#### **A close up of a purple sign Description automatically generated**

**Version 5.0.8**



PEST++ Development Team

December 2020

**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.

The writing of this manual, and improvements to the PEST++ parallel run manager, were funded by the Queensland Water Modelling Network. A little funding was also provided by the GMDSI project overseen by National Centre for Groundwater Research and Training and sponsored by BHP-Billiton and Rio Tinto.

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 modelling education based on the PEST++ suite:

Chas Egan (Queensland Department of Environment and Science)

Mike Toews (GNS Science)

Mike Fienen (USGS)

Randy Hunt (USGS)

Wes Kitlasten (USGS)

Matt Knowling (GNS Science)

Brioch Hemmings (GNS Science)

Chris Muffels (SS Papadopoulos and Associates)

Damian Merrick (HydroAlgorithmics)

Chis Nicol (Groundwater Logic)

**Preface**

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 serves as manual the current version of PEST++.

The term “PEST++” 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 (Welter et al; 2019) for details.

A theme that runs through all members of the PEST++ suite is that of adding value to environmental modelling 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, of deploying advanced regularisation devices, and of exploring parameter and predictive uncertainty. One member of the suite optimizes environmental system management in a modelling 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.

**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](#_Toc32564031)

[1.1 PEST++ and PEST 1](#_Toc32564032)

[1.2 Software Installation 3](#_Toc32564033)

[1.3 This Document 3](#_Toc32564034)

[1.4 A Model: Some Considerations 3](#_Toc32564035)

[1.4.1 Running a Model 3](#_Toc32564036)

[1.4.2 Model Input and Output Files 4](#_Toc32564037)

[1.4.3 Differentiability of Model Outputs 5](#_Toc32564038)

[1.4.4 Observations and Predictions 6](#_Toc32564039)

[2. The PEST(++) Model Interface 7](#_Toc32564040)

[2.1 Introduction 7](#_Toc32564041)

[2.2 PEST++ Input Files 7](#_Toc32564042)

[2.3 Template Files 7](#_Toc32564043)

[2.3.1 Model Input Files 7](#_Toc32564044)

[2.3.2 An Example 8](#_Toc32564045)

[2.3.3 The Parameter Delimiter 9](#_Toc32564046)

[2.3.4 Parameter Names 9](#_Toc32564047)

[2.3.5 Setting the Parameter Space Width 10](#_Toc32564048)

[2.3.6 How a Parameter Space is Filled with a Number 11](#_Toc32564049)

[2.3.7 Multiple Occurrences of the Same Parameter 12](#_Toc32564050)

[2.3.8 Preparing a Template File 12](#_Toc32564051)

[2.4 Instruction Files 12](#_Toc32564052)

[2.4.1 Precision in Model Output Files 13](#_Toc32564053)

[2.4.2 How Model Output Files are Read 13](#_Toc32564054)

[2.4.3 An Example Instruction File 14](#_Toc32564055)

[2.4.4 The Marker Delimiter 15](#_Toc32564056)

[2.4.5 Observation Names 15](#_Toc32564057)

[2.4.6 The Instruction Set 16](#_Toc32564058)

[2.4.7 Making an Instruction File 26](#_Toc32564059)

[3. Some Important PEST++ Features 27](#_Toc32564060)

[3.1 General 27](#_Toc32564061)

[3.2 Parameter Adjustment 27](#_Toc32564062)

[3.2.1 Parameter Transformation 27](#_Toc32564063)

[3.2.2 Fixed and Tied Parameters 27](#_Toc32564064)

[3.2.3 Upper and Lower Parameter Bounds 28](#_Toc32564065)

[3.2.4 Scale and Offset 28](#_Toc32564066)

[3.2.5 Parameter Change Limits 29](#_Toc32564067)

[3.3 Calculation of Derivatives 31](#_Toc32564068)

[3.3.1 General 31](#_Toc32564069)

[3.3.2 Forward or Central Differences 31](#_Toc32564070)

[3.3.3 Parameter Increments for Two and Three-Point Derivatives 32](#_Toc32564071)

[3.3.4 Settings for Three-Point Derivatives 34](#_Toc32564072)

[3.3.5 How to Obtain Derivatives You Can Trust 34](#_Toc32564073)

[3.3.6 Looking at Model Outputs under the Magnifying Glass 35](#_Toc32564074)

[3.4 The Jacobian Matrix File 36](#_Toc32564075)

[3.5 The Objective Function 37](#_Toc32564076)

[4. The PEST Control File 39](#_Toc32564077)

[4.1 General 39](#_Toc32564078)

[4.2 Naming Conventions 39](#_Toc32564079)

[4.3 Sections 40](#_Toc32564080)

[4.4 Control Variables 41](#_Toc32564081)

[4.5 The PESTCHEK Utility 44](#_Toc32564082)

[4.6 Control Data Section 44](#_Toc32564083)

[4.6.1 General 44](#_Toc32564084)

[4.6.2 First Line 45](#_Toc32564085)

[4.6.3 Second Line 45](#_Toc32564086)

[4.6.4 Third Line 45](#_Toc32564087)

[4.6.5 Fourth Line 45](#_Toc32564088)

[4.6.6 Fifth Line 46](#_Toc32564089)

[4.6.7 Sixth Line 46](#_Toc32564090)

[4.6.8 Seventh Line 46](#_Toc32564091)

[4.6.9 Eighth Line 47](#_Toc32564092)

[4.6.10 Ninth Line 47](#_Toc32564093)

[4.7 Singular Value Decomposition Section 47](#_Toc32564094)

[4.8 Parameter Groups Section 48](#_Toc32564095)

