#### **PEST++**

**Version 4**



Jeremy White, David Welter, John Doherty

CAELUM

Collective for Advancing Environmental Modelling in Decision Support

**Acknowledgements**

Initial funding for the development of PEST++ was provided by the United States Geological Survey (USGS). Further support for development of programs comprising the PEST++ suite was provided by the USGS and by GNS Science, New Zealand while Jeremy White was employed by these institutions.

The writing of this manual, and improvements to the PEST++ parallel run manager, were funded by the Queensland Water Modelling Network.

On a personal note, thanks are also due to the following people who have contributed in the distant or more recent past to the development of the PEST++ suite, and to modeling education based on the PEST++ suite:

Chas Egan (Queensland Department of Environment and Science)

Mike Toews (GNS Science)

Mike Fienen (USGS)

Randy Hunt (USGS)

Chris Muffels (SS Papadopoulos and Associates)

**Preface**

The release of version 4 of PEST++ coincides with the release of version 15 of PEST. The relationship between these two suites is described in a companion to this manual; see Doherty et al (2018).

Interoperability between programs of the PEST++ suite and those of the PEST suite has been greatly improved with the release of version 15 of PEST. This, it is hoped, will promulgate conjunctive use of these two suites. In particular, many of the utility programs that comprise the PEST suite (including its ground and surface water utilities) can perform the same support roles for programs of the PEST++ suite that they do for PEST itself. At the same time, the ever-expanding functionality of the PyEMU suite (White et al, 2016) can facilitate use of PEST as it does the same for PEST++.

The present document replaces two USGS reports that served as manuals for previous versions of PEST++. These are Welter et al (2012) and Welter et al (2015).

The term “PEST++” no longer describes a single program. It now describes a suite of programs which have certain things in common. Chief among these is their ability to conduct many model runs in parallel on a single computer, over an office network, on a high performance cluster and on the computing cloud. Members of the PEST++ suite employ the same protocol for communicating with a model as PEST does. They all employ a modular, parallel run manager named PANTHER, an upgrade from the YAMR run manager used by previous versions of PEST++. PANTHER is freely available to other programmers. Its sophisticated run management functionality is accessible through functions calls that can be issued from a number of programming languages. See the PANTHER manual for details.

A theme that runs through all members of the PEST++ suite is that of adding value to environmental modeling as it is conducted in the decision-making context. Programs which comprise the PEST++ suite apply state-of-the-art methods to model calibration and exploration of post-calibration parameter and predictive uncertainty. Their algorithms are capable of handling large numbers of parameters and of deploying advanced regularisation devices. One member of the suite optimizes environmental system management in a modeling context that respects the (sometimes high) levels of uncertainty that accompany all model predictions.

The PEST++ suite is still under active development. The theme of enhancing model-based decision-support is being maintained. The free availability of the PANTHER run manager constitutes an invitation for others to join us in this important work.

Jeremy White

Dave Welter

John Doherty

**License**

Programs of the PEST++ suite are distributed under the following license. This is the so-called “MIT license”. It is explained at the following site.

<https://opensource.org/licenses/MIT>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

**Table of Contents**

[1. Introduction 1](#_Toc520535147)

[1.1 PEST++ and PEST 1](#_Toc520535148)

[1.2 Software Installation 3](#_Toc520535149)

[1.3 This Document 3](#_Toc520535150)

[1.4 A Model: Some Considerations 3](#_Toc520535151)

[1.4.1 Running a Model 3](#_Toc520535152)

[1.4.2 Model Input and Output Files 4](#_Toc520535153)

[1.4.3 Differentiability of Model Outputs 5](#_Toc520535154)

[1.4.4 Observations and Predictions 6](#_Toc520535155)

[2. The PEST(++) Model Interface 7](#_Toc520535156)

[2.1 Introduction 7](#_Toc520535157)

[2.2 PEST++ Input Files 7](#_Toc520535158)

[2.3 Template Files 7](#_Toc520535159)

[2.3.1 Model Input Files 7](#_Toc520535160)

[2.3.2 An Example 8](#_Toc520535161)

[2.3.3 The Parameter Delimiter 9](#_Toc520535162)

[2.3.4 Parameter Names 9](#_Toc520535163)

[2.3.5 Setting the Parameter Space Width 10](#_Toc520535164)

[2.3.6 How a Parameter Space is Filled with a Number 11](#_Toc520535165)

[2.3.7 Multiple Occurrences of the Same Parameter 12](#_Toc520535166)

[2.3.8 Preparing a Template File 12](#_Toc520535167)

[2.4 Instruction Files 12](#_Toc520535168)

[2.4.1 Precision in Model Output Files 13](#_Toc520535169)

[2.4.2 How Model Output Files are Read 13](#_Toc520535170)

[2.4.3 An Example Instruction File 14](#_Toc520535171)

[2.4.4 The Marker Delimiter 14](#_Toc520535172)

[2.4.5 Observation Names 15](#_Toc520535173)

[2.4.6 The Instruction Set 15](#_Toc520535174)

[2.4.7 Making an Instruction File 26](#_Toc520535175)

[3. Some Important PEST++ Features 27](#_Toc520535176)

[3.1 General 27](#_Toc520535177)

[3.2 Parameter Adjustment 27](#_Toc520535178)

[3.2.1 Parameter Transformation 27](#_Toc520535179)

[3.2.2 Fixed and Tied Parameters 27](#_Toc520535180)

[3.2.3 Upper and Lower Parameter Bounds 28](#_Toc520535181)

[3.2.4 Scale and Offset 28](#_Toc520535182)

[3.2.5 Parameter Change Limits 29](#_Toc520535183)

[3.3 Calculation of Derivatives 30](#_Toc520535184)

[3.3.1 General 30](#_Toc520535185)

[3.3.2 Forward or Central Differences 31](#_Toc520535186)

[3.3.3 Parameter Increments for Two and Three-Point Derivatives 32](#_Toc520535187)

[3.3.4 Settings for Three-Point Derivatives 34](#_Toc520535188)

[3.3.5 How to Obtain Derivatives You Can Trust 34](#_Toc520535189)

[3.3.6 Looking at Model Outputs under the Magnifying Glass 35](#_Toc520535190)

[3.4 The Jacobian Matrix File 36](#_Toc520535191)

[3.5 The Objective Function 37](#_Toc520535192)

[4. The PEST Control File 39](#_Toc520535193)

[4.1 General 39](#_Toc520535194)

[4.2 Naming Conventions 39](#_Toc520535195)

[4.3 Sections 40](#_Toc520535196)

[4.4 Control Variables 41](#_Toc520535197)

[4.5 The PESTCHEK Utility 44](#_Toc520535198)

[4.6 Control Data Section 44](#_Toc520535199)

[4.6.1 General 44](#_Toc520535200)

[4.6.2 First Line 45](#_Toc520535201)

[4.6.3 Second Line 45](#_Toc520535202)

[4.6.4 Third Line 45](#_Toc520535203)

[4.6.5 Fourth Line 45](#_Toc520535204)

[4.6.6 Fifth Line 46](#_Toc520535205)

[4.6.7 Sixth Line 46](#_Toc520535206)

[4.6.8 Seventh Line 46](#_Toc520535207)

[4.6.9 Eighth Line 47](#_Toc520535208)

[4.6.10 Ninth Line 47](#_Toc520535209)

[4.7 Singular Value Decomposition Section 48](#_Toc520535210)

[4.8 Parameter Groups Section 48](#_Toc520535211)

[4.8.1 General 48](#_Toc520535212)

[4.8.2 Parameter Group Variables 49](#_Toc520535213)

[4.9 Parameter Data Section 52](#_Toc520535214)

[4.9.1 General 52](#_Toc520535215)

[4.9.2 First Part 52](#_Toc520535216)

[4.9.3 Second Part 55](#_Toc520535217)

[4.10 Observation Groups Section 55](#_Toc520535218)

[4.11 Observation Data Section 56](#_Toc520535219)

[4.12 Model Command Line Section 57](#_Toc520535220)

[4.13 Model Input Section 58](#_Toc520535221)

[4.14 Model Output Section 58](#_Toc520535222)

[4.15 Prior Information Section 58](#_Toc520535223)

[4.16 Regularization Section 61](#_Toc520535224)

[4.17 Control Variables for PEST++ Programs 63](#_Toc520535225)

[5. Running PEST++ Programs 67](#_Toc520535226)

[5.1 General 67](#_Toc520535227)

[5.2 Model Runs in Serial 67](#_Toc520535228)

[5.2.1 Concepts 67](#_Toc520535229)

[5.2.2 Running PESTPP-XXX 67](#_Toc520535230)

[5.3 Model Runs in Parallel 68](#_Toc520535231)

[5.3.1 Concepts 68](#_Toc520535232)

[5.3.2 Manager to Worker Communication 69](#_Toc520535233)

[5.3.3 Running PESTPP-XXX and PESTPP-WRK 69](#_Toc520535234)

[5.3.4 Worker Input File 70](#_Toc520535235)

[5.3.5 Run Management Record File 70](#_Toc520535236)

[5.3.6 Run Management Control Variables 71](#_Toc520535237)

[5.3.7 An Alternative Worker Option 72](#_Toc520535238)

[5.4 Run Book-Keeping Files 72](#_Toc520535239)

[6. PESTPP 73](#_Toc520535240)

[6.1 Introduction 73](#_Toc520535241)

[6.2 Highly Parameterized Inversion 73](#_Toc520535242)

[6.2.1 Basic Equations 73](#_Toc520535243)

[6.2.2 Choosing the Regularization Weight Factor 75](#_Toc520535244)

[6.2.3 Inter-Regularization Group Weighting 76](#_Toc520535245)

[6.2.4 Choosing Values for the Marquardt Lambda 76](#_Toc520535246)

[6.2.5 Singular Value Decomposition 77](#_Toc520535247)

[6.2.6 SVD-Assist 79](#_Toc520535248)

[6.2.7 Expediting the First Iteration 81](#_Toc520535249)

[6.2.8 Linear Analysis 82](#_Toc520535250)

[6.2.9 Model Run Failure 84](#_Toc520535251)

[6.2.10 Composite Parameter Sensitivities 84](#_Toc520535252)

[6.2.11 Other Controls 85](#_Toc520535253)

[6.2.12 Running PESTPP 85](#_Toc520535254)

[6.2.13 PESTPP Output Files 85](#_Toc520535255)

[6.3 Differential Evolution 87](#_Toc520535256)

[6.3.1 General 87](#_Toc520535257)

[6.3.2 The DE Method 87](#_Toc520535258)

[6.3.3 Using DE in PESTPP 89](#_Toc520535259)

[6.3.4 Running PESTPP 90](#_Toc520535260)

[6.3.5 PESTPP Output Files 90](#_Toc520535261)

[6.4 Summary of PESTPP Control Variables 90](#_Toc520535262)

[6.4.1 General 90](#_Toc520535263)

[6.4.2 Control Variables in the PEST Control File 90](#_Toc520535264)

[6.4.3 PEST++ Control Variables 91](#_Toc520535265)

[7. PESTPP-GSA 94](#_Toc520535266)

[7.1 Introduction 94](#_Toc520535267)

[7.1.1 General 94](#_Toc520535268)

[7.1.2 Grouped Parameters 95](#_Toc520535269)

[7.2 Method of Morris 95](#_Toc520535270)

[7.2.1 Elementary Effects 95](#_Toc520535271)

[7.2.2 Sampling Scheme 96](#_Toc520535272)

[7.2.3 Control Variables 96](#_Toc520535273)

[7.3 Method of Sobol 97](#_Toc520535274)

[7.3.1 Sensitivity Indices 97](#_Toc520535275)

[7.3.2 Control Variables 99](#_Toc520535276)

[7.4 PESTPP-GSA Output Files 99](#_Toc520535277)

[8. PESTPP-OPT 101](#_Toc520535278)

[8.1 Introduction 101](#_Toc520535279)

[8.1.1 A Publication 101](#_Toc520535280)

[8.1.2 Overview 101](#_Toc520535281)

[8.1.3 Calculation of Uncertainty 102](#_Toc520535282)

[8.1.4 Optimization 103](#_Toc520535283)

[8.1.5 Chance Constraints 105](#_Toc520535284)

[8.2 Using PESTPP-OPT 105](#_Toc520535285)

[8.2.1The PEST Control File 105](#_Toc520535286)

[8.2.2 Decision Variables and Parameters 106](#_Toc520535287)

[8.2.3 Defining the Objective Function 107](#_Toc520535288)

[8.2.4 Constraints 108](#_Toc520535289)

[8.2.5 Observations 109](#_Toc520535290)

[8.2.6 Regularization 110](#_Toc520535291)

[8.2.7 Prior Covariance Matrix 110](#_Toc520535292)

[8.2.8 Risk 110](#_Toc520535293)

[8.2.9 Jacobian and Response Matrices 111](#_Toc520535294)

[8.2.10 Solution Convergence 112](#_Toc520535295)

[8.2.11 Other Control Variables 113](#_Toc520535296)

[8.2.12 Final Model Run 113](#_Toc520535297)

[8.2.13 Restarts 113](#_Toc520535298)

[8.3 PESTPP-OPT Output Files 113](#_Toc520535299)

[8.4 Summary of Control Variables 114](#_Toc520535300)

[9. PESTPP-IES 116](#_Toc520535301)

[9.1 Introduction 116](#_Toc520535302)

[9.1.1 Publications 116](#_Toc520535303)

[9.1.2 Overview 116](#_Toc520535304)

[9.1.3 Ensemble Kalman Filters and Ensemble Smoothers 118](#_Toc520535305)

[9.1.4 Some Repercussions of Using Ensembles 119](#_Toc520535306)

[9.1.5 Iterations 120](#_Toc520535307)

[9.1.6 Measurement Noise 120](#_Toc520535308)

[9.1.7 Regularization 121](#_Toc520535309)

[9.1.8 Base Realization 122](#_Toc520535310)

[9.1.9 Parameter Transformation Status 122](#_Toc520535311)

[9.1.10 Comparative Observations 123](#_Toc520535312)

[9.1.11 Localization 123](#_Toc520535313)

[9.2 Using PESTPP-IES 124](#_Toc520535314)

[9.2.1 General 124](#_Toc520535315)

[9.2.2 Initial Realizations 125](#_Toc520535316)

[9.2.3 “Regularization” 127](#_Toc520535317)

[9.2.4 Prior Parameter Scaling 129](#_Toc520535318)

[9.2.5 The Marquardt Lambda 130](#_Toc520535319)

[9.2.6 Restarting 131](#_Toc520535320)

[9.2.7 Failed Model Runs 132](#_Toc520535321)

[9.2.8 Reporting 133](#_Toc520535322)

[9.2.9 Termination Criteria 133](#_Toc520535323)

[9.3 PESTPP-IES Output Files 134](#_Toc520535324)

[9.3.1 CSV Output Files 134](#_Toc520535325)

[9.3.2 Non-CSV Output Files 135](#_Toc520535326)

[9.4 Summary of Control Variables 135](#_Toc520535327)

[10. PESTPP-SWP 138](#_Toc520535328)

[10.1 Introduction 138](#_Toc520535329)

[10.2 Using PESTPP-SWP 138](#_Toc520535330)

[10.3 Summary of Control Variables 139](#_Toc520535331)

[11. References 140](#_Toc520535332)

[Appendix A. PEST Control File Specifications 143](#_Toc520535333)

[Appendix B. Some File Formats 160](#_Toc520535334)

[B.1 Introduction 160](#_Toc520535335)

[B.2 Matrix File 160](#_Toc520535336)

[B.2.1 General 160](#_Toc520535337)

[B.2.2 Specifications 160](#_Toc520535338)

[B.3 Uncertainty Files 161](#_Toc520535339)

[B.3.1 Introduction 161](#_Toc520535340)

[B.3.2 Specifications 161](#_Toc520535341)

# 1. Introduction

## 1.1 PEST++ and PEST

The name “PEST++” refers to a suite of programs which have some things in common but which are also very different from each other. The original “PEST++” program is part of the PEST++ suite; it is now named PESTPP. PEST++ inherits its name from the PEST package. PEST stands for “Parameter ESTimation”.

PEST was released in 1995; it has been continually improved since then. It undertakes highly parameterized inversion of environmental models. In doing so, it runs a model many times, either sequentially or in parallel. It does this in a non-intrusive manner. Before it runs a model, it records parameter values that it wishes the model to use on that particular run on input files required by the model. User-prepared template files of model input files guide it in this task. After the model run is complete, PEST reads numbers from model output files which it then compares with field measurements. User-prepared instruction files guide it in this task. All other information which PEST requires is recorded in a PEST control file.

Like PEST, PESTPP undertakes non-intrusive, highly parameterized inversion of an environmental model. As such, it can be considered as a direct replacement for PEST. While omitting some PEST functionality, it includes significant functionality that is absent from PEST. It uses the same template and instruction files that PEST uses to interact with a model. It reads a PEST control file to acquire information on problem definition. Like PEST, it can conduct model runs in serial or in parallel.

Parallelization of model runs follows the “manager” and “worker” concept. A user must start up workers in all folders in which he/she wishes that model runs be carried out. Workers can run on the same computer as the manager, on network-connected personal computers, on nodes of a high performance computing cluster, and on the cloud. Workers communicate with their manager using the TCP/IP protocol. If the manager’s machine can be “pinged” from the worker’s machine, and if a worker’s machine can be “pinged” from the manager’s machine, then communication channels are open for parallelization of model runs. Whenever PESTPP requires that a model run be undertaken, its parallel run manager chooses a worker for the task, and then sends to the worker the numbers that must be written to model input files. Using template files, the worker writes these numbers to model input files, and it commands its local operating system to run the model. When the model run is complete, it reads model output files using instruction files. It then sends the numbers which it reads from these files back to the manager using TCP/IP.

One of the original design specifications of PESTPP was that its parallel run manager be modular, and separate from PESTPP, so that it could be used by any program that conducts non-intrusive, parallelized model runs for any purpose. It is this facet of its design that has spawned the development of other members of the PEST++ suite. At the time of writing, the composition of the PEST++ suite is listed in table 1.1.

|  |  |
| --- | --- |
| **Program Name** | **Function** |
| PESTPP | Highly parameterized inversion, and global optimization using differential evolution |
| PESTPP-GSA | Global sensitivity analysis using the methods of Morris and Saltelli |
| PESTPP-OPT | Decision optimization under uncertainty using sequential linear programming and linearized chance constraints |
| PESTPP-IES | Iterative ensemble smoother for production of a suite of calibration-constrained parameter fields |
| PESTPP-SWP | Undertakes a suite of parallelized model runs for any reason |
| PESTPP-WKR | Universal worker |

Table 1.1 Programs comprising version 4 of the PEST++ suite.

The parallel run manager used by version 4 of PEST++ is named PANTHER. It is available as a public domain library for static linkage to other codes written in a number of languages. Source code (in C++) is also available. It is hoped that its free availability will spawn the development of other software that can add value to environmental models through processing the outcomes of many model runs. The PANTHER run manager is described in a companion manual.

As well as their usage of the PANTHER run manager (and hence of template and instruction files as a mechanism for non-intrusive communication with a model), the programs listed in table 1.1 have other commonalities. They all manipulate model parameters (which sometimes play the roles of decision variables). Most of them quantify mismatch between model outputs and field data. Information on parameters and field data is obtained from a PEST control file. So too is information that controls the way in which their algorithms operate.

Use of a PEST control file for storage of parameter, observation and control data was an important design consideration for the original version of PESTPP. This allowed interchangeable use with PEST. This remains a consideration for many members of the PEST++ suite. However it is not an essential element of the PANTHER run manager. Hence future versions of the PEST++ suite may dispense with the necessity to read a PEST control file. So too can third-party software developers who use the PANTHER run manager in their own programs.

Most programs of the PEST++ suite that read a PEST control file obtain the values of control variables that are unique to that program from lines within the PEST control file that begin with the “++” string. PEST, and most of its associated utility software, ignores these lines. Interoperability of the PEST and PEST++ suites is thereby maintained.

If salient to the tasks which they implement, members of the PEST++ suite write two other file types that are compatible with PEST-suite files that hold the same information. These are parameter value files (i.e. PAR files) and Jacobian matrix files (i.e. JCO files – also referred to as JCB files in PEST++ parlance). This, together with compatibility of the PEST control file, allows utilities which support the use of PEST to perform the same roles for members of the PEST++ suite. See Doherty et al (2018) for an overview of PEST utility support software.

The release of version 4 of PEST++ coincides with the release of version 15 of PEST. Enhancements introduced to version 15 of PEST include those required to promulgate inter-operability with the PEST++ suite. This inter-operability provides modelers with access to a broader range of model-value-adding functionality than that which is provided by either suite alone.

## 1.2 Software Installation

Copy PEST++ executable programs to a suitable folder on your hard disk. Then make sure that this folder is cited in the PATH environment variable, so that your computer can find them regardless of your current working folder.

Source code, Visual Studio solution files and UNIX makefiles for PEST++ suite programs are also freely available.

## 1.3 This Document

This document constitutes a manual for programs of the PEST++ suite. As such, it records their capabilities and lists variables that control their operation. However, while it discusses the algorithms which they implement to some extent, it does not describe them in detail. Instead, reference is made to publications where these explanations are made. These texts thus constitute recommended reading for those who wish to take full advantage of PEST++ software. Another relevant publication is the “PEST Book” (Doherty, 2015). This provides an extensive theoretical overview of model calibration and linear/nonlinear predictive uncertainty analysis. It also discusses the role that modeling can play in environmental decision support.

To avoid excessive cross-referencing, parts of the PEST manual are reproduced in this document. This allows a user of the PEST++ suite to dispense with the need to read the manual for PEST in addition to the present text. However the reader’s attention is drawn to a companion document (Doherty et al, 2018) which describes the relationship between the PEST and PEST++ suites. Perusal of this document may suggest ways in which PEST utility support software can assist setup and usage of PEST++ programs.

PEST++ users should also be aware of setup and processing assistance provided by the Python PyEMU library. This is discussed by White et al (2016).

## 1.4 A Model: Some Considerations

### 1.4.1 Running a Model

In the course of estimating its parameters, optimizing its decision variables, and/or computing sensitivities of model outputs to its parameters, members of the PEST++ suite must run a model many times. They can do this directly, or they can run the model through the PANTHER run manager. In either case, the model is run by submitting its command line to the operating system through a so-called “system call”. This call has the same effect as typing the name of the model (and any arguments required by the model which follow its name) in a command line window. Hence the model must be accessible to a user (and therefore to the pertinent PEST++ program) through the command line. Ideally, the folder (i.e. the directory) in which the model executable resides should be featured in the PATH environment variable so that the operating system knows where to find it.

Some models are not accessible to a user in this way because they can only be run through their own model-specific, graphical user interface; they may be called as a dynamic linked library (i.e. DLL) by this interface. Models which can only be accessed in this way cannot be used with members of the PEST++ suite unless a “driver” program is available (or written) that can call the DLL.

Nevertheless, there is a considerable amount of flexibility in what constitutes “a model” that can be run by programs of the PEST++ suite. A model can be a batch script (on a WINDOWS system) a shell script (on a UNIX system), or a python script (on either system). These scripts can be used to run one or a number of executable programs in succession. Hence, for example, a numerical simulator can be accompanied by pre-processors and postprocessors. The former may manipulate parameters (for example, it may interpolate the values of pilot point parameters to the cells of a model grid), while the latter may manipulate model outputs (for example, it may interpolate model outputs from grid cell centres to the locations of observation points). Alternatively, or as well, the model may actually be comprised of a number of simulators which are run in succession (for example, a flow model followed by a contaminant transport model).

Where simulators and pre/postprocessors are accessed through a script, it is a good idea to commence such a script with commands to delete all files which are written by one executable program and then read by another. Thus if the first executable program fails to run, the second executable program will not read an old file generated by the first executable program, mistaking it for a new one. It will therefore crash, as will all succeeding executable programs. Some or all of the files which must be read by the PEST++ program which commissioned the model run will then be absent; members of the PEST++ suite delete these files before they run the model. The PEST++ program will then cease execution gracefully with an appropriate error message.

Sometimes, in the WINDOWS environment, a model whose execution is initiated from the command line, or through a system call, returns control to the calling program (or to the user prompt) immediately, even though the model is still running. This would cause the PEST++ program which instigated the model run to look for the model’s output files, thinking that its execution was finished. This can be prevented by using the “start /w” command. Type

start /?

at the command line prompt for “start” command details.

If, on commencement of execution, a model prompts the user for keyboard input, this situation can be easily accommodated through input redirection in the batch or shell script environment. The keyboard responses to a model’s prompts can be placed in a text file. Suppose that this file is named *model.inp*. Suppose also that the name of the model executable file is *model.exe*, or simply *model* on a UNIX platform. Then if the model is run using the command

model < model.inp

the model will look to file *model.inp*, rather than to the keyboard, for its input. A PEST++ program can therefore run the model without the need for any user involvement.

### 1.4.2 Model Input and Output Files

As already discussed, programs of the PEST++ suite interact with a model through the model’s own input and output files; these constitute their non-intrusive interface with the model. A model can have many input files and many output files. PEST++ programs can interact with all of these; the number of input and output files does not matter.

A PEST++ user must prepare so-called “template files”, based on model input files, to allow programs of the PEST++ suite to recognize those parts of a model input file which it must change before running the model. Alterations to a model’s input file are only required for the purpose of providing the model with a set of parameter values which are appropriate for a particular model run. A thus-altered model input file must be an ASCII file; it cannot be a binary file. Hence even if a particular model reads much of its input dataset from one or more binary files, the file or files which contain parameter values must be ASCII (i.e. text) files.

Similar considerations apply to model output files. Programs of the PEST++ suite read numbers from model output files using directives contained in so-called “instruction files”. The files from which these numbers are read must be ASCII files. If the model writes its outputs to a binary file, then a postprocessor which follows the model in a batch or script file must be used to rewrite pertinent model outputs in ASCII format.

Template and instruction files are described in detail in the next chapter.

### 1.4.3 Differentiability of Model Outputs

Tasks performed by some (but not all) members of the PEST++ suite include regularised inversion, linear parameter and predictive uncertainty analysis, and nonlinear optimization through sequential linear programming. These methods are powerful. Furthermore, they can be used to manipulate large numbers of parameters and/or decision variables. Their power rests on an ability to employ partial derivatives of model outputs with respect to adjustable parameters or decision variables, for the purpose of estimating these parameters or optimizing these decision variables. PEST++ programs calculate these derivatives using finite differences; a PEST++ program varies a parameter or decision variable incrementally, and then divides the incremental change in pertinent model outputs by the incremental change in the parameter or decision variable to approximate the partial derivative.

If numbers calculated by a model are not continuous functions of the model’s parameters or decision variables this process fails. Non-continuity of model outputs with respect to parameters/decision variables can be an outcome of poor model design, or it can be an inadvertent outcome of numerical difficulties that a model may experience – particularly if it employs an iterative scheme to solve the equations which describe the processes which it simulates.

Often simple steps can (and should) be taken to make use of a model more tractable when it is deployed in conjunction with those members of the PEST++ suite which require continuity of model outputs with respect to adjustable parameters or decision variables. Model-calculated numbers which PEST++ programs read should be recorded on model output files with as much numerical precision as the model allows (seven significant figures if the model uses single precision arithmetic). When a PEST++ program computes a derivative using finite differences, it must subtract numbers which are very similar. Leading significant figures are lost in this process, leaving only the trailing significant figures to carry the value of the derivative. If these are absent from a number, then finite-difference derivatives lose their integrity.

Similarly, convergence criteria of iterative solvers employed by a model should be set tightly. The loss of significant figures incurred by loose convergence matters little in many modeling contexts. However this can inflict a heavy cost on finite-difference derivatives. Sometimes it may be necessary to limit the use of model numerical devices such as adaptive time stepping if it is felt that they may unwittingly contribute alterations to model outputs that do not arise exclusively from alterations to the values of model parameters or decision variables.

If a PEST++ program which relies on partial derivatives fails to perform well, this may indicate that the integrity of these derivatives is questionable. This can be checked using the JACTEST utility provided with the PEST suite. If derivatives fail the test, then there may be no alternative but to attempt the same task using another methodology that does not require derivatives of model outputs with respect to parameters/decision variables. Both the PEST++ and PEST suites provide global optimizers. The PEST++ suite also provides the powerful PESTPP-IES iterative ensemble smoother whose performance is not hampered by poor numerical derivatives.

### 1.4.4 Observations and Predictions

The numbers that programs of the PEST++ suite read from model output files often (but not always) have measured real-world counterparts. Collectively these numbers comprise a calibration dataset. Sometimes, however, numbers of interest that are read from model output files may comprise predictions for which there are no measured counterparts.

In PEST and PESTPP parlance, any number read from a model output file is referred to as “an observation”. “Measured” counterparts to these numbers are provided in the PEST control file. If a measurement is given a weight of zero, it is not actually compared with a model output; the corresponding model output is thus read for its own sake. Hence some “observations” that are provided in a PEST control file can actually be model predictions. As such, they can feature in linear and nonlinear predictive uncertainty analysis, and/or can be used by pertinent PEST utilities and PyEMU functions for purposes such as data worth analysis.

# 2. The PEST(++) Model Interface

## 2.1 Introduction

This chapter reproduces material from the seventh edition of the PEST manual. This reflects the fact that programs of both the PEST and PEST++ suites employ template and instruction files as the basis for their non-intrusive model interface.

## 2.2 PEST++ Input Files

Programs of both the PEST and PEST++ suites require three types of input file. These are:

1. template files, one for each model input file in which parameters or decision variables reside;
2. instruction files, one for each model output file from which numbers must be read; and
3. a control file, supplying the PEST or PEST++ program with the names of all template and instruction files, the names of corresponding model input and output files, the values of control variables, initial parameter values, measurement values, weights, etc.

This chapter describes the first two of these file types in detail; the PEST control file is discussed later in this manual Template files and instruction files can be written using a general-purpose text editor following specifications set out in this chapter. Alternatively, they can be written using special-purpose software that is specific to a particular modeling application. Once built, they can be checked for correctness and consistency using the utility programs TEMPCHEK, INSCHEK and PESTCHEK supplied with the PEST suite. These are documented in part II of the PEST manual (Doherty, 2018b).

In order to make the remainder of this chapter easier to read, the word “parameter” will be used to denote both a parameter and a decision variable. The word “observation” will be used to denote a number that is read from a model output file. A field measurement may correspond to this number; in this case the number read from the model output file is the model-generated counterpart of this measurement. In other cases the number extracted from a model output file may be a model prediction to which a constraint is applied (by PESTPP-OPT), a model prediction for which sensitivities are required (by PESTPP, PEST utilities or PyEMU), or simply a model output of interest.

## 2.3 Template Files

### 2.3.1 Model Input Files

Whenever a PEST++ program runs a model, as it must do many times in the course of carrying out the parameter estimation, uncertainty analysis, or optimization process for which it was designed, it must first write the values of parameters to the model input files which hold them. The model can thus access these values as it would on any other occasion that it is run.

A model may read many input files; however a template is needed only for those input files which contain parameters.

Programs of the PEST++ suite can only write parameters to ASCII (i.e. text) input files. If a model requires a binary input file, you must write a program which translates data written to an ASCII file to the binary form expected by the model. The translator program, and then the model, can be run in sequence by listing them in a batch or script file which the PEST++ program runs as “the model”. The ASCII input file to the translator program will then become a model input file as far as PEST++ programs are concerned; a template is required for this file.

It is suggested that template files be provided with the extension *.tpl* in order to distinguish them from other types of file.

### 2.3.2 An Example

A template file receives its name from the fact that it is simply a replica of a model input file, except that the space occupied by each parameter in the latter file is replaced by a sequence of characters which identify the space as belonging to that parameter.

Consider the model input file shown in figure 2.1; this file supplies data to a program which computes the “apparent resistivity” on the surface of a layered half-space for different surface electrode configurations. Suppose that we wish to use this program (i.e. model) in an inversion process through which properties of each of three half-space layers are estimated from apparent resistivity data collected on the surface of the half-space. The parameters for which we want estimates are the resistivity and thickness of the upper two layers and the resistivity of the third (its thickness is infinite). A suitable template file appears in figure 2.2.

|  |
| --- |
| MODEL INPUT FILE  3, 19 no. of layers, no. of spacings  1.0, 1.0 resistivity, thickness: layer 1  40.0, 20.0 resistivity, thickness: layer 2  5.0 resistivity: layer 3  1.0 electrode spacings  1.47  2.15  3.16  4.64  6.81  10.0  14.9  21.5  31.6  46.4  68.1  100  149  215  316  464  681  1000 |

Figure 2.1 A model input file.

|  |
| --- |
| ptf #  MODEL INPUT FILE  3, 19 no. of layers, no. of spacings  #res1 #,#t1 # resistivity, thickness: layer 1  #res2 #,#t2 # resistivity, thickness: layer 2  #res3 # resistivity: layer 3  1.0 electrode spacings  1.47  2.15  3.16  4.64  6.81  10.0  14.9  21.5  31.6  46.4  68.1  100  149  215  316  464  681  1000 |

Figure 2.2 A template file.

### 2.3.3 The Parameter Delimiter

As figure 2.2 shows, the first line of a template file must contain the letters “ptf” followed by a space, followed by a single character (“ptf” stands for “PEST template file”). The character following the space is the “parameter delimiter”. In a template file, a “parameter space” is identified as the set of characters between and including a pair of parameter delimiters. When a PEST or PEST++ program writes a model input file based on a template file, it replaces all characters between and including these parameter delimiters by a number representing the current value of the parameter that owns the space; that parameter is identified by name within the parameter space, between the parameter delimiters.

You must choose the parameter delimiter yourself. However your choice is restricted; the characters [a-z], [A-Z] and [0-9] are invalid. *The parameter delimiter character must appear nowhere within the template file except in its capacity as a parameter delimiter*, for whenever a PEST++ program encounters that character in a template file it assumes that it is defining a parameter space.

### 2.3.4 Parameter Names

All parameters are referenced by name. Parameter references are required both in template files (where the locations of parameters on model input files are identified), and in the PEST control file (where parameter initial values, lower and upper bounds and other information are provided). Parameter names can be from one to twelve characters in length, any characters being legal except for the space character and the parameter delimiter character. Parameter names are case-insensitive.

Each parameter space is defined by two parameter delimiters; the name of the parameter to which the space belongs must be written between the two delimiters.

If a model input file is such that the space available for writing a certain parameter is limited, the parameter name may need to be considerably less than twelve characters long in order that both the name and the left and right delimiters can be written within the limited space available. The minimum allowable parameter space width is thus three characters, one character for each of the left and right delimiters, and one for the parameter name.

### 2.3.5 Setting the Parameter Space Width

In general, the wider is a parameter space (up to a certain limit - see below), the better it is, for numbers can be represented with greater precision in wider spaces than they can be in narrower spaces. However, unlike the case of model-generated observations where maximum precision is crucial to obtaining useable finite-difference derivatives (see section 3.3.5), PEST++ programs can adjust to limited precision in the representation of parameters on model input files, as long as enough precision is employed such that a parameter value can be distinguished from the value of that same parameter incremented for derivatives calculation. Hence, beyond a certain number of characters, the exact number depending on the parameter value and the size and type of parameter increment employed, extra precision is not critical. Nevertheless, it is good practice to endow parameter values with as much precision as the model is capable of reading them with, so that they can be provided to the model with the same precision with which they are calculated by programs of the PEST++ suite.

Generally, a model reads numbers from an input file in either of two ways, namely from specified fields, or as a sequence of numbers, each of which may be of any length; the latter method is often referred to as “free field” input or as “list-directed” input. If the model uses the former method, then somewhere within the model program the format (i.e. field specification) for data entry is defined for every number which must be read in this fashion.

The FORTRAN code of figure 2.3 directs a program to read five real numbers. The first three are read using a format specifier, whereas the last two are read in free field fashion.

|  |
| --- |
| READ(20,100) A,B,C  100 FORMAT(3F10.0)  READ(20,\*) D,E |

Figure 2.3 Formatted and free field input.

The relevant part of the model input file may be as illustrated in figure 2.4.

|  |
| --- |
| 6.32 1.42E-05123.456789  34.567, 1.2E17 |

Figure 2.4 Numbers read using the code of figure 2.3.

Notice how no whitespace or comma is needed between numbers which are read using a field specifier. The format statement labelled “100” in figure 2.3 directs that variable *A* be read from the first 10 positions on the line, that variable *B* be read from the next 10 positions, and that variable *C* be read from the 10 positions thereafter. When the program reads any of these numbers it is unconcerned as to what characters lie outside of the field on which its attention is currently focussed. However the numbers to be read into variables *D* and *E* must be separated by whitespace or a comma in order that the program knows where one number ends and the next number begins.

Suppose all of variables *A* to *E* are model parameters, and that a PEST++ program has been assigned the task of estimating them. For convenience we provide the same names for these parameters as are used by the model code (this, of course, will not normally be the case). The template fragment corresponding to figure 2.4 may then be as set out in figure 2.5. Notice how the parameter space for each of parameters *A*, *B* and *C* is 10 characters wide, and that the parameter spaces abut each other in accordance with the expectations of the model as defined through the format specifier of figure 2.3. If the parameter space for any of these parameters is greater than 10 characters in width, then the PEST++ program, when it replaces each parameter space by the current parameter value, would construct a model input file which would be incorrectly read by the model. (You could have designed parameter spaces to be less than 10 characters wide if you wished, as long as you placed enough whitespace between each parameter space in order that the number which will replace each such space when the PEST++ program writes the model input file falls within the field expected by the model. However, defining the parameter spaces in this way would achieve nothing, as there would be no advantage in using less than the full 10 characters allowed by the model.)

|  |
| --- |
| # A ## B ## C #  # D #, # E # |

Figure 2.5 Fragment of a template file corresponding to parameters represented in figure 2.4.

Parameters *D* and *E* are treated very differently to parameters *A*, *B* and *C*. As figure 2.3 shows, the model simply expects two numbers in succession. If the spaces for parameters *D* and *E* appearing in figure 2.5 are replaced by two numbers (each will be 13 characters long) the model’s requirement for two numbers in succession separated by whitespace or a comma will have been satisfied, as will the preference for maximum precision.

Comparing figures 2.4 and 2.5, it is obvious that the spaces for parameters *D* and *E* on the template file are greater than the spaces occupied by the corresponding numbers on the model input file from which the template file was constructed; the same applies for the parameter spaces defined in figure 2.2 pertaining to the model input file of figure 2.1. In most cases of template file construction, a model input file will be used as the starting point. In such a file, numbers read using free field input will often be written with trailing zeros omitted. In constructing the template file, you should recognise which numbers are read using free field input and expand the parameter space (to the right) accordingly beyond the original number, making sure to leave whitespace or a comma between successive spaces, or between a parameter space and a neighbouring character or number.

Similarly, numbers read through field-specifying format statements may not occupy the full field width in a model input file from which a template file is being constructed (e.g. variable *A* in figure 2.4). In such a case you should, again, expand the parameter space beyond the extent of the number (normally to the left of the number only) until the space coincides with the field defined in the format specifier with which the model reads the number. (If you are not sure of this field because the model manual does not inform you of it, or you do not have the model’s source code, you will often, nevertheless, be able to make a pretty good guess as to what the field width is. As long as the parameter space you define does not transgress the bounds of the format-specified field, and the space is wide enough to allow discrimination between a parameter value and an incrementally-varied parameter value, this is good enough.)

### 2.3.6 How a Parameter Space is Filled with a Number

Programs of the PEST++ suite write as many significant figures to a parameter space as they can. They do this so that even if a parameter space must be small in order to satisfy the input field requirements of a model, there is still every chance that a parameter value can be distinguished from its incrementally-varied counterpart so as to allow proper derivatives calculation with respect to that parameter. Also, as has already been discussed, even though PEST++ programs adjust their internal representation of a parameter value to the precision with which the model can read it so that the PEST++ program and the model are using the same number, in general more precision is better.

PEST provides two control variables, PRECIS and DPOINT, that affect the manner in which numbers fill a parameter space. These can optimize representation of numbers that are written to parameter spaces that are very broad or very narrow. Programs of the PEST++ suite do not use these control variables. Instead, the PEST++ suite exclusively uses “double” PRECIS to fill template files and DPOINT set to “point”. Furthermore, as PEST++ supports parameter (and observation) names up to 200 characters, additional “filling” of template parameter spaces maybe required so that the string representation of the numeric parameter values fully occupy the correct location in resulting model input files. This is accomplished by padding the parameter string representations with leading zeros (e.g. 000001.2345678901234e+001). Testing has indicated that most model codes, as well as pre- and post-processing codes (such as python) tolerate this padding, which is needed to support model interaction with some older legacy codes that require fixed-format input.

### 2.3.7 Multiple Occurrences of the Same Parameter

The same parameter can appear multiple times on the same template file. It can also appear on multiple template files that are used to transfer numbers to multiple model input files. Where a parameter appears multiple times, and the width of the parameter space differs between these multiple occurrences, the representation of the parameter will nevertheless be the same as it is written to each parameter space. This representation will accord with the shortest of the parameter spaces.

Programs of the PEST++ suite require that each parameter cited in a PEST control file be cited on one or more template files. Conversely, they require that any parameter that is cited in a template file be cited in the PEST control file.

### 2.3.8 Preparing a Template File

Preparation of a template file is a simple procedure. For most models it can be done in a matter of moments using a text editor to replace parameter values on a typical model input file by their respective parameter space identifiers.

Once a template file has been prepared, it can be checked for correctness using the utility program TEMPCHEK supplied with PEST; see part II of the PEST manual. TEMPCHEK also has the ability to write a model input file on the basis of a template file and a user-supplied list of parameter values. If you then run your model, providing it with such a TEMPCHEK-prepared input file, you can verify that the model will have no difficulties in reading input files written by a program from the PEST++ suite.

## 2.4 Instruction Files

Of the possibly voluminous amounts of information that a model may write to its output file(s), PEST++ programs are interested in only a few numbers. As has already been discussed, these can be numbers for which corresponding field or laboratory data are available. Alternatively, they can be model predictions of particular interest, model-calculated values to which optimization constraints must be applied, model outputs for which sensitivities with respect to parameters are required, or simply model outputs whose values you need to know. These particular model-generated numbers are referred to as “observations” or “model-generated observations” in the discussion which follows.

For every model output file containing observations, you must provide an instruction file containing the directions which PEST++ programs must follow in order to read that file.

Some models write some or all of their output data to the terminal. You can redirect this screen output to a file using the “>” symbol. You can teach a PEST++ program how to read this file using a matching instruction file in the usual manner.

It is suggested that instruction files be provided with the extension *.ins* in order to distinguish them from other types of file.

### 2.4.1 Precision in Model Output Files

If there are any control variables which allow you to vary the precision with which a model’s output data are written, these should be adjusted such that model outputs which are read by PEST++ programs are recorded with maximum available precision. This applies particularly to members of the PEST++ suite which calculate finite difference derivatives, notably PESTPP and PESTPP-OPT. Unlike parameter values, for which precision is important but not essential, precision in the representation of model-generated observations is crucial for calculation of numerical derivatives. These are calculated by subtracting model-generated numbers of similar magnitude from each other. Subtraction involves loss of precision if differences are small. Unless the numbers which are read from model output files are represented with maximum precision on those files, loss of precision incurred through subtraction may be sufficient to invalidate thus-calculated derivatives. The algorithm which uses these derivatives may fail as a result.

### 2.4.2 How Model Output Files are Read

A program of the PEST++ suite must be taught how to read a model output file, and how to identify the numbers that it must extract from that file. For this to happen, model output files read by PEST++ programs must be text files; they cannot be binary files. If your model produces only binary files, you must write a simple program which reads this binary data and rewrites it in ASCII form; PEST++ programs can then search the ASCII file for the numbers they need.

Unfortunately, numbers cannot be read from model output files using the template concept. This is because many models cannot be relied upon to produce an output file of identical structure on each model run. So instead of using an output file template, you must provide programs of the PEST++ suite with a list of instructions on how to find observations on an output file. Basically, programs which use instruction files find observations on a model output file in the same way that a person does. A person runs his/her eye down the file looking for something which he/she recognises - a “marker”; if this marker is properly selected, observations can usually be linked to it in a simple manner. For example, if you are looking for the outcome of a model’s deliberations at an elapsed time of 100 days, you may instruct a PEST++ program to read the model’s output file looking for the following marker:

STRESS CALCULATED AT FINITE ELEMENT NODES: ELAPSED TIME = 100 DAYS

A particular model output which you must read may then be found, for example, between character positions 23 and 30 on the 4th line following the above marker, or as the 5th item on the 3rd line after the marker, etc. Note that for simple models, especially “home-made”, single-purpose models where little development time has been invested in highly descriptive output files, no markers may be necessary, the default initial marker being the top of the file.

Markers can be of either primary or secondary type. PEST++ programs use a primary marker as they scan the model output file line by line, looking for a reference point for subsequent observation identification or further scanning. A secondary marker is used as a reference point as a single line is examined from left to right.

### 2.4.3 An Example Instruction File

Figure 2.6 shows an output file written by the model whose input file appears in figure 2.1. Suppose that we wish to estimate the parameters appearing in the template file of figure 2.2 (i.e. the resistivities of the three half-space layers and the thicknesses of the upper two) by comparing apparent resistivities generated by the model with a set of apparent resistivities provided by field measurements. Then we need to provide instructions which teach PEST++ programs how to read each of the apparent resistivities appearing in figure 2.6. An appropriate instruction file is shown in figure 2.7.

|  |
| --- |
| SCHLUMBERGER ELECTRIC SOUNDING  Apparent resistivities calculated using the linear filter method  electrode spacing apparent resistivity  1.00 1.21072  1.47 1.51313  2.15 2.07536  3.16 2.95097  4.64 4.19023  6.81 5.87513  10.0 8.08115  14.7 10.8029  21.5 13.8229  31.6 16.5158  46.4 17.7689  68.1 16.4943  100. 12.8532  147. 8.79979  215. 6.30746  316. 5.40524  464. 5.15234  681. 5.06595  1000. 5.02980 |

Figure 2.6 A model output file.

|  |
| --- |
| pif @  @electrode@  l1 [ar1]21:27  l1 [ar2]21:27  l1 [ar3]21:27  l1 [ar4]21:27  l1 [ar5]21:27  l1 [ar6]21:27  l1 [ar7]21:27  l1 [ar8]21:27  l1 [ar9]21:27  l1 [ar10]21:27  l1 [ar11]21:27  l1 [ar12]21:27  l1 [ar13]21:27  l1 [ar14]21:27  l1 [ar15]21:27  l1 [ar16]21:27  l1 [ar17]21:27  l1 [ar18]21:27  l1 [ar19]21:27 |

Figure 2.7 A PEST instruction file.

### 2.4.4 The Marker Delimiter

The first line of an instruction file must begin with the three letters “pif” which stand for “PEST instruction file”. Then, after a single space, must follow a single character, the marker delimiter. The role of the marker delimiter in an instruction file is not unlike that of the parameter delimiter in a template file. Its role is to define the extent of a marker; a marker delimiter must be placed just before the first character of a text string comprising a marker and immediately after the last character of the marker string. In treating the text between a pair of marker delimiters as a marker, PEST++ programs do not try to interpret this text as an instruction.

You can choose the marker delimiter character yourself; however your choice is limited. A marker delimiter must not be one of the characters A - Z, a - z, 0 - 9, !, [, ], (, ), :, the space or tab characters, or &; the choice of any of these characters may result in confusion, as they may occur elsewhere in an instruction file in a role other than that of marker delimiter. Note that the character you choose as the marker delimiter should not occur within the text of any markers as this, too, will cause confusion.

### 2.4.5 Observation Names

In the same way that each parameter must have a unique name, so too must each observation be provided with a unique name. Observation names must be twenty characters or less in length. These twenty characters can be any ASCII characters except for [, ], (, ), or the marker delimiter character.

As discussed above, a parameter name can occur more than once within a parameter template file; PEST++ programs simply replace each parameter space in which the name appears with the current value of the pertinent parameter. However the same does not apply to an observation name. Every observation is unique and must have a unique observation name. In figure 2.7, observations are named “ar1”, “ar2” etc. These same observation names must also be cited in the PEST control file where measurement values and weights are provided.

There is one observation name, however, to which these rules do not apply, namely the dummy observation name “dum”. This name can occur many times, if necessary, in an instruction file; it signifies to a PEST++ program that, although the observation is to be located as if it were a normal observation, the number corresponding to the dummy observation on the model output file is not actually matched with any laboratory or field measurement. Hence an observation named “dum” must not appear in a PEST control file. As is illustrated below, the dummy observation is simply a device for model output file navigation.

### 2.4.6 The Instruction Set

Each of the available instructions is now described in detail. When creating your own instruction files, the syntax provided for each instruction must be followed exactly. If a number of instruction items appear on a single line of an instruction file, these items must be separated from each other by at least one space. Instructions pertaining to a single line on a model output file are written on a single line of a PEST instruction file. Thus the start of a new instruction line signifies that a PEST++ program must read at least one new model output file line; just how many lines it needs to read depends on the first instruction on the new instruction line. Note, however, that if the first instruction on the new line is the character “&”, the new instruction line is simply a continuation of the old one. Like all other instruction items, the “&” character used in this context must be separated from its following instruction item by at least one space.

Programs of the PEST++ suite read a model output file in the forward (top-to-bottom) direction, looking to the instructions in the instruction file to tell it what to do next. Instructions should be written with this in mind; an instruction cannot direct a PEST++ program to “backtrack” to a previous line on the model output file. Also, because PEST++ programs process model output file lines from left to right, an instruction cannot direct this program backwards to an earlier part of a model output file line than the part of the line to which its attention is currently focussed as a result of the previous instruction.

