Title: PDQ Users Manual Manual Version 2 for PDQ Code Version 1.20

Author(s): Pelak, Robert A.
McDermott, Danielle Marie

Intended for: Report

Issued: 2022-06-24

Los Alamos NATIONAL LABORATORY

Technical Report

XTD-NTA Nuclear Threat Assessment

LA-UR-22-

PDQ Users Manual
Manual Version 2

for PDQ Code Version 1.20

June 22, 2022

Robert A. Pelak, XTD-NTA, MS T082
Danielle McDermott, XTD-SS, MS T082

1 Introduction

PDQ is a tool for the management of the input and execution of batch jobs for simulation codes that use a text based input system. It accomplishes this goal by operating at two levels. First, it takes input file templates (commonly known at LANL as input deck templates) and creates multiple instantiations by performing substitutions of data from table files into symbols (variables) found in the template. Second, it provides commands to submit the created files to the SLURM batch system for execution. These two activities taken together produce a whole that is greater than the sum of its parts and provides an elegant way of executing studies across multiple similar simulations while minimizing the risk of typographical errors in the input files.

PDQ was originally developed as a job management system called XVS by Jeff McAninch while he was at LANL. Besides the capabilities described here, XVS had many other features specific for interactions with particular simulation codes. After Jeff's departure, maintenance of XVS was taken over by Rendell Carver; he added some new features as well as kept it functioning as the batch system at LANL was changed from LSF to MOAB to SLURM. In 2017, Rob Pelak decided to develop a different version that removed the additional features (many of which were rendered obsolete with the retirement of the simulation code or batch system that they supported) and produced a cleaner "bare bones" version of XVS. A few other behaviors of XVS that Rob found irksome were altered. Rob gave the resulting code a new name: PDQ.

In 2022 Danielle McDermott developed a version that runs under Python 3.X. As suggested by Rob, she used the python2to3 utility to identify most changes. Given that PDQ continues to operate with Python version 2.7 we have advanced the version number to 1.20.

2 A Partial Example

The key feature that makes PDQ useful is that it can take values from a table in a file and substitute them into an input deck template to make multiple versions of the deck while placing them in separate directories.

The following example illustrates this key capability as well as PDQ's capacity for submitting batch jobs. Full details for all of PDQ's capabilities will be discussed in the following sections.

We start with an example of a greatly simplified input deck:

```
#  Example input deck
sim_name    =  'trial1'
method      =  'old'
variable1   =   1.0
variable2   =   7.5
variable3   =   8.5
```

Suppose that it was desired to run this simulation with not just the settings given above but over several other combinations as well. Conventional practice would be to repeatedly copy the input deck and make the changes by hand. This opens the user up to possibly making typographical errors or simply forgetting to make a desired change when the deck is much more complex than this example.

PDQ simplifies this task by allowing the input deck to be turned into a template and values to be plugged in from a data file. To create a template, the text that is to undergo substitution is placed in curly braces and given a symbol name. Then a table of values using the symbol name is provided in another file named `design.txt`. So, converting the above deck to a template would look like this:

```
#  Example input deck
sim_name   =  '{run_ID}'
method     =  'old'
variable1  =   {v1}
variable2  =   {v2}
variable3  =   {v3}
```

And a `design.txt` file could look like this.

```
# Table of run values
!run_ID,   v1,   v2,   v3
trial1,   1.0,  7.5,  8.5
trial2,   3.0,  6.0,  9.0
trialX,   1.1,  2.5.  3.6
```

When PDQ is invoked, three input decks are created, each located in a separate directory with the name give in the `run_ID` column. (Note, the exclamation point in that row indicates to PDQ that the row contains column header names and not values for substitution). Again, for example, the one created in the trialX directory would look like:

```
#  Example input deck
sim_name   =  'trialX'
method     =  'old'
variable1  =   1.1
variable2  =   2.5
variable3  =   3.6
```

So far, so good, but how does a user actually accomplish this?

PDQ assumes a specific arrangement of directories in order to function. PDQ is run from a directory that is termed the study directory. Within the study directory there must be two subdirectories, one called `runs` and another called `setup`. Within the `setup` directory is placed the `design.txt` file along with the input deck template file - it can be named anything except for `design.txt` and a few other reserved names that will be discussed later.

