

Runfile users' guide

1 Introduction

`Run` is a general-purpose utility for controlling Unix jobs. It tries to look like `Make` but combines elements of shell scripts, `Make` and perl into a single ‘`Runfile`’.

A job can be described in terms of its dependencies on other jobs, as in `Make`. The order in which jobs are executed can be predetermined, as in a shell script. Since `run` knows the dependencies of the jobs, it is able to make reasonably intelligent guesses at how to restart a crashed job. `Run` keeps logs of all the jobs it starts and their output in a ‘`history`’ directory.

In addition to controlling jobs, `run` records timing and progress information. This can be used for profiling jobs and estimating time-to-completion.

`Run` itself is written in perl. You do not need to be proficient in perl to start using `run`, but there are many `hooks` which allow the advanced user to extend the functionality of `run` with additional perl code. These hooks are accessed from the user’s ‘`Runfile`’, not by modifying the `run` source code.

Since the purpose of `run` is to control jobs on ACPMAPS, there are a number of additional utilities provided for handling tapes and field files. These are not part of `run`, which deals only with Unix processes.

2 A simple Runfile

In this chapter we'll look at a very simple example of a 'Runfile'. Its task will be to create three text files and count the words in them.

2.1 The simple Runfile

Here is the 'Runfile' in full. You can copy this example into a source file to try it.

```
::perl
@runs = ( '001', '002', '003' ) ; # this is required

print "hello from the perl section\n" ;

::script

%create_text_file
%count_words

::makefile

create_text_file:
    echo "hello world" > $run.txt

count_words: create_text_file
    wc -w $run.txt
```

2.2 Invoking run

Assuming we have named our file 'Runfile' we can start things running with the command,

```
bjg|fncrd6> run
```

This is the output,

```
bjg|fncrd6> run
run 0.5 (Exp) (bjg) [26656]
hello from the perl section
```

```
#
001 starting... Sat Sep  2 23:19:01 CDT 1995
001 submitting create_text_file
001 running create_text_file [26665]
001 create_text_file completed successfully [26665] 0 s
001 submitting count_words
001 running count_words [26675]
001 count_words completed successfully [26675] 0 s
001 completed successfully 3 s
...
```

2.3 The simple runfile in detail

Let's take a look at the three sections of the 'Runfile', which are denoted by `::perl`, `::script` and `::makefile`. First there is a perl section, `::perl`,

```
::perl
@runs = ( '001', '002', '003' ) ; # this is required

print "hello from the perl section\n" ;
```

which can contain arbitrary perl code. It is executed before the script starts launching jobs. The only required part of the perl section is the definition of the list `@runs`. This gives the numbers or names of the runs to be executed. In perl `@` denotes a variable which is a list or array. In this case we will have three runs called 001, 002 and 003.

Next, in the script section of the 'Runfile' we can include shell commands to generate a list of the jobs to be executed, and the order they should be executed in,

```
::script

%create_text_file
%count_words
```

In this case we simply list the jobs that we want to execute. Typically this is all that is needed and the `script` section is very straightforward.

Note that the names of the jobs should be prefixed with a `%` sign, meaning that they refer to the Makefile section which follows.

In the `makefile` section of the script we give the definitions of the jobs, in terms of the commands needed to run them and their dependencies. This works in exactly the same way as a `Makefile`. Note in particular that the commands should be indented using tabs as they are in a `Makefile`.

```

::makefile

create_text_file:
    echo "hello world" > $run.txt

count_words: create_text_file
    wc -w $run.txt

```

In this case the job `create_text_file` writes `hello world` into the file `'$run.txt'`, where `$run` is a reserved variable which gives the current run number (001, 002 or 003). In perl `$` denotes an ordinary variable, just as in the shell.

The job `count_words` depends on `create_text_file` and must wait for it to complete successfully before starting. It counts the words in the file created by `create_text_file` using `wc -w`.

2.4 Output from run

Now let's look at the output from `run` itself and what it does. We start everything with the command `run`, which searches for a file called `'Runfile'` by default (just as `make` looks for a `'Makefile'`).

```

bjg|fncrd6> run
run 0.5 (Exp) (bjg) [26656]

```

The first message tells us that we are using `run` version 0.5 (Experimental, author bjg), and most importantly its pid, given in square brackets `[pid]`. If we need to kill a process then we can check which runscript it belongs to using this number.

Next we see the message from the perl section of the `'Runfile'` saying, `print "hello... "`. The array `@runs` is also defined at this point, but without any output so we do not see it.

```

hello from the perl section
#
001 starting... Sat Sep  2 23:19:01 CDT 1995

```

Then the first run 001 from the array `@runs` begins, noting the start time. The first column gives the run number, which is useful for `grep`.

```
001 submitting create_text_file
001 running create_text_file [26665]
```

We can now see the two phases of job execution, for `create_text_file`. First the job is "submitted" to a notional queue within `run` — this only means that the job is allowed to start, and does not have any uncompleted dependencies. After being "submitted" there are a few additional checks (to make sure there are sufficient processes available to run the job) before the command is actually run. Its process id is given in square brackets [*pid*].

If we were to look in the `'history/'` directory at this point we would see the log file for `create_text_file`,

```
bjg|fncrd6> ls -R history/
001/

history/001:
create_text_file.running
```

it contains the a record of the commands being executed, the start date and end date,

```
bjg|fncrd6> more history/001/create_text_file.running
echo "hello world" > 001.txt
# start: Sat Sep  2 23:19:02 CDT 1995
....
```

When the job exits the log file is renamed from `'create_text_file.running'` to `'create_text_file.log'`.
This is a record that it has completed successfully. If any of the commands in the job failed it would be renamed `'create_text_file.err'`. At this point `run` issues a message (noting successful completion), and the time taken (in this case, the job was so quick it took 0 seconds).

```
001 create_text_file completed successfully [26665] 0 s
```

Once the jobs have completed, we get a message giving the overall status of the run. In this case everything is successful,

```
001 completed successfully 3 s
```

and the whole run for creating the text file, and counting the words took 3 seconds from start to finish. At this point, the script goes on and perform the same tasks for runs 002 and 003 before stopping.

2.5 Errors and Restarts

Let's imagine that the job failed at some point. Suppose that the filename of the word counting program `wc` had been misspelt as `ec`

```
count_words: create_text_file
              ec -w $run.txt # a typo, should be wc
```

so that the command would fail,

```
bjg|fncrd6> ec -l file.txt
bash: ec: command not found
```

Everything would run as before up to the point where the `count_words` job starts,

```
bjg|fncrd6> run
run 0.5 (Exp) (bjg) [27029]
hello from the perl section
#
001 starting... Sun Sep  2 23:30:19 CDT 1995
001 submitting create_text_file
001 running create_text_file [27038]
001 create_text_file completed successfully [27038] 2 s
001 submitting count_words
001 running count_words [27048]
001 count_words FAILED [27048] status 1 time 1 s
001 FAILED 4 s
```

but in this case the job would fail (with exit status 1). If we look in the history directory at this point we would see some new files, in addition to the log file from `create_text_file` (which succeeded),

```
bjg|fncrda> ls -R history/
001/

history/001:
FAILED                count_words.err      create_text_file.log
```

There is an empty file 'FAILED', which marks the directory as a lost cause, and an error log file 'count_words.err' which contains the output from the failed job count_words,

```
bjg|fncrda> more history/001/count_words.err
ec -w 001.txt
# start: Sun Sep  5 23:30:22 CDT 1995
sh: ec: not found
```

Once we have looked at the error log file 'count_words.err' and figured out that the typo in the 'Runfile' was the problem we can correct it,

```
count_words: create_text_file
            wc -w $run.txt # now correct
```

and do a "restart",

```
bjg|fncrda> run -i 001
run 0.5 (Exp) (bjg) [27077]
hello from the perl section
001 previously failed... ignoring error and retrying
#
001 starting... Tue Sep  5 18:14:21 CDT 1995
001 moving existing logs into history/001/00
001 already completed create_text_file
001 submitting count_words
001 running count_words [27087]
001 count_words completed successfully [27087] 1 s
001 completed successfully 2 s
```

We use two new options when calling run. First, the -i option tells run to ignore any errors that it has previously encountered (since we have now fixed the cause of them). Without the -i we would get the message,

```
001 previously failed... skipping
```

Secondly, we specify the run number that we want to retry on the command line (001). When the script runs we note that it first tidies up existing error log files,

```
001 moving existing logs into history/001/00
```

by making a subdirectory 00. Subsequent retries would make the directories 01, 02, etc. for old error log files.

It then checks the log files to see which jobs completed successfully and notes that it has already completed the first step,

```
001 already completed create_text_file
```

so it does not need to rerun `create_text_file`. Now the corrected version of the `count_words` job can be submitted,

```
001 submitting count_words
001 running count_words [27087]
001 count_words completed successfully [27087] 1 s
001 completed successfully 2 s
```

and this time it runs successfully. The final state of the `'history'` directory is the same as before,

```
bjg|fncrda> ls -R history/
001/

history/001:
00/          count_words.log
DONE         create_text_file.log

history/001/00:
count_words.err
```

except for the presence of the subdirectory `'history/001/00'` which shows that the run had to be restarted once.