#### Primary Marker

Unless it is a continuation of a previous line, each instruction line must begin with either of two instruction items, namely a primary marker or a line advance item. The primary marker has already been discussed briefly. It is a string of characters, bracketed at each end by a marker delimiter. If a marker is the first item on an instruction line, then it is a primary marker; if it occurs later in the line, following other instruction items, it is a secondary marker, the operation of which is discussed below.

On encountering a primary marker in an instruction file, a PEST++ program reads the model output file, line by line, searching for the string between the marker delimiter characters. When it finds the string it places its “cursor” at the last character of the string. (Note that this cursor is never actually seen by the user; it simply marks the point of current processing of the model output file.) This means that if any further instructions on the same instruction line as the primary marker require further processing of this line, that processing must pertain to parts of the model output file line following the string identified as the primary marker.

Note that if there are blank characters in a primary (or secondary) marker, exactly the same number of blank characters is expected in the matching string on the model output file.

Often, as in figure 2.7, a primary marker will be part or all of some kind of header or label; such a header or label often precedes a model’s listing of the outcomes of its calculations, and thus makes a convenient reference point from which to search for the latter. It should be noted, however, that the search for a primary marker is a time-consuming process as each line of the model output file must be individually read and scanned for the marker. Hence if the same observations are always written to the same lines of a model output file (these lines being invariant from model run to model run), you should use the line advance item in preference to a primary marker.

A primary marker may be the only item on an instruction line, or it may precede a number of other items directing further processing of the line containing the marker. In the former case the purpose of the primary marker is simply to establish a reference point for further downward movement within the model output file as set out in subsequent instruction lines.

Primary markers can provide a useful means of navigating a model output file. Consider the extract from a model output file shown in figure 2.8 (the dots replace one or a number of lines not shown in the example in order to conserve space). The instruction file extract shown in figure 2.9 provides a means to read the numbers comprising the third solution vector. Notice how the “SOLUTION VECTOR” primary marker is preceded by the “PERIOD NO. 3” primary marker. The latter marker is used purely to establish a reference point from which a search can be made for the “SOLUTION VECTOR” marker; if this reference point were not established (using either a primary marker or line advance item) the program which is perusing the file would read the solution vector pertaining to a previous time period.

|  |
| --- |
| TIME PERIOD NO. 1 --->  .  .  SOLUTION VECTOR:  1.43253 6.43235 7.44532 4.23443 91.3425 3.39872  .  .  TIME PERIOD NO. 2 --->  .  .  SOLUTION VECTOR  1.34356 7.59892 8.54195 5.32094 80.9443 5.49399  .  .  TIME PERIOD NO. 3 --->  .  .  SOLUTION VECTOR  2.09485 8.49021 9.39382 6.39920 79.9482 6.20983 |

Figure 2.8 Extract from a model output file.

|  |
| --- |
| pif \*  .  .  \*PERIOD NO. 3\*  \*SOLUTION VECTOR\*  l1 (obs1)5:10 (obs2)12:17 (obs3)21:28 (obs4)32:37 (obs5)41:45  & (obs6)50:55  .  . |

Figure 2.9 Extract from an instruction file.

#### Line Advance

The syntax for the line advance item is “l*n*” where *n* is the number of lines to advance; note that “l” is “el”, the twelfth letter of the alphabet, not “one”. The line advance item must be the first item of an instruction line; it and the primary marker are the only two instruction items which can occupy this initial spot. As was explained above, the initial item in an instruction line is always a directive to move at least one line further in perusal of a model output file (unless it is a continuation character). In contrast to the use of a primary marker, however, a PEST++ program does not need to examine the entirety of each model output file line as it advances forward in the file. It simply moves forward *n* lines, placing the processing cursor just before the beginning of this new line, this point becoming the new reference point for further processing of the model output file.

Normally a line advance item is followed by other instructions. However if the line advance item is the only item on an instruction line this does not break any syntax rules.

In figure 2.6 model-calculated apparent resistivities are written on consecutive lines. Hence before reading each observation, the program reading the file is instructed to move to the beginning of a new line using the “l1” line advance item; see figure 2.7.

If a line advance item leads the first line of a PEST instruction file, the reference point for line advance is taken as a “dummy” line just above the first line of the model output file. Thus if the first instruction line begins with “l1”, processing of the model output file begins on its first line; similarly, if the first instruction line begins with “l8”, processing of the model output file begins at its eighth line.

#### Secondary Marker

A secondary marker is a marker which does not occupy the first position of a PEST instruction line. Hence it does not direct the PEST++ program which reads the file to move downwards in the model output file (though it can be instrumental in achieving this - see below); instead, it instructs this program to move its cursor along the current model output file line until it finds the secondary marker string, and to place its cursor on the last character of that string ready for subsequent processing of that line.

Figure 2.10 shows an extract from a model output file while figure 2.11 shows the instructions necessary to read the potassium concentration from this output file. A primary marker is used to place the cursor on the line above that on which the calculated concentrations are recorded for the distance in which we are interested. Then the program which reads the file is directed to advance one line and read the number following the “K:” string in order to find an observation named “kc”; the exclamation marks surrounding “kc” will be discussed shortly.

|  |
| --- |
| .  .  DISTANCE = 20.0: CATION CONCENTRATIONS:-  Na: 3.49868E-2 Mg: 5.987638E-2 K: 9.987362E-3  .  . |

Figure 2.10 Extract from a model output file.

|  |
| --- |
| pif ~  .  .  ~DISTANCE = 20.0~  l1 ~K:~ !kc!  .  . |

Figure 2.11 Extract from an instruction file.

A useful feature of secondary marker functionality is illustrated in figures 2.12 and 2.13 which represent a model output file extract and a corresponding instruction file extract, respectively. If a particular secondary marker is preceded only by other markers (including, perhaps, one or a number of secondary markers and certainly a primary marker), and the text string corresponding to that secondary marker is not found on a model output file line on which the previous markers’ strings have been located, a PEST++ program will assume that it has not yet found the correct model output line and resume its search for a line which holds the text pertaining to all three markers. Thus the instruction “%TIME STEP 10%” will cause this program to pause on its downward journey through the model output file at the first line illustrated in figure 2.12. However, when it does not find the string “STRAIN” on the same line, it re-commences its perusal of the model output file, looking for the string “TIME STEP 10” again. Eventually it finds a line containing both the primary and secondary markers and, having done so, commences execution of the next instruction line.

|  |
| --- |
| .  .  TIME STEP 10 (13 ITERATIONS REQUIRED) STRESS --->  X = 1.05 STRESS = 4.35678E+03  X = 1.10 STRESS = 4.39532E+03  .  .  TIME STEP 10 (BACK SUBSTITUTION) STRAIN --->  X = 1.05 STRAIN = 2.56785E-03  X = 1.10 STRAIN = 2.34564E-03  .  . |

Figure 2.12 Extract from a model output file.

It is important to note that if any instruction items other than markers precede an unmatched secondary marker, it will be assumed that the mismatch is an error condition; an appropriate error message will then be generated. Note also that secondary markers may be used sequentially. For example if the STRAIN variable is always in position 2, then the pertinent line in the instruction file of figure 2.13 could be replaced by "l1 %=% %=% !str1!".  This is handy for comma-delimited output files.

|  |
| --- |
| pif %  .  .  %TIME STEP 10% %STRAIN%  l1 %STRAIN =% !str1!  l1 %STRAIN =% !str2!  .  . |

Figure 2.13 Extract from an instruction file.

#### Whitespace

The whitespace instruction is similar to the secondary marker in that it allows the user to navigate through a model output file line prior to reading a non-fixed observation (see below). It directs a PEST++ program to move its cursor forwards from its current position until it encounters the next blank character. The cursor is then moved forward again until it finds a nonblank character, with the cursor finally placed on the blank character preceding this nonblank character (i.e. on the last blank character in a sequence of blank characters) ready for the next instruction. The whitespace instruction is a simple “w”, separated from its neighbouring instructions by at least one blank space.

Consider the model output file line represented below.

MODEL OUTPUTS: 2.89988 4.487892 -4.59098 8.394843

The following instruction line directs a PEST++ program to read the fourth number on the above line.

%MODEL OUTPUTS:% w w w w !obs1!

The instruction line begins with a primary marker, allowing this program to locate the above line on the model output file. After this marker is processed, the cursor rests on the “:” character of “OUTPUTS:”, i.e. on the last character of the marker string. In response to the first whitespace instruction, a PEST++ program finds the next whitespace and then moves its cursor to the end of this whitespace, i.e. just before the “2” of the first number on the above model output file line. The second whitespace instruction moves the cursor to the blank character preceding the first “4” of the second number on the above line; processing of the third whitespace instruction results in the cursor being moved to the blank character just before the negative sign. After the fourth whitespace instruction is implemented, the cursor rests on the blank character preceding the last number; the latter can then be read as a non-fixed observation (see below).

#### Tab

The tab instruction places the cursor at a user-specified character position (i.e. column number) on the model output file line which is currently being processed. The instruction syntax is “tn” where *n* is the column number. The column number is obtained by counting character positions (including blank characters) from the left side of any line, starting at 1. Like the whitespace instruction, the tab instruction can be useful in navigating through a model output file line prior to locating and reading a non-fixed observation. For example, consider the following line from a model output file:

TIME(1): A = 1.34564E-04, TIME(2): A = 1.45654E-04, TIME(3): A = 1.54982E-04

The value of *A* at TIME(3) could be read using the instruction line:

l4 t60 %=% !a3!

Here it is assumed that the fourth line prior to the above line in the model output file was being currently processed; the marker delimiter character is assumed to be “%”. Implementation of the “t60” instruction places the cursor on the “:” following the “TIME(3)” string, for the colon is in the sixtieth character position of the above line. The PEST++ program is then directed to find the next “=” character. From there it can read the last number on the above line as a non-fixed observation (see below).

#### Fixed Observations

An observation reference can never be the first item on an instruction line; either a primary marker or line advance item must come first in order to place the processing cursor on the line on which one or more observations may lie. If there is more than one observation on a particular line of the model output file, these observations must be read from left to right, backward movement along any line being disallowed.

Observations can be identified in one of three ways. The first way is to tell the PEST++ program that a particular observation can be found between, and including, columns *n*1 and *n*2 on the model output file line on which its cursor is currently resting. This is by far the most efficient way to read an observation value because the program that is reading the file does not need to do any searching; it simply reads a number from the space identified. Observations read in this way are referred to as “fixed observations”.

Figure 2.14 shows how the numbers listed in the third solution vector of figure 2.8 can be read as fixed observations. The instruction item informing the PEST++ program how to read a fixed observation consists of two parts. The first part consists of the observation name enclosed in square brackets, while the second part consists of the first and last columns from which to read the observation. Note that no space must separate these two parts of the observation instruction; a space in an instruction file is always construed as marking the end of one instruction item and the beginning of another (unless the space lies between marker delimiters).

|  |
| --- |
| pif \*  .  .  \*PERIOD NO. 3\*  \*SOLUTION VECTOR\*  l1 [obs1]1:9 [obs2]10:18 [obs3]19:27 [obs4]28:36 [obs5]37:45  & [obs6]46:54  .  . |

Figure 2.14 Extract from an instruction file.

Reading numbers as fixed observations is useful when the model writes its output in tabular form using fixed field width specifiers. However you must be very careful when specifying the column numbers from which to read the number. The space defined by these column numbers must be wide enough to accommodate the maximum length that the number will occupy over the many model runs for which the PEST++ program will read the model’s output file; if it is not wide enough, the program which is reading the file may read only a truncated part of the number or omit a negative sign preceding the number. However the space must not be so wide that it includes part of another number; in this case a run-time error will occur and the PEST++ program which is reading the file will terminate execution with an appropriate error message.

Where a model writes its results as an array of numbers, it is not an uncommon occurrence for these numbers to abut each other. Consider, for example, the following FORTRAN code fragment.

A=1236.567

B=8495.0

C=-900.0

WRITE(10,20) A,B,C

20 FORMAT(3(F8.3))

The result is

1236.5678495.000-900.000

In this case there is no choice but to read these numbers as fixed observations. (Both of the alternative ways to read an observation require that the observation be surrounded by either whitespace or a string that is invariant from model run to model run and can thus be used as a marker.) Hence to read the above three numbers as observations *A*, *B* and *C* the following instruction line may be used.

l1 [A]1:8 [B]9:16 [C]17:24

If an instruction line contains only fixed observations there is no need for it to contain any whitespace or tabs; nor will there be any need for a secondary marker, (unless the secondary marker is being used in conjunction with a primary marker in determining which model output file line the processing cursor should settle on - see above). This is because these items are normally used for navigating through a model output file line prior to reading a non-fixed observation (see below); such navigation is not required to locate a fixed observation as its location on a model output file line is defined without ambiguity by the column numbers included within the fixed observation instruction.

#### Semi-Fixed Observations

Figure 2.9 demonstrates the use of semi-fixed observations. Semi-fixed observations are similar to fixed observations in that two numbers are provided in the pertinent instruction item, the purpose of these numbers being to locate the observation’s position by column number on the model output file. However, in contrast to fixed observations, these numbers do not locate the observation exactly. When a PEST++ program encounters a semi-fixed observation instruction, it proceeds to the first of the two nominated column numbers and then, if this column is not occupied by a non-blank character, it searches the output file line from left to right beginning at this column number, until it reaches either the second identified column or a non-blank character. If it reaches the second column before finding a non-blank character, an error condition arises. However if it finds a non-blank character, it then locates the nearest whitespace on either side of the character; in this way, it identifies one or a number of non-blank characters sandwiched between whitespace (“whitespace” includes the beginning and/or the end of the model output file line). It tries to read these characters as a number, this number being the value of the observation named in the semi-fixed observation instruction. Obviously the width of this number can be greater than the difference between the column numbers cited in the semi-fixed observation instruction.

Like a fixed observation, the instruction to read a semi-fixed observation consists of two parts, namely the observation name followed by two column numbers, the latter being separated by a colon; the column numbers must be in ascending order. However for semi-fixed observations, the observation name is enclosed in round brackets rather than square brackets. Again, there must be no space separating the two parts of the semi-fixed observation instruction.

Reading a number as a semi-fixed observation is useful if you are unsure how large the representation of the number could stretch on a model output file as its magnitude grows and/or diminishes in a series of model runs; it is also useful if you do not know whether the number is left or right justified. However you must be sure that at least part of the number will always fall between (and including) the two nominated columns and that, whenever the number is written and whatever its size, it will always be surrounded either by whitespace or by the beginning or end of the model output file line. If, when reading the model output file, a PEST++ program encounters only whitespace between (and including) the two nominated column numbers, or if it encounters non-numeric characters or two number fragments separated by whitespace, an error condition will occur and the program will terminate execution with an appropriate error message.

As for fixed observations, it is normally not necessary to have secondary markers, whitespace and tabs on the same line as a semi-fixed observation, because the column numbers provided with the semi-fixed observation instruction determine the location of the observation on the line. As always, observations must be read from left to right on any one instruction line; hence if more than one semi-fixed observation instruction is provided on a single instruction line, the column numbers pertaining to these observations must increase from left to right.

For the case illustrated in figures 2.6 and 2.7, all the fixed observations could have been read as semi-fixed observations, with the difference between the column numbers either remaining the same or being reduced to substantially smaller than that shown in figure 2.7. However it should be noted that it takes more effort for a PEST++ program to read a semi-fixed observation than it does for it to read a fixed observation as it must establish for itself the extent of the number that it must read.

After a semi-fixed observation has been read, the cursor resides at the end of the number which it has just read. Any further processing of the line must take place to the right of that position.

#### Non-Fixed Observations

Figures 2.11 and 2.13 demonstrate the use of non-fixed observations. A non-fixed observation instruction does not include any column numbers because the number which must be read is found using secondary markers and/or other navigational aids such as whitespace and tabs which precede the non-fixed observation on the instruction line.

If you do not know exactly where, on a particular model output file line, a model will write the number corresponding to a particular observation, but you do know the structure of that line, then you can use this knowledge to navigate your way to the number. In the instruction file, a non-fixed observation is represented simply by the name of the observation surrounded by exclamation marks; as usual, no spaces should separate the exclamation marks from the observation name, as spaces in an instruction file are interpreted as denoting the end of one instruction item and the beginning of another.

When a PEST++ program encounters a non-fixed observation instruction it first searches forward from its current cursor position until it finds a non-blank character; it assumes this character is the beginning of the number representing the non-fixed observation. Then the program searches forward again until it finds either a blank character, the end of the line, or the first character of a secondary marker which follows the non-fixed observation instruction in the instruction file; the PEST++ program assumes that the number representing the non-fixed observation finishes at the previous character position. In this way it identifies a string of characters which it tries to read as a number; if it is unsuccessful in reading a number because of the presence of non-numeric characters or some other problem, the PEST++ program terminates execution with a run-time error message. A run time error message also occurs if it encounters the end of a line while looking for the beginning of a non-fixed observation.

Consider the output file fragment shown in figure 2.15. The species populations at different times cannot be read as either fixed or semi-fixed observations because the numbers representing these populations cannot be guaranteed to fall within a certain range of column numbers on the model output file because “iterative adjustment” may be required in the calculation of any such population. Hence we must find our way to the number using another method; one such method is illustrated in figure 2.16.

|  |
| --- |
| .  .  SPECIES POPULATION AFTER 1 YEAR = 1.23498E5  SPECIES POPULATION AFTER 2 YEARS = 1.58374E5  SPECIES POPULATION AFTER 3 YEARS (ITERATIVE ADJUSTMENT REQUIRED)= 1.78434E5  SPECIES POPULATION AFTER 4 YEARS = 2.34563E5  .  . |

Figure 2.15 Extract from a model output file.

|  |
| --- |
| pif \*  .  .  \*SPECIES\* \*=\* !sp1!  l1 \*=\* !sp2!  l1 \*=\* !sp3!  l1 \*=\* !sp4!  .  . |

Figure 2.16 Extract from an instruction file.

A primary marker is used to move the processing cursor to the first of the lines shown in figure 2.15. Then, noting that the number representing the species population always follows a “=” character, the “=” character is used as a secondary marker. After it processes a secondary marker, the processing cursor always resides on the last character of that marker, in this case on the “=” character itself. Hence after reading the “=” character, a PEST++ program is able to process the !sp1! instruction by isolating the string “1.23498E5” in the manner described above.

After it reads the model-calculated value for observation “sp1”, the program that is reading the file moves to the next instruction line. In accordance with the “l1” instruction, it reads into its memory the next line of the model output file. It then searches for a “=” character and reads the number following this character as observation “sp2”. This procedure is then repeated for observations “sp3” and “sp4”.

Successful identification of a non-fixed observation depends on the instructions preceding it. The secondary marker, tab and whitespace instructions will be most useful in this regard, though fixed and semi-fixed observations may also precede a non-fixed observation; remember that in all these cases the PEST++ program places its cursor over the last character of the string or number it identifies on the model output file corresponding to an instruction item, before proceeding to the next instruction.

Consider the model output file line shown below as a further illustration of the use of non-fixed observations.

4.33 -20.3 23.392093 3.394382

If we are interested in the fourth of these numbers but we are unsure as to whether the numbers preceding it might not be written with greater precision in some model runs (hence pushing the number in which we are interested to the right), then we have no alternative but to read the number as a non-fixed observation. However if the previous numbers vary from model run to model run, we cannot use a secondary marker either; nor can a tab be used. Fortunately, whitespace comes to the rescue, with the following instruction line taking the program reading the file to the fourth number:

l10 w w w !obs1!

Here it is assumed that, prior to reading this instruction, the processing cursor was located on the 10th preceding line of the model output file. As long as we can be sure that no whitespace will ever precede the first number, there will always be three incidences of whitespace preceding the number in which we are interested. However, if it happens that whitespace may precede the first number on some occasions, while on other occasions it may not, then we can read the first number as a dummy observation as shown below:

l10 !dum! w w w !obs1!

As was explained previously, the number on the model output file corresponding to an observation named “dum” is not actually used; nor can the name “dum” appear in the “observation data” section of a PEST control file. The use of this name is reserved for instances like the present case where a number must be read in order to facilitate navigation along a particular line of the model output file. The number is read according to the non-fixed observation protocol, for only observations of this type can be dummy observations.

An alternative to the use of whitespace in locating the observation “obs1” in the above example could involve using the dummy observation more than once. Hence the instruction line below would also enable the number representing “obs1” to be located and read.

l10 !dum! !dum! !dum! !obs1!

However had the numbers in the above example been separated by commas instead of whitespace, the commas should have been used as secondary markers in order to find “obs1”.

A number not surrounded by whitespace can still be read as a non-fixed observation with the proper choice of secondary markers. Consider the model output file line shown below.

SOIL WATER CONTENT (NO CORRECTION)=21.345634%

It may not be possible to read the soil water content as a fixed observation because the “(NO CORRECTION)” string may or may not be present after any particular model run. Reading it as a non-fixed observation appears troublesome as the number is neither preceded nor followed by whitespace. However a suitable instruction line is

l5 \*=\* !sws! \*%\*

Notice how a secondary marker (i.e. \*%\*) is referenced even though it occurs after the observation we wish to read. If this marker were not present, a run-time error would occur when a PEST++ program tries to read the soil water content because it would define the observation string to include the “%” character and, naturally, would be unable to read a number from a string which includes non-numeric characters. However by including the “%” character as a secondary marker after the number representing the observation “sws”, the program reading the file is instructed to separate the character from the string before trying to read the number. But note that if a post-observation secondary marker of this type begins with a numerical character, a PEST++ program will include this character with the number representing the value of the observation if there is no whitespace separating it from the observation. The observation will then be read incorrectly.

The fact that there is no whitespace between the “=” character and the number we wish to read causes no problems. After processing of the “=” character as a secondary marker, the processing cursor falls on the “=” character itself. The search for the first non-blank character initiated by the !sws! instruction terminates on the very next character after “=”, i.e. the “2” character. The program reading the file then accepts this character as the left boundary of the string from which it must read the soil moisture content and searches forwards for the right boundary of the string in the usual manner.

After a PEST++ program has read a non-fixed observation, it places its cursor on the last character of the observation number. It can then undertake further processing of the model output file line to read further non-fixed, fixed or semi-fixed observations, or process navigational instructions as directed.

#### Continuation

You can break an instruction line between any two instructions by using the continuation character, “&”, to inform a PEST++ program that a certain instruction line is actually a continuation of the previous line. Thus the instruction file fragment

l1 %RESULTS% %TIME (4)% %=% !obs1! !obs2! !obs3!

is equivalent to

l1

& %RESULTS%

& %TIME (4)%

& %=%

& !obs1!

& !obs2!

& !obs3!

For both of these fragments, the marker delimiter is assumed to be “%”. Note that the continuation character must be separated from the instruction which follows it by at least one space.

### 2.4.7 Making an Instruction File

An instruction file can be built using a text editor. Alternatively it can be written by software dedicated to this purpose such as the graphical user interface of a model which supports PEST and/or PEST++. Alternatively, it can be built by members of the PEST Groundwater and Surface Water Utility suites, both which are downloadable from the PEST web pages.

Caution must always be exercised in building an instruction set to read a model output file, especially if navigational instructions such as markers, whitespace, tabs and dummy observations are used. A PEST++ program which reads a model output file will always follow your instructions to the letter, but it may not read the number that you intend it to read if you get an instruction wrong. If this program tries to read an observation but does not find a number where it expects to find one, a run-time error will occur. The program will inform you of where it encountered the error, and of the instruction it was implementing when the error occurred; this should allow you to find the problem. However if the program actually reads the wrong number from the model output file, this may only become apparent if an unusually high objective function results, or if the objective function cannot be lowered. Alternatively, if the number which the PEST++ program is instructed to read is a model prediction, or if this program is being asked purely to compute sensitivities of this number to model parameters, the error may never become apparent. If in doubt, check run record and/or other files that are written by the pertinent PEST++ program for numbers that it reads from model output files to ensure that they meet expectations.

Included in the PEST suite are two programs which can be used to verify that instruction files have been built correctly. Program PESTCHEK reads all the instruction files cited in a PEST control file, ensuring that no syntax errors are present in any of these files. Program INSCHEK, on the other hand, checks a single instruction file for syntax errors. If an instruction file is error-free, INSCHEK can then use that instruction file to read a model output file, recording a list of observation values read from that file to another file. In this way you can be sure that your instruction set “works” before it is actually used by a program from the PEST++ suite.

# 3. Some Important PEST++ Features

## 3.1 General

Before describing the PEST control file, certain features that programs of the PEST++ suite have in common are discussed. This will make the task of explaining the settings of PEST and PEST++ control variables somewhat easier.

As for the previous and ensuing chapters of this manual, parts of the present chapter are taken from version 7 of the PEST manual.

## 3.2 Parameter Adjustment

### 3.2.1 Parameter Transformation

All of the programs of the PEST++ suite adjust values ascribed to a model’s parameters (and sometimes its decision variables, though they are given the umbrella name “parameters” in the present chapter). However, behind the scenes, these programs can be asked to adjust the logarithms of parameter values instead of the parameter values themselves. This adjustment strategy often renders an inversion or optimization process more numerically stable, and faster, than it would otherwise be. It also removes large inequalities in sensitivities between parameters which may be an artefact of the units employed for their representation.

Programs of the PEST++ suite require that all parameters that are cited in the “parameter data” section of a PEST control file be designated as untransformed, log-transformed, fixed or tied; the latter two options are discussed below. If a parameter is designated as log-transformed, any prior information pertaining to that parameter must pertain to the log (to base 10) of that parameter. However, despite the fact that programs of the PEST++ suite work with the logarithms of parameters if they are user-designated as log-transformed, these programs record the actual values that are estimated for these parameters rather than the logarithms of these values.

PEST++ programs learn whether a parameter is log-transformed through a parameter-specific variable named PARTRANS. You should never ask PEST to log transform a parameter whose initial value is zero or negative, or to log transform a parameter whose lower bound is zero or less. If you attempt to do this, a PEST++ program will inform you of your mistake.

Note that more complex parameter transformations can be undertaken using the PAR2PAR parameter pre-processor provided with the PEST suite; see part II of the PEST manual.

### 3.2.2 Fixed and Tied Parameters

A parameter can be referenced in a template file yet take no part in the parameter estimation or optimization process. In this case it must be declared as “fixed” in the PEST control file so that its value does not vary from that assigned to it in that file. This is accomplished by setting the PARTRANS variable associated with that parameter to “fixed” in the “parameter data” section of the PEST control file.

Programs of the PEST++ suite allow one or more parameters to be tied (i.e. linked) to a “parent” parameter. A value is not estimated for a tied parameter. Instead, a value is estimated for the parent parameter; the ratio of tied parameter value to parent parameter value is then maintained through the inversion or optimization process. Thus tied parameters “piggyback” on their parents. Note that a parameter cannot be tied to a parameter which is either fixed or tied to another parameter itself. Parameters are tied through setting PARTRANS to “tied”. As is described in the following chapter, parent parameters of tied parameters are specified in the second part of the “parameter data” section of the PEST control file.

### 3.2.3 Upper and Lower Parameter Bounds

Upper and lower bounds, defining the maximum and minimum values that a parameter is allowed to assume during an inversion or optimization process, must be supplied for all parameters. These are provided through the PARLBND and PARUBND variables in the “parameter data” section of a PEST control file.

It is important that upper and lower parameter bounds be chosen wisely. For many models, parameters can lie only within certain well-defined domains determined by the theory on which the model is based. In such cases model-generated floating-point errors may result if a PEST++ program is not prevented from adjusting a parameter to a value outside its allowed domain. For example if, at some stage during a model run, the logarithm or square root of a particular parameter is taken, then that parameter must be prevented from ever becoming negative (or zero if the model takes the log of the parameter). If the reciprocal is taken of a parameter, that parameter must never be zero.

For some programs belonging to the PEST++ suite, parameter bounds have a significance beyond that of parameter range confinement. Bounds can be used to provide an estimate of prior parameter uncertainty. By default, this estimate is based on the assumption that the difference between a parameter’s upper and lower bound is equal to four standard deviations of its prior probability distribution. (This assumption can be altered using the PEST++ *par\_sigma\_range()* control variable, discussed in detail elsewhere in this document). If a parameter is log-transformed, the difference is calculated between the logarithms of its bounds, and the standard deviation is applied to the logarithm of parameter values.

### 3.2.4 Scale and Offset

A scale and offset (variables SCALE and OFFSET in the “parameter data” section of a PEST control file) must be supplied for all parameters. Before writing the value of a parameter to a model input file, programs of the PEST++ suite multiply the parameter’s value by the scale and add the offset.

The SCALE and OFFSET variables can be very convenient in some situations. For example, in a particular model, a certain parameter may have a non-zero base level; you may wish to re-define the parameter as the actual parameter minus this base level. Elevation may be such a parameter. If a reference elevation is subtracted from the true, model-required elevation, the result may be thickness; this may be a more “natural” parameter to estimate than elevation. In particular, it may make more sense to express a derivative increment (see below) as a fraction of thickness than as a fraction of elevation, to which an arbitrary datum has been added. Also, an inversion or optimization process may be better behaved if the thickness parameter is log-transformed. (In contrast, it would be surprising if log-transformation of elevation improved inversion or optimization performance due to the arbitrary datum with respect to which an elevation must be expressed.) PEST++ programs can thus estimate thickness, converting this thickness to elevation on each occasion that they write a model input file by adding the reference elevation stored as the parameter’s OFFSET.

The SCALE variable is equally useful. A model’s parameter may be such that it can only be ascribed negative values; such a parameter cannot be log-transformed. However if a new parameter is defined as the negative of the model-required parameter, a program of the PEST++ suite can estimate this new parameter, log-transforming it if necessary to enhance inversion efficiency. Just before it writes the parameter to a model input file, the program multiplies it by its SCALE (-1 in this case) so that the model receives the parameter it expects.

If you do not wish a parameter to be scaled and offset, provide its SCALE as 1 and its OFFSET as zero.

It should be stressed that programs of the PEST++ suite are oblivious to a parameter’s SCALE and OFFSET until the moment its value is written to a model input file. It is at this point (and only this point) that it first multiplies the value of the parameter by the SCALE and then adds the OFFSET; the SCALE and OFFSET take no other part in the parameter adjustment process. Note also that fixed and tied parameters must each be supplied with a SCALE and OFFSET, just like their adjustable (log-transformed and untransformed) counterparts.

### 3.2.5 Parameter Change Limits

When programs of the PEST++ suite adjust the value of a parameter, they respect the upper and lower bound of that parameter as recorded in a PEST control file. However this is not the only limitation that can be placed on the adjustment of a parameter. In common with PEST, the PESTPP program places a further limit on the amount by which the value of a parameter is permitted to change during any one iteration of the inversion process.

In the course of estimating parameters, PESTPP attempts to lower an objective function that defines the current level of model-to-measurement misfit. Unless the model whose parameters are being estimated exhibits outrageously nonlinear behaviour, an updated parameter set will result in a lower objective function. However if a model is highly nonlinear, the parameter upgrade vector may “overshoot” the objective function minimum, and the new value of the objective function may actually be worse than the old one. This is because the equations employed for calculation of the upgrade vector are based on a local linearity assumption which may not extend as far into parameter space from current parameter estimates as the magnitude of the upgrade vector itself.

To obviate the possibility of overshoot, it is good practice to place a reasonable limit on the maximum change that any adjustable parameter is allowed to undergo in any iteration of the inversion process that is undertaken by PESTPP. The limits imposed by PESTPP may be of two types, namely “relative”, and “factor”. (PEST allows an “absolute” limit in addition to these.) You must inform PESTPP through the parameter-specific variable PARCHGLIM (residing in the “parameter data” section of the PEST control file) which type of change limit applies to each adjustable parameter. Meanwhile, the RELPARMAX and FACPARMAX control variables provide the maximum allowed relative and factor changes for all relative-limited and factor-limited parameters respectively. These variables appear in the “control data” section of the PEST control file.

Let *f* represent the user-defined maximum allowed parameter factor change for factor-limited parameters (i.e. FACPARMAX); *f* must be greater than unity. Then if *b*0 is the value of a particular factor-limited parameter at the beginning of a PESTPP iteration, the value *b* of this same parameter at the beginning of the next iteration will lie between the limits

*b*0/*f* ≤ *b* ≤ *fb*0 (3.1a)

if *b*0 is positive, and

*fb*0 ≤ *b* ≤ *b*0/*f* (3.1b)

if *b*0 is negative.Note that if a parameter is subject to factor-limited changes, it can never change sign.

Let *r* represent the user-defined maximum allowed relative parameter change for all relative-limited parameters (i.e. RELPARMAX); *r* can be any positive number. Then if *b*0 is the value of a particular relative-limited parameter at the beginning of a PESTPP iteration, its value *b* at the beginning of the next iteration will be such that

 (3.2)

In this case, unless *r* is less than or equal to unity, a parameter can, indeed, change sign. However there may be a danger in using a relative limit for some types of parameters. If *r* is one or greater, *b* may fall to a minute fraction of *b*0 (or even to zero), without transgressing the parameter change limit. For some types of parameters in some models this will be fine; in other cases a parameter factor change of this magnitude may significantly transgress model linearity limits.

In implementing the constraints set by equations 3.1 and 3.2, PESTPP limits the magnitude of the parameter upgrade vector such that neither of these equations is violated for any parameter to which the pertinent equation applies. Naturally, if only one type of parameter change limit is featured in a particular PESTPP inversion process (for example if parameters are all factor-limited or are all relative-limited) only the pertinent one of these limits is invoked.

If, in the course of the inversion process, PESTPP assigns to a parameter a value which is very small relative to its initial value, then either of equations 3.1 or 3.2 may place an undue restriction on subsequent parameter adjustments. Thus if *b*0 for one parameter is very small, the changes to all parameters may be set intolerably small so that equation 3.1 or equation 3.2 is respected for this one parameter. To circumvent this problem, PESTPP employs another variable, FACORIG (also residing in the “control data” section of the PEST control file), to limit the effect that an unduly low parameter value can have in this regard. If the absolute value of a parameter is less than FACORIG times its initial absolute value, and if PESTPP wishes to adjust that parameter such that its absolute value will increase, then the parameter’s current value *b*0 is replaced by FACORIG times its initial value in equation 3.1 and in the denominator of equation 3.2. A suitable value for FACORIG varies from case to case; a value of 0.001 is often appropriate. Note, however, that FACORIG is not used to adjust change limits for log-transformed parameters.

Problems associated with imposing change limits on relative- or factor-limited parameters when their values become very low can sometimes be prevented by providing them with suitable OFFSET values in the “parameter data” section of the PEST control file. Naturally, their upper and lower bounds must also be appropriately offset.

## 3.3 Calculation of Derivatives

### 3.3.1 General

The ability to calculate partial derivatives of model outputs with respect to adjustable parameters and/or decision variables is fundamental to implementation of the inversion and linear uncertainty analysis methodologies implemented by PESTPP, and to management optimization under chance constraints implemented by PESTPP-OPT. Because programs of the PEST++ suite interact with a model non-intrusively, they must evaluate these derivatives themselves, using model outputs calculated on the basis of incrementally-varied parameters and/or decision variables.

Accuracy in derivatives calculation is fundamental to good performance of numerical methods that solve an inversion or optimization problem through a series of local linear approximations. Solution of the overall problem is then achieved iteratively. Methods which rely on successive local linearization are often referred to as “gradient methods”.

Reduction of the numerical integrity of finite-difference derivatives may arise from the presence of round-off errors in numbers that are recorded on model output files. It can also be an outcome of poor model numerical performance, whereby changes in model outputs following incremental changes in the value of a parameter are caused not only by alterations to the parameter’s value, but also by problematic convergence of the model’s solver, or by the presence of artificial thresholds and discontinuities in the model’s simulation algorithm. Fortunately, many algorithms which rely on finite-difference derivatives have a moderate tolerance for errors in these derivatives. However if the numerical behaviour of a model is too degraded, then the performance of programs which require finite difference derivatives may suffer to the point where they cannot be used with that model. In this case alternative estimation/optimization tools must be sought. The differential evolution process supported by PESTPP, and the randomized Jacobian process supported by PESTPP-IES provide such alternatives.

The variables which control finite-difference derivatives calculation are assigned to parameter groups rather than to individual parameters. Parameters are assigned to groups in a PEST control file. The assignment of derivative-pertinent control variables to groups, rather than to individual parameters, reduces the complexity of a PEST++ input dataset. Furthermore, in many instances, parameters naturally fall into a small number of categories. For example, if the domain of a two- or three-dimensional spatial model is parameterized using pilot points, pilot-point parameters which describe system properties of the same type would normally be assigned to the same group. However, if you wish to treat each parameter differently as far as derivatives calculation is concerned, this can be achieved by assigning each parameter to a group of its own.

### 3.3.2 Forward or Central Differences

The simplest way to calculate numerical derivatives is through the method of forward differences. To calculate derivatives in this manner, a PEST++ program varies each parameter in turn by adding an increment to its current value (unless the current parameter value is at its upper bound, in which case the increment is subtracted). The program then runs the model, reads altered model outputs, and approximates the derivative of each of these outputs with respect to the incrementally-varied parameter as the model output increment divided by the parameter value increment. (For log-transformed parameters this quotient is then multiplied by the current parameter value times the natural log of 10.) Hence if derivatives with respect to all parameters are calculated by the method of forward differences, the filling of a Jacobian matrix requires that a number of model runs be carried out equal to the number of adjustable parameters. Generally, this procedure is repeated during each iteration of the inversion or optimization process.

If the parameter increment is properly chosen (see below), this method can work well. However it is often found that as the solution to a nonlinear inverse or optimization problem is approached, greater accuracy in derivatives calculation than that afforded by the method of forward differences is required to achieve that solution. For this reason, programs of the PEST++ suite also allow for derivatives to be calculated using three incrementally-varied parameter values rather than two. Experience shows that derivatives calculated on this basis are accurate enough for most occasions provided, once again, that parameter increments are chosen wisely. As three-point derivative calculations are normally carried out by first adding an increment to a current parameter value and then subtracting an increment, the method is referred to herein as the “central” or “three-point” method of derivatives calculation. Note that if a parameter value is at its upper bound, the parameter increment is subtracted once and then twice; if it is at its lower bound, the increment is added once and then twice.

Programs of the PEST++ suite use one of three methods to calculate central derivatives. In the first or “outside points” method, the two outer parameter values (i.e. that for which an increment has been added and that for which an increment has been subtracted) are used in the same finite-difference type of calculation as is used in the forward difference method. This method yields more accurate derivative values than the forward difference method because the (unused) current parameter value is at the centre of the finite-difference interval (except where the parameter is at its upper or lower bound). The second method is to define a parabola through the three parameter-model-output pairs and to calculate the derivative of this parabola with respect to the incrementally-varied parameter at the current value of that parameter. This method, referred to as the “parabolic” method, can yield very accurate derivatives if the values of model outputs can be read with sufficient precision. The third method implements the least-squares principal to define a straight line of best fit through the three parameter-model-output pairs, and to take the derivative as the slope of this line. This method may work best where model outputs cannot be read with great precision, because of deficiencies in the model’s numerical solution method, or because the model writes numbers to its output files using a limited number of significant figures.

If the central method of derivatives calculation is used for all parameters, each iteration of an inversion or optimization process requires that at least twice as many model runs be carried out as there are adjustable parameters. If the central method is used for some parameters and the forward method is used for others, the number of model runs will lie somewhere between the number of adjustable parameters and twice the number of adjustable parameters.

### 3.3.3 Parameter Increments for Two and Three-Point Derivatives

Because of the importance of reliable finite-difference derivatives calculation, programs of the PEST++ suite (in common with PEST) provide considerable flexibility in the way that parameter increments are chosen. Mathematically, a parameter increment should be as small as possible so that the finite-difference method provides a good approximation to the partial derivative in a theoretical sense (remember that the derivative is defined as the limit of the finite difference as the increment approaches zero). However, if the increment is made too small, accuracy of derivatives calculation suffers because of the effect of round-off errors as two, possibly large, numbers are subtracted to yield a much smaller number. In most cases intuition and experience, backed up by trial and error, will be your best guide in reconciling these conflicting demands on increment size.

As stated above, variables which control derivatives calculation are assigned to parameter groups, rather than to parameters themselves. These variables reside in the “parameter groups” section of the PEST control file.

Three control variables, namely INCTYP, DERINC and DERINCLB are of primary relevance to the setting of parameter increments. INCTYP determines the type of increment to use, for which there are three options, namely “absolute”, “relative” and “rel\_to\_max”. If the increment type for a parameter group is “absolute”, the increment used for all parameters in the group is supplied as the input variable DERINC; this increment is added (and subtracted for central derivatives calculation) directly to the current value of a particular group member when calculating derivatives with respect to that parameter. However if the increment type is “relative”, DERINC is multiplied by the current absolute value of a parameter in order to determine the increment for that parameter. In this way the parameter increment is adjusted upwards and downwards as the parameter itself is adjusted upwards and downwards; this may have the effect of maintaining significance in the difference between model outcomes calculated on the basis of the incrementally varied parameter. If the increment type for a group is “rel\_to\_max”, the increment for all members of that group is calculated as DERINC times the absolute value of the group member of currently greatest absolute value. This can be a useful means by which to calculate increments for parameters whose values can vary widely, including down to zero. The “relative” aspect of the “rel\_to\_max” option may maintain model outcome difference significance as described above; however, because the increment is calculated as a fraction of the maximum absolute value occurring within a group, rather than as a fraction of the value of each parameter, an individual parameter can attain near-zero values without its increment simultaneously dropping to zero.

Further protection against the occurrence of near-zero increments for “relative” and “rel\_to\_max” increment types is provided through the variable DERINCLB. This variable provides a standby absolute increment which can be used in place of the “relative” or “rel\_to\_max” increment if the increment calculated for a particular parameter using either of these latter methods falls below the absolute increment value provided by DERINCLB.

If a parameter is log-transformed then it is wise that its increment be calculated using the “relative” method, though programs of the PEST++ suite do not insist on this.

When choosing an increment for a parameter, care must be taken to ensure that the parameter can be written to the model input file with sufficient precision to distinguish an incremented parameter value from one which has not been incremented. Thus, for example, if a model input file template is such that a particular parameter value must be written to a space which is only four characters wide, and if the increment type for that parameter is “absolute” and the increment value is 0.0001 while the current parameter value is .01, it will not be possible to discriminate between the value of the parameter with and without its increment added. To rectify this situation, you should either increase the parameter field width in the template file (making sure that the model can still read the parameter) or increase the increment to a value whereby the incremented parameter’s value is distinguishable from its non-incremented value in a field width of only four characters.

It will be recalled that programs of the PEST++ suite write parameter values to model input files with the maximum possible precision, given parameter field widths provided in the pertinent template file. Also, for the purposes of derivatives calculation, PEST++ programs adjust a parameter increment to be exactly equal to the difference between a current parameter value and the incremented value of that parameter as represented (possibly with limited precision) in the model input file, as read by the model.

### 3.3.4 Settings for Three-Point Derivatives

The FORCEN control variable, featured in the “parameter groups” section of the PEST control file, determines whether derivatives for the parameters belonging to a particular group are calculated using the forward-difference method, the central-difference method, or whether the method should change as the inversion process progresses. FORCEN can be designated as “always\_2”, “always\_3” or “switch”. If it is supplied as “always\_2”, derivatives calculation is through forward differences for all parameters within the group for the duration of the inversion or optimization process; if it is set to “always\_3”, central (i.e. three-point) derivatives are used for the entirety of the inversion or optimization process. However if FORCEN is supplied as “switch”, a PEST++ program will commence the inversion process using forward differences for all members of the group, and switch to using central differences on the first occasion that the relative reduction in the objective function between iterations is less than the value of the variable PHIREDSWH (which resides in the “control data” section of the PEST control file).

Two parameter group variables pertain specifically to the calculation of derivatives using higher order methods. These are DERINCMUL and DERMTHD. If FORCEN is set to “always\_3” or “switch”, DERMTHD must be set to one of “outside\_pts”, “parabolic” or “best\_fit”; this determines the method of central derivatives calculation to be used by programs of the PEST++ suite. These three options are discussed above.

The variable DERINCMUL is the parameter increment multiplier; this is the value by which DERINC is multiplied when it is used to evaluate increments for any of the three central derivatives methods. Sometimes it is useful to employ larger increments for central derivatives calculation than for forward derivatives calculation, especially where the model output dependence on parameter values is “bumpy” (see the next section). However if the increment is raised too high, derivative precision must ultimately fall. Note that through DERINCMUL you can also reduce the increment used for central derivatives calculation if you wish.

For increments calculated using the “relative” and “rel\_to\_max” methods, the variable DERINCLB has the same role in central derivatives calculation as it does in forward derivatives calculation, namely to place a lower limit on the absolute increment value. Note, however, that DERINCLB is not multiplied by DERINCMUL when central derivatives are calculated.

### 3.3.5 How to Obtain Derivatives You Can Trust

Reliability of derivatives calculation can suffer if the model which you are trying to calibrate, or which you are employing for management optimization, does not write its outcomes to its output file(s) using many significant figures.If you have any control over the precision with which a model writes its output data, you should request that the maximum possible precision of representation be used. Although PESTPP and PESTPP-OPT will happily attempt adjustment of parameters or decision variables on the basis of limited-precision model outputs, their ability to perform their parameter estimation or decision variable optimization tasks decreases as the precision of model outputs decreases.

If a model is comprised of multiple sub-model executables run through a batch or script file, then you should also ensure that numbers are transferred between these various sub-models with maximum precision. Thus every sub-model comprising the composite model should record numbers to those of its output files which are read by other sub-models with full numerical precision.

Many models employ finite difference or finite element approximations to partial differential equations. Through these models, problems which are continuous in space and time are approximated by discrete representations of the same problem in order that the partial differential equation(s) describing the original problem can be cast as a matrix equation of high order. The resulting matrix equation is then often solved by an iterative technique in which the solution vector is successively approximated until “convergence” is judged to have been attained. Most iterative solvers deem a solution to be acceptable when no element of the solution vector varies by more than a user-specified tolerance between successive iterations. If this threshold is set too large, model output precision is reduced. This may not matter for some model applications. However it may seriously compromise the numerical integrity of finite-difference derivatives calculated by programs of the PEST++ suite. Hence a model’s solution convergence criterion may need to be set tighter than normal when that model is being used with PESTPP or PESTPP-OPT than when it is being used on its own. Caution must be exercised, however. If the closure criterion is set too small, this may result in excessively long run times, or even model-perceived failure of the iterative solution process.

Sometimes the adaptive time-stepping scheme employed by a highly non-linear model can contribute to numerical granularity of its outputs. An incremental variation in the value of a particular parameter may lead to adoption of a different time-stepping strategy by a model from that which it employed for the non-incremented parameter. Differences in model outputs at a particular simulation time may then partially reflect the solution trajectory; they may not therefore be a unique function of the difference in parameter values.

If numerical models of these types are to be used with PESTPP and PESTPP-OPT whose algorithms rely on finite difference derivatives, it is important that model control variables that govern its numerical solution procedure be set in favour of precision over execution speed. Although the model run-time may increase as a result, the time spent in solving an inversion or optimization problem may actually be reduced.

Even after you have instructed the model to write to its output file(s) with as much precision as possible, and you have adjusted the model’s solution settings for heightened precision, model outputs may still be “granular” because of the nature of a simulator’s algorithm. In this case it may be wise to set parameter increments larger than you normally would, in order that local “bumps” in model outputs are bridged by these increments. While use of a large increment incurs penalties due to poor representation of a derivative by a finite difference (especially for highly nonlinear models), this can be mitigated by the use of one of the three-point methods of derivatives calculation offered by programs of the PEST++ suite. Because of its second order representation of the relationship between model output and parameter values, the parabolic method can generate reliable derivatives even for large parameter increments. However, if model outputs are really bumpy, the best-fit method may be more accurate.

### 3.3.6 Looking at Model Outputs under the Magnifying Glass

If PESTPP or PESTPP-OPT does not perform as well as you think it should, then bad numerical derivatives may be the cause of the problem. The PEST suite includes a number of utility programs which allow you to explore the integrity of finite-difference derivatives. See documentation for JACTEST, POSTJACTEST and MULJCOSEN in part II of the PEST manual.

## 3.4 The Jacobian Matrix File

Elements of a Jacobian matrix represent partial derivatives of model outputs with respect to model parameters. In the present context, “model outputs” are model-calculated numbers that are named in the “observation data” section of a PEST control file; these are read from model output files using instruction files. “Parameters” are those which are declared as adjustable in the “parameter data” section of a PEST control file.

Each column of the Jacobian matrix contains partial derivatives of model outputs with respect to a particular parameter. Each row of a Jacobian matrix contains partial derivatives of a particular model output with respect to all adjustable parameters. If a parameter is denoted as log-transformed in a PEST control file, then partial derivatives contained in a Jacobian matrix file are with respect to the log of that parameter.

In common with PEST, programs PESTPP and PESTPP-OPT of the PEST++ suite record the Jacobian matrix which they calculate in a binary, compressed file named *case.jco* where *case* is the filename base of a PEST control file. (This is commonly referred to as a “JCO file” in PEST parlance.) PESTPP also records a file named *case.jcb* for Jacobian matrix storage. The protocol used by programs of the PEST++ suite for storage of a Jacobian matrix in these binary files is identical to that used by PEST. Hence this type of file is interchangeable between the two suites.