PDQ is then invoked with the following command issued from within the study directory:

```
>>> PDQ.py setup
```

(This assumes that the location of the file PDQ.py can be found in the user's search path.) When it is run, this command will create a new directory in the `runs` directory for each entry in the `run_ID` column found in the `design.txt` file. Within each of these run directories will be found a `design.txt` file that contains the parameter settings for that run as well as a copy of the input deck file with all of the substitutions completed. Any other files found in the `setup` directory (with a few exceptions noted later) will be copied into the run directory with substitutions performed in the same manner. If one of these files is named `sbatch.input` and contains commands for a SLURM job submission, all of the jobs can be sent to the batch system with the command:

```
>>> PDQ.py submit
```

3 PDQ Reference

3.1 Directories and Files

As mentioned in Section 2, PDQ relies on a certain directory structure in which to operate. It assumes that it is invoked from a directory (known as the study directory) that has two specific subdirectories; one named `setup` and one named `runs`. PDQ operates on all files in the `setup` directory (with very few exceptions) and creates files in new subdirectories of the `runs` directory. The names of these subdirectories are drawn from the `run_ID` column of the `design.txt` file. The actions taken on those files are determined by the instructions and symbols contained within the file and other files; if no instructions or symbols are present, the file is simply copied over to the subdirectory verbatim.

There are four special files that can be placed into the `setup` directory:

3.1.1 `configure.txt`

This file is required to be present in the setup directory even if it is empty. It is intended to hold instructions for PDQ (made using the instructions listed in section 3.2.2) that configure its operation on subsequent files, and is therefore read in and acted upon first when PDQ operates on files. Unlike other files, it is not transcribed into a run directory.

3.1.2 `design.txt`

Unlike all other files put into the `setup` directory the `design.txt` file is not processed by performing substitutions for its symbols or actions specified by instructions. Instead, it provides names of symbols and tables of values for substitution into other files. The format of the table is that of comma separated values. The first column must be labeled `!run_ID`. It provides the names for the subdirectories (runs) that will be created in the `runs` directory. The exclamation point indicates that this row holds names of symbols rather than values. Comment lines are allowed in this file. No instructions may be used in this file. The file is partially transcribed into the run directory; only the symbols and their values for the particular run are placed in the `design.txt` file written in the particular run directory.

3.1.3 `sbatch.input` and `restart.input`

These files are processed in an identical manner as that of any other file found in the `setup` directory (except for the `configure.txt` and `design.txt` files). They undergo symbol substitution and any instructions are operated upon. However, they are special in that they provide the instruction scripts sent to SLURM when the `submit` or `restart` commands are issued, respectively. Also, only the `sbatch.input` file is processed when a `setup` command is issued (the `restart.input` file is ignored) and vice versa when the `setrestart` command is issued.

3.2 Symbols and Instructions

3.2.1 Symbols

Symbols can be thought of as variables; they designate places in template files where a value substitution will take place. They are identified in files by simply placing the symbol name within curly braces. A symbol is built from any combination of alphabetic characters (both upper and lower case), numerical digits or the characters _ (underscore), - (hyphen), . (period) or ~ (tilde).

Values for symbols can be assigned in two ways. First, symbols are created in the table found in a `design.txt` file. The values of the symbols created this way will vary with `run_ID` and be substituted accordingly. The second way is through use of an instruction.

Symbol values are initially handled by the PDQ program as strings. PDQ will then attempt to interpret the symbol value as a float. If it is successful, it will treat it as a float; if not it will next try to interpret it as an int. If successful, it will treat the value as an int; if not it will treat it as a string. This can create problems: leading zeroes in strings that can be interpreted as floats or ints will be discarded, possibly thwarting the symbol's use in a filename. One exception to this general rule is that symbols that have the string `ID` in them will be treated as strings regardless of its content.

Symbols can be nested one inside another; if the symbol `v1` is set to `cd` and the symbol `abcd` is set to 3.14159, the expression `{ab{v1}}` will return 3.14159. PDQ makes only one pass through the files it operates on so symbols must be assigned values prior to their use.

3.2.2 Instructions

Instructions tell PDQ to perform operations. They are distinguished from other portions of the file by being set in curl braces.

