Notes on the UNIX version of PEST

John Doherty
Watermark Numerical Computing
January, 2015

1. Introduction

1.1 General

The UNIX version of PEST is supplied as source code, makefiles and example files. Compilation instructions are provided in Section 2 of this document. A few notes on usage of the UNIX version of PEST are provided in Section 3.

1.2 Installation

PEST is provided as a *tar* file (for example *pest13.tar*).

Copy *pestx.tar* to a suitable directory on your machine, for example a directory named */pest*. Then execute the command:-

```
tar –xvf pestx.tar
```

PEST source code files and makefiles will be placed in the directory from which the above command was issued. A number of subdirectories will also be created. Subdirectories *pestex*, *papestex*, *edpestex* and *ppestex* will contain files pertaining to the worked examples described in the PEST manual. Don't forget to compile the models used in these examples before working through them; see Section 2.10 for details.

Another two subdirectories named *itptest* and *vtp* will also be present. Files in these directories pertain to verification and testing procedures used in the qualification of PEST and its utilities for use at U.S. Department of Energy sites. Pertinent documentation can be forwarded on request.

2. Compiling PEST

2.1 Source Code

PEST must be compiled with a FORTRAN 90 compiler.

Most of the source code files for PEST and its utilities possess an extension of "*.F*". Hence, when compiled using the f90 command, they are first preprocessed by the C preprocessor. This preprocessing involves the selection or elimination of code fragments (through `#ifdef/#else/#endif` statements spread throughout these source code files), depending on symbols defined (using the -D compiler option) on the f90 command line. Symbols should be defined which are appropriate to your compiler. As is discussed below, certain groups of symbols have already been preselected; these selections are suitable for use with most UNIX compilers. However differences between compilers, and the idiosyncratic nature of some of them, may require the use of other symbols on some platforms.

A rudimentary C preprocessor named *cppp* has been supplied with PEST. The *cppp* preprocessor supports only a limited number of directives, these being listed in Table 1.

PEST makefiles are such that *cppp* is invoked for source code preprocessing rather than the preprocessor resident on the machine on which PEST is being compiled. This situation can be readily altered if desired through appropriate editing of these makefiles. If it is used, *cppp* must be compiled before all other programs of the PEST suite.

```
#ifdef
#ifndef
#else
#endif
#define
```

Table 1. Directives supported by *cppp*.

2.2 Symbols used with Compiler Directives

A list of symbols appearing in PEST source code files is provided in Table 2.

Symbol	**Use**
AT	Selection of code fragments specific to a version of PEST used with a particular commercial model graphical user interface.
BEO	Selects code for inclusion in BEOPEST.
CAPFILE	Selects code for capitalisation of filenames.
GMS	Selection of code fragments specific to a version of PEST used with a particular commercial model graphical user interface.
FLUSHFILE	Used to access the non-standard (but common) *flush()* subroutine supported by many compilers.
INTEL	Selects code for use with the INTEL compiler in the PC environment.
LAHEY	Used to access certain non-standard features offered by LAHEY compilers.
LF90	Used to access non-standard features offered by the LF90 compiler, particularly in relation to file opening.
MPEST	Selects code for inclusion in a special version of PEST for which parallel run management is undertaken by an external manager.
NO_CMDLINE	Selects code that allows command line arguments to be entered through a prompted string.
PARALLEL	Selects code for inclusion in Parallel PEST and BEOPEST.
PESTMOD	Selects code fragments used in compilation of the PESTMOD module.
SCRIPT	Used for a version of Parallel PEST, run on a CRAY, which implements parallelisation by continuously “topping up” the batch queue.

SLEEP	Selects code that calls the SLEEP function for those compilers that support it (used by Parallel PEST and PSLAVE). Be sure to use this option if it is available as alternative methods of "killing time" are very CPU intensive.
SPACE	Used for optional insertion of a space preceding SENSAN error messages.
SYS_FUNCTION	Used for compatibility with compilers for which the (non-standard) intrinsic subprogram SYSTEM is an integer function rather than a subroutine.
UNICOS	Used for selection of the ISHELL function in place of the SYSTEM function.
UNIX	Selects code used in the UNIX version of PEST.
USE_D_FORMAT	Employed for compatibility with compilers that do not allow the "E" symbol for exponentiation of double precision numbers.