A number of utility programs provided with PEST record part or all of the contents of a JCO file in ASCII format where it can be read by a human being. See the JACWRIT, JCO2MAT and JROW2VEC utilities in particular. Another useful PEST utility is JCO2JCO. This builds a JCO file corresponding to a new PEST control file from that corresponding to an existing PEST control file, provided that the contents of the former file are a subset of those of the latter file. Other PEST utilities support the construction of a large JCO file from component JCO files built on the basis of smaller PEST (and hence PEST++) datasets. See part II of the PEST manual for details. PyEMU provides similar capabilities for JCO file manipulation within a python environment.

A JCO file has many uses. Many of these uses are as important as the model calibration process itself. Hence it is not unusual for PEST or PESTPP to be run purely for the purpose of filling a Jacobian matrix and writing a JCO file. For either of these programs, this is achieved by setting the NOPTMAX variable in the “control data” section of the PEST control file to -1 or ‑2.

Uses to which a JCO file may be put include the following.

* Examination of local sensitivities of model outputs to parameters and/or decision variables.
* Giving PEST or PESTPP a “head start” in calibrating a model by providing it with a pre-calculated Jacobian matrix to use in its first iteration. PEST uses this matrix if started with the “/i” switch. For PESTPP this is achieved through use of the *base\_jacobian()* control variable.
* To support the many types of linear analysis implemented by utility programs supplied with PEST, and functions provided by PyEMU; these calculate
  + parameter identifiability;
  + parameter and predictive uncertainty;
  + parameter contributions to predictive uncertainty;
  + data worth;
  + the effects of model defects.

## 3.5 The Objective Function

Most programs of the PEST and PEST++ suites minimize a least-squares objective function. This is calculated as the sum of squared weighted differences between measurements (or prior information equations) and corresponding model outputs. The difference between a measurement and a model output to which it corresponds is referred to as a residual. Let the *i*th residual be designated as *ri*. Let the weight associated with the *i*th observation (which may be a prior information equation) be designated as *wi*. Then the objective function Φ is calculated as

 (3.3)

Obviously, if an observation is ascribed a weight of zero, then the residual associated with that observation makes no contribution to the total objective function. An observation can therefore be removed from an inversion process by assigning it a weight of zero. This provides a far easier mechanism for removal of an observation from the calibration dataset than that of re-building the PEST control file (and the pertinent instruction file) with this observation absent.

Observations can be divided into so-called “observation groups”. (Note that the term “observation group” includes prior information groups. Prior information is, after all, “observations” which pertain directly to parameters, or to linear combinations of parameters; see chapter 4 of this manual for full details.) Where observations and prior information equations are grouped in this manner, programs of the PEST and PEST++ suites calculate an objective function for each such group. These objective function components are listed to the screen and to pertinent run record files. A user can employ this information to ensure that no observation group dominates the overall objective function, or is invisible to the objective function. Either situation can be rectified by adjusting weights assigned to members of pertinent groups. Utility programs supplied with PEST, and functionality provided by PyEMU can facilitate this operation. Programs within the PEST Groundwater and Surface Water Utility suites can also be used for inter-group weights adjustment.

Where Tikhonov regularization is employed to promulgate parameter uniqueness during highly parameterized inversion, the total objective function is further partitioned into two components, namely the measurement objective function and the regularization objective function. The former quantifies model-to-measurement misfit. The latter quantifies departures of parameters from a preferred state which acts as a kind of “parameter fall-back position” which parameters should adopt unless there is information to the contrary in the calibration dataset. See Doherty (2015) for full details.

The philosophy of weights assignment is a complex topic. Obviously, higher weights should be awarded to measurements that are accompanied by less noise. However measurement noise is not the only consideration to be taken into account when assigning weights. So-called “structural noise” often makes a significant contribution to model-to-measurement misfit. Weighting strategies that can accommodate various types of structural noise are addressed by Doherty (2015), Doherty and Welter (2010) and White et al (2014). A simple procedure that often works well is to assign observations of different types to different observation groups, and to then use a utility program such as PWTADJ1 from the PEST suite to ensure that the contribution made to the objective function by each observation group is about the same as that made by any other observation group at the commencement of the inversion process.

When calibrating a surface water model, or when including contaminant concentrations as a component of the calibration dataset of a groundwater model, it may be useful for the magnitudes of weights to reflect the magnitudes of measurements. Using this strategy, small flows/concentrations (which are often high in information content) can be rendered visible in the inversion process. They can therefore influence parameter values that are estimated through that process. Programs supplied with the PEST Ground and Surface Water Utilities provide ideas and assistance in designing a suitable weighting strategy.

Unlike PEST, programs of the PEST++ suite do not (at the time of writing) support use of covariance matrices in formulation of an objective function.

# 4. The PEST Control File

## 4.1 General

Programs of the PEST++ suite, like PEST itself, require three types of input file. Two of these were discussed in chapter 2 of this manual, namely template and instruction files. On any occasion that a PEST++ program is run, as many of each of these must be provided as there are model input files in which parameters and/or decision variables reside, and model output files from which numbers must be read, respectively. However there is only one PEST control file.

This chapter describes the PEST control file. However, it will not be discussed in the same detail as in part I of the PEST manual. This is because programs of the PEST++ suite do not read all of the variables that reside in this file. Furthermore, some entire sections of a PEST control file have no relevance for programs of the PEST++ suite because they pertain to functionality which the PEST++ suite does not support. Programs of the PEST++ suite do require their own specific control variables, however. As will be discussed later in this chapter, these can be provided anywhere in a PEST control file following a line that begins with the “++” character string. Conversely, lines that begin with the “++” string are ignored by programs of the PEST suite.

With the exception of section headers and lines that being with the “++” string, each line of a PEST control file is comprised of the values of variables listed in sequence. These values may be integers, real numbers or text strings, the latter often providing the names of files. Each such item must be separated from its neighbour by whitespace, i.e. spaces or tabs. Multiple consecutive spaces or tabs are permitted. To avoid confusion, a text variable which contains a space must be enclosed in quotes. The variable to which each number or text string pertains is identified by the line on which it resides, and by its position on that line.

The first line of a PEST control file must begin with the string “pcf”; this stands for “PEST control file”. No other variables must appear on this first line. Versions of PEST from 15 onwards, and versions of PEST++ from 4 onwards, allow blank lines to be interspersed with data lines in a PEST control file. They also allow for the presence of comments. Comments are preceded by the “#” character. Any text to the right of that character is treated as a comment, and hence ignored by a PEST++ program (including the entirety of a line if the “#” character leads the line). However, unless the “#” character is the first character on a line, it must be preceded by a space; also, it must not be part of a string that is enclosed in quotes. These precautions prevent the occurrence of the “#” character within a filename from being misinterpreted as introducing a comment.

## 4.2 Naming Conventions

A PEST control file must have an extension of *.pst*. Suppose that its filename base is *case*. Then the PEST control file must be named *case.pst*.

As they run, programs of the PEST++ suite produce many files. The number and type of files that are written depend on the program. All of these files have the same filename base as that of the PEST control file on which their run is based (*case* in the present example).

## 4.3 Sections

The PEST control file is subdivided into sections. Each section begins with a header. A header is easily recognized because it begins with the “\*” character followed by a space. Table 4.1 shows sections that can appear in a PEST control file. For PEST, some of these sections are optional and can be omitted. Those that are present must be provided in the order shown in this table. Programs of the PEST++ suite ignore many of these sections.

|  |  |  |  |
| --- | --- | --- | --- |
| **Section** | **Status for PEST programs** | **Status for PEST++ programs** | **Contents** |
| control data | mandatory | mandatory | Problem dimensions, mode of PEST operation, termination criteria, change limits and other control variables. |
| automatic user intervention | optional | ignored | Variables which govern operation of PEST’s automatic user intervention functionality. |
| singular value decomposition | optional | optional | Variables which govern operation of singular value decomposition when used as an inverse problem solution device. |
| lsqr | optional | ignored | Variables which govern operation of the LSQR algorithm when used as an inverse problem solution device. |
| sensitivity reuse | optional | ignored | Variables which determine whether and how PEST re-uses some parameter sensitivities in consecutive iterations. |
| svd assist | optional | ignored | Variables which specify the manner in which the SVD-assist methodology is implemented in solving an inverse problem. |
| parameter groups | mandatory | mandatory | Variables which govern the way in which finite-difference derivatives are calculated. |
| parameter data | mandatory | mandatory | Parameter initial values, transformation status, bounds, groups, scales and offsets. |
| observation groups | mandatory | mandatory | Lists observation groups, and provides the names of files which hold observation covariance matrices. |
| observation data | mandatory | mandatory | Lists observations, weights and groups to which observations belong. |
| derivatives command line | optional | ignored | Provides the command used to run the model if the model calculates some or all of its own derivatives, and the file from which these derivatives are read. |
| model command line | mandatory | mandatory | Provides one or a number of commands used to run the model. |
| model input | mandatory | mandatory | Lists template and corresponding model input files. |
| model output | mandatory | mandatory | Lists instruction and corresponding model output files. |
| prior information | optional | optional | Provides linear prior information equations employed in the inversion process. |
| predictive analysis | optional | ignored | Contains variables which govern implementation of PEST’s “predictive analysis” functionality. |
| regularization | optional | optional | Contains variables which govern implementation of PEST’s “regularization” functionality. |
| pareto | optional | ignored | Contains variables which govern implementation of PEST’s “pareto” functionality. |

Table 4.1 Sections of a PEST control file.

Note that in table 4.1, separate sections are required for model input files and model output files. In previous versions of the PEST control file, these were combined into a single section. New versions of PEST++ programs tolerate the old convention in order to maintain backwards compatibility with PEST control files prepared for previous versions.

## 4.4 Control Variables

Figure 4.1 names variables which appear in a PEST control file. They are listed according to the position that they occupy in the file. This figure omits sections of the PEST control file that programs of the PEST++ suite ignore. It also omits control variables which are optional for PEST but are ignored by PEST++ programs. It does include some variables, however, which must be present within a PEST control file even though their values are not used by programs of the PEST++ suite. The presence of these variables defines line numbers and/or the locations of variables that follow them on a certain line. A control file provided to a PEST++ suite program should include all of these variables.

In figure 4.1, variables whose values are actually used by a program of the PEST++ suite to control its operations are shaded; these are described below. Of these variables, those which are optional are enclosed in square brackets. The roles of all other variables which appear in figure 4.1 are discussed in PEST documentation. Nevertheless, a number of these variables are discussed herein as PEST-acceptable values for these variables will preclude objections being raised by the PESTCHEK checking utility (see below).

It is also important to note that not all programs of the PEST++ suite use all of the variables that are shaded in figure 4.1. In fact, only PESTPP and PESTPP-OPT use variables which appear in the “parameter groups” section of the PEST control file (these govern calculation of finite-difference derivatives); furthermore PESTPP does not use these variables when it implements optimization through differential evolution.

|  |
| --- |
| pcf  \* control data  RSTFLE PESTMODE  NPAR NOBS NPARGP NPRIOR NOBSGP  NTPLFLE NINSFLE PRECIS DPOINT [NUMCOM]  RLAMBDA1 RLAMFAC PHIRATSUF PHIREDLAM NUMLAM  RELPARMAX FACPARMAX FACORIG  PHIREDSWH  NOPTMAX PHIREDSTP NPHISTP NPHINORED RELPARSTP NRELPAR  ICOV ICOR IEIG  \* singular value decomposition  SVDMODE  MAXSING EIGTHRESH  EIGWRITE  \* parameter groups  PARGPNME INCTYP DERINC DERINCLB FORCEN DERINCMUL DERMTHD  (*one such line for each parameter group*)  \* parameter data  PARNME PARTRANS PARCHGLIM PARVAL1 PARLBND PARUBND PARGP SCALE OFFSET DERCOM  (*one such line for each parameter*)  PARNME PARTIED  (*one such line for each tied parameter*)  \* observation groups  OBGNME  (*one such line for each observation group*)  \* observation data  OBSNME OBSVAL WEIGHT OBGNME  (*one such line for each observation*)  \* model command line  COMLINE  (*one such line for each model command line*)  \* model input  TEMPFLE INFLE  (*one such line for each template file*)  \* model output  INSFLE OUTFLE  (*one such line for each instruction file*)  \* prior information  PILBL PIFAC \* PARNME + PIFAC \* log(PARNME) ... = PIVAL WEIGHT OBGNME  (*one such line for each article of prior information*)  \* regularization  PHIMLIM PHIMACCEPT [FRACPHIM]  WFINIT WFMIN WFMAX  WFFAC WFTOL [IREGADJ] |

Figure 4.1 Variables comprising a minimalist PEST control file.

Figure 4.2 provides an example of a simple PEST control file.

|  |
| --- |
| pcf  \* control data  restart regularization  5 19 2 2 3  2 3 single point  10.0 -3.0 0.3 0.03 10  10.0 10.0 0.001  0.1  50 0.005 4 4 0.005 4  1 1 1  \* parameter groups  ro relative 0.01 0.0 switch 2.0 parabolic  h relative 0.01 0.0 switch 2.0 parabolic  \* parameter data  ro1 fixed factor 0.5 .1 10 ro 1.0 0.0  ro2 log factor 5.0 .1 10 ro 1.0 0.0  ro3 tied factor 0.5 .1 10 ro 1.0 0.0  h1 none factor 2.0 .05 100 h 1.0 0.0  h2 log factor 5.0 .05 100 h 1.0 0.0  ro3 ro2  \* observation groups  obsgp1  obsgp2  prgp1  \* observation data  ar1 1.21038 1.0 obsgp1  ar2 1.51208 1.0 obsgp1  ar3 2.07204 1.0 obsgp1  ar4 2.94056 1.0 obsgp1  ar5 4.15787 1.0 obsgp1  ar6 5.7762 1.0 obsgp1  ar7 7.7894 1.0 obsgp1  ar8 9.99743 1.0 obsgp1  ar9 11.8307 1.0 obsgp2  ar10 12.3194 1.0 obsgp2  ar11 10.6003 1.0 obsgp2  ar12 7.00419 1.0 obsgp2  ar13 3.44391 1.0 obsgp2  ar14 1.58279 1.0 obsgp2  ar15 1.1038 1.0 obsgp2  ar16 1.03086 1.0 obsgp2  ar17 1.01318 1.0 obsgp2  ar18 1.00593 0.0 obsgp2  ar19 1.00272 0.0 obsgp2  \* model command line  model.bat  \* model inputoutput  ves1.tpl a\_model.in1  ves2.tpl a\_model.in2  \* model output  ves1.ins a\_model.ot1  ves2.ins a\_model.ot2  ves3.ins a\_model.ot3  \* prior information  pi1 1.0 \* h1 = 1.0 3.0 prgp1  pi2 1.0 \* log(ro2) + 1.0 \* log(h2) = 2.6026 2.0 prgp1  \* regularization  125.0 130.0 0.1000000  1.0 1.0e-10 1.0e10  1.3 1.0e-2 1 |

Figure 4.2 Example of a PEST control file.

## 4.5 The PESTCHEK Utility

PESTCHEK is a utility program that is supplied with the PEST suite. It reads an entire PEST input dataset, this being comprised of a PEST control file and all template and instruction files cited therein. It checks all of these files for correctness and consistency. In doing so, it performs all of the tasks performed by the TEMPCHEK and INSCHEK utilities that were mentioned in chapter 2 of this manual. However, it goes further than this. It ensures that any parameter that is cited in a template file is also cited in a PEST control file (and vice versa), and that any observation that is cited in an instruction file is also cited in the PEST control file (and vice versa). The benefits of such checking are obvious.

PESTCHEK also subjects a PEST control file to thorough error and consistency checking. For example, it informs a user if a parameter is not between its upper and lower bounds, or if a parameter which is denoted as being log-transformed is endowed with a zero or negative initial value. It checks that the number of tied-parent parameter relationships that are provided in the second half of the “parameter data” section of a PEST control file are in accordance with the number of parameters that are denoted as being tied in the first half of this section. Consistency checks are made between equations comprising the “prior information” section of a PEST control file and the transformation status of parameters in the “parameter data” section of the PEST control file.

PESTCHEK ignores lines in a PEST control file that begin with the “++” string. Hence it performs no checking of control variables that are specific to members of the PEST++ suite. The above examples of its checking functionality indicate, however that it can still be used to at least partially ensure the quality of a PEST++ input dataset. However if it is to provide this service, certain conditions must be met. These include the following.

* Variables such as NOBS, NPARGP, NOBSGP and NPRIOR that are ignored by programs of the PEST++ suite, but that are read by PESTCHEK (and PEST), should have the correct values. These indicate the number of observations, parameter groups, observation groups and prior information equations respectively that are featured in a PEST control file. (Programs of the PEST++ suite evaluate these numbers themselves based on the contents of respective sections of a PEST control file.)
* Control variables such as those which govern PEST’s calculation of the Marquardt lambda (RLAMBDA1, RLAMFAC, PHIRATSUF, PHIREDLAM and NUMLAM) should be coherent. Suitable values are suggested below for these and other variables, though little will be said about their usage by PEST.

Variables appearing in a PEST control file which are used by members of the PEST++ suite are now described. Sensible, PESTCHEK-safe placeholder values are provided for all variables whose presence is required in a PEST control file, but which are not actually used by members of the PEST++ suite. The interested reader is referred to part I of the PEST manual for further details.

## 4.6 Control Data Section

### 4.6.1 General

Variables appearing in the “control data” section of a minimalist PEST control file are shown in figure 4.3 (which is reproduced from figure 4.1).

|  |
| --- |
| \* control data  RSTFLE PESTMODE  NPAR NOBS NPARGP NPRIOR NOBSGP  NTPLFLE NINSFLE PRECIS DPOINT [NUMCOM]  RLAMBDA1 RLAMFAC PHIRATSUF PHIREDLAM NUMLAM  RELPARMAX FACPARMAX FACORIG  PHIREDSWH  NOPTMAX PHIREDSTP NPHISTP NPHINORED RELPARSTP NRELPAR  ICOV ICOR IEIG |

Figure 4.3 Variables appearing in the “control data” section of a PEST control file.

### 4.6.2 First Line

The first line of the “control data” section of a PEST control file must contain the string “\* control data”. Note that the “\*” character is followed by a space.

### 4.6.3 Second Line

This line holds two variables, namely RSTFLE and PESTMODE. The first of these variables informs PEST whether it should record files that hold restart information. Set this to “restart”. The second variable sets the mode of PEST behaviour. Set this to “estimation” unless you wish to run PESTPP in “regularization” mode. If this is the case, then PESTPP will read the “regularization” section of the PEST control file if it is present; if a “regularization” section is not present, PESTPP provides default values for all regularization control variables.

### 4.6.4 Third Line

Although programs of the PEST++ suite read only the first variable on this line, the other variables that are featured on this line in figure 4.3 must be set to correct values to pass the scrutiny of PESTCHEK. All of these variables are integers. Their values must be greater than zero, except for NPRIOR which can be zero.

NPAR is the number of parameters featured in the “parameter data” section of a PEST control file. NOBS is the number of observations featured in the “observation data” section of a PEST control file. NPARGP is the number of parameter groups featured in the “parameter groups” section, while NPRIOR is the number of prior information equations featured in the “prior information” section of a PEST control file. The final variable on this line (NOBSGP) is the number of observation groups featured in the “observation groups” section of a PEST control file.

### 4.6.5 Fourth Line

The two leading variables on this line are NTPFLE and NINSFLE. Both of these are integers. Both must be greater than 1. Even though these variables are not read by programs of the PEST++ suite, both should be given correct values to ensure that a PEST control file is PESTCHEK-friendly. NTPFLE is the number of template files featured in the “model input” section of the PEST control file, while NINSFLE is the number of instruction files featured in the “model output” section of the PEST control file.

PRECIS and DPOINT are text variables which guide PEST in the writing of parameter values on model input files. Programs of the PEST++ suite do not use these variables. Set them to “single” and “point” respectively to render a PEST control file PESTCHEK-friendly.

NUMCOM, an integer variable, is the number of model command lines featured in the “model command line” section of a PEST control file. If there is only a single command line, set this to 1 or omit it altogether.

### 4.6.6 Fifth Line

The fifth line of the “control data” section of a PEST control file contains variables which control the way in which PEST selects a Marquardt lambda. Programs of the PEST++ suite which employ a Marquardt lambda do not read this line because they use their own lambda control variables. For PESTCHEK-friendliness, set RLAMBDA1, RLAMFAC, PHIRATSUF, PHIREDLAM and NUMLAM to 10.0, -2.0, 0.3, 0.01 and 10 respectively.

### 4.6.7 Sixth Line

This line contains the three real-valued variables, RELPARMAX, FACPARMAX and FACORIG. As is discussed in section 3.2.5 of this manual, these are used by PESTPP to limit the change that any parameter can undergo during any iteration of an inversion process. Where a model is highly nonlinear, and where adjustments to parameter values are calculated using local gradients (i.e. using a Jacobian matrix that is based on current parameter values), the calculated amounts by which parameters should change in order to accrue a reduction in an objective function may actually cause the objective function to rise because of large changes in the objective function gradient along the trajectory of parameter adjustment. Use of these control variables to limit the change that any parameter can undergo during any iteration of an inversion process can safeguard that process from severe oscillations caused by successive parameter over-adjustment and ensuing over-compensation. All parameters are subject to a change limit. The length of the overall parameter adjustment vector is reduced (but its direction is not changed) to ensure that these limits are respected.

Values of 10.0 are often suitable for both RELPARMAX and FACPARMAX. Note, however, that FACPARMAX can never be less than 1.0; RELPARMAX can be less than 1.0 as long as no parameter’s upper and lower bounds are of opposite sign. PESTCHEK will inform you if you get this wrong.

A suitable value for FACORIG is 0.001. See section 3.2.5 for further details.

### 4.6.8 Seventh Line

The seventh line of the minimalist “control data” section of figure 4.3 contains only one variable, namely PHIREDSWH. The only program of the PEST++ suite which reads this variable is PESTPP. Furthermore, it is ignored by PESTPP if it implements global optimization using differential evolution.

If the parameter-group-specific FORCEN variable appearing in the “parameter groups” section of the PEST control file is set to “switch”, then PESTPP will automatically switch from forward to central derivatives calculation if progress of an inversion process appears to be slowing. Switching takes place if the relative reduction in the objective function between successive iterations is less than the user-supplied value for PHIREDSWH. Thus if, for the *i*th iteration of an inversion process implemented by PESTPP,

(Φ*i-*1 - Φ*i* )/Φ*i-*1  ≤ PHIREDSWH (4.1)

(where Φ*i* is the objective function achieved at the end of the *i*th iteration), then PESTPP will use three-point derivatives in iteration *i+*1 (and all successive iterations) for all parameter groups for which FORCEN is set to “switch”.

A value of 0.1 is often suitable for PHIREDSWH.

### 4.6.9 Eighth Line

The eighth line of the “control data” section of a PEST control file holds variables which control termination of an inversion process. These are NOPTMAX, PHIREDSTP, NPHISTP, NPHINORED, RELPARSTP and NRELPAR. Suitable (PESTCHEK-friendly) values for these variables are provided in figure 4.2, namely 50, 0.005, 4, 4, 0.005 and 4. PESTPP uses all of these variables; PESTPP-IES uses all except RELPARSTP and NRELPAR. Other programs of the PEST++ suite use only NOPTMAX.

NOPTMAX, an integer variable, sets the maximum number of iterations that an inversion or optimization process is allowed to run. However values of 0, -1 and -2 trigger behaviour that is dependent on the PEST++ program that reads the PEST control file.

If NOPTMAX is set to 0, PESTPP does not estimate parameters. Instead it completes one model run using parameter values supplied in the “parameter data” section of the PEST control file. It computes objective function components based on the result of this single model run. This can be used to ensure that PESTPP setup is correct, and that weights assigned to observations in the “observation data” section of the PEST control file are suitable. Other programs of the PEST++ suite behave similarly. However because PESTPP-IES works with ensembles, rather than with an individual parameter field, it runs the model once for each parameter realization, calculates mean values of parameters and model outputs, and then ceases execution without updating the ensemble.

If NOPTMAX is set to -1 or -2, PESTPP calculates a Jacobian matrix and stores it in a Jacobian matrix file (i.e. a JCO file). If asked to do so, it also calculates linear statistics based on that Jacobian matrix. It then ceases execution. The same Jacobian matrix is available for more complex linear analysis using pertinent PEST utilities, as well as PyEMU. The Jacobian matrix that is stored in the JCO file can also be put to other uses; see section 3.4.

PHIREDSTP is a real variable whereas NPHISTP is an integer variable. If, in the course of a PESTPP inversion process, there have been NPHISTP optimization iterations for which

(Φ*i* - Φ*min* )/Φ*i* ≤ PHIREDSTP (4.2)

(Φ*i* being the objective function value at the end of the *i*th optimization iteration and Φmin being the lowest objective function achieved to date), PESTPP considers that the inversion process is at an end. Alternatively, if PESTPP has failed to lower the objective function over NPHINORED successive iterations, it ceases execution. PESTPP-IES supplies these same termination criteria to ensemble-mean objective functions.

If the magnitude of the maximum relative parameter change is less than RELPARSTP over NRELPAR successive iterations, then PESTPP ceases execution. The relative parameter change between iterations for any parameter is calculated using equation 3.2. PESTPP evaluates this change for all adjustable parameters at the end of all iterations, and determines the relative parameter change with the highest magnitude. If this maximum relative change is less than RELPARSTP, a counter is advanced by one; if it is greater than RELPARSTP, the counter is zeroed.

### 4.6.10 Ninth Line

The integer ICOV, ICOR and IEIG variables recorded on the 9th line of the control data section of a PEST control file are not used by any members of the PESTPP suite. Set all of these to 0 for PESTCHEK-friendliness.

## 4.7 Singular Value Decomposition Section

Singular value decomposition (i.e. SVD) as a solution device, and as a mechanism for ensuring numerical stability in solution of an ill-posed inverse problem, is discussed at length by Doherty (2015).

PESTPP always uses singular value decomposition, or a closely related numerical device, to solve an inverse problem. It uses default settings to govern the operation of this process. However these can be over-ridden by a user if he/she includes a “singular value decomposition” section in a PEST control file.

If a PEST control file includes a “singular value decomposition” section, then PESTPP only reads the MAXSING and EIGTHRESH variables from this section. These are used to determine the dimensionality of the calibration solution space, that is the number of combinations of parameters that are estimated through the inversion process. MAXSING sets the dimensionality of this space directly. EIGTHRESH sets it implicitly to the index immediately preceding that at which the square of the singular value is EIGTHRESH times the maximum singular value. Singular values pertain to **Q**½**J**, where **Q** is the weight matrix and **J** is the Jacobian matrix. The squares of these singular values pertain to **J**t**QJ**. See Doherty (2015), and section 6 of this manual, for further details.

Unless you wish to over-ride internal PESTPP settings for implementation of singular value decomposition, it is best to omit the “singular value decomposition” section from the PEST control file. If you do decide to include it, PESTCHEK-friendly values for control variables are those shown in figure 4.2. However set MAXSING to a number equal to, or greater than, the number of parameters featured in the PEST control file unless you specifically wish to reduce the dimensionality of the inverse problem solution space.

An issue that sometimes causes confusion is the different roles played by the PEST MAXSING and EIGTHRESH variables on the one hand, and the PEST++ *max\_n\_super()* and *super\_eigthresh()* control variables on the other hand. As is described later in this manual, the latter two variables are used to determine how many super parameters are used in SVD-assisted inversion. MAXSING and EIGTHRESH, on the other hand, control the operation of the singular value decomposition solution process, regardless of whether this is being used to estimate base parameters or super parameters.

Note that PESTPP ignores the SVDMODE variable in the “singular value decomposition” section of a PEST control file. If this section exists, then MAXSING and EIGTHRESH override internal PESTPP settings, even if SVDMODE is set to 0.

## 4.8 Parameter Groups Section

### 4.8.1 General

Every parameter must belong to a parameter group; the group to which each parameter belongs is denoted by the parameter-specific PARGP variable supplied in the “parameter data” section of the PEST control file.

Each parameter group must possess a unique name of twelve characters or less in length. However certain advantages accrue if the name is less than six characters in length. As is described in part II of the PEST manual, the ADDREG1 utility (which adds Tikhonov regularization to a PEST control file) groups prior information equations associated with different parameter groups into different regularization groups. A prefix of “regul\_” is appended to parameter group names in formulating prior information group names. The last six characters of a parameter group name are lost (if they exist) in formulating these names. If the first six characters of two parameter group names are the same, prior information group names formulated in this way are not unique.

Notwithstanding the role of the PEST ADDREG1 utility, the primary purpose of parameter groups is to provide a basis for assignment of variables which govern calculation of finite-difference derivatives. These variables are assigned to parameter groups, rather than to individual parameters, because the number of the latter may be very large in many inversion contexts. At the time of writing, programs of the PEST++ suite which calculate derivatives using finite parameter differences are PESTPP (unless it us implementing global optimization) and PESTPP‑OPT. Note that PESTPP-OPT calculates derivatives with respect to both model parameters and decision variables; the word “parameter” is used to refer to both of these in the discussion that follows.

Tied and fixed parameters must also be assigned to groups; however, as derivatives are not calculated with respect to these parameters, the groups to which they belong are of no significance (except in calculating the derivative increment for adjustable members belonging to the same group if the increment type is “rel\_to\_max”).

The many options available for finite-difference derivatives calculation are discussed in section 3.3 of this manual, as are the variables which control these options. That discussion is not repeated here; only a short description of the role of each variable is presented below.

Minimalist specifications (i.e. specifications which pertain only to functionality offered by members of the PEST++ suite) of the “parameter groups” section of a PEST control file are provided in figure 4.4.

|  |
| --- |
| \* parameter groups  PARGPNME INCTYP DERINC DERINCLB FORCEN DERINCMUL DERMTHD  (*one such line for each parameter group*) |

Figure 4.4 Minimalist specifications for the “parameter groups” section of a PEST control file.

### 4.8.2 Parameter Group Variables

#### PARGPNME

PARGPNME is the parameter group name. This must be a maximum of twelve characters in length, though for reasons stated above, there are occasions when its length is best limited to six characters.

If a group is featured in the “parameter groups” section of a PEST control file it is not essential that any parameters belong to that group. However if, in the “parameter data” section of a PEST control file, a parameter is declared as belonging to a group that is not featured in the “parameter groups” section of the same PEST control file, an error condition will arise.

#### INCTYP and DERINC

INCTYP is a character variable which can assume the values “relative”, “absolute” or “rel\_to\_max”. If it is “relative”, the increment used for forward-difference calculation of derivatives with respect to any parameter belonging to the group is calculated as a fraction of the current value of that parameter; that fraction is provided as the real variable DERINC. However if INCTYP is “absolute” the parameter increment for parameters belonging to the group is fixed, being again provided as the variable DERINC. Alternatively, if INCTYP is “rel\_to\_max”, the increment for any group member is calculated as a fraction of the group member with highest absolute value, that fraction again being DERINC.

Thus, for example, if INCTYP is “relative” and DERINC is 0.01 (a suitable value in many cases), the parameter increment for each group member is calculated as 0.01 times the current value of that parameter; this applies in all iterations of the inversion or optimization process. However if INCTYP is “absolute” and DERINC is 0.01, the parameter increment is the same for all members of the group over all iterations, this being 0.01. If INCTYP is “rel\_to\_max” and DERINC is again 0.01, the parameter increment for all group members is the same for any one iteration, this being 0.01 times the absolute value of the group member of highest current magnitude; however the increment may vary from iteration to iteration.

If a group contains members which are fixed and/or tied, it is important to note that the values of these parameters are taken into account when calculating parameter increments using the “rel\_to\_max” option.

For the “relative” and “rel\_to\_max” options, a DERINC value of between 0.01 and 0.02 is often appropriate. However no suggestion for an appropriate DERINC value can be provided for the “absolute” increment option; the most appropriate increment will depend on parameter magnitudes and types.

#### DERINCLB

If a parameter increment is calculated as “relative” or “rel\_to\_max”, it is possible that it may become too low if the parameter’s value becomes very small or, in the case of the “rel\_to\_max” option, if the magnitude of the largest parameter in the group becomes very small. A parameter increment becomes “too low” if it does not allow reliable derivatives to be calculated with respect to that parameter because of round-off errors incurred in the subtraction of nearly equal model outputs.

To circumvent this possibility, an absolute lower bound can be placed on parameter increments; this lower bound is the same for all group members, and is provided as the value of the DERINCLB control variable. Thus if a parameter value is currently 1000.0 and it belongs to a group for which INCTYP is “relative”, DERINC is 0.01, and DERINCLB is 15.0, the parameter increment will be 15.0 instead of 10.0 calculated on the basis of DERINC alone. If you do not wish to place a lower bound on parameter increments in this fashion, you should provide DERINCLB with a value of 0.0.

Note that if INCTYP is “absolute”, DERINCLB is ignored.

#### FORCEN

The character variable FORCEN (an abbreviation of “forward/central”) determines whether derivatives for group members are calculated using forward differences, or using one of the variants of the central difference method.

If FORCEN for a particular group is “always\_2”, derivatives for all parameters belonging to that group will always be calculated using the forward difference method; filling of the columns of the Jacobian matrix corresponding to members of the group will require as many model runs as there are adjustable parameters in the group. If FORCEN is provided as “always\_3”, the filling of these same columns will require twice as many model runs as there are parameters within the group. However the derivatives will be calculated with greater numerical precision.

If FORCEN is set to “switch”, derivatives calculation for all adjustable group members begins using the forward difference method. However it switches to the central difference method for the remainder of the inversion process on the iteration after which the relative objective function reduction between successive iterations is less than PHIREDSWH. A value for PHIREDSWH is supplied in the “control data” section of the PEST control file.

Experience has shown that in most instances, the most appropriate value for FORCEN is “switch”. This allows speed to take precedence over accuracy in the early stages of an inversion or optimization process when accuracy is not critical to objective function improvement, and accuracy to take precedence over speed later in the process when achievement of an objective function improvement requires that derivatives be calculated with as much precision as possible.

#### DERINCMUL

If derivatives are calculated using one of the three-point methods, the parameter increment is added to the current parameter value prior to a model run, and then subtracted prior to another model run. In some cases it may be desirable to increase the value of the increment when calculating derivatives using a three-point method. The real variable DERINCMUL allows you to do this. If three-point derivatives calculation is employed, the value of DERINC is multiplied by DERINCMUL; this applies whether DERINC holds the increment factor, as it does for the “relative” or “rel\_to\_max” increment types, or holds the parameter increment itself, as it does for the “absolute” increment type.

A DERINCMUL value of between 1.0 and 2.0 is usually satisfactory.

#### DERMTHD

As is described in section 3.3 of this manual, the three-point method of derivatives calculation implemented by programs of the PEST++ suite has three variants. If FORCEN for a particular parameter group is set to “always\_3” or “switch”, the program must be informed of which alternative to employ. This is accomplished through the character variable DERMTHD, which must be supplied as “parabolic”, “best\_fit” or “outside\_pts”. If FORCEN is set to “always\_2” for a particular group, you must still provide one of these three legal values for DERMTHD; however for such a parameter group, the value of DERMTHD has no bearing on derivatives calculation for its member parameters.

On most occasions, FORCEN should be set to “switch” while DERMTHD should be set to “parabolic”.

#### PESTCHEK-Friendly Values

Programs of the PEST++ suite which do not calculate finite difference derivatives ignore the contents of the “parameter groups” section of a PEST control file. Nevertheless, as was discussed above, sensible values should still be provided for variables appearing in this section so that PESTCHEK does not object. PESTCHEK will raise no objections to the values listed in figure 4.2 unless the initial values of some parameters are zero. Then a non-zero value for DERINCLB is required.

## 4.9 Parameter Data Section

### 4.9.1 General

It is the task of programs of the PEST++ suite to adjust parameters or decision variables in order to achieve some goal. This goal varies from program to program. They include

* achievement of a good fit with a calibration dataset (PESTPP);
* management optimization under chance constraints (PESTPP-OPT);
* exploration of global sensitivity (PESTPP-GSA);
* sampling of a posterior parameter distribution (PESTPP-IES).

For the sake or brevity in the following discussion, the term “parameter” refers to both parameters and decision variables.

For every parameter that is cited in a template file, ten pieces of information must be provided in the PEST control file in which the name of that parameter file is cited. Conversely, every parameter featured in a PEST control file must be cited at least once in a template file.

The “parameter data” section of a PEST control file is divided into two parts; in the first part of this section a line must be dedicated to each parameter. The second part of this section provides extra information for tied parameters, namely the name of the parameter to which each such tied parameter is linked. If there are no tied parameters, the second part of the “parameter data” section of a PEST control file is omitted.

Each item of parameter data is now discussed in detail. Specifications of the “parameter data” section of a PEST control file are provided in figure 4.5.

|  |
| --- |
| \* parameter data  PARNME PARTRANS PARCHGLIM PARVAL1 PARLBND PARUBND PARGP SCALE OFFSET DERCOM  (*one such line for each parameter*)  PARNME PARTIED  (*one such line for each tied parameter*) |

Figure 4.5 Specifications of the “parameter data” section of a PEST control file.

### 4.9.2 First Part

#### PARNME

PARNME is the parameter name. Each parameter name must be unique and of twelve characters or less in length; the name is case insensitive.

#### PARTRANS

PARTRANS is a character variable which must assume one of four values, these being “none”, “log”, “fixed” or “tied”.

If you wish that a parameter be log-transformed throughout an inversion or uncertainty analysis process, the value “log” must be provided. (Note that PESTPP-OPT does not allow log transformation of decision variables.) Logarithmic transformation of some parameters may have a profound effect on the success of these processes. If a parameter is log-transformed, programs of the PEST++ suite adjust the log of the parameter rather than the parameter itself. For those programs which calculate derivatives, the column of the Jacobian matrix pertaining to that parameter actually contains derivatives with respect to the log of the parameter. However, when you supply the parameter’s initial value (PARVAL1), and its upper and lower bounds (PARUBND and PARLBND), these must pertain to the parameter itself; likewise, at the end of the inversion or uncertainty analysis process, a PEST++ program provides the value of the parameter value itself rather than the log of its value.

Experience has shown repeatedly that log transformation of at least some parameters can make the difference between a successful inversion process and an unsuccessful one. This is because, in many cases, the relationship between model outputs and parameters is more linear if certain parameters are log-transformed. It also follows from the fact that, implied in log-transformation is normalization of parameters with respect to their innate variability. An inversion process is thereby able to access, to some extent at least, the benefits which follow from Kahunen-Loève transformation of parameters; see Doherty (2015).

However caution must be exercised when designating parameters as log-transformed. A parameter which can become zero or negative in the course of an inversion process must not be log-transformed; hence if a parameter’s lower bound is zero or less, programs of the PEST++ suite will disallow logarithmic transformation of that parameter. (Note, however, that by using an appropriate SCALE and OFFSET, you can ensure that parameters never become negative. Thus if you are estimating the value for a parameter whose domain, as far as the model is concerned, is the interval [‑9.99, 10], you can shift its effective domain to [0.01, 20] by designating a SCALE of 1.0 and an OFFSET of -10.0. Similarly, if a parameter’s domain is entirely negative, you can make its effective domain entirely positive by supplying a SCALE of -1.0 and an OFFSET of 0.0.)

If a parameter is fixed, it takes no part in an inversion, uncertainty analysis or optimization process. This is implemented by endowing PARTRANS for that parameter with a value of “fixed”. If a parameter is linked to another parameter, this is signified by a PARTRANS value of “tied”. In the latter case the parameter plays only a limited role in the inversion or optimization process. However the parameter to which the tied parameter is linked (this “parent” parameter must be neither fixed nor tied itself) takes an active part in this process; the tied parameter simply “piggy-backs” on the parent parameter, the value of the tied parameter maintaining at all times the same ratio to the parent parameter as the ratio of their initial values. The parent parameter of each tied parameter must be provided in the second part of the “parameter data” section of the PEST control file.

If a parameter is neither fixed nor tied, and is not log-transformed, the parameter transformation variable PARTRANS must be supplied as “none”.

If a particular inversion, uncertainty analysis or optimization problem would benefit from a more complex parameter transformation type than logarithmic, and/or if more complex relationships between parameters than ratio maintenance are required, these can be accomplished through use of the parameter preprocessor PAR2PAR supplied with the PEST suite; see part II of the PEST manual for details.

#### PARCHGLIM

This character variable is used to designate whether an adjustable parameter is relative-limited, factor-limited or absolute-limited; see section 3.2.5 of this manual, and the discussion of the RELPARMAX and FACPARMAX control variables. PARCHGLIM must be provided with a value of “relative” or “factor”. The former designates that alterations to a parameter’s value are factor-limited whereas the latter designates that alterations to its value are relative-limited.

The following aspects of change limit specifications should be noted.

* The only member of the PEST++ suite which imposes limits on parameter changes in this way is PESTPP.
* If a parameter is tied or fixed, its change limit is ignored.
* Parameters that are log-transformed cannot be assigned a relative change limit; they can only be assigned a factor change limit.
* At the end of each iteration of the inversion process, PESTPP lists the maximum factor and relative change undergone by any parameter.

#### PARVAL1

PARVAL1, a real variable, is a parameter’s initial value. For a fixed parameter, this value remains invariant during an inversion, uncertainty analysis or optimization process. For a tied parameter, the ratio of PARVAL1 to the parent parameter’s PARVAL1 sets the ratio between these two parameters that is maintained throughout the inversion, uncertainty analysis or optimization process. For an adjustable parameter PARVAL1 is the parameter’s starting value which, together with the starting values of all other adjustable parameters, is successively improved during an inversion or optimization process.

If using PESTPP then ideally, as explained by Doherty (2015), the initial value of a parameter should be the pre-calibration estimate of the parameter’s value based on expert knowledge alone. Through use of the PEST ADDREG1 utility (see part II of the PEST manual), this value can also be specified as the parameter’s “preferred value” when implementing Tikhonov regularization.

Caution should be exercised in choosing an initial parameter value of zero for the following reasons.

* If using PESTPP, a parameter cannot be subjected to relative or factor change limits during the first iteration of an inversion process if its value at the start of that iteration is zero. Furthermore, FACORIG cannot be used to modify the action of RELPARMAX and FACPARMAX (the relative- and factor-limiting control variables) in later iterations.
* A relative increment for derivatives calculation cannot be evaluated during the first iteration of parameter adjustment for a parameter whose initial value is zero. If the parameter belongs to a group for which derivatives are specified as “relative”, a non-zero DERINCLB variable must be provided for that group.
* If a parameter has an initial value of zero, the parameter can be neither a tied nor a parent parameter as the tied:parent parameter ratio cannot be calculated.

#### PARLBND and PARUBND

These two real variables represent a parameter’s lower and upper bound respectively. For adjustable parameters the initial parameter value (PARVAL1) must lie between these two bounds. However for fixed and tied parameters the values you provide for PARLBND and PARUBND are ignored. (The upper and lower bounds for a tied parameter are determined by the upper and lower bounds of the parameter to which it is tied, and by the ratio between the two.)

#### PARGP

PARGP is the name of the group to which a parameter belongs. As discussed above, parameter group names must be twelve characters or less in length and are case-insensitive. Any group which is cited in the “parameter data” section of a PEST control file must be properly defined in the “parameter groups” section of that file.

#### SCALE and OFFSET

Just before a parameter value is written to a model input file, it is multiplied by the real variable SCALE, after which the real variable OFFSET is added. The use of these two variables allows you to redefine the domain of a parameter. Because they operate on the parameter value “at the last moment” before it is written to a model input file, they take no part in an inversion/optimization process; in fact they can “conceal” from programs of the PEST++ suite the true value of a parameter as seen by the model. Instead, a PEST++ program adjusts the parameter *kp* where

*kp* = (*km* - *o*)/*s* (4.3)

Here *kp* is the parameter adjusted by a PEST++ program, *km* is the parameter seen by the model, and *s* and *o* are the SCALE and OFFSET for that parameter. If you wish to leave a parameter unaffected by SCALE and OFFSET (the usual case), set the SCALE to 1.0 and the OFFSET to 0.0.

#### DERCOM

PESTPP (but not PESTPP-OPT) can use different commands to run a model when calculating derivatives with respect to different parameters. The value of the DERCOM variable associated with a particular parameter is the number of that command. Commands are listed in the “model command line” section of a PEST control file. They are implicitly numbered in order of appearance, starting at 1.

In most instances of PEST++ usage, only one command is used to run the model. DERCOM should be set to 1 for all parameters.

### 4.9.3 Second Part

The second part of the “parameter data” section of a PEST control file is comprised of one line for each tied parameter; if there are no tied parameters, the second part of the “parameter data” section must be omitted.

Each line within the second part of the “parameter data” section of a PEST control file consists of two entries. The first is PARNME. This is the name of a parameter already cited in the first part of the “parameter data” section of the PEST control file; PARTRANS must be “tied” for this parameter. The second entry on the line, the character variable PARTIED, holds the name of the parameter to which the first-mentioned parameter is tied, i.e. the “parent parameter” of the first-mentioned parameter. The parent parameter must not be a tied or fixed parameter itself.

Note that multiple tied parameters can be linked to a single parent parameter. However a tied parameter can, naturally, be linked to only one parent parameter.

## 4.10 Observation Groups Section

Specifications for the “observation groups” section of a PEST control file are provided in figure 4.6.

|  |
| --- |
| \* observation groups  OBGNME  (*one such line for each observation group*) |

Figure 4.6 Specifications of the “observation groups” section of a PEST control file.

In the “observation groups” section of a PEST control file, a name is supplied for every observation group. These names must be provided one to a line. Observation group names must be twelve characters or less in length and are case insensitive. A name assigned to one observation group must not be assigned to any other observation group.

Note that prior information equations are also collected into groups; these groups are also referred to as “observation groups” as they perform an identical role to the groups that represent collected observations.

The same observation group cannot contain both observations and prior information equations.

Observation groups whose name begins with “regul” are special. Observations belonging to these groups are considered to comprise regularization constraints on an inverse problem. Programs of the PEST++ suite (and PEST itself), give these groups special treatment when PESTMODE is set to “regularization”. At a minimum, two objective functions are computed. One of these is named the “regularization objective function”; it is calculated using observations that belong to regularization groups, i.e. observation groups whose names begin with “regul”. The other objective function (i.e. that computed using members of observation groups whose names do not begin with “regul”) is named the “measurement objective function”.

## 4.11 Observation Data Section

The “observation data” section of a PEST control file is particularly simple. Its specifications are provided in figure 4.7.

|  |
| --- |
| \* observation data  OBSNME OBSVAL WEIGHT OBGNME  (*one such line for each observation*) |

Figure 4.7 Specifications of the “observation data” section of a PEST control file.

For every observation cited in a PEST instruction file, there must be one line of data in the “observation data” section of the PEST control file. Conversely, every observation for which data is supplied in the PEST control file must be represented in an instruction file.

Each line within the “observation data” section of a PEST control file must contain four entries. Each of these four entries is now discussed.

#### OBSNME

This is a character variable containing the observation name. An observation name must be twenty characters or less in length and contain no spaces. It is case-insensitive. Observation names must be unique.

#### OBSVAL

OBSVAL, a real variable, is the field or laboratory measurement corresponding to a model-generated observation. It is the role of many programs of the PEST++ suite to minimize the difference between this number and the corresponding model-calculated number through adjustment of parameter values. This difference is referred to as a residual.

#### WEIGHT

This is the weight attached to each residual in calculation of an objective function. The manner in which weights are used in objective function formulation is discussed in section 3.5 of this manual. See also chapter 6.

#### OBGNME

OBGNME is the name of the observation group to which an observation is assigned. When recording objective functions in their respective output files, programs of the PEST++ suite list the contribution made to the total objective function by all observation groups. It is good practice to assign observations of different types to different observation groups. This gives you the ability to choose observation weights in a way that ensures that one observation type does not dominate others in the inversion process by virtue of a vastly greater contribution to the objective function, or that one observation group is not dominated by others because of a diminutive contribution to the objective function.

The observation group name supplied through OBGNME must also be listed in the “observation groups” section of the PEST control file. Observation group names must be 12 characters or less in length.

## 4.12 Model Command Line Section

The “model command line” section of a PEST control file supplies the command which programs of the PEST++ suite must use to run the model. Optionally, this section can contain multiple commands; note however that, at the time of writing, not all programs of the PEST++ suite support multiple command line functionality.

PESTPP (like PEST) supports the use of different commands for running a model to compute derivatives with respect to different parameters. The model command number associated with each parameter is provided by the parameter-specific DERCOM variable which appears in the “parameter data” section of a PEST control file.

Figure 4.8 shows specifications for the “model command line” section of a PEST control file.

|  |
| --- |
| \* model command line  COMLINE  (*one such line for each model command line*) |

Figure 4.8 Specifications of the “model command line” section of a PEST control file.

The model command line may be simply the name of an executable file, or it may be the name of a batch or script file containing a complex sequence of steps. You may include the path name in a model command if you wish. If a program of the PEST++ suite is to be successful in running the model, then the model batch or script file, and all executable programs cited therein, must reside in the working folder (i.e. directory) of that program. Alternatively, full paths must be provided for all of these, or it must be ensured that the model batch or script file, and all executable programs cited therein, reside in a folder that is cited in the PATH environment variable. The safest option when parallelizing model runs over many computers is to include the batch/script file, and all executable programs cited therein, in the folder of each PEST++ worker.