[4.8.1 General 48](#_Toc32564096)

[4.8.2 Parameter Group Variables 49](#_Toc32564097)

[4.9 Parameter Data Section 51](#_Toc32564098)

[4.9.1 General 51](#_Toc32564099)

[4.9.2 First Part 52](#_Toc32564100)

[4.9.3 Second Part 55](#_Toc32564101)

[4.10 Observation Groups Section 55](#_Toc32564102)

[4.11 Observation Data Section 56](#_Toc32564103)

[4.12 Model Command Line Section 57](#_Toc32564104)

[4.13 Model Input Section 58](#_Toc32564105)

[4.14 Model Output Section 58](#_Toc32564106)

[4.15 Prior Information Section 58](#_Toc32564107)

[4.16 Regularization Section 61](#_Toc32564108)

[4.17 Control Variables for PEST++ Programs 63](#_Toc32564109)

[4.18 Keyword and External File Control File Format 67](#_Toc32564110)

[4.18.1 Keyword and Consolidated Algorithmic Variables 67](#_Toc32564111)

[4.18.2 External file support 67](#_Toc32564112)

[5. Running PEST++ Programs 70](#_Toc32564113)

[5.1 General 70](#_Toc32564114)

[5.2 Model Runs in Serial 70](#_Toc32564115)

[5.2.1 Concepts 70](#_Toc32564116)

[5.2.2 Running PESTPP-XXX 70](#_Toc32564117)

[5.3 Model Runs in Parallel 71](#_Toc32564118)

[5.3.1 Concepts 71](#_Toc32564119)

[5.3.2 Manager to Worker Communication 72](#_Toc32564120)

[5.3.3 Running PESTPP-XXX as Manager and Worker 72](#_Toc32564121)

[5.3.4 Run Management Record File 73](#_Toc32564122)

[5.3.5 Run Management Control Variables 73](#_Toc32564123)

[5.4 Run Book-Keeping Files 74](#_Toc32564124)

[6. PESTPP-GLM 75](#_Toc32564125)

[6.1 Introduction 75](#_Toc32564126)

[6.2 Highly Parameterized Inversion 75](#_Toc32564127)

[6.2.1 Basic Equations 75](#_Toc32564128)

[6.2.2 Choosing the Regularization Weight Factor 77](#_Toc32564129)

[6.2.3 Inter-Regularization Group Weighting 78](#_Toc32564130)

[6.2.4 Choosing Values for the Marquardt Lambda 78](#_Toc32564131)

[6.2.5 Singular Value Decomposition 79](#_Toc32564132)

[6.2.6 SVD-Assist 81](#_Toc32564133)

[6.2.7 Expediting the First Iteration 83](#_Toc32564134)

[6.2.8 First Order, Second Moment Uncertainty Analysis and Monte Carlo 84](#_Toc32564135)

[6.2.9 Model Run Failure 86](#_Toc32564136)

[6.2.10 Composite Parameter Sensitivities 87](#_Toc32564137)

[6.2.11 Other Controls 87](#_Toc32564138)

[6.2.12 Running PESTPP 87](#_Toc32564139)

[6.2.13 PESTPP-GLM Output Files 87](#_Toc32564140)

[6.3 Differential Evolution 89](#_Toc32564141)

[6.3.1 General 89](#_Toc32564142)

[6.3.2 The DE Method 90](#_Toc32564143)

[6.3.3 Using DE in PESTPP 92](#_Toc32564144)

[6.3.4 Running PESTPP 92](#_Toc32564145)

[6.3.5 PESTPP-GLM Output Files 92](#_Toc32564146)

[6.4 Summary of PESTPP-GLM Control Variables 93](#_Toc32564147)

[6.4.1 General 93](#_Toc32564148)

[6.4.2 Control Variables in the PEST Control File 93](#_Toc32564149)

[6.4.3 PEST++ Control Variables 94](#_Toc32564150)

[7. PESTPP-SEN 97](#_Toc32564151)

[7.1 Introduction 97](#_Toc32564152)

[7.1.1 General 97](#_Toc32564153)

[7.1.2 Grouped Parameters 98](#_Toc32564154)

[7.2 Method of Morris 98](#_Toc32564155)

[7.2.1 Elementary Effects 98](#_Toc32564156)

[7.2.2 Sampling Scheme 99](#_Toc32564157)

[7.2.3 Control Variables 100](#_Toc32564158)

[7.3 Method of Sobol 101](#_Toc32564159)

[7.3.1 Sensitivity Indices 101](#_Toc32564160)

[7.3.2 Control Variables 102](#_Toc32564161)

[7.4 PESTPP-SEN Output Files 103](#_Toc32564162)

[8. PESTPP-OPT 104](#_Toc32564163)

[8.1 Introduction 104](#_Toc32564164)

[8.1.1 A Publication 104](#_Toc32564165)

[8.1.2 Overview 104](#_Toc32564166)

[8.1.3 Calculation of Uncertainty 105](#_Toc32564167)

[8.1.4 Optimization 106](#_Toc32564168)

[8.1.5 Chance Constraints 108](#_Toc32564169)

[8.2 Using PESTPP-OPT 108](#_Toc32564170)

[8.2.1The PEST Control File 108](#_Toc32564171)

[8.2.2 Decision Variables and Parameters 109](#_Toc32564172)

[8.2.3 Defining the Objective Function 110](#_Toc32564173)

[8.2.4 Constraints 110](#_Toc32564174)

[8.2.5 Observations 111](#_Toc32564175)

[8.2.6 Regularization 112](#_Toc32564176)

[8.2.7 Prior Covariance Matrix 113](#_Toc32564177)