Table 2. Symbols used with compiler directives in PEST source code files.

Note that, in addition to accommodating differences between different compilers and operating systems, symbols and accompanying compiler directives are used for selection of parallelisation code required for the compilation of Parallel PEST and BEOPEST.

2.3 Makefiles

Makefiles used for compilation of PEST and its utilities are listed in Table 3.

Name of file	**Role**
makefile	• Removal of non-essential files after compilation. • Removal of executable programs prior to re-compilation. • Compilation of *cppp.* • Installation of compiled executables.
pest.mak	Generation of the *pest* executable, the *predvar** and *predunc** suites of executable programs, as well as executable versions of the *infstat*, *infstat1*, *wtsenout*, *pnulpar*, *muljcosen*, *identpar*, *supcalc*, *simcase* and *ssstat* utility programs.
ppest.mak	Generation of *ppest* (i.e. Parallel PEST) executable program.
beopest.mak	Generation of *beopest* executable program (TCP/IP version only).
pestutl1.mak	Generation of executable versions of the PEST utility programs *eigproc*, *inschek*, *jacwrit*, *jco2jco*, *jcotrans*, *jcochek*, *par2par*, *paramfix*, *parrep*, *pestchek*, *jcosub*, *pestgen*, *picalc*, *ppause*, *pslave*, *pstop*, *pstopst*, *punpause*, *svdaprep*, *tempchek*, *wtfactor*, *ppd2asc* and *ppd2par*.
pestutl2.mak	Generation of the *parcalc* and *obscalc* executable programs.
pestutl3.mak	Generation of executable versions of the PEST utility programs *dercomb1*,*genlin*, *jco2mat*, *jcoaddz*, *jcocomb*, *jcoorder*, *jcopcat*, *jrow2mat*, *jrow2vec*, *obsrep*, *paramerr*, *pclc2mat*, *pcov2mat*, *pest2vec*, *pestlin*, *prederr*, *prederr1*, *prederr2*, *pwtadj1*, *regerr*, *resproc*, *reswrit*, *scalepar*, *vec2pest*, *veclog*, *prederr3*, *pwtadj2*, *jcodiff*, *regpred*, *addreg1*, *randpar*, *mulpartab*, *comfilnme*, *paramid*, *postjactest*, *genlinpred*, *subreg1*, *phistats*, *lhs2pest*, *pest2lhs*, *parreduce* and *assesspar*.

pestutl4.mak	Generation of executable versions of the PEST utility programs *cov2cor, covcond, mat2srf, matadd, matcolex, matdiag, matdiff, matinvp, matjoinc, matjoind, matjoinr, matorder, matprod, matquad, matrow, matsmul, matspec, matsvd, matsym, mattrans, matxtxi, matxtxix* and *mat2jco*.
pestutl5.mak	Generation of executable versions of the *jactest, obs2obs* and *rdmulres* PEST utility programs and of the *CMAES_P* and *SCEUA_P* global optimisers.
pestutl6.mak	Generation of the *supobsprep, supobspar* and *supobspar1* executable programs.
sensan.mak	Generation of *sensan* and *senschek* executable programs.

Table 3. Makefiles for compilation of PEST and its utilities.

In each of these makefiles, symbols for use by the *cppp* preprocessor are assigned to the DEFINES macro. The F90 and LD macros are assigned the compile and link commands pertinent to the local FORTRAN 90 compiler. Compiler options applicable to code compilation and linkage are assigned to the FFLAGS and LDFLAGS macros respectively.

2.4 Compilation

Before compiling PEST, each of the makefiles listed in Table 3 should be opened, and appropriate compiler commands, compiler options and preprocessor symbols selected. A number of preselections have already been made for different compilers; uncomment the block at the top of the makefile that is most appropriate for you.

