LA-UR-17-27588
Approved for public release; distribution is unlimited.

Title: Duplicating MC-15 Output with Python and MCNP

Author(s): McSpaden, Alexander Thomas

Intended for: Report

Issued: 2018-07-11 (rev.1)

Duplicating MC-15 Output With Python and MCNP

Alex McSpaden

July 9, 2018

Abstract

Two Python scripts have been written that process the output files of MCNP6 into a format that mimics the list-mode output of Los Alamos National Laboratory's MC-15, NoMAD, and NPOD neutron detection systems. This report details the methods implemented in these scripts and instructions on their use.

1 Introduction

The MC-15, NoMAD and NPOD neutron detection systems all consist of a series of 15 ^{3}He tubes, arranged in rows and embedded in a matrix of high-density polyethylene (HDPE). One of the outputs of a measurement with these detection systems is an `.lmx` file, which stores the time a capture interaction happens and in which tube the neutron was captured. Such a file allows for analysis using correlated neutron methods, such as determining singles and doubles count rates.

Being able to replicate such a file from simulations would allow for a much more direct comparison between results of the two, as the same processing methods could then be used on each. To this end, two Python scripts have been written that run [1]MCNP6®[1] and process the particle track output file to create the same `.lmx` file that would be output by the detector system. These scripts are intended for use on computing clusters that use the Slurm[2] job management system, and takes advantage of parallel computing to finish the computations faster.

2 Theory and Methods

To accomplish the task of running MCNP and automatically creating an `.lmx` file, a Python script called `multiLMX.py` was written. This script submits two jobs to the Slurm resource management system: one to run MCNP and another to do the particle track processing once the simulation finishes. If multiple MCNP cases need to be run to get the proper number of total histories simulated, the script creates sub-directories for each (named `case001-caseXXX`), assigns a new random number seed to each of these cases, and submits jobs for these simulations to be run inside those sub-directories. The final particle track processing job uses another Python script called `ptracLMX.py`, which reads in the necessary information on the measurement time from the MCNP `outp` file, and all of the particle track information from the `ptrac` file. A flow chart depicting a high-level overview of the process is shown in Fig. 1.

When MCNP is configured to produce the `ptrac` output file, it stores, among other data, the time an event happened and the cell in which that event occurred. In order to create the `.lmx` file this information needs to be harvested from the file, rearranged in order of time, and combined into certain time bins. To do this, the MCNPTools[3] package was used to pull out the information for each of the events, as the package contains a Python module providing a number of functions for extracting information from the ptrac file. For the purposes of this script, these functions were used to create a list of all the times and cells that capture events occurred in, which was then sorted by time in ascending order.

[1]MCNP and Monte Carlo N-Particle are registered trademarks owned by Los Alamos National Security, LLC, manager and operator of Los Alamos National Laboratory. Any third party use of such registered trademarks should be properly attributed to Los Alamos National Security, LLC, including the use of the designation as appropriate. For the purposes of visual clarity, the registered trademark symbol is assumed for all references to MCNP within the remainder of this paper.

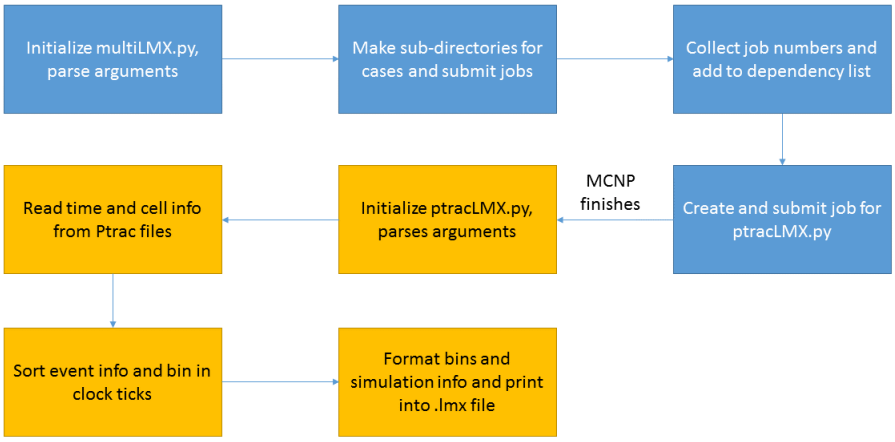

Figure 1: An overview of the process for running MCNP and making an .lmx file from the output. Blue boxes represent processes in `multiLMX.py`, while yellow boxes represent `ptracLMX.py`.

The MC-15 records events in "ticks" in time, meaning that in the output, the events are lumped into consecutive time bins of (typically) 100 or 128 nanosecond width. For example if one tube is hit at 46 ns, another at 80 ns, and a third at 93 ns, the tick containing those times will record 3 counts. Because of this, the previously made list of all of the event times needs to be lumped into these ticks. Additionally, each tube in the system has an associated dead time after an interaction in which it is incapable of having another. To account for this, the event list was iterated through after sorting by time to make sure no events happened too close in time to another in the same tube. An NPOD also operates on this tick system, but with a tick length of one microsecond and a tube dead time of four microseconds, whereas the MC-15 dead time is typically 1000 nanoseconds. The NoMAD is functionally similar to the MC-15, so the operation and analysis are similar.

3 Usage Instructions

3.1 multiLMX.py

Running `multiLMX.py` requires that `ptracLMX.py` and the input file that is to be run exist within the same folder. Additionally, the `rand` card has to be included in the data section of the MCNP input deck. When `multiLMX.py` is run, this line in the input file will be replaced by a card telling MCNP to use L'Ecuyer 63-bit generator number 1 with a randomly assigned seed, so that the simulation cases will not be duplicates of each other.

On the command line the script has three required and five optional arguments. The three required arguments are for the name of the input file, the number of cases to be run, and the version of MCNP to be used. The optional arguments are the time each case will take on the cluster, the dead time of the detector system, the tick length to be used for analysis, the type of detector system (NPOD, NoMAD, or MC-15), and whether `ptracLMX.py` should treat the detectors as linked or separated. A description of these command line arguments is depicted in Table 1. To specify which version of MCNP is to be run, the user types either 6.1.1 or 6.2, but this is based on what options are currently usable, and can easily be adapted to fit either updates in MCNP or the versions available. These tell the program to load the 6.1.1beta and 6.2 modules respectively. For the time that each case will run, the format that must be used is HH:MM:SS. An example for a simulation to be analyzed in separated mode using the input file `input.i` that will use version 6.1.1 in ten cases, each of which will take less than three hours is

Table 1: The optional command line arguments used for multiLMX.py

Argument	Usage	Default Value
`--time`	The time each case will take on the cluster	08:00:00
`--deadtime`	The dead time of a tube in nanoseconds	1000
`--tickLength`	The length of each time tick in nanoseconds	100
`--type`	The type of detector, either 'MC15' or 'NPOD'	'MC15'
`--linked`	If two units, either 1 for a linked or 0 for a separated measurement	1 (linked)

```
python multiLMX.py input.i 10 6.1.1 --time 03:00:00 --linked 0
```

In order for the script to function, the Python module must be loaded, enabling the use of Python 3.5 features and modules such as NumPy. This can be done with the command

```
module load python
```

Also, the NPS listed in the MCNP input file must be the total desired number of histories divided by the number of cases. For example if 10 million histories need to be run over all the cases, ten of them could be run with the input file specifying an NPS of 1 million. Running the script multiple times within the same directory will lead to issues, therefore the directory should be cleaned of any products of a previous run (e.g. case folders) before a new run can be done.