3 Dealing with Tapes

Before using launching into the complexities of controlling a real-world job using a `runfile`, let's first take a look at some of the tape handling utilities that are available to make life easier.

One of the features of using the ACPMAPS tape system is that it isn't like a normal filesystem – and so it doesn't necessarily integrate with Unix or physics. For example, a tapeset may not be big enough to hold all the configurations for a given project, so probably it gets extended to multiple tapes, with different names `.1`, `.2`, `.3`, etc. A script then has to deal with this to know which tape to use.

There are a couple of utilities called `map` and `tapemap` which allow your script to deal with tapes in a more flexible and Unix-like way.

3.1 Tape Maps

The basic idea is to use a `'tape-map'` to index the files. This is a text file containing a list of configuration numbers and the tape file names that they correspond to.

Here is a `'tape-map'` for the D lattice gauge configurations, `'Could-09.map'`

```
004000 Could0-09#coul_oooy_6.1_004000
008000 Could0-09#coul_oooy_6.1_008000
012000 Could0-09#coul_oooy_6.1_012000
.....
096000 Could0-09#coul_oooy_6.1_096000
100000 Could0-09#coul_oooy_6.1_100000
104000 Could1-09#coul_oooy_6.1_104000
108000 Could1-09#coul_oooy_6.1_108000
.....
```

We only need to know the configuration number that we are interested in and we can get the correct tape filename from the map file. It doesn't matter whether the file is on `'Could0-09'` or `'Could1-09'` (note the change at configuration 104000), or even if the configurations are in a strange order. We could extract the filename for a given configuration number using `awk` or `perl`, but there is a special command called `map` to save us the trouble,

```
bjg|fncrd6> map -n 004000 Could-09.map
Could0-09#coul_oooy_6.1_004000
```

Note that you need to specify the configuration with its leading zeros or you will get an error,

```
bjg|fncrd6> map -n 4000 Could-09.map
Configuration 4000 not found in Could-09.map
```

That is something which it is easy to forget to do.

Within a shell script, we will need a line which looks something like this,

```
$gauge_file='map -n $conf Could-09.map'
```

to get the name of tape file. We don't have to worry about dividing the configuration number by the number of files on the tape to find the tape volume number etc, etc, all of which is no fun to do in a shell.

3.2 Making a map file for existing tapes

We'll want to make `map` files for tapes that we already have, and also for tapes which we want to initialize. When we already have the tape, it's easy to make an index of the files on it and produce a `map` file. All the information that we need is in the 'CANTapes' directory.

```
bjg|fncrd6> more /usr/local/spool/CANTapes/Could0-09
Could0-09 last unmounted: Wed May  3 14:26:52 1995
Directory for 4 volume tape set Could0-09 created on 1993 Oct  1 17:10:34
Drives: [c202] [c204] [c302] [c303]
      Date           Blksize   Blks File Name
1993 Oct  1 17:29:03   32768    1540 Could0-09#coul_oooy_6.1_004000
1993 Oct  1 17:37:59   32768    1540 Could0-09#coul_oooy_6.1_008000
1993 Oct  1 18:10:26   32768    1540 Could0-09#coul_oooy_6.1_012000
1993 Oct  1 18:32:01   32768    1540 Could0-09#coul_oooy_6.1_016000
1993 Oct  1 18:54:08   32768    1540 Could0-09#coul_oooy_6.1_020000
1993 ...
```

There are some headers and other pieces of information that we don't need, but there is a `tapemap` utility which looks at the 'CANTapes' files, gets rid of the junk and produces a `tapemap` file,

```
bjg|fncrd6> tapemap Could0-09 Could1-09 Could2-09 Could3-09
004000 Could0-09#coul_oooy_6.1_004000
008000 Could0-09#coul_oooy_6.1_008000
012000 ....
```

We can specify all the tapes that we want to index with a filename pattern,

```
bjg|fncrd6> tapemap 'Could*-09'
004000 Could0-09#coul_oooy_6.1_004000
008000 Could0-09#coul_oooy_6.1_008000
012000 ....
```

The wildcard '*' needs to be quoted, because the `tapemap` program is going to look for them in the 'CANTapes' directory, not the current directory where we're running the command.

Obviously, this magic is only going to work if the filenames end with a configuration number like `_001000`. Fortunately most of the filenames do.

Sometimes there will be a problem with several files for each configuration being saved on the same tape. For example there might be,

```
004000 Could0-09#coul_oooy_6.1_004000
004000 Could0-09#coul_bcccx_6.1_004000
```

on the same tape. When this happens there isn't a one-to-one mapping from configuration numbers to filenames, and the `tapemap` program will complain about **duplicate configurations**.

We can get around this by narrowing down our search using the `-r` *regex* option on `tapemap`,

```
bjg|fncrd6> tapemap -r 'oooy' 'Could*-09'
004000 Could0-09#coul_oooy_6.1_004000
008000 Could0-09#coul_oooy_6.1_008000
012000 ....
```

to restrict the listing to the `oooy` configurations.

3.3 Making a map file for new tapes

If we need to make a tape map file for tapes which don't yet exist then we have to prepare the names of the files that we are going to use.

Suppose we want to generate some new propagators for a B99 run at `kappa=0.1423`, using smeared sources (`1S`). We'll want to use a tape name like `'B99qf0.1423_1S'`, but since we'll be generating 300 configurations we'll probably need to use several tapes.

First of all, let's make a list of the configuration numbers we'll be dealing with, using the `newconfigs` command,

```
bjg|fncrd6> newconfigs -n 300 001000 1000
001000
002000
003000
004000
.....
299000
300000
```

The `newconfigs` command has the form,

```
newconfigs -n number-of-configs start-config step-size
```

So in this case, we have generated 300 configuration numbers, starting from configuration 001000, with a step size (or sweep count) of 1000.

We need this list, so let's put it into a file (which we'll call '@CONFIGS') for future reference,

```
bjg|fncrd6> newconfigs -n 300 001000 1000 > @CONFIGS
```

Now we can generate a map file for our new filenames, using a command called `newtapemap`. Let's suppose that we want to call each individual file something like,

```
'B99qf0.1423_1S.1#B99_qf_1S_d_0.1423_1.57_001000'
```

where `.1` will be a tapeset number (which might be 2, or 3, or higher, depending on how many files we can fit on a tape), and 001000 is the configuration number.

We can generalize the form of this filename, using the variables `$n` and `$conf` to represent the tape volume and configuration number,

```
'B99qf0.1423_1S.${n}#B99_qf_1S_d_0.1423_1.57_${conf}'
```

We'll use this general form to tell the `newtapemap` command the names of our files.

Finally, we need to know how many files are meant to go on each tape volume. Let's suppose that we are going to put 100 propagators on each 2- tape tapeset. This will be an option to the `newtapemap` command.

Now we can generate a list of the files we need using the command,

```
bjg|fncrd6> newtapemap -n 100 \
-f 'B99qf0.1423_1S.${n}#B99_qf_1S_d_0.1423_1.57_${conf}' \
@CONFIGS
```

The `-n` option gives the number of files on each tape volume, and the `-f` option gives the general form of the filename. The configuration numbers are taken from the file '@CONFIGS'. The output of the command is a tape map for the 300 configurations in '@CONFIGS',

```
001000 B99qf0.1423_1S.0#B99_qf_1S_d_0.1423_1.57_001000
002000 B99qf0.1423_1S.0#B99_qf_1S_d_0.1423_1.57_002000
003000 B99qf0.1423_1S.0#B99_qf_1S_d_0.1423_1.57_003000
.....
298000 B99qf0.1423_1S.2#B99_qf_1S_d_0.1423_1.57_298000
299000 B99qf0.1423_1S.2#B99_qf_1S_d_0.1423_1.57_299000
300000 B99qf0.1423_1S.2#B99_qf_1S_d_0.1423_1.57_300000
```

Notice that the tape volume number (given by `${n}`) has increased automatically from `.0` to `.2`, along with the configuration numbers (which came from the file '@CONFIGS'). We will need to initialize 3 tapesets (and these will have to be 2-tape tapesets).

The `newtapemap` command has the form

```
newtapemap -n files-per-tapeset -f tape-filename config-list
```

The *tape-filename* can include two special variables,

\$n Variable

This gives the tapeset number (files/files-per-tapeset)

\$conf Variable

This gives the configuration number

These variables need to be protected from the shell, by enclosing the filename in single quotes ‘*tape-filename*’, so that the `newtapemap` program can interpret them itself.

3.4 Tape inits

As a safety feature, there is a command called `tapeinit` which does the same job as `canreserve tapeset -n`, but performs additional checks before initializing a tape. These are to prevent a runaway script from initializing unwanted tapes with strange tape names.

If we initialize a tape, for example with the command,

```
tapeinit B99qf0.1423_1S.1
```

then `tapeinit` will first check in an authorisation file ‘@INITS’ to see if the tapename is acceptable. If the tape name is not found then the tape will not be initialized, and `tapeinit` will stop with an error.

A suitable authorisation file ‘@INITS’ would contain the lines,

```
B99qf0.1423_1S.1 -2
B99qf0.1423_1S.2 -2
....
```

it is just a list of the the tapes that we really do want to initialize, and the number of tapes in the `tapeset`, the `-n` option on `canreserve`.