Next, if necessary, edit the directory name assigned to the INSTALLDIR symbol in *makefile*. Then compile and install PEST and all of its utility programs using the following sequence of commands:-

```
make cppp
make -f pest.mak all
make clean
make -f ppest.mak all
make clean
make -f pestutl1.mak all
make clean
make -f pestutl2.mak all
make clean
make -f pestutl3.mak all
make clean
make -f pestutl4.mak all
make clean
make -f pestutl5.mak all
make clean
make -f pestutl6.mak all
make clean
make -f sensan.mak all
make clean
make –f beopest.mak all
make clean
```

```
make install
```

Repetition of "make clean" in the above sequence of commands ensures that intermediate preprocessor-generated and compiler-generated files are removed prior to subsequent compilation events. Deletion of these files is required due to re-use of source code files in compiling different executable programs, with different preprocessor symbols selected in each case.

2.5 The USE_D_FORMAT Symbol

You should only define the USE_D_FORMAT symbol during compilation if, when running PEST, you ever receive an error message such as:

```
Error writing parameter to model input file:
Internal error type 1.
```

However before selecting this directive you should test whether it has any chance of working for your compiler by running the program *d_test* provided with PEST. This is most easily done using the command:

```
make d_test
```

This command will compile *d_test.f* and then run it. The screen output provided by *d_test* will inform you if it is appropriate to define the USE_D_FORMAT symbol.

2.6 Making PEST Run Faster: Using Optimiser Switches at Compilation

Compiler optimisation capabilities can be activated by adding the pertinent switches to the FFLAGS symbol in PEST makefiles. The compiler should then generate executables for PEST and its utilities which run faster. It should be noted, however, that the rate of execution of the model which is run by PEST mostly exerts a far greater influence on overall parameter estimation time than PEST's rate of execution.

Some compilers introduce "phantom" errors into PEST execution when PEST is compiled with the optimisation switches set. Such errors may cause PEST to terminate execution with a run-time compiler-generated error which, in reality, is not an error at all. Hence, if you compile PEST with your compiler's optimiser switches set and PEST does not successfully complete the parameter estimation process, re-compile it with no optimisation and see if the error still occurs. If it does not, then it may be that there is a problem with your compiler's optimiser.

2.7 Making PEST Run Faster: Decreasing the Model File Width Specification

There is a simple change that you can make to the PEST source code that may induce a dramatic reduction in execution time (more dramatic than code optimisation). PEST has been programmed to write model input files and read model output files of up to 2000 characters in width. If none of your model input or output files is this wide, you can decrease this maximum width specification. This will save PEST a great deal of work in checking empty spaces in character strings for entries which will never be present.

For example, if you know that your model input and output files will never be greater than 300 characters in width, open *runpest.F* and search for the line:

```
C --- File width
```

Change the ensuing line from:

```
CHARACTER*2000 BUF
```

to

```
CHARACTER*300 BUF
```

Then, in *pestdata.F*, alter the line:-

```
CHARACTER(LEN=2000)     :: CLINE=' '     ! Work string
```

to

```
CHARACTER(LEN=300)      :: CLINE=' '     ! Work string
```

Then recompile PEST. This is certain to result in a considerable decrease in execution time where there are many observations. Note, however, that **you must not declare a string length of less than 300 characters**, as both CLINE and BUF have other uses within PEST for which a length of 300 characters is necessary.

Though similar changes can be made to files *pestchek.F*, *tempchek.F* and *inschek.F*, it is advised that these files not be altered as such alterations may interfere with the normal checking operations carried out by these programs.

2.8 Maximum Array Sizes for TEMPCHEK, INSCHEK and PESTGEN

If any of TEMPCHEK, INSCHEK or PESTGEN complain that more parameters or observations are cited in the current case than they can handle, this is easily fixed.

In each of *tempchek.F* and *inschek.F* only one variable needs to be adjusted to rectify this problem, viz. ZNPAR (maximum number of parameters) in the former case and ZNOBS (maximum number of observations) in the latter case. For *pestgen.F*, both ZNPAR and ZNOBS need to be adjusted (once each, in sequential source code lines). In all three cases the relevant source code line is situated near the beginning of the file, immediately following a comment line such as:

```
C -- Number of parameters
```

This problem should arise in no other PEST programs.

2.9 PATH

Some of the programs of the PEST suite run other members of that suite to undertake certain tasks. For example, when undertaking SVD-assisted parameter estimation, PEST uses the PICALC and PARCALC executables to manipulate “base parameters” while PEST (or Parallel PEST) estimates super parameters. SENSAN uses TEMPCHEK and INSCHEK to write model input files and read model output files respectively. It is thus important that all PEST and PEST utility executable programs reside in a directory cited in the PATH environment variable.