If the user needs to run the scripts on a system that does not use Slurm, the blocks of statements that write the Slurm file will need to be modified to fit the standards of whatever resource management system is being used. If the particular system in use does not support jobs being dependent, meaning that a job is not entered into the queue until other jobs are finished, then `multiLMX.py` will need to be further modified to not submit the final processing job, and `ptracLMX.py` will have to be run manually. No other modification to `multiLMX.py` or `ptracLMX.py` needs to be done.

3.2 ptracLMX.py

3.2.1 Operating Requirements and Basic Function

For processing the ptrac files produced once the MCNP runs are finished, a job runs executing a Python script that manufactures the final `.lmx` file. In order for this script to work, MCNP must be configured to produce the ptrac file, with the events recorded filtered down to only capture events in the active volume of the tubes. This script can either process a single ptrac file or multiple if the measurement has too many histories to fit into the time limit of a node.

While `multiLMX.py` handles the submission of the processing job automatically, this script can be run individually if desired. To do so, the Python and MCNPTools modules must be loaded. Part of the MCNPTools module uses the GCC compilers, so if Intel or other compilers are already loaded for another application, this will lead to a conflict error keeping the compiler from being loaded. To address this issue, either purge the modules that are loaded or set your computing environment to ignore this conflict. On Los Alamos National Laboratory's HPC clusters, loading these modules and running the script can be done with the following commands.

```
module load python
module use /usr/projects/mcnp/modules
module load mcnptools

python ptracLMX.py <any command line arguments>
```

Since this script is meant to be part of the `multiLMX.py` framework, the files it processes are assumed to be similar to those that would be made from `multiLMX.py` runs. That means the MCNP inputs and output are in separate sub-directories called `case001-caseXXX`, and the output file within those directories are called `outp` and the particle track files are called `ptrac`. However, the latest version of this script allows

Table 2: Command line arguments used in ptracLMX.py

Argument	Usage	Default Value
`--numCases`	The number of cases the MCNP run was split into	1
`--deadtime`	The dead time of an individual He-3 tube in nanoseconds	1000
`--tickLength`	The length of each time tick in the detector in nanoseconds	100
`--type`	The type of detector, either `'NOMAD'`, `'MC15'` or `'NPOD'`	`'NOMAD'`
`--linked`	If two units, either 1 for a linked or 0 for a separated measurement.	1 (linked)
`--offset`	The amount of time to skip at the beginning of a simulation in seconds.	0
`--outp`	The name of the MCNP output file.	`'outp'`
`--ptrac`	The name of the MCNP Ptrac file.	`'ptrac'`
`--parFile`	The name of the parameter file, if used.	`'ptracLMXparams'`

for different names for these files and folders. Additionally, the Python dictionary in the script containing the cell numbers of the ^{3}He active region must be changed to match the input file used for the simulations. Failing to do so will result in a fatal error.

The final product of this script is an `.lmx` file. The header of these files contains text detailing the total count rate, row ratios, measurement time, and other information pertinent to the measurement that was either performed or simulated. Following the text header is binary data representing the time of an interaction in nanoseconds, and which tubes were hit in each time tick. The name of this file indicates the time and date that it was produced. For example, an .lmx file made at 2:51:30 pm on May 25th, 2017 would be called `2017_05_25_145130.lmx` with the time being in the 24-hour clock format. If measurements are performed in separated mode, two `.lmx` files will be made that can be identified by a trailing number on the time stamp. For the previous example, these files would be `2017_05_25_145130_1.lmx` and `2017_05_25_145130_2.lmx` for detectors 1 and 2.

3.2.2 Command Line Arguments

The script has multiple command line arguments to handle various parameters of the detector setup. These arguments include aspects such as detector dead time, tick length, detector type, and if two detectors are used, whether they were linked or separated. Details of the command line arguments and what they represent are shown in Table 2. All arguments are optional, and omitting one causes the default value to be used.

If multiple detectors were used in a measurement, the `--linked` argument decides whether the script should output one .lmx file for a linked measurement, or two .lmx files where the events are separated based on the detector they occur in and binned individually. To separate the events in the two detectors, the script looks at the coordinates of events, and sees if there are coordinates that differ by more than 40 centimeters. If so, the script stores the first two values found that occur in separate detectors, and subsequent events are sorted according to which reference coordinate they occur closest to. This means that if the corresponding tubes in the detectors are less than 40 centimeters apart, the script cannot differentiate between them. If only one detector was used in a measurement, this argument should be omitted and the default value of 1 will properly handle the binning. An example for a run split over ten cases with two linked NPODs with a dead time of four microseconds and a tick length of one microsecond follows.

```
python ptracLMX.py --numCases 10 --deadtime 4000 --tickLength 1000 --type NPOD --linked 1
```

The offset feature was introduced in version 1.1 to skip a certain amount of time at the beginning of the simulation. This is meant to better replicate a real measurement, as it is not very common for a neutron source to start at time zero of a detector run. This could lead to lower than expected count rates at the start of a simulation, since there has been no chance for source neutrons to multiply and build up to the steady state before the detector has started. When using this feature it is important to run your simulation for a longer time period than the desired detection window. For example, if a ten second offset is desired and 300 seconds of counting time is needed, run the MCNP simulation for 310 seconds.

Table 3: The required components of the parameter file

Parameter Name	Description
`numCases`	The number of simulation cases to analyze.
`deadtime`	The dead time of a detector tube in nanoseconds.
`tickLength`	The tick length of the detector system in nanoseconds.
`detType`	The type of detector (`'NPOD'`, `'NOMAD'`, or `'MC15'`).
`linked`	If multiple detectors, whether the analysis should be linked or separated. Separated creates a different .lmx file for each detector. Enter 0 for separate, 1 for linked.
`offset`	The amount of time to be skipped at the beginning of the simulation.
`tubeCell`	The dictionary for converting the simulation cell number to the tube number of the detector.
`outp`	The desired names for the outp and Ptrac files (default is `'outp'` and `'ptrac'`).
`ptrac`	
`defaultDirs`	In a multiple case run, whether the script-default names of the subdirectories should be used (e.g. `case001-caseXXX`). True/False.
`dirNames` (only if `defaultDirs` is False)	If the subdirectories are not using the default names, a list of strings representing their names. Example: `dirNames = ['meas1', 'meas2', 'meas3']`
`combPtrac`	If a multiple case run, whether or not the data in the Ptrac files should be combined into a single .lmx file. If the data is not to be combined, a separate .lmx file will be made for each subdirectory. If a single case run, must be True. True/False.