Normally it is not necessary to run `tapeinit` manually (although that is certainly possible). Tape initialization is typically carried out automatically by the higher level commands `archive_quark_field` and `archive_gauge_field`.

However, the @INITS file must still be there. There is a command to make a suitable @INITS file, for example,

```
bjg|fncrd6> newinits -n 2 B99qf0.1423_*.map
B99qf0.1423_1S.1 -2
B99qf0.1423_1S.2 -2
....

bjg|fncrd6> newinits -n 2 B99qf0.1423_*.map > @INITS
```

which searches in the map files 'B99qf0.1423_*.map' and lists the tapes which they use. The general form of the command is,

```
newinits -n volumes FILE > @INITS
```

where the option `-n volumes` gives the number for the `canreserve -n` option, the number of tapes in the tapeset.

At the moment the filename `@INITS` is hard-wired into `tapeinit` (and assumed to be in the current directory). That restriction could be lifted if required.

4 New tape commands

This chapter contains the detailed usage and command-line option information for the new tape handling commands.

4.1 canrm

canrm

Command

This is actually a disk command, not a tape command

usage: canrm FILE ...

canunlinks the given files on the distributed disks. Refuses to remove tape files. If multiple filenames are given, then as many as possible will be removed (i.e. canrm will not exit after the first failure). An error is returned if any of the files could not be unlinked.

4.2 canls

canls

Command

This is actually a disk command, not a tape command

```
usage: canls [-l] DISK ...
```

```
-l          long listing, all information in candir output  
-u user    list files for the specified user
```

gives a directory listing of a distributed disk. Only files owned by your effective userid are listed (or the user specified with `-u`). The default output can be used directly in shell variables or `' '` because it lists the filenames in their simplest form, one per line.

```
Example: canls disk8b                # to list disk8b  
        canrm 'canls disk8b | grep B59' # to remove all B59 files on disk8b
```

4.3 map

map

Command

```
usage: map -n conf tapemap ...
```

```
-n conf      choose configuration number  
-h          help
```

```
tapemap     lookup file to convert from configuration number to filename
```

Uses a tapemap file to convert a configuration number into an appropriate filename.

Example, for the tapemap file 'Could-09':

```
004000 Could0-09#coul_oooy_6.1_004000  
008000 Could0-09#coul_oooy_6.1_008000
```

```
$file='map -n 008000 Could-09' #gives Could0-09#coul_oooy_6.1_008000
```

The required two column lookup tables can be generated automatically using 'tapemap'.

4.4 newconfigs

newconfigs

Command

```
usage: newconfigs -n configs start sweeps
```

```
-n files      number of configurations  
-h           help
```

```
start starting configuration number (include leading zeros)  
sweeps      sweep count between configurations
```

Generates a list of configuration numbers.

e.g. `newconfigs -n 100 004000 2000`

```
004000  
006000  
008000...
```

4.5 newinits

newinits

Command

```
usage: newinits -n volumes FILE ...
```

```
-n volumes    number of tapes in a set (e.g. 2 or 4)  
-h           help
```

```
FILE file containing maps to tape filenames
```

Generates an init authorisation file for a set of map files. The init authorisation file contains a list of tapes which can be initialised, and the number of tapes in each set (for canreserve name `-tapes`).

```
e.g newinits -n 4 B53qf0.1423_d.map  
    B53qf0.1423_d.1 -4  
    B53qf0.1423_d.2 -4  
    .....
```

4.6 newtapemap

newtapemap

Command

```
usage: newtapemap -n files -f tapefile FILE
```

```
-n files      number of files per tape
-f tapefile   specify tape filenames using $n and $conf
-h           help
```

FILE file containing list of configuration numbers

Generates a mapping from configuration number to filename for a general tape file. The file name can include the variables

```
$n the 'tapeset' number
$conf the configuration number
```

This lookup table is needed by the 'map' command, which converts a single configuration number to a full filename.

e.g. `newtapemap -n 20 -f 'Could${n}-09#coul_ooy_6.1_${conf}' configs`

```
004000 Could0-09#coul_ooy_6.1_004000
008000 Could0-09#coul_ooy_6.1_008000...
```

4.7 tapeinit

tapeinit

Command

```
usage: tapeinit TAPESET ...
```

```
    -h      help
```

```
initialise the specified tapesets, after checking that the tapeset
does not already exist and looking for permission in an authorisation
file in the current directory. The initialisation file must be called
@INITS.
```

4.8 tapemap

tapemap

Command

```
usage: tapemap [-r regexp] tapeset ...
```

```
-r regexp    select files containing the given perl regular expression
-h          help
tapeset     specify tapesets, using file pattern
```

Lists the mapping from configuration number to filename for the given tapesets. The information is taken from the CANTapes directory. This lookup table is needed by the 'map' command, which converts a single configuration number to a full filename.

```
e.g. tapemap -r oooy 'Could*-09'
```

```
004000 Could0-09#coul_oooy_6.1_004000
008000 Could0-09#coul_oooy_6.1_008000...
```

The [-r] regexp option can be used to avoid including any spurious files that happen to be on the tape.

5 New canopy commands

To run canopy programs under `run` they need to have an interface with command line options, since it is difficult to use stdin flexibly from a Makefile.

There are perl "wrappers" for most of the existing canopy programs, to convert them to a command line form.

5.1 archive_field_file

archive_field_file

Command

```
usage: archive_field_file -r recs src dest
```

```
-r recs      set number of RECORDS_PER_FILE
-f          force, delete any existing file and recopy
-h          help

src         input field file or directory (e.g. disk#filename)
dest       output field file
```

Copies a field file from disk to tape. Skips the copy if the destination file already exists (unless using `-f`, force). For safety it will NEVER canunlink or overwrite anything, even using `-f`. Using the `-f` option will cause the script to exit with a warning if you try to overwrite existing tape files.

The number of `RECORDS_PER_FILE` should be 1 for gauge fields and 12 for quark fields.

For convenience, source directories of `disk8a`, `disk8b`, ... will have the filename of the destination appended automatically.

Example: `archive_field_file -r 1 disk8b Could0-09#coul_ooy_6.1_004000`

5.2 archive_gauge_field

archive_gauge_field

Command

```
usage: archive_gauge_field [-r recs] src dest
```

```
-r recs      set GAUGE_RECORDS_PER_FILE (defaults to 1)
-f          force, delete any existing file and recopy
-h          help

src         input field file or directory (e.g. disk#filename)
dest       output field file
```

Copies a gauge field file from disk to tape. Skips the copy if the destination file already exists (unless using `-f`, force). For safety it will NEVER canunlink or overwrite anything, even using `-f`. Using the `-f` option will cause the script to exit with a warning if you try to overwrite existing tape files.

For convenience, source directories of `disk8a`, `disk8b`, ... will have the filename of the destination appended automatically.

Example: `archive_gauge_field -r 1 disk8b Could0-09#coul_ooy_6.1_004000`

5.3 archive_quark_field

archive_quark_field

Command

```
usage: archive_quark_field [-r recs] src dest
```

```
-r recs      set FERMION_RECORDS_PER_FILE (defaults to 12)
-f          force, delete any existing file and recopy
-h          help

src         input field file or directory (e.g. disk#filename)
dest       output field file
```

Copies a quark field file from disk to tape. Skips the copy if the destination file already exists (unless using `-f`, force). For safety it will NEVER canunlink or overwrite anything, even using `-f`. Using the `-f` option will cause the script to exit with a warning if you try to overwrite existing tape files.

For convenience, source directories of `disk8a`, `disk8b`, ... will have the filename of the destination appended automatically.

Example:

```
archive_quark_field -r 1 disk8b D41qf0.1390_d.1#D41_qf_d_d_0.1390_1.4_004000
```

5.4 get_field_file

get_field_file

Command

```
usage: get_field_file -r recs src dest
```

```
-r recs      set number of RECORDS_PER_FILE
-f          force, delete any existing file and recopy
-h          help
```

```
src         input field file (e.g. tape#filename)
dest        output field file or directory
```

Copies a field file from tape to disk. Skips the copy if the destination file already exists (unless using `-f`, force).

The number of `RECORDS_PER_FILE` should be 1 for gauge fields and 12 for quark fields.

For convenience, destination directories of `disk8a`, `disk8b`, ... will have the filename appended automatically.

Example: `get_field_file -r 1 Could0-09#coul_ooy_6.1_004000 disk8b`

5.5 get_gauge_field

get_gauge_field

Command

```
usage: get_gauge_field [-r recs] [-f] src dest
```

```
-r recs      set GAUGE_RECORDS_PER_FILE (defaults to 1)
-f           force, delete any existing file and recopy
-h           help
```

```
src          input field file (e.g. tape#filename)
dest         output field file or directory
```

Copies a field file from tape to disk. Skips the copy if the destination file already exists (unless using -f, force).

For convenience, destination directories of disk8a, disk8b, ... will have the filename appended automatically.

Environment variables: GAUGE_RECORDS_PER_FILE

5.6 get_quark_field

get_quark_field

Command

```
usage: get_quark_field [-r recs] [-f] src dest
```

```
-r recs      set FERMION_RECORDS_PER_FILE (defaults to 12)
-f           force, delete any existing file and recopy
-h           help
```

```
src          input field file (e.g. tape#filename)
dest         output field file or directory
```

Copies a field file from tape to disk. Skips the copy if the destination file already exists (unless using -f, force).