[8.2.8 Risk 113](#_Toc32564178)

[8.2.9 Jacobian and Response Matrices 114](#_Toc32564179)

[8.2.10 Solution Convergence 115](#_Toc32564180)

[8.2.11 Other Control Variables 115](#_Toc32564181)

[8.2.12 Final Model Run 115](#_Toc32564182)

[8.2.13 Restarts 116](#_Toc32564183)

[8.2.14 Zero Run Solution 116](#_Toc32564184)

[8.3 PESTPP-OPT Output Files 116](#_Toc32564185)

[8.4 Summary of Control Variables 117](#_Toc32564186)

[9. PESTPP-IES 120](#_Toc32564187)

[9.1 Introduction 120](#_Toc32564188)

[9.1.1 Publications 120](#_Toc32564189)

[9.1.2 Overview 120](#_Toc32564190)

[9.1.3 Ensemble Kalman Filters and Ensemble Smoothers 122](#_Toc32564191)

[9.1.4 Some Repercussions of Using Ensembles 123](#_Toc32564192)

[9.1.5 Iterations 124](#_Toc32564193)

[9.1.6 Measurement Noise 125](#_Toc32564194)

[9.1.7 Regularization 126](#_Toc32564195)

[9.1.8 Base Realization 126](#_Toc32564196)

[9.1.9 Parameter Transformation Status 127](#_Toc32564197)

[9.1.10 Inequality Observations 127](#_Toc32564198)

[9.1.11 Localization 128](#_Toc32564199)

[9.1.12 Use of observation noise covariance matrices 131](#_Toc32564200)

[9.1.13 Detecting and resolving prior-data conflict 131](#_Toc32564201)

[9.2 Using PESTPP-IES 132](#_Toc32564202)

[9.2.1 General 132](#_Toc32564203)

[9.2.2 Initial Realizations 133](#_Toc32564204)

[9.2.3 “Regularization” 136](#_Toc32564205)

[9.2.4 Prior Parameter Scaling 138](#_Toc32564206)

[9.2.5 The Marquardt Lambda 138](#_Toc32564207)

[9.2.6 Restarting 140](#_Toc32564208)

[9.2.7 Failed Model Runs 141](#_Toc32564209)

[9.2.8 Reporting 142](#_Toc32564210)

[9.2.9 Termination Criteria and Objective Functions 142](#_Toc32564211)

[9.3 PESTPP-IES Output Files 143](#_Toc32564212)

[9.3.1 CSV Output Files 143](#_Toc32564213)

[9.3.2 Non-CSV Output Files 145](#_Toc32564214)

[9.4 Summary of Control Variables 145](#_Toc32564215)

[10. PESTPP-SWP 150](#_Toc32564216)

[10.1 Introduction 150](#_Toc32564217)

[10.2 Using PESTPP-SWP 150](#_Toc32564218)

[10.3 Summary of Control Variables 151](#_Toc32564219)

[11. PESTPP-PSO 153](#_Toc32564220)

[11.1 Introduction 153](#_Toc32564221)

[11.1.1 Publications and Overview 153](#_Toc32564222)

[1.1.1 Basic Single-Objective Particle Swarm Optimization 155](#_Toc32564223)

[11.1.2 Multi-Objective Particle Swarm optimization 156](#_Toc32564224)

[11.1.3 Decision Variable Transformations 157](#_Toc32564225)

[11.2 Using PESTPP-PSO 157](#_Toc32564226)

[11.2.1 General 157](#_Toc32564227)

[11.2.2 Estimation Mode 159](#_Toc32564228)

[11.2.3. Pareto mode 163](#_Toc32564229)

[11.2.3 PESTPP-PSO External Initial-Swarm File 166](#_Toc32564230)

[11.3 PESTPP-PSO Output Files 168](#_Toc32564231)

[12. References 170](#_Toc32564232)

[Appendix A. PEST Control File Specifications 173](#_Toc32564233)

[Appendix B. Some File Formats 190](#_Toc32564234)

[B.1 Introduction 190](#_Toc32564235)

[B.2 Matrix File 190](#_Toc32564236)

[B.2.1 General 190](#_Toc32564237)

[B.2.2 Specifications 190](#_Toc32564238)

[B.3 Uncertainty Files 191](#_Toc32564239)

[B.3.1 Introduction 191](#_Toc32564240)

[B.3.2 Specifications 191](#_Toc32564241)

[B.4 JCO File 195](#_Toc32564242)

[B.4.1 Introduction 195](#_Toc32564243)

[B.4.2 Specifications 195](#_Toc32564244)

[B.5 JCB File 195](#_Toc32564245)

[B.5.1 Introduction 195](#_Toc32564246)

[B.5.2 Specifications 196](#_Toc32564247)

[B.5.3 Distinguishing between a JCO and a JCB File 196](#_Toc32564248)

# 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 names of all programs which comprise the suite begin with “PESTPP”. 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.

PESTPP-GLM is the original member of the PEST++ suite; it was originally named “PEST++” itself. Like PEST, PESTPP-GLM 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.

For all members of the PEST++ suite, 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 a member of the PEST++ suite 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; then 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-GLM was that its parallel run manager be modular, and separate from PESTPP-GLM, 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-SEN | 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 |

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; see Welter et al (2019).

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-GLM. 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.

When reading a PEST control file, most programs of the PEST++ suite 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). 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 (2018c) 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.

With the exception of PESTPP-PSO, the PEST++ tools are C++. Users who want to compile PEST++ will need a C++11 or greater compiler (MSVC for windows, gcc 4.9 or greater, intel C++). Users who want to compile PESTPP-PSO will also need a fortran compiler.