## 4.13 Model Input Section

The “model input” section of a PEST control file relates PEST template files to model input files Its specifications are provided in figure 4.9.

|  |
| --- |
| \* model input  TEMPFLE INFLE  (*one such line for each template file*) |

Figure 4.9 Specifications of the “model input” section of a PEST control file.

For each template file - model input file pair, there should be a line within the “model input” section of the PEST control file containing two entries, namely the character variables TEMPFLE and INFLE. The first of these is the name of a PEST template file while the second is the name of the model input file to which the template file is matched. Filenames which contain a space should be enclosed in quotes. Pathnames should be provided for both the template file and the model input file if they do not reside in the current working folder. Construction details for template files are provided in chapter 2 of this manual.

It is possible for a single template file to be linked to more than one model input file; a separate line must be provided for each such pair of files in the “model input” section of the PEST control file. A model input file cannot be linked to more than one template file.

## 4.14 Model Output Section

The “model output” section of a PEST control file relates PEST instruction files to model output files.

Specifications of the “model output” section of a PEST control file are provided in figure 4.10.

|  |
| --- |
| \* model output  INSFLE OUTFLE  (*one such line for each instruction file*) |

Figure 4.10 Specifications of the “model output” section of a PEST control file.

The “model output” section of the PEST control file contains instruction file - model output file pairs. If a filename contains a space, its name must be enclosed in quotes. Pathnames must be provided for both instruction files and model output files if they do not reside in the current working folder. Construction details for instruction files are provided in chapter 2 of this manual.

A single model output file may be read by more than one instruction file. However any particular observation can only be read once; hence an instruction file cannot be matched to more than one model output file.

## 4.15 Prior Information Section

Prior information can be thought of as observations which pertain directly to parameters themselves. As such, they comprise part of a calibration dataset which, together with the observations themselves, assists in the estimation of parameters. An “observation” which comprises a prior information equation can involve more than one parameter. Relationships between parameters that are encapsulated in prior information equations must be linear. If parameter-constraining “observations” pertain to nonlinear relationships between parameters, these relationships must be calculated by the model itself. The PEST PAR2PAR utility can accomplish this task; see part II of the PEST manual.

The manner in which prior information equations are recorded in the “prior information” section of a PEST control file is not unlike that in which you would write an equation on paper yourself; however certain strict protocols must be observed. Refer to figure 4.2 for an instance of a PEST control file containing prior information. (Note that PEST utilities such as ADDREG1 and ADDREG2, as well as some of the programs comprising the Groundwater Data Utility suite, add prior information to a PEST control file automatically, this saving you the trouble of having to add it yourself. Similar functionality is available through the PyEMU library.)

Each item on a prior information line must be separated from its neighbouring items by at least one space. Each new article of prior information must begin on a new line. Note that while programs of the PEST suite require that no item of prior information exceed 300 characters in length, this limit does not apply to programs of the PEST++ suite. A continuation character (“&” followed by a space at the start of a line) allows you to write a lengthy prior information equation over several successive lines.

Prior information lines must adhere to the syntax set out in figure 4.11.

|  |
| --- |
| PILBL PIFAC \* PARNME + PIFAC \* log(PARNME) ... = PIVAL WEIGHT OBGNME  (one such line for each article of prior information) |

Figure 4.11. The syntax of a prior information line.

Each prior information article must begin with a prior information label; this is the character variable PILBL depicted in figure 4.11. Like observation names, this label must be no more than twenty characters in length, is case insensitive, and must be unique to each prior information equation.

Following the prior information label is the prior information equation. To the left of the “=” sign there are one or more combinations of a factor (PIFAC) plus parameter name (PARNME), with a “log” prefix to the parameter name if appropriate. PIFAC and PARNME are separated by a “\*” character (which must be separated from PIFAC and PARNME by at least one space) signifying multiplication. All parameters referenced in a prior information equation must be adjustable parameters; i.e. you must not include any fixed or tied parameters in an article of prior information. Furthermore, any particular parameter can be referenced only once in any one prior information equation; however, it can be referenced in more than one equation.

The parameter factor must never be omitted. Suppose, for example, that a prior information equation consists of only a single term, namely that an untransformed, adjustable parameter named “par1” has a preferred value of 2.305, and that you would like this information included in the inversion process with a weight of 1.0. If this article of prior information is given the label “pi1”, the pertinent prior information line can be written as

pi1 1.0 \* par1 = 2.305 1.0 pr\_info

If you had simply written

pi1 par1 = 2.305 1.0 pr\_info

programs of the PEST++ suite would have objected, complaining of a syntax error.

If a parameter is log-transformed, you must provide prior information pertinent to the log of that parameter, rather than to the parameter itself. Furthermore, the parameter name must be placed in brackets and preceded by “log” (note that there is no space between “log” and the following opening bracket). Thus, in the above example, if parameter “par1” is log-transformed, the prior information equation should be rewritten as

pi1 1.0 \* log(par1) = .362671 1.0 pr\_info

Note that logs are taken to base 10.

The left side of a prior information equation can be comprised of the sum and/or difference of a number of factor-parameter pairs of the type already illustrated; these pairs must be separated from each other by a “+” or “-” sign, with a space to either side of the sign. For example

pi2 1.0 \* par2 + 3.43435 \* par4 - 2.389834 \* par3 = 1.09e3 3.00 group\_pr

Prior information equations which include log-transformed parameters must express a relationship between the logs of those parameters. For example, if you would like the ratio between the estimated values of parameters “par1” and “par2” to be about 40.0, the prior information equation may be written as

pi3 1.0 \* log(par1) - 1.0 \* log(par2) = 1.60206 2.0 group\_pr

To the right of the “=” sign of each article of prior information are two real variables and a character variable, namely PIVAL, WEIGHT and OBGNME. The first of these is the “observed value” of the prior information equation. The second is the weight assigned to the article of prior information in the parameter estimation process. This can be zero if you wish (thereby removing the prior information equation from consideration); however it must not be negative.

The final item associated with each article of prior information must be the observation group to which the prior information equation belongs. Recall that each observation, and each prior information equation, cited in a PEST control file must be assigned to an observation group. In the course of implementing an inversion process, programs such as PESTPP calculate the contribution made to the objective function by each such observation group. The name of any observation group to which an item of prior information is assigned, must also be cited in the “observation groups” section of the PEST control file. As was discussed above, the name of an observation group must be twelve characters or less in length.

When adding prior information to a PEST control file, you should note that no two prior information equations should say the same thing. Thus the following pair of prior information lines is illegal.

pi1 2.0 \* log(par1) + 2.5 \* log(par2) - 3.5 \* log(par3) = 1.342 1.00 obgp1

pi2 4.0 \* log(par1) + 5.0 \* log(par2) - 7.0 \* log(par3) = 2.684 1.00 obgp2

If you wish to break a single prior information equation into more than one line, use the continuation character “&”. This must be placed at the beginning of each continuation line, separated from the item which follows it by a space. The line break must be placed between individual items of a prior information equation; not within an item. Thus the following lines convey the same information as does the first of the above pair of prior information lines.

pi1

& 2.0

& \*

& log(par1)

& +

& 2.5

& \*

& log(par2)

& -

& 3.5

& \*

& log(par3)

& =

& 1.342

& 1.00

& obgp1

However the following article of prior information is illegal because of the break between “log” and “par2":

pi1 2.0 \* log(par1) + 2.5 \* log

& (par2) - 3.5 \* log(par3) = 1.342 1.00 obgp1

## 4.16 Regularization Section

The regularization section of a PEST control file is optional. If PESTMODE is not set to “regularization”, it is redundant. If it is set to “regularization” and a “regularization” section is not provided, the PESTPP program (the only program of the PEST++ suite which uses this section) provides default values for the control variables that are featured in it. These variables are now described. To clarify the meanings of some of the terms that appear in the following explanation, see the description of PESTPP in chapter 6 of this manual. For more general information on regularization, see Doherty (2015).

#### PHIMLIM

This is the target measurement objective function; it is the measurement objective function that PESTPP “aims for” while keeping the regularization objective function as low as possible. During every iteration of the inversion process, PESTPP calculates a regularization weight factor that attempts to achieves this balance.

If you do not know how good a fit can be achieved between model outputs and the calibration dataset, set PHIMLIM very low (for example 1.0E-10) just to see how good a fit you can get between model outputs and field measurements. Meanwhile, the FRACPHIM control variable (see below) can be set to a value such as 0.1; this prevents regularization constraints on parameter values from being totally ignored on the way to achieving the best possible fit with the data. If it turns out that the fit between model outcomes and the calibration dataset is poor, this may indicate that the model needs refinement; by seeking the best possible fit with the calibration dataset you get to find this out early in the calibration process. However if the fit is good, then PHIMLIM can be set to a value 5 percent to 10 percent higher than the best measurement objective function attained during this PESTPP run, before running PESTPP again. Parameter values achieved during the next PESTPP run should be far more pleasing than those achieved during the first run; it is normally in achieving the last 5 percent to 10 percent reduction in the measurement objective function that parameters are assigned values that violate sensibility constraints as over-fitting occurs.

#### PHIMACCEPT

During each iteration, just after it has linearized the inverse problem through calculating the Jacobian matrix, and just before it begins calculation of the parameter upgrade vector, PESTPP calculates the optimal value of the regularization weight factor for that iteration. This is the value which, under the linearity assumption encapsulated in the Jacobian matrix, results in a parameter upgrade vector for which the measurement objective function is equal to PHIMLIM. However, due to the approximate nature of the linearity assumption, PESTPP may not be able to lower the measurement objective function to PHIMLIM in spite of the fact that it uses a number of different values for the Marquardt lambda in attempting to do so. If it cannot lower the measurement objective function to an acceptable level, it simply accepts the upgraded parameters, proceeds to the next iteration and tries again. However if it does succeed in lowering PHIMLIM to an acceptable level, or if it has succeeded in doing this on previous iterations, then PESTPP slightly alters its philosophy of choosing new Marquardt lambdas, in that it now attempts to lower the regularization objective function while maintaining the measurement objective below this acceptable level. This acceptable level is PHIMACCEPT; it should be set slightly higher than PHIMLIM in order to give PESTPP some “room to move” in its attempts to lower the regularization objective function while keeping the measurement objective function below, or close to, PHIMLIM. It needs this “room to move” because of the fact that it bases its calculations on a linearity assumption that is only approximately satisfied.

Normally PHIMACCEPT should be set between 2 percent and 5 percent greater than PHIMLIM.

#### FRACPHIM

The FRACPHIM variable allows you to set PHIMLIM very low (possibly lower than is achievable), but still retain the benefits of regularization.

If FRACPHIM is provided with a value of zero or less (or if this variable is omitted from the PEST control file), then it has no effect on the inversion process. However if FRACPHIM is provided with a value of between 0.0 and 1.0 (values of 1.0 or greater are illegal), then PESTPP calculates a new value for PHIMLIM at the beginning of each iteration of the inversion process. This value is calculated as the current value of the measurement objective function times FRACPHIM, or the user-supplied value of PHIMLIM, whichever is higher. Thus PESTPP always “aims for” a measurement objective function that is lower than the current one. However it does not pursue a target that is so low as to require the complete abandonment of regularization.

As well as adjusting the value of PHIMLIM during every iteration, PESTPP also adjusts the value of PHIMACCEPT. This adjustment is made such that, during every iteration, the ratio of PHIMACCEPT to PHIMLIM is the same as that supplied in the PEST control file.

The recommended value for FRACPHIM is 0.1.

#### WFINIT

WFINIT is the initial regularization weight factor. During every iteration of the inversion process PESTPP calculates a suitable regularization weight factor to employ during that iteration using an iterative, numerical solution procedure; its initial value when implementing this procedure for the first iteration is WFINIT. If there are many adjustable parameters, calculation of the regularization weight factor for the first iteration can be time-consuming if WFINIT is far from optimal. Hence if you have any idea of what the weight factor should be (for example from a previous PESTPP run), then you should provide WFINIT with this value. Otherwise simply set it to 1.0.

#### WFMIN, WFMAX

These are the minimum and maximum permissible values that the regularization weight factor is allowed to take. Normally settings of 1.0E-10 and 1.0E10 are suitable; settings of 1.0E-15 and 1.0E15 are normally fine as well. If PESTPP wishes to transgress these limits it will notify you of this. This normally indicates that regularization constraints are too weak to ensure inverse problem uniqueness (if the upper weight factor limit is encountered), or that measurement weights are too low (if the lower weight factor limit is encountered).

#### WFFAC, WFTOL

In order to calculate an appropriate regularization weight factor to use during any iteration of the inversion process, PESTPP uses an iterative procedure which begins at the value of the regularization weight factor calculated during the previous iteration; for the first iteration it uses WFINIT to start the procedure. In the process of finding the weight factor which, under the linearity assumption used in its calculation, will result in a measurement objective function of PHIMLIM, PESTPP first travels along a path of progressively increasing or decreasing weight factor (it decides which one of these alternatives to explore on the basis of the value of the current measurement objective function with respect to PHIMLIM). In undertaking this exploration, it either multiplies or divides the weight factor by WFFAC; it continues to do this until it has found two successive weight factors which lie on either side of the optimal weight factor for that iteration. Once it has done this, it uses Newton’s method to calculate the optimal weight factor, through a series of successive approximations. When two subsequent weight factors calculated in this way differ from each other by no more than a relative amount of WFTOL, the optimal weight factor is deemed to have been calculated.

Experience has shown that a suitable value for WFFAC is about 1.3; it must be greater than 1.0. WFTOL is best set to somewhere between 1.0E-3 and 1.0E-2. However if there are many adjustable parameters and PESTPP consumes a large amount of time in determining the optimal weight factor, a tolerance of somewhat higher than 1E-2 may prove suitable.

#### IREGADJ

PESTPP does not support the same range of values for IREGADJ as does PEST. When using PESTPP, IREGADJ should be set to 1 or 0; omitting it from the PEST control file has the same effect as setting it to 0. An IREGADJ setting of 1 instructs PESTPP to adjust relatively of weighting between regularization groups prior to calculating an overall, regularization weight factor. See section 6.2.3 for further details.

## 4.17 Control Variables for PEST++ Programs

Each of the programs comprising the PEST++ suite requires its own control variables. Unlike variables that control the operation of PEST, variables that are specific to PEST++ programs are not identified by their position in the PEST control file. Instead, they are introduced through a variable-specific keyword.

Any line in a PEST control file that begins with the character string “++” is ignored by PEST-suite programs, and by PESTCHEK. However programs of the PEST++ suite read these lines, expecting to find one or more keywords.

Figure 4.12 shows a PEST control file that includes the values of some PEST++ control variables (and a comment line). Wherever a PEST++ keyword is supplied, one or more values for the control variable that is associated with that keyword must follow it in brackets. Where more than one value is associated with a keyword, these values must be comma-delimited within the brackets. More than one keyword can be supplied on a “++” line. If so, they must be separated by one or more whitespace characters.

|  |
| --- |
| pcf  \* control data  restart estimation  5 19 2 2 3  2 3 single point  10.0 -3.0 0.3 0.03 10  10.0 10.0 0.001  0.1  50 0.005 4 4 0.005 4  1 1 1  \* parameter groups  ro relative 0.01 0.0 switch 2.0 parabolic  h relative 0.01 0.0 switch 2.0 parabolic  \* parameter data  ro1 fixed factor 0.5 .1 10 ro 1.0 0.0  ro2 log factor 5.0 .1 10 ro 1.0 0.0  ro3 tied factor 0.5 .1 10 ro 1.0 0.0  h1 none factor 2.0 .05 100 h 1.0 0.0  h2 log factor 5.0 .05 100 h 1.0 0.0  ro3 ro2  \* observation groups  obsgp1  obsgp2  prgp1  \* observation data  ar1 1.21038 1.0 obsgp1  ar2 1.51208 1.0 obsgp1  ar3 2.07204 1.0 obsgp1  ar4 2.94056 1.0 obsgp1  ar5 4.15787 1.0 obsgp1  ar6 5.7762 1.0 obsgp1  ar7 7.7894 1.0 obsgp1  ar8 9.99743 1.0 obsgp1  ar9 11.8307 1.0 obsgp2  ar10 12.3194 1.0 obsgp2  ar11 10.6003 1.0 obsgp2  ar12 7.00419 1.0 obsgp2  ar13 3.44391 1.0 obsgp2  ar14 1.58279 1.0 obsgp2  ar15 1.1038 1.0 obsgp2  ar16 1.03086 1.0 obsgp2  ar17 1.01318 1.0 obsgp2  ar18 1.00593 0.0 obsgp2  ar19 1.00272 0.0 obsgp2  \* model command line  model.bat  \* model inputoutput  ves1.tpl a\_model.in1  ves2.tpl a\_model.in2  \* model output  ves1.ins a\_model.ot1  ves2.ins a\_model.ot2  ves3.ins a\_model.ot3  \* prior information  pi1 1.0 \* h1 = 1.0 3.0 prgp1  pi2 1.0 \* log(ro2) + 1.0 \* log(h2) = 2.6026 2.0 prgp1  # This is a comment line  ++ forecasts(ar18,ar19) parcov(param.unc)  ++ lambdas(0.1, 1.0, 10,100)  ++ n\_iter\_base(-1)  ++ n\_iter\_super(4)  ++ base\_jacobian(pest.jco)  ++ par\_sigma\_range(6) |

Figure 4.12 A PEST control file which includes PEST++ control variables.

Values that are supplied with a keyword can be integer, real or text (for example filenames), this depending on the keyword. Text can be optionally surrounded by single or double quotes; this option becomes a necessity if a filename provided with a keyword includes blanks.

If a program of the PEST++ suite does not recognize a keyword, it simply ignores it. Hence a PEST control file can contain keywords that are pertinent to a number of members of the suite. The PEST++ program that is currently using the PEST control file only reads values for control variables that it recognizes.

# 5. Running PEST++ Programs

## 5.1 General

To simplify the following discussion, let PESTPP-XXX signify the name of a program belonging to the PEST++ suite. This can be any of the programs listed in table 1.1 except for PESTPP-WRK. As will be discussed, PESTPP-WRK is a specialist member of the PEST++ suite that can be used in conjunction with any of them whenever model runs are parallelized.

## 5.2 Model Runs in Serial

### 5.2.1 Concepts

Programs of the PEST++ suite have in common the fact that they run a model many times. These runs can be undertaken in serial or in parallel. Where a PEST++ program undertakes model runs in serial, it issues a command directly to the operating system whenever it requires that a model run be undertaken. The command which it issues is provided in the “model command line” section of the PEST control file.

On most occasions of its use, execution of PESTPP-XXX should be initiated from the folder in which a PEST control file resides. In many cases, this folder will also contain all files that are needed by the model. Once the model has run to completion, this folder may then contain all files that the model produces. For this simple setup there is no need to prefix template, instruction and model input/output filenames with relative or absolute paths when referencing them in the PEST control file. However where setup is not so simple, pathnames must be prefixed to model input and/or output files which do not reside in the PESTPP-XXX working folder.

If the model program resides in the folder from which PESTPP-XXX is run, or resides in a folder that is cited in the PATH environment variable, there is no need to prefix the model command with an absolute or relative pathname in the PEST control file. If the model is a batch or shell script, the same applies to executable programs which are cited in this script. However if this is not the case, then pathnames are required.

### 5.2.2 Running PESTPP-XXX

Where model runs are undertaken in serial, PESTPP-XXX is run using the command

pestpp-xxx *case*

where *case* is the filename base of the PEST control file. If you wish, you can include the *.pst* extension at the end of the name of the PEST control file.

PESTPP, a member of the PEST++ suite, can also be run using the command

pestpp *case /r*

The “/r” switch informs PESTPP that it must restart a previously halted run. It recommences execution of the previously terminated run either at the point where filling of the latest Jacobian matrix had been completed, or at the beginning of the current iteration, whichever is the most recent. Note that “/r” is case-insensitive; “/R” works just as well.

## 5.3 Model Runs in Parallel

### 5.3.1 Concepts

Tasks that are carried out by programs of the PEST++ suite require that a model be run many times. By undertaking these model runs in parallel, the time required for completion of an inversion or optimization task can be reduced considerably. Nowadays, most modellers have ready access to parallel computing facilities. Modern-day personal computers have multiple CPUs. Most offices have multiple computers connected to each other through an office network. Many modellers have access to a high performance computing cluster which may provide hundreds of cores. All modellers have access to the computing cloud.

Where model runs that are commissioned by a program of the PEST++ suite are parallelised, each instance of the model must run in its own working folder. It can therefore read its own input files and write its own output files with no interference from other instances of the model. Working folders for these model runs may be situated on one or many different computers. In each case, the computer which owns the folder is the one on which the model should be run. The system command to run the model is issued by a local supervising program named PESTPP-WRK. “WRK” stands for “worker”.

Execution of PESTPP-WRK should be initiated in all folders in which parallel model runs are to be undertaken. Hence as many instances of PESTPP-WRK must be initiated as there are folders which are set up for the undertaking of these runs. As stated above, these folders will normally be equipped with a complete set of model input files. Meanwhile, execution of the PESTPP-XXX program that is undertaking inversion, optimization, sensitivity or uncertainty analysis is initiated in its own working folder on its own computer. There is no reason why this folder cannot coincide with the folder of one of its workers.

In ways which will be described shortly, each instance of PESTPP-WRK establishes communication with PESTPP-XXX as soon as both of these programs are running. (It does not matter which of them is started first. Nor does it matter if workers are added belatedly to a computing pool, or are subtracted from that pool if their computers are needed for other tasks.) PESTPP-XXX is thus always aware of the workers that it has at its disposal. Furthermore, before it notifies PESTPP-XXX of its existence, each worker performs some local calculations in order to find out how fast is its local machine. It provides this information to PESTPP-XXX so that the latter is aware whether some of the machines to which it has access are faster than others. Note however that PESTPP-XXX updates its database of relative computing speeds as model runs are completed so that it always knows which worker can commission the fastest model run. (Model execution speed can be affected by other tasks that a worker’s computer may be carrying out, including that of undertaking other model runs under the control of other workers.)

When PESTPP-XXX wishes that a model run be carried out, it chooses one of its workers (i.e. one instance of PESTPP-WRK) to carry out that run. This is normally the worker that is operating on the fastest available machine. It then informs the worker of the values of the parameters that it would like the model to use for that particular run. If there is more than one model command line option, it also informs the worker which command must be used to run the model on this particular occasion. On receipt of this information, the local instance of PESTPP-WRK writes model input files using local copies of model template files. It then issues the system command to run the model. When the model has completed execution, PESTPP-WRK reads pertinent numbers from model output files using local copies of instruction files. Once it has read these numbers, it transmits them back to the PESTPP-XXX manager. Having completed its run supervisory tasks, PESTPP-WRK awaits a request from PESTPP-XXX to supervise another model run.

In order to write model input files and read model output files, each instance of PESTPP-WRK must have access to template files and instruction files. Normally copies of all template and instruction files that are listed in the PEST control file used by PESTPP-XXX are placed in the working folder of each PESTPP-WRK worker, together with files required by the model. As will be discussed shortly, PESTPP-WRK knows of the existence of these template and instruction files through an input file of its own that it must read on commencement of execution.

### 5.3.2 Manager to Worker Communication

PESTPP-XXX and PESTPP-WRK communicate with each other using the TCP/IP communications protocol. Where a worker resides on a different machine from that of PESTPP-XXX, network management must permit this kind of communication between them. If the PESTPP-XXX machine can be “pinged” from the worker’s machine, and if the worker’s machine can be “pinged” from the PESTPP-XXX machine, then you have this permission.

When PESTPP-XXX commences execution, it opens a so-called “port”. Workers must be informed of the IP address or hostname of the machine on which the manager is operating, and of the number of this port (see below). In contrast, PESTPP-XXX does not need to know the locations of its workers. It knows that a worker exists through the TCP/IP connection which the worker initiates when it commences execution. Acceptance of this connection is sufficient for the communications pathway to exist. Then, whenever PESTPP-XXX requires that a worker carry out a model run, it sends parameter values to that worker through this connection. When the model run that is supervised by the worker is complete, PESTPP-XXX receives the values of model-calculated observations read from model output files by the worker through the same connection. It is oblivious to the location of the worker, and hence of the computer and folder in which these model runs are being carried out.

### 5.3.3 Running PESTPP-XXX and PESTPP-WRK

When model runs are parallelized, execution of PESTPP-XXX must be initiated using the following command:

pestpp-xxx case /h :port

As usual, *case* is the filename base of the PEST control file. *port* in the above command signifies a TCP/IP port number that has not already been opened by another program on the machine on which PESTPP-XXX is running. Choosing a high number such as 4000 or above generally forestalls conflicts. “/h” is case-insensitive; “/H” is also acceptable.

When it is run using the above command, PESTPP-XXX opens the nominated port for communication, and then “listens” on that port for any workers that try to connect to it. Once a connection has been established by a particular worker, communication between PESTPP-XXX and that worker can take place freely.

Execution of each instance of PESTPP-WRK must be initiated as follows:

pestpp-wrk case /h textstring:port

Options for *case* appearing in the above command are discussed shortly. *textstring* can be any of the following:

* the IP version 4 address of the machine on which PESTPP-XXX is running;
* the IP version 6 address of the machine on which PESTPP-XXX is running;
* the UNC (uniform naming convention) hostname of the machine on which PESTPP-XXX is running.

If you are running PESTPP-XXX together with multiple instances of PESTPP-WRK on the same machine, and if that machine is not connected to the internet, then the last of the above three options is the only one available to you. Type

hostname

in a command line window to ascertain the hostname of a machine.

If, for some reason, a worker ceases execution, or the computer on which it resides loses its connection with PESTPP-XXX, it should be re-started using the above command. A worker does not have to be restarted using the “/r” switch as its tasks are repetitive and simple, namely to receive parameters, run the model, and then send model outputs to PESTPP-XXX.

### 5.3.4 Worker Input File

PESTPP-WRK must be informed of the template and instruction files which it must use to write and read model input and output files. It must also be informed of the names of the model input and output files that it must write and read using these template and instruction files. In addition to these, it must be informed of the command(s) that it must use to run the model. This information must be provided to it through its own input file. The name of this file is provided on its command line. (Actually, there is no physical reason why a worker cannot receive this information from the PESTPP-XXX manager. However security is maintained, especially in relation to the model command, if this information is read locally.)

The input file for PESTPP-WRK (*case* in the above command) can be a PEST control file – the same PEST control file as that used by PESTPP-XXX. However PESTPP-WRK only reads the following sections from this file:

* model input,
* model output,
* model command line.

If PESTPP-WRK is supplied with a file that contains only these sections, then it can still do its job. This file does not need to be a PEST control file. Furthermore, the file can contain information other than that appearing in these three sections. Thus PESTPP-WRK can serve as a worker for any program which uses the PANTHER run manager, regardless of whether this program belongs to the PEST++ suite or not, and regardless of whether this program receives its information from a PEST control file. If it does not employ a PEST control file, then any of the above sections of its input file can be terminated with a line that begins with the “\*” character. PESTPP-WRK interprets a line which begins with this character as the end of a section. The above sections can be provided in any order.

### 5.3.5 Run Management Record File

PESTPP-XXX records all communications between it and its workers in a run management record file. This file is named *case.rmr* where *case* is the filename base of the PEST control file.

### 5.3.6 Run Management Control Variables

A number of PESTPP-XXX control variables are used to control parallel run management. These are now described.

As has already been discussed, on commencement of execution PESTPP-XXX reads a PEST control file. Variables in a PEST control file that are specific to programs of the PEST++ suite are provided on lines that begin with “++”. Those variables which control parallel run management are passed directly to the PANTHER parallel run manager used by all members of the PEST++ family; their action is thus the same, regardless of the PEST++ program which reads the PEST control file.

As was discussed in section 4.17, the value of a PEST++ control variable must be supplied as an argument to a keyword. It must be surrounded by brackets immediately following the keyword that announces its presence.

#### overdue\_resched\_fac()

Suppose that the value of *overdue\_resched\_fac()* is supplied as *r*, where *r* is a real number. If the run manager is still awaiting completion of a simulation that has already been running for more than *r* times the average model run time (calculated on the basis of all previous successful runs), then it will ask other workers to start the same run if they are available. The maximum number of concurrent runs is set by the *max\_run\_fail()* control variable (this variable is further discussed below). This action is taken as a precautionary measure. It guards against the possibility that the computer on which the delayed model run executing place has become busy with other tasks. A competition is started between workers. The first of the concurrent model runs to finish successfully is accepted; meanwhile, the remaining concurrent model runs are terminated by PESTPP-XXX. Note, however, that if *max\_run\_fail()* is set to 1, concurrent model run scheduling is not undertaken.

The value supplied for the *overdue\_resched\_fac()* variable must be a real number that is 1.0 or greater. Its default value is 1.15; this is the value which the run manager uses if you do not supply a value for this control variable yourself. A value greater than 1.15 may be desirable in circumstances where the time required for a model run is sensitive to parameter values.

#### overdue\_giveup\_fac()

The value of *overdue\_giveup\_fac()* is also a real number. Suppose that it is *r*.If an overdue simulation has already taken *r* times the average model run time but is still not complete, then the run manager declares the run to have failed. It instructs the worker that is supervising the run to kill it. The worker is then free to supervise another model run based on another set of parameters. Handling of run failure by programs of the PEST++ suite is discussed in more detail below. The default value for *overdue\_giveup\_fac()* is 100.0 for all programs in the suite except for PESTPP-IES, which uses a default value of 2.0.

#### overdue\_giveup\_minutes()

This control variable can be used to place a “hard” upper bound on the acceptable model run time. The value supplied for *overdue\_giveup\_minutes()*must be a real number. Suppose that it is *r*. Then a model run is deemed to have failed if it has been executing for more than *r* minutes. This model run time criterion works as an “or” condition with *overdue\_giveup\_fac()*; if the duration of a given model run exceeds *overdue\_giveup\_minutes()* or *overdue\_giveup\_fac()* times the average model run time, it is marked as a failure. The default value for *overdue\_giveup\_fac()* is 1.0E+30, this effectively disabling it.

*max\_run\_fail()*

A model run is declared to have failed if PESTPP-WRK cannot read one or more of the model’s output files. Excessive execution time can also be deemed as model run failure, as discussed above. Run failure is normally an outcome of model hostility to the set of parameters with which it was provided. However sometimes it can be caused by external problems that are not the model’s fault, for example operating system quirks or network failure. Hence, the parallel run manager can be instructed to attempt a failed model run again using another worker. However only a certain number of model run failures can be tolerated. This “certain number” (an integer) must be supplied as the value of the *max\_run\_fail()* control variable. A value of 1 is used to request no repetition of model runs. The default value is 3 is used for all programs except PESTPP-IES, which uses a default value of 1.

As discussed above, *max\_run\_fail()* is also used to control the number of concurrent runs that are allowed when the *overdue\_resched\_fac()* criterion is exceeded.

### 5.3.7 An Alternative Worker Option

At the time of writing, all members of the PEST++ suite can actually be used as both manager and worker. Thus, in the above description of parallelized model runs, PESTPP-XXX can be used in place of PESTPP-WRK. However if PESTPP-XXX is used in this capacity, it should be provided with the current PEST control file as its input file rather than an abbreviated worker input file.

As discussed above, a useful feature of PESTPP-WRK is that it can be used as a general worker by programs that are not part of the PEST++ suite, but which employ the PANTHER parallel run manager. On the other hand, use of PESTPP-XXX as a worker for itself has the advantage that a PESTPP-XXX working folder can be copied to folders residing on working machines without needing to add the PESTPP-WRK executable to this folder, as long as this folder already contains the PESTPP-XXX executable.

## 5.4 Run Book-Keeping Files

After running a program of the PEST++ suite, you may notice a number of (possibly large) files in the folder from which it was run. These are *case.rns*, *case.rnu* and *case.rnj*, where *case* is the filename base of the PEST control file. These are binary files that are used for temporary storage of run results, and sometimes copies of these results. They contain information that assists in parallel run management, and that facilitates restart of an interrupted PEST++ run.

# 6. PESTPP

## 6.1 Introduction

PESTPP was the original member of the PEST++ suite. As is described in an overview document that accompanies this manual (Doherty et al, 2018), the intention behind its creation was to reproduce much of the functionality of PEST in code that is modular, object oriented and supportive of collaborative programming. At the same time, it was hoped that certain aspects of PEST’s performance could be improved by taking advantage of the “new slate” that was offered by PESTPP.

As the name suggests, PESTPP is written in C++. Version 4 of PESTPP uses the modular, general purpose, PANTHER parallel run manager.

Like PEST, PESTPP undertakes highly parameterized inversion. However, if requested, it can also undertake global optimization using the differential evolution (DE) method. The task that it implements on any given run is determined by the values that are supplied for pertinent control variables.

## 6.2 Highly Parameterized Inversion

### 6.2.1 Basic Equations

When used to undertake highly parameterized inversion, PESTPP implements theory and methodologies that are programmed into PEST. However many implementation details have been improved. In order to explain these details, it is necessary to present some theory. This theory employs matrices and vectors. These are used to describe the linearized inverse problem on which so-called “gradient methods” are based. Through repeated linearization of the inverse problem over successive iterations, these methods achieve their purpose of model calibration, notwithstanding the nonlinear relationship that exists between model outputs and model parameters. Full details of this theory are presented in Doherty (2015).

Let the matrix **Z** represent the (linearized) action of the model on its parameters; let these parameters be characterized by the vector **k**. The action of a model under calibration conditions can be described by the following equation:

**h** = **Zk** + **ε** (6.1)

In equation 6.1, the vector **h** represents observations comprising the calibration dataset, while **ε** represents measurement noise associated with these observations.

Tikhonov regularization may be introduced to an inverse problem to promulgate parameter uniqueness and sensibility. This can be comprised of observations and/or prior information that pertain directly to parameters and/or to relationships between parameters. For PEST and PESTPP, an observation or prior information equation is classed as “regularization” if the following conditions are met:

* The PESTMODE control variable is set to “regularization”;
* The observation group to which the observation or prior information equation belongs has a name that begins with “regul”.

If regularization constraints are referenced explicitly in equation 6.1, it becomes

(6.2)

In equation 6.2, the elements of **h**r are “regularization observations” (which can include the “observed” values of prior information equations). They normally penalize departure from a preferred parameter condition. This condition may be comprised of a set of values that parameters should adopt unless there is information to the contrary in the calibration dataset. **ε**r is the “noise” associated with this condition; it reflects the tolerance of a modeller for departures from this condition incurred by the necessity for model outputs to fit a calibration dataset. Generally a modeller does not know this tolerance ahead of the calibration process. Instead, he/she knows his/her tolerance for model-to-measurement misfit, and is prepared to adjust his/her tolerance for parameter departures from preferred values (at least to some extent) on that basis.

We employ the symbol **k** to designate the parameter set (hopefully of minimum error variance) estimated through the inversion process. **k** can be calculated using the following equation.

**k** = [**ZQZ**t + *μ*2**Z**r**Q**r­**Z**tr]-1[**Z**t**Qh** + *μ*2**Z**r**Q**r**h**r] (6.3)

In equation 6.3 **Q** is a weight matrix ascribed to measurement noise while **Q**r is a weight matrix ascribed to “regularization noise”. Ideally

(6.4a)

(6.4b)

where C() signifies a covariance matrix. The *μ*2 term in equation 6.3 accommodates flexibility in the strength with which regularization constraints are enforced. Like PEST, PESTPP is given permission to calculate the value of *μ*2 itself. It is referred to herein as the “regularization weight factor”.

In practice, the Jacobian matrix **J** replaces the **Z** matrix of equation (6.3). This is the matrix of partial derivatives of model outputs with respect to parameter values. As such, it constitutes a local linearization of the action of the model. As was discussed in previous chapters of this manual, the Jacobian matrix is filled through finite parameter differencing. Where parameters are large in number, this can be by far the most laborious component of the inversion process.

Because calculation of **k** using equation 6.3 relies on a linearity assumption that is invoked when **J** replaces **Z**, solution of the overall inverse problem of model calibration becomes an iterative procedure. Once **k** is calculated using 6.3, a new **J** is calculated using that **k**. An improved **k** is then calculated. When **k** stops changing, or when model-to-measurement misfit stops improving, the inversion process is terminated.

Nonlinear model behaviour is also accommodated by introducing a so-called Marquardt lambda to equation 6.3. The equation becomes

**k** = [**ZQZ**t + *μ*2**Z**r**Q**r­**Z**tr + *λ***I**]-1[**Z**t**Qh** + *μ*2**Z**r**Q**r**h**r] (6.5)

A nonzero lambda increases diagonal elements of the matrix [**ZQZ**t + *μ*2**Z**r**Q**r­**Z**tr + *λ***I**]. It is apparent from equation 6.5 that this matrix must be inverted. Increasing its diagonal elements has two effects. Firstly, it contributes to invertibility of the matrix. Secondly, it migrates the direction of parameter improvement so that it is more aligned with the gradient (with respect to parameters **k**) of the objective function which is used to characterize model-to-measurement and parameter-to-preferred-value misfit. This increases the efficiency of early iterations of the inversion process when implemented in conjunction with a nonlinear model.

The objective function that is employed by PEST and PESTPP (designated herein as Փ) represents model-to-measurement misfit as a sum of weighted squared discrepancies between model outputs and field observations **h**, and between model parameters and regularization observations **h**r. It is calculated using the following equation:

Փ = (**h** – **Zk**)t**Q**(**h** – **Zk**) + *μ*2(**h**r – **Z**r**k**)t**Q**r(**h**r – **Z**r**k**) (6.6)

That is

Փ = Փm + *μ*2Փr (6.7)

where Փm is referred to as the “measurement objective function” and Փr is referred to as the “regularization objective function”.

If PESTPP is not run in “regularization” mode then, or course, regularization terms in the above equations are omitted. This does not preclude the use of prior information in the inversion process; however the ability of PESTPP to adjust the importance that it gives to prior information through the regularization weight factor *μ*2 is lost.

### 6.2.2 Choosing the Regularization Weight Factor

The value that is assigned to *μ*2 strongly influences the inversion process. If it is too small, then overfitting may occur. If it is too large, then the fit that is attained with the calibration dataset may not be satisfactory.

When using PESTPP (and PEST), a user does not have to choose a value for *μ*2 him/herself. Instead, he/she must choose a value for the target measurement objective function. This is the PEST variable PHIMLIM that appears in the “regularization” section of the PEST control file. PESTPP then calculates the regularization weight factor itself. This is done through an iterative procedure that is controlled by the values supplied for the PHIMACCEPT, WFINIT, WFMIN, WFMAX, WFFAC and WFTOL regularization control variables that are described in section 4.16 of this document. These play the same roles for PESTPP that they do for PEST.

As was discussed in section 4.16, the level of fit that can be achieved with a calibration dataset is often difficult to predict. To accommodate this, it may be useful to endow PHIMLIM with a very low value (for example 1.0E-10). This allows you to find out just how good a fit you can get with the calibration dataset, possibly at the cost of over-fitting that dataset. At the same time, the value of the FRACPHIM control variable should be set to 0.1. If FRACPHIM is set to a value greater than zero, PESTPP adjusts the target measurement objective function internally to be FRACPHIM times the current value of the actual measurement objective function. Hence regularization is operative, notwithstanding the pursuit of what may turn out to be too good a fit with the calibration dataset. Then, on a subsequent PESTPP run, PHIMLIM should be set about five percent greater than the best measurement objective function attained through the preceding “range finder” run.

PESTPP provides an alternative mechanism for balancing the measurement and regularization objective functions. A value can be provided for the *reg\_frac()* control variable on a line beginning with “++” placed anywhere within a PEST control file. Suppose that this is supplied as *reg\_frac(r)*, where *r* is a real number less than 1.0. Then, at the beginning of each iteration of the inversion process, PESTPP adjusts *μ*2 such that

Փr = *r*(Փm + Փr) (6.8)

A suitable value for *reg\_frac* is 0.1; values between 0.05 and 0.25 can also work well. Where there are many parameters and many regularization observations, use of this strategy can result in considerable savings in computation time as PESTPP does not pursue the expensive iterative procedure of calculating *μ*2 from PHIMLIM.

### 6.2.3 Inter-Regularization Group Weighting

Experience gained in using PEST suggests that a Tikhonov-regularised inversion process can benefit from automatic balancing of weights between different regularization groups (i.e. observation groups that contain regularisation observations). Ideally, a higher weight factor should be applied to regularization groups that feature parameters that are mildly influenced by the calibration dataset than to groups which feature parameters whose values are heavily influenced by the calibration dataset. Not only can this ensure that the former set of parameters do not depart much from their preferred values (which should also be their initial values); it can contribute to numerical stability of the inversion process.

The IREGADJ variable supplied in the “regularization” section of a PEST control file controls this aspect of the regularized inversion process implemented by PESTPP. If IREGADJ is set to zero, or omitted from the PEST control file, then no inter-regularization group weight factor adjustment takes place. Alternatively, if it is set to 1, then weight factor adjustment is implemented. Note however that, unlike PEST, PESTPP does not modify its weights balancing behaviour if the value supplied for IREGADJ is greater than 1; it simply reduces it internally to 1.

For a particular observation or prior information equation (let us specify its index as *j*), the “composite observation sensitivity” is defined as

 (6.9)

where *m* is the number of adjustable parameters featured in the PEST control file. Equation 6.9 specifies that the composite sensitivity of observation *j* is the magnitude of the *j*th row of the Jacobian matrix multiplied by the weight associated with that observation; this magnitude is then divided by the number of adjustable parameters. It is thus a measure of the sensitivity of the model output corresponding to that observation to all parameters involved in the inversion process. The total composite observation sensitivity of a particular regularization group can be calculated by summing the composite sensitivities of all observations or prior information equations that comprise the group.

If IREGADJ is set to 1, PESTPP multiplies the weights pertaining to all members of each regularization group by a group-specific factor. This factor is chosen so that, after this operation has been performed, the total composite sensitivities of all regularization groups are the same.

### 6.2.4 Choosing Values for the Marquardt Lambda

In contrast to PEST, PESTPP does not use the control variables specified on the fifth line of the “control data” section of the PEST control file to govern how it chooses Marquardt lambdas. Instead, it receives Marquardt lambda control information from PEST++ control variables. Two of these are *lambdas()* and *lambda\_scale\_fac()*. Default settings for these variables are as follows:

*lambdas(0.1, 1, 10,100,100)*

*lambda\_scale\_fac(0.9, 0.8, 0.7, 0.5)*

As is apparent, more than one value can be associated with each of these control variables. Values supplied for *lambdas()* specify the Marquardt lambdas that PESTPP must use during each iteration of the inversion process. Then, if values are supplied for *lambda\_scale\_fac()*, PESTPP calculates multiple upgraded parameter sets for each Marquardt lambda. These sets are those that correspond to various fractional lengths along the upgrade vector that is calculated on the basis of the Marquardt lambda alone, the latter corresponding to a *lambda\_scale\_fac()* value of 1.0.

If only a single value is supplied for *lambdas()*, then this value of lambda is used in all iterations of the inversion process. In contrast, if more than one value is supplied for *lambdas()*, then PESTPP expands the user-supplied lambda list over the course of the inversion process. The expanded list includes other lambda values that are above and/or below those already comprising the list. A lambda is added to the list if the best parameters obtained during any particular iteration of the inversion process were calculated using an end member of the current list. The value chosen for a new lambda ensures that the previous best lambda is bracketed by members of the expanded list.

The *upgrade\_augment()* control variable provides another means through which the lambda list can be tailored to meet the changing needs of the inversion process. If *upgrade\_augment()* is set to *true*, then the lambda list is augmented at the end of every iteration of the inversion process by including lambda values that are equal to twice and half that of the most successful lambda for that iteration. *upgrade\_augment()* also instigates additional upgrade vector calculations when upgraded parameters encounter their bounds. Where model runs are undertaken in a highly parallelized environment, the additional model runs that are employed to test upgrades in this manner may contribute little to the overall inversion time if they are done on computing cores that are otherwise idle because they are waiting for the next occasion on which a Jacobian matrix must be filled.

### 6.2.5 Singular Value Decomposition

In order to simplify the following equations, it is assumed that regularization observations and/or prior information equations are assimilated into the rows of **Z** using a weight factor that has already been evaluated.

If the Marquardt lambda is ignored, equation 6.5 becomes

**k** = (**Z**t**QZ**)-1**Z**t**Qh** (6.10)

This equation cannot be employed if an inverse problem is nonunique, or borders on nonuniqueness despite inclusion of Tikhonov regularization. Under these circumstances **Z**t**QZ** is a singular or near-singular matrix. As such, it cannot be inverted, or can be inverted only with a high potential for numerical error. Attempts to solve an inverse problem through direct use of equation 6.10 under these circumstances can lead to numerical instability at best, and cessation of progress of the inversion process at worst. However while these problems may afflict PEST if using its default solver (that has been in place since its inception), they will never afflict programs of the PEST++ suite. This is because they employ singular value decomposition (or methods closely related to it) for solution of the inverse problem.

A matrix has a null space if there exists at least one non-zero vector δ**k** for which

**0** = **Z**δ**k** (6.11)

By adding equation 6.10 to 6.11, it is readily apparent that the existence of a null space promulgates nonuniqueness of solution of an inverse problem; if **k** is a solution to that problem, then so is **k**+δ**k**.

Like any matrix, the matrix **Q**½**Z** can, through singular value decomposition, be expressed as

**Q**½**Z** = **USV**t (6.12)

For convenience of the following discussion, let us suppose that **Q**½**Z** has more columns than rows. It will definitely have a null space under these circumstances. Equation 6.12 can then be expanded as

**Q**½**Z** = **US**1**V**t1 + **US**2**V**t2 (6.13)

where **S**1 contains only positive singular values and **S**2 contains singular values which are zero or near-zero. The columns of **V**2 then span the (effective) null space of **Q**½**Z** while the columns of **V**1 span its orthogonal complement which we describe herein as “the solution space”.

A unique solution of minimum error variance to the inverse problem posed by equation 6.1 can then be calculated as

**k** = **V**1**S**-11**U**t **Q**½**h** (6.14)

Numerical stability of the solution process implemented through equation 6.14 is guaranteed because the only matrix that needs to be inverted is **S**1; this has positive-valued diagonal entries and zero-valued off-diagonal entries.

It is easy to show that singular value decomposition of **Z**t**QZ** of equation 6.10 leads to

**Z**t**QZ** = **VS**2**V**t (6.15)

where **V** and **S** in equation 6.15 are the same as those of equation 6.12. The same solution **k** to the inverse problem as that obtained through equation 6.14 can then be calculated as

**k** = (**V**1**S**-21**V**t1)**Z**t**Qh** (6.16)

Like the solution calculated using equation 6.14, the solution to the inverse problem calculated using 6.16 is numerically stable regardless of the ill-posedness of the inverse problem. This is because the solution is restricted to the orthogonal complement of the null space of **Q**½**Z**, and because no singular or near-singular matrices need to be inverted in obtaining this solution. While the solution is not unique, it is of minimized error variance. See Doherty (2015) for details.

As stated above, PESTPP uses singular value decomposition, or variants thereof, to solve the inverse problem that is defined through the PEST control file with which it is provided. Both of the above options are available. That is, singular value decomposition can be undertaken on **Q**½**Z** or on **Z**t**QZ**. (Actually, although not discussed in the present sub-section, the Marquardt lambda is included in these matrices before they are decomposed in order to enhance inversion performance in nonlinear contexts.)

Two PEST++ control variables govern the way in which singular value decomposition is undertaken by PESTPP. These are *mat\_inv()* and *svd\_pack()*. *mat\_inv()* can take either of two possible values, both of which are supplied as text strings. These are “jtqj” and “q1/2j”. In the former case, singular value decomposition users equations 6.15 and 6.16. Alternatively, if *mat\_inv()* is set to “q1/2j”, singular value decomposition is applied according to equation 6.12.

Three choices are available for the *svd\_pack()* control variable. These select the numerical library that is employed for implementing singular value decomposition. Options are as follows.

* “eigen”. This uses a Jacobi method supplied with the Eigen library. Eigen is a C++ template library for linear algebra; see <http://eigen.tuxfamily.org>
* “propack”. This provides a fast iterative Lanczos type solution scheme. See <http://sun.stanford.edu/~rmunk/PROPACK/> This is not unlike the LSQR solver provided with PEST. However the singular vectors that emerge from this process (i.e. the vectors which comprise columns of the **V**1 matrix) are of superior quality.
* “redsvd”. This uses the redsvd library available from <https://github.com/cequencer/redsvd>. Randomized methods are employed to undertake singular value decomposition. These are very fast; where matrices are large and sparse, their performance is outstanding.

The default values for *mat\_inv()* and *svd\_pack()* are “qtqj” and “redsvd”. These are the methods which PESTPP uses if you do not supply values for these control variables.

Two variables that are recorded in the PEST control file can also affect PESTPP’s deployment of singular value decomposition to solve an inverse problem. These are the MAXSING and EIGTHRESH variables featured in the “singular value decomposition” section of that file. If a “singular value decomposition” section is not provided in the PEST control file that is read by PESTPP, then PESTPP assigns MAXSING a value that is equal to the number of adjustable parameter that define the current inverse problem; at the same time, EIGTHRESH is assigned a value of 1.0E-7. (Recall from section 4.7 that NUMSING and EIGTHRESH are used to define the singular value truncation point.) However if the PEST control file includes a “singular value decomposition” section, then PESTPP employs values for MAXSING and EIGTHRESH that it reads from this file.

### 6.2.6 SVD-Assist

Use of PEST’s “SVD-assist” methodology can promulgate significant increases in the numerical efficiency of highly parameterized inversion. In implementing this methodology, PEST estimates the values of so-called “super parameters” in place of the parameters themselves.

