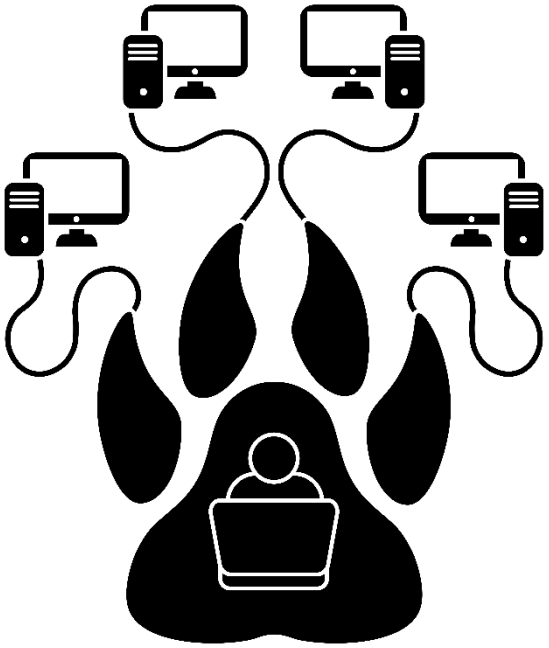
PANTHER

Parallel Model Run Manager



PANTHER

Dave, Welter, John Doherty and Jeremy White

November 2018

Acknowledgements

The PANTHER run manager was developed from the YAMR run manager. The latter is used by programs of the PEST++ suite. Development of early versions of the PEST++ suite was funded by the United States Geological Survey.

Funding for the development of PANTHER was provided by the Queensland Water Modelling Network (QWMN). For more details of QWMN see:

https://www.des.qld.gov.au/science/government/science-division/water-modelling-network/

Thanks to Max Ramgraber who helped with the picture for the front cover.

Table of Contents

1. Introduction

1.1 Adding Value to Models

Models which support decisions pertaining to environmental management cannot support these decisions on their own. They benefit the decision-making process best when they are used in partnership with software packages that enable the following (and other) numerical tasks to be undertaken:

- local and global sensitivity analysis;
- history-matching;
- parameter and predictive uncertainty analysis;
- management outcome optimization;
- optimization under uncertainty.

These tasks are not easy. Algorithms and software for their implementation are constantly undergoing development and refinement. Regardless of how they are done, they require that many model runs be undertaken. Where model runs are long (as they often are), the time required for their completion may be prohibitive. These tasks also require flexible exchange of information between a model and the packages which supervise model runs in order to implement them. Provision of this interface is difficult if programming, as well as access to a model's source code, is required.

Because these factors make implementation of the above tasks difficult in everyday modelling practice, they are often done manually. Model-support for environmental management does not therefore realize its potential.

With the adoption of two relatively simple strategies, barriers to computer-based implementation of the above tasks can be removed. These are:

- parallelization of model runs;
- provision of a model-independent, non-intrusive interface between a model partner package and the model itself.

A non-intrusive interface between computer software which supervises model runs and the model itself is one that allows communication between these two packages to take place through the model's own input and output files. This eliminates the need for access to a model's source code, and for re-compilation of the model. It also allows definition of "a model" to be as flexible as a decision-support context demands. In particular, a model can be comprised of a batch or script file that includes not just a simulator, but also pre- and post-processors for that simulator that all run sequentially. A simulator pre-processor may manipulate parameters and/or decision variables, promulgating flexibility in their definition and application. A simulator post-processor may interpolate gridded model outputs to the sites and times at which field measurements were made, and/or evaluate the costs associated with a model-calculated management outcome. (For simplicity of expression, the term "parameter" will be used to describe both of these types of simulator input data in the present document.) Inclusion of commands to run one or more simulators (or incidences of the same simulator) in a batch file may support the simultaneous use of multiple simulators in the above tasks. This may be useful, for example, where a water flow model provides the flow

field for a water quality model, or where multiple land use models are calibrated simultaneously so that parameter regionalization relationships can provide "intelligent regularisation" for an inherently ill-posed inverse problem.

Large increases in the efficiency of model-value-adding tasks can be realized through model run parallelization. This can be accomplished with little difficulty in model computing environments. The benefits of running many instances of a model in parallel are obvious. Modellers have many options available to them for model run parallelization. Modern computers have multiple CPU's. Most modellers work in an office environment where they have access to multiple computers. Many modellers have access to a high-performance computing cluster. Computing clouds are available to all modellers; as Hunt et al (2010) point out, this has the potential to revolutionize modelling. However, as is stated above, these efficiencies can only be realized if ready access is available to non-intrusive model-partner software that can parallelize model runs across many computing nodes.

The present document comprises a manual for a public domain, nonintrusive, parallel run manager. Development of this manager was spawned by that of the PEST suite (Doherty, 2018a; 2018b). It is at the heart of the PEST++ suite of software (White et al, 2018).

1.2 The PANTHER Run Manager

"PANTHER" is a loose acronym for "parallel, non-intrusive handling of model runs in environmental management". It is a library to which a programmer can link his/her own software. PANTHER manages non-intrusive communication between a model and any program that adds value to that model by running it many times in support of any of the tasks listed above, or any other task which requires that a model be run multiple times. These model runs can be undertaken in serial or in parallel.

Run management functionality provided by PANTHER is accessed through a series of function calls. At the time of writing, programs written in FORTRAN, C, C++ and Anaconda Python can access these functions. They relieve a programmer from having to manage parallelized, non-intrusive run management him/herself. His/her software can thus concentrate on implementing the numerical task that it is designed to perform.

The PANTHER run manager is an advanced version of the YAMR run manager. YAMR was originally developed for use by the PEST++ suite of programs; see Welter et al (2015). It employs the same non-intrusive model interface as do programs of the PEST suite. That is, it uses template files to write model input files, and instruction files to read model output files. It employs the manager/worker concept to parallelize model runs; a manager communicates with workers using the TCP/IP protocol.

Security was a major consideration in the design of PANTHER. Its design is such that interception of communication between a manager and its workers cannot allow a hostile agent to damage a host machine or network. The worst that can happen is that a model may be provided with a nonsensical set of parameter values, or that the manager may be provided with a nonsensical set of model output values. The most likely outcome of the first of these occurrences is model run failure; the most likely outcome of the second of these occurrences is an impossibly high objective function calculated by the PANTHER calling program. Both of these outcomes are easily recognized; neither is damaging to the system on which either the manager or worker is running.

1.3 Setting the Parallelization Context

Execution of PANTHER workers must be initiated on all machines on which parallelized model instances are run. A newly initiated worker announces its presence to the manager through the TCP/IP communications port that is opened by the manager. It then awaits orders from the manager, initializing a model run when asked to do so. The worker instructs the local operating system to run the model after communicating with the model through the latter's own input and output files. The manager informs each worker of the numbers that it must write to pertinent model input files on each occasion that it commissions a model run. On completion of the model run, the worker transmits numbers which it reads from pertinent model output files back to the manager using the same TCP/IP communications channel.

The user of a program which employs PANTHER to manage parallel model runs must take responsibility for starting up all workers him/herself. This can be done manually, or it can be automated; see, for example, Schumacher et al, (2017). Worker instances that reside on the cloud can be initiated manually, or using the application programming interface provided by the cloud platform.

Where PANTHER manages model runs over an office or institutional network, freely-available, high throughput computing (HTC) software such as HTCondor can be used to optimize PANTHER's use of the resources to which it has access. See Fienen and Hunt (2015) for further details.

2. Manager and Workers

2.1 Parallel and Serial Run Managers

While the PANTHER run manager was designed to supervise parallel model runs, PANTHER users also have access to a serial run manager. The serial run manager does not require a worker. It runs the model on the same machine as the calling program by issuing a call to the local operating system.

Once the serial or parallel run manager has been initiated by the PANTHER calling program through a pertinent function call, functions through which model runs are queued, initiated or queried are the same for both of them. Hence the serial and parallel run managers can be used interchangeably. Use of the serial run manager can be convenient where model run times are short; it can be used for testing methodologies prior to engaging in parallel model runs on one or multiple machines.

A programmer who links his/her code to the PANTHER run manager does not need to worry about issuing the command to run a model. Nor does his/her code need to address issues pertaining to non-intrusive communication with a model. Whether he/she employs the serial or parallel run manager, these tasks are carried out automatically by the pertinent run manager.

PESTPP-WRK is a generic worker for the PANTHER parallel run manager. Where model runs are undertaken in parallel, PESTPP-WRK must be started in each folder from which model runs are initiated. Often this folder will contain all input files required by the model; often the model will write all of its output files to this same folder. Other incidences of PESTPP-WRK can be initiated in other folders on the same computer. Use of separate folders ensures that input and output files pertaining to different incidences of the model are not overwritten by each other.

2.2 Running the Model

PESTPP-WRK runs the model by issuing a “system command”. This command is the same as that which a user would employ if he/she were running the model from a command-line window. As is discussed in the next section of this document, this command is provided to PESTPP-WRK through its input file.

Use of a system command to run the model implies that the model exists as a discrete executable program or as a batch file which cites a number of discrete executable programs. A copy of this single or multiple executable program(s) must reside in the working folder of each instance of PESTPP-WRK (or in that of the manager if serial run management is employed). Alternatively, they must reside in a folder which is cited in the PATH environment variable of each machine on which one or a number of workers is started.

Note that use of a system command to run a model precludes the use of models which are encapsulated in DLLs or static libraries, as these cannot be run in this fashion.

Before it runs a model, PESTPP-WRK records on pertinent model input files the values of parameters that it would like the model to use on that particular run. After the model has run to completion, PESTPP-WRK reads from model output files model-calculated numbers in

which it is interested. The means through which it accomplishes both of these tasks is described in the next chapter of this manual. The means through which it is informed by the manager of the values of parameters that it must write to model input files, and the means through which it informs the manager of the values of numbers that it reads from model output files, is described next.

2.3 Manager-to-Worker Communication

The PANTHER run manager and its PESTPP-WRK worker communicate with each other using the TCP/IP communications protocol. Where PESTPP-WRK resides on a different machine from that of the manager, communication of this type may be precluded by network firewalls. However this is generally not the case in office networks. And it is absolutely not the case where PANTHER and PESTPP-WRK are running on the same machine. (Where a firewall prevents machine-to-machine TCP/IP communication, this is easily rectified by the network manager.)

When initiated by its calling program, the PANTHER run manager opens up a TCP/IP communications port. This port is identified by its number; this number must be supplied by the calling program. The calling program may ask the user for this number. (This is the case for members of the PEST++ suite). It is wise to choose a high-valued port number (4000 or above) to avoid conflicts with other programs that are engaged in TCP/IP communication. Once it has been initiated, the PANTHER run manager listens on this port for communications from workers.

Each instance of PESTPP-WRK must be started using the following command:

```
pestpp-wrk infile /h hostname:nnnn
```

In the above command, *infile* is the name of a PESTPP-WRK input file. Specifications for this file are provided in section 3.4 of this document. The text *hostname* must be replaced by the IP address (version 4 or version 6) or the name of the machine on which the PANTHER manager is running. The string *nnnn* must be replaced by the TCP/IP port number that the PANTHER manager has opened. As soon as its execution is initiated, PESTPP-WRK tries to contact the manager. Once it has established communication with the manager, it and the manager exchange information. During this exchange, PESTPP-WRK informs the manager of the speed of the computer on which it is running. (It establishes the speed of its host computer by monitoring the time required to undertake some numerical calculations.) It then awaits a model run order from the manager.

It is of interest to note that the PANTHER run manager has no way of knowing the machine on which any of its workers is running. It knows of the existence of each worker because the worker has contacted it through the TCP/IP port which the opened. However it knows nothing about each worker other than this. Note also that it does not matter whether any or all instances of PESTPP-WRK commence execution before or after the PANTHER run manager; if an instance of PESTPP-WRK is initiated prior to the manager, it repeats its attempts to establish communication with the manager until it is successful. Nor does it matter if an instance of PESTPP-WRK is lost while the manager remains alive. The PANTHER run manager simply continues to manage model runs without it; if necessary, it re-assigns an uncompleted model run that was previously being supervised by the lost worker to another PESTPP-WRK worker. A worker can disappear for a while and then reappear

(provided it is restarted using the above command). PANTHER assigns model runs to a worker as long as the TCP/IP communications channel between it and the worker remains open.

When the PANTHER run manager requires that a model run be carried out, it first chooses a worker to supervise that run. Normally this will be the fastest available worker. Once it has chosen a suitable worker, it then transmits to that worker the values of parameters that it would like the model to use on this occasion of its execution. It also informs the worker of the command that it must use to run the model on this occasion. As will be discussed below, different commands can be used to run the model on different occasions. It is important to note, however, that the command is conveyed as a number rather than as a text string. This forestalls damage incurred by a hostile interception of TCP/IP manager-to-worker communications wherein the command to run the model is replaced by a command to do something more sinister. PESTPP-WRK reads the actual model command to which this number pertains from its input file.