## 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 modelling 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 Part 2 of the PEST manual (Doherty, 2018b) which describes PEST utility support software. Attention should also be given to documentation of the PEST Groundwater Data Utilities. At the time of writing, both of these utility suites are undergoing expansion in order to provide better support for members of the PEST++ suite.

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 (under uncertainty) through sequential linear programming. These methods are powerful. Furthermore, they can be used to manipulate large numbers of parameters and/or decision variables. The power of some of these methods 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. Where needed, 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 some 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 modelling 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 PEST++ parlance, any number that is 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 modelling 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-GLM, PESTPP-SEN, PEST utilities and 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). Programs of the PEST suite require that parameter names be from 1 to 12 characters in length; however parameter name lengths are unlimited in the programs of the PEST++ suite. Any characters in a parameter name are 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 two hundred 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 those that 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 declares PRECIS to be exclusively “double”. Furthermore, where parameter spaces are greater than about 23 characters in width (which can arise often because of PEST++ support for unlimited length parameter names), numbers are padded to the left with zeroes to fill the space (for example, “000001.2345678901234e+001”). Testing has indicated that most numerical models, as well as pre- and post-processing codes (written in languages such as Python) tolerate this padding. This padding is off by default and is controlled with the *fill\_tpl\_zeros* option.

### 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 has no difficulties in reading input files written by a program from the PEST++ suite.

Note that TEMPCHEK, like PEST, sets a 12 character limit on the length of parameter names.

## 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-GLM 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. For PEST, observation names must be 20 characters or less in length; for programs of the PEST++ suite, they are unlimited. These 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). Note, unlike PEST, PEST++ treats the comma (“,”) as a whitespace-type character, given its very-common use. Additionally, PEST++ also supports an optional “++” style argument *additional\_ins\_delimiters* that users can employ to include additional characters to treat as delimiters (in addition to space, tab, and comma). Using these additional delimiters makes instruction file processing much faster than using repeating secondary markers, especially for reading large CSV-format files.

#### 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 or comma (or addtitional instruction file delimiters) 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, comma (or additional delimiters) 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 or comma between (and including) the two nominated column numbers, or if it encounters non-numeric characters or two number fragments separated by whitespace or comma, 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/comma (or additional delimiters) 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, non-comma, non-delimiter 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, non-comma, non-delimiters, 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/comma/addtitional delimiter 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!

If the numbers in the above example been separated by commas instead of whitespace, PEST++ would have read this line in the same way

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

As line lengths in PEST++ are unlimited, PEST++ does not support the line continuation character of the PEST instruction set.

### 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. Note that by default, PEST++ programs will check the consistency between observations listed in the control file and the instruction file(s). This functionality can be disable with the *check\_tplins* option.

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. (Note that INSCHEK and PESTCHEK, like PEST, set a 20 character limit on the length of observation names.)

# 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.

Programs of the PEST++ suite allow groups of parameters to be adjusted as a single parameter through use of a PEST++-specific control variable named *tie\_by\_group()*. This has a similar effect to multiple PARTRANS settings of “tied”, but can be easier to implement in many contexts.

In PEST, the bounds of tied parameters are not enforced. The same is true in PEST++ unless users activate tied parameter bounds enforcement via the *enforce\_tied\_bounds* option. This option should be used with caution because it can effectively limit the bounds range of the adjustable parameters–this has a number of implications across the various codes of the PEST++ suite. However, under some circumstances, it can be important to maintain all parameters within their bounds for stability reasons.

### 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-GLM 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-GLM 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-GLM. The limits imposed by PESTPP-GLM may be of two types, namely “relative”, and “factor”. (PEST allows an “absolute” limit in addition to these.) You must inform PESTPP-GLM 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-GLM 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-GLM 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-GLM 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-GLM 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-GLM 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-GLM 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-GLM 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-GLM, 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-GLM, 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-different 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-GLM 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 system state 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-GLM 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-GLM 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-GLM 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-GLM 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.) The protocol used by programs of the PEST++ suite for storage of a Jacobian matrix in a JCO file is identical to that used by PEST; see appendix 3.5. 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-GLM 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-GLM 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-GLM 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 offer. 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. Furthermore, any particular PEST++ program may use a particular PEST control variable under certain circumstances.

|  |
| --- |
| 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. At the same time, 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-GLM in “regularization” mode. If this is the case, then PESTPP-GLM will read the “regularization” section of the PEST control file if it is present; if a “regularization” section is not present, PESTPP-GLM 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-GLM 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-GLM. Furthermore, it is ignored by PESTPP-GLM 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-GLM 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-GLM,

(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-GLM 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-GLM 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-GLM 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-GLM setup is correct, and that weights assigned to observations in the “observation data” section of the PEST control file are suitable.

If NOPTMAX is set to -1 or -2, PESTPP-GLM 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-GLM 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-GLM considers that the inversion process is at an end. Alternatively, if PESTPP-GLM 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-GLM ceases execution. The relative parameter change between iterations for any parameter is calculated using equation 3.2. PESTPP-GLM 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-GLM 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-GLM 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-GLM 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-GLM 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-GLM 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-GLM 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. For PEST this name must be 12 characters or less in length; for programs of the PEST++ suite it must be 200 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 PEST 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-GLM (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 200 characters in length. However if compatibility with the PEST suite is sought, it should be 12 characters or less 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-SEN);
* 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. For PEST-suite programs a parameter name must be 12 characters or less in length while for programs of the PEST++ suite, its name can extend to 200 characters 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. It can also be accommodated using “secondary parameter” functionality available through PEST\_HP; see documentation of PEST\_HP 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 members of the PEST++ suite which imposes limits on parameter changes in this way are PESTPP-GLM and (optionally) PESTPP-IES (through the *ies\_enforce\_chglim*  option).
* 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-GLM 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-GLM or PESTPP-IES, 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-GLM, 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-GLM (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.