For convenience, destination directories of disk8a, disk8b, ... will have the filename appended automatically.

Environment variables: FERMION_RECORDS_PER_FILE

5.7 make_2pt

make_2pt Command

```
usage: make_2pt -t type -g gauge [-n lat] -o dir -t string [-f] [-z]
          -q quark1 --src quark2 --snk sf

-m meson      specify meson, hh (heavy-heavy), hl (heavy-light)
               or ll (light-light)

-g gauge      specify input gauge field filename
-q quark1     specify input quark field filename (quark 1)
--src quark2  specify source smeared quark field filenames (quark 2)
--snk sf      specify sink smearing field filenames

-t string     specify output filename tags (kappa_conf is recommended)
-o dir        specify output directory
-z           make compressed tar file instead of output directory

-n lattice    set lattice size, (e.g. '12,12,12,24')
```

-f force, delete any existing file and recompute [N/A]
-h help

Computes correlators with additional smearing at the sink.

```
(quark1). snk . (quark2 src)
```

For light-light mesons the first quark line (quark 1) should be delta-delta. The second quark line (quark 2) can include smearing at the source. Neither quark line should include smearing at the sink.

For heavy-heavy mesons the first quark line is not required, as all the source combinations are calculated.

The source and sink filenames are given as

```
--src label:qf_filename,label:qf_filename, ...
--snk label:sf_filename,label:sf_filename, ...
```

where 'delta' is a possible smearing function filename, e.g.

```
--src d:disk8b#C51_qf_d_d_0.140_001000,1S:disk8b#C51_qf_1S_d_0.140_001000
--snk d:delta,1S:disk8b#C51_sf_1S_0.140
```

Environment variables:

LATTICE

```
GAUGE_RECORDS_PER_FILE (defaults to 1)
FERMION_RECORDS_PER_FILE (defaults to 12, i.e. all spin-color combinations)
make_2pt_NODES          (defaults to 64)
make_2pt_TIME           (defaults to 3:00)
```

5.8 make_quark

make_quark Command

```
usage: make_quark [-t algorithm] -g gauge [-s source] [-q quark] \
                 -n lattice [-b b.c.] -k kappa -c clover \
                 [-m sweeps] [-u sweeps] [-p sweeps] \
                 [-r rel_err] [-a alpha] [-d level]\
                 outputfile
```

-t algorithm choose algorithm type (make_quark5, make_quark=default)

-g gauge specify input gauge field filename

-s source specify input source (smearing) filename (default 'none')

-q quark specify input quark field filename (optional)

-n lattice choose lattice size (e.g. 12,12,12,24)

-b b.c. choose boundary conditions (periodic (default), antiperiodic)

-k kappa choose kappa value for inversion

-c clover choose clover coefficient

-m sweeps maximum number of sweeps (when required, default infinite)

-u sweeps refresh update interval (when required, default 1000)

-p sweeps print interval (when required, default 10)

-a alpha over-relaxation parameter (when required, default 1.0)

-r rel_err convergence criterion for relative error (when required)

-d level set debugging level (when available)

-f force, delete any existing output file and recompute

-h help

outputfile name of output quark field file

Computes a quark propagator for a given gauge field and source term. Skips the calculation if the output file already exists (unless forced, using -f). The optional input quark field allows for 'polishing' existing propagators. The boundary conditions can be 'periodic', 'antiperiodic' or a combination, for example,

```
-b p,p,p,a:23.5
```

gives a lattice which is antiperiodic in the time direction, with the boundary between timeslices 23 and 24(=0 on a 12³x24 lattice). A ':' indicates a newline in the input to the quark inverter prompts.

Environment variables:

make_quark_NODES (defaults to 64)

make_quark_TIME (defaults to 6:00)

5.9 invert_method

invert_method

Command

usage: invert_method [OPTIONS] outputfile

-t, --type algorithm	choose algorithm type (list algorithms with -t ?)
-g, --gauge file	specify input gauge field filename
-s, --source file	specify input source (smearing) filename
-q, --quark file	specify input quark field filename (optional)
-n, --lattice size	choose lattice size (e.g. 12,12,12,24)
-k, --kappa num	choose kappa value for inversion
-c, --clover num	choose clover coefficient
-m, --max-sweeps sweeps	maximum number of sweeps (default 1000)
-p, --print sweeps	print interval (default 50)
--rel err	convergence criterion for relative error (default 0)
--spins num	number of source spins (default 4)
--colors num	number of source colors (default 3)
--wilson-r num	fermion r parameter (default 1)
--next-r num	next neighbor r parameter
--k-ratio num	next neighbor kappa ratio, $k_2/kappa$
--accel-k num	kappa acceleration parameter
--accel-r num	r acceleration parameter
--omega num	over-relaxation parameter
-f, --force	delete any existing output file and recompute
-h, --help	help
outputfile	name of output quark field file

Computes a quark propagator for a given gauge field, source term and starting guess for the quark propagator. Skips the calculation if the output file already exists (unless forced, using -f). The optional input quark field allows for 'polishing' existing propagators.

Many algorithms have been implemented, they can be listed with `invert_method --type ?`

Environment variables:

`invert_method_NODES` (defaults to 64)

`invert_method_TIME` (defaults to 6:00)

5.10 check_quark

check_quark Command

```
usage: check_quark -g gauge -s source -q quark \
               [-n lattice] [-b b.c.] -k kappa -c clover
```

```
-g gauge      specify input gauge field filename (default 'none')
-s source     specify input source (smearing) filename (default 'none')
-q quark      specify input quark field filename (default 'none')

-n lattice    choose lattice size (e.g. 12,12,12,24)
-b b.c.       choose boundary conditions (defaults to periodic p,p,p,p)
-k kappa      choose kappa value for inversion
-c clover     choose clover coefficient

-h           help
```

Calculates the relative error of an existing solution of a quark inversion for a given gauge field and source term. The calculation is performed in double precision

Environment variables:

```
check_quark_NODES (defaults to 32)
check_quark_TIME   (defaults to 1:00)
```

5.11 diff_quark

diff_quark

Command

```
usage: diff_quark [-n lattice] quark1 quark2
```

```
-n lattice      choose lattice size (e.g. 12,12,12,24)
```

```
quark1         specify input quark field filename (default 'none')
```

```
quark2         specify input quark field filename (default 'none')
```

```
-h             help
```

Calculates the differences between two propagators, quark1 and quark2. The relative error and absolute error are both computed. The relative error is given by $|quark2-quark1|/|quark2|$.

Environment variables:

```
diff_quark_NODES (defaults to 32)
```

```
diff_quark_TIME   (defaults to 1:00)
```

5.12 make_source

make_source Command

usage: make_source -t type -a params [-n lat -r recs] outputfile

-t type	choose from delta, cube, exp1s, gauss, exp2s, exp1px, exp1py, exp1pz, read_file_real, read_file_complex, gen_exp2s
-a params	set parameters (e.g. radius)
-n lat	set lattice size, (e.g. '12,12,12,24')
-r recs	set FERMION_RECORDS_PER_FILE
-f	force, delete any existing file and recompute
-h	help
outputfile	name of output field file

Computes a smearing function using /usr/home/bjg/challenge/canopy/make_source/d860/spread. Skips the computation if the output file already exists.

Environment variables:

LATTICE

FERMION_RECORDS_PER_FILE (defaults to 12, i.e. all spin-color combinations)

make_source_SOURCES (defaults to 0,0,0,0)

make_source_NODES (defaults to 64)

make_source_TIME (defaults to 1:00)

5.13 make_wf

make_wf Command
 usage: make_wf [-n lat] -c config -o dir -t string [-f] [-z] quark1 [quark2]

-c config	specify config id (e.g. 001000)
-t string	specify output filename tags (kappa_config is recommended, wf_config default)
-o dir	specify output directory
-z	make compressed tar file instead of output directory
-n lattice	set lattice size, (e.g. '12,12,12,24')
-f	force, delete any existing file and recompute [N/A]
-h	help
quark1	specify input quark field filename (quark 1)
quark2	specify input quark field filename (quark 2, if required)

Computes the wavefunctions of mesons made from the two quark propagators, quark1 and quark2.

$$wf(r) = \sum_x \text{quark1}(x) \text{Gamma quark2}(x+r) \exp(-ip(x+r))$$

If quark2 is not specified then the wavefunction of quark1 with itself is computed.

$$wf(r) = \sum_x \text{quark1}(x) \text{Gamma quark1}(x+r) \exp(-ip(x+r))$$

The output wavefunction files are pi_wf_config and ro_wf_config by default

Environment variables:

LATTICE
 GAUGE_RECORDS_PER_FILE (defaults to 1)
 FERMION_RECORDS_PER_FILE (defaults to 12, i.e. all spin-color combinations)
 make_wf_NODES (defaults to 64)
 make_wf_TIME (defaults to 3:00)

6 New unix utilities

This chapter describes some utilities which are convenient for dealing with jobs, their logfiles and output.

6.1 tardir

tardir

Command

usage: tardir [-h] [-f] directory ...