On receiving parameter values from the PANTHER run manager, PESTPP-WRK writes these values to pertinent locations on model input files (see below). It then issues the system command to run the model. Once the model has run to completion PESTPP-WRK reads from model output files the numbers in which it is interested (also see below). Then, using the TCP/IP communications channel, it informs the manager that the model run is complete and sends to the manager the numbers that it has read from model output files. (These are referred to as “observations” in PEST parlance.)

2.4 Parameters and Observations

The above protocol for exchange of information between the manager and its workers assumes that each has been independently informed of the following:

- the number and names of parameters whose values are to be written to model input files;
- the number and names of observations whose values are to be read from model output files;
- the number and names of command(s) used to run the model.

Immediately following initiation of communications between a worker and its manager, these programs establish compatibility of the information that they have separately received. Not every detail is checked, however; for example, model command line texts are not compared as a security measure.

As has been mentioned, each PESTPP-WRK worker must be provided with an input file containing the above information. Presumably, a program which calls the PANTHER run manager reads this information from its own input file. This information is then provided to the PANTHER run manager through pertinent function calls from that program.

3. Non-Intrusive Model Interface

3.1 General

As has already been discussed, PESTPP-WRK (the PANTHER universal worker) communicates with a model through the model's own input and output files. It is important to note that these files must be ASCII (i.e. text) files. This type of non-intrusive interaction with the model is implemented using template and instruction files; it is identical to that employed by programs of the PEST and PEST++ suites.

Template and instruction files are now briefly described; refer to documentation of PEST and PEST++ (Doherty, 2018a and White et al, 2018) for full details. Template and instruction files are easily written using a text editor. However, where many numbers are read from model output files it may be more convenient to write an instruction file using a computer program. Utility programs supplied with PEST (for example the PEST ground and surface water utility suites), and PyEMU (White et al, 2016) provide this functionality.

As was described in the previous section, each instance of PESTPP-WRK must be run from its own working folder. This working folder will often contain a complete set of model input files – both those which contain parameters and those which do not. While this leads to duplication of model input files which do not contain parameters (and hence which are not model-run-specific), it does make worker setup easier. A modeller normally sets up a parameter estimation or optimization process such that it can operate in a single working folder on his/her own computer. Once he/she is assured that everything is working properly (possibly by carrying out a number of model runs using the serial run manager), the modeller can then copy the entire contents of this folder to each worker folder, regardless of the computer in which the worker will operate. Each such folder should include all template and instruction files used by PESTPP-WRK for communication with the model.

For simplicity of the discussion that follows, numbers that are written to model input files are denoted as "parameters". This is despite the fact that they may represent decision variables or other entities that link a model to the real world. Similarly, numbers that are read from model output files are referred to as "observations". This is despite the fact that they may represent management-pertinent model predictions or other model outputs that are pertinent to environmental management. The nomenclature of "parameters" and "observations" is in accordance with that used in PEST and PEST++ documentation.

3.2 Template Files

Before it undertakes a model run, PESTPP-WRK alters certain numbers on model input files (i.e. the values of parameters) so that these numbers are in accordance which those that the manager program wishes the model to employ on that particular run. The locations where these numbers reside on model input files are identified on user-prepared templates of these files. One such template file is required for each model input file on which numbers requiring adjustment reside.

Each parameter is endowed with a name. At the time of writing, these names are restricted to a maximum of 200 characters in length. (This is in contrast to the PEST protocol which limits parameter names to 12 characters in length.) The locations on model input files to which the

current values of parameters are written are referred to as "parameter spaces" in templates of these files. The beginning and end of each of these spaces is defined by a special character known as a "parameter delimiter". These delimiters surround the name of the parameter; hence a program that uses the template file (e.g. PESTPP-WRK) knows the parameter to which the space belongs. In writing a model input file on the basis of a template file, PESTPP-WRK replaces the space between and including the parameter delimiters by the current value of the named parameter. In writing this value, PESTPP-WRK employs as many significant figures as the width of the parameter space allows.

3.3 Instruction Files

Model output files are read using instruction files. An instruction file must be provided for ever model output file from which at least one number must be read. Instructions provide a means of navigation in a forward journey through a model output file. Instructions contained in an instruction file inform the program which uses them to peruse a model output file by counting line numbers or looking for fragments of text, or a combination of the two. The instruction to read a particular number from a particular line of a model output file includes the name of the observation to which the number pertains. At the time of writing, observation names are 200 characters or less in length. (This is in contrast to the PEST protocol which limits observation names to 20 characters in length.) Numbers can be read according to the character positions that they occupy on a particular line of a model output file, or according to the text which they follow on a particular line.

3.4 PESTPP-WRK Input File

3.4.1 General

The name of a PESTPP-WRK input file must be supplied on its command line whenever it is run. PESTPP-WRK reads this file immediately on commencement of its execution.

Figure 3.1 exemplifies a PESTPP-WRK input file. Some additional specifications for this type of file are provided in the following chapter of this manual. There it is explained that an additional section can optionally be added to this file to support file transfer between manager and workers.

```
* model command line
model_command_line_1
model_command_line_2                    # optional comment line
* model input
template_file_1  model_input_file_1
template_file_2  model_input_file_2
etc
* model output
instruction_file_1 model_output_file_1
instruction_file_2 model_output_file_2
etc
# another optional comment
* any text
```

Figure 3.1 An example of a PESTPP-WRK input file.

A PESTPP-WRK input file is divided into sections. Each section begins with the "*" character followed by a space, followed by header text. The (case-insensitive) text comprising a header must be one of those appearing in figures 3.1 and 4.1. Each section ends with a line that begins with the "*" character; this line may be the header for the next section.

Blank lines can appear anywhere within a section. The same applies to comment lines. A comment line begins with the "#" character. Comments can also be placed following valid data items on any line of a PESTPP-WRK input file; these too commence with the "#" character. Note however that a "#" character is not denoted as starting a comment if it occurs between quotes or is not preceded by whitespace or the beginning of a line. This allows for the inclusion of this character in the name of a file.

Sections within a PESTPP-WRK input file can be provided in any order. They do not need to be contiguous; unrelated data can be recorded between sections. Similarly, other information can be recorded before the first section and following the last section. It is thus apparent that a PEST control file can serve as an input file for PESTPP-WRK.

Filenames appearing in the PESTPP-WRK input file can optionally be enclosed by double quotes. If a filename contains a space, then the use of quotes is mandatory.

The sections appearing in a PESTPP-WRK input file will now be described in detail. The specifications of two further sections are provided in the next chapter of this manual.

3.4.2 Model Command Line Section

The commands to run the model must be listed in this section, with each command recorded on its own line. If the name of an executable program or a batch/script file provided as a model command contains a space, then this name must be enclosed in quotes. If necessary, command line switches and command line arguments can be included following the command to run a model.

On most occasions, only one command will be provided to run the model. However, there may be occasions where different commands are used to run the model during different phases of an inversion, optimization or other process that is being implemented by the PANTHER calling program. In cases such as this, the order in which commands are listed in the PESTPP-WRK input file is important. When PANTHER informs PESTPP-WRK what command it must use to run the model, it references this command by index number, starting from 1 and proceeding in order of appearance in the "model command line" section of its input file. (This is a security measure.) Hence the PANTHER calling program must be supplied with the same commands in the same order.

3.4.3 Model Input Section

Each line of the "model input" section of a PESTPP-WRK input file must have two entries. The first is the name of a template file, while the second is the name of the model input file with which the template file is associated. For each of these paired files, PESTPP-WRK uses the template file to write a model input file in which parameter values are those that PANTHER wishes the model to use for that particular model run. The names of parameters that are cited in these template files must be in accordance with those supplied to the PANTHER run manager by its calling program.

Note that the same template file can be used to write multiple model input files. However multiple template files cannot be associated with a single model input file. Note also that the same parameter can appear on multiple template files.

Filenames appearing in the “model input” section of a PESTPP-WRK input file can be preceded by paths if desired (full or relative). However there is no need to prefix a file with its path if the file resides in the PESTPP-WRK working folder.

3.4.4 Model Output Section

Each line of the “model output” section of a PESTPP-WRK input file must have two entries. The first is the name of an instruction file, while the second is the name of the model output with which it is associated. For each pair of these files, PESTPP-WRK uses instructions contained in the instruction file to read model-generated observations from the associated model output file. The number and names of observations that are cited in all instruction files appearing in the “model output” section of a PESTPP-WRK input file must be in accordance with those supplied to the PANTHER run manager by its calling program.

Note that a single model output file can be read by multiple instruction files. However a single instruction file cannot be used to read more than one model output file. Note also that the name of an observation cannot appear in more than one place in a single instruction file, nor appear in more than one instruction file.

Filenames appearing in the “model output” section of a PESTPP-WRK input file can be prefixed by paths if desired (full or relative). However there is no need prefix a file with its path if the file resides in the PESTPP-WRK working folder.

4. Some Other Run Management Details

4.1 Management Efficiency

When PESTPP-WRK commences execution, it tests the speed of the computer on which it is running before initiating communications with the PANTHER manager. It informs the manager of this speed. Before it commissions any model runs, the PANTHER manager is thus aware of the relative speeds of the workers to which it has access.

As model runs are completed, the PANTHER manager keeps a record of model execution times. It can thus update its ranking of relative worker speeds. (These may change over time if the computer on which a worker is running has been given other tasks to do.) Whenever it allocates a model run to a worker, PANTHER chooses the fastest available worker for this task, using the latest information that it has at its disposal for differentiating between relative worker speeds.

4.2 Model Run Failure

Once a model run is complete, PESTPP-WRK attempts to read the model's output files using the instruction set with which it was provided. If one or more of these files is absent, or if PESTPP-WRK cannot read one of them, it assumes that the model has failed to run to completion, or cannot calculate the numbers that it was asked to calculate. It informs the PANTHER run manager of this. PANTHER's next actions are based on settings provided to it by its calling program.

When the PANTHER run manager is initiated, it must receive the value of a variable named N_MAX_FAIL. This variable sets the number of times that PANTHER must attempt to re-run a failed model run. If N_MAX_FAIL is set to 1 or greater, PANTHER attempts to re-run a failed model using a different worker from that on which model run failure was previously experienced. If N_MAX_FAIL is set to zero, then no attempt is made to repeat a failed model run. It is thereby assumed that model run failure is an outcome of the parameters with which the model was provided and is not an outcome of local conditions on the machine on which the model was run (for example a missing file).

Note that a similar protocol is adopted for fun failure under serial run management. However repetition of a failed model run is likely to accomplish little in this environment.

4.3 Overdue Model Runs

When a calling program initializes the PANTHER run manager, it provides the value of a variable named OVERDUE_RESCHED_FAC. The number assigned to this variable must be greater than 1.0. It is a factor that is applied to the average of recent model run times.

Some models may take offence with some sets of parameters with which they are provided – particularly models which solve large matrix equations using an iterative solver. A particular set of parameters may slow a model's execution. On other occasions a model may run slowly because the machine on which it is running has been given other tasks by other users. PANTHER cannot distinguish between these two causations. However if it has spare worker capacity, it can act on the premise that local computing conditions are the problem; it can re-assign the slow model run to an unoccupied worker on another machine.

If PANTHER notices that a model has been running for more than OVERDUE_RESCHED_FAC times the recent average of model run times, it asks another worker to undertake the same model run provided the following conditions are met:

- a worker is available to do the job;
- no other jobs are pending in the run queue.

Following initiation of the second model run, two model runs are being undertaken using the same set of parameters. As soon as the first of these is completed and PANTHER has received model outcomes from the supervising worker, it instructs the worker that is supervising the other model run to terminate execution of the model.

There may be some sets of parameters for which model solution convergence is never achieved, or is achieved so slowly as to render the model results worthless. As it initializes the PANTHER run manager, a calling program must supply a value for the OVERDUE_GIVEUP_TIME variable. If PANTHER detects that a model has been running for a time that exceeds this threshold, it instructs the worker that is supervising that model run to terminate it. The worker is then free to supervise other model runs. However if there are no runs pending in the run queue, PANTHER lets the run continue, on the basis that its continuation is not causing any harm.

The serial run manager which can be used as an alternative to the PANTHER parallel run manager handles overdue model runs in a similar fashion to PANTHER.

4.4 Model Run Termination

Two circumstances which precipitate model run termination have already been discussed. These both pertain to overdue model runs. A third circumstance is caller-initiated model run termination. The program which uses the PANTHER parallel run manager (or the serial run manager that can be used in its place) can instruct the run manager to terminate a particular model run if, for example, it is a forward-looking optimization algorithm which undertake pre-emptive, parallelized model runs in order to keep computing nodes busy while awaiting the results of other model runs.