Programs of the PEST++ suite support a protocol for tying parameters together that may be convenient in some contexts. This is activated using the *tie\_by\_groups()* control variable. Denoting this variable as *true* effectively ties all parameters in a group to a single member of that group so that initial, intra-group parameter ratios are maintained as parameters are adjusted. However it is important to note that this action does not affect any parameters that a user designates as tied to another parameter, or that has another parameter tied to it.

## 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 12 characters or less in length for PEST-suite programs. In contrast, for programs of the PEST++ suite, observation group names can extend to 200 characters in length. In both cases these names 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

OBSNME, character variable, contains the observation name. For programs of the PEST suite an observation name must be 20 characters or less in length; for programs of the PEST++ suite, an observation name can extend to 200 characters in length. In both cases it must contain no spaces, and 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

WEIGHT, a real variable, 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 a PEST control file. As has been previously stated, observation group names must be 12 characters or less in length if PEST compatibility is sought. However PEST++ allows an observation group name to extend to 200 characters 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-GLM (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 20 characters in length if PEST-compatibility is required. However programs of the PEST++ suite will permit a 200 character length for prior information labels. Prior information labels 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-GLM and PESTPP-IES 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 12 characters or less in length for PEST-compatibility; however programs of the PEST++ suite allow observation group names of 200 characters 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-GLM 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-GLM in chapter 6 of this manual. For more general information on regularization, see Doherty (2015).

#### PHIMLIM

PHILIM is the target measurement objective function; it is the measurement objective function that PESTPP-GLM “aims for” while keeping the regularization objective function as low as possible. During every iteration of the inversion process, PESTPP-GLM 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-GLM run, before running PESTPP-GLM again. Parameter values achieved during the next PESTPP-GLM 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

DuringPHIMACCEPT 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-GLM 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-GLM 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-GLM 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-GLM 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-GLM 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-GLM 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-GLM 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-GLM 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-GLM run), then you should provide WFINIT with this value. Otherwise simply set it to 1.0.

#### WFMIN, WFMAX

WFMIN and WFMAX 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-GLM 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-GLM 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-GLM 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-GLM 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-GLM does not support the same range of values for IREGADJ as does PEST. When using PESTPP-GLM, 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-GLM 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 use 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. If a ++ argument is not recognized, this will raise an exception; if users want to “forgive” unrecognized ++ args, the “++forgive\_unknown\_args(true)” should be supplied.

## 4.18 Keyword and External File Control File Format

As of version 4.3.0, the programs in the PEST++ suite support an enhanced control file format, which has been designed to support an increasingly diverse set of tools. For the users perspective, this new format requires significantly less “in depth” knowledge of the algorithmic controls over the PEST++ tools as all of these control varaibles now have internal default values, so arguments that are not specified in the control file simply use these internal defaults. Additionally, the sections of the control file with listed of data (e.g. “\* parameter data”, “\* observation data”, among others) can now be stored in external files; only the name of this file and some optional parsing information is needed in the control file.

Empty blanks lines are tolerated in the enhanced format as are lines that start with “#” as comment lines. Note once a “#” is encountered on a line, the remaining characters to the right are ignored. This allows users to have both full comment lines and partial comment lines.

Below is a more detailed description of this enhanced format.

## 4.18.1 Keyword and Consolidated Algorithmic Variables

The enhanced control file format now accepts a “\* control data keyword” section. This section replaces the following section in the standard control file format:

* \* control data
* \* singular value decomposition
* \* regularization
* ++ arguments

Therefore, if users construct a “\* control data keyword” section, these cannot also be listed–an error message will be issued if you try. The format of the “\* control data keyword”, as expected is by keywords. An example section is shown on Figure 4.13. Keyword-value pairs should be separated by one or more whitespace (tabs acceptable as well). For values that have multiple entires (like the PESTPP-GLM control variable “lambdas”), users should comma separate each separate value (as shown on Figure 4.13). If a ++ argument is not recognized, this will raise an exception; if users want to “forgive” unrecognized ++ args, the “forgive\_unknown\_args true” should be supplied.

## 4.18.2 External file support

As shown in Figure 4.13, the enhanced control file format allows users to store list-directed input in external files. These files must have the same number of entrires on each line and the location of these files in the users directory structure must be the path from where the control file is located to where the external file is located. For example, if the control file is saved in the directory “model” and parameter data is stored in the file “parameters.csv”, the entry in the “\* parameter data external” must be “parameters.csv”, regardless of where the calling program is instantiated.

Users will no doubt realize that counters for the number of parameters, observations, etc are no longer needed. Additionally, the \* parameter group and \* observation group sections are now optional – default values for all \* parameter group entries are available internally for each PEST++ tool that needs parameter group information.

The external file support for listed-directed sections of control file is activated by appending “external” to the section name (Figure 4.13). Multiple external files can be listed for each external section, allowing users to group inputs into separate files if desired.

Each line in the external sections is required to have one entry and may have additional, optional entries. The one required entry is the filename, the optional entries provide instructions for how to parse the file. Currently supported optional entries are:

* “sep” (for separator). Default is “,” (comma). For whitespace-delimited (one or more whitespaces) use “w”
* “missing\_values”. Default is empty/whitespace (for comma-separated). Users are strongly encouraged to supply this option for whitespace-delimited files.