From equation 6.14, the minimum error variance solution **k** to an ill-posed inverse problem can be written as:

**k** = **V**1**p** (6.17)

In equation 6.17 the elements *pi* of **p** are the coefficients by which the vectors **v***i* comprising the columns of **V**1 are multiplied before these columns are added to achieve **k**. Where an inverse problem is ill-posed, the dimensionality of the calibration solution space may be far smaller than the number of elements of **k** (i.e. the dimensionality of parameter space). The same therefore holds for the number of elements comprising the vector **p**. If these elements are estimated directly, finite differences can be taken with respect to them instead of with respect to base parameters comprising the elements of **k**. Filling of the super parameter Jacobian matrix therefore requires only a relatively small number of model runs. Once **p** has been estimated, **k** can be obtained using equation 6.17.

It is apparent, therefore, that estimation of super parameters can reduce the computational burden of highly parameterized inversion considerably. However a Jacobian matrix based on the full parameter set **k** must be calculated prior to commencement of the super parameter inversion process so that **V**1 can be obtained through singular value decomposition of that matrix.

Unfortunately, the large computational savings accrued through SVD-assisted inversion are accompanied by a number of costs. Chief among these is that, for a nonlinear model, the partitioning of parameter space into solution and null spaces based on singular decomposition of a full Jacobian matrix calculated on the basis of initial parameter values may not remain valid as parameter values change. Hence, as the super parameter inversion process progresses, it may become necessary to recalculate a full Jacobian matrix so that super parameters can be re-defined. For moderately to highly nonlinear models, super parameter redefinition may be required every few iterations. With intermittent super parameter redefinition, model run savings accrued through SVD-assisted inversion may still be considerable; however they will not be as great as for a linear model where re-computation of a full Jacobian matrix is never required.

A second problem that is associated with SVD-assisted inversion using PEST is that of inconvenience. The SVDAPREP utility must be employed to construct a special PEST input dataset based on super parameters. If re-definition of super parameters is required after a few iterations of an inversion process because of model nonlinearity, then SVDAPREP must be run again.

PESTPP eradicates the second of these problems completely. There is no need to construct a PEST input dataset based on super parameters, as this is all done behind the scenes. Furthermore, PESTPP can be instructed to undertake intermittent re-construction of a full-scale Jacobian matrix, and subsequent redefinition of super parameters automatically and unattended.

Four PEST++ control variables govern the operation of SVD-assisted inversion as undertaken by PESTPP. If any of these are present, SVD-assisted inversion is implemented; default values are supplied for any of these control variables that a user fails to supply.

The number of super parameters to estimate can be set using either or both of the *max\_n\_super()* and ­*super\_eigthresh()* control variables. The value supplied for *max\_n\_super* must be an integer greater than zero. This sets an upper limit on the number of super parameters to employ. *super\_eigthresh()* performs a similar role to the PEST EIGTHRESH variable. The number of estimated super parameters is set by the singular value index at which the ratio of the corresponding singular value of **J**t**QJ** to the maximum singular value of this matrix is equal to the user-supplied value of *super\_eigthresh()*. The default value for *super\_eigthresh()* is 1.0E‑8. The default value for *max\_n\_super()* is the number of adjustable parameters so that the number of super parameters is determined by the value of *super\_eigthresh()*.

As was mentioned above, PESTPP allows mixing of inversion iterations based on base parameters and super parameters. Where an iteration is used to estimate base parameters the Jacobian matrix that is computed during that iteration is used to define super parameters for ensuing iterations in which the latter are estimated; however base parameters are adjusted in any iteration in which a base Jacobian matrix is calculated before the ensuing super parameter iterations are commenced. The *n\_iter\_base()* control variable sets the number of base parameter iterations that are done in succession. These are followed by *n\_iter\_super()* super parameter iterations. The cycle is then repeated.

A special setting for *n\_iter\_base()* instructs PESTPP to vary from this behavior. If *n\_iter\_base()* is set to -1, then PESTPP carries out only one base parameter iteration. This comprises the first iteration of the inversion process. Furthermore, it does not upgrade base parameters using this Jacobian matrix before proceeding to the next iteration (which is a super parameter iteration). Instead the Jacobian matrix is used only for definition of super parameters; parameter upgrades are restricted to super parameter iterations. (This is the same behavior as that undertaken by PEST when it implements SVD-assisted inversion using a super parameter PEST control file constructed by SVDAPREP).

Two other aspects of PESTPP’s behavior in undertaking SVD-assisted inversion are worth mentioning.

Like PEST, PESTPP writes a parameter value file (i.e. a PAR) file at the end of every iteration of the inversion process, this containing best parameters achieved to date. (The filename base of this file is the same as that of the PEST control file.) Unlike PEST however, PESTPP does not need to write a separate BPA file containing best parameters. This is because the PAR file that it records when undertaking super parameter iterations does not contain best-fit values of super parameters. Instead it contains the best values of base parameters. Nevertheless, for the sake of conformity with PEST, it also records best parameters in a BPA file (as well as in a file named *case.parb* whose contents are identical to those of *case.bpa* and *case.par*).

The handling of Jacobian matrix files (i.e. JCO files) is somewhat different. PESTPP records a Jacobian matrix file at the completion of every iteration of the inversion process. For those iterations in which it adjusts super parameters, this file is named *case.jco*, where *case* is the filename base of the PEST control file. The elements of this Jacobian matrix are comprised of partial derivatives of model outputs with respect to super parameters. For those iterations in which it computes sensitivities with respect to base parameters this file is named *case.jcb*. The elements of this Jacobian matrix are comprised of partial derivatives of model outputs with respect to native (i.e. base) model parameters.

Note, however, that if PESTPP is not undertaking SVD-assisted inversion, then the JCO file that is recorded at the end of each iteration of the inversion process is named *case.jco*. Under these circumstances, this file contains sensitivities with respect to base model parameters.

### 6.2.7 Expediting the First Iteration

In the normal course of events, PESTPP commences an inversion process by running the model in order to determine the value of the objective function based on initial parameter values. In doing this, it also determines the reference values of all model outputs for use in finite difference derivatives calculation. It then commences the long process of filling the Jacobian matrix. As has been explained, this requires at least as many model runs as there are adjustable parameters.

If model runs are parallelized (see section 5.3), then the first batch of parallelized model runs commissioned by PESTPP actually includes the initial model run, as well as those required for filling of the initial Jacobian matrix. However if this initial model run has already been completed, then PESTPP can use the outcomes of this already-completed run rather than having to repeat it at the start of the inversion process. This action is instigated through use of the *hotstart\_resfile()* control variable. The argument for this variable is the name of the residuals file (see below) which PESTPP recorded when it completed the initial model run. As is explained elsewhere in this manual, PESTPP can be run with the NOPTMAX termination control variable set to 0 specifically to undertake a single model run for the purpose of writing this (and other) files. A residuals file is named *case.res* where *case* is the filename base of the PEST control file.

Significant savings can also be made by employing an already-calculated Jacobian matrix for the first iteration of an inversion process. The *base\_jacobian()* control variable instructs PESTPP to take this action. The value of this variable is the name of a Jacobian matrix (i.e. JCO) file. This file is named *case.jco* or *case.jcb* where *case* is the filename base of the PEST control file on which its calculation was based. This file can be produced by running PESTPP with the NOPTMAX control variable set to -1 or -2. (NOPTMAX is the first variable appearing on the eighth line of the “control data” section of a PEST control file.)

If both of the *hotstart\_resfile()* and *base\_jacobian()* options are selected at the same time, PESTPP does not need to run the model at all prior to calculating and testing parameter upgrades. This can sometimes be useful when fine-tuning PESTPP settings for optimal inversion performance.

### 6.2.8 Linear Analysis

A Jacobian matrix calculated by PESTPP can be used as a basis for linear parameter and predictive uncertainty analysis. This type of analysis is often referred to as “FOSM” (for “first order second moment”) analysis. The equations used by PESTPP for implementation of FOSM analysis are derived from Bayes equation. They are outlined by Fienen et al (2010) and Doherty (2015). These same equations form the basis for analyses undertaken by the PEST PREDUNC suite of utility programs and by the PyEMU library.

The integrity of FOSM analysis is based on the following assumptions:

* The relationship between model outputs and parameters is linear;
* Prior parameter uncertainties (and hence posterior parameter uncertainties) are described by multiGaussian probability distributions;
* Measurement noise is also Gaussian.

When implemented by PESTPP an additional assumption is made. It is that the standard deviation of measurement noise associated with each observation is proportional to the inverse of the weight assigned to that observation in the PEST control file which defines the inverse problem. The proportionality constant is such that the (measurement) objective function is equal to the number of non-zero-weighted observations featured in the file.

FOSM analysis is implemented by PESTPP if the *uncertainty()* control variable is set to *true*. In implementing FOSM analysis, PESTPP ignores regularization observations and prior information equations. However this does not prevent a user from denoting prior correlation between parameters; this can be supplied through a covariance matrix associated with the *parcov()* control variable.

If parameters are log-transformed in a PEST control file, then linear analysis pertains to the log (to base 10) of these parameters. Calculations that involve log-transformation are invisible to the user. However if a user elects to provide prior parameter uncertainties, these must pertain to the logs of log-transformed parameters. Similarly variances that appear in the posterior parameter covariance matrix that is produced by PESTPP pertain to the logs of log-transformed parameters.

Linear uncertainty analysis requires a Jacobian matrix. PESTPP uses the last Jacobian matrix that it has calculated (or read using the *base\_jacobian()* control variable) as a basis for this analysis. It also requires that prior parameter uncertainties be provided. PESTPP offers a number of options for this.

Unless a *parcov()* control variable is provided in the PEST control file, PESTPP assumes that all adjustable parameters are statistically independent. In this case, by default, the prior standard deviation of each parameter is calculated as a quarter of the difference between its upper and lower bounds as provided in the PEST control file. However the *par\_sigma\_range()* control variable (the default value for which is 4.0) can be used to specify that the difference between parameter bounds is equivalent to a different number of standard deviations from this. If a parameter is log-transformed, the prior standard deviation of its log is calculated from the difference between the logs of its bounds.

Alternatively, the name of a file can be supplied as the value of the *parcov()* control variable. If so, PESTPP reads prior parameter uncertainties from this file. Options for this file are as follows:

* A parameter uncertainty file; see appendix B of this manual for the format of this type of file. PESTPP recognizes this file type through the fact that it possesses an extension of *.unc*.
* A file holding a single prior covariance matrix; see appendix B of this manual for the format of this type of file. PESTPP recognizes this file type through an extension of *.cov*.
* A binary file that stores a covariance matrix using the same protocol as that in which programs of the PEST and PEST++ suites store a Jacobian matrix. PESTPP recognizes this type of file by an extension of *.jco* or *.jcb*.

When asked to undertake FOSM analysis, PESTPP calculates a posterior parameter covariance matrix. This is stored in a file named *case.post.cov* where *case* is the filename base of the PEST control file. As already stated, posterior variances and covariances pertaining to parameters that are log-transformed in the PEST control file, pertain to the logs of these respective parameters. Note that this is the same posterior covariance matrix as that calculated by the PEST PREDUNC7 utility, and by the PyEMU Schur object.

PESTPP-calculated prior and posterior parameter uncertainties are recorded at the end of its run record file. They are also recorded in a comma delimited file named *case.par.usum.csv* (“usum” stands for “uncertainty summary”). Prior and posterior means, standard deviations and bounds in the latter file pertain to the logs (to base 10) of parameters which are log-transformed in the PEST control file. These upper and lower posterior parameter bounds are calculated as the parameter’s estimated or initial value (depending on whether or not inversion has been carried out) plus and minus two standard deviations. Under the Gaussian assumption, bounds calculated in this way approximately span the parameter’s 95% posterior confidence interval.

As well as calculating parameter uncertainties, PESTPP can also be asked to calculate the prior and posterior uncertainties of some predictions. This functionality is activated through use of the *forecasts()* control variable. The values which must be supplied for this variable are the names of predictions whose uncertainties are sought. For example *forecasts(ar10,ar11)* requests that prior and predictive uncertainties be evaluated for model outputs named “ar10” and “ar11” in the PEST control file on which PESTPP’s operations are based. Despite the fact that these model outputs are predictions, they must be listed in the “observation data” section of the PEST control file; hence sensitivities of these model outputs to parameters are available as rows of the Jacobian matrix which is calculated by PESTPP. Model predictions should be endowed with weights of zero in a PEST control file; this is because predictions are not used to constrain parameters, and hence do not form part of a calibration dataset. (PESTPP issues a warning message if this is not the case.) The uncertainties and lower/upper bounds of predictions that are specified in this way are listed at the end of the PESTPP run record file, and in a comma-delimited file named *case.pred.usum.csv*. Posterior predictive lower and upper bounds are calculated by subtracting and adding two standard deviations from/to the value of the prediction as calculated by the model using initial or estimated parameter values.

### 6.2.9 Model Run Failure

The inversion process implemented by PESTPP is an iterative procedure. Each iteration is subdivided into two parts. Finite-difference derivatives are calculated in the first part of each iteration; parameter upgrades are calculated and tested in the second part. Model run failure is much more likely to occur during the second of these parts than in the first of these parts as parameter values may vary significantly from model run to model run in the latter case. Where an updated parameter set precipitates model run failure, PESTPP deems the objective function to be very high; the offending parameter set is therefore judged to be far from optimal.

Model run failure during finite difference derivatives calculation is more worrisome, for a single parameter undergoes only an incremental change from its current value for each model run. A strategy for handling failed model runs under these circumstances is to omit this parameter from the inversion process for the remainder of the current iteration, effectively freezing it at its current value. On some occasions, this may be a shortsighted strategy, for if incremental variation of one parameter instigates model run failure, incremental variation of another parameter may do the same. An alternative strategy is to abandon the inversion process as the likelihood of further parameter improvement has been caste into doubt.

The *der\_forgive()* control variable can be used to govern PESTPP’s behavior under these circumstances. It must be supplied as either *true* or *false*. If it is supplied as *true* (its default value) then model run failure when finite difference derivatives are being calculated is accommodated using the first of the above alternatives, that is through temporary freezing of the parameter at its current value. However if it supplied as *false*, then model run failure during calculation of finite-difference derivatives precipitates cessation of PESTPP execution.

### 6.2.10 Composite Parameter Sensitivities

PESTPP records composite parameter sensitivities in a file named *case.sen* where *case* is the filename base of the PEST control file. These are recorded during each iteration of the inversion process. Two composite parameter sensitivities are recorded. The first is the *csp* statistic of Doherty (2015), calculated using the equation

(6.18)

where **J** is the Jacobian matrix, **Q** is the weight matrix and *n* is the number of non-zero-weighted observations. PESTPP also records the composite scaled sensitivity of Hill and Tiedeman (2007) in this same file; see that text for details of its computation. Where regularization is employed in the inversion process, two sets of these two composite sensitivities are calculated. Regularization observations and prior information equations are included in one of them, while these are excluded from the other. Where they are included, the weights applied to regularization are multiplied by the current regularization weight factor.

### 6.2.11 Other Controls

If the control variable *iteration\_summary()* is set to *true*, then PESTPP records (and continually updates) a comma-delimited file named *case.upg.csv* (where “upg” stands for “upgrade”). This file lists the values of parameters used for every model run in which parameter upgrades are tested. Iteration numbers, lambda values and fractional lengths along parameter upgrade vectors are also recorded in this file. PESTPP also records files named *case.ipar*, *case.iobj*, *case.isen* and *case.rid*. The first three of these are comma-delimited files; they list iteration-specific values of parameters, objective function components and composite parameter sensitivities respectively. *case.rid* links model runs undertaken for derivatives calculation to run identifiers in the parallel run management file; a user can thus be informed of the worker that undertook a particular finite-difference model run.

If the *jac\_scale()* control variable is set to *true*, the equations that are used for calculating parameter upgrades are slightly modified from those presented in section 6.2.1. Prior to estimation, parameters are scaled by their sensitivities. Estimated scaled parameters then undergo post-estimation back-transformation; see equation 5.4.6 of Doherty (2015). This strategy can reduce numerical errors in some instances; however it can also increase computation times. The default value for *jac\_scale()* is *true*.

### 6.2.12 Running PESTPP

See section 5 of this manual for how to run PESTPP. As is described in that section, model runs can be undertaken in series or in parallel. In either case, a prematurely terminated PESTPP run can be restarted by commencing PESTPP execution using the “/r” command line switch.

### 6.2.13 PESTPP Output Files

The following table summarizes the contents of files that are recorded by PESTPP when it is asked to undertake highly-parameterized inversion. Most of these have been discussed above. It is assumed that the PEST control file on which the inversion process is based is named *case.pst*.

|  |  |
| --- | --- |
| **File** | **Contents** |
| *case.rec* | Run record file. This file records a complete history of the inversion process. It is available for user-inspection at any time during that process. |
| *case.rmr* | Parallel run management record file. |
| *case.par* | Parameter value file. This is compatible with a file of the same name written by PEST. It records the best parameter values obtained up until any point of the inversion process, and at its end. |
| *case.parN* | Parameter value file for iteration *N*. This file contains best parameters calculated during iteration *N* of the inversion process. |
| *case.bpa* | Same as *case.par*. This file records base parameter values (which are actual parameter values) when PESTPP conducts SVD-assisted inversion. It is included in the set of PESTPP output files for compatibility with PEST. |
| *case.parb* | Same as *case.bpa*. |
| *case.res* | Residuals file. This file lists model outputs, observations, residuals and related data calculated using best parameters achieved through the inversion process. The latter are listed in file *case.par*. |
| *case.reiN* | An intermediate residuals file. This file lists model outputs, observations, residuals and related data calculated using best parameters achieved during iteration *N* of the inversion process. The latter are listed in file *case.parN*. |
| *case.rei* | The latest incidence of *case.reiN*. |
| *case.sen* | Parameter sensitivity file. This file contains composite parameter sensitivities; it is updated during every iteration of the inversion process. |
| *case.ipar* | This file is recorded unless the *iteration\_summary()* control variable is set to *false*. It is a CSV file listing best parameters attained at all iterations of the inversion process. |
| *case.iobj* | This file is recorded unless the *iteration\_summary()* control variable is set to *false*. It is a CSV file listing objective function components attained at the end of all iterations of the inversion process. |
| *case.isen* | This file is recorded unless the *iteration\_summary()* control variable is set to *false*. It is a CSV file containing composite parameter sensitivities calculated during all iterations of the inversion process. |
| *case.upg.csv* | This file is recorded unless the *iteration\_summary()* control variable is set to *false*. It is a CSV file containing upgraded parameter values calculated for each Marquardt lambda and each scale factor employed for all iterations of the inversion process. |
| *case.rid* | Links model runs undertaken for calculating finite-difference derivatives with respect to each parameter to model run indices recorded in file *case.rmr*. |
| *case.svd* | Information is appended to this file every time a parameter upgrade direction is calculated using a new Marquardt lambda. It lists the singular values of the (**J**t**QJ** + λ**I**) matrix. |
| *case.pfm* | PESTPP performance record. This file records the times at which PESTPP commenced and completed various processing tasks. |
| *case.fpr* | Records a history of when parameters were frozen at their bounds, and when they were subsequently released. |
| *case.jco* | A PEST-compatible Jacobian matrix file that is refreshed during every iteration of the inversion process. This is a binary file. On iterations that are dedicated to estimation of super parameter parameters, it contains super parameter sensitivities. On other iterations it contains base parameter sensitivities. |
| *case.jcb* | A PEST-compatible Jacobian matrix file that contains base parameter sensitivities. It is refreshed on every occasion that base parameter sensitivities are re-calculated. |
| *case.post.cov* | Contains the posterior parameter covariance matrix if linear uncertainty (i.e. FOSM) analysis is requested. |
| *case.par.usum.csv* | Summarizes linear uncertainty analysis as it pertains to parameters. |
| *case.pred.usum.csv* | Summarizes linear uncertainty analysis as it pertains to user-specified predictions. |
| *case.rst* | A binary file containing restart information. |
| *case.rns, case.rnj, case.rnu* | Binary files used by the run manager. |

Table 6.1. Files recorded by PESTPP.

## 6.3 Differential Evolution

### 6.3.1 General

As well as performing highly parameterized inversion using a so-called “gradient method” that requires calculation of a Jacobian matrix, PESTPP also performs global optimization using the differential evolution (DE) method. This method does not require calculation of derivatives. As such, it is classed as a “global method”. In general, global methods do not offer the same possibilities for highly parameterized, regularised inversion as do gradient methods. However they do possess some attractive features. Chief among them are the following:

* They are less susceptible to entrapment in local objective function minima than are gradient methods;
* Their performance does not deteriorate with model numerical difficulties that can erode the integrity of finite-difference derivatives on which gradient methods depend.

The good performance of global methods in difficult numerical circumstances comes at a cost, however. Their model run requirements are generally far greater than those of gradient methods.

### 6.3.2 The DE Method

#### Overview

Differential Evolution (DE) is a population based stochastic optimization technique developed by Ken Price and Rainer Storn. It is described by Storn and Price (1995; 1997). It is also described in detail at the following web site, from which source code in a number of different computing languages can be downloaded:

<http://www1.icsi.berkeley.edu/~storn/code.html>

The algorithmic basis of the DE method is particularly simple. Despite this, it performs well, and is well-cited in published literature. It possesses the following desirable properties:

1. It has the ability to handle non-differentiable, nonlinear and multimodal objective functions;
2. It is easily parallelizable;
3. It is easy to use, requiring few control variables;
4. It has good convergence properties.

DE is a direct search method. It works with a population of parameter vectors (i.e. a population of parameter sets) which are progressively replaced from generation to generation. However, during any iteration of the objective function minimization process that is implemented by DE, not all vectors which comprise the current population of parameter vectors are replaced by new vectors in formation of the next generation. In common with most direct search methods, DE uses the “greedy” selection criterion to determine whether a new parameter set (i.e. vector) is part of the next generation; selection of a new parameter set is guaranteed if it lowers the objective function.

While greedy selection speeds convergence, it raises the risk of entrapment in a local objective function minimum. However, because a population of vectors is retained in each generation for use in computing new members for the next generation, this risk is reduced. Furthermore, unlike many other global methods which use predetermined probability density functions to compute parameter perturbations (and hence parameter upgrades), the DE method uses differences between vectors randomly selected from the current generation to compute these perturbations. Using distance and direction information from the current generation to guide the search process not only simplifies the method; it greatly reduces the number of control variables that govern its operation (Engelbrecht, 2007).

Parameter set updating as implemented by the DE algorithm is comprised of four steps. These are:

1. initialization;
2. mutation;
3. crossover;
4. selection.

Steps 2 to 4 comprise a single iteration of the DE method. The collection of parameter sets that exists at the beginning of each iteration is modified to form a new generation of parameter sets. This process is then repeated during subsequent iterations.

#### Initialization

The initial generation of parameter sets (i.e. the initial collection of parameter sets) is typically accumulated by randomly sampling each parameter individually on the basis of a probability distribution that is uniform between its upper and lower bounds. However, if a preferred parameter set exists prior to the DE process, the initial parameter probability distribution can be multiGaussian, and centered on this set.

The number of parameter sets which characterizes a parameter generation remains unchanged throughout the DE process. In the following discussion, we use *NP* to represent this number. We use the symbol *D* to characterize the number of dimensions in parameter space, i.e. the number of individual parameters that comprise a parameter set.

#### Iteration

Once an initial generation of parameter sets is populated, the iterative process of parameter set refinement can begin. During any iteration of the optimization process, each of the *NP* parameter sets that comprise the current generation is selected in turn. Let us refer to a selected parameter set (i.e. vector) as **x***i*. An attempt is made to replace this vector by another vector that has something in common with it and something that is different. The things that it has in common are the values of some of its *D* elements (i.e. the values of some individual parameters); however the vector which may replace it (it is not a foregone conclusion that it will actually be replaced), has other of its *D* elements changed.

#### Mutation

For each selected vector, another three parameter vectors from the current generation are selected at random. Let us refer to these as **x***r1*, **x***r2* and **x***r3*. As these must be different from the selected vector, it is apparent that the minimum value of *NP* required by the DE method is 4. A so-called “mutant vector” is calculated by taking the difference between two of these randomly-selected vectors, multiplying it by a factor *F*, and adding it to the third vector to form a new vector, which we refer to as **v***i*. That is:

**v***i* = **x***r1* + *F*(**x***r2* - **x***r3*) (6.19)

If this mutation carries **v***i* outside its permissible change, it is reflected back at the boundary of allowed parameter space (i.e. the space formed by the upper and lower bounds of each parameter). Selecting randomly from the interval [0.5, 1.0], either for each generation or for each mutation vector, is known as “dithering”; this can aid convergence in noisy objective function terrains. PESTPP supports dithering at the level of individual mutation vectors.

#### Crossover

A trial vector, which we refer to as **u***i*, is now calculated. At least one element of **v***i* must feature in **u***i*. This element is selected randomly from its *D* elements. For all other elements of **v***i* (let us characterize each of them by the index *j*), a random number between 0 and 1 is generated. If this number is less than a user-specified number *CR*, then the *j*th element of **u***i* (i.e. **u***ij*) is equated to the *j*th element of **v***i* (i.e. **v***ij*). Otherwise it is assigned the *j*th element of **x***i*, the currently selected parameter set for which a replacement is being sought. *CR* is referred to as the “crossover constant”. Obviously it must lie in the interval [0, 1]. *CR* can be considered to be the probability that a particular parameter in the trial vector **u***i* (i.e. a particular element of **u***i*) is the same as that of the mutation vector **v***i*.

#### Selection

The model is now run using the parameter set **u***i*. If the objective function is lower than that associated with the parameter set **x***i*, then **x***i* is replaced by **u***i* in the next generation of parameter sets. Otherwise **x***i* is retained in the next generation. (As stated above, this selection process is referred to as “greedy”.)

The above process is undertaken for all *i* in the range 1 to *NP*.

### 6.3.3 Using DE in PESTPP

PESTPP implements differential evolution rather than gradient-based inversion if the *global\_opt()* control variable is supplied as “de”, that is if the string “global\_opt(de)” appears on a line of the PEST control file that begins with the characters “++”. At the time of writing, “de” is the only permissible argument for *global\_opt()*. To perform gradient-based inversion, omit this keyword from the PEST control file.

The population size (i.e. *NP* in the above description) can be supplied as the value of the *de\_pop\_size()* control variable. The default value for this variable is 40. Recommendations differ on the best value for *NP*. Ten times the number of parameters has been suggested. However Rainer Horn (one of the creators of the DE method) has suggested that values above 40 rarely improve convergence. Perhaps the best setting for *NP* is the larger of 40 and the number of parameters whose values are being optimized.

The control variable *de\_f()* selects *F* of equation 6.19. Its default value is 0.8. Alternatively, dithering of *F* can be enabled through use of the *de\_dither\_f()* control variable. The value of *de\_dither\_f()* can be either *true* or *false* (i.e. it must be supplied as either “de\_dither\_f(true)” or “de\_dither\_f(false)”). *True* is the default, as this can dramatically improve convergence; in this case the value assigned to *F* is randomly drawn from the interval whose lower bound is 0.5 and whose upper bound is 1.0.

The crossover constant *CR* is selected through the *de\_cr()* control variable. The value for this variable must lie between 0.0 and 1.0; its default is 0.9.

As presently programmed, the DE process terminates after a user-supplied number of iterations has elapsed. This number is supplied as the value of the *de\_max\_gen()* control variable. The default value for this variable is 100.

### 6.3.4 Running PESTPP

See chapter 5 of this manual for how to run PESTPP, with model runs undertaken in serial and with model runs undertaken in parallel. At the time of writing, a prematurely terminated PESTPP run cannot be restarted when implementing differential evolution. If started using the “/r” switch, it re-commences the DE process.

### 6.3.5 PESTPP Output Files

When run in order to implement differential evolution optimization, a number of the output files recorded by PESTPP are actually empty as they pertain to gradient-based inversion. Other output files pertain to run management; these are discussed in section 6.2.13. The only output files that are relevant to DE-based optimization are those listed in the following table. In this table it is assumed that the PEST control file on which DE optimization is based is named *case.pst*.

|  |  |
| --- | --- |
| **File** | **Contents** |
| *case.par* | This file (referred to as a “parameter value file”) is available at any stage of the optimization process. It records best parameters that have been achieved so far. |
| *case.rec* | The run record file. This file records a history of the DE optimization process. It is available for inspection at any stage of that process. |
| *case.de1* | A binary file to assist in storage of model run outcomes. |

Table 6.2 PESTPP output files that are pertinent to DE optimization. It is assumed that the name of the PEST control file is *case.pst*.

## 6.4 Summary of PESTPP Control Variables

### 6.4.1 General

This section summarizes variables that control the operation of PESTPP. First those that feature in the PEST control file are discussed; see chapter 4 of this manual for a full description of the functions that they perform. The roles of PEST++ variables which control the operation of PESTPP are listed in table 6.3.

### 6.4.2 Control Variables in the PEST Control File

The PESTMODE variable determines whether PESTPP runs in “regularization” or “estimation” modes. It makes no sense to set this variable to “regularization” if global optimization is performed (that is, if the PEST++ *global\_opt()* control variable is set to “de”). Nevertheless, if it is set to “regularization” when PESTPP is asked to undertake DE optimization, PESTPP calculates and reports measurement and regularization components of the objective function. However it will not calculate a regularization weight factor; this is set to 1.0.

If PESTMODE is set to “regularization”, and the *global\_opt()* control variable is missing from the PEST control file which it reads, PESTPP employs Tikhonov regularization in gradient-based inversion. If the *reg\_frac()* control variable appears in the PEST control file, PESTPP ignores the contents of a “regularization” section that may be present in this file. Otherwise it reads the “regularization” section in order to obtain the values of variables which govern this mode of its operation (in particular, the target measurement objective function PHIMLIM, and variables which control calculation of the regularization weight factor - see section 4.16 of this manual for details). On the other hand, if PESTPP does not find a “regularization” section in the PEST control file, despite the fact that PESTMODE is set to “regularization”, it provides values for all regularization control variables itself.

If PESTPP undertakes gradient based inversion, then the NOPTMAX, PHIREDSTP, NPHISTP, NPHINORED, RELPARSTP and NRELPAR variables read from the eight line of the “control data” section of the PEST control file are used as termination criteria. Calculation of finite difference derivatives is governed by the INCTYP, DERINC, DERINCLB, FORCEN, DERINCMUL and DERMTHD variables recorded in the “parameter groups” section of the PEST control file. The PHIREDSWH control variable controls the switch from two to three point derivatives calculation; RELPARMAX, FACPARMAX and FACORIG impose limits on the change that any parameter can undergo during any iteration of the inversion process.

If the PEST control file on which the inversion process is based contains a “singular value decomposition” section, then the variables NUMSING and EIGTHRESH that appear in this section determine the singular value truncation point. If no “singular value decomposition” section is present in the PEST control file, then the default value for NUMSING is the number of adjustable parameters featured in the PEST control file; the default value for EIGTHRESH is 1.0E-7.

### 6.4.3 PEST++ Control Variables

Table 6.3 lists PEST++ control variables. All of these are optional. If a variable is not supplied, a default value is employed. The value of the default is presented along with the name of each variable in the table below. Variables are grouped in approximate accordance with their roles.

Variables discussed in section 5.3.6 that control parallel run management are not listed in table 6.1.

Note also that the number of control variables may change with time. Refer to the PEST++ web site for variables used by the latest version of PESTPP.

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Role** |
| *max\_n\_super()* | integer | The maximum number of super parameters to use when conducting SVD-assisted inversion. The default is the number of adjustable parameters, in which case the number of super parameters is effectively set by *super\_eigthresh()*. |
| *super\_eigthresh(1.0E-8)* | real | The ratio to maximum singular value of **J**t**QJ** at which truncation takes place to form super parameters. Note, however, that if the number of super parameters calculated in this way exceeds *max\_n\_super()* then the value of the latter variable takes precedence. |
| *n\_iter\_base(1)* | integer | Where super parameters are estimated in some iterations and base parameters are estimated in other iterations, this variable sets the number of sequential base parameter iterations to undertake before commencing an iteration in which super parameters are adjusted. If *n\_iter\_base()* is set to -1, this instructs PESTPP to emulate PEST behaviour; a base parameter Jacobian matrix is calculated; then super parameters are estimated as soon as they are defined on the basis of this matrix. Super parameters are estimated in all succeeding iterations. |
| *n\_iter\_super(4)* | integer | Where super parameters are estimated in some iterations and base parameters are estimated in other iterations, this variable sets the number of sequential super parameter iterations to undertake before commencing an iteration in which a base parameter Jacobian matrix is recalculated and base parameters are adjusted. |
| *mat\_inv(jtqj)* | text | The matrix that PESTPP uses to calculate parameter upgrades. Options are “jtqj” and “q1/2j”. |
| *jac\_scale(true)* | Boolean | Scale parameters by their sensitivities when calculating parameter upgrades. This can increase numerical precision; however it may incur a numerical cost. |
| *svd\_pack(redsvd)* | text | This informs PESTPP of the package that it must employ to undertake singular value decomposition of either the **J**t**QJ** or **Q**½**J** matrix (appropriately modified to include the Marquardt lambda and regularization). Options are “propack”, “eigen” and “redsvd”. |
| *lambdas(0.1,1,10,100,1000)* | set of real numbers | Values for the Marquardt lambda used in calculation of parameter upgrades. Note that this base list is augmented with values bracketing the previous iteration’s best lambda. However, if a single value is specified, only that lambda (and no other lambda) is used in all iterations. |
| *lambda\_scale\_fac(1.0)* | set of real numbers | These values are used to scale each parameter upgrade vector calculated using different values of lambda. This results in a line search along each upgrade vector direction. The number of tested parameter upgrades (and hence model runs) is equal to the number of lambdas times the number of scaling factors. Set *lambda\_scale\_fac()* to 1.0 to disable an upgrade direction line search. |
| *upgrade\_augment(true)* | Boolean | Augment the values of lambda used in parameter upgrade calculation by including the best lambda from the previous iteration, as well as twice the best lambda and half the best lambda from that iteration. If supplied as *false*, then only lambda values that are listed as arguments to the *lambda()* keyword will be tested during each iteration, and the lambda list will not be extended. |
| *base\_jacobian()* | text | Provide the name of a JCO file. The Jacobian matrix contained in this file will be used for the first iteration of the inversion process. |
| *hotstart\_resfile()* | text | Specify the name of a residuals file from a previous PESTPP run. PESTPP will assume that these are model outputs corresponding to initial parameter values. It will use these instead of undertaking the initial model run. |
| *uncertainty(true)* | Boolean | Flag to active or deactivate FOSM-based parameter and (optionally) forecast uncertainty estimation. |
| *parcov()* | text | Provide the name of a JCO, JCB, UNC or COV file from which the prior parameter covariance matrix used in FOSM analysis is read. |
| *par\_sigma\_range(4.0)* | real | The difference between a parameter’s upper and lower bounds expressed as standard deviations. |
| *forecasts(..,..,..)* | series of text strings | Provide the names of one or more observations featured in the “observation data” section of the PEST control file; these are treated as predictions in FOSM predictive uncertainty analysis. |
| *iteration\_summary(true)* | Boolean | This flag activates or deactivates the writing of CSV files summarizing parameters (*case.ipar*), objective functions (*case.iobj*), sensitivities (*case.isen*), trial parameter upgrades (*case.upg.csv*) and parameter-to-run-id mapping (*case.rid*). |
| *der\_forgive(true)* | Boolean | If set to *true*, then if model run failure occurs when calculating finite-difference derivatives with respect to a certain parameter, that parameter is frozen at its current value for the remainder of the iteration. If set to *false*, PESTPP terminates execution with an appropriate message if this occurs. |
| *global\_opt(de)* | text | Turn on differential evolution optimization. The only permitted argument is “de”. |
| *de\_pop\_size(40)* | integer | The differential evolution population size. This must be at least 4. Suggested values are 40 or the number of parameters undergoing optimization, whichever is highest. |
| *de\_max\_gen(100)* | integer | At the time of writing, this is the only available DE termination criterion. This is the number of iterations of DE optimization that is undertaken by PESTPP. |
| *de\_dither\_f(true)* | Boolean | This switch toggles dithering applied to *F* of equation 6.19. |
| *de\_f(0.8)* | real | The value of *F* of equation 6.19. |
| *de\_cr(0.9)* | real | The value of the DE crossover constant. |

Table 6.3 PESTPP control variables. Variables which control parallel run management can be supplied in addition to these. See section 5.3.6.

# 7. PESTPP-GSA

## 7.1 Introduction

### 7.1.1 General

The purpose of global sensitivity analysis (GSA) is to characterize how model parameters affect model outputs (or a function of model outputs such as an objective function) over a wide range of acceptable parameter values. In doing this, it strives for greater robustness, and for the provision of more information, than local sensitivity analysis based on partial derivatives of model outputs with respect to model parameters. Because local sensitivity analysis pertains to a single point in parameter space, the information that it yields is often insufficient to support an understanding of the behavior of nonlinear models whose outputs depend on combinations of model parameters in complicated and parameter-value-dependent ways.

The information provided by global sensitivity analysis differs widely between the different methodologies that implement it. In general, the more information that a global sensitivity analysis method provides, the greater is its computational burden. This information can vary from a simple measure of the influence that each parameter has on selected model outputs, to a complete description of how a parameter’s influence varies with its value, and the values of other model parameters. See Saltelli et al (2004; 2008) for an excellent coverage of these methods.

Global sensitivity analysis can support environmental modeling in at least the following ways.

1. Once it is recognized that some parameters affect model outputs to only a limited extent, these parameters can be fixed at user-specified values while others are estimated through calibration. This can accrue considerable numerical savings in the calibration of models whose run times are long.
2. Global sensitivity analysis may support construction of a simple model in place of a more complex one for use in calibration, uncertainty analysis and decision support.
3. By establishing the parameters that are important, and the interactions between parameters that are important, much can be learned about the system which a model attempts to simulate. This information can sharpen the intuition of those who must manage that system.

PESTPP-GSA currently supports two GSA meth­ods. These are

1. the Method of Morris (Morris, 1991), with extensions proposed by Campolongo et al (2005), and
2. the Method of Sobol (Sobol, 2001).

The Method of Morris is a “one-at-a-time” method (Saltelli et al, 2004). It is computationally efficient, and is therefore suitable for use with models whose run times are high. It provides estimates of the first two moments (mean and variance) of the effect that each parameter has on a model output of interest. These statistics acknowledge that a parameter’s sensitivity may be a function not just of its own value, but of the values of other parameters. In doing so, they reveal those parameters that have the most influence on model outputs of interest, and the consistency of these influences over parameter space. The information that it provides may justify the omission of some parameters from a calibration exercise; and/or it may support the design of a simple, fast-running, surrogate model. In contrast, the Method of Sobol has the potential to provide much more detailed information than the Method of Morris. Because it is based on decomposition of variance (Saltelli et al, 2004), it can reveal details of parameter nonlinearity that are beyond the reach of other methods. It can also reveal complex parameter interactions and, by inference, interaction of the processes to which these parameters pertain. Unfortunately, this information comes with a high computational cost. Hence unless Sobol-based global sensitivity analysis is restricted to only a few parameters and a relatively fast-running model, it is generally computationally unaffordable.

### 7.1.2 Grouped Parameters

There are many occasions where a modeler wishes to explore sensitivities of one or more model outputs to groups of parameters rather than to a single parameter. For example, in the groundwater modeling context, a modeler may wish to explore the sensitivity of a particular model output, or of the calibration objective function, to all of the pilot point parameters which collectively describe the vertical hydraulic conductivity of an aquitard. Analysis of the sensitivity of a model output to a group of parameters is easily accomplished by tying all but one of the members of that group to the remaining parameter; the latter then represents the group. In the above example, a single pilot point parameter would be selected as parent to all other pilot point parameters that collectively represent the vertical hydraulic conductivity of the aquitard. Where the sensitivity of the parent parameter is assessed using a global sensitivity analysis methodology, the joint sensitivity of it, and all of the parameters that are tied to it, are thereby assessed. In implementing this strategy, a modeler must ensure that the values assigned to tied parameters in relation to the parent parameter are realistic, because the ratio of these values will be preserved during the entire sensitivity analysis process.

## 7.2 Method of Morris

### 7.2.1 Elementary Effects

The Method of Morris focuses on quantities called “elementary effects”. In describing this methodology, notation used by Morris (1991) and by Saltelli (2008) is adopted. The latter text explains the method particularly well (see chapter 3 of that text); hence the description provided herein is brief.

In the previous chapter of this manual, parameters employed by a model were denoted by the vector **k**. In the present chapter we transform these parameters, and describe them using the vector **x**. The transformation is such that variability of each element of **x** (i.e. each parameter *xi*) spans the interval [0, 1]. If a parameter is log-transformed in a PEST control file, then its log spans this interval. Furthermore, the transformation is such that each parameter has a uniform probability distribution over this interval, and that each parameter is statistically independent of every other parameter. Hence if there are *m* elements in **k** (and hence in **x**), the domain of **x** is a hypercube in parameter space with a length of 1.0 on all sides.

A *p-*level grid is constructed inside this hypercube. Thus each side *i* of the hypercube is discretized so that each *xi* can adopt only the values {0, 1/(*p*-1), 2/(*p*-1), …,1}. In implementing Method of Morris analysis, any individual *xi* can vary between one of these values and another (not necessarily its neighbor). We refer to this difference as Δ. It applies to all *xi*. Only one *xi* is varied at a time.

For each of the values taken by **x** the model is run to calculate an output *y* (which can actually be a function of model outputs such as an objective function). From two model runs we can calculate the elementary effect EE*i* of the *i*th parameter on this output as

EE*i* = [*y(x1, x2, …xi-1, xi+*Δ*, xi+1,…xn*) – y(**x**)]/Δ (7.1)

On any given application of the Method of Morris, Δ does not change. Morris (1991) suggests that *p* be even and that Δ be calculated as:

Δ = *p*/[2(*p*-1)] (7.2)

Using an efficient sampling strategy (see below), the model is run enough times to calculate *r* elementary effects for each element *xi* of **x**.

The elementary effect of each parameter has a probability distribution. Using standard statistical formulas, a mean *μ* and standard deviation *σ* can be estimated for this probability distribution. The mean characterizes the influence that parameter *i* has on the selected model output. The standard deviation characterizes the variability of this influence, this being a function of model nonlinearity and of interactions between the selected parameter and other parameters. Campolongo et al (2005) suggest that a further statistic be calculated which they denote as *μ*\*. This is the mean of the absolute values of EE*i*. Campolongo et al suggest that this provides a more robust expression of nonlinearity and parameter interactions than does *μ* as the latter can be diminished through cancellation if EE*i* is non-monotonic with parameter value.

For each adjustable parameter featured in the “parameter data” section of a PEST control file, PESTPP-GSA computes all of *μ*, *μ*\* and *σ* for the objective function (calculated using equation 3.3) and, optionally, for each model output (i.e. observation) featured in the “observation data” section of the PEST control file.

### 7.2.2 Sampling Scheme

While the Method of Morris does not have the same theoretical foundations as variance-based sensitivity analysis, its major attraction is that it can provide reasonably robust indications of parameter influence, and parameter influence variability, in a relatively few number of model runs. Its efficiency is an outcome of the way in which it chooses parameters on which to base these model runs. By varying a single, randomly-chosen parameter at a time by the amount Δ described above, the cost of each new addition to the pool through which *μ*, *μ*\* and *σ* are calculated for each parameter is only one model run. At the same time, the sequencing of model runs makes the effects of model nonlinearity and parameter interactions visible in these statistics. Model runs are performed in sequences of *m* runs (where *m* is the number of parameters used by the model), in which each parameter is varied in random order while all other parameters retain their values from the previous model run. At the end of this sequence a revised estimate of statistics pertaining to all EE*i*’s is available. A new model run sequence is then initiated, starting at another random point within the gridded parameter domain. See Morris (1991) and chapter 3 of Saltelli (2008) for full details.

The number of model run sequences (denoted as *r* by Morris, 1991) is provided by the user. For a highly nonlinear model, a greater value of *r* leads to more robust estimates of *μ*, *μ*\* and *σ*.

### 7.2.3 Control Variables

In common with other programs comprising the PEST++ suite, PESTPP-GSA obtains case-defining information from a PEST control file. PEST++ control variables which govern the operation of the Method of Morris can be placed in this file on lines that begin with the “++” character string.

PEST++ variables which control the operation of the Method of Morris are listed in the following table. Default values are provided with each variable in this table. Note that if no GSA-pertinent variables are provided in a PEST control file used by PESTPP-GSA, it implements the Method of Morris.

|  |  |  |
| --- | --- | --- |
| **Control variable** | **Type** | **Role** |
| *method(morris)* | text | Methods are “morris” and “sobol”. |
| *rand\_seed()* | unsigned integer | Seed for the random number generator. |
| *morris\_r(4)* | integer | Sample size. The number of times that an elementary effect is computed for each parameter. That is, the number of sequences of *m* model runs undertaken by PESTPP-GSA, where *m* is the number of adjustable parameters featured in the PEST control file. |
| *morris\_p(4)* | integer | The number of levels employed to grid the interval [0,1] associated with each transformed parameter. The number of intervals into which [0, 1] is therefore subdivided is *p*-1. |
| *morris\_delta()* | real | The default value for *morris\_delta()* is *p*/2[(*p*-1)]. The value supplied for this variable must be a multiple of 1/2[(*p*-1)]. If this is not the case, PESTPP-GSA alters the user-supplied value to the nearest appropriate multiple. |
| *morris\_obs\_sen(true)* | Boolean | If supplied as *false*, PESTPP-GSA computes parameter sensitivities for the objective function only. If supplied as *true*, PESTPP-GSA computes parameter sensitivities for the objective function, as well as for each model output corresponding to observations featured in the “observation data” section of the PEST control file. |

Table 7.1 Variables used by PESTPP-GSA to control the operation of the Method of Morris.

Salteli et al (2004) suggest the following values for Method of Morris control variables:

* 4 for *morris\_p()*
* 0.667 for *morris\_delta()*
* 4 to 10 for *morris\_r()*

## 7.3 Method of Sobol

### 7.3.1 Sensitivity Indices

The Method of Sobol is based on the decomposition of variance. It employs theory derived by Sobol (2001) through which it can be shown that any function of an arbitrary number of parameters can be decomposed into summed functions of individual parameters, pairs of parameters, triplets of parameters, etc. From this it follows that the variance of any model output (or function of a model output) can be expressed as separate variances arising from individual parameters, pairs of parameters, triplets of parameters, etc. By discovering and separating these variances, the importance of each parameter to the model output can be revealed. So too can the extent to which the influence of any particular parameter on a model’s output results from its interaction with other parameters rather than being a direct outcome of its own influence. See Sobol (2001) and Homma and Saltelli (1996) for further details. See Saltelli et al (2004, 2008) for a comprehensive and easily understood discussion of the method, and for further details of concepts that are presented in summary form below.

The total variance *V*T(*y*) of the output *y* of a model with *m* parameters can be decomposed as follows:

(7.3)

where

(7.4)

(7.5)

(7.6)

etc.

In the above equations, the expression should be interpreted as “the variance with respect to *xi* of the expected value of *y*, with the latter calculated at multiple values at which *xi* is fixed while every other parameter is varied”. Similar interpretations apply to higher order terms in the above equations. *Vi* expresses the so-called “first order” dependence of *y* on *xi*. Meanwhile, *Vij* expresses the dependence of *y* on *xi* and *xj* together; note that the dependence of *y* on *xi* and *xj* individually has been subtracted from the first term on the right of this equation in order to obtain the collective variance.

Application of the Method of Sobol culminates in the calculation of so-called “sensitiv­ity indices”. The first order sensitivity index for parameter *i*, (i.e. *Si*), is defined as the ratio of the its first order variance to the total variance of *y*. That is

(7.7)

Meanwhile the second order sensitivity index that describes the effects of parameters *i* and *j* collectively on *y* is defined as

(7.8)

The variances featured in the above equations are not easy to calculate. They must be computed empirically by running the model many times and taking averages of pertinent model outputs and of pertinent squared model outputs. Sometimes the model output of interest must be computed many times using many different set of parameters before the averaged quantity stabilizes. This is especially the case for quantities such as *Vij* which measure the effect of more than one parameter on the model output *y*.

For this reason, Sobol-based sensitivity analysis is normally restricted to calculation of the first order sensitivity index *Si* for each parameter *i*, as well as the total sensitivity index *STi* for each parameter *i*. The first order index measures the individual importance of the parameter. The total sensitivity index includes terms that describe how the parameter effects the model output of interest in nonlinear ways, and through its interaction with other parameters. This is because

*STi* = *Si* + *Sij* + … + *Sij…m* (7.9)

Fortunately S*Ti* can be calculated without the need to compute the expensive cross terms appearing in equation 7.9 using the relationships

(7.10)

where the symbol **x**~i signifies all parameters but *xi* being allowed to vary. Calculation of *STi* using equation 7.10 requires, once again, many model runs and appropriate averaging of model outputs and squared model outputs. However the numerical burden is far smaller than that required for calculation of second and higher order effects directly.

Notwithstanding its ability to provide a comprehensive characterization of the relationship between a model output and all parameters employed by a model, use of Sobol’s method is compromised by its high model run requirements. Hence the Method of Morris is normally a more practical alternative unless model run times are minimal.