4.5 Transient Workers

A PESTPP-WRK worker can be taken off-line at any time. If this happens while it is supervising a model run, PANTHER re-assigns that run to an alternative worker. Execution of the worker can be recommenced at a later time. It is re-started using the same command as that which was used to start it in the first place, namely:

```
pestpp-wrk infile /h hostname:nnnn
```

On recommencement of execution, PESTPP-WRK follows its usual protocol of input file perusal, testing the speed of the machine on which it is running, announcing its presence to the PANTHER manager by connecting to the latter's TCP/IP port, and exchanging information with PANTHER so that the two programs can verify parameter, observation and command-line compatibility. Once this has been done, the PANTHER manager treats the re-commenced worker as if it were a new worker, assigning it model runs according to a priority that is determined by the speed of the machine on which the worker is running.

4.6 The Model Run Queue

On initialization by its calling program, the PANTHER manager sets up a model run queue. When a program which uses PANTHER (or its serial counterpart) adds an entry to the queue, it associates a set of parameters with this entry. Optionally it can associate other information with this entry, namely a 40 character text string and a double precision number. This extra information is not used by PANTHER for run management; it is used to assist in run recognition by the calling program.

Once a particular model run has finished, a set of model outputs are also associated with that run. Meanwhile, the values of parameters associated with that run may have been slightly altered. PESTPP-WRK follows the same protocol as that employed by members of the PEST and PEST++ suites when they write numbers to model input files. If the space to which a number is written does not allow representation of that number with the same precision with which it is stored in the computer's memory, the latter is adjusted to conform with the former. This strategy promulgates increased accuracy in certain numerical procedures, for example in calculation of finite difference derivatives.

A PANTHER calling program can remove a model run from the run queue before, during or after commissioning of that run. If a run is removed from the run queue while the model run is actually taking place, then PANTHER instructs the worker that is supervising the run to terminate it. (If two workers are currently supervising the same model run for reasons discussed above, then PANTHER gives this instruction to both of the supervising workers.)

PANTHER stores the contents of the run management queue in a binary file whose name is provided by its calling program. This file remains intact after PANTHER has been shut down (on purpose or inadvertently) and the calling program has ceased execution (on purpose or inadvertently). This provides the means for re-commencement of interrupted run management; see below.

4.7 Batched or Interactive Model Runs

Model run management, as undertaken by PANTHER, requires repeated traversal of a programming loop. Within this loop PANTHER performs the following tasks (not necessarily in the following order):

- It checks whether any model runs are complete. If a model run is complete it receives the outcomes of that run from the pertinent worker and stores them in the same binary file as that which holds the run queue. It registers the model run as complete and the worker as free.
- It checks whether any runs are overdue. If a model run is overdue and the N_MAX_FAIL variable is set appropriately, the run is declared as available for worker assignment when other runs have been completed.
- It checks the model run queue for runs that have not yet been initiated or are marked for re-initiation. It checks whether any workers are free. It assigns the next run on the model run queue to the fastest available worker.

PANTHER offers a number of options to its calling program on whether, or not, its run management duties should be interrupted so that control can be returned to the calling program for a while. The simplest option is for PANTHER to undertake run management

through repeated traversal of its management loop until the model run queue is empty. Hence control is returned to the calling program only after all of the runs in the run queue have been carried out (or flagged as having failed).

As another alternative, the calling program can request that PANTHER manages model runs for a while, and then hands control back to it. This gives the calling program an opportunity to review the results of recently completed runs. It may then decide to remove runs from the run queue and/or replace them with other runs that are associated with other sets of parameters. In exercising this option, the calling program can instruct PANTHER to complete a certain number of management loops, or to cycle through the management loop for a certain amount of time.

4.8 Moving Files

4.8.1 General

A calling program can request that the PANTHER run manager transfer a file from a worker's folder to its own folder. It can also request that PANTHER distribute this, or another, file to the folders of other workers. This may be a useful strategy in calibration and optimization contexts where model run times are reduced if initial conditions are updated as model parameters change. For example, when calibrating a steady state groundwater or a natural state geothermal model, it may be advantageous to distribute model-calculated system states corresponding to the best set of parameters achieved during one iteration of the inversion process to the working folders of all workers so that these can serve as initial states for model runs conducted during the next iteration of the inversion process.

As a security measure, PANTHER encrypts files before transferring them. They are de-encrypted on arrival at their destination. As an extra precaution, PANTHER identifies transferred files by index rather than by name. When requesting a particular PESTPP-WRK worker to send it a file, PANTHER informs the worker of the index of this file. The worker then associates this index with a filename recorded on its input file; see below. When it receives the file from the worker, the calling program can save it locally using whatever name it pleases. Similarly, when sending a file to a worker, the PESTPP-WRK worker that receives the file is also given a filename index. It associates this index with a filename recorded on its input file. This name is given to the file as it saves a local copy of it.

4.8.2 File Transfer Section in a PESTPP-WRK Input File

Figure 4.1 shows a PESTPP-WRK input file. This is the same file as that listed in figure 3.1, except for the addition of an extra two sections. Unlike other sections of the PESTPP-WRK input file, these sections are optional. It is important to note that if one of these sections is present than the other section must also be present.

```
* model command line
model_command_line_1
model_command_line_2                    # optional comment line
* model input
template_file_1  model_input_file_1
template_file_2  model_input_file_2
etc
* model output
instruction_file_1 model_output_file_1
instruction_file_2 model_output_file_2
etc
# another optional comment
* file transfer
data_file_1
data_file_2
data_file_3
data_file_1
etc
* file transfer security
hmac
my_password
* any text
```

Figure 4.1 A PESTPP-WRK file containing a "file transfer" section and a "file transfer security" section.

The sections depicted in figure 4.1 that do not appear in figure 3.1 are the "file transfer" and "file transfer security" sections.

The "file transfer" section must contain a list of filenames. (As usual, a filename with a space should be surrounded by quotes.) If the PANTHER run manager asks that PESTPP-WRK transfer a file to it, it identifies this file by sequence number in the "file transfer" section. Hence when PANTHER sends a file transfer request to a PESTPP-WRK worker, it includes this integer in the request. The file is identified through counting from the top of the list provided in the "file transfer" section, starting at 1. Naturally, the PANTHER calling program must be aware of the sequence order; presumably, it will have received this information through its own input dataset. As the file is transferred to the PANTHER run manager (using TCP/IP), PANTHER records its local copy of the file under a name provided by the calling program.

If the PANTHER run manager informs a PESTPP-WRK worker that it is sending a file to it for local storage, it selects the name under which this file will be stored by the worker by sequence number in the "file transfer" section of the PESTPP-WRK input file.

The first line following the header of the "file transfer security" section must denote the security type. At present, the only supported type is HMAC; others (for example AES) may be allowed in later versions of PANTHER and PESTPP-WRK. The line following this is the password used for encryption and de-encryption of transferred (and compressed) files. This password can optionally be enclosed in quotes; however it must not contain a space. Naturally, the program which calls the PANTHER parallel run manager must be provided with the same password as the worker.

4.9 Run Management Record

PANTHER records a complete history of communications between it and its workers in a human-readable text file. This is similar to the run management record file that is written by BEOPEST. It is the same file that programs of the PEST++ suite use to record run management details. The name of this file is provided to PANTHER by its calling program.

PANTHER also lists run management progress to the screen. A screen snapshot appears in the following figure.

```
PANTHER progress
   runs(C = completed  | F = failed | T = timed out)
   workers(R = running | W = waiting | U = unavailable)
------------------------------------------------------------------------------
06/29 09:57:26 runs(C=7    | F=2    | T=0    ): workers(R=1    | W=0    | U=0    )
```

Figure 4.2 An example of PANTHER screen output.

4.10 Re-Starting

If a program which calls the PANTHER run manager ceases execution prematurely, the contents of the run management queue (including the outcomes of model runs that have already been completed) are not lost. PANTHER provides a function for re-initiating a previously interrupted run management process. Provided that the binary file in which PANTHER stores the model run queue has not been deleted, PANTHER reads the contents of this file when this function call is issued. Calls to other PANTHER functions can then instruct it to resume run management in accordance with the contents of this queue.

Naturally, the PANTHER calling program must take care of whatever file storage it requires to support its own restart capabilities.

5. Linking to PANTHER

5.1 General

To use PANTHER run management functionality, a calling program must link to all of the libraries listed in table. 5.1.

Library	**Function**
panther.lib	The PANTHER run manager
wrappers.lib	Wrappers on the PANTHER library that enable access to its functionality from FORTRAN and C
abstract_base.lib	The base class from which PANTHER and the serial run manager are derived
common.lib	Utility functions used by members of the PEST++ suite
mio.lib	Supports the non-intrusive model interface using template and instruction files
pestpp_common.lib	Defines many of the C++ data structures used by PEST++ and PANTHER
serial.lib	The serial run manager
ws2_32.lib	Windows only. Required for socket programming (i.e. for TCP/IP communications)

Table 5.1. Libraries to which a PANTHER calling program must link.

5.2 Compilation Under Windows

Linkage to static libraries avoids the creation of an executable program which depends on dynamic link libraries (DLL's). This promulgates convenience of installation of PANTHER-calling software on another machine. Debug and release versions of static libraries, compiled using the Microsoft Visual Studio 2015 compiler, can be obtained from the PESTPP Github repository.

Alternatively, the entire visual studio can also be downloaded from this repository and compiled on your own PC.

Compilation of programs which call PANTHER functions can be undertaken by introducing them to the PESTPP Visual Studio project, or by creating a new project. Alternatively linkage to the static libraries listed in table 5.1 can be done from the command line.

As an example, consider the FORTRAN program listed in figure 5.1 which uses PANTHER functionality.

```
    program run_manager_fortran_test

    implicit none

! -- Declare variables
    integer nruns, npar, nobs
```

```
    integer err
    integer ipar, iobs, irun
    integer nfail
    integer n_total_runs
    integer n_max_fail
    integer modcomind
    integer condition
    integer no_ops
    integer return_cond
    integer id(100)

    double precision pars(5)
    double precision obs(19)
    double precision time_sec
    double precision runtime
    double precision overdue_resched_fac
    double precision overdue_giveup_fac
    double precision overdue_giveup_time

    character*20 comline(1)
    character*20 tpl(2)
    character*20 inp(2)
    character*20 ins(3)
    character*20 out(3)
    character*20 storfile
    character*20 port
    character*20 rmi_info_file
    character*20 p_names(5)
    character*50 o_names(19)
    character buf

! -- Declare PANTHER functions
    integer rmif_create_serial
    integer rmif_create_panther
    integer rmif_initialize
    integer rmif_add_run
    integer rmif_run_until
    integer rmif_get_run
    integer rmif_get_n_total_runs
    integer rmif_get_n_failed_runs
    integer rmif_delete

    external rmif_create_serial
    external rmif_create_panther
    external rmif_initialize
    external rmif_add_run
    external rmif_run_until
    external rmif_get_run
    external rmif_get_n_total_runs
    external rmif_get_n_failed_runs
    external rmif_delete

! -- Values are assigned to variables that pertain to the model.
    data comline   /'ves.exe             '/
    data tpl       /'ves1.tpl            ',      &
                    'ves2.tpl            '/
    data inp       /'a_model.in1         ',      &
                    'a_model.in2         '/
    data ins       /'ves1.ins            ',      &
                    'ves2.ins            ',      &
                    'ves3.ins            '/
    data out       /'a_model.ot1         ',      &
                    'a_model.ot2         ',      &
                    'a_model.ot3         '/
    data storfile  /'tmp_run_data.bin    '/

! -- Names are assigned to parameters and observations.
    data p_names   /'ro1                 ',   &
                    'ro2                 ',   &
                    'ro3                 ',   &
                    'h1                  ',   &
                    'h2                  '/

    data o_names   /'ar1                 ',   &
                    'ar2                 ',   &
                    'ar3                 ',   &
```

```
                    'ar4                ',   &
                    'ar5                ',   &
                    'ar6                ',   &
                    'ar7                ',   &
                    'ar8                ',   &
                    'ar9                ',   &
                    'ar10               ',   &
                    'ar11               ',   &
                    'ar12               ',   &
                    'ar13               ',   &
                    'ar14               ',   &
                    'ar15               ',   &
                    'ar16               ',   &
                    'ar17               ',   &
                    'ar18               ',   &
                    'ar19               '/

! -- A set of values are assigned to parameters.
    data pars / 1.0, 1.0, 2.0, 2.0, 10.0 /

! -- Values are assigned to variables which govern run management.
    rmi_info_file ='run_manager_info.txt'
    n_max_fail    = 20
    overdue_resched_fac=1.15
    overdue_giveup_fac=100.0
    overdue_giveup_time=1.0d30

! -- Instantiate PANTHER parallel run manager
9   write(6,10,advance='no')
10  format(' Enter port number: ')
    read(5,*,err=9) port
    err = rmif_create_panther(storfile, 20,                        &
            port, 20,                                              &
            rmi_info_file, 20,                                     &
            n_max_fail, overdue_resched_fac, overdue_giveup_fac,   &
            overdue_giveup_time)

! -- Initialize the run manager for this particular problem.
    nruns = 10
    npar = 5
    nobs = 19
    err = rmif_initialize(p_names, 20, npar, o_names, 50, nobs)

! -- Add NRUNS model runs to the queue
    modcomind=1
    par(2)=1.0d0
    err = rmif_add_run(pars, npar, modcomind, id(1))
    do irun = 2, nruns
        pars(2) = pars(2) + pars(2) * 1.2
        err = rmif_add_run(pars, npar, modcomind, id(irun))
    end do

! -- Perform the model runs.
    write(*,*) ' Performing model runs...'
    condition=0
    no_ops=0
    time_sec=0.0d0
    err = rmif_run_until(condition, no_ops, time_sec, return_cond)
    write(*,*) ' Model runs complete...'

! -- See if any runs failed.
    err = rmif_get_n_failed_runs(nfail)
    write(*,*) 'Number of failed runs =', nfail

! -- Read results
    do irun = 1, nruns
        err = rmif_get_run(id(irun), pars, npar, obs, nobs)
        write(*,*)
        write(*,*)
        write(*,20) irun
20      format(' Results for model run ',i2,': -')
        write(*,30)
30      format(/,'    Parameter_name      Parameter_value')
        do ipar = 1, npar
            write(*,40) trim(p_names(ipar)),pars(ipar)
40          format(t10,a,t30,1pg14.7)
```

```
        end do
        write(*,50)
50      format(/,'    Observation_name    Observation_value')
        do iobs = 1, nobs
            write(*,40) trim(o_names(iobs)),obs(iobs)
        end do
    end do

! -- Record the total number of model runs.
    err = rmif_get_n_total_runs(n_total_runs)
    write(*,*)
    write(*,*) ' Total number of model runs =', n_total_runs

! -- clean up
    err = rmif_delete()

    end program run_manager_fortran_test
```