These options should be supplied as whitespace-separated options on the same line as the filename (see Figure 4.13 for an example)

Using the “sep”, each line of the external file must have the same number entries. Additionally, each external file must have a header line for the first row that labels the fields in the file. The names of the fields must be either the formal PEST variable names or an alias. The formal names (and aliases in parentheses if available) for each are listed below:

* \* parameter data external
  + parnme(name) – parameter name
  + partrans(transform) – parameter transformation
  + parval1(value) – initial parameter value
  + parubnd(upper\_bound) – parameter upper bound
  + parlbnd(lower\_bound) – parameter lower bound
  + pargp(group) – parameter group
* \* observation data external
  + obsnme(name) – observation name
  + obsval(value) – observation value
  + weight – observation weight
  + obgnme(group) – observation group
* \* model input external
  + pest\_file – template file name
  + model\_file – corresponding model input file name
* \* model output external
  + pest\_file – instruction file name
  + model\_file – corresponding model output file name
* \* prior information external
  + pilbl – the name of the PI equation
  + equation – the PI equation, including both LHS and RHS
  + weight – the PI equation weight
  + obgnme – the PI equation group

Each of the listed formal names (or its alias) must be found in the header row of the external file. Uesrs can put any additional columns in these external files that they wish – this is a nice way to carry metadata through a PEST++ analysis. It is anticipated that current and future PEST++ tools may require additional columns in external files – those requirements will be listed in the individual tool sections of this manual.

Through the use of these external files, PEST++ programs support the use of following optional quantities:

* *standard\_deviation*: If this column appears in any external file for either \* *parameter data external* or *\* observation data external* section, then the rows of that external file with values in the *standard\_deviation* column that can be parsed to double-precision floating point values will be used as the prior parameter standard deviation and observation noise standard deviation, respectively. This is to allow the parameter bounds to serve soley an algorithmic function and the weights to be used soley to define a composite objective function. In this way, users can define a prior uncertainty for parameters separate from the parameter bounds and an observation noise separate from observation weights
* *upper\_bound, lower\_bound*: if this column appears in any external file for *\* observation data external* section, then the rows of that external file with values in the *upper\_bound*  and/or *lower\_bound* column that can be parsed to double-precision floating point values will be used to limit the values of realized observation ensemble values. This is primarily used in the ensemble tools where realizations of noise are drawn and added to the observation values in the control file; In some cases, extreme noise values may be drawn and the use of *upper\_bound* and *lower\_bound* can reduce the effect of these extreme values by replacing realized observation values greater than *upper\_bound* with the value of ­*upper\_bound* and visa versa from *lower\_bound*.

Note that not all external files or even every row in an external file must have a valid value for these optional quantities. For rows where the values are “missing”, the standard operating procedure is applied.

|  |
| --- |
| # comment line  pcf  \* control data keyword # more comments here:  pestmode estimation  maxsing 100  #Phimlim 1234 variable commented out  forecasts ar18,ar19  Parcov param.unc # the prior cov matrix in unc file format  lambdas 0.1, 1.0, 10,100 #some lambda values  n\_iter\_base -1  n\_iter\_super 4  base\_jacobian pest.jcb  par\_sigma\_range 6  ies\_par\_en par.jcb  \* parameter data external  Par\_hk.csv  Par\_rech.dat sep w missing\_values nan  # observation data split into separate file for each type  \* observation data external  head\_obs.dat sep w missing\_values missing  flux\_obs.csv  additional\_valuable\_obs.csv  \* model command line  model.bat  \* model input external  Model\_input.csv  \* model output external  Model\_output.csv  \* prior information external  Pi.csv |

Figure 4.13 An enhanced PEST control file.

# 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.

## 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.

Note that all members of the PEST++ suite support multithreaded template and instruction file processing, which can be an important consideration for very high-dimensional problems where the number of template and instruction files may number in the hundreds. The number of threads to use for processing template and instruction files is controlled by the *num\_tpl\_ins\_threads* option, which is set to 1 by default. Note the panther workers also use this argument.

## 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 the same member of the PEST++ suite, but acting in the capacity of a local supervising program.

Execution of PESTPP-XXX when acting as a local run supervisor (or “worker”) should be initiated in all folders in which parallel model runs are to be undertaken. Hence as many instances of PESTPP-XXX 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 (referred to as the PETPP-XXX “manager”) 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 a PESTPP-XXX worker establishes communication with the PESTPP-XXX manager 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.) The PESTPP-XXX manager is thus always aware of the workers that it has at its disposal. Furthermore, before it notifies the PESTPP-XXX manager 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 the PESTPP-XXX manager so that the latter is aware whether some of the machines to which it has access are faster than others. Note however that the PESTPP-XXX manager 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 the PESTPP-XXX manager wishes that a model run be carried out, it chooses one of its workers 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 the PESTPP-XXX worker 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, the PESTPP-XXX worker 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, the PESTPP-XXX worker awaits a request from the PESTPP-XXX manager to supervise another model run.

In order to write model input files and read model output files, each instance of the PESTPP-XXX worker 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 the PESTPP-XXX manager are placed in the working folder of each PESTPP-XXX worker, together with files required by the model. As will be discussed shortly, the PESTPP-XXX worker knows of the existence of these template and instruction files because it reads the PEST control pertaining to the current problem as it commences execution.

### 5.3.2 Manager to Worker Communication

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

When the PESTPP-XXX manager 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, the PESTPP-XXX manager 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 the manager 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, the manager 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 as Manager and Worker

When model runs are parallelized, execution of the PESTPP-XXX manager 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 the PESTPP-XXX manager 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, the PESTPP-XXX manager 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 the manager and that worker can take place freely.