2.10 Compiling the Example and Test Cases

Program TWOLINE in both the *pestex* and *papestex* subdirectories must be compiled before you can run the example case documented in the PEST manual. There are no preprocessor symbols to be selected as the TWOLINE source code does not need to be preprocessed. Hence it can be compiled using, for example, the command:-

```
f90 -o twoline twoline.f
```

Similarly, program POLYMOD in the *edpestex* subdirectory (see "Model-Calculated Derivatives" in the PEST manual) should be compiled using the command:-

```
f90 -o polymod polymod.f
```

Make sure that file *model.bat* in the *edpestex* subdirectory is designated as an executable file. This can be achieved using the command:-

```
chmod u+x model.bat
```

Program A_MODEL in the *ppestex* subdirectory (see "Parallel PEST" in the PEST manual) should be compiled using the command:-

```
f90 -o a_model a_model.f
```

3. Using UNIX PEST

3.1 General

Use of the UNIX version of PEST is very similar to that of the PC version of PEST as described in the PEST manual. However there are a few differences, as set out below.

3.2 Case Sensitivity

On a PC platform filenames are case-insensitive. In the UNIX version, however, all filenames are case sensitive. Hence all references to template files, instruction files and model input/output files in the PEST control file must use the correct case. The same applies to the model command line.

It should be noted, however, that case sensitivity does not extend to parameter and observation names, nor to the instructions contained in PEST instruction files.

3.3 Run-Time Inspection of PEST Output Files

The run record file, the parameter sensitivity file and the run management file produced by PEST can be viewed by opening other UNIX shell windows and inspecting them from there. Unfortunately, however, the latest information written to these files by PEST may not always be accessible to viewing programs run from these other windows. This is because data written to these output files is first written to a buffer, where it accumulates until the buffer is full. The buffer is then automatically flushed, at which stage the information is actually recorded on the pertinent output file. To overcome this problem, code can be included within PEST by which this buffer can be flushed on a regular basis whether it is full or not. If this is done the optimisation history can be inspected at any time right up until the moment of viewing the respective PEST output file. This is implemented using the *flush()* subroutine which is invoked using the FLUSHFILE symbol during PEST source code preprocessing. Unfortunately, however, this subroutine is not supported by all compilers.

If desired, one or a number of windows can be dedicated solely to inspection of PEST output files. If, at any stage of PEST execution, the user opens a new window, transfers to the PEST working directory within this window, and types the command:-

```
tail -f file
```

where *file* is any of *case.rec*, *case.sen*, *case.svd*, *case.rmr* etc (where *case* is the current casename), the last few lines of the pertinent file will be continuously displayed and updated in that window. Alternatively, these files can be inspected in their entirety by running `cat | more`, `pg`, `vi` or any other appropriate UNIX text viewing utility.

The UNIX version of PEST generates a file called *jacob.runs* which records the number of model runs that have been completed in calculation of the Jacobian matrix for the current optimisation iteration. This file can be viewed in similar fashion to the run record file; it is a particularly good candidate for viewing using the `tail -f` command, as a continuous record is then available of how many runs have been completed during the current optimisation iteration. Note that up-to-date viewing of *jacob.runs* depends on use of the *flush()* subroutine discussed above.

3.4 Restarting with the "/s" and "/d" switches

As is explained in the PEST manual, if Parallel PEST or BEOPEST is restarted with the "/s" switch or serial PEST is restarted with the "/d" switch, it will re-commence execution at the exact model run at which its execution was previously terminated. However this is only possible if PEST compilation takes place with the FLUSHFILE symbol selected for source code preprocessing. As explained above, this can only happen if the compiler supports an intrinsic *flush()* function.