Figure 5.1. A FORTRAN program which calls PANTHER functions.

Using the Intel FORTRAN compiler, the above program can be compiled to an object module with the following command line:

```
ifort fortran_test.f90  /names:lowercase /assume:underscore /c /check:all /trace
```

It can then be linked to the libraries listed in table 5.1 using the following command:

```
link  fortran_test.obj  /SUBSYSTEM:CONSOLE  /LIBPATH:"E:\project\src\x64\Release\\"
panther.lib  wrappers.lib  abstract_base.lib  common.lib  mio.lib  pestpp_common.lib
serial.lib ws2_32.lib
```

In the above command the string "`E:\project\src\x64\Release\\`" must be replaced by the name of the folder in which these libraries reside on your own machine.

Program FORTRAN_TEST runs a model 10 times using 10 different parameter sets. If you run it, PESTPP-WRK should be run from another command line window open to the same folder.

FORTRAN_TEST's prompt, a suitable response, and its other screen outputs are as follows.

```
Enter port number: 4004

             Generalized Run Manager Interface
                       developed by:

                        Dave Welter
          Computational Water Resource Engineering


          starting PANTHER master...

IP addresses:
  0.0.0.0:4004 (IPv4)

PANTHER master listening on socket: 0.0.0.0:4004 (IPv4)
  Performing model runs...
    running model 10 times

      waiting for workers to appear...


PANTHER progress
   runs(C = completed | F = failed | T = timed out)
   workers(R = running | W = waiting | U = unavailable)
------------------------------------------------------------------------------
```

```
10/24 12:14:04 runs(C=10   | F=0    | T=0    ): workers(R=0   | W=0   | U=1   )

   10 runs complete :  0 runs failed


  Model runs complete...
 Number of failed runs =           0


 Results for model run  1: -

    Parameter_name      Parameter_value
         ro1                   1.000000
         ro2                   1.000000
         ro3                   2.000000
         h1                    2.000000
         h2                    10.00000

    Observation_name    Observation_value
         ar1                   1.000051
         ar2                   1.000160
         ar3                   1.000500
         ar4                   1.001557
         ar5                   1.004786
         ar6                   1.014266
         ar7                   1.040043
         ar8                   1.100935
         ar9                   1.216050
         ar10                  1.379702
         ar11                  1.556220
         ar12                  1.710884
         ar13                  1.827468
         ar14                  1.904856
         ar15                  1.950798
         ar16                  1.975702
         ar17                  1.988344
         ar18                  1.994501
         ar19                  1.997427
etc
```

Meanwhile PESTPP-WRK should be run using the following command:

```
pestpp-wrk case.pst /h hostname:4004
```

In the above command *case.pst* should be replaced by the name of a PEST control file that features the same parameters and observations for which values are provided in the *fortran_test.f90* source code. *hostname* should be replaced by the hostname of your computer.

6. FORTRAN Interface

6.1 Nomenclature

The terms “parameters” and “observations” are used herein to respectively denote numbers which are recorded on model input files through the agency of template files and numbers which are read from model output files through the agency of instruction files. In practice, these numbers may have other roles (such as decision variables and management outcomes). Nevertheless, use of this nomenclature simplifies the description presented below.

The names of FORTRAN-callable functions start with the string “rmif”. This stands for “run manager interface FORTRAN”.

6.2 Choice of Run Manager

Before undertaking any run management, a PANTHER calling program must instantiate either the PANTHER parallel run manager or the alternative, serial run manager. Once a run manager is instantiated, run management functions can be called. These are the same, regardless of whether runs are being undertaken in serial or in parallel.

rmif_create_serial

Purpose

Function *rmif_create_serial* initializes the serial run manager. Until this run manager is de-initialized, all run management requests delivered through the functions listed in the following subsection are delivered to this run manager.

Specifications

```
integer function rmif_create_serial (comline, comline_len,
comline_array_dim, tpl, tpl_len, tpl_array_dim, inp, inp_len,
inp_array_dim, ins, ins_len, ins_array_dim, out, out_len, out_array_dim,
storefile, storefile_len, rundir, rundir_len, n_max_fail)

character(len=comline_len), intent(in)    :: comline(comline_array_dim)
integer, intent(in)                       :: comline_len
integer, intent(in)                       :: comline_array_dim
character(len=tpl_len), intent(in)        :: tpl(tpl_array_dim)
integer, intent(in)                       :: tpl_len
integer, intent(in)                       :: tpl_array_dim
character(len=inp_len), intent(in)        :: inp(inp_array_dim)
integer, intent(in)                       :: inp_len
integer, intent(in)                       :: inp_array_dim
character(len=ins_len), intent(in)        :: ins(ins_array_dim)
integer, intent(in)                       :: ins_len
integer, intent(in)                       :: ins_array_dim
character(len=out_len), intent(in)        :: out(out_array_dim)
integer, intent(in)                       :: out_len
integer, intent(in)                       :: out_array_dim
character (len=storefile_len), intent(in) :: storefile
integer, intent(in)                       :: storefile_len
character(len=rundir_len), intent(in)     :: rundir
integer, intent(in)                       :: rundir_len
integer, intent(in)                       :: n_max_fail
```

Arguments: on entry

`comline`
An array of character strings holding model command lines.

`comline_len`
The length of the character variable stored in each element of the `comline` array.

`comline_array_dim`
The number of elements in the `comline` array.

`tpl`
An array of character variables holding the names of template files.

`tpl_len`
The length of the character variable stored in each element of the `tpl` array.

`tpl_array_dim`
The number of elements in the `tpl` array.

`inp`
An array of character strings holding the names of model input files that correspond to template files.

`inp_len`
The length of the string stored in each element of the `inp` array.

`inp_array_dim`
The number of elements in the `inp` array. This must be the same as the number of elements in the `tpl` array.

`ins`
An array of character strings holding the names of instruction files.

`ins_len`
The length of the string stored in each element of the `ins` array.

`ins_array_dim`
The number of elements in the `ins` array.

`out`
An array of character strings holding the names of model output files that correspond to instruction files.

`out_len`
The length of the string stored in each element of the `out` array.

`out_array_dim`
The number of elements in the `out` array. This must be the same as the number of elements in the `ins` array.

`storefile`
Character variable that specifies the name of the binary file used by the run manager to store parameters and observations associated with model runs (i.e. the model run queue).

`len_storefile`
Length of the `storefile` character variable.

`rundir`
The folder to which the serial run manager should transfer its focus before issuing the command to run the model.

`rundir_len`
The length of the `rundir` text string.

`n_max_fail`

The number of times that the run manager should attempt to re-run a failed model run before declaring it to have failed.

Return value
0 for success; 1 for failure.

rmif_create_panther

Purpose
Function *rmif_create_panther* initializes the PANTHER parallel run manager. Until this run manager is de-initialized, all run management requests delivered through the functions listed in the following subsection are delivered to this run manager.

Note that the PANTHER parallel run manager does not need to be informed of the names of template and instruction files, nor of model input and output files with which they are associated, because non-intrusive interaction with the model is the job of the worker.

Specifications
```
integer function rmif_create_panther (storefile, storefile_len, port,
port_len, info_filename, info_filename_len, n_max_fail,
overdue_resched_fac, overdue_giveup_fac, overdue_giveup_time)

character(len=storefile_len), intent(in)      :: storefile
integer, intent(in)                           :: storefile_len
character(len=port), intent(in)               :: port_len
integer, intent(in)                           :: port_len
character (len=info_filename_len), intent(in) :: info_filename
integer, intent(in)                           :: info_filename_len
integer, intent(in)                           :: n_max_fail
double precision, intent(in)                  :: overdue_resched_fac
double precision, intent(in)                  :: overdue_giveup_time
```

Arguments: on entry
`storefile`
Character variable that specifies the name of the binary file used by the run manager to store parameters and observations associated with model runs (i.e. the model run queue).

`len_storefile`
Length of the `storefile` character variable.

`port`
Character variable that specifies the port number which PANTHER opens for TCP/IP communication between itself and its workers.

`port_len`
Length of the `port` character variable.

`info_filename`
Character variable that specifies the name of the run management record file that PANTHER should write.

`info_filename_len`
Length of the `info_filename` character variable.

`n_max_fail`
The number of times that the run manager should attempt to re-run a failed model run before declaring it to have failed.

`overdue_resched_fac`
This factor is applied to the average model run time; it must be greater than 1.0. If a particular model run has been running for longer than the average model run time multiplied by this factor, then the run is re-initiated on another worker. This process is repeated up to N_MAX_FAIL times if the second (and subsequent) runs are similarly slow.

`overdue_giveup_fac`
If a model runs for longer than `overdue_giveup_fac` times the average model run time, and the model run queue is not empty, then the model run is terminated and recorded as having timed out. However if the run queue is empty, the model run is allowed to continue unless terminated by PANTHER in response to a request by the calling program.

`overdue_giveup_time`
If a model runs for longer than `overdue_giveup_time`, and the model run queue is not empty, the model run is terminated and recorded as having timed out. However if the run queue is empty, the model run is allowed to continue unless terminated by PANTHER in response to a request by the calling program. This feature can be disabled by specifying a number less than or equal to 0.

Return value
0 for success; 1 for failure.

6.3 Run Management Functions

rmif_initialize

Purpose
Function *rmif_initialize* provides the names of parameters and observations to the run manager. These names must be fully compatible with the names of parameters featured in template files pertaining to the current problem, and with the names of observations appearing in instruction files pertaining to the current problem. The names of template and instruction files are provided to the serial run manager through the *rmif_create_serial* function. They are provided to the PESTPP-WRK worker program through its input file.

Specifications
```
integer function rmif_initialize(parnames, parnames_len,
parnames_array_dim, obsnames, obsnames_len, obsnames_array_dim)

character(len=parnames_len), intent(in)  :: parnames(parnames_array_dim)
integer, intent(in)                      :: parnames_len
integer, intent(in)                      :: parnames_array_dim
character(len=obsnames_len), intent(in)  :: obsnames(obsnames_array_dim)
```

```
integer, intent(in)                    :: obsnames_len
integer, intent(in)                    :: obsnames_array_dim
```

Arguments: on entry

`parnames`
The ordered names of parameters whose values will be supplied or retrieved in subsequent calls to the run manager.

`parnames_len`
Length of character variables comprising elements of the `parnames` array.

`parnames_array_dim`
The number of elements in the `parnames` array.