-f fast, don't wait before deleting
-h help

tars up the specified directories, verifies the tar files, and then deletes the original directories. The directories can be absolute or relative pathnames, with an optional trailing '/'.
The tarfile is left just above the directory, ready for untarring.

e.g. `tardir DATA_TREE/001000/`
converts the directory name `DATA_TREE/001000/` to `DATA_TREE/001000.tar.gz`

The tarfiles are compressed (`tar.gz`) automatically.

Tries to do things safely. There is a default 30 second delay before the deletion, to give you time to interrupt. This can be switched off with the `-f` option.

6.2 watch

watch

Command

```
usage: watch [directory|file] ...
```

```
-t  
-h    help
```

Tails all files and directories, including newly created files. Tailing a directory searches all subdirectories recursively (using find).

May get a bit slow on really large directory trees. Be selective about what you tail. To prevent the program becoming a memory hog, as it watches more and more files, it will automatically exit after 10 minutes.

When started it displays only those files which have been modified within the last hour.

6.3 dt

dt

Command

usage: dt [-h] FILE ...

-h help

Takes the ages of the specified files, sorts them and computes the differences to give the times between files. This can be used to measure how rapidly the files are being produced.

6.4 rprof

rprof

Command

```
usage: rprof [-d] [-e NUM] FILE ...
```

```
-d      print times as 'per day'  
-e NUM  estimate time to finish NUM jobs  
-s NUM  assume NUM streams (default 1)  
-h      help
```

Parses the run LOG output to produce profiling information. The average time taken to complete each job is printed, either in seconds or in 'per day' form. e.g. 3600 seconds per job = 24 per day

6.5 hms

hms

Command

usage: hms [-h] [FILE] ...

-h help

A filter which converts log files to a more readable form by changing large numbers of seconds to (days), hours, minutes, seconds.

The forms '# seconds', '# secs' and '# s' are recognised as times in seconds. Integer times are always recognised and decimals of the form '1.23 seconds' work too, but scientific formats do not.

6.6 logdate

logdate

Command

```
usage: logdate [message]
```

```
-h    help
```

```
prints a logging date in the form 'message'date.
```

Example:

```
logdate '# start: '  
# start: Wed Sep 13 13:55:59 CDT 1995
```

6.7 stream

stream

Command

```
usage: stream -n streams FILE ...
```

```
-n streams      number of streams
```

```
-h              help
```

```
FILE           file to be split into streams
```

takes inputs files and splits them into 'streams', writing new files FILEa, FILEb, FILEc... It has an effect which is orthogonal to 'split'.

e.g. `streams -n 3 txt` produces

```
txta txtb txtc
line1  line2  line3
line4  line5  line5
...    ...    ...
```

6.8 fixuphistory

fixuphistory

Command

```
usage: fixuphistory [-h] DIR ...
```

```
-h    help
```

```
DIR   history directory to be fixed
```

Fixes up inconsistencies in a history directory after a system crash. Searches for '.running' files which are associated with processes which have died -- renames them to '.err', and marks the directory as FAILED (as a warning).

```
Example: bfg|fncrdm> fixuphistory 0061/history
          searching 0061/history/002000
          0061/history/002000/read_ahead_gauge_field.running has crashed
          .....
          marking 0061/history/002000 as FAILED
          fixing 0061/history/002000/read_ahead_gauge_field.running
          bfg|fncrdm>
```


7 Run

run

Command

usage: run [options] target ...

```

-C DIRECTORY  change to DIRECTORY before doing anything
-d            debug, print lots of debugging information
-f FILE       read FILE as a runfile
-i           ignore errors from commands, retry anyway
-r           ignore checkpoint files, restart from scratch
-k           keep going when some targets can't be made
-n           don't actually run any commands; just print them
-v           print the version number and exit.

-h           help

```

7.1 Stopping everything in an emergency

7.1.1 Allowing existing jobs to continue running

When some disaster occurs you will want to shut down the script quickly. This is done by putting the script into **HALT** mode. When the script receives a **HALT** message, it will immediately stop launching new jobs. It will wait for any jobs that are still running, to keep an accurate record of whether they actually finished without error (in which case it will not be necessary to rerun them).

There are several ways to send a **HALT** message to the script,

`kill -INT pid` using the `pid` of the runscript.

Create a control file, if you are using `std_control_hooks.pl` containing the lines

```

# shut down
print "sending halt message...\n";
$run_status = 'HALT' ;

```

For a script running in the background, bring the script to the foreground, using `fg`, press **Control-C** once (and once only), and then put the script into the background again, using `bg`.

Once the runscript has received the halt message it will stop submitting new jobs. However, it will continue to wait for those jobs which are currently executing. Don't start a new runscript until the current one has actually finished. You'll know this has happened when you see the message FAILED or HALT on a line on its own,

```
.....
094000 check_quark_2S completed successfully [18768] 334 s
094000 2pt_analysis completed successfully [18955] 920 s
094000 FAILED 962 s
```

```
.....
094000 check_quark_2S completed successfully [18768] 334 s
094000 2pt_analysis completed successfully [18955] 920 s
094000 HALT
# mailing bjpg,ask with subject: 'runscript exited'
```

7.1.2 Killing all the jobs currently running.

This method will kill all the jobs running, but ensure that the history directory remains up to date. This is to be used in emergencies when the jobs cannot be allowed to continue running, or when the host machine is about to be rebooted.

First tell the script to go into HALT mode. For example, assuming the runscript process number is `pid`, just do

```
kill -INT pid
```

as before. You can find this `pid` by looking in the appropriate log output

```
bjg|fncrdm> grep "^run" LOG* | tail
.....
run 0.16 (Exp) (bjg) [19101]
```

where the `pid` appears in square bracket, or by checking your running processes with,

```
bjg|fncrdm> ps -fu bjpg | grep "run "
bjg 19101 9734 0 16:07:13 pts/28 0:01 /usr/bin/perl /usr/home/bjpg/challenge/runscript.
bjg 21971 12038 1 16:18:49 pts/21 0:00 grep run
```

Once the script has been put into `HALT` mode, send a `TSTP` signal to all the running jobs controlled by that script using

```
kill -TSTP pid
```

for the `pid` of the runscript itself. The runscript will then pass this signal on to the jobs that it has started, so there is no need to kill each process individually, the run script will do this automatically when it receives the `TSTP` message. The signal `TSTP` is passed on to all the jobs as the equivalent of pressing `Control-c`. You should watch the log output and directory to make sure that all the jobs respond, by failing. If not, send the `TSTP` signal to `run` again until they do. Some tape jobs will persist because they cannot distinguish a stop signal from a SCSI read error (which they retry). They will give up after several attempts, so you may need to send the signal this many times. This may get fixed in future version of `cancopy`.

Note, signals are complicated. Sometimes these things don't work. There is a difference when you are trying to kill jobs that you started in your current session, because your shell will interfere with the signals. The safest thing to do is log out and log in again and then kill them.

7.2 Restarting after a system crash

After a complete system crash, individual processes will have failed without being able to send their exit status back to `run`. This means that the history directory will be out of date. In particular, some jobs will appear to be running, even though they no longer exist.

The following example shows the effects of a real system crash – according to the history directory many processes appear to be running (although in fact none of them exist),

```
bjg|fncrd6> find history/ -name '*.running'
history/132000/invert_kappa_d.running
history/270000/invert_kappa_2S.running
history/168000/invert_kappa_d.running
history/168000/invert_kappa_1S.running
history/218000/read_ahead_gauge_field.running
history/218000/invert_kappa_d.running
history/218000/invert_kappa_1S.running
history/218000/archive_gauge.running
```

To restart the job, these incorrect files should be removed. There is a `fixuphistory` command to do this automatically,

```

bjg|fncrdm> fixuphistory ./history
searching ./history/132000
./history/132000/invert_kappa_d.running has crashed
.....
marking ./history/132000 as FAILED
fixing ./history/132000/invert_kappa_d.running
bjg|fncrdm>

```

At the simplest level, this can all be done by hand. If you have a rename command available you can do,

```
ren "*.running" "#1.err"
```

in each affected directory to convert all the ‘running’ files into ‘err’ files. If you have `gnu-find` this can (approximately) be done automatically with

```
find . -name '*.running' -exec mv {} {}.err \;
```

This isn’t possible with the version of `find` that is sold with most systems because they often only substitute for the filename characters `{}` once, you just end up with a single file called ‘`{}.err`’. Dumb or what. Obviously this is because the companies are only interested in your money and locking you into their products through incompatibility, not in actually providing good software. If they were interested in providing good software they would just ship the gnu utilities as part of their standard installation. Anyway, once the history directory correctly represents the state of the run just before the crash, the jobs can be restarted as normal, with

```
run
```

If there are ‘FAILED’ files in any of the directories then the `-i` option can be used to restart them. After a system crash ‘FAILED’ files are not usually generated because the crash happens too quickly. However, `run` will figure out what to do once the history directory is correct. If you decide that rerunning a given job will be impossible then you can add a ‘FAILED’ file by hand using,

```
touch FAILED
```

8 Runfile

8.1 Makefile Section

8.1.1 Exiting

The set of commands should try to avoid using `exit` commands . For example, a bad way to do things is

```
if [ $file = "none" ] ; then
    exit          # this is a bad way to exit
fi
get_field_file $file ...
```

The reason is that additional code is appended to your commands to do things like renaming the `.running` file to `.log`, showing successful exit status. The extra commands look something like this,

```
if [ $file = "none" ] ; then
    exit          # this is a bad way to exit
fi
get_field_file $file ...
# these commands are automatically appended by run
echo "finished at " `date`
mv thisfile.running thisfile.log
```

If the code `exits` prematurely then these housekeeping commands will not be executed. Normally `run` will fix up these files automatically when it notices that they have not been renamed, and issue a warning. However, if your `run` process has died then the `history` files could get out of date.

