) Note, however, that if <i>hor_averaging</i> is set to “maximum” or “minimum”, then this subargument must be omitted as weighting is irrelevant. The default value is “no”.
<i>vert_averaging</i>	This argument must be supplied if <i>stages</i> is set to 2 but must be omitted if <i>stages</i> is set to 1. Options are “geometric”, “arithmetic”, “harmonic”, “maximum” or “minimum”.
<i>vert_averaging;</i> <i>thicknessweight</i>	This optional subargument of the <i>vert_averaging</i> argument must be given a value of either “yes” or “no”. The default is “no”. If supplied as “yes”, weights applied during vertical averaging of stage-1-calculated sublayer averages are equal to the average sublayer thicknesses; see below. Note, however, that if the value supplied for <i>vert_averaging</i> is “maximum” or “minimum”, then this subargument must be omitted as weighting is irrelevant in this case.
<i>upper_limit</i>	Optional. No target PLIST element is assigned a value greater than this.
<i>lower_limit</i>	Optional. No target PLIST element is assigned a value lower than this.

Discussion

The figure below shows the grid of a coarse model; cell boundaries are defined by dark lines. Part of the grid has been refined both laterally and vertically to produce a fine model; fine model cell boundaries are represented by light grey lines. Grid nesting functionality required to produce a locally fine model from a coarse model in this way is available in many MODFLOW graphical user interfaces.



A coarse model grid with local horizontal and vertical cell refinement.

Suppose that hydraulic properties (for example hydraulic conductivities) are available for all fine model cells and it is desired that corresponding properties be calculated for all coarse model cells. This may be required where PEST is calibrating the fine model using a parameterization device such as pilot points. Fine model hydraulic properties may have been assigned using, for example, the *rbf_interpolate_2d()* or *rbf_interpolate_3d()* functions. PEST may then be using a coarse model counterpart of the fine model through its observation re-referencing functionality to undertake parameter derivative calculations, restricting use of the fine model to testing of parameter upgrades (see PEST documentation for details). Where a coarse and fine model are used together in this manner, a means must exist to rapidly calculate coarse model properties from fine model properties.

In that part of the model domain which has not been subject to grid refinement, no upscaling is necessary. All of the upscaling methods supported by the *upscale_by_averaging()* function provide the same upscaled hydraulic property in this case, this being the hydraulic property used by the fine model for the same cell.

Consider now the coarse model cell in the south western corner of the grid (the cell closest to the observer in the above picture). This contains 45 fine model cells. If the *stage* argument of the *upscale_by_averaging()* function is set to 1 (or omitted) then the coarse cell hydraulic property is calculated from all 45 fine cell hydraulic properties at once. Five options are available for implementation of this calculation, these being (weighted) geometric, arithmetic or harmonic averaging, and using the maximum or minimum value of the hydraulic property over all of the 45 cells.

Weighted averaging is implemented using the following formulae:

Geometric averaging

$$k_c = \exp \left[\frac{\sum w_i \ln(k_i)}{\sum w_i} \right]$$

Arithmetic averaging

$$k_c = \frac{\sum w_i k_i}{\sum w_i}$$

Harmonic averaging

$$\frac{1}{k_c} = \frac{\sum \frac{w_i}{k_i}}{\sum w_i}$$

In all of these cases k_c refers to the calculated coarse cell hydraulic property. Meanwhile k_i refers to the property of fine cell i ; w_i is the weight applied to this hydraulic property during the averaging process. The weight can be assigned as equal to the area (in plan view) of each fine cell and/or to the vertical thickness of each fine cell; alternatively the user can specify that uniform weighting be applied.

A second alternative is to calculate upscaled hydraulic properties using a two-stage process in which horizontal (weighted) averaging (or maximization/minimization) is undertaken first; this is then followed by vertical (weighted) averaging or maximization/minimization. For the coarse cell at the south western corner of the above grid, a hydraulic property would first be calculated for each of the five fine-cell sublayers. In each case only 9 fine model cells would be involved in the (weighted) averaging or maximization/minimization process through which each of these sublayer properties is calculated, namely those belonging to each horizontal sublayer. These calculations would result in five horizontally-upscaled hydraulic properties. These five hydraulic property values would then be combined to form a total coarse cell hydraulic property. The same five options are available for computation of this final hydraulic property value, namely (weighted) geometric, arithmetic or harmonic averaging, or use of the maximum or minimum of the five sublayer hydraulic properties. If averaging is undertaken, then only one weighting option is available, this being weighting according to sublayer thickness. The sublayer thickness is calculated as the weighted arithmetic average of fine cell thicknesses within each sublayer, with weighting being proportional to cell area.

Note that if thickness weighting is requested when undertaking either horizontal, vertical or global averaging, then fine cell thicknesses are read from the node association file. Recording of fine cell thicknesses is optional for the USGDUALMODEL program that writes this file, and so must be specifically requested when running that program.