`obsnames`
The ordered names of observations whose values are retrieved on subsequent calls to the run manager.

`obsnames_len`
Length of character variables comprising elements of the `obsnames` array.

`obsnames_array_dim`
The number of elements in the `obsnames` array.

Return value

0 for success; 1 for failure.

rmif_reinitialize

Purpose

The *rmif_reinitialize* function reinitializes the run manager. The names of parameters and observations supplied in the previous call to *rmif_initialize* are maintained. However the model run queue is emptied, and any running instances of the model are terminated. The run manager is then ready for re-population of the run queue by the calling program.

Specifications

```
integer function rmif_reinitialize()
```

Arguments

Function *rmif_reinitialize* has no arguments.

Return value

0 for success; 1 for failure.

rmif_initialize_restart

Purpose

If the run manager was deleted, or if previous execution of a program which calls the run manager was prematurely terminated, calling of function *rmif_initialize_restart* allows run management to continue where it was prematurely interrupted. The restarted run manager obtains the information which it needs from the binary file in which the model run queue and other run management information was stored during the previous run management session.

Specifications

```
integer function rmif_initialize_restart(storefile, storefile_len)
```

```
character(len=storefile_len), intent(in) :: storefile
integer, intent(in)                      :: storefile_len
```

Arguments: on entry

`storefile`

Character variable that specifies the name of the binary file used by the run manager to store the values of parameters and observations associated with model runs.

`len_storefile`

Length of the `storefile` character variable.

Return value

0 for success; 1 for failure.

rmif_add_run

Purpose

Once a run manager has been initialized, the *rmif_add_run* function can be called repeatedly. On each occasion that it is called, it adds a new run to the run queue.

Specifications

```
integer function rmif_add_run(pardata, npar, modcomind, id)
double precision, intent(in) :: pardata(npar)

integer, intent(in)          :: npar
integer, intent(in)          :: modcomind
integer, intent(out)         :: id
```

Arguments: on entry

`pardata`

This array contains the values of parameters that the model must employ on this run. Parameter values must be supplied in the same order as were parameter names in the previous call to the *rmif_initialize* function.

`npar`

The number of elements in the `pardata` array. This must equal the value supplied for `npar` in the previous call to the *rmif_initialize* function.

`modcomind`

The index of the command that must be used to run the model. This number must equal or exceed unity.

Argument: on exit

`id`

The integer identifier assigned to this run by the run manager. This identifier is used in all future references to this model run.

Return value

0 for success; 1 for failure.

rmif_add_run_with_info

Purpose

Function *rmif_add_run_with_info* is similar to function *rmif_add_run* in that it adds a run to the model run queue. However, it allows the calling program to associate a text string and double precision number with the model run. These are used for informational purposes only. They can assist a calling program in tracking the run**.**

Specifications

```
integer function rmif_add_run_with_info(pardata, npar, modcomind, infotext,
infotext_len, infovalue, id)

double precision, intent(in)            :: pardata(npar)
integer, intent(in)                     :: npar
integer, intent(in)                     :: modcomind
character(len=infotext_len), intent(in) :: infotext
integer, intent(in)                     :: infotext_len
double precision, intent(in)            :: infovalue
integer, intent(out)                    :: id
```

Arguments: on entry

`pardata`

This array contains the values of parameters that the model must employ on this run. Parameter values must be supplied in the same order as were parameter names in the previous call to the *rmif_initialize* function.

`npar`

The number of elements in the `pardata` array. This must equal the value supplied for `npar` in the previous call to the *rmif_initialize* function.

`modcomind`

The index of the command that must be used to run the model. This number must equal or exceed unity.

`infotext`

A text string that the calling program wishes to associate with the model run. The run manager does not use this information. It is stored with the run, and can optionally be retrieved with model results. The run manger can store a maximum of 40 characters in this run-associated string; it truncates characters supplied in `infotext` after the fortieth.

`infotext_len`

The length of the `infotext` character variable.

`infovalue`

A double precision number that the calling program wishes to associate with the model run. The run manager does not use this information. It is stored with the run, and can optionally be retrieved with model results.

Arguments: on exit

`id`

The integer identifier assigned to this run by the run manager. This identifier is used in all future references to this model run.

Return value
0 for success; 1 for failure.

rmif_run

Purpose
Function *rmif_run* instructs the run manager to carry out all of the runs in the model run queue, and to store observations that are generated by these runs in a binary file whose name was provided through the *rmif_create* function. Control is not returned to the calling program until all model runs have been completed.

Specifications
```
integer function rmif_run()
```

Arguments
This function has no arguments.

Return value
0 for success; 1 for failure.

rmif_run_until

Purpose
The *rmif_run_until* function instructs the PANTHER parallel run manger to manage runs in the model run queue. However it may return control to the calling program before emptying the queue. Note that if the serial run manager is being used, this function has the same effect as the *rmif_run* function.

Specifications
```
integer function rmif_run_until(condition, no_ops, time_sec, return_cond)

integer, intent(in)           :: condition
integer, intent(in)           :: no_ops
double precision, intent(in) :: time_sec
integer, intent(out)          :: return_cond
```

Arguments: on entry
`condition`

Specifies when function *rmif_run_until* should return control to the calling program. Options are:

- 0 return after all model runs are complete;
- 1 return after *no_opts* consecutive run management loops have not received any TCP/IP signals from workers;
- 2 return after *time_sec* seconds have elapsed;
- 3 return after *no_opts* consecutive run management loops have not received any TCP/IP signals from workers, or after *time_sec* seconds have elapsed, whichever is sooner.

`no_ops`

The number of run management loops to undertake while listening for TCP/IP signals from workers before returning control to the calling program.

`time_sec`

The maximum time in seconds that function *rmif_run_until* will devote to managing runs before returning control to the calling program

Arguments: on exit

`return_cond`

The reason that function *rmif_run_until* is returning control to the calling program:

0	all runs are complete;
1	the *no_opts* condition was active and was met;
2	the *time_sec* condition was active and was met.

Return value

0 for success; 1 for failure.

rmif_get_run

Purpose

Function *rmif_get_run* retrieves the values of parameters and observations associated with a particular model run. The values of parameters may be slightly altered from those which were originally supplied to the run manager through a call to the *rmif_add_run* or *rmif_add_run_with_info* function; they are exactly equal to the values recorded on model input files.

Specifications

```
integer function rmif_get_run(id, pardata, npar, obsdata, nobs)

integer, intent(in)            :: id
double precision, intent(out) :: pardata(npar)
integer, intent(in)            :: npar
double precision, intent(out) :: obsdata(nobs)
integer, intent(in)            :: nobs
```

Arguments: on entry

`id`

The run-manager-supplied identifier of the run for which parameter and observation values are sought.

`npar`

The number of parameters, and hence the size of the *`pardata`* array. The value of *`npar`* supplied with this function must be the same as that supplied through the *rmif_initialize* function.

`nobs`

The number of observations, and hence the size of the `obsdata` array. The value of `nobs` supplied with this function must be the same as that supplied through the *rmif_initialize* function.

Arguments: on exit

`pardata`

The values of parameters associated with the specified model run. The ordering of parameters in this array respects that established in the previous call to the *rmif_initialize* function.

`obsdata`

The values of observations associated with the specified model run. The ordering of observations in this array respects that established in the previous call to the *rmif_initialize* function.

Return value

0 for success; 1 for failure.

rmif_get_run_with_info

Purpose

Function *rmif_get_run_with_info* returns the values of parameters and observations associated with a particular model run. In doing this, it performs the same role as the *rmif_get_run* function. However, it also returns the text string and double precision number that was associated with that model run using the *rmif_add_run_with_info* function. Note that the values of parameters that are associated with a particular model run may be slightly altered from those which were originally supplied to the run manager through a call to the *rmif_add_run* or *rmif_add_run_with_info* function; they are exactly equal to the values recorded on model input files.

Specifications

```
integer function rmif_get_run_with_info(id, par_data, npar, obsdata, nobs
infotext, infotext_len, infovalue)

integer, intent(in)                          :: id
double precision, intent(out)                :: pardata(npar)
integer, intent(in)                          :: npar
double precision, intent(out)                :: obsdata(nobs)
integer, intent(in)                          :: nobs
character (len=infotext_len), intent(out) :: info_text
integer, intent(in)                          :: info_txt_len
double precision, intent(out)                :: info_value
```

Arguments: on entry

`id`

The run-manager-supplied identifier of the model run for which parameter and observation values are required.

`npar`

The number of parameters, and hence the size of the *`pardata`* array. The value of `npar` supplied with this function must be the same as that supplied through the *rmif_initialize* function.

`nobs`

The number of observations, and hence the size of the `obsdata` array. The value of `nobs` supplied with this function must be the same as that supplied through the *rmif_initialize* function.

`infotext_len`

Length of the `infotext` character variable.

Arguments: on exit

`pardata`

The values of parameters associated with the specified model run. The ordering of parameters in this array respects that established in the previous call to the *rmif_initialize* function.

`obsdata`
The values of observation associated with the specified model run. The ordering of observations in this array respects that established in the previous call to the *rmif_initialize* function.

`infotext`
A text string supplied through function *rmif_add_run_with_info*. The run manager does not use this string; it is for the benefit of the calling program only. Although the length of the *infotext* character variable can be greater than 40 characters, this is the maximum size of the text string that function *rmif_get_run_with_info* can return.

`infovalue`
A double precision number associated with this model run supplied through function *rmif_add_run_with_info*. The run manager does not use this string; it is stored only for the benefit of the calling program.

Return value
0 for success; 1 for failure.

rmif_cancel_run

Purpose
Removes a run from the model run queue. If the job is running, it kills the job.

Specifications
```
integer function rmif_cancel_run(id)

integer, intent(in) :: id
```

Arguments: on entry
`id`
The integer identifier for the model run that was assigned to it by the run manager when the run was added to the run queue.

Return value
0 for success; 1 for failure.

rmif_get_run_status

Purpose
As the name implies, function *rmif_get_run_status* allows a calling program to retrieve the status of a model run.

Specifications
```
integer function rmif_get_run_status (id, run_status, max_runtime,
n_concurrent_runs)

integer, intent(in)           :: id
integer, intent(out)          :: run_status
double precision, intent(out) :: max_runtime
integer, intent(out)          :: n_concurrent_runs
```

Arguments: on entry

`id`

The run-manager-supplied identifier of the model run whose status is sought.

Arguments: on exit

`run_status`

The status of the model run. Options are as follows:

0	the run has not yet been completed;
-90	the run was cancelled by the calling program;
-1 to -89	the run failed this number of times;
1	the run completed successfully.

`max_runtime`

The length of time that the model has been running. If concurrent runs of the same model are being undertaken, this is the runtime of the longest active run.

`n_concurrent_runs`

The number of the node on which the specified model run is currently running.

Return value

0 for success; 1 for failure.

rmif_get_run_info

Purpose

Function *rmif_get_run_info* allows a calling program to retrieve the information associated with a model run. This information includes that which is associated with a model run by the calling program.

Specifications

```
integer function rmif_get_run_info (id, run_status, modcomind, infotext,
infotext_len, info_value)

integer, intent(in)                       :: id
integer, intent(out)                      :: run_status
integer, intent(out)                      :: modcomind
character(len=infotext_len), intent(out)  :: infotext
integer, intent(in)                       :: infotext_len
double precision, intent(out)             :: infovalue
```

Arguments: on entry

`id`

The run-manager-supplied identifier of the model run whose status is sought.

Arguments: on exit

`run_status`

The status of the model run. Options are as follows:

0	the run has not yet been completed;
-90	the run was cancelled by the calling program;
-1 to -89	the run failed this number of times;
1	the run completed successfully.

`modcomind`

The index of the command used to run the model.

`infotext`
The text string associated with the model run by the calling program. Characters after the fortieth are truncated.

`infotext_len`
The length of the `infotext` character variable.

`infovalue`
A double precision number associated with the model run by the calling program.

Return value
0 for success; 1 for failure.

rmif_get_n_failed_runs

Purpose
Function *rmif_get_n_failed_runs* returns the number of model runs that failed to execute successfully even after repeated attempts to run them.

Specifications
```
integer function rmif_get_n_failed_runs(nfail)

integer,intent(out) :: nfail
```

Argument: on exit
`nfail`
The total number of runs on the model run queue that failed to successfully execute, even after repeated attempts.

Return value
0 for success; 1 for failure.

rmif_get_failed_run_ids

Purpose
Function *rmif_get_failed_run_ids* provides integer identifiers of failed model runs. The *rmif_get_n_failed_runs* function should be called prior to calling function *rmif_get_failed_run_ids* so that arrays can be properly dimensioned.