`default` *symbol value*
`define` *symbol value*
These two instructions allow the assignment of *value* to *symbol*. `define` takes precedence over `default` meaning that if a symbol is assigned a value first by a `define` instruction and then the same symbol is assigned a value by a `default` instruction, the `default` instruction will be ignored and the value assigned by the `define` instruction is retained. `define` instructions can supersede `default` instructions as well as previous `define` instructions; `default` instructions can not be superseded by other `default` instructions.

Note that either of these instructions causes a message to be left in the output file by the PDQ parser. If this is a problem, either make sure that the appropriate output comment character is set (see Section 3.2.3) or issue the instruction in another file such as the `config.txt` file.

`python` *expression*
This instruction allows *expression* to be evaluated as if it were a piece of python code. This is accomplished by executing a statement of the form `exec value =` *expression*. The primary use for this instruction is to allow the implementation of mathematical formulas that depend on symbols. For instance, a statement of the form

```
variable3 = {python {v1} + {v2}}
```

would be processed to produce (if the symbols `v1` and `v2` were set to 1.0 and 7.5):

```
variable3 = 8.5
```

In this way, the above statement could be used to replace the `v3` symbol in the example of Section 2. Users need to be careful about variable types for symbols and pay attention to the guidelines presented in Section 3.2.1.

`file` *filename*
This instruction allows the importation of the file given by *filename* into the current file. The contents of *filename* are processed as if they were part of the original file.

`field` *column row filename*
This instruction allows values from tables in other files be accessed. The file must be formatted in the same manner as that in the `design`.txt file.

`format` *format expression*
This instruction causes the value returned by *expression* to be output according to the python *format* expression.

`replace` *str1 str2 target*
This instruction generates an output where occurrences of *str1* are replaced with *str2* within *target*.

`set_jobDepend`
This instruction adds lines of the form

```
#SBATCH --dependency=value
#SBATCH --job-name=newname
```

to the output file for the purpose of constructing restart batch scripts (typically these files are named `restart.input`. The quantity *value* is set to be the job identification number for the most recently submitted job for this simulation (found in the `sbatch.jobid` file created in the run directory). The quantity *newname* is constructed from the original job name with the suffix 'ResX' attached where 'X' is a number indicating the current restart number.

`set_jobDepend0`
This instruction does what `set_jobDepend` does but adds two additional lines for directing job output:

```
#SBATCH --output=newoutfilename
#SBATCH --error=newerrfilename
```

where `newoutfilename` is a file name (with 'ResX' suffix) to which data sent by SLURM to standard output is directed. Similarly, `newerrfilename` is a file name (with 'ResX' suffix) to which data sent by SLURM to standard error is directed.

echo_symbol_table
This instruction causes the internal PDQ symbol table to be written out. This is made available for debugging purposes.

3.2.3 Pre-defined Symbols

When PDQ is run, regardless of the command issued, a number of symbols are set so as to be available to the user. For the most part, the values of these symbols should not be changed by the user. They are present so that the values they hold can be accessed for use in files such as `sbatch.input` or `restart.input`. Exceptions are the symbols `source_comment_char` and `object_comment_char` which are provided to the user so that they can be changed in accordance with the simulation codes that they are using.

source_comment_char and object_comment_char When a file is processed by PDQ, the value of the symbol `source_comment_char` is replaced with `object_comment_char` and the rest of the input line following `source_comment_char` is not acted upon but is transcribed. This substitution takes place after other symbol substitutions are complete so be very careful if the value of `source_comment_char` occurs in the value of a symbol. The default value for both of these symbols is a hash, `#`.

Another important note about comments: Lines that begin with two or more hash characters, `##`, will not be operated on nor transcribed into an output file. This behavior is not affected by the settings of `source_comment_char` or `object_comment_char`. At this time, this behavior is unalterable.

run_dir and run_root These symbols hold the path to the `runs` directory and that of the directory immediately above the `runs` directory.

setup_dir This symbol holds the path to the `setup` directory.

study_dir, study_root and study_name These symbols hold the path to the `study` directory (the one in which the `setup` and `runs` directory are found), the path to the directory immediately above the `study` directory, and the name of the `study` directory.

run_ID, run_ID_list These symbols hold the current `run_ID` from the `design.txt` file while `run_ID_list` holds a list of all of the valid `run_ID`s that will or have been processed during the execution of the current command.

jobname Reserved for internal use by PDQ. It is a composite of the study name and the value of the `run_ID` symbol.