### 7.3.2 Control Variables

PEST++ variables which control the operation of the Method of Sobol are listed in the following table. As for variables which control PESTPP-GSA’s implementation of the Method of Morris, these must be provided on “++” lines within a PEST control file.

|  |  |  |
| --- | --- | --- |
| **Control variable** | **Type** | **Role** |
| *method()* | text | Select “sobol” to implement Method of Sobol. |
| *rand\_seed()* | unsigned integer | Seed for the random number generator. |
| *sobol\_samples()* | integer | The number of samples to use in computing variances. The number of model runs is actually twice this number because of the need to employ two series of parameter samples. See Saltelli et al (2008) for details. |
| *sobol\_par\_dist()* | text | Specifies whether parameter samples should be drawn from a uniform or normal distribution. Values are “unif” or “norm” respectively. In the latter case, samples are centered on parameter values provided in the PEST control file, while the standard deviation is a quarter of the difference between a parameter’s upper and lower bounds. Log-uniform and log-normal distributions are employed for parameters which are denoted as log-transformed in the PEST control file. |

Table 7.2 Variables used by PESTPP-GSA to control the operation of the Method of Sobol.

## 7.4 PESTPP-GSA Output Files

PESTPP-GSA writes the following output files. It is assumed that the filename base of the PEST control file on which global sensitivity analysis is based is named *case.pst*.

|  |  |
| --- | --- |
| **File** | **Contents** |
| *case.msn* | Method of Morris only. Lists method of Morris outputs (*μ*, *μ*\* and *σ*) for each adjustable parameter. The model-generated quantity for which these provide sensitivity measures is the objective function. This is calculated using equation 3.3. |
| *case.mio* | Method of Morris only. This file is recorded if the *morris\_obs\_sen()* control variable is set to *true* (the default). It records *μ*, *μ*\* and *σ* for all model outputs (i.e. observations) featured in the “observation data” section of the PEST control file. |
| *case.raw* | Method of Morris only. Lists parameter values and objective functions for all model runs. |
| *case.sbl* | Method of Sobol only. Lists first order and total sensitivity indices for all model outputs listed in the “observation data” section of the PEST control file with respect to all adjustable parameters listed in the “parameter data” section of the PEST control file. |
| *case.rmr* | Parallel run management record. This file is written if model runs are conducted in parallel. |
| *case.rns* | Binary file used for model run management. |

Table 7.3 Files written by PESTPP-GSA. It is assumed that the name of the PEST control file is *case.pst*. Data elements in all of the above files are comma-delimited.

# 8. PESTPP-OPT

## 8.1 Introduction

### 8.1.1 A Publication

PESTPP-OPT is described by White et al (2018), where examples of its use are also provided. The following description summarizes information available from this source. See also Wagner and Gorelick (1987) where a similar methodology is described.

### 8.1.2 Overview

Sustainable management of a natural system often requires that an optimization problem be solved. Something must be maximized or minimized through adjustment of so-called “decision variables”, subject to certain constraints. For example, it may be desirable to maximize the amount of water extracted from a number of wells (where pumping rates are the decision variables), subject to the constraints that flow in an adjacent stream does not fall below a specified rate, and that groundwater levels in certain observation wells are maintained above certain levels. Design of a contaminant remediation system may attempt to ensure that the cost of water extraction and treatment is minimized subject to the constraint that the contaminant is captured; pumping and injection rates, and the locations of pumping and injection wells, comprise the decision variables in this example.

Models are used to predict the response of a natural system to natural and management-imposed stresses. Where management is optimized, some of the quantities which the model calculates comprise system behaviour on which constraints must be imposed. These outputs are normally uncertain, reflecting the fact that the model’s parameters are uncertain. Model outputs to which constraints are applied have this in common with any other predictions made by a model.) The question then arises as to whether imposition of constraints should take account of these uncertainties. Where violation of a constraint can result in an unacceptable cost, the answer to that question is obvious: respect for valuable societal and/or environmental assets requires that model-calculated quantities to which constraints are applied be adjusted to include the range of possibilities that are compatible with the range of reasonable parameters that a model can employ. For example, in the stream flow example discussed above, exercise of the precautionary principle may dictate that the constraint be applied to the lowest streamflow that would be calculated by the model if it were parameterized with the most pessimistic set of parameters (with respect to that particular model output) that are compatible with expert knowledge on the one hand, and the necessity to fit the model calibration dataset on the other hand.

In most environmental modeling contexts, a model is calibrated before it is deployed. As is described by Doherty (2015), if properly undertaken, the calibration process yields a parameter field of minimized error variance. This is its “passport to uniqueness”. The parameter field is not correct; its potential for wrongness (which may be large) is merely minimized. Any prediction that the model makes inherits this status. That is, the prediction is not correct; however its potential for wrongness has been minimized. Hence a prediction made by a calibrated model lies somewhere near the centre of the posterior probability distribution of that prediction. The same concept can be extended to model outputs that describe environmental behaviour to which constraints must be applied.

If uncertainty is to be taken into account in imposition of an optimization constraint, the width of the probability distribution associated with the model output to which the constraint is applied must be calculated so that the constraint can be applied to a predictive value that is adjusted in order to accommodate its uncertainty. Often (but not always – see below) it will be adjusted towards the pessimistic end of its probability range.

PESTPP-OPT not only solves a constrained optimization problem. It solves a constrained optimization problem that accommodates uncertainties in model outputs to which constraints are applied. These are often referred to as “chance constraints”. In applying chance constraints, PESTPP-OPT assumes that model predictive uncertainty is an outcome of model parameter uncertainty. The latter is, in turn, an outcome of prior parameter uncertainty (i.e. the uncertainty range that emerges from the stochastic nature of expert knowledge), and the extent to which this uncertainty is reduced through the model calibration process. Parameter uncertainty reduction is a function of the information content of the calibration dataset, and the extent to which flow of this information is hampered by the presence of noise within that dataset.

### 8.1.3 Calculation of Uncertainty

PESTPP-OPT calculates post-calibration parameter and predictive uncertainties using linear methods – the same methods that are available through PESTPP (if the *uncertainty()* control variable is set to *true*), PyEMU and the PEST PREDUNC utilities. These are also referred to as “first order second moment”, or FOSM, methods. FOSM methods require user-characterization of prior parameter uncertainty, user-characterization of measurement noise associated with the calibration dataset, a Jacobian matrix that provides sensitivities of model outputs used in the calibration process to adjustable parameters, and the sensitivities of pertinent model predictions (i.e. the predictions to which chance constraints will be applied during the optimization process) to parameters used by the model. If asked to do so, PESTPP-OPT can calculate these sensitivities itself. Optionally, it can deduce prior parameter uncertainties from parameter bounds supplied in a PEST control file. It calculates measurement noise from the values of observation weights that appear in this same PEST control file.

The variance (square of standard deviation) of the post-calibration uncertainty of a model prediction *s* can be calculated using either of the following equations; see Doherty (2015) for details.

σ2*s* = **y**tC(**k**)**y** – **y**tC(**k**)**J**t[**J**C(**k**)**J**t + C(**ε**)]-1**J**C(**k**)**y**t (8.1a)

σ2*s* = **y**t[**J**tC-1(**ε**)**J** + C-1(**k**)]-1**y** (8.1b)

In the above equations

**k** is the vector of adjustable parameters;

C(**k**) is the prior parameter covariance matrix;

**ε** is the vector of measurement noise;

C(**ε**) is the covariance matrix of measurement noise (normally diagonal);

**J** is the Jacobian matrix pertaining to the model calibration process; and

**y** is a vector whose elements are the sensitivities of the prediction *s* to model parameters; in the present context this is the prediction to which a constraint is applied.

Both of the above equations are mathematically equivalent. The choice of which one to use in any numerical circumstance is an outcome of the size of the matrix that must be inverted. For equation 8.1a it is *n × n*, where *n* is the number of observations comprising the calibration dataset. For equation 8.1b it is *m × m*, where *m* is the number of parameters that are estimated through the calibration process.

Because the Jacobian matrix **J** that is featured in equations 8.1 pertains to the calibration process, it contains sensitivities of model outputs to which there are corresponding field measurements to parameters that are adjusted to fit the calibration dataset. **J** can be supplied by the user. It can also be calculated by PESTPP-OPT itself; in the latter case its calculation can be repeated if sensitivities encapsulated in **J** are likely to change as the values of decision variables change during the process through which the latter are optimized.

Elements of the vector **y** must, in general, be calculated by PESTPP-OPT. While the **y** vector is similar to the **J** matrix featured in the above equation in that it contains sensitivities of model outputs to parameters, its sensitivities pertain to the model when run to simulate management conditions rather than when run to simulate historical (i.e. calibration) conditions.

It is of interest to note that the first of the above equations (i.e. 8.1a) illustrates well the role that history matching plays in reducing the uncertainty of a model prediction. The first term on the right is the prior uncertainty of the prediction *s*. The second term on the right denotes the extent to which this prior uncertainty is reduced through the history-matching process. Where history matching does not take place, this term is zero.

The posterior parameter covariance matrix can be calculated by removing predictive sensitivities from the above equations. Let us denote this matrix as Cʹ(**k**). Then

Cʹ(**k**) = C(**k**) –C(**k**)**J**t[**J**C(**k**)**J**t + C(**ε**)]-1**J**C(**k**) (8.2a)

Cʹ(**k**) = [**J**tC-1(**ε**)**J** + C-1(**k**)]-1 (8.2b)

The posterior uncertainty of a model output can then be calculated directly from Cʹ(**k**) using the equation

σ2*s* = **y**tCʹ(**k**)**y** (8.3)

### 8.1.4 Optimization

An optimization problem can be formulated in many ways. For the moment it will be characterized as minimizing an objective function. (A maximization problem can be turned into a minimization problem simply through reversing the sign of the objective function.) The objective function which PESTPP-OPT minimizes must be distinguished from that which is minimized through model calibration. Use of PESTPP-OPT assumes that the model has already been calibrated (or, if not, it assumes that it does not need to be calibrated). The set of parameters that it uses must therefore be those that emerge from the calibration process or (if the model has not been calibrated) those that are of minimized error variance from an expert knowledge point of view. Hence the objective function that is the focus of model calibration is not considered when using PESTPP-OPT. Nevertheless, as will be discussed below, it is implicitly taken into account through the weights that are assigned to observations comprising the calibration dataset that are featured in the PEST control file on which PESTPP-OPT’s operations are based.

PESTPP-OPT minimizes a linear objective function φ defined as follows:

φ = **c**t**x** (8.4a)

In equation 8.4 both **c** and **x** are vectors. The vector **x** contains the current values of decision-variables. As stated above, these are management related quantities (such as pumping or injection rates in various wells at various times) that PESTPP-OPT adjusts in order to achieve system management goals. For constrained optimization to make sense, all of the elements of **c** and **x** must be positive. In practice they do not need to be provided as such; PESTPP-OPT makes the necessary transformations internally.

The vector **c** contains constants which must be supplied by the user. These are often factors by which decision variables must be multiplied to obtain monetary units, which are then used to express costs. Hence the *i*th element of **c** is the cost associated with the *i*th element of **x**. This is clear if equation 8.4a is re-written as

(8.4b)

where *d* is the number of decision variables.

While undergoing adjustment to minimize φ, the elements of **x** are subject to one or more linear constraints. Because they are linear they can be expressed through a matrix equation such as

**Ax** ≤ **b** (8.5)

**A** is often referred to as a “response matrix”. For constraints applied to model-calculated quantities, **Ax** represents model outputs calculated on the basis of current values of decision variables. Representation of the model’s action on decision variables as a matrix is an approximation. PESTPP-OPT accommodates this approximation in ways discussed below. Meanwhile, the elements of **b** must be supplied by the user. They are specific to his/her management problem. For example, one particular element of **b** may represent the rate of flow in a stream that must be maintained. This is actually a “greater than” constraint rather than the “less than” constraints embodied in equation 8.5. PESTPP-OPT handles “greater than” constraints through appropriate internal transformation.

The *i*th constrain (i.e. the *i*th row of **A**) can be written as

*ai*1*x*1 + *ai*2*x*2 + *ai*3*x*3 + …. + *aidxd* ≤ *bi* (8.6)

where *aij* is the element of **A** that occupies its *i*th row and *j*th column. Where a constraint represents the action of the model, the coefficients in equation 8.6 are calculated by PESTPP-OPT using finite differences under an assumption of local model linearity with respect to decision variables. This is similar to the way in which programs like PEST and PESTPP calculate sensitivities when estimating parameters. However in this case, finite differences are taken of decision variables rather than parameters in order to calculate sensitivities of model outputs to which constraints are applied rather than outputs corresponding to field measurements comprising a calibration dataset. As will be discussed, PESTPP-OPT offers the same range of options in calculating these derivatives as PEST and PESTPP offer for calculation of derivatives with respect to model parameters. The response matrix which is formed in this way is used in place of all or part of the matrix **A** appearing in equation 8.5. For other constraints (i.e. constraints which are directly applied to decision variables), the coefficients appearing in equation 8.5 can be supplied directly by the user.

The optimization algorithm employed by PESTPP-OPT employs a so-called “linear programming” or “simplex” methodology that is accessed through the open-source CLP optimization library (Forrest et al., 2016), developed through the Computational Infrastructure for Operations Research (COIN-OR) project; see Lougee-Heimer (2003). This algorithm is fast and efficient; it can handle hundreds of thousands of decision-variables. The assumption of a linear relationship between model outputs and decision variables is accommodated by repeating the linear optimization process in a series of iterations in which the decision variable response matrix (i.e. **A** of equation 8.5) is re-computed on each occasion. Where decision-variables are many, this can be a time-consuming process. The iterative nature of the optimization process earns it the name “sequential linear programming”, or simply SLP for short. See Ahlfield and Mulligan (2000) for further details.

### 8.1.5 Chance Constraints

A user of PESTPP-OPT can inform it whether he/she would like the optimization process which it implements to be risk neutral, risk averse, or risk tolerant. In the latter two cases he/she can specify the degree of aversion or tolerance that should characterize that process. Tolerance or aversion is introduced through the way in which model output uncertainty affects the imposition of optimization constraints.

Suppose that a user specifies that a model output *o* shall have a value no greater than *b*. Suppose also that the standard deviation of post-calibration uncertainty associated with model output *o* is *σ*o (*σ*o is calculated by PESTPP-OPT using equations 8.1 to 8.3). PESTPP-OPT assumes that model output uncertainties are characterized by a normal distribution. A user can specify, in recognition of the uncertainty associated with *o*, that he/she must be 95% sure that the constraint is not violated. In this case PESTPP-OPT applies the constraint to the model output *o* plus an amount δ*o* calculated to ensure that, according to the normal probability distribution, the chances of *o* being smaller than *o*+ δ*o* are 95%. Alternatively, a user may adopt a risk tolerant strategy by specifying that he/she will be happy as long as there is a 5% chance that the constraint is respected. In that case PESTPP-OPT applies the constraint to *o* minus this same δ*o*. A risk neutral approach results in the constraint being applied to *o*. This technique of shifting model-simulated values to which constraints are applied up or down in accordance with risk tolerance or risk aversion is referred to as chance-constraint programming (Charnes and Cooper, 1959; Miller and Wagner, 1965; Tung, 1986; Wagner and Gorelick, 1987; Hantush and Marino, 1989; Chan, 1994).

A PESTPP-OPT user must provide one number to characterize his/her approach to risk. This number must be between zero and one. A model-output-specific number, representing the uncertainty of that output, is then added or subtracted from it prior to imposition of optimization constraints on that output. Provision of a value of 0.5 for this variable (signifying risk neutrality) is equivalent to ignoring parameter, and hence predictive, uncertainty. Under these circumstances, PESTPP-OPT does not calculate model output uncertainties at all. This reduces input requirements, at the same time as it accelerates the optimization process by foregoing the need to (re)calculate the **J** matrix and/or **y** vectors of equations 8.1 to 8.3. On the other hand, a value of 0.95 specifies that constraints are applied to model outputs which are corrected to represent the upper end of the 95% one-sided confidence level of that prediction.

## 8.2 Using PESTPP-OPT

### 8.2.1The PEST Control File

Like other members of the PEST++ suite, execution of PESTPP-OPT is initiated using a command line that references a PEST control file. See chapter 5 of this manual for details. The PEST control file supplied to PESTPP-OPT must define the optimization problem that it must solve. In particular, this PEST control file must inform it of the following:

* model parameters whose post-calibration uncertainties are responsible for the uncertainties of model outputs;
* model outputs for which there are counterparts in the calibration dataset;
* the noise associated with members of the calibration dataset;
* decision variables that must be optimized;
* model outputs to which constraints are applied during the optimization process;
* constraints that are exerted on linear combinations of parameters that do not require a model run to calculate;
* how the decision objective function is defined;
* the values of control variables that govern the constrained optimization process.

As is the normal protocol for members of the PEST++ suite, variables which control the optimization process that is implemented by PESTPP-OPT are supplied through keywords that can be placed anywhere within a PEST control file on lines that begin with the “++” string.

Each of the above issues is now discussed in detail.

### 8.2.2 Decision Variables and Parameters

In a PEST control file, each parameter is assigned to a parameter group. PEST control variables which govern the operation of finite difference derivatives are assigned to these groups. These variables are just as important for PESTPP-OPT as they are for other members of the PEST++ suite (and indeed PEST), as PESTPP-OPT may be required to calculate multiple Jacobian and response matrices using finite parameter differences.

However for PESTPP-OPT, parameter groups play another important role. They provide the means through which entities defined in the “parameter data” section of a PEST control file are separated into decision variables on the one hand and model parameters on the other hand. The former are adjusted through the constrained optimization process that is implemented by PESTPP-OPT. The latter are used in calculation of the uncertainties of model outputs to which optimization constraints are applied; however they are not actually adjusted.

The PEST++ control variable *opt\_dec\_var\_groups()* informs PESTPP-OPT of the names of parameter groups whose members comprise decision variables. A comma-delimited list of parameter group names must be supplied as the value of this control variable. If this variable is omitted from the PEST control file, then PESTPP-OPT assumes that all parameter groups are comprised of decision variables (and hence that there is no uncertainty associated with any model output).

It is by no means essential that every decision-variable be a model input. As will be discussed below, PESTPP-OPT allows arbitrary designation of the objective function defined by equation 8.4. All that is required is that all “parameters” that are used in calculation of the objective function belong to parameter groups that are nominated as containing decision variables. It is good practice to place decision variables which are not used by the model into different groups from those which influence model behaviour. The former are denoted herein as “external” decision variables. Then, when it fills the management response matrix (i.e. the **A** matrix of equation 8.5), PESTPP-OPT knows that it does not need to run the model to calculate finite-difference derivatives with respect to these decision variables.

The names of parameter groups which house external decision variables can be supplied through the *opt\_external\_dev\_var\_groups()* control variable. The value of this variable is a comma-delimited list of the names of parameter groups which hold external decision variables. Parameter groups whose names are supplied through the *opt\_external\_dev\_var\_groups()* keyword must also be supplied through the *opt\_dec\_var\_groups()* keyword; the former are a subset of the latter. It is also important to note that, for the sake of PEST++ protocol consistency, all external parameters must appear in a template file, despite the fact that they are not used by the model. This can be easily accomplished by including their names in a single template file which can be matched (in the “model input” section of the PEST control file) to a model input file which the model never actually reads.

The simplex algorithm that PESTPP-OPT employs to minimize the objective function defined by equation 8.4 is quite different from that used by PEST and PESTPP to minimize the type of objective function that quantifies model-to-measurement misfit. Hence some of the control variables that are pertinent to the latter optimization process are not pertinent to the former process. In particular, decision variables that are adjusted by PESTPP-OPT cannot be log-transformed; however they can be tied or fixed. As they are altered in order to minimize the management objective function, they are not subject to limits imposed by the FACPARMAX and RELPARMAX variables that are featured in the “control data” section of the PEST control file.

### 8.2.3 Defining the Objective Function

The coefficients which are employed in formulating the objective function of equation 8.4 are supplied through the *opt\_obj\_func()* control variable. PESTPP-OPT is informed whether this function must be maximized or minimized through the *opt\_direction()* control variable. The latter must be supplied as either “min” or “max”.

The *opt\_obj\_func()* control variable provides three options for defining the objective function. The first is to supply the name of a prior information equation. This equation must feature all decision variables; the coefficients of these variables in the prior information equation become the coefficients *ci* of equation 8.4b. Alternatively, the name of a file can be provided. In this case PESTPP-OPT reads the coefficient associated with each decision variable from that file. The file must have two columns; entries on each line must be space, tab or comma-delimited. The first entry on each line must be the name of a decision variable while the second must be the coefficient associated with that variable. All decision variables featured in the PEST control file must appear in this external file. Any line that begins with the “#” character is treated as a comment and is therefore ignored. Figure 8.1 exemplifies such a file. The third option for the *opt\_obj\_func* input is to name an observation. This results in the objective function coefficients for the each of the decision variables to be taken from a row in the jacobian matrix – that is, the objective coefficients are the sensitivities of each of the decision variables to the named observation.

|  |
| --- |
| # decision\_variable coefficient  pump\_rate1 3.345  pump\_rate2 3.034  inj\_rate1 4.321  inj\_rate2 4.287  etc. |

Figure 8.1 An external file whose name is supplied with the *opt\_obj\_func()* variable.

If the *opt\_obj\_func()* control variable is not provided in the PEST control file that is featured on the PESTPP-OPT command line, PESTPP-OPT assigns coefficients to decision variables itself; each is assigned a coefficient of 1.0.

### 8.2.4 Constraints

Constraints can be applied on model outputs (read from model output files using instruction files) or on prior information equations. PESTPP-OPT allows constraints to be either “less than” or “greater than” constraints; the latter are internally reformulated as “less than” constraints to meet the demands of the linear programming algorithm that it implements. Constraints are identified using observation groups. If an observation group contains “less than” constraints, then its name must begin with “l\_” (that is, the letter “el” followed by an underscore) or “less\_”; if an observation group contains “greater than” constraints then its name must begin with “g\_” or “greater\_”. Do not forget that prior information equations are also assigned to observation groups.

While the naming of observation groups in this way is sufficient to denote them as containing constraints, you can also nominate groups which contain constraints using the *opt\_constraint\_groups()* control variable. The value of this control variable is a comma-delimited set of observation groups. The name of each such group must begin with “l\_” or “g\_” (or “less\_” or “greater\_”) in accordance with the protocol described above so that PESTPP-OPT knows what type of constraints they are. If an *opt\_constraint\_groups()* keyword is supplied, and if a group name that begins with “l\_” or “g\_” (or “less\_” or “greater”) is omitted from the list of groups supplied through this variable, then members of that group do not comprise optimization constraints. They are therefore observations which comprise part of the calibration dataset (unless the pertinent observations are assigned weights of zero in the PEST control file).

It is important to note that weights assigned in the PEST control file to model outputs to which constraints are applied have no bearing on their operation, unless they are set to zero. If the weight associated with a constraint is set to zero, then the constraint is disabled. However, if the weight is positive, its value does not affect the strength with which the constraint is applied, with one exception: if the *opt\_std\_weights* option is used, then the weight values for the non-zero weight constraints are treated as standard devivations (uncertainty) for use it the chance constraint process. This option can help speed up the repeated application of pestpp-opt by allowing users to pre-calculate constraint uncertainty through application of PREDUNC, pyEMU or through empirical constraint uncertainty estimation via ensembles. In this way, PESTPP-OPT does not need to solve any linear analysis equations and instead simply uses these standard deviations directly.

In the simplex algorithm, all constraints are “hard”. It is the nature of most constrained optimization problems that the solution to that problem lies at a “corner” of decision variable space that is defined by the intersection of constraint surfaces. In that corner, at least one constraint specification is exactly met.

In addition to constraints applied to model outputs and prior information equations, the upper and lower bounds on decision variables supplied in the “parameter data” section of a PEST control file comprise a further set of constraints which are respected by the constrained optimization process that is implemented by PESTPP-OPT. Obviously, these are applied to each decision variable individually.

As presently programmed, PESTPP-OPT requires that at least one constraint be applied to a model output. Hence if the only constraints that are specified in a PEST control file that is read by PESTPP-OPT are those on individual decision variables through their bounds, and/or on prior information equations that feature those decision variables, PESTPP-OPT will cease execution with an appropriate message.

### 8.2.5 Observations

If an observation that is featured in a PEST control file is not denoted as a constraint, then it is used in the notional calibration process through which posterior parameter uncertainties are calculated. (It is important to note that the same does not apply to prior information equations; if a prior information equation does not feature a decision variable, then PESTPP\_OPT simply ignores it.) These posterior parameter uncertainties are used to calculate the uncertainties of model outputs to which constraints are applied; see equations 8.1 to 8.3. The calibration process is described as “notional” because model parameters are not actually adjusted to fit the calibration dataset. Instead, PESTPP-OPT assumes that parameter values have already been adjusted. The posterior uncertainties associated with these parameters are then calculated using equation 8.2. Observations that are employed in the notional calibration process must be featured in the PEST control file that is provided to PESTPP-OPT so that it can calculate the terms of this equation. However it is important to note that, in calculating posterior parameter and model output uncertainties, PESTPP-OPT takes no notice of the values of observations that must be supplied in the “observation data” section of the PEST control file, for these do not appear in equations 8.1 to 8.3; only sensitivities are featured in these equations. The same applies to parameters; hence their values do not need to change in implementing the notional calibration exercise that is embodied in these equations.

The weights assigned to observations that are cited in the “observation data” section of the PEST control are important. These are used in calculating the C(**ε**) matrix that appears in equations 8.1 and 8.2. PESTPP-OPT assumes that each such weight is equal to the inverse of the standard deviation of noise associated with the corresponding observation. Hence each weight is squared, inverted, and then assigned to the appropriate diagonal element of C(**ε**) in these equations. In many instances, the weights that are associated with observations in a PESTPP-OPT control file may therefore be different from those that were used in a preceding model calibration process in which parameters were actually adjusted. Utilities such as PWTADJ2 from the PEST suite can be used to adjust calibration weights so that their magnitudes are commensurate with the level of model-to-measurement fit attained through the calibration process. A good rule of thumb is that weights should be such that the (measurement) objective function attained through the calibration process is roughly equal to the number of non-zero weighted observations featured in that process (collectively or on a group-by-group basis). To remove an observation from the notional calibration process, set its weight to zero.

In some optimization contexts, it may be desirable to set all observation weights to zero. The notional calibration exercise through which a posterior covariance matrix is calculated from a prior covariance matrix is then foregone. In this case, PESTPP-OPT uses the prior covariance matrix to calculate the uncertainties associated with model outputs to which constraints are applied. (This covariance matrix is supplied to PESTPP-OPT in the manner described below.) Alternatively, if all weights are set to zero, a user may provide PESTPP-OPT with a self-calculated posterior parameter covariance matrix in place of the prior parameter covariance matrix for use in calculating parameter and model output uncertainty. For reasons which will be discussed below, this strategy may enhance the efficiency of PESTPP-OPT usage.

### 8.2.6 Regularization

It is recommended practice when calibrating a model to include Tikhonov regularization in the inversion process (often through a suite of prior information equations). Tikhonov regularization expresses expert knowledge as it pertains to parameters. When using PESTPP-OPT, this same expert knowledge is expressed through the prior covariance matrix featured in equations 8.1 and 8.2. All regularization that was employed in a previous calibration exercise should be removed from a PEST control file before using PESTPP-OPT.

### 8.2.7 Prior Covariance Matrix

A prior covariance matrix can be supplied to PESTPP-OPT using the PEST++ *parcov()* control variable; this variable is also used by PESTPP and PESTPP-IES. The name of a covariance matrix file (with extension *.cov*), parameter uncertainty file (with extension *.unc*), or Jacobian matrix file containing a covariance matrix (with extension *.jco* or *.jcb*) can be supplied as the value of this keyword. (See appendix B of this manual for specifications of the first two of these file types.) Variances and covariances featured in this file must pertain to the logs (to base 10) of parameters that are declared as log-transformed in the PEST control file. There is no need to feature any decision variables in this file, as these do not appear in equations 8.1 and 8.2. The user is reminded that parameter covariance matrices can be calculated using PyEMU, as well as the PPCOV, PPCOV3D, PPCOV\_SVA and PPCOV3D\_SVA utilities available from the PEST Groundwater Data Utilities suite.

If a covariance matrix is not supplied, then PESTPP-OPT calculates a prior covariance matrix itself. In doing so, it assumes that parameters are statistically independent, and that the difference between the upper and lower bounds of a parameter (with log transformation taken into account) is equal to 4 standard deviations of its prior probability distribution. (An alternative number of standard deviations can be provided through the *par\_sigma\_range()* control variable.)

### 8.2.8 Risk

Using the *opt\_risk()* control variable, a user specifies his/her disposition with respect to risk. The setting of this variable determines the value of δ*o* discussed in section 8.1. This is the value that is added/subtracted to/from a model output before a constraint is applied to that output.

The value supplied for *opt\_risk()* should be greater than zero and less than one. If the value supplied for *opt\_risk()* is 0.5, then operation of PESTPP-OPT is risk neutral. In this case δ*o* is zero. Under these circumstances PESTPP-OPT does not calculate model output uncertainty at all. It therefore does not need to read (or calculate for itself) a prior parameter covariance matrix. Also, it does not need to read (or calculate for itself) derivatives of model outputs with respect to model parameters which are not decision-variables.

An *opt\_risk()* setting of greater than 0.5 indicates risk aversion. For “less than” constraints (i.e. constraints for which a system state or flux must be less than a certain value), δ*o* is added to the model-calculated system state or flux (i.e. *o*) so that the constraint is applied to *o* + δ*o*. The opposite applies for “greater than” constraints. Suppose, for example, that *opt\_risk()* is supplied as 0.95. Then, for a “less than” constraint, δ*o* is calculated to be such that there is a 95% chance that the real system state or flux is less than *o* + δ*o*. Similarly, for a “greater than” constraint, δ*o* is calculated to be such that there is a 95% chance that the real system state or flux is greater than *o* – δ*o*. In both of these cases *o* is the value that the model calculates for this quantity based on the current values of decision variables.

A setting for *opt\_risk()* that is less than 0.5 indicates risk tolerance. For “less than” constraints, the constraint is applied to *o* - δ*o*. The opposite applies for “greater than” constraints. Suppose that *opt\_risk()* is supplied as 0.05. Then, for a “less than” constraint, δ*o* is calculated to be such that there is a 5% chance thatthe real system state or flux corresponding to a certain set of decision variables is less than *o* - δ*o*. Similarly, for a “greater than” constraint, δ*o* is calculated to be such that there is a 5% chance that the real system state or flux corresponding to a certain set of decision variables is greater than *o* + δ*o*.

### 8.2.9 Jacobian and Response Matrices

During every iteration of the constrained optimization process, PESTPP-OPT calculates derivatives of model outputs to which constraints are applied to decision variables whose values are optimized. In accordance with the normal PEST/PEST++ protocol, control variables which govern calculation of finite-difference derivatives are read from the “parameter data” section of the PEST control file.

Unless *opt\_risk()* is set to 0.5, PESTPP-OPT must obtain partial derivatives which comprise the **J** and **y** vectors appearing in equations 8.1 to 8.3. The **J** matrix contains partial derivatives of calibration-relevant model outputs to parameters featured in the PEST control file which are not decision variables. The **y** vector contains partial derivatives of model outputs to which constraints are applied to these same parameters. These partial derivatives are also calculated using finite parameter differences.

Unlike derivatives of model outputs with respect to decision variables, derivatives of model outputs with respect to model parameters do not necessarily need to be re-calculated during every iteration of the PESTPP-OPT optimization process. In fact PESTPP-OPT can calculate values for δ*o* using a set of derivatives that are supplied externally, at the start of the optimization process (see below). It is possible, however, that sensitivities of model outputs to parameters will change with the values of decision variables. If a user is worried about this, he/she can inform PESTPP-OPT that it should update the **J** matrix and **y** vector every now and then. The iteration update interval is set through the *opt\_recalc\_fosm\_every()* control variable. This must be provided as an integer. If it is set to 1, then **J** and **y** are updated at the beginning of every iteration of the constrained optimization process. If it is set to 2, then **J** and **y** are calculated at the beginning of the constrained optimization process (if they are not supplied externally), and then at the beginning of every second iteration of that process. A similar protocol applies for higher values of this variable. On the other hand, if *opt\_recalc\_fosm\_every()*is set to an exceedingly high value, then **J** and **y** are not updated at all. Nor, therefore, is the value for δ*o* updated for all model outputs to which chance constraints are applied.

It may be possible to avoid calculation of at least some partial derivatives through use of the *base\_jacobian()* control variable that is described in documentation for PESTPP (see section 6.2.7). Recall that the value of this variable is the name of a binary Jacobian matrix file (with extension *.jco* or *.jcb*). If this file provides sensitivities for all calibration-relevant model outputs with respect to all model parameters, then PESTPP-OPT does not need to undertake any model runs for calculation of **J** during the first iteration of the optimization process. If it also includes sensitivities of constraint-relevant model outputs to all model parameters, then calculation of **y** during the first iteration of the optimization process is also avoided. If it also includes sensitivities of constraint-relevant model outputs to all decision variables, then calculation of optimization sensitivities during the first iteration of the optimization process is precluded. After reading an external Jacobian matrix, PESTPP-OPT works out for itself what model runs must be undertaken during the first iteration of the optimization process to calculate partial derivatives that are missing from this matrix.

Filling of the **J** matrix of equations 8.1 and 8.2 can also be avoided if weights assigned to all calibration-relevant observations in the “observation data” section of the PEST control file are set to zero, or if no calibration-relevant observations are included in this section at all. This signifies to PESTPP-OPT that the model is uncalibrated. PESTPP-OPT then uses prior parameter uncertainties, rather than posterior parameter uncertainties, for calculation of δ*o* values for constraint-relevant model outputs. Using the *parcov()* control variable, a user may wish to supply a covariance matrix to PESTPP-OPT instead of letting PESTPP-OPT calculate prior parameter uncertainties itself from parameter bounds. Under these circumstances he/she may wish to provide PESTPP-OPT with a posterior parameter covariance matrix instead of a prior covariance matrix. Because PESTPP-OPT “thinks” that this is a prior parameter covariance matrix, and because it has been informed that this matrix does not need modification in accordance with the notional calibration exercise that is embedded in equations 8.1 and 8.2, it simply uses this matrix for calculation of δ*o*; it does not expend model runs to calculate **J**. This strategy can speed up the optimization process considerably, at the same time as it ensures that δ*o* is calculated using post-calibration uncertainties.

### 8.2.10 Solution Convergence

Notwithstanding the nonlinear nature of most models, the constrained optimization problem that is solved by PESTPP-OPT is formulated as a linear problem. Model nonlinearities are accommodated by solving this problem in a progressive fashion through a series of iterations in which sensitivities to decision variables are re-calculated during every iteration. This sequential linear programming (SLP) process is deemed to be complete when neither the objective function, nor any decision variable, changes by more than a certain (small) amount from one iteration to the next. This amount is supplied by the user as the value of the *opt\_iter\_tol()* control variable. PESTPP-OPT provides a default value of 0.001 for this variable.

As was stated above, PESTPP-OPT uses the open source CLP library supplied by the Computational Infrastructure for Operational Research (COIN-OR) project. This algorithm provides a history of the SLP solution process, with a level of detail that is set by the calling program. A PESTPP-OPT user can gain access to this history using the *opt\_coin\_log()* control variable. An integer in the range 1 to 4 must be supplied for its value, with a higher value requesting greater verbosity. The record is written to a file named *case.coin\_log* where *case.pst* is the PEST control file whose name is supplied on the PESTPP-OPT command line.

Nonlinearities of constraint-relevant model outputs with respect to parameters which are not decision variables can be accommodated through intermittent re-calculation of **J** and **y** during the SLP process. However it is important to keep in mind that this strategy constitutes only partial accommodation of this type of nonlinearity, as model parameters which are not decision-variables are not actually varied from iteration to iteration of the SLP process. Re-calculation of **J** and **y** accommodates only the effect that changes in the values of decision variables have on these sensitivities. It does not accommodate changes in **J** and **y** that may be incurred by variability of model parameters over ranges denoted by their posterior uncertainties. Nor does it accommodate the fact that equations 8.1 to 8.3 assume model linearity with respect to these parameters.

### 8.2.11 Other Control Variables

In common with all other members of the PEST++ suite, a PEST control file used by PESTPP-OPT can include variables that govern parallel run management. See section 5.3 of this manual.

### 8.2.12 Final Model Run

Once it has completed the constrained optimization process, PESTPP-OPT undertakes one final model run in which it employs optimized values of all decision variables. The optimized values of these variables thus remain on model input files when PESTPP-OPT ceases execution; model output files reflect these inputs. Undertaking of the final model run can be foregone through the use of the aptly named *opt\_skip\_final* option.

### 8.2.13 Restarts

As presently programmed, a prematurely-terminated PESTPP-OPT run cannot be restarted. However, PESTPP-OPT does support use of the *hotstart\_resfile()* and *base\_jacobian()* control variables. Strategic use of functionality provided by these variables effectively constitutes a restart.

PESTPP-OPT offers an advanced functionality to solve the chance-constrainted LP problem without requiring any additional model runs. If users activate the *base\_jacobian*, *hotstart\_resfile*and *opt\_skip\_final* options and the control variable NOPTMAX is set to 1, then PESTPP-OPT will not make any model runs. Instead, it will solve the chance constrainted linear programming problem described in the control file, report optimal decision variable values and final objective function, then terminate. This can be a very useful option for exploring the implications of changing decision variable bounds, constraints, risk, and the components related to the chance constraints: number of and prior uncertainty of parameters contributing to constraint uncertainty, number and weight of observations used to condition parameters, etc.

## 8.3 PESTPP-OPT Output Files

Files recorded by PESTPP-OPT are listed in the following table. The contents of this table are based on the assumption that the PEST control file on which constrained optimization is based is named *case.pst*.

|  |  |
| --- | --- |
| **File** | **Contents** |
| *case.rec* | The run record file. This file begins by echoing information provided in the PEST control file. It then records the history of the constrained optimization process undertaken by PESTPP-OPT. Optimization results are recorded at the end of this file. |
| *case.par* | A parameter value file. This file records best parameters that have been calculated up to any stage in the constrained optimization process. Note that only the values of parameters which are identified as decision variables change from iteration to iteration. |
| *case.N.par* | A parameter value file listing parameters calculated at the end of iteration *N* of the constrained optimization process. |
| *case.res* | A residuals file, recorded at the end of the PESTPP-OPT constrained optimization process. This is identical to the file of the same named recorded by PESTPP at the end of an inversion process. Note that calibration-pertinent model outputs do not change from the beginning to the end of the constrained optimization process, as parameters that are not decision variables are not altered by PESTPP-OPT. However model outputs to which constraints are applied change as decision variables are updated. |
| *case.N.rei* | A residuals file calculated using the values for decision variables available at the end of optimization iteration *N*. |
| *case.pfm* | Performance log. This file is similar to those recorded by other members of the PEST++ suite. It records the times at which PESTPP-OPT began and ended various processing tasks. |
| *case.coin\_log* | Log of operations performed by the public domain SLP optimization algorithm. |
| *case.rmr* | Parallel run management record file. This is recorded if PESTPP-OPT undertakes model runs in parallel. |
| *case.rnj* | A binary file used for run management. |

Table 8.1 PESTPP-OPT output files. It is assumed that the name of the PEST control file is *case.pst*.

## 8.4 Summary of Control Variables

Table 8.2 tabulates PEST++ control variables used by PESTPP-OPT. All of these are optional. If a particular control variable is not supplied, then PESTPP-OPT provides a default value. Where appropriate, the value of the default is presented with the name of the variable in the table below. Variables discussed in section 5.3.6 that control parallel run management are not listed in the following table.

Note also that the number of control variables may change with time. Refer to the PEST++ web site for variables used by the latest version of PESTPP-OPT.

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Role** |
| *opt\_dec\_var\_groups()* | text | Comma-delimited string identifying which parameter groups are to be treated as decision variables. If not supplied, all adjustable parameters are treated as decision variables. |
| *opt\_external\_dec\_var\_groups()* | text | Comma-delimited string identifying which parameter groups are to be treated as "external" decision variables, that is decision variables that do not influence model outputs and that therefore do not require a finite-difference run of the model to fill the pertinent column of the response matrix. |
| *opt\_constraint\_groups()* | text | Comma- delimited string identifying which observation and prior information groups are to be treated as constraints. Group names for "less than" constraints must start with "l\_" or “less\_”; group names for "greater than" constraints must start with "g\_" or “greater\_”. If this control variable is omitted, all observation and prior information groups that meet these naming conventions are treated as constraints. |
| *opt\_obj\_func()* | text | String identifying the prior information equation or two-column ASCII file that contains coefficients used in formulation of the objective function (see equation 8.4). If this control variable is not supplied, then each decision variable is given a coefficient of 1.0 in formulation of the objective function. |
| *opt\_direction(min)* | text | Either "min" or "max". “min” specifies that the objective function be minimized, while “max” specifies that it be maximized. |
| *opt\_risk(0.5)* | real | A number between 0.0 and 1.0. A value of 0.5 signifies risk neutrality. A value of 0.95 seeks a 95% risk averse application of optimization constraints, while a value of 0.05 seeks a 5% risk tolerant application of optimization constraints. |
| *opt\_recalc\_fosm\_every(1)* | integer | Number of iterations of the SLP process over which FOSM-based chance constraints are re-used. If set to 1, a calibration Jacobian matrix is calculated during every iteration of the SLP constrained optimization process. |
| *parcov()* | text | Provide the name of a JCO, JCB, UNC or COV file from which the prior covariance matrix used in FOSM analysis is read. |
| *par\_sigma\_range(4.0)* | real | The difference between a parameter’s upper and lower bounds expressed as standard deviations. |
| *opt\_iter\_toll(0.001)* | real | Solution closure criterion applied to objective function and decision variables. |
| *base\_jacobian()* | text | Provide the name of a Jacobian matrix file (with extension *.jco* or *.jcb*). Sensitivities read from this file are used for the first iteration of the constrained optimization process. |
| *hotstart\_resfile()* | text | The name of a residuals file produced by PESTPP or PESTPP-OPT. PESTPP-OPT assumes that model output values contained in this file correspond to the values of parameters (including decision variables) listed in the PEST control file. Hence it does not carry out the initial model run. |
| *opt\_coin\_log(1)* | integer | Level of verbosity of solution information recorded by optimization library functions. |
| *opt\_std\_weights* | boolean | Flag that indicates constraint weights are standard deviations. This causes PESTPP-OPT to skip the FOSM-based constraint uncertainty calculation and use the weights directly in the calculation of risk. These standard deviations can be calculated externally via PREDUNC or pyEMU or can be derived empirically from an ensemble. |
| *opt\_skip\_final* | boolean | Flag to skip the final model run. |

Table 8.2 PESTPP-OPT control variables. Parallel run management variables can be supplied in addition to these. See section 5.3.6.

# 9. PESTPP-IES

## 9.1 Introduction

### 9.1.1 Publications

PESTPP-IES is described by White (2018). The reader is referred to that paper for a description of what it does and how it works, together with an example of its use. Chen and Oliver (2013) describe in detail the theory on which the iterative ensemble smoother methodology implemented by PESTPP-IES rests; they provide further deployment examples.

### 9.1.2 Overview

Predictions made by environmental models are accompanied by uncertainty. This applies particularly to models of fluid flow and mass transport through the subsurface, where hydraulic properties and system stresses are often only poorly known. Where models are used to support management decisions, the uncertainties associated with their predictions should be quantified. Unless this is done, it is not possible to assess the range of possible outcomes of a particular management action. Without knowledge of the risks associated with an envisioned management strategy, the basis for a decision to adopt it may be flawed.

Model predictive uncertainty can arise from many sources. These include the conceptual basis of the model itself. In the present discussion we ignore this. We assume that model predictive uncertainty can be explored by sampling the probability distribution of its parameters, so that model predictive uncertainty arises only from model parameter uncertainty (where the definition of parameters includes specification of system stresses).

Theoretically, model parameter uncertainty can be quantified using Bayes equation. In its simplest form, this can be expressed as follows.

*P*(**k**|**h**) = *P*(**h**|**k**)*P*(**k**) (9.1)

In equation 9.1 model parameters comprise the elements of the vector **k**. Meanwhile calibration data comprise the elements of the vector **h**. The symbol *P*() signifies a probability distribution. *P*(**k**) is known as the prior parameter probability distribution. This expresses pre-calibration parameter uncertainty. That is, it expresses the uncertainties in parameters that accompanies expert knowledge of their values, supplemented by direct measurements of system properties at a number of discrete points in space and/or time. *P*(**h**|**k**) is known as the “likelihood function”. This increases with goodness of fit between model outputs and members of the calibration dataset. It thus acts as a kind of a “filter” on the prior probability distribution whereby only those parameters that allow the model to fit the calibration dataset are transferred to the left side of equation 9.1 where *P*(**k**|**h**) represents the posterior parameter probability distribution, that is, the probability distribution of **k** conditioned by **h**.

Where models are used to support environmental decision-making, it is normally the posterior parameter probability distribution that is of most interest. This is because most environmental models are deployed in contexts where there is at least some data available for history-matching. Unfortunately, it is rarely possible to depict the posterior parameter probability distribution using a mathematical formula. Hence it must be inferred through sampling it. When the model is then used to make a prediction, that prediction is made using all of these samples. The posterior probability density function of the prediction is thereby sampled as well.

PESTPP-IES calculates samples of the posterior parameter probability distribution. It is not the only numerical methodology that provides this service. However it has properties which allow it to serve this purpose well in many modeling contexts. Unlike FOSM methods (such as those implemented by PESTPP, PyEMU and programs of the PEST PREDUNC suite), PESTPP-IES’s sampling of the posterior parameter probability distribution is not compromised by an assumption of model linearity. At the same time, where parameter numbers are moderate to high, the numerical burden incurred by use of PESTPP-IES is far smaller than that incurred by use of popular Bayesian sampling methodologies such as Markov chain Monte Carlo.

The numerical algorithm that is implemented by PESTPP-IES is referred to as an “iterative ensemble smoother”. The job of an ensemble smoother is to assimilate data in a “batch” sense. (This in contrast to recursive data assimilation that is carried out by the familiar ensemble Kalman filter.) For PESTPP-IES this data comprises a calibration dataset. PESTPP-IES implements an approximate form of Bayes equation wherein efficiencies are gained by combining Bayesian methods with subspace methods. These efficiencies reduce the computational burden of giving effect to Bayesian principles enormously, especially where parameter numbers are high. In fact, beyond a certain threshold set by the information content of measurements comprising a calibration dataset, its numerical burden is almost insensitive to the number of parameters that are employed by a model. This number can thus rise to the hundreds of thousands, or even millions, with little extra computational cost. Hence hydraulic property heterogeneity can be represented at the grid of cell level in a groundwater or subsurface reservoir model.

A PESTPP-IES user commences the model parameterization process with a suite of random parameter fields sampled from the prior parameter probability distribution. These must be generated using an appropriate random parameter field generator. Where parameters are based on pilot points, this may be the RANDPAR or RANDPAR1 utilities available with the PEST suite. PyEMU provides similar functionality. Alternatively, cell-by-cell parameter heterogeneity may be introduced with the help of geostatistical software such as SGEMS (Remy et al, 2011). In accordance with normal PEST and PEST++ protocols, all parameters must be given a name, regardless of their number; these names must be recorded in the “parameter data” section of a PEST control file, and in template files that are cited in the PEST control file.

The suite of initial parameter fields comprises an “ensemble”. In general, the greater the number of parameter fields that comprise this ensemble, the better is the performance of PESTPP-IES. In practice however, the number of realizations is limited by computing resources. A good rule of thumb is to ensure that the number of parameter fields comprising the ensemble exceeds the dimensionality of the solution space of the inverse problem that is posed by the history-matching process. This ensures that the iterative ensemble smoother has access to the directions in parameter space that it requires in order to provide a good fit with the calibration dataset. Normally, this number can only be guessed. However if a Jacobian matrix is available, the SUPCALC utility from the PEST suite, or functionality available through PyEMU, can be used to assess solution space dimensionality.

Through a series of successive iterations, PESTPP-IES modifies the members of the ensemble such that they all conform with calibration constraints. That is, it adjusts each member of the ensemble such that, when the model is run using each of them, model-to-measurement misfit is good enough for the model to be considered as “calibrated”. The outcome of this multiple parameter field adjustment process is thus an ensemble of parameter fields which can be considered to be samples of the posterior parameter probability distribution.

It is apparent that use of PESTPP-IES constitutes a significant departure from the “calibrate first and do uncertainty analysis later” approach that underpins much environmental model history matching. PESTPP-IES does not seek a parameter field that is deemed to “calibrate” a model. Instead, it seeks a suite of parameter fields which collectively express posterior parameter uncertainty. Astonishingly, as will be discussed below, the numerical cost of obtaining this suite is often low compared with that required to obtain a single “calibrated” parameter field of post-calibration minimum error variance, particular where parameter numbers are high (which is often required to avoid underestimation of predictive uncertainty).

### 9.1.3 Ensemble Kalman Filters and Ensemble Smoothers

The Kalman filter is widely used in time series processing and in engineering control. It provides an efficient means of processing noisy measurement data in order to provide increasingly better estimates of system state, and of parameters that govern processes which determine system state.

Typically, when deployed to process environmental data, the Kalman filter is used in conjunction with an environmental model that calculates a series of new system states over a series of successive time steps; this process is referred to as “recursive data assimilation”. The calculated state at any time step depends on the previous state of the system, and on the properties of the system, as embodied in model parameters. Model-calculated quantities that are dependent on system state are then compared with corresponding real world measurements; the latter are invariably accompanied by noise. The Kalman filter then corrects the model-represented system state, as well as the parameters that were used in its calculation. The corrected system state and parameters are then used as a basis for calculation of the next system state, where the process is repeated.