Specifications
```
integer function rmif_get_failed_run_ids(run_id_array, run_id_array_dim)

integer, intent(out) :: run_id_array(run_id_array_dim)
integer, intent(in)  :: run_id_array_dim
```

Arguments: on entry
`run_id_array_dim`
Dimension of the `run_id_array` array.

Arguments: on exit
`run_id_array`
The integer identifiers of failed model runs.

Return value
0 for success; 1 for failure.

rmif_get_n_cur_runs

Purpose

Function *rmif_get_n_cur_runs* returns the number of unique model runs currently stored in the model run queue. This number includes active runs, waiting runs, cancelled runs and failed runs. Each run is only counted once, even if multiple instances of it were run or are running.

Specifications

```
integer function rmif_get_n_cur_runs(nruns)

integer, intent(out) :: nruns
```

Arguments: on exit

`nruns`
The number of model runs comprising the run queue.

Return value

0 for success; 1 for failure.

rmif_get_n_total_runs

Purpose

Function *rmif_get_n_total_runs* returns the total number of model runs performed by the run manager. This does not get reset when the run manger is reinitialized.

Specifications

```
integer function rmif_get_n_total_runs(nruns)

integer, intent(out) :: nruns
```

Arguments: on exit

`nruns`
The number of model runs that the run manager has performed.

Return value

0 for success; 1 for failure.

rmif_err_msg

Purpose

If a run management function call fails, this function should be called to retrieve the text of an error message saved by the run manger.

Specifications

```
integer function rmif_err_msg(errtxt, errtxt_len)

character(len=infotext_len), intent(out) :: errtxt
integer, intent(in)                      :: errtxt_len
```

Arguments: on entry

`errtext_len`
The length of the `errtext` character variable.

Arguments: on exit

`errtext`

A text string used to retrieve an error message saved by the run manager. Because the run manager stores the error message internally as a C++ string, there are no hard limits on its length. However the FORTRAN function wrapper truncates the message at the number of characters supplied in `infotext`, or after the fortieth character, whichever is less.

Return value

0 for success; 1 for failure.

rmif_delete

Purpose

This function is the destructor for the run manager. It should be called only after the program which uses the run manager has finished commissioning model runs so that run management resources can be freed. After function *rmif_delete* has been called, the current instance of the run manger can no longer be used.

Specifications

```
integer function rmif_delete()
```

Arguments

This function has not arguments.

Return value

0 for success; 1 for failure.

7. C Interface

7.1 Nomenclature

The terms “parameters” and “observations” are used herein to respectively denote numbers which are recorded on model input files through the agency of template files, and numbers which are read from model output files through the agency of instruction files. In practice, these numbers may have other roles (such as decision variables and management outcomes). Nevertheless, use of this nomenclature simplifies the following description.

The names of the C-callable functions listed below start with the string “rmic”. This stands for “run manager interface C”.

7.2 Include Files

A C program file in which calls are made to PANTHER functions should include the following line at the start of the file.

```
#include "RunManagerCWrapper.h"
```

Obviously, file *RunManagerCWrapper.h* should reside in the source code folder.

7.3 Choice of Run Manager

Before undertaking any run management, a PANTHER calling program must instantiate either the PANTHER parallel run manager or the alternative, serial run manager. Once a run manager is instantiated, run management functions can be called. These are the same, regardless of whether runs are being undertaken in serial or in parallel.

rmic_create_serial

Purpose

Function *rmic_create_serial* initializes the serial run manager. Until this run manager is de-initialized, all run management requests delivered through the functions listed in the following subsection are delivered to this run manager.

Specifications

```
RunManager* rmic_create_serial(
      char **comline, int comline_array_dim,
      char **tpl, int tpl_array_dim,
      char **inp, int inp_array_dim,
      char **ins, int ins_array_dim,
      char **out, int out_array_dim,
      char *storefile,
      char *rundir,
      int n_max_fail)
```

Arguments: on entry

`comline`

An array of character strings holding model command lines.

`comline_array_dim`

The number of elements in the `comline` array.

`tpl`

An array of character strings holding the names of template files.

`tpl_array_dim`
The number of elements in the `tpl` array.

`inp`
An array of character strings holding the names of model input files that correspond to template files.

`inp_array_dim`
The number of elements in the `inp` array. This must be the same as the number of elements in the `tpl` array.

`ins`
An array of character strings holding the names of instruction files.

`ins_array_dim`
The number of elements in the `ins` array.

`out`
An array of character strings holding the names of model output files that correspond to instruction files.

`out_array_dim`
The number of elements in the `out` array. This must be the same as the number of elements in the `ins` array.

`storefile`
Character string that specifies the name of the binary file used by the run manager to store parameters and observations associated with model runs (i.e. the model run queue).

`rundir`
The folder to which the serial run manager should transfer its focus before issuing the command to run the model.

`n_max_fail`
The number of times that the run manager should attempt to re-run a failed model run before declaring it to have failed.

Return value
On success, function *rmic_create_serial* returns a pointer to an instance of a run manager object. On failure it returns the null pointer.

rmic_create_panther

Purpose
Function *rmic_create_panther* initializes the PANTHER parallel run manager. Until this run manager is de-initialized, all run management requests delivered through the functions listed in the following subsection are delivered to this run manager.

Note that the PANTHER parallel run manager does not need to be informed of the names of template and instruction files, nor of model input and output files with which they are associated because non-intrusive interaction with the model is the job of the worker.

Specifications

```
RunManager* rmic_create_panther(
      char *storefile,
      char *port,
      char *info_filename,
      int n_max_fail,
      double overdue_resched_fac,
      double overdue_giveup_fac,
      double overdue_giveup_time)
```

Arguments: on entry

`storefile`

Character string that specifies the name of the binary file used by the run manager to store parameters and observations associated with model runs (i.e. the model run queue).

`port`

Character string that specifies the port number which PANTHER opens for TCP/IP communication between itself and its workers.

`info_filename`

Character string that specifies the name of the run management record file that PANTHER should write.

`n_max_fail`

The number of times that the run manager should attempt to re-run a failed model run before declaring it to have failed.

`overdue_resched_fac`

This factor is applied to the average model run time; it must be greater than 1.0. If a particular model run has been running for longer than the average model run time multiplied by this factor, then the run is re-initiated on another worker. This process is repeated up to N_MAX_FAIL times if the second (and subsequent) run(s) are similarly slow.

`overdue_giveup_fac`

If a model runs for longer than `overdue_giveup_fac` times the average model run time, and the model run queue is not empty, then the model run is terminated and recorded as having timed out. However if the run queue is empty, the model run is allowed to continue unless terminated by PANTHER in response to a request by the calling program.

`overdue_giveup_time`

If a model runs for longer than this time and the model run queue is not empty, then the model run is terminated and recorded as having timed out. However if the run queue is empty, the model run is allowed to continue unless terminated by PANTHER in response to a request by the calling program. This feature can be disabled by specifying a number less than or equal to 0.

Return value

On success, function *rmic_create_panther* returns a pointer to an instance of a run manager object. On failure it returns the null pointer.

7.4 Run Management Functions

rmic_initialize

Purpose

Function *rmic_initialize* provides the names of parameters and observations to the run manager. These names must be fully compatible with the names of parameters featured in template files pertaining to the current problem, and with the names of observations appearing in instruction files pertaining to the current problem. The names of template and instruction files are provided to the serial run manager through the *rmic_create_serial* function. They are provided to the PESTPP-WRK worker program through its input file.

Specifications

```
int rmic_initialize(RunManager *run_manager_ptr,
      char **parnames, int parnames_array_dim,
      char **obsnames, int obsnames_array_dim)
```

Arguments: on entry

`run_manager_ptr`

A previously initialized run manager object.

`parnames`

The ordered names of parameters whose values will be supplied or retrieved in subsequent calls to the run manager.

`parnames_array_dim`

The number of elements in the `parnames` array.

`obsnames`

The ordered names of observations whose values are retrieved on subsequent calls to the run manager.

`obsnames_array_dim`

The number of elements in the `obsnames` array.

Return value

0 for success; 1 for failure.

rmic_reinitialize

Purpose

The *rmic_reinitialize* function reinitializes the run manager. The names of parameters and observations supplied in the previous call to the *rmic_initialize* function are maintained. However the model run queue is emptied, and any running incidences of the model are terminated. The run manager is then ready for re-population of the model run queue.

Specifications

```
int rmic_reinitialize(RunManager *run_manager_ptr)
```

Arguments: on entry

`run_manager_ptr`

A previously initialized run manager object.

Return value

0 for success; 1 for failure.

rmic_intialize_restart

Purpose

If the run manager was deleted, or if previous execution of a program which calls the run manager was prematurely terminated, calling of function *rmic_initialize_restart* allows run management to continue where it was previously interrupted. The restarted run manager obtains the information which it needs from the binary file in which the model run queue, together with other run management information, was stored during the previous run management session.

Specifications

```
int rmic_initialize_restart(RunManager *run_manager_ptr, char *storefile)
```

Arguments: on entry

`run_manager_ptr`

A previously initialized run manager object.

`storefile`

Character array that specifies the name of the binary file used by the run manager to store parameters and observations associated with model runs.

Return value

0 for success; 1 for failure.

rmic_add_run

Purpose

Once a run manager has been initialized, the *rmic_add_run* function can be called repeatedly. On each occasion that it is called, it adds a new run to the run queue.

Specifications

```
int rmic_add_run(RunManager *run_manager_ptr,
      double *pardata, int npar,
      int modcomind,
      int *id)
```

Arguments: on entry

`run_manager_ptr`

A previously initialized run manager object.

`pardata`

This array contains the values of parameters that the model must employ on this run. Parameter values must be supplied in the same order as were parameter names in the previous call to the *rmic_initialize* function.

`npar`

The number of elements in the `pardata` array. This must equal the value supplied for `npar` in the previous call to the *rmic_initialize* function.

`modcomind`

The index of the command that must be used to run the model. This number must equal or exceed unity.

Arguments: on exit

`id`

The integer identifier assigned to this run by the run manager. This identifier is used in all future references to this model run.

Return value

0 for success; 1 for failure.

rmic_add_run_with_info

Purpose

Function *rmic_add_run_with_info* is similar to function *rmic_add_run* in that it adds a run to the model run queue. However, it allows the calling program to associate a text string and double precision number with the model run for informational purposes only. These can assist a calling program in tracking the run**.**

Specifications

```
int rmic_add_run_with_info(RunManager *run_manager_ptr,
      double *pardata, int npar,
      int modomind,
      char* infotext, double infovalue, int *id)
```

Arguments: on entry

`run_manager_ptr`

A previously initialized run manager object.

`pardata`

This array contains the values of parameters that the model must employ on this run. Parameter values must be supplied in the same order as were parameter names in the previous call to the *rmic_initialize* function.

`npar`

The number of elements in the `pardata` array. This must equal the value supplied for `npar` in the previous call to the *rmic_initialize* function.

`modcomind`

The index of the command that must be used to run the model. This number must equal or exceed unity.

`infotext`

A text string that the calling program wishes to associate with the model run. The run manager does not use this information. It is stored with the run, and can optionally be retrieved with model results. The run manger can store a maximum of 40 characters in this run-associated string; it truncates characters supplied in `infotext` after the fortieth.

`infovalue`

A double precision number that the calling program wishes to associate with the model run. The run manager does not use this information. It is stored with the run, and can be retrieved with model results.

Arguments: on exit

`id`

The integer identifier assigned to this run by the run manager. This identifier is used in all future references to this model run.

Return value

0 for success; 1 for failure.

rmic_run

Purpose

Function *rmic_run* instructs the run manager to carry out all of the runs in the model run queue, and to store observations that are generated by these runs in a binary file whose name was provided in the *rmic_create* function. Control is not returned to the calling program until all model runs have been completed.

Specifications

```
int rmic_run(RunManager *run_manager_ptr)
```

Arguments: on entry

`run_manager_ptr`

A previously initialized run manager object.

Return value

0 for success; 1 for failure.

rmic_run_until

Purpose

The *rmic_run_until* function instructs the PANTHER parallel run manger to manage runs in the model run queue. However it may return control to the calling program before emptying the queue. Note that if the serial run manager is being used, this function has the same effect as the *rmic_run* function.

Specifications

```
int rmic_run_until(RunManager *run_manager_ptr,
      int condition, int no_ops, double time_sec,
      int *return_cond)
```

Arguments: on entry

`run_manager_ptr`

A previously initialized run manager object.

`condition`

Specifies when function *rmic_run_until* should return control to the calling program. Options are:

- 0 return after all model runs are complete;
- 1 return after *no_opts* consecutive run management loops have not received any TCP/IP signal from workers;
- 2 return after *time_sec* seconds have elapsed;

3 return after *no_opts* consecutive run management loops have not received any TCP/IP signals from workers, or after *time_sec* seconds have elapsed, whichever is sooner.