It is simplest to avoid exiting in the middle of the commands. For example, in this case we can easily rewrite the commands in a way which eliminates any potential problems.

```
if [ $file != "none" ] ; then
    get_field_file $file ...    # rewriting the 'if' statement is much better
fi
```

8.2 Tips and Tricks

By default, the first occurrence of a target in a Runfile is used. This allows you to override 'wildcard' targets, such as,

```
make_source_d:
    echo 'no need to make a delta source!'

make_source_${smearing}:
    make_source ....
```

where the special case `make_source_d` will be picked up before the wildcard `make_source_...` catches it.

However, sometimes it's useful to override a wildcard, while still making use of its definition. In the special case where you don't include any actions for a specific target, then the Runfile will fall through to any wildcards which match.

For example, to put files onto a tape in the order gauge, d, 1S, 2S, we could use the following rules,

```
archive_kappa_d: archive_gauge invert_kappa_d
    archive_quark_field $disk_qf{'d'} $tape_qf{'d'}

archive_kappa_1S: archive_kappa_d invert_kappa_1S
    archive_quark_field $disk_qf{'1S'} $tape_qf{'1S'}

archive_kappa_2S: archive_kappa_1S invert_kappa_2S
    archive_quark_field $disk_qf{'2S'} $tape_qf{'2S'}
```

which will work, but forces us to explicitly specify the action for each case. We can get the same effect more compactly by allowing the specific cases to fall through to a more general rule,

```
archive_kappa_d: archive_gauge
archive_kappa_1S: archive_kappa_d
archive_kappa_2S: archive_kappa_1S
archive_kappa_${smearing}: invert_kappa_${smearing}
    archive_quark_field $disk_qf{${smearing}} $tape_qf{${smearing}}
```

In this case, there are no actions for the specific cases `archive_kappa_d`, `archive_kappa_1S` and `archive_kappa_2S` — they all fall through to the general wildcard target `archive_kappa_`

`{smearing}`. The dependencies of the specific and wildcard targets are combined appropriately to have the same effect as in the longer explicit form. This can be very convenient.

8.2.1 STREAMS

be tape oriented

8.2.2 READ AHEAD

usually worth it

8.3 Useful lines to include in the perl header

8.3.1 Checking for correct userids

To make sure you're 'su'ed to run a job (so that a disk allocation is available to you) you can include a check in the perl header. For example, the following extract of code,

```
die "please su to pbm, this script uses disk8d!\n" if getpwuid($<) ne 'pbm' ;
```

will exit the script with an error if your userid is not 'pbm'. The function `getpwuid` returns the userid of the current effective user, `$<`. See the Perl Manual for more details.

9 User hooks and Variables

This chapter lists some of the variables and hooks available to the user. The best way to see how to use these is to look at the examples in the library functions.

9.1 Variables

There are some useful internal variables which are usually available. Modifying them is not recommended, as it could produce unpredictable results.

The variables can depend on the current context, for example when calling 'user_pid_failed_hook' \$job will be the name of the job which has just failed, rather than the most recently launched job.

\$run	Variable
gives the current run number	
@runs	Variable
is the list of all valid runs	
\$logdir	Variable
controls the directory where log files appear.	
\$job	Variable
is the name of a target, in the current context	
\$run_status	Variable
controls the overall behaviour of the outer loop. The value "HALT" prevents any further jobs being submitted.	
\$debug	Variable
causes verbose debugging information to be printed on stdout if set.	
@queue	Variable
an array containing the targets which have been submitted but are not yet running.	

%running{\$pid}=\$job	Variable
An array of running jobs, indexed by pid.	

9.2 Hooks

The following hooks are available to modify the behavior of `run`. To avoid conflicts it is advisable to name any new variables in the hooks with the prefix `$user_...`. All the internal variables are available in the hooks. Modifying them is not recommended, as it could produce unpredictable results.

user_control_hook	Function
called before jobs are launched	

user_exit_hook	Function
called before script finally exits	

user_job_hook	Function
called before tests are made to see if a job can be submitted	

user_limit_hook	Function
<i>max_pids used_pids free_pids</i> called when the <code>\$user_limit</code> switches on	

user_pid_done_hook	Function
called after a launched process exits successfully	

user_pid_failed_hook	Function
called when a launched process fails	

user_pid_limit_hook	Function
called when the <code>\$user_pid_limit</code> switches on	

user_run_control_hook	Function
called at the start of each new run in the outer loop	

user_run_done_hook	Function
called after the successful completion of a run in the outer loop	
user_run_failed_hook	Function
called when a run in the outer loop fails	
user_run_start_hook	Function
called at the start of a run when the message " starting... " appears	
user_sig_hook	Function
called whenever the script catch a signal (held in \$sig).	

10 Libraries

There are some standard libraries to perform useful and common tasks. They extend `run` by making use of hooks.

A library can be included in your own ‘`Runfile`’ with a `require` command in the `perl` section, for example,

```
:: perl

require 'std_email_hooks.pl' ; # send email to user if job fails
```

Since the `user_hooks` are set by these libraries, you should append your own commands to the hooks, rather than redefining them, if you want to add extra features.

10.1 B lattice defaults

std_B_lattice_defs.pl

Include file

contains default parameters for runs on the B lattice. It defines the environment variable `LATTICE=12,12,12,24` and reasonable `TIME` and `NODE` allocations for `make_quark`, `make_source` and `make_2pt`. These settings are not optimal, they are fairly generous on `TIME`. You can quickly get a job up and running using these defaults (without worrying too much that it will crash from a `TIMEOUT`), and then optimise the parameters once you have some timing information.

```
# reasonable default parameters for B lattice runs
```

```
$ENV{"LATTICE"} = "12,12,12,24" ;
```

```
# Creating smearing files
```

```
$ENV{"make_source_NODES"} = "12" ;
```

```
$ENV{"make_source_TIME"} = "3:00" ;
```

```
# Quark inversion
```

```
$ENV{"make_quark_NODES"} = "12" ;
```

```
$ENV{"make_quark_TIME"} = "4:00" ;
```

```
# Checking inversion
```

```
$ENV{"check_quark_NODES"} = "12" ;
```

```
$ENV{"check_quark_TIME"} = "0:30" ;
```

```
# Comparing with previous propagators
```

```
$ENV{"diff_quark_NODES"} = "12" ;
```

```
$ENV{"diff_quark_TIME"} = "0:30" ;
```

```
# Calculation of correlators
```

```
$ENV{"make_2pt_NODES"} = "12" ;
```

```
$ENV{"make_2pt_TIME"} = "0:30" ;
```

```
# Calculation of wavefunctions
```

```
$ENV{"make_wf_NODES"} = "12" ;
```

```
$ENV{"make_wf_TIME"} = "0:30" ;
```

10.2 C lattice defaults

std_C_lattice_defs.pl

Include file

contains default parameters for runs on the C lattice. It defines the environment variable `LATTICE=16,16,16,32` and reasonable `TIME` and `NODE` allocations for `make_quark`, `make_source` and `make_2pt`. These settings are not optimal, they are fairly generous on `TIME`. You can quickly get a job up and running using these defaults (without worrying too much that it will crash from a `TIMEOUT`), and then optimise the parameters once you have some timing information.

```
# reasonable default parameters for C lattice runs
```

```
$ENV{"LATTICE"} = "16,16,16,32" ;
```

```
# Creating smearing files
```

```
$ENV{"make_source_NODES"} = "16" ;
```

```
$ENV{"make_source_TIME"} = "3:00" ;
```

```
# Quark inversion
```

```
$ENV{"make_quark_NODES"} = "32" ;
```

```
$ENV{"make_quark_TIME"} = "6:00" ;
```

```
# Checking inversion
```

```
$ENV{"check_quark_NODES"} = "32" ;
```

```
$ENV{"check_quark_TIME"} = "0:30" ;
```

```
# Comparing with previous propagators
```

```
$ENV{"diff_quark_NODES"} = "32" ;
```

```
$ENV{"diff_quark_TIME"} = "0:30" ;
```

```
# Calculation of correlators
```

```
$ENV{"make_2pt_NODES"} = "32" ;
```

```
$ENV{"make_2pt_TIME"} = "1:00" ;
```

```
# Calculation of wavefunctions
```

```
$ENV{"make_wf_NODES"} = "32" ;
```

```
$ENV{"make_wf_TIME"} = "0:30" ;
```

```
1;
```