3.2.3 Parameter File

Instead of using command line arguments to specify the arguments, the user can now use a parameter file. If a parameter file is specified, anything input into the command line will be overridden. The required components are listed in Table 3. If not all of the components are present the file will be ignored and the command line used instead. The use of this file allows for more customization of the run than if the command line arguments are used. An example of a parameter file is listed in Appendix C.

4 Validation

4.1 Testing with MC-15/NoMAD

Measurements of the Beryllium-Reflected Plutonium Ball (BeRP Ball) were performed with a single MC-15 at a distance of 50 centimeters from the center of the ball to the front face of the detector. The BeRP ball is a sphere of mostly ^{239}Pu and ~6% ^{240}Pu that has been commonly used for many subcritical experiments and benchmarks, in which more information on the characteristics of the source can be found[4, 5, 6, 7]. The measurement was five minutes long, so the input files were made to simulate the appropriate number of spontaneous fissions and (α, n) reactions matching this time period. The simulation model is intended to be detailed, and includes the MC-15, the ball, cladding, stand, the cart that they rest on, and the walls and floor of the room.

Table 4: Comparison of MC-15 simulation results with measured data

Method	R_1	Difference from Measured (%)	R_2	Difference from Measured %
Measured	8519.23 ± 4.60	-	1440.64 ± 14.38	-
`multiLMX.py`	8494.51 ± 6.50	0.3	1455.42 ± 16.92	1.0

To run the simulations, the required number of histories was split into 20 cases in order for them to finish relatively quickly on the computing cluster. Once these jobs were completed, `ptracLMX.py` was run to process the detector interactions into an `.lmx` file. For this simulation, the processing took approximately a minute and a half to run.

When the `.lmx` file was created, it was processed by Momentum, a Los Alamos National Laboratory neutron multiplicity analysis code[8]. This code uses Feynman histograms and their moments to compute the fit for singles and doubles rates in the detectors, with the uncertainty of that fit based on a covariance matrix. These results used the random binning method, where a time bin of a certain length was placed randomly along the event timeline, and the number of events within that timeline counted, and the appropriate bin in the histogram incremented[9]. The results of the simulation and processing can be seen in Table 3, where it is compared to the measured result. The table shows that the scripts are quite capable of properly analyzing the data from MCNP, as both the simulated singles and doubles rates differ from the measured results by a percent or less.

4.2 Testing with NPOD

A second test case was to replicate the results for a benchmark experiment in which 2 NPODs were used to measure the BeRP ball reflected with various thicknesses of nickel shells[6]. Getting the results from this Python script close to that of the benchmark would show that even though the script was intended for an MC-15, it could still be used to simulate measurements done with an NPOD. Additionally, this would demonstrate that the script could handle multiple detectors in a separated measurement, meaning that the output from each detector was analyzed individually.

To do this, the input file for the detailed model was adapted to split into 40 cases to match the measurement time of 10 minutes and run. Since the output of the NPODs was analyzed individually for the benchmark, the script was run in separated mode to duplicate these results. The count rate moments produced when the resultant `.lmx` files were processed with Momentum are compared with the benchmark measured and MCNP6 results in Table 4. The benchmark data used here can be found in Table 13 for the measured data, and Table 143 in Reference 6 for the MCNP6 data.

Table 5: Comparison of separated NPOD simulation results with the nickel shell benchmark

Method	R_1	Difference from Measured (%)	R_2	Difference from Measured %
Benchmark Measured	21843.04 ± 5.49	-	22679.81 ± 46.08	-
Benchmark Detailed Model	23060.84 ± 6.96	5.6	25602.97 ± 62.28	12.9
`multiLMX.py` - 1	22670.40 ± 11.57	3.8	24811.38 ± 84.32	9.4
`multiLMX.py` - 2	22653.81 ± 11.57	3.7	24797.54 ± 79.69	9.3

As the table shows, the results produced by `multiLMX.py` are close to the measured and simulated data from the benchmark. This shows that the scripts are capable of recreated measurements that contain an NPOD, and those that have multiple detectors not linked together. Running the analysis in `ptracLMX.py` took 20 minutes of time on a computing cluster, much longer than the MC-15 test case due to the larger number of events and separated event array processing.

5 Common Issues

There are a few common causes of errors with the running of ptracLMX.py that can cause some headaches. If an error or unexpected behavior, it is suggested to make sure that all of the cases of the MCNP problem finished running. If a problem didn't finish due to timing out or errors, then ptracLMX will be unable to determine how many events were printed to the ptrac file. This will cause an index out of bounds error during the execution of the python script. There are a couple of other oft-encountered errors that the script prints an error message for. One of which is not including a time distribution in the source. This would effectively have all source events occurring at time zero, which is not logical for a time-tagged output. Second, without

a cut card neutrons may continue their flight paths well outside of the detection window. To stop this, enter a cut card entry as follows:

```
cut:n 300E8 j 0
```

If you encounter any other issues, please inform the author so that updates may be made to future versions.

6 Conclusions

The comparisons between the measured and simulated results show that the scripts `multiLMX.py` and `ptracLMX.py` are capable of faithfully simulating the MC-15 and NPOD neutron detection systems. These scripts will allow for much easier modeling of measurements with this detection system, as the user only needs to run one script to have all the simulation and processing job submission done for them. These scripts do have some limitations, however. One such limitation is that only up to two detectors can be simulated with the script as it currently exists. Additionally, in order for `ptracLMX.py` to differentiate between events in the detectors in a separated measurement, the individual tubes of each need to be more than 40 centimeters away from each other.

References

[1] J.T. GOORLEY et. al., "Initial MCNP6 Release Overview," Nuclear Technology, 180, 298-315 (2012).

[2] M. JETTE, M. GRONDONA, "Slurm: Simple Linux Utility for Resource Management," Proceedings of ClusterWorld Conference and Expo, San Jose, CA, June 2003.

[3] C.J. SOLOMON, C. BATES, J. KULESZA, "The MCNPTools Package: Installation and Use," LA-UR-17-21779, Los Alamos National Laboratory, (2017).

[4] J. HUTCHINSON, D. LOAIZA, "Plutonium Sphere Reflected by Beryllium." PU-MET-FAST-038, International Handbook of Evaluated Criticality Safety Benchmark Experiments, NEA/NSC/DOC/(95)03, Nuclear Energy Agency, Organisation for Economic Co-operation and Development (Sep. 2007).

[5] J. HUTCHINSON, T. VALENTINE, "Subcritical Measurements of a Plutonium Sphere Reflected by Polyethylene and Acrylic." Nucl. Sci. Eng., 161, 357-362, 2009.

[6] B. RICHARD, J. HUTCHINSON, "Nickel-Reflected Plutonium-Metal-Sphere Subcritical Measurements." FUND-NCERC-PU-HE3-MULT-001, International Handbook of Evaluated Criticality Safety Benchmark Experiments, NEA/NSC/DOC/(95)03, Nuclear Energy Agency, Organisation for Economic Co-operation and Development (Sep. 2014).