`no_ops`

The number of run management loops to undertake while listening for TCP/IP signals from workers before returning control to the calling program.

`time_sec`

The maximum time in seconds that function *rmic_run_until* will devote to managing runs before returning control to the calling program

Arguments: on exit

`return_cond`

The reason that function *rmic_run_until* is returning control to the calling program:

0 all runs are complete;
1 the *no_opts* condition was active and was met;
2 the *time_sec* condition was active and was met.

Return value

0 for success; 1 for failure.

rmic_cancel_run

Purpose

Removes a run from the model run queue. If the job is running, it kills the job.

Specifications

```
int rmic_cancel_run(RunManager *run_manager_ptr, int run_id)
```

Arguments: on entry

`run_manager_ptr`

A previously initialized run manager object.

`id`

The integer identifier for the run that was provided by the run manager when the run was added to the run queue.

Return value

0 for success; 1 for failure.

rmic_get_run_status

Purpose

As the name implies, function *rmic_get_run_status* informs a calling program of the status of a model run.

Specifications

```
int rmic_get_run_status(RunManager *run_manager_ptr,
      int run_id,
      int *run_status,
      double *max_runtime,
      int *n_concurrent_runs)
```

Arguments: on entry

`run_manager_ptr`

A previously initialized run manager object.

`run_id`

The run-manager-supplied identifier of the model run whose status is sought.

Arguments: on exit

`run_status`

The status of the model run. Options are as follows:

0	the run has not yet completed;
-90	the run was cancelled by the calling program;
-1 to -89	the run failed this number of times;
1	the run completed successfully.

`max_runtime`

The length of time that the model has been running. If concurrent runs of the same model are being undertaken, this is the runtime of the longest active run.

`n_concurrent_runs`

The number of the node on which the specified model run is currently running.

Return value

0 for success; 1 for failure.

rmic_get_run_info

Purpose

Function *rmic_get_run_info* allows a calling program to retrieve the information associated with a model run. This information includes that which is associated with a model run by the calling program.

Specifications

```
int rmic_get_run_info(RunManager *run_manager_ptr,
      int run_id,
      int *run_status,
      int *modcomid,
      char *infotext,
      int info_text_len,
      double * infovalue)
```

Arguments: on entry

`run_manager_ptr`

A previously initialized run manager object.

`run_id`

The run-manager-supplied identifier of the model run whose status is sought.

Arguments: on exit

`run_status`

The status of the model run. Options are as follows:

0	the run has not yet completed;
-90	the run was cancelled by the calling program;
-1 to -89	the run failed this number of times;

1 the run completed successfully.

`modcomind`
The index of the command used to run the model.

`infotext`
The text string associated with the model run by the calling program. Characters after the fortieth are truncated.

`infotext_len`
The length of the `infotext` character variable.

`infovalue`
A double precision number the calling program associates with the model run. The run manager does not use this information but it is stored with the run.

Return value
0 for success; 1 for failure.

rmic_get_run

Purpose
Function *rmic_get_run* retrieves the values of parameters and observations associated with a particular model run. The values of parameters may be slightly altered from those which were originally supplied to the run manager through a call to the *rmic_add_run* or *rmic_add_run_with_info* function; they are exactly equal to the values recorded on model input files.

Specifications
```
int rmic_get_run(RunManager *run_manager_ptr,
      int run_id,
      double *pardata, int npar,
      double *obsdata, int nobs)
```

Arguments: on entry
`run_manager_ptr`
A previously initialized run manager object.

`run_id`
The run-manager-supplied identifier of the run for which parameter and observation values are sought.

`npar`
The number of parameters, and hence the size of the *`pardata`* array. The value of *`npar`* supplied with this function must be the same as that supplied through the *rmic_initialize* function.

`nobs`
The number of observations, and hence the size of the `obsdata` array. The value of `nobs` supplied with this function must be the same as that supplied through the *rmic_initialize* function.

Arguments: on exit
`pardata`

The values of parameters associated with the specified model run. The ordering of parameters in this array respects that established in the previous call to the *rmic_initialize* function.

`obsdata`
The values of observation associated with the specified model run. The ordering of observations in this array respects that established in the previous call to the *rmic_initialize* function.

Return value
0 for success; 1 for failure.

rmic_get_run_with_info

Purpose
Function *rmic_get_run_with_info* returns the values of parameters and observations associated with a particular model run. In doing this, it performs the same role as the *rmic_get_run* function. However, it also returns the text string and double precision number that was associated with that model run using the *rmic_add_run_with_info* function. Note that the values of parameters that are associated with a particular model run may be slightly altered from those which were originally supplied to the run manager through a call to the *rmic_add_run* or *rmic_add_run_with_info* function; they are exactly equal to the values recorded on model input files.

Specifications
```
int rmic_get_run_with_info(RunManager *run_manager_ptr,
      int run_id,
      double *pardata, int npar,
      double *obsdata, int nobs,
      char *infotext, int infotext_len, double infovalue)
```

Arguments: on entry
`run_manager_ptr`
A previously initialized run manager object.

`run_id`
The run-manager-supplied identifier of the model run for which parameter and observation values are required.

`npar`
The number of parameters, and hence the size of the *`pardata`* array. The value of *`npar`* supplied with this function must be the same as that supplied through the *rmic_initialize* function.

`nobs`
The number of observations, and hence the size of the `obsdata` array. The value of `nobs` supplied with this function must be the same as that supplied through the *rmic_initialize* function.

`infotext_len`
Maximum length of the `infotext` character variable.

Arguments: on exit

`pardata`

The values of parameters associated with the specified model run. The ordering of parameters in this array respects that established in the previous call to the *rmic_initialize* function.

`obsdata`

The values of observations associated with the specified model run. The ordering of observations in this array respects that established in the previous call to the *rmic_initialize* function.

`infotext`

A text string supplied through function *rmic_add_run_with_info*. The run manager does not use this string; it is for the benefit of the calling program only. Although the length of the *infotext* character variable can be greater than 40 characters, this is the maximum size of the text string that function *rmic_get_run_with_info* can return.

`infovalue`

A double precision number associated with this model run supplied through function *rmic_add_run_with_info*. The run manager does not use this variable; it is stored only for the benefit of the calling program.

Return value

0 for success; 1 for failure.

rmic_get_n_failed_runs

Purpose

Function *rmic_get_n_failed_runs* returns the number of model runs that failed to execute successfully even after repeated attempts to run them.

Specifications

```
int rmic_get_failed_runs(RunManager *run_manager_ptr, int *nfail)
```

Arguments: on entry

`run_manager_ptr`

A previously initialized run manager object.

Arguments: on exit

`nfail`

The total number of runs on the model run queue that failed to successfully execute, even after repeated attempts.

Return value

0 for success; 1 for failure.

rmic_get_failed_run_ids_alloc

Purpose

Function *rmic_get_failed_run_ids_alloc* provides the integer identifiers of model runs that failed to successfully execute after repeated attempts. This function allocates memory in the *run_id_array* to store these values. The calling program is responsible for freeing this memory before it ceases execution.

Specifications

```
int rmic_get_failed_run_ids_alloc(RunManager *run_manager_ptr,
      int *run_id_array,
      int *nfail)
```

Arguments: on entry

`run_manager_ptr`

A previously initialized run manager object.

Arguments: on exit

`run_id_array`

The integer identifiers of failed model runs.

`nfail`

The total number of runs on the model run queue that failed to successfully execute, even after repeated attempts.

Return value

0 for success; 1 for failure.

rmic_get_failed_run_ids

Purpose

Function *rmic_get_failed_run_ids* provides the integer identifiers of model runs that failed to successfully execute after repeated attempts. Function *rmic_get_n_failed_runs* should be called prior to calling function *rmic_get_failed_run_ids* so that arrays can be properly dimensioned.

Specifications

```
int rmic_get_failed_run_ids(RunManager *run_manager_ptr,
      int *run_id_array,
      int nfail)
```

Arguments: on entry

`run_manager_ptr`

A previously initialized run manager object.

`nfail`

The length of the *run_id_array* array.

Arguments: on exit

`run_id_array`

The integer identifiers of failed model runs.

`nfail`

The total number of runs on the model run queue that failed to successfully execute, even after repeated attempts.

Return value

0 for success; 1 for failure.

rmic_get_n_cur_runs

Purpose

Function *rmic_get_n_cur_runs* returns the number of unique model runs currently stored in the model run queue. This number includes active runs, waiting runs, cancelled runs and failed runs. Each run is counted only once, even if multiple instances of that run are currently running or have already been run.

Specifications

```
int rmic_get_n_cur_runs(RunManager *run_manager_ptr, int *nruns)
```

Arguments: on entry

`run_manager_ptr`

A previously initialized run manager object.

Arguments: on exit

`nruns`

The number of runs stored in the PANTHER model run queue.

Return value

0 for success; 1 for failure.

rmic_get_n_total_runs

Purpose

Function *rmic_get_n_total_runs* retrieves the total number of model runs commissioned by the run manager. This does not get reset if the run manager is reinitialized.

Specifications

```
int rmic_get_n_total_runs(RunManager *run_manager_ptr,
      int *nruns)
```

Arguments: on entry

`run_manager_ptr`

A previously initialized run manager object.

Arguments: on exit

`nruns`

The number of model runs that the run manager has commissioned.

Return value

0 for success; 1 for failure.

rmic_err_msg

Purpose

If a run management function call fails, this function should be called to retrieve the text of an error message saved by the run manger.

Specifications

```
const char* rmic_err_msg()
```

Return value

The error message text string.

rmic_delete

Purpose

This function is the destructor for the run manager. It should be called only after the program which uses the run manager has finished requesting model runs so that its resources can be freed. After function *rmic_delete* has been called, the current instance of the run manger can no longer be used.

Specifications

```
int rmic_delete(RunManager *run_manager_ptr)
```

Arguments: on entry

`run_manager_ptr`

A previously initialized run manager object.

Return value

0 for success; 1 for failure.

8. Python Interface

8.1 Nomenclature

The Python interface to the PANTHER run manager is comprised of a module named *panther_pi*.

As is usual practice, the terms “parameters” and “observations” are used in the following description to denote numbers which are recorded on model input files through the agency of template files, and numbers which are read from model output files through the agency of instruction files. In practice, these numbers may have other roles (such as decision variables and management outcomes respectively). Nevertheless, use of this nomenclature simplifies the following description.

8.2 Installing panther_pi

The *panther_pi* module can be used with version 5.2.0 and later of Anacoda Python on a machine which runs 64 bit Windows as the operating system. This module was built using the Visual Studio 2015 compiler.

To install *panther_pi*, first copy the following wheel file from the PEST++ GitHub repository:

pestpp\src\panther_pi\dist\panther_pi-0.0.1-py3-none-any.whl

Then install it using pip:

```
pip install panther_pi-0.0.1-py3-none-any.whl
```

8.3 Using panther-pi

Once *panther_pi* has been installed it can be imported into a project using the command:

```
from panther_pi import *
```

This defines the class `panther` which provides a Python interface to the PANTHER run manger. The methods available in this class are presented in the following sections.

8.4 Choice of Run Manager

Before undertaking any run management, a PANTHER calling program must instantiate either the PANTHER parallel run manager or the alternative, serial run manager. Once a run manager is instantiated, run management functions can be called. These are the same, regardless of whether runs are being undertaken in serial or in parallel.

panther.create_serial

Purpose

Function *create_serial* initializes the serial run manager. Until this run manager is de-initialized, all run management requests delivered through the functions listed in the following subsection are delivered to this run manager.

Specifications

```
panther.create_serial(
        comline, tpl, inp, ins, out,
```

```
        storefile, rundir, n_max_fail)
```

Arguments: on entry

`comline`
A list holding model command lines.

`tpl`
A list holding the names of template files.

`inp`
A list holding the names of model input files that correspond to template files.

`ins`
A list holding the names of instruction files.

`out`
A list holding the names of model output files that correspond to instruction files.

`storefile`
The name of the binary file used by the run manager to store parameters and observations associated with model runs (i.e. the model run queue).

`rundir`
The folder to which the serial run manager should transfer its focus before issuing the command to run the model.

`n_max_fail`
The number of times that the run manager should attempt to re-run a failed model run before declaring it to have failed.

panther.create_panther

Purpose

Function *create_panther* initializes the PANTHER parallel run manager. Until this run manager is de-initialized, all run management requests delivered through the functions listed in the following subsection are delivered to this run manager.

Note that the PANTHER parallel run manager does not need to be informed of the names of template and instruction files, nor of model input and output files with which they are associated, because non-intrusive interaction with the model is the job of the worker.