Examples

Example 1

```
kx_coarse = kx_fine.upscale_by_averaging(                                &
    node_assoc_file="node_assoc.dat",                                   &
    hor_averaging='geometric')
```

In this case only one stage of averaging is employed, regardless of the number of fine cell sublayers in any coarse model cell. Averaging thus takes place over all fine cells within any coarse cell, regardless of the disposition of these fine cells. Averaging is geometric with weights equated to fine cell areas (this being the default if the *areaweighting* subargument is

not used). Note that even though averaging is global within each coarse cell, the *hor_averaging* argument is nevertheless employed to specify the averaging process.

Example 2

```
kx_coarse (select=(layer_coarse>2)) =           &
    kx_fine.upscale_by_averaging(                 &
    node_association_file="node_assoc.dat",       &
    stages=2,                                     &
    hor_averaging=geometric; areaweight='yes';thicknessweight='yes', &
    vert_averaging=harmonic;thicknessweight=yes,  &
    select=(ibound_fine.ne.0))
```

This example illustrates two-stage averaging. Horizontal averaging is geometric with averaging weights calculated as the product of fine cell areas and thicknesses. Vertical averaging is harmonic, with weights calculated as the average thickness of each fine cell sublayer. If any coarse model cell is comprised of only one layer of fine model cells, then stage 2 interpolation is not carried out as it is unnecessary.

Example 2 also demonstrates the use of source and target selection equations. Where target cells are not assigned upscaled values due to operation of one or both of these selection equations, their values are unaltered from those assigned through previous PLPROC function calls.

write_column_data_file()

General

Function `write_column_data_file()` provides an easy means to export SLIST, PLIST and MLIST data to a file which can then be readily imported into a spreadsheet or other software package. Exported data can be space, comma or tab delimited. Optionally, CLIST element identifiers and/or coordinates can also be exported as their own data columns.

Function Specifications

Function value

Function `write_column_data_file()` makes no assignment. Its name must lead the PLPROC command through which it is invoked.

Arguments and subarguments

<i>file</i>	The name of the file to which LIST data is exported. Optionally the name of the file can be surrounded by quotes; this is mandatory if the name of the file contains a space.
<i>file; delim</i>	The type of delimiter used in the file. Legal values for this subargument are “space”, “tab” or “comma”. If this subargument is omitted the column delimiter defaults to “space”.
<i>select</i>	Optionally a selection equation can be provided. LIST data export is restricted to elements that are selected through this equation.
<i>header</i>	Optionally a header can be written to the top of the file. This contains the names of SLISTs and PLISTs whose elemental data fills respective columns. Values for this argument must be supplied as “yes” or “no”. If the argument is omitted, the default is “no”.
<i>slist</i>	The name of an exported SLIST. This argument must be repeated for each exported SLIST.
<i>plist</i>	The name of an exported PLIST. This argument must be repeated for each exported PLIST.
<i>mlist</i>	The name of an MLIST. Data for all SLISTs or PLISTs associated with the MLIST are exported. This argument must be repeated for each exported MLIST.
<i>clist_spec</i>	Optionally the character, integer or index identifier of the parent CLIST to all exported SLISTs and PLISTs can be written to the output file. So too can <i>x</i> , <i>y</i> and <i>z</i> (if the CLIST is three-dimensional) element coordinates. Values for this argument are “id”, “x”, “y” or “z”. Repeat the argument to request export of these different specifications.

Discussion

All SLISTs and PLISTs denoted for export through the `write_column_data_file()` function must have a common parent CLIST. SLISTs and PLISTs can be named individually for export through `slist` and `plist` function arguments. Alternatively (or as well) SLISTs or PLISTs can be named collectively through reference into an MLIST in which a suite of LISTs is assimilated.

LISTs are written to the output file in the order in which they are cited as function arguments. However LISTs which are contained in MLISTs always follow LISTs which are specified individually. If CLIST specifications are exported, these always comprise leading columns of the file written by the `write_column_data_file()` function, with the CLIST element id (if requested) being followed by element x , y , and z coordinates.

Where a selection equation is supplied, data pertaining to only selected elements is exported.

Examples

Example 1

```
write_column_data_file(header='yes',           &
                      file='export1.csv';delim="comma", &
                      slist=z1,                &
                      slist=z2,                &
                      mlist=pu*,                &
                      plist=pp1,               &
                      mlist=ppu*)
```

A comma-delimited file containing all data pertaining to cited SLISTs, PLISTs and MLISTs is recorded. The ordering of columns in the file is $z1$, $z2$, $pp1$, LISTs associated with pu^* and finally LISTs associated with ppu^* .

Example 2

```
write_column_data_file(header='yes',           &
                      file='export1.csv';delim="comma", &
                      slist=z1,                &
                      slist=z2,                &
                      mlist=pu*,                &
                      plist=pp1,               &
                      mlist=ppu*,              &
                      clist_spec='id',          &
                      clist_spec='x',          &
                      clist_spec='y',          &
                      clist_spec='z')
```

This is identical to the previous example except that the file to which LIST element data is written has four leading columns containing, in order, element identifiers, and the x , y and z coordinates of elements.

write_in_sequence()

General

Function *write_in_sequence()* is employed as an embedded function in a template file. It must therefore appear in that file on a line that begins with the string “\$#p”. As was discussed in the first part of this manual, no information that appears on any line that begins with this character string is transferred to the model input file which PLPROC writes on the basis of the template file; instead such a line is used to govern PLPROC behaviour.