[7] B. RICHARD, J. HUTCHINSON, "Tungsten-Reflected Plutonium-Metal-Sphere Subcritical Measurements." FUND-NCERC-PU-HE3-MULT-002, International Handbook of Evaluated Criticality Safety Benchmark Experiments, NEA/NSC/DOC/(95)03, Nuclear Energy Agency, Organisation for Economic Co-operation and Development (Sep. 2016).

[8] M. SMITH-NELSON, "Momentum: version 0.36.3," LANL Software, March 29, 2015.

[9] T. CUTLER, M.A. SMITH-NELSON, J.D. HUTCHINSON "Deciphering the Binning Method Uncertainty in Neutron Multiplicity Measurements." LA-UR-14-23374, Los Alamos National Laboratory, (2014).

Appendix A

The following is the source code for multiLMX, the python script that serves as the framework for submitting the MCNP simulation jobs.

```
# multiLMX.py - a Python 3 script to run multiple cases of MCNP and create a
# lmx file based on their outputs to mimic the MC15 response.
# written by Alex McSpaden, NEN-2 (mcspaden@lanl.gov)
# last edit August 2017

try:
    import argparse
    import datetime
    import sys
    import os.path
    import atexit
    import subprocess as sp
    import random as rnd
#    from datetime import timedelta
except ImportError:
    print('Couldn\'t import modules. Make sure Python and is loaded.')
    print('to load the module, use the following command:')
    print('module load python')

def printTime(start):
    # print the difference in time between the program start and finish
    print('Time elapsed: '+str(datetime.datetime.now()-start))


# start the clock, and set the printTime function to run at the termination of
# this script
start = datetime.datetime.now()
atexit.register(printTime,start)

# parse argument for input file
parser = argparse.ArgumentParser(description='Runs MCNP6 and creates lmx data files')
parser.add_argument('inputName',metavar='i',nargs=1, help='name of MCNP input file')
parser.add_argument('numCases',metavar='n',nargs=1, help='how many cases to run',
                    type=int)
parser.add_argument('version', metavar='v', nargs=1, help='mcnp version')
parser.add_argument('--time',metavar='t', nargs='?', help='cluster walltime', default='08:00:00')
parser.add_argument('--deadtime',metavar='d', nargs='?',
                    help='the amount of deadtime in a tube in ns', default=1000)
parser.add_argument('--tickLength',metavar='t',nargs='?',
                    help='how long the tick length is in ns', default=100)
parser.add_argument('--type',metavar='ty',nargs='?',
                    help='type of detector, NPOD or MC15', default='MC15')
parser.add_argument('--linked',metavar='l',nargs='?',
                    help='Detectors linked (1) or separated (0)', default=1)
args = parser.parse_args()

inputFile=str(args.inputName[0])
nCases = args.numCases[0]
version=str(args.version[0])
walltime=str(args.time)
deadtime=args.deadtime
tickLength=args.tickLength
detType=str(args.type)
linked=args.linked
```

```
# check to make sure MCNP version is valid
if version == '6.1':
    pass
elif version =='6.1.1':
    version = '6.1.1'
elif version == '6.2':
    version = '6.2'
elif version == 'bleedingedge':
    pass
else:
    sys.exit('Invalid version. Options are: 6.1 6.1.1 6.2 bleedingedge')

# check to make sure ptracLMX.py is in the folder too
if os.path.isfile('ptracLMX.py'):
    print('ptracLMX IS in folder, continuing on')
else:
    sys.exit('ptracLMX is missing, stopping')

job = []
print('Processed arguments, reading in input file')
# read in MCNP input file
with open(inputFile,'r') as inp:
    mcnpBase = inp.readlines()

parent=os.getcwd()

# make the directories for the sub cases and populate
print('Making subdirectories')
for i in range(1,nCases+1):
    formNum = str(i).zfill(3)
    caseNum = 'case'+formNum
    os.mkdir(caseNum)

    os.chdir(caseNum)

    # copy over and modify mcnp input files to insert random seeds
    with open(caseNum,'w') as newInp:
        for line in mcnpBase:
            if 'rand ' in line:
                randNum = 2*rnd.randrange(10000000000)+1
                newInp.write('rand gen=2 seed='+str(randNum)+'\n')
            else:
                newInp.write(line)

    # make slurm scripts to submit jobs
    fileID = 'sub'+formNum
    with open(fileID,'w') as slurmFile:
        slurmFile.write('#!/bin/tcsh\n')
        slurmFile.write('#SBATCH --time='+walltime+'\n')
        slurmFile.write('#SBATCH --nodes=1\n')
        slurmFile.write('#SBATCH --ntasks-per-node=1\n')
        slurmFile.write('date\n')
        slurmFile.write('module purge\n')
```

```
        slurmFile.write('module use /usr/projects/mcnp/modules\n')
        slurmFile.write('module load mcnp6/'+version+'\n')
        slurmFile.write('module list\n')
        slurmFile.write('\n')
        slurmFile.write('/usr/projects/mcnp/mcnpexe inp='+caseNum+'\n')

    #submit job
    output = sp.getoutput(['sbatch '+fileID])
    print('created and submitted case '+str(i))
    print(output)
    output = output.split(' ')
    #output = list(filter(None,output))
    #print(output)
    job.append(output[3])
    os.chdir(parent)

#print(job)
# create string of dependencies to be used in lmx job
dependencylist = ''
for i in range(0,nCases-1):
    dependencylist += str(job[i])+':'

dependencylist += str(job[nCases-1])

# create job to process to LMX using ptracLMX.py
fileID = 'subLMX'
with open(fileID,'w') as lmxSub:
    lmxSub.write('#!/bin/tcsh\n')
    lmxSub.write('#SBATCH --time=00:30:00\n')
    lmxSub.write('#SBATCH --nodes=1\n')
    lmxSub.write('#SBATCH --ntasks-per-node=1\n')
    lmxSub.write('#SBATCH --dependency=afterok:'+dependencylist+'\n')
    lmxSub.write('date\n')
    lmxSub.write('module purge\n')
    lmxSub.write('module load python\n')
    lmxSub.write('module use /usr/projects/mcnp/modules\n')
    lmxSub.write('module load mcnptools\n')
    lmxSub.write('module list\n')
    lmxSub.write('\n')
    lmxSub.write('python ptracLMX.py --numCases '+str(nCases)+' --deadtime '
                 +str(deadtime)+' --tickLength '+str(tickLength)+' --type '+
                     detType+' --linked '+str(linked))
    lmxSub.write('\n')
    lmxSub.write('ls -lh')

output = sp.getoutput(['sbatch '+fileID])
print(output)
print('Above is job for final LMX creation')
```