Execution of each instance of the PESTPP-XXX worker must be initiated as follows:

pestpp-xxx case /h textstring:port

*case* is the filename base of the PEST control file. *textstring* can be any of the following:

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

If you are running the manager together with multiple worker instances 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 the manager, 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 the PESTPP-XXX manager.

### 5.3.4 Run Management Record File

The PESTPP-XXX manager 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. The workers that execute runs write all information related to communications with the master to the *panther\_agent.rec* file, which is written in the local worker directory.

### 5.3.5 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 previous successful runs), then it will ask another worker to start the same run if one is available. The maximum number of concurrent runs that employ the same set of parameters is set by the *max\_run\_fail()* control variable (this variable is further discussed below). The re-running of a simulation using another worker is done as a precautionary measure. It guards against the possibility that the computer on which the delayed model run is executing 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 run (or runs) is 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 completion of a simulation 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 of 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.

#### panther\_agent\_restart\_on\_error()

In certain Wide Area Network (WAN) environments, manager-worker communications can become “broken”, or “half-open”, where one side does not know that the other has closed the connection. This can happen with TCP/IP connections when there are long periods of no communication, such as during extended periods when the PEST++ manager is busy calculating parameter upgrades or undertaking localization calculations in PESTPP-IES. To help alleviate this, a discrete run manager ping thread is invoked which pings agents at least once every two minutes when the run manager is idle (i.e., when the manager is not communicating with workers as they undertake model runs but is busy doing other intensive calculations). In addition, workers can optionally be restarted in the case that communication errors still occur, instead of terminating or remaining in a half-open state. By default, this option is activated (i.e. it is set to *true)* but can be inactivated using the ++*panther\_agent\_restart\_on\_error(false)* control variable.

#### panther\_agent\_no\_ping\_timeout\_secs()

Related to the above worker restart option, workers can be instructed to terminate (if *panther\_agent\_restart\_on\_error()* is set to *false*) or restart (if *panther\_agent\_restart\_on\_error()* is set to *true*), if no ping message has been received from the run manager in more than a specified time interval. This interval is configurable in seconds via the *panther\_agent\_no\_ping\_timeout\_secs()* control variable, with a default value of 300 (i.e., 5 minutes).

*panther\_agent\_freeze\_on\_fail()*

In some settings, when starting to use PEST++, it can be difficult to debug why runs may be failing in a parallel run environment. This is especially true when workers are on separate physical hosts, which can make monitoring worker progress difficult and when a run fails, the panther run manager will immediate try to schedule another run on that same worker, which will cause the template files to be rewritten (nearly immediately) and any temporary files to be erased, making it nearly impossible to investigate the cause of the run failure. If users want to “slow down the process” so they debug run failures, adding *panther\_agent\_freeze\_on\_fail* as *true* to a (worker) control file will cause a worker to “freeze” on the occurrence of a run failure. This freeze can only be undone by forcing the worker to exit and restarting it, but, nevertheless, freezing a worker when a run failure occurs can be very useful to diagnosing issues related to parallelization of the PEST++ process because it allows direct inspection of all (temporary) files related to the failed run.

*panther\_echo()*

If users are piping the master instance stdout and stderr to a file (through a redirect), then the panther master run summary, which echos to the file with a line return to overwrite the output in place), can fill up these file because the line return character is ignored. In this case, suppling *panther\_echo(false)* will turn off this stdout updating during the run sequence.

*num\_tpl\_ins\_threads*

When using the PEST++ tools for very high-dimensional problems, the time required to process template and/or instruction files can be considerable. To speed things up, the PANTHER orkers can multithread these input and output processing operations. By default, only one thread is used and the number of threads to use is controlled by the *num\_tpl\_ins\_threads* arg. Note that the number of threads used to process template and/or instruction files is set to the minimum of the number files and the value of *num\_tpl\_ins\_threads*.

## 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 “raw” run results. They contain information that assists in parallel run management, and that facilitates restart of an interrupted PEST++ run. These run storage files can be read and processed using pyEMU.

# 6. PESTPP-DA

## 6.1 Introduction

PESTPP-DA is a …

## 6.2 Theory

### 6.2.1 Basic Equations

PESTPP-DA uses the….

### 6.2.2 Batch Data Assimilation with PESTPP-DA

Batch data assimilation with PESTPP-DA is conceptually the same process as used the iterative process in PESTPP-IES. However…

### 6.2.3 Sequential Data Assimilation with PESTPP-DA

The concept of sequential data assimilation…

The concept of assimilation “cycles”…

Evaluting the prior ensemble through cycles

Limiting the range of cycles to assimilate over

State estimation, parameter estimation and joint state-parameter estimation

Model structures and the implied data assimilation problem

Ensemble Kalman filter

### 6.2.4 Iteration and multiple data assimilation

The batch and sequential data assimilation process PESTPP-DA can be wrapped in the multiple data assimilation iterative process of REF…

Inflation factors

### 6.2.12 Running PESTPP-DA

See section 5 of this manual for how to run the tools in the PEST++ suite. As is described in that section, model runs can be undertaken in series or in parallel. In either case, a prematurely terminated PESTPP-DA run can be restarted by supplying the requisite parameter and observation ensemble files.

The version 2 pest control file format is required to specify cycle numbers. If no cycle numbers of found, then PESTPP-DA proceeds with a smoother analysis assuming all parameters and observations are in cycle 0.

If NOPTMAX is set to zero…

If NOPTMAX is set to -1…

### 6.2.13 PESTPP-DA Output Files

The following table summarizes the contents of files that are recorded by PESTPP-DA 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.log* | PESTPP-GLM performance record. This file records the times at which PESTPP-GLM commenced and completed various processing tasks. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Table 6.1. Files recorded by PESTPP-DA.