10.3 D lattice defaults

std_D_lattice_defs.pl

Include file

contains default parameters for runs on the D lattice. It defines the environment variable `LATTICE=24,24,24,48` and reasonable `TIME` and `NODE` allocations for `make_quark`, `make_source` and `make_2pt`. These settings are not optimal, they are fairly generous on `TIME`. You can quickly get a job up and running using these defaults (without worrying too much that it will crash from a `TIMEOUT`), and then optimise the parameters once you have some timing information.

```
# reasonable default parameters for D lattice runs
```

```
$ENV{"LATTICE"} = "24,24,24,48" ;
```

```
# Creating smearing files
```

```
$ENV{"make_source_NODES"} = "24" ;
```

```
$ENV{"make_source_TIME"} = "3:00" ;
```

```
# Quark inversion
```

```
$ENV{"make_quark_NODES"} = "162" ;
```

```
$ENV{"make_quark_TIME"} = "6:00" ;
```

```
# Checking inversion
```

```
$ENV{"check_quark_NODES"} = "128" ;
```

```
$ENV{"check_quark_TIME"} = "0:30" ;
```

```
# Comparing with previous propagators
```

```
$ENV{"diff_quark_NODES"} = "128" ;
```

```
$ENV{"diff_quark_TIME"} = "1:00" ;
```

```
# Calculation of correlators
```

```
$ENV{"make_2pt_NODES"} = "128" ;
```

```
$ENV{"make_2pt_TIME"} = "3:00" ;
```

```
# Calculation of wavefunctions
```

```
$ENV{"make_wf_NODES"} = "128" ;
```

```
$ENV{"make_wf_TIME"} = "1:00" ;
```

```
1 ;
```

10.4 Control hooks

`std_control_hooks.pl`

Include file

allow you to modify `Runfile` parameters while the script is running. It searches for a `control` file called, '`NAME.control`' if you invoked the script with `run NAME`. Alternatively the files '`PID.control`' and '`control`', where `PID` is the pid of the top-level run command, are always available.

Note that if you have several scripts running in the same directory they will all read the file '`control`'.

If a control file exists, it is regarded as arbitrary perl code and executed. A typical control file might contain the lines,

```
# don't start any more jobs, then wait for running jobs to finish
$run_status="HALT" ;
```

to shut down a script as soon as possible.

A control file will be executed once. Its modification time is recorded to prevent it being executed many times. If necessary you can reactivate a control file with a `touch`.

When a script is started all existing control files will be read. This means that a "halt" control file will always shutdown subsequent scripts immediately unless the control file is renamed or deleted.

A good use of this feature is to modify the number of nodes being used if machine conditions change. For example, if you are running with small numbers of nodes and the machine suddenly becomes empty you can use

```
print "# updated nodes to 128\n";
$ENV{'make_quark_NODES'}=128 ;
```

to make use of the empty nodes without stopping and restarting the script. This change becomes effective immediately. It is guaranteed to occur before the next job that is launched.

```
# check for 'control' file
# if it exists, execute the perl commands in it and rename it 'control.done'
```

```
$user_control_hook= q~  
  
for $user_control_file ( ( "$ARGV[0].control", "$$.control", "control")){  
  
    next if ! -f "$user_control_file" ;  
  
    @user_control_stat=stat($user_control_file) ;  
    $user_control_mtime=$user_control_stat[9] ;  
  
    next if (  
        $user_control_mtime == $user_control_last_mtime{$user_control_file}  
    ) ;  
  
    $user_control_last_mtime{$user_control_file} = $user_control_mtime ;  
    $user_control_run_status=$run_status ;  
  
    print "# reading $user_control_file...\n" ;  
    do "$user_control_file" ;  
    if ($?) {  
warn "# control: $@\n" ;  
    } else {  
print "# done\n";  
    } ;  
    print "# run status $run_status\n" if  
    ( $run_status ne $user_control_run_status ) ;  
  
}  
  
~ ;  
  
1 ;
```

10.5 Email hooks

std_email_hooks.pl

Include file

sends you email when things go wrong. It tends to send too much.

```
require "ctime.pl" ;

# send the error log if a process fails
$user_pid_failed_hook = q^
    push(@err_mail_msg,"==> ${logdir}/${run}/${job}.err <==") ;
    push(@err_mail_msg,join("","'cat ${logdir}/${run}/${job}.err'")) ;
    &mail("$ENV{'USER'}","process failed","$run $job failed\n\n" .
        'cat ${logdir}/${run}/${job}.err') if $mail_verbose ;
^ ;

# send a short message if a run fails
$user_run_failed_hook = q^
    unshift(@err_mail_msg,"run $run failed\n") ;
    &mail("$ENV{'USER'}","bad news","run $run failed") if $mail_verbose ;
^ ;

# add to summary message if a run fails
$user_run_done_hook = q^
    push(@err_mail_msg,"run $run completed successfully\n") ;
^ ;

# send a message if the whole script has exited
$user_exit_hook = q^
    &mail("$ENV{'USER'}","runscript exited",join("\n",@err_mail_msg)) ;
^ ;

sub mail {
    local ($user,$subject,$message)=@_ ;
    $user || do { $user="$ENV{'USER'}" ; } ;
    $subject || do { $subject="message from run script" ;};
    print "# mailing $user with subject: '$subject'\n" ;

    if ( ! $dry_run ) {
        open (MAIL,"| /usr/sbin/Mail -s '$subject' $user") ;
    } else {
        open (MAIL,">&STDOUT") ;
        print MAIL "==> Mail Message <==\n" ;
    } ;

    print MAIL "Running: [run] @ARGS\n" ;
    print MAIL "Script:   $0 [$$]\n" ;
    print MAIL "Directory: ",`pwd` ;
```

```
print MAIL "Start time: ",&ctime($^T) ;  
print MAIL "Exit time: ",&ctime(time),"\\n" ;  
print MAIL "Message: $message\\n" if $message ;  
  
close(MAIL) ;  
}  
  
1 ;
```

10.6 Signal hooks

std_sig_hooks.pl

Include file

will HALT the script when any INT signal is received (e.g. Control-C or kill -INT ...). A halt means that no further jobs will be launched, but that the script will wait for existing jobs to finish.

```
# std_sig_hooks.pl -- deal with signals
#

$user_sig_hook = q^

for $user_sig_hook_pid (keys %running) {
    print "$user_sig_hook_pid\t$running{$user_sig_hook_pid}\n" ;
} ;

&hook('user_control_hook') ;

# handle interrupt signals from the user
#
if ( $sig eq "INT" ) {
    $run_status='HALT' ;
    print "# halt (due to SIGINT)\n" ;
} ;

# attempt to deal with an imminent shutdown signalled by SIGTERM
#
if ( $sig eq "TERM" ) {
    $run_status='HALT' ;
    print "# halt (due to SIGTERM)\n" ;
    print "# initiating shutdown sequence...\n" ;
    $std_sig_hooks_killed = kill 'TSTP', $$ ;
    print "# sent SIGTSTP to $std_sig_hooks_killed processes\n" ;
} ;

^ ;

1 ;
```

10.7 Standard log directory hooks

std_log_dir_hooks.pl

Include file

Sets a default log directory of 'history/' unless there is a user-specified value of \$logdir in the perl header.

In the case of a dry run, using the option -n, the log directory become \$logdir.dryrun.

```
# use a log directory called 'history' or 'history.dryrun'

$user_setup_logdir_hook=q^

$logdir='history' unless $logdir ; # keep logs in 'history' unless specified

if ($dry_run) {
    $logdir="$logdir.dryrun" ; # send dry run logs to a separate directory
} ;

^ ;

1 ;
```

10.8 Standard read ahead

`std_read_ahead.pl`

Include file

Prepares an array `$next_run{$run}`, giving the next runs for each run. This allows convenient reading ahead for the next run to save time.

```
# std_read_ahead.pl
#
# prepare array of $next_run{$run} for specified runs

$user_start_hook=q^

print "# run next_run\n" if $debug ;

for ($i=0 ; $i < scalar(@runs) ; $i++) {
    $next_run=$runs[$i+1] ;
    $next_run{$runs[$i]}=$next_run ;
    print "$runs[$i] $next_run{$runs[$i]}\n" if $debug ;
} ;

^ ;

1 ;
```

10.9 Standard limit hooks

std_limit_hooks.pl

Include file

prints a warning when the pid limit is reached (typically 20 pids below the maximum number)

```
# print a warning message if the pid limit is reached
```

```
$user_pid_limit_hook='
```

```
print "# warning: pid limit reached, $user_pid_limit\n" ;
```

10.10 Standard hooks

std_hooks.pl

Include file

loads some of the above libraries. A convenience feature.

```
require "std_control_hooks.pl" ;  
require "std_sig_hooks.pl" ;  
require "std_logdir_hooks.pl" ;  
require "std_limit_hooks.pl" ;
```


11 An example runfile

In this chapter we will examine a real Runfile for generating wilson propagators and correlators on a small $8^3 \times 16$ lattice. The order of steps involved is

```
make sources (d,1S,2S)
invert (d,1S,2S)
generate smeared smeared correlators (d,1S,2S)x(d,1S,2S)
```

Here is the full Runfile,

```
runfile goes here
```

Beginning with the perl section, we define the list of runs by reading in the file '@CONFIGS',

```
:: perl
open(CONF,"<@CONFIGS") ; chop(@runs = <CONF>) ; close (CONF) ;
```

This is required – the variable @runs must be defined so that we know what runs to do. Now we read in some useful library routines, to set up defaults for the B lattice, email notification and other useful things,

```
require 'std_B_lattice_defs.pl' ; # import optimum canopy settings
require 'std_hooks.pl' ; # standard hooks
require 'std_email_hooks.pl' ; # send email to user if job fails
```