Appendix B

Once the MCNP runs finish, the python script ptracLMX is used to turn the output files into the final .lmx file. Its source code follows.

```
# ptracLMX.py version 1.1 - a script to make a single LMX file out of the multiple ptrac
# files created during a pstudy run
# written by Alex McSpaden, NEN-2, last edit April 2018
# send feedback/bug reports to mcspaden@lanl.gov

# CELL NUMBER --> TUBE NUMBER DICT
# cell numbers for tubes - change depending on model
# what follows is a rough ASCII sketch of tube numbers as used here
#       14    15
#   8  9 10 11 12 13
# 1  2  3  4  5  6  7
# if the detector in use is an NPOD, the tube numbers used for this script
# are as follows
#   9  10  11  12  13  14  15
# 1  2   3   4   5   6   7   8
# make dictionary of cell numbers : tube numbers, edit if necessary for model
tubeCell =  {227:1,  228:2,  229:3,  230:4,  231:5,  232:6,  233:7,  234:8,
             235:9,  236:10, 237:11, 238:12, 239:13, 240:14, 241:15}
# the above dict will be overwritten if a parameter file is used


try:
    import argparse
    import datetime
    import sys
    import os
    import math       as m
    import numpy      as np
    from mcnptools  import Ptrac
    from time import localtime, strftime
except ImportError:
    print('Couldn\'t import modules. Make sure Python and mcnptools are loaded.')
    print('to load the modules, use the following commands:')
    print('module load python')
    print('module use /usr/projects/mcnp/modules')
    print('module load mcnptools')
    print('put the above in your slurm launch script')

# define function to modify binary string based on an event happening in a tube
def modifyBin(currentString, tubeNumber):
    # function takes in the current string as it is and modifies it based on
    # which tube had the count
    # in binary, each tube is represented by the last 15 digits in the binary
    # version of an uint32. To mark that a tube has fired, increase by the amount
    # that corresponds to that tube's place in the binary number
    # For example, to mark that the third tube has fired, change third to last
    # binary digit to one, so add 2^2, as last digit is 2^0.
    return currentString+(2**(tubeNumber-1))

def readOutP(eventsSoFar, outp):
```

```
    # function that reads that outp file currently in the folder, and returns
    # the number of events that have been written to the corresponding ptrac
    # file
    with open(outp,'r') as mcOut:
        while True:
            line = mcOut.readline()
            if not line: break
            if 'binary file' in line:
                ptracInfo = line.split(' ')
                ptracInfo = list(filter(None,ptracInfo))
                eventsSoFar = eventsSoFar + int(ptracInfo[5])
            if 'for source variable tme' in line:
                # skip the next six lines
                for i in range(6):
                    line = mcOut.readline()
                # read the next line, split it into a list of strings, get rid of
                # spaces, convert the specific index into an integer, and divide
                # by 1E8 to convert from shakes to seconds
                line = mcOut.readline()
                totalTime = line.split(' ')
                totalTime = int(float(list(filter(None,totalTime))[1]))
                totalTime = totalTime /1E8
    try:
        return {'updEvents':eventsSoFar,'countTime':totalTime}
    except UnboundLocalError:
        print('MCNP input file lacks a time distribution in the source\n')
        print('LMX files are time dependent files, so tme distribution needed\n')
        print('If source does have tme distribution, print table 10\n')
        sys.exit()

def rowCounts(tube, detType, rows):
    # function to increment the proper row counter for the detector
    if detType == 'NPOD':
        if tube < 9:
            rows[0] += 1
        else:
            rows[1] += 1
    elif detType == 'MC15' or detType == 'NOMAD':
        if tube < 8:
            rows[0] += 1
        elif tube < 14:
            rows[1] += 1
        else:
            rows[2] += 1
    return rows

def sortDT(eventArr, length):
    # clean dead time rejected events from array
    print('Sorting out dead time rejected events')
    i = 0
    rejects = 0
    while i < length-1:
        # go through each event and look ahead to find events that happen within
        # the detector dead time and the same tube. When such an event is found,
```

```
        # set the tube number to zero so it is ignored by the writing loop later
        j = 1
        cTube = eventArr[i][0] # current tube
        cTime = eventArr[i][1] # current time
        if cTube != 0:
            while eventArr[i+j][1]-cTime < deadtime:
                if eventArr[i+j][0] == cTube: # in the same tube and within deadtime
                    eventArr[i+j][0] = 0
                    rejects +=1
                j += 1
                if i+j >= length:
                    break
        i+=1
    return rejects

# now go through and create the event array eventArr
# for separate ptrac cases, could add a dimension on to eventArr
# however ptracEvents will not be the same size for all cases, will definitely
# need to turn everything below into a function
def collectAndWrite(ptracEvents, ptracFile, detType, linked, offset, tubeCell, nCases, totalTime):
    if linked:
        eventArr = np.zeros((int(ptracEvents),2))
    else:
        eventArr = np.zeros((int(ptracEvents),5))
        eventArr1 = np.zeros((int(ptracEvents),2))
        eventArr2 = np.zeros((int(ptracEvents),2))
    # [tube, time, x, y, z]
    if detType == 'NPOD':
        rows=[0,0]
    else:
        rows=[0,0,0]
    # fill array with entries for events
    with open('plout.txt','w') as plo:
        plo.write('Reading Ptrac file(s)\n')
        print('Reading Ptrac file(s)\n')
        index = 0
        if nCases > 1:
            for i in range(1,nCases+1):
                dirName='case'+str(i).zfill(3)
                parent=os.getcwd()
                os.chdir(dirName)
                pFile = Ptrac(ptracFile)
                hists = pFile.ReadHistories(10000)
                while hists:
                    for h in hists:
                        # get the number of events and iniitalize event counter
                        numEvents = h.GetNumEvents()
                        evNum = 0
                        event = h.GetEvent(evNum)
                        # get the cell the event occurs in
                        try:
                            tube = tubeCell[event.Get(Ptrac.CELL)]
                        except KeyError:
                            sys.exit('Tube number dict wrong, update for model')
```

```
                rows = rowCounts(tube,detType,rows)
                # get time in ns (MCNP outputs shakes)
                time = (10 * event.Get(Ptrac.TIME)) - offset
                # determine if time is after offset
                if offset == 0 or time > 0:
                    eventArr[index,0]= tube
                    eventArr[index,1]= time
                    #print(tube)
                    if not linked:
                        eventArr[index,2]= event.Get(Ptrac.X)
                        eventArr[index,3]= event.Get(Ptrac.Y)
                        eventArr[index,4]= event.Get(Ptrac.Z)
                    index += 1
                    # grab the rest of the events in this history
                while evNum < numEvents - 1:
                    evNum += 1
                    event = h.GetEvent(evNum)
                    tube = tubeCell[event.Get(Ptrac.CELL)]
                    rows = rowCounts(tube,detType,rows)
                    time = (10 * event.Get(Ptrac.TIME)) - offset
                    if offset == 0 or time > 0:
                        eventArr[index,0] = tube
                        eventArr[index,1] = time
                        if not linked:
                            eventArr[index,2]= event.Get(Ptrac.X)
                            eventArr[index,3]= event.Get(Ptrac.Y)
                            eventArr[index,4]= event.Get(Ptrac.Z)
                        index += 1
            hists = pFile.ReadHistories(10000)
        os.chdir(parent)
else:
    pFile=Ptrac(ptracFile)
    hists = pFile.ReadHistories(10000)
    while hists:
        for h in hists:
            # get the number of events and iniitalize event counter
            numEvents = h.GetNumEvents()
            evNum = 0
            event = h.GetEvent(evNum)
            # get the cell the event occurs in
            try:
                tube = tubeCell[event.Get(Ptrac.CELL)]
            except KeyError:
                sys.exit('Tube number dict wrong, update for model')
            rows = rowCounts(tube,detType,rows)

            # get time in ns (MCNP outputs shakes)