## 6.4 Summary of PESTPP-DA Control Variables

### 6.4.1 General

This section summarizes variables that control the operation of PESTPP-DA. 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-GLM are listed in table 6.3.

### 6.4.2 Control Variables in the PEST Control File

PESTPP-DA…

### 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-DA.

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Role** |
| *da\_parameter\_ensemble* | text |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

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

# 12. References

Ahlfeld, D.P. and Mulligan, A.E., 2000. Optimal Management of Flow in Groundwater Systems. Vol 1. Academic Press.

Alfonso, M., and Oliver, D.S. 2019. Evaluating prior predictions of production and seismic data. Computational Geosciences.

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.

Chen, Y. and Oliver, D.S., 2016. Localization and regularization for iterative ensemble smoothers. *Comput. Geosci.* DOI 10.1007/s10596-016-9599-7.

Coello, C.A.C., Pulido, G.T., Lechuga, M.S., (2004). Handling multiple objectives with particle swarm optimization. *IEEE Transactions on Evolutionary Computing* 8. doi: <https://doi.org/10.1109/tevc.2004.826067>.

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., White, J. and Welter, D., 2018c PEST and PEST++: An Overview. 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>

Eberhart, R., Kennedy, J., (1995). A new optmizer using particle swarm theory. *Micro Machine and Human Science, Sixth International Symposium on*. doi: <https://doi.org/10.1109/MHS.1995.494215>.

Engelbrecht, A.P., 2007. Computational intelligence: an introduction. John Wiley & Sons.

Evans, M., Moshonov, H. 2006. Checking for prior data conflict. Bayesian Analysis.

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.

Kennedy, J. (1998), The behavior of particles, *Evolutionary Programming VII: Proceedings of the Seventh Annual Conference on Evolutionary Programming*, pp. 581–589.

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.

Luo, X., Bhakta, T. and Naevdal, G., 2018. Correlation-based adaptive localization with applications to ensemble-based 4d seismic history-matching. *SPE Journal, April 2018, 396-427*

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.

Schafer, J. and Strimmer, K. 2005. A shrinkage approach to large scale covarince matrix estimation and implications for functional genomics. Statistical applications in genetics and molecular biology.

Siade, A. J., Rathi, B., Prommer, H., Welter, D., and Doherty, J., (2019). Using heuristic multi-objective optimization for quantifying predictive uncertainty associated with groundwater flow and reactive transport models, *Journal of Hydrology*, 577. doi: <https://doi.org/10.1016/j.jhydrol.2019.123999>.

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.

Welter, D., Doherty, J. and Egan, C., 2019. Manual for the PANTHER Parallel Run Manager. Watermark Numerical Computing, Downloadable from [www.pesthomepage.org](http://www.pesthomepage.org)

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.

## B.4 JCO File

### B.4.1 Introduction

A JCO file is a binary file. It is used by members of the PEST and PEST++ suites to hold a Jacobian matrix. However it can hold any matrix in which rows and columns are labelled. When holding a Jacobian matrix, columns pertain to parameters. In accordance with the old PEST protocol, parameter names are at most 12 characters in length. Rows pertain to observations; in accordance with the old PEST protocol, observation names are at most 20 characters in length.

### B.4.2 Specifications

Specifications are shown in Figure B.4.1

|  |
| --- |
| negncol, nrow 32 bit integers  ncount 32 bit integer  index, value 32 bit integer, 64 bit real  *repeat the above ncount times*  parname 12 bit character  *repeat the above ncol times*  obsname 20 bit character  *repeat the above nrow times* |

Figure B.4.1. Protocol of a JCO file.

Variables cited in figure B.4.1 are as follows:

|  |  |
| --- | --- |
| **Variable name** | **Role** |
| negncol | Negative of the number of columns in the matrix |
| nrow | Number of rows in the matrix |
| index | Index number of matrix element. Indices starts at 1. The index of an element is computed as (icol-1)\*nrow+icol |
| value | The value of the matrix element |
| parname | The name of a matrix column |
| obsname | The name of a matrix row |

If a value is not provided for a particular matrix element in a JCO file, its value is assumed to be zero.

## B.5 JCB File

### B.5.1 Introduction

A JCB file is a binary file. It is used by members of the PEST++ suite to hold a matrix for which rows and columns are labelled (often members of an ensemble). Row and column names can be up to 200 characters in length.

### B.5.2 Specifications

Specifications are shown in Figure B.5.2.

|  |
| --- |
| ncol, nrow 32 bit integers  ncount 32 bit integer  irow, icol, value 32 bit integer, 32 bit integer, 64 bit real  *repeat the above noount times*  colname 200 bit character  *repeat the above ncol times*  rowname 200 bit character  *repeat the above nrow times* |

Figure B.5.2. Protocol of a JCB file.

Variables cited in figure B.4.1 are as follows:

|  |  |
| --- | --- |
| **Variable name** | **Role** |
| ncol | Number of columns in the matrix |
| nrow | Number of rows in the matrix |
| irow | Row number of matrix element; starts at 0. |
| icol | Column number of matrix element; starts at 0. |
| value | The value of the matrix element |
| colname | The name of a matrix column |
| rowname | The name of a matrix row |

If a value is not provided for a particular matrix element in a JCB file, its value is assumed to be zero.

### B.5.3 Distinguishing between a JCO and a JCB File

A PEST++ program recognizes a JCO file from integers that it reads from the first line of this file, regardless of the filename extension. If the first integer that it reads is negative then the file is a JCO file. If it is positive, then it is a JCB file.