The mathematics through which corrections to system states and parameters are computed from field measurements is very similar to that employed for parameter estimation. See section 6.2.1. However alternative forms of these equations are often employed. These are better equipped to deal with the large number of system states and parameters which must be adjusted on the basis of information gleaned from what is often a parsimonious measurement dataset.

Sensitivity matrices replace nonlinear model operators in determining adjustments that the Kalman filter must make to parameters and system states. In many Kalman filter applications these are calculated analytically or using relatively fast adjoint methods. Sensitivities are updated during every time step.

While the mathematical underpinnings of the iterative ensemble smoother are similar to those of the Kalman filter, the focus of the smoother is on the estimation of parameters rather than on the simultaneous estimation of parameters and updated system states. Furthermore, while the Kalman filter seeks to recursively estimate states and/or parameters, assimilating information one step at a time, the ensemble smoother works in “batch” mode, running the model over the entirety of its simulation time (in the same way that PEST and PESTPP do). Moreover its formulation of the inverse problem allows estimation of a virtually unlimited set of parameters, provided the matrices that are required for this estimation can be quickly filled. This requires a rapid means of computing sensitivities. Because the equations which are used by the ensemble smoother are rooted in Bayesian theory, an outcome of its use is a characterization of post-calibration parameter uncertainty. Unfortunately however, where a model is highly nonlinear, this characterization is often flawed because Bayes-theorem-derived parameter adjustment equations assume model linearity. Hence samples of the posterior parameter probability distribution that is characterized using these sensitivities may not provide acceptable fits with the calibration dataset when used by the model.

This situation can be remedied by solving the ensemble smoother equations iteratively, while applying them to individual parameter fields comprising an ensemble. Parameter field ensembles actually play a number of important roles in this process. Their initial task is to sample the prior parameter probability distribution. The ensemble smoother then adjusts each member of the ensemble individually until it constitutes an approximation to a sample of the posterior parameter probability distribution. At the same time, the ensemble of evolving, random parameter fields is used by the smoother to estimate sensitivities (i.e. partial derivatives) that are then used to further adjust individual members of the ensemble so that they provide a better fit with the calibration dataset.

It is apparent from the above description that, unlike PESTPP and PEST, PESTPP-IES does not calculate derivatives using finite parameter differences. Instead, by running the model using each member of the parameter ensemble, it calculates approximate partial derivatives from cross-covariances between parameter values and model outputs that are calculated using members of the ensemble. The same model runs that are used to test the efficacy of upgrades to each member of the ensemble are used to calculate an approximate set of sensitivities which then form the basis for the next parameter ensemble upgrade.

### 9.1.4 Some Repercussions of Using Ensembles

The most obvious advantage of using ensembles is their ability to sample the posterior parameter probability distribution. This is critical to a model’s support for environmental decision-making.

There are a number of numerical advantages as well. While partial derivatives calculated using model runs based on random parameters are only approximate, they are often good enough to support a parameter ensemble adjustment process that achieves a good fit between model outcomes and field measurements. Furthermore, because the number of parameter fields that comprises an ensemble need only be as great as the dimensionality of the inverse problem solution space, the number of model runs required to obtain a useable Jacobian matrix to serve as the **Z** matrix in equations similar to 6.3 can be far fewer than the number of parameters that actually feature in the ensembles. In contrast, when using finite parameter differences to fill a Jacobian matrix, the number of required model runs is equal to the number of parameters.

Another advantage that accrues from use of a randomized Jacobian matrix in place of a finite-difference Jacobian matrix is that the Jacobian-filling process does not require that model outputs have the same numerical precision as that which is required for calculation of derivatives through finite differences. It may therefore be possible to loosen numerical solver convergence settings employed by the model; model run times may fall as a consequence. In addition to this, the chance of being trapped in a local objective function minimum is lowered as partial derivatives that are approximated using an ensemble are at least partially informed by broad scale parameter gradients in addition to local parameter gradients.

The use of an ensemble to calculate partial parameter derivatives does not come without a cost however. Because partial derivatives are approximate, it may not always be possible to achieve as good a fit with the calibration dataset for some or all members of an ensemble as that which can be achieved through adjustment of a single parameter field using partial derivatives calculated through finite parameter differences. Another problem is that even though the number of ensemble members may exceed the dimensionality of the inverse problem solution space, this does not forestall inadvertent entrainment of null space parameter components during the parameter adjustment process. This may introduce bias to some ensemble members. Both of these problems can be reduced by increasing the number of parameter fields that comprise the ensemble. Obviously, this comes at a numerical cost.

Another issue associated with the use of random parameter fields is that of model stability. If the model (or models) that is run by PESTPP-IES experiences numerical difficulties when provided with some parameter sets, then use of an ensemble of random parameter fields is likely to trigger occasional model run failures, or occasional simulations whose run times are excessively long. As will be discussed below, PESTPP-IES includes functionality to accommodate this problem.

### 9.1.5 Iterations

The parameter adjustment algorithm that is implemented by PESTPP-IES is referred to as an “iterative ensemble smoother”. This reflects the fact that the process through which a prior ensemble becomes a posterior ensemble requires a number of iterations of parameter field adjustment, these iterations being necessary to deal with nonlinearity of the parameter-to-model output relationship. During each of these iterations, a new approximation to the Jacobian matrix is calculated based on the latest ensemble. Upgrades to parameter sets are then calculated using this matrix. The process is then repeated.

An important prerequisite to success of the parameter upgrade process is use of an appropriately-valued Marquardt lambda. The Marquardt lambda plays the same role in ensemble parameter field adjustment as it does in single parameter field adjustment such as that undertaken by PESTPP and PEST. A high value of lambda results in calculation of a damped (i.e. shortened) parameter upgrade vector that is aligned with the gradient of the objective function. This promulgates rapid improvement in the objective function where the latter is far above its minimum. A low value of lambda promulgates better navigation of an objective function surface in which the minimum lies at the bottom of a narrow valley, the existence of which is an outcome of a high degree of post-calibration parameter correlation. Ideally the Marquardt lambda should fall from a high value to a low value as the iterative parameter adjustment process progresses. Like PEST and PESTPP, PESTPP-IES employs a trial and error procedure to find the best value of the Marquardt lambda to employ at any iteration of the ensemble-based inversion process. However, numerical efficiency precludes testing of multiple lambdas when adjusting each parameter field in the entire ensemble, as this would incur an unduly large computation burden. Instead, lambda values are tested using only a few parameter fields. Once the best lambda is identified through this process, the remaining members of the ensemble are adjusted using this lambda. As was stated above, these same runs that are used for testing parameter upgrades are also used for calculation of a new Jacobian matrix.

### 9.1.6 Measurement Noise

Variability between parameter fields that sample the posterior parameter probability distribution (and hence quantify the uncertainty of this distribution) arises from three sources. The first is prior parameter uncertainty. The second is a lack of information in the calibration dataset by which to reduce this uncertainty. The third arises from the fact that the calibration dataset is contaminated by measurement noise.

Presumably, the real-world calibration dataset is accompanied by one realization of measurement noise. Another set of measurements (if they were available) would be accompanied by another realization of measurement noise. Unfortunately, this alternative measurement dataset is not available. Nevertheless, realizations of measurement noise are needed if the effect of this noise on parameter uncertainty is to be taken into account. Hence they must be manufactured as part of the ensemble smoother parameter adjustment process.

Because the ensemble smoother must employ realizations of both random parameter fields and of measurement noise to explore posterior parameter (and hence predictive) uncertainty, a slightly different “calibration dataset” is used for adjustment of each random parameter field. This dataset is altered from the original calibration dataset through addition to the latter of a realization of measurement noise. To generate these realizations, PESTPP-IES uses the weights associated with measurements listed in the “observation data” section of the PEST control file as a statistical descriptor of noise stochasticity. It assumes that the weight assigned to each observation is the inverse of the standard deviation of measurement noise associated with that observation. Alternatively, as is further discussed below, a user can supply a comma-delimited file (i.e. a CSV file) containing observation values with noise already added according to his/her specifications.

Because of the role that weights play in the ensemble smoother parameter adjustment process, a PESTPP-IES user may need to pay more attention to weights when preparing a PEST control file for the use of PESTPP-IES than when preparing a PEST control file for the use of PESTPP or PEST. For the latter two programs, correct relativity of weights is all that matters. For PESTPP-IES the absolute values of weights are of great importance.

If weights are “correct” in an absolute sense, the objective function achieved through the calibration process should be roughly equal to the number of observations comprising the calibration dataset. This is because each weighted residual squared will, on average, be equal to 1.0. On many occasions of PESTPP-IES usage, the final objective function is not known ahead of the parameter adjustment process. Hence it must be guessed in order to support calculation of a suitable set of measurement weights. Fortunately however, an incorrect guess rarely matters; this is especially the case where parameter uncertainty arises predominantly from lack of information in the calibration dataset, and hence is null-space dominated.

### 9.1.7 Regularization

The term “regularization” generally describes numerical devices that are employed to achieve uniqueness of ill-posed inverse problems. Regularization is thus fundamental to the notion of calibration. If regularization is properly implemented, the calibrated parameter field is one of minimized error variance. In most calibration contexts, regularization eliminates from the calibrated parameter field any heterogeneity that is not supported by direct measurements of hydraulic properties or by the calibration dataset.

In the iterative ensemble smoother process implemented by PESTPP-IES, regularization has a related, but slightly different purpose; it is used to constrain parameter realizations comprising an ensemble to remain as close as possible to their initial (purely stochastic) values as they are adjusted in order for model outputs calculated using these realizations to achieve a good fit with the calibration dataset. Regularization as implemented by PESTPP-IES thus attempts to ensure that parameters comprising each realization are changed from their initial values by the smallest amount required for model outputs to reproduce field observations “acceptably” well. “Acceptably” is defined by the user through the *ies\_reg\_factor()* control variable; see below. Regularisation, as applied to a parameter field ensemble in this fashion, thus attempts to ensure that the ensemble retains as much of its stochastic character as possible. The resulting algorithm resembles the randomized maximum likelihood methodology described by Oliver et al (2008).

As is further discussed below, implementation of this type of regularization can only be approximate. At the same time, its use may require more iterations of the iterative ensemble smoother process than would otherwise be required to attain a good fit with the calibration dataset. It thus carries a numerical cost.

It is important to note that if PESTPP-IES is provided with a PEST control file that instructs PEST to run in “regularization” mode (through provision of a “regularization” setting for the PESTMODE control variable), it ignores this setting. Furthermore, whether or not the PEST control file instructs PEST to run in “regularization” mode, PESTPP-IES ignores any prior information that it finds in this file.

### 9.1.8 Base Realization

Optionally (and by default), the parameter ensemble used by PESTPP-IES can include a “base realization”. Parameter values that comprise this realization are those which are listed in the “parameter data” section of the PEST control file which PESTPP-IES reads on commencement of execution. Ideally, this set of parameter values are those of minimum pre-calibration error variance; that is, they comprise the expected values of parameters from an expert knowledge point of view. PESTPP-IES pairs this realization with an observation dataset that has no “manufactured” measurement noise associated with it; this dataset is comprised of measurements that appear in the “observation data” section of the PEST control file.

By monitoring the fate of the parameter set comprising this base realization, a user can witness the effect that the ensemble smoother process has on a non-random parameter field. Any bias that is introduced to this parameter field, or any incredulous heterogeneity that is introduced to this parameter field, is also presumably introduced to the other random parameter fields which comprise the ensemble. Inspection of this field can aid a modeller in assessing whether, in his/her opinion, the parameter field ensemble that emerges from the ensemble smoother process comprises a legitimate and symmetric sample of the posterior parameter probability distribution.

### 9.1.9 Parameter Transformation Status

The number of individual parameters which comprise a realization is equal to the number of parameters that are listed in the “parameter data” section of a PEST control file. However parameters that are fixed in the “parameter data” section of the PEST control file are never varied from their initial values. Tied parameters maintain a fixed ratio with parent parameters, this ratio being determined by values provided for respective parameters in the “parameter data” section of the PEST control file. Note however that if maintenance of this ratio takes a tied parameter beyond its bound (as provided in the PEST control file), then the bound is ignored, regardless of the setting of the *ies\_enforce\_bounds()* control variable; see below. (PEST and PESTPP treat tied parameters in the same way.)

Where a user provides his/her own ensembles (see below), there are circumstances where he/she may wish to provide values for fixed parameters which differ from realization to realization within the ensemble, and which differ from those supplied in the PEST control file. PESTPP-IES allows this. Within each member of the ensemble, PESTPP-IES maintains all fixed parameters at their initial values as it adjusts the values of all other parameters. Note also that if a user-supplied realization omits values for fixed or tied parameters, then PESTPP-IES provides them with values from the PEST control file.

As is usual practice in the PEST++ suite, if a parameter is designated as log-transformed in the “parameter data” section of a PEST control file, then adjustments that are made to the value of the parameter are actually made to the log of the parameter’s value. This takes place behind the scenes, and is invisible to the user. Strategic log-transformation of parameters can render a nonlinear problem much more linear; this can enhance the speed with which the objective function is reduced as parameter realizations comprising an ensemble are adjusted. In fact, most parameters are better log transformed than untransformed; nevertheless, sometimes log-transformation is inappropriate, for example if a parameter can adopt both positive and negative values, or if it represents the value of a quantity whose datum is arbitrary.

### 9.1.10 Comparative Observations

PESTPP-IES introduces a special observation type that is not available in PESTPP or PEST, but resembles constraints supported by PESTPP-OPT. This is the “one way observation” type. For observations of this type, a residual is zero unless model outputs are either greater than or less than its “measured” value; the user specifies which of these apply. This reflects the nature of some types of measurements. However their use is broader than this. “Greater than” and “less than” observations can comprise a powerful mechanism for inserting “soft knowledge” into the history matching process.

If an observation belongs to an observation group whose name begins with the string “g\_” or “greater\_”, this observation is a “greater than” observation. No objective function penalty is incurred if the modelled value of the pertinent quantity is greater than its observed value as listed in the “observation data” section of the PEST control file. However if the model-calculated value is less than the observed value, the objective function penalty is calculated in the usual manner, that is as the squared residual times the squared weight.

Similarly, if an observation belongs to an observation group whose name begins with the string “l\_” or “less\_”, then this observation is a “less than” observation. No objective function penalty is incurred if the modelled value of the pertinent quantity is less than the measured value listed in the “observation data” section of the PEST control file. However if the model-calculated value is greater than the measured value, the objective function penalty is calculated in the usual manner, that is as the squared residual times the squared weight.

### 9.1.11 Localization

Calculating the empirical cross-covariance (sensitivity) between large numbers of parameters and observations from a limited number of realizations will likely result in spurious cross-correlations, and as such, certain parameters will be adjusted when they shouldn’t be. Furthermore, when large numbers of unique (independent) obseravtions are used for history matching, a small ensemble size will almost certaintly not provide enough degrees of freedom to reproduce these data. To combat these problem, users can employee localization. The term “localization” comes from the Ensemble Kalman filter parlance and refers to only allowing “local” covarainces to inform unmeasured states in a spatially-distributed filtering problem. PESTPP-IES supports localization through use of a localization matrix. This matrix has rows that are observation names and/or observation group names and columns that are parameter names and/or parameter group names. Elements of the matrix should range between zero and one. Figure 9.1 shows an example localization matrix. In this example, a mixture of observation names and an observation group (flx\_river) are used for row names and parameter group names are used for the column names. The parameter groups “r1” and “w1” are parameters representing future recharge and pumping, respectively. In this case, the localization matrix “zeros out” and spurious sensitivity between historical observations and future recharge and pumping. If a localization matrix is specified, PESTPP-IES builds up the upgrade matrices for each lambda value sequentially by each row of the localization matrix.

When applying localization in a history matching problem with large numbers of parameters and observations, users may wish to define a “local” neighbourhood around each observation location where parameters are expected to influence simulated equivalents to observations. This, in effect, creates a “local” history matching problem using a subset of adjustable parameters and obseravtions. The outcome of this local problem is a great increase in the degrees of freedom of the inverse problem, effectively allowing a small ensemble size to better reproduce large numbers of unique observations. In this case, the localization matrix will likely contain one column for each adjustable parameter and each row will be a single observation (location) for point-type observationss. If large numbers of parameters are used, then the upgrade calculation process for a given lambda will require as many truncated SVD solves as there are adjustable parameters, which can take considerable wall time. To combat this, the localized upgrade solution process in PESTPP-IES has been multithreaded (since each local solve is independent). Users can opt to use multiple threads through the *ies\_num\_threads* argument. Note, the optimal number of threads to use is problem specific, but should not exceed the number of physical cores of the host machine that the master is running on.

|  |
| --- |
| 4 5 2  1.0 1.0 0.0 1.0 0.0  1.0 1.0 0.0 1.0 0.0  1.0 1.0 0.0 1.0 0.0  1.0 1.0 0.0 1.0 0.0  \* row names  c001cr03c10\_19700102  c001cr03c16\_19700102  c001cr04c09\_19700102  flx\_river  \* column names  hk  r0  r1  w0  w1 |

Figure 9.1. An example localization matrix.

## 9.2 Using PESTPP-IES

### 9.2.1 General

The algorithm implemented by PESTPP-IES is described in detail by Chen and Oliver (2013). The reader is referred to that publication for a complete mathematical description of that algorithm; see also the description presented by White (2018). We now focus on PESTPP-IES implementation details, and on control variables that are available for tuning the Chen and Oliver algorithm implemented by PESTPP-IES to particular history-matching contexts.

As is the usual protocol for members of the PEST++ suite, values of control variables are supplied as arguments to keywords which must be placed on lines within a PEST control file that begin with the “++” character string.

Like most programs of the PEST++ suite, PESTPP-IES can be run without any PEST++ control variables. PESTPP-IES then provides default values for all of these variables. For example, it assumes that the prior parameter covariance matrix is diagonal, and that parameter bounds span four standard deviations of the prior probability distribution of each parameter. Nevertheless, users are encouraged to use PESTPP-IES control variables in order to ensure that its performance is optimized for their particular modeling context.

### 9.2.2 Initial Realizations

#### Realizations generated by PESTPP-IES

Initial parameter realizations can be generated by PESTPP-IES (the default), or they can be supplied by the user. The number of realizations that PESTPP-IES generates is set by the *ies\_num\_reals()* control variable; the default value is 50.

If you do not provide specifications for prior parameter uncertainty, then PESTPP-IES will make the following assumptions when generating initial parameter realizations.

1. The prior mean of a parameter is the value supplied for that parameter in the “parameter data” section of the PEST control file.
2. Every parameter is statistically independent of every other parameter.
3. The difference between the upper and lower bound of each parameter is equal to four standard deviations of its prior probability distribution. This defines the 95% confidence interval of the parameter if it is normally distributed. PESTPP-IES adopts this protocol even if the initial value of a parameter is not centrally located with respect to these bounds. Where a parameter is log transformed, the difference between the logarithms of the parameter’s upper and lower bounds is equated to four standard deviations of the prior probability distribution of the log of the parameter.

In generating prior parameter realizations, and in adjusting these realizations through the ensemble smoother process, PESTPP-IES ensures that a parameter’s bounds are not violated if the *ies\_enforce\_bounds()* control variable is set to *true* (it is *true* by default). The parameter’s probability density function is therefore effectively truncated by these bounds. In some situations, truncation may affect the performance of PESTPP-IES. Incidences of this occurrence may be reduced by ensuring that bounds are symmetrical with respect to initial parameter values cited in the “parameter data” section of the PEST control file (taking log transformation into account). Alternatively, the *par\_sigma\_range()* control variable may be used to reduce prior parameter standard deviations so that bounds are less likely to be encountered.

PESTPP-IES allows you to specify the properties of an assumed multiGaussian prior parameter probability distribution yourself; this distribution can include correlation between parameters. This is done using the *parcov()* control variable – the same variable that is available to users of PESTPP and PESTPP-OPT. As for these other programs, the filename that is supplied as the argument for this control variable can be a parameter uncertainty file (recognized by an extension of *.unc*), a covariance matrix file (recognized by an extension of *.cov*), or a binary file holding a covariance matrix that adopts the Jacobian matrix protocol for matrix storage (recognized by an extension of *.jco* or *.jcb*); see appendix B of this manual for specifications of the first two of these file types. In generating random parameter realizations comprising the initial ensemble, parameter values listed in the “parameter data” section of the PEST control file are taken as prior mean parameter values. If the *ies\_enforce\_bounds()* control variable is set to *true*, then parameter realizations are truncated at their bounds; nor are these bounds transgressed during subsequent parameter adjustment.

Unless PESTPP-IES is advised to the contrary, realizations of measurement noise to add to the values of observations provided in the “observation data” section of the PEST control file are generated in the manner already stated. That is, the noise associated with any observation is assumed to be statistically independent of the noise associated with all other observations. The standard deviation of noise associated with each observation is assumed to be the inverse of the weight associated with that observation. However if an observation is given a weight of zero, it plays no part in the history-matching process; nevertheless, these observations are “carried” through the PESTPP-IES reporting process.

“Greater than” and “less than” observations are treated differently from ordinary observations. PESTPP-IES adds no noise to values for these observations provided in the “observation data” section of the PEST control file when formulating observation realizations. This reflects the fact that, on most occasions of their use, observations of this type reflect something that is known about a system rather than something that is measured. At the same time, the penalty for transgressing an observation of this type that is encapsulated in its associated weight should reflect a modeller’s capacity to tolerate this transgression in pursuit of fitting other data. If you are unhappy with this approach, you are free to generate your own observation realizations in which noise is added to these one-way observations; see below.

If the ­*ies\_add\_base()* control variable is suppled as *true* (which is the default), then PESTPP-IES adds an extra realization to each of the parameter and observation ensembles. For parameters, this “realization” is comprised of parameter values listed in the “parameter data” section of the PEST control file. For observations, this “realization” is comprised of measurements read from the “observation data” section of the PEST control file.

#### User-Supplied Realizations

Realizations of parameters and/or observations can be prepared by the user and provided to PESTPP-IES in comma-delimited files (i.e. CSV files), and in binary files which adopt the PEST Jacobian matrix file protocol. PESTPP-IES recognizes the former through an extension of *.csv* and the latter through an extension of *.jco* or *.jcb*. Actually, as CSV files containing parameter and observation ensembles comprise part of PESTPP-IES’s output dataset, initial realizations can also be prepared by PESTPP-IES itself.

The value of the optional *ies\_parameter\_ensemble()* control variable is the name of a CSV file containing a suite of parameter realizations which collectively comprise an ensemble. This file should contain *m*+1 columns, where *m* is the number of parameters featured in the “parameter data” section of the PEST control file. The first column must be comprised of text strings which endow each realization of the ensemble with a unique identifier. Each of the subsequent *m* columns must pertain to a single parameter; parameter names must occupy the first row of the CSV file as column headers. It is not necessary that these names be supplied in the same order as in the “parameter data” section of the PEST control file.

PESTPP-IES reads as many rows of the CSV as this file contains, unless a value has been provided for the *ies\_num\_reals()* control variable in the PEST control file. If this is the case, PESTPP-IES reads no more than *ies\_num\_reals()* rows (after the header row) of the CSV file. If *ies\_num\_reals()* exceeds the number of data rows that are present in the CSV file, then the number of realizations comprising the PESTPP-IES parameter ensemble is reduced in accordance with the contents of this file.

Parameters that are declared as tied and fixed in a PEST control file can be included in the user-supplied parameter ensemble CSV file. As has already been discussed, a user can decide for him/herself whether the values of fixed parameters are invariant from ensemble to ensemble, or whether they are the same as the values provided for these parameters in the PEST control file. However it is the user’s responsibility to ensure that tied parameters maintain the correct ratios to their parent parameters in all realizations. If the user fails in his/her responsibility, PESTPP-IES will detect this; values attributed to tied parameters will then be altered so that correct tied-to-parent parameter ratios are maintained. Furthermore, if omitted from user-supplied parameter realizations, values for fixed and tied parameters will be provided by PESTPP-IES in accordance with values supplied in the “parameter data” section of the PEST control file.

Optionally, one of the realizations provided in a user-supplied parameter ensemble can be named “base”. The significance of the base realization has already been explained. If a realization named “base” is provided in a CSV file, and if the *ies\_add\_base()* control variable has been set to *true*, then PESTPP-IES assumes that the base realization has been provided by the user; it will not create one itself. However if a base realization is not provided in a CSV file, and *ies\_add\_base()* has been set to *true*, then PESTPP-IES creates the base realization; as has already been discussed, parameter values comprising a PESTPP-IES-generated base realization are read from the “parameter data” section of the PEST control file.

If the *ies\_enforce\_bounds()* control variable has been set to *true*, and if the values of any parameters supplied in a CSV file violate bounds provided in the PEST control file, PESTPP-IES alters parameter values to respect them.

The optional *ies\_observation\_ensemble()* keyword provides the name of a CSV file containing realizations which comprise an observation ensemble. Similar protocols apply to this file as those that apply to a parameter ensemble CSV file. If both a parameter and an observation ensemble CSV file are supplied, then PESTPP-IES links realizations in these files according to the order in which they are supplied, regardless of realization names. If a base realization is supplied in each of these files, it is the user’s responsibility to ensure that these occupy the same rows of their respective files. If this is not the case, PESTPP-IES will detect this inconsistency; it will then cease execution with an appropriate error message.

### 9.2.3 “Regularization”

Chen and Oliver (2013) provide two equations through which parameters comprising a particular realization within an ensemble are adjusted to provide a better fit with the calibration dataset. These are equations 18 and 19 in their paper. Equation 18 is expensive to compute. It incorporates a term that penalizes adjusted parameter fields whose statistical properties depart too far from those which characterize the prior parameter distribution. Equation 19 is simpler, and less expensive to compute. It adjusts parameters on the basis of current model-to-measurement misfit only. A user can inform PESTPP-IES to use the simpler and less expensive equation by providing the value *true* to the *ies\_use\_approx()* control variable. Despite the fact that the default value for this variable is *false*, *true* works well on most occasions.

A phenomenon that is sometimes encountered in using an ensemble smoother is a collapse in diversity of parameter realizations as the iterative adjustment process progresses. Sometimes this collapse can invalidate the integrity of posterior parameter and predictive probability distributions that the ensemble attempts to characterize.

This phenomenon can be prevented if “regularization” is introduced to the process through which each realization comprising an ensemble is adjusted to obtain a better fit with field data. This regularization is of the “preferred value” type. It instructs PESTPP-IES to maintain proximity of adjusted parameter values to the original values of respective realizations, that is to parameter values that were generated using the prior parameter probability distribution. As for all constraints of this type, its use raises the problem of how to balance weights assigned to residuals that are used to quantify model-to-measurement misfit on the one hand, against those that are used to quantify parameter departures from initial values on the other hand.

PEST and PESTPP calculate regularization weights through a numerical procedure that evaluates these weights on the basis of a user-supplied “measurement objective function”. This defines the level of model-to-measurement fit that a modeller is not prepared to exceed. Unfortunately, numerical calculation of a regularization weight factor through this means is an expensive undertaking. PESTPP-IES uses a simpler option, but with less intuitive appeal.

Optionally, a user can supply a value for the *ies\_reg\_factor()* control variable. If this variable is given a value greater than zero, then preferred value regularization is implemented. That is, an objective function penalty is incurred to the extent that the values of individual parameters that comprise a particular parameter realization depart from their initial values that were generated on the basis of the prior parameter probability distribution. A “regularization” objective function is computed which quantifies this departure. The weight assigned to each individual parameter departure is proportional to the inverse of the prior standard deviation of the respective parameter. (At the time of writing, a covariance matrix cannot be employed in formulation of the regularization objective function, so that each parameter departure is assumed to be statistically independent from every other parameter departure.) During every iteration of the parameter adjustment process, PESTPP-IES calculates a weight factor that is then uniformly applied to all of these parameter departures. The value assigned to this weight factor is such that the “regularization objective function” is equal to a fraction *F* of the current value of the objective function which characterizes model-to-measurement misfit. *F* is the value assigned to the *ies\_reg\_factor()* control variable.

The PESTPP-IES default value for *ies\_reg\_factor()* is zero; that is, no “regularization” is applied. Under these circumstances, PESTPP-IES seeks to minimize model-to-measurement misfit for all parameter realizations. Where regularization is applied, determination of a suitable value for *ies\_reg\_factor()* can be difficult. Its value depends on the nature of the inverse problem, on the number of parameters comprising a realization, and on the desired level of model-to-measurement misfit. Despite its dependence on all of these variables, a value in the vicinity of 0.25 often works well. Just as for specification of the PHIMILIM control variable in PEST and PESTPP, a little trial and error may be warranted in choosing a suitable value for *ies\_reg\_factor()*. Assignment of too high a value slows the progress of PESTPP-IES in reducing the objective function. Assignment of too low a value may provide insufficient insurance against parameter field collapse.

In some contexts, a more heuristic approach may be taken to balancing goodness of fit against ensemble diversity. A user may ascribe a low value to *ies\_reg\_factor()* so that, after a number of iterations have elapsed, a very good fit is attained between model outputs and the calibration dataset. He/she can then select an ensemble from a previous iteration that, in his/her opinion, best balances fit with the calibration dataset against parameter field diversity.

In addition to its role in generating initial parameter realizations, the prior parameter probability distribution supports the following aspects of PESTPP-IES calculations.

1. It features in equation 18 of Chen and Oliver (2013). This term of the equation is omitted if equation 19 is used for parameter field adjustment in its stead; as stated above, this occurs if *ies\_use\_approx()* is set to *true*.
2. It provides weights to individual “regularization observations” that measure departures of adjusted parameter fields from initial parameter fields. This occurs if *ies\_reg\_factor()* is set to a value greater than zero.
3. It is used in prior parameter scaling; see below.

Where a user-supplied CSV file provides initial parameter realizations, the prior covariance matrix that is used for the above purposes can be calculated empirically from these realizations. Alternatively, a prior parameter covariance matrix can be read from a file whose name is supplied with the *parcov()* control variable. Regardless of whether the *parcov()* control variable has been supplied, PESTPP-IES calculates prior parameter variances empirically for the second and third of the above tasks from user-supplied initial parameter realizations (if the latter are available) if a value of *true* is supplied for the *ies\_use\_empirical\_prior()* control variable. If this is done, off-diagonal elements of the prior parameter covariance matrix are assigned values of zero for use in the above calculations, this expediting their implementation.

Note that the default value for *ies\_use\_empirical\_prior()* is *false*. Hence PESTPP-IES employs a user-supplied covariance matrix for the above roles if one is available. Note also that, even if *ies\_use\_empirical\_prior()* is set to *true*, a user-supplied covariance matrix is always used in the first of the above roles if it can be accessed through a *parcov()* file. If this file is not available so that an empirical covariance matrix is used for this task, off-diagonal elements of this matrix are not automatically set to zero.

### 9.2.4 Prior Parameter Scaling

Like PESTPP and PEST, PESTPP-IES uses a Jacobian matrix as a basis for parameter adjustment. The mathematics of parameter adjustment which all of these programs implement is very similar, differing only in some of the details of how to handle problem ill-posedness. The major difference between PESTPP-IES on the one hand and PESTPP/PEST on the other hand is in how the Jacobian matrix is calculated. PESTPP-IES does not use finite parameter differences. Instead, it runs the model using the suite of random parameter fields that comprise an ensemble. It then inspects model outputs that correspond to members of the calibration dataset and calculates cross-covariances between these and individual parameters. From these covariances, with some matrix manipulation, it calculates an approximation to the Jacobian matrix. Where the number of realizations that comprise an ensemble is less than the number of adjustable parameters featured in the PEST control file, this Jacobian matrix is column-rank-deficient. Nevertheless, provided its rank is higher than the dimensionality of the calibration solution space, it can support attainment of parameter values which provide a good fit between model outputs and members of the calibration dataset.

In calculating model-output-to-parameter cross-covariances, certain numerical advantages can be gained if differences between individual parameter realizations and the mean parameter field are scaled. This is done by dividing the difference between each parameter and its mean by the prior standard deviation of that parameter. PESTPP-IES performs this scaling if the *ies\_use\_prior\_scaling()* control variable is set to *true*. Experience has demonstrated that prior scaling can be beneficial for problems that involve a very high number of parameters (over three hundred thousand), but that it is not so effective for problems that involve fewer parameters. The default value for *ies\_use\_prior\_scaling()* is *false*.

### 9.2.5 The Marquardt Lambda

The Marquardt lambda plays a pivotal role in gradient based inversion. Use of a high lambda value in early iterations of an inversion process, and a low lambda value later in that process, can have a large impact on the rate at which a good fit with the calibration dataset is attained. However the best value of the Marquardt lambda to use during any particular iteration must often be determined by trial and error. PESTPP and PEST test a number of parameter upgrades, calculated using different values of the Marquardt lambda; the cost is one model run for each tested lambda. Optionally, upgrades calculated using fractional lengths along these parameter upgrade vectors can also be tested. The parameter set that leads to the lowest objective function is selected as the upgraded parameter set.

Where a single parameter field is being estimated in a standard model calibration process, this procedure constitutes a good investment in model runs. (In parallel computing environments the cost may actually be very small, as computing cores that would otherwise be idle while waiting to fill the next Jacobian matrix are given something to do.) This investment in finding the best parameter upgrade vector provides the best return on the previously much larger investment of filling a Jacobian matrix.

The situation is different, however, when many parameter fields comprising an ensemble are adjusted, such as is done by PESTPP-IES. For an ensemble smoother, the testing of parameter upgrades and the filling of a Jacobian matrix are the same operation. The cost of this operation would become prohibitive if a trial and error lambda search procedure accompanied the adjustment of each parameter realization comprising an ensemble. Hence a different strategy must be adopted. This strategy is to undertake lambda testing for only a limited number of parameter fields, and to then use the best lambda that emerges from that testing process when upgrading the rest of them. This limited number of realizations used to evaluate objective function values during upgrade testing is referred to as the “subset”. Options for how to pick the subset are available through the *ies\_subset\_how* argument; valid choices for *ies\_subset\_how* are “first” (the first *ies\_subset\_size* realizations),”last” (the last *ies\_subset\_size* realizations) ,”random” (randomly select *ies\_subset\_size* realizations each iteration), or “phi\_based” (select *ies\_subset\_size* realizations across the previous composite phi distribution). Note that if the “base” parameter realization is present, it is always included in the subset as the objective function calculated for this realization has special significance.

The number of realizations that comprise the ensemble subset used for lambda testing is set by the value of the *ies\_subset\_size()* control variable. During each iteration of the ensemble smoother process, values of the Marquardt lambda used for testing realization upgrades are determined by applying a set of multipliers to the best lambda found during the previous iteration. These multipliers are provided through the *ies\_lambda\_mults()* control variable. A comma separated list of multipliers should be supplied by the user as arguments to this keyword; at least one of these multipliers should be less than 1.0 while, or course, one of them should be greater than 1.0. Line search factors (otherwise known as scale factors) that are applied to each of these lambdas can also be supplied. If so, this is done through the *lambda\_scale\_fac()* control variable, the same variable that is used by PESTPP. As for *ies\_lambda\_mults()*, scale factors should be supplied as a comma-separated list of numbers spanning a range from below 1.0 to greater than 1.0. The total number of model runs required to test parameter upgrades during a given iteration is thus *ies\_subset\_size()* times the number of multipliers supplied with the *ies\_lambda\_mults()* control variable times the number of factors supplied with the *lambda\_scale\_fac()* control variable.

The value of the Marquardt lambda to use during the first iteration of the Kalman smoother process can be supplied through the *ies\_initial\_lambda()* control variable. Lambda multipliers supplied through *ies\_lambda\_mults()* are applied to this value during the first iteration of this process. The PESTPP-IES default value for *ies\_initial\_lambda()* is where *μ*Փ is the mean of objective functions achieved using realizations comprising the initial ensemble, and *n* is the number of non-zero-weighted observations featured in the “observation data” section of the PEST control file.

Suppose for example that the following lines appear in a PESTPP-IES control file.

|  |
| --- |
| ++ ies\_initial\_lambda(100)  ++ ies\_subset\_size(4)  ++ ies\_lambda\_mults(0.1,1.0,10.0)  ++ ies\_lambda\_scale\_fac(0.9,1.0,1.1) |

Figure 9.2 Part of a PESTPP-IES control file.

From figure 9.1, the initial value of the Marquardt lambda is 100.0. During each iteration of the ensemble smoother process, PESTPP-IES employs three values of the Marquardt lambda, these being equal to 0.1, 1.0 and 10 times the value of the best Marquardt lambda from the previous iteration (or the initial Marquardt lambda in the first iteration). PESTPP-IES selects the first 4 realizations from the parameter ensemble and calculates updated parameter fields using these 3 Marquardt lambdas. It also calculates parameter upgrades corresponding to lengths along these lambda upgrade directions of 0.9 and 1.1 times that which is calculated using the Marquardt lambda alone (this corresponding to a line search factor of 1.0). Hence PESTPP‑IES commits a total 36 model runs to establishing the best value of lambda and the best line search factor.

After carrying out these model runs, PESTPP-IES chooses the lambda and scale factor combination that yields the lowest mean objective function for the subset of the ensemble that is dedicated to testing these variables. It then compares these with the prevailing mean objective function calculated for the same subset. If the ratio between the new and prevailing mean is greater than the value of the *ies\_accept\_phi\_fac()* control variable (whose default value is 1.05), PESTPP-IES does not accept the lambda and scale factor that it has just determined. Instead, it multiplies lambda by a factor that is equal to the *lambda\_inc\_fac()* control variable (the default value for which is 10.0), and repeats the lambda and scale factor testing procedure described above on the same ensemble subset. If, after NPHINORED attempts, PESTPP-IES is not able to find a lambda and line search factor for which the objective function is less than *ies\_accept\_phi()* times the prevailing objective function for the ensemble subset, it declares the iterative smoothing process to be over and ceases execution. NPHINORED is a termination criterion supplied on the eight line of the “control data” section of the PEST control file.

Alternatively, if the mean objective function attained through the lambda and line search factor process described above is less than *ies\_accept\_phi\_fac()* times the prevailing mean objective function for the ensemble subset, PESTPP-IES applies the best lambda and line search factor to the remainder of the ensemble. If the mean objective function for the entire ensemble was reduced from its prevailing mean then, on the next iteration of the smoother process, PESTPP-IES lowers the Marquardt lambda by applying a factor of *lambda\_dec\_fac()* to its current value. The default value of *lambda\_dec\_fac()* is 0.75.

### 9.2.6 Restarting

As has already been discussed, if either you or PESTPP-IES have generated a set of random parameter fields (or PESTPP-IES has improved them from a set of previous parameter fields), these can be provided to a newly-restarted PESTPP-IES using the *ies\_parameter\_ensembles()* control variable. If a model run has been undertaken for each of these fields (either by you or by PESTPP-IES), then the iterative ensemble smoother can be initiated from these parameter fields in conjunction with the model outputs which correspond to them. PESTPP-IES is instructed to do this using the *ies\_restart\_obs\_en()* control variable. The value for this variable is the name of a CSV file containing model outputs corresponding to the set of parameter fields which PESTPP-IES already has in its possession.

As is the PESTPP-IES convention, the number of columns in the user-supplied, run-results CSV file must be equal to the number of non-zero weighted observations featured in the PEST control file plus one. (Actually PESTPP-IES will forgive you if model outputs corresponding to zero-weighted observations are also provided in this file.) The first row of the CSV file must name these model outputs; names must be in accordance with those listed in the “observation data” section of the PEST control file. The first column must contain unique observation set identifiers. There must be as many of these as there are members of the model output ensemble. Members of the model output ensemble are linked to members of the parameter ensemble by row number. However if an ensemble of measured values is provided through the *ies\_observation\_ensemble()* keyword, the names of observation ensemble members in the filename supplied through this keyword must agree with those provided in the model output ensemble filename supplied with the *ies\_restart\_obs\_en()* keyword.

In the event of model run failure for certain realizations, the parameter and simulated observation ensemble CSV files produced by PESTPP-IES have fewer rows than when the PESTPP-IES process commenced. This is because they now possess fewer realizations than the “measured observation” ensemble, this being comprised of measurements plus realizations of measurement noise. Lack of coherence between these ensembles can make a PESTPP-IES restart difficult. On restart, PESTPP-IES does not require that the names of user-supplied parameter and observation ensembles provided with the *ies\_parameter\_ensemble()* and *ies\_restart\_obs\_en()* keywords be the same (these are recorded in the first column of respective CSV files). Nor do these names need to be the same as realization names associated with the measurement observation ensemble supplied with the *ies\_observation\_ensemble()* keyword. Ensembles are simply read sequentially and matched to each other according to their order of appearance in respective files.

PESTPP-IES makes an exception to this protocol, however, if realization names are the same in filenames supplied with the *ies\_parameter\_ensemble()* and *ies\_restart\_obs\_en()* keywords. (This happens automatically if these files were written by PESTPP\_IES.) In this case PESTPP-IES links these names to realization names appearing in the *ies\_observation\_ensemble()* CSV file, ignoring “lost” realizations from this file in the process. Alternatively, a user can remove lost realizations from the *ies\_observation\_ensemble()* file him/herself. This is easily accomplished using the Python Pandas library.

### 9.2.7 Failed Model Runs

Where model runs are based on random parameter realizations, the risk of occasional model run failure is high for some models. The PANTHER run manager used by programs of the PEST++ suite is able to accommodate model run failure in ways described in section 5.3 of this manual. When model run failure is encountered, PESTPP-IES drops the parameter set that precipitated this failure from the ensemble. The ensemble thus loses a member.

PESTPP-IES provides a mechanism for detection of model run failure that extends those provided by its run manager. If the objective function associated with a particular model run is calculated to be greater than a certain threshold, PESTPP-IES deems the model run to have failed. This threshold is supplied as the value of *ies\_bad\_phi()* control variable.

To forestall excessive PESTPP-IES run times incurred by occasional model failure, it is a good idea to set the *max\_run\_fail()* model run control variable to 1 (the default value for PESTPP‑IES), and to choose values for the *overdue\_giveup\_fac()* and/or *overdue\_giveup\_minutes()* control variables judiciously; see section 5.3. Note also that model run failure does not hurt PESTPP-IES as much as it hurts PESTPP or PEST. This is because the value of any model run undertaken by PESTPP-IES is lower than that undertaken by PEST or PESTPP. For the latter programs a failed model run during finite-difference derivatives calculation may lead to an empty column of the Jacobian matrix. In contrast, because PESTPP-IES uses an entire ensemble to fill a Jacobian matrix, a single failed model run does not result in an empty Jacobian matrix column. The outcome of model run failure is that the number of model runs employed in the averaging process through which this column is calculated is reduced by one.

### 9.2.8 Reporting

PESTPP-IES records its progress to the screen and to its run record file. In addition to this, it records a plethora of output files. These are discussed in the next section. These output files can be supplemented by additional files that record, in ASCII format, matrices that PESTPP-IES formulates in the course of upgrading parameter realizations. The extent of its output file prediction can be controlled using the *ies\_verbose\_level()* variable. This can be awarded values of 0,1 or 2. The default is 1.

If a model is numerically unstable, a user may wish to be informed of parameter values that precipitate run failure. As is discussed below, PESTPP-IES records the values of all parameters in all realizations comprising an ensemble, together with model run results, in iteration-specific CSV files. Parameter sets used in lambda testing can also be recorded if the *save\_lambda\_ensembles()* control variable is set to “true”.

### 9.2.9 Termination Criteria

Like PEST and PESTPP, PESTPP-IES reads termination criteria from the eighth line of the “control data” section of the PEST control file.

PESTPP-IES ceases execution after NOPTMAX iterations have elapsed. However during these NOPTMAX iterations it applies the PHIREDSTP, NPHISTP and NPHINORED termination criteria to the mean objective function calculated using all realizations of the ensemble over successive smoother iterations. If the relative reduction in the mean objective function is less than PHIREDSTP over NPHISTP iterations, or if NPHINORED iterations have elapsed since the last reduction in the mean objective function has occurred, PESTPP-IES ceases execution.

Note that, as described above, PESTPP-IES also ceases execution if, during any particular iteration of the smoother process, it cannot find a lambda and scale factor that allows it to calculate parameter upgrades for which the objective function is less than *ies\_accept\_phi\_fac()* times the current objective function. NPHINORED attempts to find these precipitate cessation of execution.

Special values of NOPTMAX can instigate special PESTPP-IES behaviour. If NOPTMAX is set to -1, PESTPP-IES does not upgrade random parameter sets which comprise an ensemble. It simply runs the model once for each parameter set, records model output values, and then ceases execution, thereby effective undertaking unconstrained Monte Carlo analysis. If NOPTMAX is set to zero, execution of PESTPP-IES is even shorter. It generates the mean parameter set from the random parameter sets that comprise the initial ensemble. It undertakes one model run using this mean parameter set. It records the outcomes of this run, and then ceases execution.

## 9.3 PESTPP-IES Output Files

### 9.3.1 CSV Output Files

PESTPP-IES writes a suite of output files. Many of these are comma-delimited files (i.e. CSV files). Some of these files list values of parameters and corresponding model outputs. Others list objective functions and related quantities. These files are ready for immediate inspection and processing in a spreadsheet package such as Microsoft EXCEL. CSV files written by PESTPP-IES are discussed in the current sub-section. Other files that are written by PESTPP-IES are discussed in the following sub-section.

As always, it is assumed that the filename base of the PEST control file on which history-matching is based is named *case.pst*. *N* is used to signify iteration number in the following filenames. An iteration number of 0 corresponds to initial parameter fields and corresponding model outputs.

|  |  |
| --- | --- |
| **File** | **Contents** |
| *case.N.par.csv* | Values of all parameters in all realizations comprising an ensemble. For N=0, this is the initial ensemble, sampled from the prior parameter probability distribution. |
| *case.N.obs.csv* | Values of model outputs which correspond to observations listed in the “observation data” section of the PEST control file. These are calculated using all realizations comprising the parameter ensemble. |
| *case.base.obs.csv* | Base observation values. These are calculated by generating realizations of measurement noise and adding this noise to measured values listed in the “observation data” section of the PEST control file. |
| *case.N.L.lambda.F.scale.csv* | These files are produced if the *save\_lambda\_ensembles()* control variable is set to *true*. They record parameter values used in testing the effects of different Marquardt lambdas and line search factors. *L* is the value of the Marquardt lambda; *F* is the value of the line search factor. |
| *case.phi.actual.csv* | Objective functions calculated during all iterations of the ensemble smoother process for all members of the ensemble. Objective functions are computed from differences between model outputs and measurements recorded in the “observation data” section of the PEST control file using weights that are also provided in this section. |
| *case.phi.group.csv* | Objective function components pertaining to different observation groups calculated for all iterations of the ensemble smoother process. Progression of the regularization objective function is also recorded; this measures discrepancies between current and initial parameter values. Contributions by different parameter groups to the regularization objective function are also listed. |
| *case.phi.meas.csv* | Objective functions calculated during all iterations of the ensemble smoother process for all members of the ensemble. Objective functions recorded in this file are computed from differences between model outputs and observation ensembles, i.e. the measurements recorded in the “observation data” section of the PEST control file supplemented with realizations of measurement noise. |
| *case.phi.regul.csv* | Regularization objective functions calculated during each iteration of the ensemble smoother process for all members of the ensemble. For a particular realization this is calculated using differences between current and initial parameter values. The weight applied to a particular difference is the inverse of the prior standard deviation of the parameter. |
| *case.phi.composite.csv* | The composite objective function is the measurement objective function plus the regularization objective function multiplied by the value of the *regul\_frac()* control variable. |

Table 9.2 CSV files written by PESTPP-IES. It is assumed that the name of the PEST control file is *case.pst*.

### 9.3.2 Non-CSV Output Files

Non-CSV output files written by PESTPP-IES are listed in the following table.

|  |  |
| --- | --- |
| **File** | **Contents** |
| *case.rec* | The run record file. This echoes information in the PEST control file, and then records a history of the ensemble smoother process. |
| *case.rmr* | Parallel run management record. This file is written if model runs are conducted in parallel. |
| *case.pfm* | Performance log. This file records the times at which various processing steps begin and end. |
| *case.rns* | Binary file used for model run management. |

Table 9.3 Non-CSV files written by PESTPP-IES. It is assumed that the name of the PEST control file is *case.pst*.

## 9.4 Summary of Control Variables

Table 9.4 lists PESTPP-IES control variables. All of these are optional. If a variable is not supplied, then a default is assumed for its value. Where appropriate, the value of the default is presented along with the name of the variable in the table below. Variables discussed in section 5.3.6 of this manual that control parallel run management are not listed in the following table.

Note also that the number of control variables may change with time. Refer to the PEST++ web site for variables used by the latest version of PESTPP-IES.