            time = (10 * event.Get(Ptrac.TIME)) - offset
            if offset == 0 or time > offset:
                eventArr[index,0]= tube
                eventArr[index,1]= time
                if not linked:
                    eventArr[index,2]= event.Get(Ptrac.X)
```

```
                    eventArr[index,3]= event.Get(Ptrac.Y)
                    eventArr[index,4]= event.Get(Ptrac.Z)
                index += 1
            # grab the rest of the events in this history
            while evNum < numEvents - 1:
                evNum += 1
                event = h.GetEvent(evNum)
                tube = tubeCell[event.Get(Ptrac.CELL)]
                rows = rowCounts(tube,detType,rows)
                time = (10 * event.Get(Ptrac.TIME)) - offset
                if offset == 0 or time > 0:
                    eventArr[index,0] = tube
                    eventArr[index,1] = time
                    if not linked:
                        eventArr[index,2]= event.Get(Ptrac.X)
                        eventArr[index,3]= event.Get(Ptrac.Y)
                        eventArr[index,4]= event.Get(Ptrac.Z)
                    index += 1
        hists = pFile.ReadHistories(10000)

# at this point, should have an array that holds all the events
# now to sort it by time
plo.write('Sorting events by time\n')
print('Sorting events by time\n')
eventArr=eventArr[eventArr[:,1].argsort()]

# separate events into separate arrays
if not linked:
    if 17 in eventArr[:,0]:
        i = 0
        c1 = 0
        c2 = 0
        plo.write('separating based on cell number')
        print('Separating based on cell number')
        while i < index-1:
            # compare based on tube numbers
            if eventArr[i,0] > 16:
                eventArr2[c2,:] = eventArr[i,0:2]
                c2 += 1
            else:
                eventArr1[c1,:] = eventArr[i,0:2]
                c1 += 1
            i += 1
        # print(c1, c2)
        eventArr1=eventArr1[~np.all(eventArr1==0, axis=1)]
        eventArr2=eventArr2[~np.all(eventArr2==0, axis=1)]
    else:
        print('Separating based on detector position')
        plo.write('Separating based on detector position')
        i=1 # counter for eventArr
        c1 = 1 # counter for eventArr1
        c2 = 0 # counter for eventArr2
        # iterate through and decide if event belongs to detector 1,
        # which is the first detector and event is picked up in, or 2
```

```
        # get first x,y,z coords
        x = [ eventArr[0,2], -999]
        y = [ eventArr[0,3], -999]
        z = [ eventArr[0,4], -999]
        eventArr1[0,:] = eventArr[0,0:2]
        while i < index-1:
            #compare corrdinates of event to detector 1 coords
            xDist = abs(eventArr[i,2] - x[0])
            yDist = abs(eventArr[i,3] - y[0])
            zDist = abs(eventArr[i,4] - z[0])
            if (xDist > 40) or (yDist > 40) or (zDist > 40):
                # event is in second detector
                if xDist > 40 and x[1] == -999:
                    plo.write('Noticed detectors differ in x-axis\n')
                    print('Noticed detectors differ in x-axis')
                    x[1] = eventArr[i,2]
                    # detectors differ in x-axis
                if (yDist > 40) and (y[1] == -999):
                    plo.write('Noticed detectors differ in y-axis\n')
                    print('Noticed detectors differ in y-axis')
                    y[1] = eventArr[i,3]
                if (zDist > 40) and (z[1] == -999):
                    plo.write('Noticed detectors differ in z-axis\n')
                    print('Noticed detectors differ in z-axis')
                    z[1] = eventArr[i,4]
                eventArr2[c2,:] = eventArr[i,0:2]
                c2 += 1
            else:
                #event is in detector 1
                eventArr1[c1,:] = eventArr[i,0:2]
                c1 += 1
            i += 1

# make the filename of the file
fileID = datetime.date.today()
timestarted=strftime("%H%M%S", localtime())
dateForm=fileID.strftime('%Y_%m_%d')
fileID =dateForm +'_'+timestarted+'.lmx'
if linked:
    fileID = dateForm+'_'+timestarted+'.lmx'
else:
    fileID1 = dateForm+'_'+timestarted+'_1.lmx'
    fileID2 = dateForm+'_'+timestarted+'_2.lmx'


if detType == 'MC15' or detType == 'NOMAD':
    rr12 = rows[0]/rows[1]
    rr13 = rows[0]/rows[2]
    rr23 = rows[1]/rows[2]
else:
    rr12 = rows[0]/rows[1]
    rr13 = 0
    rr23 = 0
```

```
        # run functions to clean dead time rejected events
        if linked:
            rejects = sortDT(eventArr, eventArr.shape[0])
            r1 = (index-rejects)/totalTime
        else:
            rej1 = sortDT(eventArr1, eventArr1.shape[0])
            r1d1 = (c1-rej1)/totalTime
            rej2 = sortDT(eventArr2, eventArr2.shape[0])
            r1d2 = (c2-rej2)/totalTime

        print('Calling function to write .LMX file...')
        plo.write('Calling function to write .LMX file...\n')
        # call function to write LMX file
        if linked:
            writeLMX(eventArr,fileID, rr12, rr13, rr23, linked, r1, detType, totalTime)
        else:
            writeLMX(eventArr1,fileID1, rr12, rr13, rr23, linked, r1d1, detType, totalTime)
            writeLMX(eventArr2,fileID2, rr12, rr13, rr23, linked, r1d2, detType, totalTime)

def writeLMX(writeArr, fileID, rr12, rr13, rr23, linked, r1, detType, totalTime):
    # write header of .LMX file
    with open((fileID),"wb") as lmxFile:
        length = writeArr.shape[0]
        print('Writing .LMX file')
        print('This detector saw ', writeArr.shape[0], ' events.')
        doARoll = False
        # different binary event flags that can be seen in .LMX files
        eventFlag     = b'\x00\x00\x00\x00'
        clockRollFlag = b'\x01\x00\x00\x00'
        # gateStartFlag = 0x00000002
        # gateEndFlag   = 0x00000003
        eofFlag       = b'\xFF\xFF\xFF\xFF'
        lmxFile.write(b'ListModeDataFileVersion       : 1.06\r\n')
        lmxFile.write(b'InstrumentType                : Neutron Multiplicity\r\n')
        if detType == 'NOMAD':
            lmxFile.write(b'InstrumentModel               : NoMAD\r\n')
            lmxFile.write(b'SerialNumber                  : S/N LANL 001\r\n')
            lmxFile.write(b'HardwareVersion               : MB6-E17P\r\n')
        if detType == 'MC15':
            lmxFile.write(b'InstrumentModel               : MC-15\r\n')
            lmxFile.write(b'SerialNumber                  : S/N LANL 001\r\n')
            lmxFile.write(b'HardwareVersion               : MB6-E17P\r\n')
        else:
            lmxFile.write(b'InstrumentModel               : NPOD\r\n')
        lmxFile.write(b'MeasurementID                 : '+fileID.encode('utf-8')+b'\r\n')
        lmxFile.write(b'MeasurementDescription        : Simulated Measurement\r\n')
        lmxFile.write(b'MeasurementMode               : Single\r\n')
        if linked:
            lmxFile.write(b'FrontPanelConfig              : Together\r\n')
        else:
            lmxFile.write(b'FrontPanelConfig              : Separate\r\n')
        lmxFile.write(b'AnalysisChannels              : 0x00007FFF\r\n')
        lmxFile.write(b'DateStart                     : '+
                      str(datetime.date.today()).encode('utf-8')+b'\r\n')
```

```
lmxFile.write(b'TimeStart                       : '+
              str(datetime.datetime.now().time()).encode('utf-8')+
              b'\r\n')
lmxFile.write(b'DurationRealTime                : '+
              str(totalTime).encode('utf-8')+b' [s]\r\n')
lmxFile.write(b'InternalScaler                  : '+
              str(length).encode('utf-8')+b'\r\n')
lmxFile.write(b'FifoLostCounts                  : 0\r\n')
lmxFile.write(b'AverageCountRate                : '+
              str(round(r1,2)).encode('utf-8')+b' [counts/s]\r\n')
lmxFile.write(b'DistanceDetFaceToSource         : See Input\r\n')
lmxFile.write(b'DistanceDetCenterToFloor        : See Input\r\n')
lmxFile.write(b'FirmwareChannelDeadtime         : '+str(deadtime).encode('utf-8')+
              b' [ns]\r\n')
if detType == 'MC15' or detType == 'NOMAD':
    lmxFile.write(b'RowRatio                          : '+str(rr12).encode('utf-8')+b'\r\n')
    lmxFile.write(b'RowRatio(1/3)                     : '+str(rr13).encode('utf-8')+b'\r\n')
    lmxFile.write(b'RowRatio(2/3)                     : '+str(rr23).encode('utf-8')+b'\r\n')
else:
    lmxFile.write(b'RowRatio                          : '+str(rr12).encode('utf-8')+b'\r\n')

lmxFile.write(b'Comment                         : '+
              b'If all channels are zero, this indicates channels\r\n')
lmxFile.write(b'Comment                         : '+
              b'in next event is a flag (not a real event)\r\n')
lmxFile.write(b'Comment                         : '+