The *write_in_sequence()* function commands PLPROC to transfer values associated with some or all elements belonging to a single PLIST to the model input file which it is writing, starting at the current line of that file.

Function Specifications

Function value

Function *write_in_sequence()* makes no assignment.

Object associations

Function *write_in_sequence()* is associated with a PLIST. Its call must follow the name of the PLIST whose output it governs, with a dot between the two.

Arguments

<i>start_index</i>	The index of the first PLIST element for which values are transferred to the model input file. This argument may be written as <i>start</i> for short. It may be omitted if the entire PLIST is written to the model input file.
<i>end_index</i>	The index of the last PLIST element for which values are transferred to the model input file. This argument may be written as <i>end</i> for short. It may be omitted if the entire PLIST is written to the model input file.
<i>format</i>	The value of this argument must be a FORTRAN format string which specifies the manner in which PLIST element values are transferred to a model input file. Alternatively, the text “free” can be used for space-delimited output wherein each number, regardless of its size, occupies 14 characters on the model input file. If this argument is omitted, the “free” option is employed.
<i>number_per_line</i>	The number of element values to record on each line of the model input file before wrapping occurs. If omitted, a default value of 8 is assumed.
<i>new_line_interval</i>	Regardless of the <i>number_per_line</i> setting, the recording of PLIST element values will start on a new line at intervals of <i>new_line_interval</i> PLPROC element values. If omitted, new line functionality is not invoked and the <i>number_per_line</i> argument exercises complete control over the number of PLIST element values recorded on each line of the

model input file.

Examples

Example 1

```
$#p sm1.write_in_sequence()
```

This invocation of the *write_in_sequence()* function instructs PLPROC to write all elements of the *sm1* PLIST to the model input file which it is currently writing on the basis of the template file in which the function is embedded. Numbers are written eight to a line with up to 14 character for each, and with a space between each number.

Example 2

```
$#p sm1.write_in_sequence(format="free",number_per_line=8)
```

This has the same effect as the function call of example 1.

Example 3

```
$#p sm1.write_in_sequence(format="(8(1x,1pg14.7))")
```

This has the same effect as the function call of examples 1 and 2.

Example 4

```
$#p sm1.write_in_sequence(new_line_interval=115,number_per_line=8)
```

Numbers are written 8 to a line using up to 14 characters per number with a space in between each number. However a new line is started at intervals of 115 numbers, regardless of how many numbers are written to the line containing the 115th number. This would be a suitable protocol to employ where members of the *sm1* PLIST represent hydraulic property values associated with a model grid which possesses 115 columns. Property values are therefore written in wrapped-line, free-field array format.

write_matrix_to_file()

General

Function *write_matrix_to_file* records a MATRIX in a text or binary file.

Function Specifications

Function value

Function *write_matrix_to_file()* makes no assignment. The name of the function must follow the name of a PLPROC MATRIX entity, with a dot separating the two.

Arguments and subarguments

<i>file</i>	The name of the file in which the MATRIX is recorded.
<i>file; format</i>	The <i>format</i> subargument of the <i>file</i> argument must be supplied as “formatted” or “binary”. In the former case an ASCII (i.e. text) file is written, whereas binary storage is employed in the latter case. (The user can employ “text” or “ascii” instead of “formatted” if he/she wishes, and “unformatted” instead of “binary” if he/she so desires.) If this subargument is omitted it is assumed to be “formatted”.

Examples

Example 1

```
covmat.write_matrix_to_file(file=covmat.dat)
```

This function instructs PLPROC to record the matrix COVMAT in text format in file *covmat.dat*.

Example 2

```
covmat.write_matrix_to_file(file=covmat.dat;form=binary)
```

This example is identical to the above except for *covmat.dat* being a binary file.

write_model_input_file()

General

Function *write_model_input_file()* is used to write the values of scalars and PLIST elements to model input files using templates prepared from these files.

See Chapter 5 of this document for a discussion of how PLPROC transfers data to model input files using template files.

Function Specifications

Function value

Function *write_model_input_file()* makes no assignment. Hence its name must lead the PLPROC command from which it is invoked.

Arguments

template_file The name of a user-prepared template of a model input file. Parameter delimiters and embedded functions contained within this file determine the nature of PLPROC data transfer to the model input file.

model_input_file The name of model input file which must be written by PLPROC.

Example

```
write_model_input_file(template_file='input.tpl',                                &
                        model_input_file= 'input.dat')
```

The above command tells PLPROC to read a template file named *input.tpl*. Using parameter spaces and embedded commands found in that file, PLPROC must write a model input file named *input.dat*. Note that either single quotes or double quotes can surround a filename. Neither of these is required if the name of the file contains no spaces.