Specifications

```
panther.create_panther(
       storefile, port, info_filename, n_max_fail,
       overdue_resched_fac, overdue_giveup_fac,
       overdue_giveup_time)
```

Arguments: on entry

`storefile`
The name of the binary file used by the run manager to store parameters and observations associated with model runs (i.e. the model run queue).

`port`
The port number which PANTHER opens for TCP/IP communication between itself and its workers.

`info_filename`
The name of the run management record file that PANTHER should write.

`n_max_fail`
The number of times that the run manager should attempt to re-run a failed model run before declaring it to have failed.

`overdue_resched_fac`
This factor is applied to the average model run time; it must be greater than 1.0. If a particular model run has been running for longer than the average model run time multiplied by this factor, then the run is re-initiated on another worker. This process is repeated up to N_MAX_FAIL times if the second (and subsequent) run(s) are similarly slow.

`overdue_giveup_fac`
If a model runs for longer than `overdue_giveup_fac` times the average model run time, and the model run queue is not empty, then the model run is terminated and recorded as having timed out. However if the run queue is empty, the model run is allowed to continue unless terminated by PANTHER in response to a request by the calling program.

`overdue_giveup_time`
If a model runs for longer than `overdue_giveup_time`, and the model run queue is not empty, then the model run is terminated and recorded as having timed out. However if the run queue is empty, the model run is allowed to continue unless terminated by PANTHER in response to a request by the calling program. This feature can be disabled by specifying a number less than or equal to zero for this argument.

8.5 Run Management Functions

panther.initialize

Purpose

The *initialize* function provides the names of parameters and observations to the run manager. These names must be fully compatible with the names of parameters featured in template files pertaining to the current problem, and with the names of observations appearing in instruction files pertaining to the current problem. The names of template and instruction files are provided to the serial run manager through the *create_serial* function. They are provided to the PESTPP-WRK worker program through its input file.

Specifications

```
panther.initialize(self, parnames, obsnames)
```

Arguments: on entry

`parnames`
A list of the ordered names of parameters whose values will be supplied or retrieved in subsequent calls to the run manager.

`obsnames`
A list of the ordered names of observations whose values are retrieved on subsequent calls to the run manager.

panther.reinitialize

Purpose

The *reinitialize* function reinitializes the run manager. The names of parameters and observations supplied in the previous call to the *initialize* function are maintained. However the model run queue is emptied, and any running incidences of the model are terminated. The run manager is then ready for re-population of the model run queue.

Specifications

```
panther.reinitialize(self)
```

panther.intialize_restart

Purpose

If the run manager was deleted, or if previous execution of a program which calls the run manager was prematurely terminated, calling of function *initialize_restart* allows run management to continue where it was previously interrupted. The restarted run manager obtains the information which it needs from the binary file in which the model run queue, together with other run management information, was stored during the previous run management session.

Specifications

```
panther.initialize_restart(self, storefile)
```

Arguments: on entry

`storefile`

The name of the binary file used by the run manager to store parameters and observations associated with model runs.

panther.add_run

Purpose

Once a run manager has been initialized, the *add_run* function can be called repeatedly. On each occasion that it is called, it adds a new run to the run queue.

Specifications

```
run_id = panther.add_run(self, pardata, modcomind)
```

Arguments: on entry

`pardata`

This list contains the values of parameters that the model must employ on this run. Parameter values must be supplied in the same order as were parameter names in the previous call to the *initialize* function.

`modcomind`

The index of the command that must be used to run the model. This number must equal or exceed unity.

Return value

`run_id`

The integer identifier assigned to this run by the run manager. This identifier is used in all future references to this model run.

panther.add_run_with_info

Purpose

Function *add_run_with_info* is similar to function *add_run* in that it adds a run to the model run queue. However, it allows the calling program to associate a text string and double precision number with the model run for informational purposes only. These can assist a calling program to track the run**.**

Specifications

```
run_id = panther.add_run_with_info(self,
         pardata, modomind, infotext, infovalue)
```

Arguments: on entry

`pardata`

This list contains the values of parameters that the model must employ on this run. Parameter values must be supplied in the same order as were parameter names in the previous call to the *initialize* function.

`modcomind`

The index of the command that must be used to run the model. This number must equal or exceed unity.

`infotext`

A text string that the calling program wishes to associate with the model run. The run manager does not use this information. It is stored with the run, and can optionally be retrieved with model results. The run manger can store a maximum of 40 characters in this run-associated string; it truncates characters supplied in `infotext` after the fortieth.

`infovalue`

A double precision number that the calling program wishes to associate with the model run. The run manager does not use this information. It is stored with the run, and can be retrieved with model results.

Return value

`run_id`

The integer identifier assigned to this run by the run manager. This identifier is used in all future references to this model run.

panther.run

Purpose

Function *run* instructs the run manager to carry out all of the runs in the model run queue, and to store observations that are generated by these runs in a binary file whose name was provided in the *create* function. Control is not returned to the calling program until all model runs have been completed.

Specifications

```
run(self)
```

panther.run_until

Purpose

The *run_until* function instructs the PANTHER parallel run manger to manage runs in the model run queue. However it may return control to the calling program before emptying the queue. Note that if the serial run manager is being used, this function has the same effect as the *run* function.

Specifications

```
return_cond = panther.run_until(self, condition, no_ops, time_sec)
```

Arguments: on entry

`condition`

Specifies when function *run_until* should return control to the calling program. Options are:

- 0 return after all model runs are complete;
- 1 return after *no_opts* consecutive run management loops have not received any TCP/IP signal from workers;
- 2 return after *time_sec* seconds have elapsed;
- 3 return after *no_opts* consecutive run management loops have not received any TCP/IP signals from workers, or after *time_sec* seconds have elapsed, whichever is sooner.

`no_ops`

The number of run management loops to undertake while listening for TCP/IP signals from workers before returning control to the calling program.

`time_sec`

The maximum time in seconds that function *rmic_run_until* will devote to managing runs before returning control to the calling program.

Return value

`return_cond`

The reason that function *run_until* is returning control to the calling program:

- 0 all runs are complete;
- 1 the *no_opts* condition was active was met;
- 2 the *time_sec* condition was active was met.

panther.cancel_run

Purpose

Removes a run from the model run queue. If the job is running, it kills the job.

Specifications

```
panther.cancel_run(self, run_id)
```

Arguments: on entry

`id`

The integer identifier for the run that was provided by the run manager when the run was added to the run queue.

panther.get_run_status

Purpose

As the name implies, function *get_run_status* informs a calling program of the status of a model run.

Specifications

```
run_status, max_runtime, n_concurrent_runs =
panther.get_run_status_info(self, run_id)
```

Arguments: on entry

`run_id`

The run-manager-supplied identifier of the model run whose status is sought.

Return values

`run_status`

The status of the model run. Options are as follows:

0	the run has not yet completed;
-90	the run was cancelled by the calling program;
-1 to -89	the run failed this number of times;
1	the run completed successfully.

`max_runtime`

The length of time that the model has been running. If concurrent runs of the same model are being undertaken, this is the runtime of the longest active run.

`n_concurrent_runs`

The number of the node on which the specified model run is currently running.

panther.get_run_info

Purpose

Function *get_run_info* allows a calling program to retrieve the information associated with a model run. This information includes that which is associated with a model run by the calling program.

Specifications

```
panther.get_run_info(self, run_id)
```

Arguments: on entry

`run_id`

The run-manager-supplied identifier of the model run whose status is sought.

Return values

Function *get_run_info* returns four values. Its calling syntax is:

```
run_status, modcomind, infotext, infovalue
= panther.get_run_info(self, run_id)
```

`run_status`

The status of the model run. Options are as follows:

0	the run has not yet completed;
-90	the run was cancelled by the calling program;
-1 to -89	the run failed this number of times;

1 the run completed successfully.

`modcomind`
The index of the command used to run the model.

`infotext`
The text string associated with the model run by the calling program. Characters after the fortieth are truncated.

`infovalue`
A double precision number which the calling program associates with the model run. The run manager does not use this information; however it is stored with the run.

panther.get_run

Purpose
Function *get_run* retrieves the values of parameters and observations associated with a particular model run. The values of parameters may be slightly altered from those which were originally supplied to the run manager through calls to the *add_run* or *add_run_with_info* function; they are exactly equal to the values recorded on model input files.

Specifications
```
pardata, obsedata = panther.get_run(self, run_id)
```

Arguments: on entry
`run_id`
The run-manager-supplied identifier of the run for which parameter and observation values are sought.

Return values
`pardata`

The values of parameters associated with the specified model run. The ordering of parameters in this array respects that established in the previous call to the *initialize* function.

`obsdata`
The values of observation associated with the specified model run. The ordering of observations in this array respects that established in the previous call to the *initialize* function.

panther.get_run_with_info

Purpose
Function *get_run_with_info* returns the values of parameters and observations associated with a particular model run. In doing this, it performs the same role as the *get_run* function. However, it also returns the text string and double precision number that was associated with that model run using the *add_run_with_info* function. Note that the values of parameters that are associated with a particular model run may be slightly altered from those which were originally supplied to the run manager through a call to the *add_run* or *add_run_with_info* function; they are exactly equal to the values recorded on model input files.

Specifications
```
pardata, obsdata, infotext, infovalue
```

```
= panther.get_run_with_info(self, run_id)
```

Arguments: on entry

`run_id`

The run-manager-supplied identifier of the model run for which parameter and observation values are required.

Return values

`pardata`

The values of parameters associated with the specified model run. The ordering of parameters in this array respects that established in the previous call to the *initialize* function.

`obsdata`

The values of observation associated with the specified model run. The ordering of observations in this array respects that established in the previous call to *initialize* function.

`infotext`

A text string supplied through function *add_run_with_info*. The run manager does not use this string; it is for the benefit of the calling program only. Although the length of the *infotext* character variable can be greater than 40 characters, this is the maximum size of the text string that function *get_run_with_info* can return.

`infovalue`

A double precision number associated with this model run supplied through function *add_run_with_info*. The run manager does not use this variable; it is stored only for the benefit of the calling program.

panther.get_n_failed_runs

Purpose

Function *get_n_failed_runs* returns the number of model runs that failed to execute successfully even after repeated attempts to run them.

Specifications

```
nfail = panther.get_num_failed_runs(self)
```

Return value

`nfail`

The total number of runs on the model run queue that failed to successfully execute, even after repeated attempts.

panther.get_failed_run_ids

Purpose

Function *get_failed_run_ids* provides the integer identifiers of model runs that failed to successfully execute after repeated attempts.

Specifications

```
run_id_list = panther.get_failed_run_ids(self)
```

Return value

`run_id_list`

A list with the integer identifiers of failed model runs.

panther.get_n_cur_runs

Purpose

Function *get_n_cur_runs* returns the number of unique model runs currently stored in the model run queue. This number includes active runs, waiting runs, cancelled runs and failed runs. Each run is counted only once, even if multiple instances of that run are currently running or have already been run.

Specifications

```
nruns = panther.get_n_cur_runs(self)
```

Return value

`nruns`

The number of runs stored in the PANTHER model run queue.

panther.get_n_total_runs

Purpose

Function *get_n_total_runs* retrieves the total number of model runs commissioned by the run manager. This does not get reset if the run manager is reinitialized.

Specifications

```
nruns = panther.get_total_runs(self)
```

Return value

`nruns`

The number of model runs that the run manager has commissioned.

9. References

Doherty, J., 2018a. PEST: Model-Independent Parameter Estimation. Part A of PEST Manual. Watermark Numerical Computing, Brisbane.

Doherty, J., 2018b. PEST: Model-Independent Parameter Estimation. Part B of PEST Manual. Watermark Numerical Computing, Brisbane.

Fienen, M.N. and Hunt, R.J., 2015. High-throughput computing versus high-performance computing for groundwater applications. *Groundwater*, 53(2), 180-184.

Hunt, R.J., Luchette, J., Shreuder, W.A., Rumbaugh, J., Doherty, J., Tonkin, M.J. and Rumbaugh, D., 2010. Using the cloud to replenish parched groundwater modeling efforts. Rapid Communication for *Groundwater*, 48(3) doi: 10.1111/j.1745-6584.2010.00699

Schumacher, J., Hayley, K., Boutin, L-C. and White, E., 2018. PPAPI: A program for groundwater modelling tasks in distributed parallel computing environments. *Groundwater*, 56(2) 248-250.

Welter, D.E., White, J.T., Hunt, R.J., and Doherty, J.E., 2015. Approaches to Highly Parameterized Inversion: PEST++ Version 3, a Parameter ESTimation and Uncertainty Analysis Software Suite Optimized for Large Environmental Models. United States Geological Survey Techniques and Methods 7-C12.

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., Welter, D. and Doherty, J., 2018. Manual for Version 4 of PEST++. *Downloadable from http://www.pesthomepage.org*