              b'Flag = 0x00000001 (Clock rollover occurred)\r\n')
lmxFile.write(b'Comment                         : '+
              b'Flag = 0x00000002 (Gate input started)\r\n')
lmxFile.write(b'Comment                         : '+
              b'Flag = 0x00000003 (Gate input ended)\r\n')
lmxFile.write(b'Comment                         : '+
              b'Flag = 0xFFFFFFFF (End of binary data)\r\n')
lmxFile.write(b'BinaryDataEventSizeInBytes      : 8\r\n')
lmxFile.write(b'BinaryDataChannelFormat         : unsigned int32\r\n')
lmxFile.write(b'BinaryDataClockFormat           : unsigned int32\r\n')
lmxFile.write(b'BinaryDataClockTickLength       : '+
              str(tickLength).encode('utf-8')+b' [ns]\r\n')
lmxFile.write(b'BinaryDataActiveChannels        : 0x0000FFFF\r\n')
lmxFile.write(b'BinaryDataFollows               : ')

# pad the file with spaces until needed length is reached
while (lmxFile.tell()-14)%16 !=0:
    lmxFile.write(b' ')
lmxFile.write(b'\r\n')

# iterate over the whole array to lump together ticks and
# filter events for dead time rejection
clockRolls=0
i=0
while i < length-1:
    if writeArr[i][0] == 0:
        i += 1
        continue
```

```
# find the tick number and initialize binary string
tick = m.ceil(writeArr[i][1]/tickLength)
tubeString = modifyBin(0,int(writeArr[i][0]))
# look forward in the events  for dead time and tick lumping
j=1
past=False
while not(past):
    if (i+j)<length-1:
        # if the end of the array hasn't been reached yet
        if (writeArr[i+j][0] == 0):
            # event was deadtime rejected, skip
            j += 1
        elif (writeArr[i+j][0] != writeArr[i][0]) and
             (tick == m.ceil(writeArr[i+j][1]/tickLength)):
            # the tick corresponding to the next event is the same as
            # the current event, and should be lumped in
            tubeString = modifyBin(tubeString,int(writeArr[i+j][0]))
            j +=1
        elif (writeArr[i+j][1] >= (totalTime * 1E9)):
            # next event is beyond measurement time, stop writing
            i=length # make sure events stop being processed
            past = True
        else:
            # event is neither to be rejected by deadtime or to be
            # lumped in with current. Therefore it is to become the new
            # current event
            # check for a clock rollover
            past = True
            nextTick = m.ceil(writeArr[i+j][1]/tickLength)
            if m.ceil(tick/maxInt) < m.ceil(nextTick/maxInt):
                # clock rollover is happening
                doARoll = True
    else:
        past = True # since i+j goes past the number of events
# increment i so that the next iteration starts on the next "new" event
i += j
# write the current tube string and tick to the lmx file
lmxFile.write(tubeString.to_bytes(4,byteorder=sys.byteorder,signed=False))
writeTick = tick - (clockRolls*maxInt)
lmxFile.write(writeTick.to_bytes(4,byteorder=sys.byteorder,signed=False))
# if a clock rollover just happened, increment clock rolls and print
# the event flags
if doARoll:
    #print('roll')
    #print(tick)

    lmxFile.write(eventFlag)
    lmxFile.write(writeTick.to_bytes(4,byteorder=sys.byteorder,signed=False))
    lmxFile.write(clockRollFlag)
    lmxFile.write(writeTick.to_bytes(4,byteorder=sys.byteorder,signed=False))
    clockRolls+=1
    doARoll = False
```

```
        endTick=m.ceil((totalTime*1E9)/tickLength)-(clockRolls*maxInt)
        #print(endTick)
        #print(writeArr[-1,1])
        #print(clockRolls)
        #print(maxInt)
        lmxFile.write(eventFlag)
        lmxFile.write(endTick.to_bytes(4,byteorder=sys.byteorder,signed=False))
        # write the EOF flag
        lmxFile.write(eofFlag)
        lmxFile.write(endTick.to_bytes(4,byteorder=sys.byteorder,signed=False))

# largest number that can be represented by a uint32
maxInt = 4294967295
doARoll = False

parser = argparse.ArgumentParser(description='Goes through the multiple ptrac'+
                                 ' files and makes an lmx file, the use of a parameter'+
                                 ' file is recommended instead of command line arguments.'+
                                 ' The parameter file is also where the tube number'+
                                 ' assignment dict and options for subdirectory'+
                                 ' naming is located. Please direct any'+
                                 ' questions or bug reports to mcspaden@lanl.gov')
parser.add_argument('--numCases',metavar='n',nargs='?',help='how many cases were run',
                    type=int, default=1)
parser.add_argument('--deadtime',metavar='d', nargs='?',
                    help='the amount of deadtime in a tube in ns', default=1000)
parser.add_argument('--tickLength',metavar='t',nargs='?',
                    help='how long the tick length is in ns', default=100)
parser.add_argument('--type',metavar='ty',nargs='?',
                    help='NPOD, NOMAD, or MC15', default='NOMAD')
parser.add_argument('--linked',metavar='l',nargs='?',
                    help='Detectors linked (1) or separated (0)', default=1)
parser.add_argument('--parFile', metavar='pf', nargs='?',
                    help='Name of parameter file including args and other info',
                    default='ptracLMXparams')
parser.add_argument('--offset', metavar='o', nargs='?',
                    help='Number of seconds to skip at beginning of simulation'+
                    ' NOTE!: If you are going to use an offset, ensure your'+
                    ' simulation time duration is the desired measurement' +
                    ' time plus the offset. e.g. a simulation for a 300s measurement' +
                    ' plus a 5s offset should be 305s.',
                    default=0)
parser.add_argument('--outp', metavar='op', nargs='?',
                    help='Name of the output file (default outp)', default='outp')
parser.add_argument('--ptrac', metavar='pt', nargs='?',
                    help='Name of the ptrc file default ptrac)', default='ptrac')
args = parser.parse_args()
nCases = int(args.numCases)
deadtime=int(args.deadtime)
tickLength=int(args.tickLength)
detType=args.type
linked=bool(int(args.linked))
parFile = args.parFile
offset= int(args.offset) * 1E9 # convert to nanoseconds
```

```
outp=args.outp
ptracFile=args.ptrac

# default values for items in parameter file
defaultDirs = True
combPtrac = True

try:
    module = __import__(parFile)
    nCases = module.numCases
    deadtime = module.deadtime
    tickLength = module.tickLength
    detType = module.detType
    linked = module.linked
    offset = module.offset * 1E9
    tubeCell = module.tubeCell
    outp = module.outp
    ptracFile = module.ptrac
    defaultDirs = module.defaultDirs
    if not defaultDirs:
        dirNames = module.dirNames
    combPtrac = module.combPtrac
except:
    print('Parameter file either not present or incomplete, using command arguments\n')

if detType == 'MC15' or detType == 'NPOD' or detType == 'NOMAD':
    pass
else:
    sys.exit('Detector type invalid, enter either \'NPOD\', \'NOMAD\' or \'MC15\'')



if defaultDirs and nCases > 1:
    dirNames = [''] * nCases
    for i in range(0,nCases):
        dirNames[i] = 'case'+str(i+1).zfill(3)
        #print(dirNames[i])

if not defaultDirs and nCases == 1:
    sys.exit('You don\'t need to name a single directory, run this there.')


# first step, look in all the outp files to see how many events need to be counted

if not combPtrac:
    ptracEvents = np.zeros([nCases]) #initialize
else:
    ptracEvents = np.zeros([1])


if nCases > 1:
    for i in range(0,nCases):
        parent=os.getcwd()
        if combPtrac:
```

```
            os.chdir(dirNames[i])
            readValues = readOutP(ptracEvents, outp)
            ptracEvents = readValues['updEvents']
            os.chdir(parent)
        else:
            os.chdir(dirNames[i])
            print('Moved into directory ', dirNames[i])
            readValues = readOutP(ptracEvents[i], outp)
            ptracEvents[i] = readValues['updEvents']
            print('Processing ', ptracEvents[i], 'events')
            totalTime = readValues['countTime'] - (offset/1E9)
            collectAndWrite(ptracEvents[i], ptracFile, detType, linked,
                            offset, tubeCell, 1, totalTime)
            os.chdir(parent)
else:
    readValues = readOutP(ptracEvents, outp)
    ptracEvents = readValues['updEvents']

totalTime = readValues['countTime'] - (offset/1E9)
print('Processing ', ptracEvents, ' events')

# call functions to collect and collate the data, and write the .LMX file
if combPtrac:
    collectAndWrite(ptracEvents, ptracFile, detType, linked,
                    offset, tubeCell, nCases, totalTime)
```