3.3 The PDQ Command Line

The previous sections document the symbols and instructions that are part of the contents of the files that PDQ acts upon. They are required to be issued from the study directory (the directory in which the `runs` and `setup` directories are found). The command line commands are used to initiate the file processing and job submission. The format of the commands are:

`PDQ.py` *command [options] [runlist]*

Options exist for only two of the commands `setup` and `kill`; they will be detailed with those commands. For all of the commands, the *runlist* is a list of the names of runs as given in the `design.txt` file that are to be operated upon. If no runlist is given, all of the names in the `run_ID` column in the `design.txt` file are entered into the *runlist*.

3.3.1 `setup`

This command causes new directories for each entry in the runlist to be created within the `runs` directory. If a directory for an entry in the run list already exists, an error occurs and execution is terminated. After the directories are made, the files in the `setup` directory are acted upon, first the file `config.txt`, then the file `design.txt` and then all other files in alphabetic order with the exception that the file `restart.input` is ignored.

There are two options available to this command. `--rm` will cause any run directories corresponding to names in the runlist to be removed and their contents deleted prior to creation of new directories. `--ow` will allow execution to proceed even if a directory corresponding to an entry in the runlist is present but will overwrite any files in that directory as needed to complete the transcription process.

3.3.2 `submit`

Issuance of this command results in the submission of an `sbatch.input` file to the SLURM `squeue` command for each entry in the runlist, thereby performing a job submission for each job in the runlist. The job name and numerical ID are stored in a file named `sbatch.jobid` for use by other PDQ commands.

3.3.3 `setrestart`

This command performs a transcription of the file `restart.input` after processing first the `configure.txt` and then the `design.txt` file. This action is performed for each entry in the runlist. All other files in the `setup` directory are ignored.

3.3.4 `restart`

Issuance of this command results in the submission of a `restart.input` file to the SLURM `squeue` command for each entry in the runlist, thereby performing a restart job submission for each job in the runlist. The job name and numerical ID are stored in a file named `sbatch.jobid` for use by other PDQ commands.

3.3.5 `kill`

This command causes a SLURM `scancel` command to be issued for each job (running or pending) for each entry in the runlist. If the `--si` option is issued, a `--signal=23` option is added to the `scancel` command.

3.3.6 `help`

This command displays a short, not-particularly-helpful message (especially if you are already reading this document) and terminates execution of the code. The same behavior can be triggered with a `-h` or a `--help` option on the command line regardless of what command is present.

4 Manual Revision History

4.1 Manual Version 1 (08/14/2020)

- Initial release for code version 1.12.

4.2 Manual Version 2 (06/22/2022)

- Release for code version 1.20

5 Code Revision History

5.1 Code Version 1.00 (05/19/2020)

- Initial release.

5.2 Code Version 1.10 (06/15/2020)

- Renamed some of the python source files so that `PDQ.py` is the one called by users.
- Removed `--de` debug option.
- Added welcome message, pared down help message.
- Corrected copyright holder to Triad National Security.
- Fixed bug with `--si` flag.
- Eliminated options for alternate specifications for some files.
- Changed command line option prefix from single dash to double dash.
- Changed headers in `design.txt` files written to run directories.

5.3 Code Version 1.11 (07/29/2020)

- Fixed typographic error in job_status_run function in `PDQstudy.py`.
- Fixed typographic error in get_table function in `PDQtables.py`
- Overhauled get_table function in `PDQtables.py` to eliminate all vestigial references to different table types.
- Eliminated functions id_header_line, str_table and write_table as well as global variables table_types and status_keys from `PDQtables.py` because they are no longer used.
- Removed exception raising from `PDQtables.py` and `PDQparse.py`; replace it with a print statement and sys.exit() call.
- Removed 'abbreviation' tests from make_jobname function in `PDQstudy.py`.

5.4 Code Version 1.12 (08/14/2020)

- Changed overlooked XVS references to PDQ in error messages.
- Added python version information to welcome message.

5.5 Code Version 1.13 (03/22/2022)

- Added hyphen and period as acceptable characters for use in runIDs; prohibit hyphen and period from being the first character and prohibit hyphen from being the last character.

5.6 Code Version 1.20 (06/22/2022)

- Works with either Python 2 or Python 3