3.5 Command-Line Arguments

No standard FORTRAN functionality is available for reading command-line arguments. Hence, whether PEST and its utilities are run on a PC or on a UNIX platform, access to command-line arguments must be made using non-standard, compiler-specific functions. For just about all UNIX compilers the *getarg()* function can be used for this purpose. However, just in case this function is not available, all programs of the PEST suite except SENSAN can be compiled with a preprocessing option (invoked by the NO_CMDLINE symbol of Table 2) that allows the command line to be read as a character string using standard FORTRAN data input functionality. Executable versions of PEST and its utilities which are compiled using this option will prompt the user for command line arguments as soon as they commence execution. Thus, for example, instead of running INSCHEK with the command:

```
inschek file.ins modfile.out
```

(where *file.ins* is an instruction file and *modfile.out* is a model output file), it should be run using the command:

```
inschek
```

Then, as soon as it commences execution, it prompts:

```
Enter command line arguments >
```

to which you should respond:

```
file.ins modfile.out
```

This option is not supported with SENSAN because it must run each of TEMPCHEK and INSCHEK in the course of its execution; it is programmed to do so using command line arguments.

3.6 Run-Time Differences

Apart from the differences outlined above, every effort has been made to make the use of PEST identical across platforms and operating systems. Nevertheless some differences may remain. For example the optimised objective function for a particular case may be slightly different for versions of PEST generated by different compilers. It is also possible that PEST will need to perform an extra optimisation iteration or two for parameter fine tuning on some machines, and that parameter estimates may differ slightly from platform to platform. These discrepancies result from differences in the way processors manipulate real numbers, and on the way such numbers are stored on different platforms. However such differences are expected to be minor and will not impair PEST's ability to operate correctly on any particular system.

Other slight variations may be noticed in the operation of some of the PEST utilities. For example, one of the compilers that we have encountered reads the string 2.0* as the number "2" instead of generating an error condition based on the fact that the string 2.0* is not, in fact, a number. This has repercussions when PESTCHEK (when compiled with that compiler) checks the "prior information" section of a PEST control file in which the user has forgotten to separate the "*" character from the parameter value multiplier, as instructed in Section 4 of the PEST manual. In this case PESTCHEK will read 2.0* as "2" and will report an error when it reads the ensuing parameter name while expecting to find the "*" character. The resulting error message will state that the parameter name is an illegal prior information item when, in fact, the previous item, (viz. "2.0*") was the illegal item.

3.7 Using SVD-Assist

The PEST manual describes SVD-assisted parameter estimation; this is a device, unique to PEST, by which optimisation efficiency and stability can be increased enormously in highly parameterised contexts. When using this methodology, preparation of a PEST control file based on "super parameters" is undertaken automatically using the SVDAPREP utility. SVDAPREP obtains the information that it needs for generation of this new control file from the PEST control file pertaining to an existing regularised parameter estimation problem that uses "base" parameters, these being the native parameters for that particular problem.

When PEST undertakes SVD-assisted parameter estimation on the basis of super parameters, the "model" must be a batch file on the PC platform, or a script file on the UNIX platform. This is because the model must include commands to run the PEST utilities PARCALC and PICALC to generate model input files and calculate prior information respectively, using base parameter values (even though the PEST control file only references super parameters). It is thus essential that the *parcalc* and *picalc* executable programs reside in a directory cited in the PATH environment variable.

Part of SVDAPREP's job in creating input files for a super-parameter PEST run is to modify the existing batch or script file to include the new commands. (The modified file is actually written as a new file; the existing batch or script file is not destroyed.) Hence, even for the base parameter control file, the "model" must actually be a batch or script file; it must not be an executable program. In the PC environment, the difference between the two is easily detected as batch files possess an extension of "*.bat*" and executable programs possess an extension of "*.exe*". No such protocols are followed in the UNIX system. So while SVDAPREP can detect the difference on a PC, it cannot on a UNIX system. If the existing

model command is an executable program rather than a script file, then the new script file named *svdabatch.bat* created by SVDAPREP will be comprised of nonsense and will not run. It is the user's responsibility, therefore, to ensure that the original base PEST control file uses a model command line which runs a script file rather than an executable program to prevent this occurrence. (Note that SVDAPREP uses the `chmod` command to give file *svdabatch.bat* which it creates executable status.)

3.8 BEOPEST

BEOPEST can use MPI or TCP/IP for communication between master and slaves. The version of BEOPEST that is compiled using the *beopest.mak* makefile discussed above, can use only the TCP/IP protocol.