Appendix C

Below is an example of a parameter file to be used with the ptrac processing script.

```
# parameter file for ptracLMX.py

# can contain (and will overwite) any of the command line arguments these inlcude
# number of cases to combine
numCases=1

# dead time in the detector
deadtime = 256

# the tick length of the detector
tickLength = 100

# the type of detector ('NPOD', 'NOMAD' or 'MC15')
detType = 'NOMAD'

# if multiple detectors, whether they are linked (1) or seperated (0)
linked = 1

# the time offset, or number of seconds at the beginning of the measurement
# that should be skipped
```

```
# !!!!!!!!
# NOTE!: If you are going to use an offset in your measurement, ensure that you
# extend the time duration of your simulation enough so that you end up with
# the proper duration in the final data file
# For example, if you want a 600 second measurement with a five second offset,
# have your simulation  605 seconds.
# !!!!!!!!
offset = 0

# will also contain the tube number dict for converting the model cell numbers
# to the detector tube numbers
# cell numbers for tubes - change depending on model
# what follows is a rough ASCII sketch of tube numbers as used here
#       14    15
#   8  9 10 11 12 13
# 1  2  3  4  5  6  7
# if the detector in use is an NPOD, the tube numbers used for this script
# are as follows
#   9  10  11  12  13  14  15
# 1  2   3   4   5   6   7   8
# make dictionary of cell numbers : tube numbers, edit if necessary for model
tubeCell =  {800:1,  801:2,  802:3,  803:4,  804:5,  805:6,  806:7,  807:8,
             808:9,  809:10, 810:11, 811:12, 812:13, 813:14, 814:15, 815:17,
             816:18, 817:19, 818:20, 819:21, 820:22, 821:23, 822:24, 823:25,
             824:26, 825:27, 826:28, 827:29, 828:30, 829:31}
# if there are two detectors with distinct cell numbers (i.e. did not use
# universe fills) then populate the dict with those cells as tubes 17-31.

# may also contain the name of the output and ptrac files to be processed
# the default is 'outp' and 'ptrac'
outp='outp'
ptrac='ptrac'


# options for how the subdirectories are named in multiple case runs
# and whether those multiple cases should be combined or left separate
# default is case001-caseXXX, if other names are desired, make defaultDirs False
# and create a list of strings that represent the desired names
# e.g. ['dir1' 'dir2' 'dir3'] etc.
defaultDirs = True

# if defaultDirs = False, enter a list of strings like the one below
# dirNames = ['case010', 'case011', 'case012', 'case013', 'case014', 'case015']
# except with your directory names

# whether the separate ptrac files should be combined or left separated
combPtrac = True
```