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Role** |
| *ies\_num\_reals(50)* | integer | The number of realizations to draw in order to form parameter and observation ensembles. |
| *parcov()* | text | The name of a file containing the prior parameter covariance matrix. This can be a parameter uncertainty file (extension *.unc*), a covariance matrix file (extension *.cov*) or a binary file which observes the same protocol as a Jacobian matrix file (extension *.jco* or *.jcb*). |
| *par\_sigma\_range(4.0)* | real | The difference between a parameter’s upper and lower bounds expressed as standard deviations. |
| *ies\_parameter\_ensemble()* | text | The name of a CSV file containing user-supplied parameter realizations comprising the initial parameter ensemble. If this keyword is omitted, PESTPP-IES generates the initial parameter ensemble itself. |
| *ies\_obsevation\_ensemble()* | text | The name of a CSV file containing user-supplied observation realizations comprising the observation ensemble. If this keyword is omitted, PESTPP-IES generates the observation ensemble itself. |
| *ies\_add\_base(true)* | Boolean | If set to true, instructs PESTPP-IES to include a “realization” in the initial parameter ensemble comprised of parameter values read from the “parameter data” section of the PEST control file. The corresponding observation ensemble is comprised of measurements read from the “observation data” section of the PEST control file. |
| *ies\_restart\_obs\_en()* | text | The name of a CSV file containing model outputs calculated using a parameter ensemble. If it reads this file, PESTPP-IES does not calculate these itself. |
| *ies\_enforce\_bounds(true)* | Boolean | If set to *true* PESTPP-IES will not transgress bounds supplied in the PEST control file when generating or accepting parameter realizations, and when adjusting these realizations. |
| *ies\_initial\_lambda()* | real | The initial Marquardt lambda. The default value is |
| *ies\_lambda\_mults(0.1,0.5,1.0,2.0,5.0)* | comma-separated reals | Factors by which to multiply the best lambda from the previous iteration to yield values for testing parameter upgrades during the current iteration. |
| *lambda\_scale\_fac(0.75,0.95,1.0,1.1)* | comma-separated reals | Line search factors along parameter upgrade directions computed using different Marquardt lambdas. |
| *ies\_subset\_size(5)* | integer | Number of realizations used in testing and evaluation of different Marquardt lambdas. |
| *ies\_use\_approx(false)* | Boolean | Use complex or simple formula provided by Chen and Oliver (2013) for calculation of parameter upgrades. The more complex formula includes a function which constrains parameter realizations to respect prior means and probabilities. |
| *ies\_reg\_factor(0.0)* | real | Regularization objective function as a fraction of measurement objective function when constraining parameter realizations to respect initial values. |
| *ies\_bad\_phi(1.0E30)* | real | If the objective function calculated as an outcome of a model run is greater than this value, the model run is deemed to have failed. |
| *ies\_use\_prior\_scaling(false)* | Boolean | Use a scaling factor based on the prior parameter distribution when evaluating parameter-to-model-output covariance used in calculation of the randomized Jacobian matrix. |
| *ies\_use\_empirical\_prior(false)* | Boolean | Use an empirical, diagonal parameter covariance matrix for certain calculations. This matrix is contained in a file whose name is provided with the *ies\_parameter\_ensemble()* keyword. |
| *save\_lambda\_ensembles(true)* | Boolean | Save a set of CSV files that record parameter realizations used when testing different Marquardt lambdas. |
| *ies\_verbose\_level(1)* | 0, 1 or 2 | The level of diagnostic output provided by PESTPP-IES. If set to 2, all intermediate matrices are saved to ASCII files. This can require a considerable amount of storage. |
| *ies\_accept\_phi\_fac(1.05)* | real > 1.0 | The factor applied to the previous best mean objective function to determine if the current mean objective function is acceptable. |
| *ies\_lambda\_dec\_fac(0.75)* | real < 1.0 | The factor by which to decrease the value of the Marquardt lambda during the next IES iteration if the current iteration of the ensemble smoother process was successful in lowering the mean objective function. |
| *ies\_lambda\_inc\_fac(10.0)* | real > 1.0 | The factor by which to increase the current value of the Marquardt lambda for further lambda testing if the current lambda testing cycle was unsuccessful. |
| *ies\_subset\_how(phi\_based)* | “first”,”last”,  ”random”,  ”phi\_based | How to select the subset of realizations for objective function evaluation during upgrade testing. Default is “phi\_based” |
| *ies\_localize\_how(parameters)* | “parameters” or “observartions | A flag to process the localization matrix row-by-row (observations) or column-by-column (parameters). Default is “parameters” |
| *ies\_num\_threads(0)* | Integer > 1 | The number of threads to use during the localized upgrade solution process. If the localizer contains many (>10K) rows, then multithreading can be used to substantially speed up the upgrade calculation process. *ies\_num\_threads* should not be greater than the number of physical cores on the host machine |
| *ies\_localizer(local.mat)* | text | The name of a matrix to use for localization. The extension of the file is used to determine the type: .mat is an ASCII matrix file, .jcb/.jco is a jacobian binary format and .csv is a comma-separated value file. Note adjustable parameters not listed in the localization matrix columns are implicitly treated as “fixed” and non-zero weight observations not listed in the rows are implicitly treated as zero weighted. |

Table 9.4 PESTPP-IES control variables with default values. Parallel run management variables can be supplied in addition to these. See section 5.3.6.

# 10. PESTPP-SWP

## 10.1 Introduction

PESTPP-SWP runs a model using a suite of parameter fields. Parameter values that comprise these fields are supplied in a comma-delimited file (i.e. a CSV file). It records the values of model outputs calculated using these parameter fields in another CSV file, together with objective function components calculated from these outputs. The PEST control file which PESTPP-SWP reads informs it of observation values, observation weights, observation groups, and the setting of the PESTMODE control variable. If PESTMODE is set to “regularization”, then PESTPP-SWP calculates a regularization objective function in addition to the measurement objective function.

On most occasions of PESTPP-SWP usage, model runs are conducted in parallel. Use of PESTPP-SWP gives a modeller easy access to model run parallelization for the completion of model runs undertaken for any purpose whatsoever. A significant amount of functionality available through PyEMU makes use of PESTPP-SWP to undertake parallelized model runs; it bases its calculations on the outcomes of these runs.

## 10.2 Using PESTPP-SWP

As usual, variables which control how PESTPP-SWP operates must be placed in the PEST control file whose name is supplied on its command line; these variables should appear on lines that begin with the “++” character string.

PESTPP-SWP is directed to a comma-delimited input file through the value supplied for its *sweep\_parameter\_csv\_file()* control variable. This file must have as many columns as there are parameters featured in the PEST control file, plus one extra column on the left. The first column is reserved for the user-supplied realization name. However the filename that is supplied as the argument to this control variable need not actually be a CSV file. If the filename extension is *.jco* or *.jcb*, PESTPP-SWP expects a binary file in which parameter sets are stored using the same protocol as that which is used by PEST and programs of the PEST++ suite for storage of a Jacobian matrix. For a given number of parameter sets, this type of file is far smaller than a comma-delimited ASCII file. It can also be read more quickly.

The number of rows contained in the user-prepared CSV file depends on the number of parameter sets for which model runs are required. This file must contain one more row than the number of these parameter sets. PESTPP-SWP knows how many model runs to commission from the number of rows in the CSV file.

The first row of the user-provided CSV file is reserved for column headers. All but the first column header must be the name of a parameter featured in the PEST control file. Parameters do not need to be supplied in the same order as they appear in the PEST control file.

If the *sweep\_parameter\_csv\_file()* control variable does not appear in the PEST control file that is cited on the PESTPP-SWP command line, PESTPP-SWP assumes an input filename of *sweep\_in.csv*.

Note the following.

* If, in the PEST control file read by PESTPP-SWP, a particular parameter is tied to another parameter (i.e. a parent parameter), PESTPP-SWP insists that the child/parent ratio is observed in the user-supplied CSV file. If it is not observed, it alters the value of the child parameter accordingly.
* If a parameter is fixed in the PEST control file, and the value provided for that parameter in the CSV file differs from that in the PEST control file, the value in the PEST control file overrides that in the CSV file.

PESTPP-SWP can fill in values for fixed and tied parameters if these are missing from the user-supplied CSV file. Actually, it can provide values for other missing parameters as well if the *sweep\_forgive()* control variable is set to *true*. These missing values are taken from the PEST control file which is read by PESTPP-SWP.

PESTPP-SWP writes a model output file whose name is provided through the *sweep\_output\_csv\_file()* control variable. If this variable is not provided, PESTPP-SWP employs the name *sweep\_out.csv* for its output file.

The control variable *sweep\_chunk()* pertains to parallelization of model runs. Runs are done in bundles of size *N*, where *N* is the value supplied for this variable. (A chunk of 500 is the default). This number should be chosen wisely. It should be a multiple of the number of workers that PESTPP-SWP can use for carrying out model runs.

## 10.3 Summary of Control Variables

Table 10.1 tabulates PESTPP-SWP control variables. As usual, all of these variables are optional. If a variable is not supplied, then a default is assumed. Default values are presented along with the names of variables in the table below. Variables discussed in section 5.3.6 that control parallel run management are not listed in this table.

The number of control variables may change with time. Refer to the PEST++ web site for variables used by the latest version of PESTPP-SWP.

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Role** |
| *sweep\_parameter\_csv\_file(sweep\_in.csv)* | text | A user-provided CSV file that lists the parameter values employed for model runs. If this filename ends with .jcb or .jco, then a binary format ensemble is read. |
| *sweep\_forgive(false)* | Boolean | If set to *true* PESTPP-SWP provides values for missing variables in the user-supplied CSV file. |
| *sweep\_output\_csv\_file(sweep\_out.csv)* | text | The CSV file written by PESTPP-SWP. |
| *sweep\_chunk(500)* | text | The number of runs to batch queue for the run manager. Each chunk is read, run and written as a single batch. |

Table 10.1 PESTPP-SWP control variables. Parallel run management variables can be supplied in addition to these; see section 5.3.6.

# 11. References

Ahlfeld, D.P. and Mulligan, A.E., 2000. Optimal Management of Flow in Groundwater Systems. Vol 1. Academic Press.

Campolongo, F., Cariboni, J., Saltelli, A. and Schoutens, W., 2005, Enhancing the Morris Method, *in* Hanson, K.M., and Hemez, F.M, eds., Sensitivity analysis of model output - Proceedings of the 4th International Conference on Sensitivity Analysis of Model Output (SAMO 2004) Santa Fe, New Mexico, March 8–11, 2004: Los Alamos National Laboratory Research Library, p. 369–379.

Chan, N., 1994. Partial infeasibility method for chance-constrained aquifer management. *J. Water Resour. Plan. Manag.* 120 (1), 70-89.

Charnes, A., Cooper, W.W., 1959. Chance-constrained programming. *Manag. Sci*. 6 (1), 73-79.

Chen, Y. and Oliver, D. S., 2013. Levenberg–Marquardt forms of the iterative ensemble smoother for eﬃcient history matching and uncertainty quantiﬁcation. *Computational Geosciences*. 17(4), 689–703.

Doherty, J., 2015. Calibration and uncertainty analysis for complex environmental models. Published by Watermark Numerical Computing, Brisbane, Australia. 227pp. ISBN: 978-0-9943786-0-6. Downloadable from [www.pesthomepage.org](http://www.pesthomepage.org).

Doherty, J., 2018a. Manual for PEST: Model-Independent Parameter Estimation. Part 1: PEST, SENSAN and Global Optimisers. Watermark Numerical Computing, Brisbane, Australia. Downloadable from [www.pesthomepage.org](http://www.pesthomepage.org).

Doherty, J., 2018b. Manual for PEST: Model-Independent Parameter Estimation. Part 2: PEST Utility Support Software. Watermark Numerical Computing, Brisbane, Australia. Downloadable from [www.pesthomepage.org](http://www.pesthomepage.org).

Doherty, J. and Welter, D., 2010, A short exploration of structural noise, *Water Resour. Res*., 46, W05525, doi:10.1029/2009WR008377.

Doherty, J., Welter, D. amd White, J.T. 2018. Overview of the PEST and PEST++ Suites. *Caelum*. Downloadable from http://www.caelumsoftware.org

Engelbrecht, A.P., 2007. Computational intelligence: an introduction. John Wiley & Sons.

Fienen, M.N., Doherty, J.E., Hunt, R.J., and Reeves, H.W., 2010. Using Prediction Uncertainty Analysis to Design Hydrologic Monitoring Networks: Example Applications from the Great Lakes Water Availability Pilot Project. U.S. Geological Survey Scientific Investigations Report 2010–5159, 44 p. [http://pubs.usgs.gov/sir/2010/5159 ]

Forrest, J., Nuez, D., Lougee-Heimer, R., 2016. CLP: COIN-OR Linear Programming Solver. https://projects.coin-or.org/Clp. (Accessed 9 November 2016).

Hantush, M.M., Marino, M.A., 1989. Chance-constrained model for management of stream-aquifer system. *J. Water Resour. Plan. Manag.* 115 (3), 259-277.

Hill, M.C. and Tiedeman, C.R., 2007. Effective groundwater model calibration: With analysis of data, sensitivities, predictions, and uncertainty. Wiley and Sons, New York, New York, 455 p

Homma T, Saltelli A. 1996. Importance measures in global sensitivity analysis of model output. *Reliability Engineering and System Safety*. 52(1), 1-17.

Lougee-Heimer, R., 2003. The common optimization interface for operations research: promoting open-source software in the operations research community. IBM J. Res. Dev. 47 (1), 57-66.

Miller, B.L., Wagner, H.M., 1965. Chance constrained programming with joint constraints. *Operations Res.* 13 (6), 930-945.

Morris, M.D., 1991, Factorial sampling plans for preliminary computational experiments. *Technometrics*, 33 (2), 161–174.

Oliver, D.S., Reynolds, A.C. and Liu, N., 2008. Inverse theory for petroleum reservoir characterization and history matching*.* Cambridge University Press.

Remy, N., Boucher, A. and Wu, J., 2011. Applied Geostatistics with SGEMS. A Users Guide. Cambridge University Press.

Saltelli, A., Tarantola, S., Campolongo, F., and Ratto, M., 2004. Sensitivity Analysis in Practice—A Guide to Assessing Scientific Models. John Wiley & Sons Ltd., 219p.

Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D. Saisana, M., and Tarantola, S., 2008, Global Sensitivity Analysis—The Primer. John Wiley & Sons Ltd., 292p.

Sobol, I.M., 2001, Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates. *Math­ematics and Computers in Simulation*. 55, 271–280.

Storn, R. and Price, K., 1995. Differential Evolution - A Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces. *Tech. Report, International Computer Science Institute (Berkeley)*.

Storn, R. and Price, K., 1997. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces, *Journal of Global Optimization*. 11, 341-359.

Tung, Y.-K., 1986. Groundwater management by chance-constrained model. *J.Water Resour. Plan. Manag.* 112 (1), 1-19.

Wagner, B.J. and Gorelick, S.M., 1987. Optimal groundwater quality management under parameter uncertainty. *Water. Resour. Res.* 23(7), 1162-1174.

Welter, D.E., Doherty, J.E., Hunt, R.J., Muffels, C.T., Tonkin, M.J., and Schreüder, W.A., 2012. Approaches in highly parameterized inversion - PEST++, a Parameter ESTimation code optimized for large environmental models: U.S. Geological Survey Techniques and Methods, book 7, section C5, 47 p., available at <https://pubs.usgs.gov/tm/tm7c5>.

Welter, D.E., White, J.T., Hunt, R.J., and Doherty, J.E., 2015. [Approaches in highly parameterized inversion: PEST++ Version 3, a Parameter ESTimation and uncertainty analysis software suite optimized for large environmental models](https://pubs.er.usgs.gov/publication/tm7C12). U.S. Geological Survey Techniques and Methods, book 7, section C12, 54 p.

White, J.T., 2018. A model-independent iterative ensemble smoother for efficient history-matching and uncertainty quantification in very high dimensions. Accepted for publication in *Environmental Modelling and Software*.

White, J.T., Fienen, M.N., Barlow, P.M. and Welter, D.E., 2018. A tool for efficient, model-independent management optimization under uncertainty. *Environmental Modelling and Software*, 100, 213-221.

White, J.T., Fienen, M.N. and Doherty, J.E., 2016. A Python framework for environmental model uncertainty analysis. *Environmental Modelling and Software*, 85, 217-228.

White, J.T., Doherty, J.E. and Hughes, J.D., 2014. Quantifying the predictive consequences of model error with linear subspace analysis. *Water Resour. Res*. 50 (2): 1152-1173.DOI: 10.1002/2013WR014767

# Appendix A. PEST Control File Specifications

This appendix provides specifications for a PEST control file.

Variables are recognized by their position in the file. They must be placed on the correct line of this file, and separated from their neighbours by at least one space.

PEST, BEOPEST and many of the PEST-support utility programs which are documented in part II of the PEST manual tolerate the presence of the following items in a PEST control file:

* blank lines;
* comments;
* lines that begin with the character string “++”.

Lines that begin with “++” are used for the insertion of variables which control the operation of the PEST++ suite of programs.

Comments can be placed on their own line. Alternatively, they can be placed at the end of a line which provides PEST control data. In either case, a comment follows a “#” character. Note, however, that this character is not construed as denoting the presence of an ensuing comment under any of the following circumstances:

* it is not preceded by a space, tab or the beginning of a line;
* it is part of a string that is enclosed in quotes.

These exceptions preclude mis-construing the presence of the “#” character in a filename as signifying the start of a comment.

Some of the older utilities that are documented in part II of the PEST manual do not tolerate the presence of blank lines, “++” lines or comments. All of these items can be removed from a PEST control file using the PSTCLEAN utility supplied with the PEST suite.

Figure A1.1 Names and locations of variables in the PEST control file. Optional variables are enclosed in square brackets.

pcf

\* control data

RSTFLE PESTMODE

NPAR NOBS NPARGP NPRIOR NOBSGP [MAXCOMPDIM][DERZEROLIM]

NTPLFLE NINSFLE PRECIS DPOINT [NUMCOM JACFILE MESSFILE] [OBSREREF]

RLAMBDA1 RLAMFAC PHIRATSUF PHIREDLAM NUMLAM [JACUPDATE] [LAMFORGIVE] [DERFORGIVE]

RELPARMAX FACPARMAX FACORIG [IBOUNDSTICK UPVECBEND] [ABSPARMAX]

PHIREDSWH [NOPTSWITCH] [SPLITSWH] [DOAUI] [DOSENREUSE] [BOUNDSCALE]

NOPTMAX PHIREDSTP NPHISTP NPHINORED RELPARSTP NRELPAR [PHISTOPTHRESH] [LASTRUN] [PHIABANDON]

ICOV ICOR IEIG [IRES] [JCOSAVE] [VERBOSEREC] [JCOSAVEITN] [REISAVEITN] [PARSAVEITN] [PARSAVERUN]

\* sensitivity reuse

SENRELTHRESH SENMAXREUSE

SENALLCALCINT SENPREDWEIGHT SENPIEXCLUDE

\* singular value decomposition

SVDMODE

MAXSING EIGTHRESH

EIGWRITE

\* lsqr

LSQRMODE

LSQR\_ATOL LSQR\_BTOL LSQR\_CONLIM LSQR\_ITNLIM

LSQRWRITE

\* automatic user intervention

MAXAUI AUISTARTOPT NOAUIPHIRAT AUIRESTITN

AUISENSRAT AUIHOLDMAXCHG AUINUMFREE

AUIPHIRATSUF AUIPHIRATACCEPT NAUINOACCEPT

\* svd assist

BASEPESTFILE

BASEJACFILE

SVDA\_MULBPA SVDA\_SCALADJ SVDA\_EXTSUPER SVDA\_SUPDERCALC SVDA\_PAR\_EXCL

\* parameter groups

PARGPNME INCTYP DERINC DERINCLB FORCEN DERINCMUL DERMTHD [SPLITTHRESH SPLITRELDIFF SPLITACTION]

(*one such line for each of NPARGP parameter groups*)

\* parameter data

PARNME PARTRANS PARCHGLIM PARVAL1 PARLBND PARUBND PARGP SCALE OFFSET DERCOM

(*one such line for each of NPAR parameters*)

PARNME PARTIED

(*one such line for each tied parameter*)

\* observation groups

OBGNME [GTARG] [COVFLE]

(*one such line for each of NOBSGP observation group*)

\* observation data

OBSNME OBSVAL WEIGHT OBGNME

(*one such line for each of NOBS observations*)

\* derivatives command line

DERCOMLINE

EXTDERFLE

\* model command line

COMLINE

(*one such line for each of NUMCOM command lines*)

\* model input/output

TEMPFLE INFLE

(*one such line for each of NTPLFLE template files*)

INSFLE OUTFLE

(*one such line for each of NINSLFE instruction files*)

\* prior information

PILBL PIFAC \* PARNME + PIFAC \* log(PARNME) ... = PIVAL WEIGHT OBGNME

(*one such line for each of NPRIOR articles of prior information*)

\* predictive analysis

NPREDMAXMIN [PREDNOISE]

PD0 PD1 PD2

ABSPREDLAM RELPREDLAM INITSCHFAC MULSCHFAC NSEARCH

ABSPREDSWH RELPREDSWH

NPREDNORED ABSPREDSTP RELPREDSTP NPREDSTP

\* regularization

PHIMLIM PHIMACCEPT [FRACPHIM] [MEMSAVE]

WFINIT WFMIN WFMAX [LINREG][REGCONTINUE]

WFFAC WFTOL IREGADJ [NOPTREGADJ REGWEIGHTRAT [REGSINGTHRESH]]

\* pareto

PARETO\_OBSGROUP

PARETO\_WTFAC\_START PARETO\_WTFAC\_FIN NUM\_WTFAC\_INC

NUM\_ITER\_START NUM\_ITER\_GEN NUM\_ITER\_FIN

ALT\_TERM

OBS\_TERM ABOVE\_OR\_BELOW OBS\_THRESH NUM\_ITER\_THRESH (*only if ALT\_TERM is non-zero*)

NOBS\_REPORT

OBS\_REPORT\_1 OBS\_REPORT\_2 OBS\_REPORT\_3..(*NOBS\_REPORT items*)

Table A1.1 Variables in the “control data” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| RSTFLE | text | “restart” or “norestart” | instructs PEST whether to write restart data |
| PESTMODE | text | “estimation”, “prediction”, “regularization”, “pareto” | PEST’s mode of operation |
| NPAR | integer | greater than zero | number of parameters |
| NOBS | integer | greater than zero | number of observations |
| NPARGP | integer | greater than zero | number of parameter groups |
| NPRIOR | integer | any integer value | absolute value is number of prior information equations; negative value indicates supply of prior information in indexed format |
| NOBSGP | integer | greater than zero | number of observation groups |
| MAXCOMPDIM | integer | zero or greater | number of elements in compressed Jacobian matrix |
| NTPLFLE | integer | greater than zero | number of template files |
| NINSFLE | integer | greater than zero | number of instruction files |
| PRECIS | text | “single” or “double” | format for writing parameter values to model input files |
| DPOINT | text | “point” or “nopoint” | omit decimal point in parameter values if possible |
| NUMCOM | integer | greater than zero | number of command lines used to run model |
| JACFILE | integer | 0, 1 or -1 | indicates whether model provides external derivatives file |
| MESSFILE | integer | zero or one | indicates whether PEST writes PEST-to-model message file |
| OBSREREF | text | “obsreref”, “obsreref\_N” or “noobsreref” | activates or de-activates observation re-referencing (with an optional pause after re-referencing runs) |
| RLAMBDA1 | real | zero or greater | initial Marquardt lambda |
| RLAMFAC | real | positive or negative, but not zero | dictates Marquardt lambda adjustment process |
| PHIRATSUF | real | between zero and one | fractional objective function sufficient for end of current iteration |
| PHIREDLAM | real | between zero and one | termination criterion for Marquardt lambda search |
| NUMLAM | integer | one or greater; possibly negative with Parallel or BEOPEST | maximum number of Marquardt lambdas to test |
| JACUPDATE | integer | zero or greater | activation of Broyden’s Jacobian update procedure |
| LAMFORGIVE | text | “lamforgive” or “nolamforgive” | treat model run failure during lambda search as high objective function |
| DERFORGIVE | text | “derforgive” or “noderforgive” | accomodates model failure during Jacobian runs by setting pertinent sensitivities to zero |
| RELPARMAX | real | greater than zero | parameter relative change limit |
| FACPARMAX | real | greater than one | parameter factor change limit |
| FACORIG | real | between zero and one | minimum fraction of original parameter value in evaluating relative change |
| ABSPARMAX(*N*) | real | greater than zero | parameter absolute change limit – *N’*th instance |
| IBOUNDSTICK | integer | zero or greater | instructs PEST not to compute derivatives for parameter at its bounds |
| UPVECBEND | integer | zero or one | instructs PEST to bend parameter upgrade vector if parameter hits bounds |
| PHIREDSWH | real | between zero and one | sets objective function change for introduction of central derivatives |
| NOPTSWITCH | integer | one or greater | iteration before which PEST will not switch to central derivatives computation |
| SPLITSWH | real | zero or greater | the factor by which the objective function rises to invoke split slope derivatives analysis until end of run |
| DOAUI | text | “aui”, “auid”, or “noaui” | instructs PEST to implement automatic user intervention |
| DOSENREUSE | text | “senreuse” or “nosenreuse” | instructs PEST to reuse parameter sensitivities |
| BOUNDSCALE | text | “boundscale” or “noboundscale” | parameters are scaled by the inter-bounds interval if using singular value decomposition, LSQR or SVDA |
| NOPTMAX | integer | -2, -1, 0, or any number greater than zero | number of optimization iterations |
| PHIREDSTP | real | greater than zero | relative objective function reduction triggering termination |
| NPHISTP | integer | greater than zero | number of successive iterations over which PHIREDSTP applies |
| NPHINORED | integer | greater than zero | number of iterations since last drop in objective function to trigger termination |
| RELPARSTP | real | greater than zero | maximum relative parameter change triggering termination |
| NRELPAR | integer | greater than zero | number of successive iterations over which RELPARSTP applies |
| PHISTOPTHRESH | real | zero or greater | objective function threshold triggering termination |
| LASTRUN | integer | zero or one | instructs PEST to undertake (or not) final model run with best parameters |
| PHIABANDON | real or text | a positive number or name of a file | objective function value at which to abandon optimization process or filename containing abandonment schedule |
| ICOV | integer | zero or one | record covariance matrix in matrix file |
| ICOR | integer | zero or one | record correlation coefficient matrix in matrix file |
| IEIG | integer | zero or one | record eigenvectors in matrix file |
| IRES | integer | zero or one | record resolution data |
| JCOSAVE | text | “jcosave” or “nojcosave” | save best Jacobian file as a JCO file - overwriting previously-saved files of the same name as the inversion process progresses |
| VERBOSEREC | text | “verboserec” or “noverboserec” | if set to “noverboserec”, parameter and observation data lists are omitted from the run record file |
| JCOSAVEITN | text | “jcosaveitn” or “nojcosaveitn” | write current jacobian matrix to iteration-specific JCO file at the end of every optimization iteration |
| REISAVEITN | text | “reisaveitn” or “noreisaveitn” | store best-fit residuals to iteration-specific residuals file at end of every optimization iteration |
| PARSAVEITN | text | “parsaveitn” or “noparsaveitn” | store iteration specific parameter value files |
| PARSAVERUN | text | “parsaverun” or “noparsaverun” | store run specific parameter value files |

Table A1.2 Variables in the optional “sensitivity reuse” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| SENRELTHRESH | real | zero to one | relative parameter sensitivity below which sensitivity reuse is activated for a parameter |
| SENMAXREUSE | integer | integer other than zero | maximum number of reused sensitivities per iteration |
| SENALLCALCINT | integer | greater than one | iteration interval at which all sensitivities re-calculated |
| SENPREDWEIGHT | real | any number | weight to assign to prediction in computation of composite parameter sensitivities to determine sensitivity reuse |
| SENPIEXCLUDE | text | “yes” or “no” | include or exclude prior information when computing composite parameter sensitivities to determine sensitivity reuse |

Table A1.3 Variables in the optional “automatic user intervention” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| MAXAUI | integer | zero or greater | maximum number of AUI iterations per optimization iteration |
| AUISTARTOPT | integer | one or greater | optimization iteration at which to commence AUI |
| NOAUIPHIRAT | real | between zero and one | relative objective function reduction threshold triggering AUI |
| AUIRESTITN | integer | zero or greater, but not one | AUI rest interval expressed in optimization iterations |
| AUISENSRAT | real | greater than one | composite parameter sensitivity ratio triggering AUI |
| AUIHOLDMAXCHG | integer | zero or one | instructs PEST to target parameters which change most when deciding which parameters to hold |
| AUINUMFREE | integer | greater than zero | cease AUI when only AUINUMFREE parameters are unheld |
| AUIPHIRATSUF | real | between zero and one | relative objective function improvement for termination of AUI |
| AUIPHIRATACCEPT | real | between zero and one | relative objective function reduction threshold for acceptance of AUI-calculated parameters |
| NAUINOACCEPT | integer | greater than zero | number of iterations since acceptance of parameter change for termination of AUI |

Table A1.4 Variables in the optional “singular value decomposition” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| SVDMODE | integer | zero or one | activates truncated singular value decomposition for solution of inverse problem |
| MAXSING | integer | greater than zero | number of singular values at which truncation occurs |
| EIGTHRESH | real | zero or greater, but less than one | eigenvalue ratio threshold for truncation |
| EIGWRITE | integer | zero or one | determines content of SVD output file |

Table A1.5 Variables in the optional “LSQR” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| LSQRMODE | integer | zero or one | activates LSQR solution of inverse problem |
| LSQR\_ATOL | real | zero or greater | LSQR algorithm *atol* variable |
| LSQR\_BTOL | real | zero or greater | LSQR algorithm *btol* variable |
| LSQR\_CONLIM | real | zero or greater | LSQR algorithm *conlim* variable |
| LSQR\_ITNLIM | integer | greater than zero | LSQR algorithm *itnlim* variable |
| LSQR\_WRITE | integer | zero or one | instructs PEST to write LSQR file |

Table A1.6 Variables in the optional “SVD-assist” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| BASEPESTFILE | text | a filename | name of base PEST control file |
| BASEJACFILE | text | a filename | name of base PEST Jacobian matrix file |
| SVDA\_MULBPA | integer | zero or one | instructs PEST to record multiple BPA files |
| SVDA\_SCALADJ | integer | -4 to 4 | sets type of parameter scaling undertaken in super parameter definition |
| SVDA\_EXTSUPER | integer | 0, 1, 2, -2, 3 | sets means by which super parameters are calculated |
| SVDA\_SUPDERCALC | integer | zero or one | instructs PEST to compute super parameter sensitivities from base parameter sensitivities |
| SVDA\_PAR\_EXCL | integer | 0, 1 or -1 | if set to 1, instructs PEST to compute super parameters on basis only of observation group in base parameter PEST control file to which pareto-adjustable weighting is assigned in super parameter PEST control file. If set to -1 all groups other than this form basis for super parameter definition |

Table A1.7 Variables required for each parameter group in the “parameter groups” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| PARGPNME | text | 12 characters or less | parameter group name |
| INCTYP | text | “relative”, “absolute”, “rel\_to\_max” | method by which parameter increments are calculated |
| DERINC | real | greater than zero | absolute or relative parameter increment |
| DERINCLB | real | zero or greater | absolute lower bound of relative parameter increment |
| FORCEN | text | “switch”, “always\_2”, “always\_3”, “switch\_5”, “always\_5” | determines whether higher order derivatives calculation is undertaken |
| DERINCMUL | real | greater than zero | derivative increment multiplier when undertaking higher order derivatives calculation |
| DERMTHD | text | “parabolic”, “outside\_pts”, “best\_fit”, “minvar”, “maxprec” | method of higher order derivatives calculation |
| SPLITTHRESH | real | greater than zero (or zero to deactivate) | slope threshold for split slope analysis |
| SPLITRELDIFF | real | greater than zero | relative slope difference threshold for action |
| SPLITACTION | text | text | “smaller”, “zero” or “previous” |

Table A1.8 Variables required for each parameter in the “parameter data” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| PARNME | text | 12 characters or less | parameter name |
| PARTRANS | text | “log”, “none”, “fixed”, “tied” | parameter transformation |
| PARCHGLIM | text | “relative”, “factor”, or absolute(*N*) | type of parameter change limit |
| PARVAL1 | real | any real number | initial parameter value |
| PARLBND | real | less than or equal to PARVAL1 | parameter lower bound |
| PARUBND | real | greater than or equal to PARVAL1 | parameter upper bound |
| PARGP | text | 12 characters or less | parameter group name |
| SCALE | real | any number other than zero | multiplication factor for parameter |
| OFFSET | real | any number | number to add to parameter |
| DERCOM | integer | zero or greater | model command line used in computing parameter increments |
| PARTIED | text | 12 characters or less | the name of the parameter to which another parameter is tied |

Table A1.9 Variables required for each observation group in the “observation groups” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| OBGNME | text | 12 characters or less | observation group name |
| GTARG | real | positive | group-specific target measurement objective function |
| COVFILE | text | a filename | covariance matrix file associated with group |

Table A1.10 Variables required for each observation in the “observation data” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| OBSNME | text | 20 characters or less | observation name |
| OBSVAL | real | any number | measured value of observation |
| WEIGHT | real | zero or greater | observation weight |
| OBGNME | text | 12 characters or less | observation group to which observation belongs |

Table A1.11 Variables in the optional “derivatives command line” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| DERCOMLINE | text | system command | command to run model for derivatives calculation |
| EXTDERFLE | text | a filename | name of external derivatives file |

Table A1.12 Variables in the “model command line” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| COMLINE | text | system command | command to run model |

Table A1.13 Variables in the “model input” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| TEMPFLE | text | a filename | template file |
| INFLE | text | a filename | model input file |

Table A1.14 Variables in the “model output” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| INSFLE | text | a filename | instruction file |
| OUTFLE | text | a filename | model output file |

Table A1.15 Variables in the optional “prior information” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| PILBL | text | 20 characters or less | name of prior information equation |
| PIFAC | text | real number other than zero | parameter value factor |
| PARNME | text | 12 characters or less | parameter name |
| PIVAL | real | any number | “observed value” of prior information equation |
| WEIGHT | real | zero or greater | prior information weight |
| OBGNME | text | 12 characters or less | observation group name |

Table A1.16 Variables in the optional “predictive analysis” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| NPREDMAXMIN | integer | -1 or 1 | maximise or minimise prediction |
| PREDNOISE | integer | 0 or 1 | instructs PEST to include predictive noise in prediction |
| PD0 | real | greater than zero | target objective function |
| PD1 | real | greater than PD0 | acceptable objective function |
| PD2 | real | greater than PD1 | objective function at which Marquardt lambda testing procedure is altered as prediction is maximised/minimised |
| ABSPREDLAM | real | zero or greater | absolute prediction change to terminate Marquardt lambda testing |
| RELPREDLAM | real | zero or greater | relative prediction change to terminate Marquardt lambda testing |
| INITSCHFAC | real | greater than zero | initial line search factor |
| MULSCHFAC | real | greater than one | factor by which line search factors are increased along line |
| NSEARCH | integer | greater than zero | maximum number of model runs in line search |
| ABSPREDSWH | real | zero or greater | absolute prediction change at which to use central derivatives calculation |
| RELPREDSWH | real | zero or greater | relative prediction change at which to use central derivatives calculation |
| NPREDNORED | integer | one or greater | iterations since prediction raised/lowered at which termination is triggered |
| ABSPREDSTP | real | zero or greater | absolute prediction change at which to trigger termination |
| RELPREDSTP | real | zero or greater | relative prediction change at which to trigger termination |
| NPREDSTP | integer | two or greater | number of iterations over which ABSPREDSTP and RELPREDSTP apply |

Table A1.17 Variables in the optional “regularization” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| PHIMLIM | real | greater than zero | target measurement objective function |
| PHIMACCEPT | real | greater than PHIMLIM | acceptable measurement objective function |
| FRACPHIM | real | zero or greater, but less than one | set target measurement objective function at this fraction of current measurement objective function |
| MEMSAVE | text | “memsave” or “nomemsave” | activate conservation of memory at cost of execution speed and quantity of model output |
| WFINIT | real | greater than zero | initial regularization weight factor |
| WFMIN | real | greater than zero | minimum regularization weight factor |
| WFMAX | real | greater than WFMAX | maximum regularization weight factor |
| LINREG | text | “linreg” or “nonlinreg” | informs PEST that all regularization constraints are linear |
| REGCONTINUE | text | “continue” or “nocontinue” | instructs PEST to continue minimising regularization objective function even if measurement objective function less than PHIMLIM |
| WFFAC | real | greater than one | regularization weight factor adjustment factor |
| WFTOL | real | greater than zero | convergence criterion for regularization weight factor |
| IREGADJ | integer | 0, 1, 2, 3, 4 or 5 | instructs PEST to perform inter-regularization group weight factor adjustment, or to compute new relative weights for regularization observations and prior information equations |
| NOPTREGADJ | integer | one or greater | the optimization iteration interval for re-calculation of regularization weights if IREGADJ is 4 or 5 |
| REGWEIGHTRAT | real | absolute value of one or greater | the ratio of highest to lowest regularization weight; spread is logarithmic with null space projection if set negative |
| REGSINGTHRESH | real | less than one and greater than zero | singular value of **J**t**QJ** (as factor of highest singular value) at which use of higher regularization weights commences if IREGADJ is set to 5 |

Table A1.18 Variables in the optional “pareto” section of the PEST control file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Values** | **Description** |
| PARETO\_OBSGROUP | text | 12 characters or less | name of observation group whose weights are subject to multiplication by a variable weight factor |
| PARETO\_WTFAC\_START | real | zero or greater | initial weight factor for user-specified observation group |
| PARETO\_WTFAC\_FIN | real | greater than PARETO\_WTFAC\_START | final weight factor for user-specified observation group |
| NUM\_WTFAC\_INT | integer | greater than zero | number of weight factor increments to employ in traversing Pareto front |
| NUM\_ITER\_START | integer | zero or greater | number of optimization iterations to employ when using initial weight factor |
| NUM\_ITER\_GEN | integer | greater than zero | number of optimization iterations to employ when using any weight factor other than PARETO\_WTFAC\_START or PARETO\_WTFAC\_FIN |
| NUM\_ITER\_FIN | integer | zero or greater | number of optimization iterations to employ when using final weight factor |
| ALT\_TERM | integer | zero or one | set to one in order to activate PEST termination determined by value of a specified model output |
| OBS\_TERM | text | 20 characters or less | the name of an observation cited in the “observation data” section of the PEST control file whose value will be monitored for possible PEST run termination |
| ABOVE\_OR\_BELOW | text | “above” or “below” | determines whether the monitored model output must be above or below the threshold to precipitate run termination |
| OBS\_THRESH | real | any number | value that monitored model output must exceed or undercut to precipitate model run termination |
| ITER\_THRESH | integer | zero or greater | the number of optimization iterations for which the model output threshold must be exceeded or undercut to precipitate run termination |
| NOBS\_REPORT | integer | zero or greater | number of model outputs whose values to report |
| OBS\_REPORT\_N | text | 20 characters or less | the name of the *N*’th observation whose value is reported in the POD and PPD files written by PEST when run in “pareto” mode |

# Appendix B. Some File Formats

## B.1 Introduction

This appendix provides formats for two types of file that are used by a number of programs belonging to the PEST++ suite. These file types are also used by members of the PEST suite. Most of this appendix is reproduced from Part II of the PEST manual.

## B.2 Matrix File

### B.2.1 General

A number of programs of the PEST++ suite read and/or write a parameter covariance matrix. Optionally, this matrix can reside in a file that adopts “PEST matrix file” specifications. Programs of the PEST++ suite identify this type of file by an extension of *.cov*; however this is not part of its specifications.

### B.2.2 Specifications

The specifications of a matrix file are illustrated by example. A PEST-compatible matrix file holding a matrix with three rows and four columns is illustrated in figure B.1.

|  |
| --- |
| 3 4 2  3.4423 23.323 2.3232 1.3232  5.4231 3.3124 4.4331 3.4442  7.4233 5.4432 7.5362 8.4232  \* row names  apar1  apar2  apar3  \* column names  aobs1  aobs2  aobs3  aobs4 |

Figure B.1 An example of a matrix file.

The first line of a matrix file contains 3 integers. The first two integers (NROW and NCOL) indicate the number of rows and the number of columns in the following matrix. The next integer (named ICODE) is a code, the role of which is discussed shortly.

Following the header line is the matrix itself. The matrix is read row by row, with each row beginning on a new line.

If ICODE is set to 2, the string “\* row names” must follow the matrix. This string must be recorded on the line immediately after the last line of the matrix. On the following and ensuing NROW lines must be recorded NROW character strings; each of these strings is the name associated with the respective row of the matrix. Names must be 20 characters or less in length. The string “\* column names” must follow that. NCOL column names must then follow in a similar format.

For a square matrix ICODE can be set to “1”. This indicates that rows and columns are assigned the same names (as is the case for a covariance matrix). In this case the string “\* row and column names” follows the matrix. The pertinent names are listed on the NROW lines following that.

A special ICODE value is reserved for diagonal matrices. If NCOL is equal to NROW, then ICODE may be set to “-1”. In this case only the diagonal elements of the matrix need to be presented following the integer header line; these should be listed one to a line as illustrated in figure B.2. Following that should be the string “\* row and column names” (for if ICODE is set to “-1” it is assumed that these are the same), followed by the names themselves.

|  |
| --- |
| 5 5 -1  4.5  4.5  2.4  7.53  5.32  \* row and column names  par1  par2  par3  par4  par5 |

Figure B.2 A matrix file containing a diagonal matrix.

## B.3 Uncertainty Files

### B.3.1 Introduction

The PEST++ *parcov()* control variable can accept the name of a parameter uncertainty file as its argument. Programs of the PEST++ suite recognize this type of file by an extension of *.unc*. A parameter uncertainty file gives these programs access to a number of covariance matrices that specify prior uncertainties for subsets of parameters. Individual uncertainties can also be provided for parameters that have no statistical correlation with other parameters.

Note that specifications provided herein for a parameter uncertainty file differ slightly from those provided in section 2.5 of Part II of the PEST manual. Certain options that are not used by programs of the PEST++ suite are omitted from the following specifications.

### B.3.2 Specifications

Figure B.3 illustrates an uncertainty file.

|  |
| --- |
| # An example of an uncertainty file  START STANDARD\_DEVIATION  std\_multiplier 3.0  ro9 1.0  ro10 1.0  ro4 1.0  END STANDARD\_DEVIATION  START COVARIANCE\_MATRIX  file "mat.dat"  variance\_multiplier 1e-2  END COVARIANCE\_MATRIX  START COVARIANCE\_MATRIX  file "cov.mat"  variance\_multiplier 1.0  parameter\_list\_file “list.dat”  END COVARIANCE\_MATRIX  START COVARIANCE\_MATRIX  file "cov1.mat"  first\_parameter kpp1  last\_parameter kpp129  END COVARIANCE\_MATRIX |

Figure B.3 Example of an uncertainty file.

An uncertainty files provides two options for characterizing the uncertainties of subsets of a random vector **k**. These are

1. a list of individual entity standard deviations; and
2. a covariance matrix file.

If desired, a single one of these options can be used to specify the entirety of the covariance matrix of **k**, symbolized by C(**k**). Alternatively, different mechanisms can be used within the same uncertainty file to characterise different parts of the total covariance matrix.

In the following discussion it is assumed that **k** holds a set of model parameters. However this is not integral to its specifications.

An uncertainty file is subdivided into blocks. Each block implements one of the mechanisms of uncertainty characterisation described above. An uncertainty file can have as many blocks as desired. However the following protocols must be observed.

1. Parameters cited in an uncertainty file, and the files cited therein, are matched by name to those featured in a PEST control file which defines a particular inverse problem.
2. The uncertainty of an individual element of an overall **k** vector can be specified only once. Thus, for example, any particular element of a **k** vector cannot be cited in a STANDARD\_DEVIATION block of an uncertainty file if it is also cited in a matrix provided through a COVARIANCE\_MATRIX block of the same uncertainty file.
3. If a parameter is log-transformed in the current inverse or uncertainty analysis problem (as specified in the PEST control file which governs that problem), then specifications of variance, covariance or standard deviation provided in an uncertainty file must pertain to the log (to base 10) of the parameter.

Each block of an uncertainty file must begin with a START line and finish with an END line as illustrated in figure B.3; in both cases the type of block must be correctly characterised following the START and END designators. Within each block, data entry must follow the keyword protocol. Thus each line must comprise a keyword, followed by the value (numerical or text) associated with that keyword. Filenames must be surrounded by quotes if they contain spaces. With one exception (the *std\_multiplier* keyword in the STANDARD\_DEVIATION block), keywords within a block can be supplied in any order; some can be omitted if desired. Keywords and block names are case insensitive.

Blank lines can appear anywhere within an uncertainty file. So too can comment lines; these are recognised through the fact that their first character is “#”.

Each of the blocks appearing in an uncertainty file is now discussed in detail.

#### B.3.2.1 The STANDARD\_DEVIATION Block

In a STANDARD\_DEVIATION block, entity names (i.e. individual parameters) are listed one to a line followed by their standard deviations. As stated above, if a parameter is log-transformed in a parameter estimation or uncertainty analysis process, then this standard deviation should pertain to the log (to base 10) of the parameter. Parameters can be supplied in any order. Optionally a *std\_multiplier* keyword can be supplied in the STANDARD\_DEVIATION block; if so, it must be the first item in the block. All standard deviations supplied on ensuing lines are multiplied by this factor (the default value of which is 1.0).

Parameters cited in a STANDARD\_DEVIATION block are assumed to be uncorrelated with other parameters/observations. Thus off-diagonal elements of the covariance matrix corresponding to these items are zero. Pertinent diagonal elements of the covariance matrix are calculated by squaring standard deviations (after multiplication by the *std\_multiplier*).

If a parameter is featured in a STANDARD\_DEVIATION block but is not featured in the PEST control file which defines an inverse, optimization of uncertainty analysis problem, programs of the PEST++ suite ignore it.

#### B.3.2.2 The COVARIANCE\_MATRIX Block

Where a parameter uncertainty file provides one or more covariance matrices, each for a subgroup of **k** which shows within-group parameter correlation, all of these matrices are collectively included in the larger C(**k**) covariance matrix, together with variances calculated from parameter standard deviations supplied in one or more STANDARD\_DEVIATION blocks which may also be featured in the parameter uncertainty file. Optionally, all elements of a user-supplied covariance matrix provided through a COVARIANCE\_MATRIX block can be multiplied by a factor. This factor (for which the default value is 1.0) is supplied following the *variance\_multiplier* keyword.

A number of options are available for storage of the matrix housed in the covariance matrix file cited in a COVARIANCE\_MATRIX block. Matrix storage may follow the PEST matrix file protocol described in section B.2. If this is the case, then the first line of this file must include 3 integers, the first two of which (specifying the number of rows and columns in the matrix) must be identical. The third integer must be “1” or “-1”. The matrix itself must follow this integer header line. Elements within this matrix must be space-delimited; rows can be wrapped onto consecutive lines, but each new matrix row must start on a new line. This matrix must be followed by a line containing the “\* row and column names” string. Following this must be the names of the parameters to which the matrix pertains. If the matrix file follows this protocol then the COVARIANCE\_MATRIX block provided in the parameter uncertainty file must be identical in format to the first of the COVARIANCE\_MATRIX blocks shown in figure B.3; it can only feature a *file* keyword and an optional *variance\_multiplier* keyword.

The PLPROC parameterization utility supplied with PEST writes matrix files whose format is slightly different from that described in section B.2 of this manual. This format retains a three-integer header. The first two numbers in this header must specify the number of rows and number of columns in the matrix. The third number must be 1 or -1; 1 indicates that the matrix is non-diagonal while -1 specifies a diagonal matrix. The matrix itself follows this header line. However the matrix is not followed by a list of row and column names; instead the end of the file coincides with the end of the matrix. In this case the COVARIANCE\_MATRIX block must adopt either the second or third protocols illustrated in figure B.3 for linking matrix rows and columns to the names of parameters cited in the “parameter data” section of a PEST control file. For the second option the user provides a file in which parameters are listed one to a line. There must be as many such lines are there are rows/columns in the covariance matrix. Alternatively, the third protocol can be followed. In this case the COVARIANCE\_MATRIX block must contain both of the *first\_parameter* and *last\_parameter* keywords. These refer back to the PEST control file which defines the current problem. Parameters within this PEST control file between and including the user-nominated *first\_parameter* and *last\_parameter* parameters are then associated with rows and columns of the covariance matrix, with the ordering of parameters in the matrix file being the same as that in the PEST control file. Naturally, the number of parameters in this implied parameter list must be the same as the number of rows and columns in the covariance matrix.

In all of the above cases the *variance\_multiplier* keyword is optional. If omitted, it is assumed to be 1.0.

The following should be noted.

1. A covariance matrix must be positive definite.
2. If the first or second of the above COVARIANCE\_MATRIX block protocols is adopted, then the order of rows and columns of the covariance matrix (which corresponds to the order of parameters listed either within the matrix file itself or in a user-supplied parameter list file) is arbitrary. The PEST++ program which reads a parameter uncertainty file will re-arrange matrix rows and columns so that they correspond to the order of adjustable parameters supplied in the PEST control file on which the current inverse or uncertainty analysis problem is based.
3. A covariance matrix that is cited in a parameter uncertainty file need not describe all of the parameters associated with the current inverse problem, for it need only pertain to a subset of these. Other parameters can be associated with other COVARIANCE\_MATRIX blocks and/or can be cited in one or more STANDARD\_DEVIATION blocks supplied in the same uncertainty file. However a covariance matrix must not be associated with any parameters which do NOT pertain to the current inverse or uncertainty analysis problem.
4. If a parameter is log-transformed in the PEST control file, the variance and covariances pertaining to that parameter as supplied in a covariance matrix file must in fact pertain to the log of that parameter.