Now we need to set the lattice size, since the B lattice defaults are good enough for the number of nodes we need, but the lattice size (which is kept in the environment variable LATTICE) needs to be changed,

```
$ENV{'LATTICE'}="8,8,8,16" ; # actually running on  $8^3 \times 16$ 
```

Next we set the users who will get email when the job fails. This is kept in the environment variable USER

```
$ENV{'USER'}='bjg,onogi' ;
```

To prevent the job from running under the wrong userid we check the current user using the perl expression `getpwuid($<)`,

```
die "please su to bjg, this script uses disk8b!\n" if getpwuid($<) ne 'bjg' ;
```

Now we set up the perl variables that we need to specify the job, the directory name `b5.5`, `kappa` value and clover coefficient,

```
$DIR      = "b5.5" ;
$kappa    = "0.175" ;
$clover   = "0.0" ;
```

To define the smearing parameters we use a perl associative array, with the keys `'1S'` and `'2S'`,

```
$smearing_params{'1S'} = "0.75" ;
$smearing_params{'2S'} = "0.466,0.429" ;
```

Now we define `user_run_start_hook`, which will be called whenever a new configuration is started. A `hook` is a piece of code which allows some user-defined commands to be executed at a given point in a program.

In this definition we need to prevent variable substitution. We want the variables such as `$tape_gf` (which gives the current gauge file) to be redefined each time the hook is `eval`'ed. If the variables are substituted directly at this point then they cannot change as the `$run` variable changes.

To prevent variable substitution it is necessary to use single quotes `'`. However in general we might also want to single quotes in our definition of `user_run_start_hook` – we could do this by escaping the quote characters. Here we make use of a perl trick by using `q^` which temporarily changes the single quote character `'` to a `^` (or whatever other character you choose, e.g. `q/`, `q%`, etc).

Everything that appears between `q^` and the final `^` is included in the definition unchanged, i.e. variables are not substituted at this point.

```
$user_run_start_hook = q^
    $conf=$run ;
```

```

chop($tape_gf = 'map -n ${run} Tapes/Coulb.map' ) ;
($tapeset_gf,$file_gf) = split("#",$tape_gf) ;
$disk_gf      = "\"coulomb/$file_gf\"";
$disk_sf{'d'} = "none" ;
$disk_sf{'1S'} = "disk8b#${DIR}_sf_${kappa}_1S" ;
$disk_sf{'2S'} = "disk8b#${DIR}_sf_${kappa}_2S" ;
$disk_qf{'d'}  = "disk8b#${DIR}_qf_d_d_${kappa}_${conf}" ;
$disk_qf{'1S'} = "disk8b#${DIR}_qf_1S_d_${kappa}_${conf}" ;
$disk_qf{'2S'} = "disk8b#${DIR}_qf_2S_d_${kappa}_${conf}" ;

```

In this definition we first define a *\$conf* variable for convenience,

```
$conf=${run} ;
```

Then we read in appropriate tape file name for this run, using the map command on tapemap 'Tapes/Coulb.map' within backquotes,

```
chop($tape_gf = 'map -n ${run} Tapes/Coulb.map' ) ;
```

Surrounding everything with a `chop` command is the perl way of removing the trailing newline from the end of the filename returned by the `map` command.

Now we split the tape filename (which looks something like 'Coulb1-10#Coul_5.5_8^3x16_gf_001000') into the tapeset and file,

```
($tapeset_gf,$file_gf) = split("#",$tape_gf) ;
```

Now we can construct all the other filenames that we will need,

```

$disk_gf      = "\"coulomb/$file_gf\"";
$disk_sf{'d'} = "none" ;
$disk_sf{'1S'} = "disk8b#${DIR}_sf_${kappa}_1S" ;
$disk_sf{'2S'} = "disk8b#${DIR}_sf_${kappa}_2S" ;
$disk_qf{'d'}  = "disk8b#${DIR}_qf_d_d_${kappa}_${conf}" ;
$disk_qf{'1S'} = "disk8b#${DIR}_qf_1S_d_${kappa}_${conf}" ;
$disk_qf{'2S'} = "disk8b#${DIR}_qf_2S_d_${kappa}_${conf}" ;

```

Note that we have to use some trickery on definition of `$disk_gf`. This name contains the characters `8^3x16`. The existence of a `^` character in a filename really confuses the bourne shell parser so we have to quote the filename with double quotes, `\"$filename\"`.

Moving on from the perl header to the main script, we simply list the commands we would like to execute (their definitions appear in the makefile section which follows). We prefix the commands with a `%` sign which shows that they refer to makefile commands.

```

:: script
%make_d_source
%make_1S_source
%make_2S_source

%invert_kappa_d
%invert_kappa_1S
%invert_kappa_2S

%2pt_analysis

%clean

```

It is possible to include normal shell commands (not prefixed by a percent sign `%`) in this section too but it is not usually necessary.

The makefile section explicitly lists all the commands needed to perform the job.

First we consider making the sources,

```

:: makefile
make_d_source:
    echo 'no need to make a delta source'

make_1S_source:
    make_source -t exp1s -a $smearing_params{'1S'} $disk_sf{'1S'}

make_2S_source:
    make_source -t gen_exp2s -a $smearing_params{'2S'} $disk_sf{'2S'}

```

Of course there is no need to make a delta source, but for symmetry reasons it is neater to regard a delta function as a source that is on the same level as the real sources 1S, 2S which have to be created.

If we make all the sources equivalent then we can write a single rule for doing the inversion, `invert_kappa_${smearing}`,

```
invert_kappa_${smearing}: make_${smearing}_source
    invert_method -t minimum_residual_red_black \
        -k 0.175 -n 8,8,8,16 \
        -g $disk_gf \
        -s $disk_sf${smearing} \
        -k $kappa -c $clover \
        -m 1000 -p 100 -rel 0 \
        --wilson-r 1 --next-r 0 --k-ratio 0 \
        --accel-k 0 --accel-r 0 --omega 1 \
        $disk_qf${smearing}
```

this can only begin once the corresponding "make_source" command is done. We choose a lot of parameters to control the inversion, we can split these over several lines using the backslash \ continuation character. Note that variables are substituted as you would expect (this is why we had to be careful with the definition of `user_run_start_hook`) – the associative arrays are handled properly too. We only need to put curly braces around variables where they are associative arrays or if it would be ambiguous if we didn't include them (e.g. is `$a_b` equivalent to `${a}_b` or `${a}_b` ?? if `a` and `a_b` are valid variables then both are possible and curly braces are needed to make everything unambiguous).

The `2pt_analysis` step can only proceed when all three inversions have finished,

```
2pt_analysis: invert_kappa_d invert_kappa_1S invert_kappa_2S
    make_2pt -m ll \
        -g $disk_gf \
        -o DATA_TREE/2-pt/${conf} \
        -t ${kappa}_${conf} \
        -q $disk_qf{'d'} \
        -z \
        --src d:$disk_qf{'d'},1S:$disk_qf{'1S'},2S:$disk_qf{'2S'} \
        --snk d:delta,1S:$disk_sf{'1S'},2S:$disk_sf{'2S'}
```

When and only when the `2pt_analysis` is finished we can safely remove all the quark field files,

```
clean: 2pt_analysis
       canrm  $disk_qf{'d'} $disk_qf{'1S'} $disk_qf{'2S'}
```

Once we have cleaned the disk all the jobs for this configuration are complete and the runfile will move onto the next configuration.

Concept Index

S

Sample index entry 11

Table of Contents

1	Introduction	1
2	A simple Runfile	3
2.1	The simple Runfile	3
2.2	Invoking run	3
2.3	The simple runfile in detail	4
2.4	Output from run	5
2.5	Errors and Restarts	7
3	Dealing with Tapes	11
3.1	Tape Maps	11
3.2	Making a map file for existing tapes	12
3.3	Making a map file for new tapes	13
3.4	Tape inits	16
4	New tape commands	19
4.1	canrm	20
4.2	canls	21
4.3	map	22
4.4	newconfigs	23
4.5	newinits	24
4.6	newtapemap	25
4.7	tapeinit	26
4.8	tapemap	27
5	New canopy commands	29
5.1	archive_field_file	30
5.2	archive_gauge_field	31
5.3	archive_quark_field	32
5.4	get_field_file	33
5.5	get_gauge_field	34
5.6	get_quark_field	35
5.7	make_2pt	36
5.8	make_quark	38
5.9	invert_method	39
5.10	check_quark	40
5.11	diff_quark	41

5.12	make_source	42
5.13	make_wf	43
6	New unix utilities	45
6.1	tardir	46
6.2	watch	47
6.3	dt	48
6.4	rprof	49
6.5	hms	50
6.6	logdate	51
6.7	stream	52
6.8	fixuphistory	53
7	Run	55
7.1	Stopping everything in an emergency	55
7.1.1	Allowing existing jobs to continue running	55
7.1.2	Killing all the jobs currently running	56
7.2	Restarting after a system crash	57
8	Runfile	59
8.1	Makefile Section	59
8.1.1	Exiting	59
8.2	Tips and Tricks	60
8.2.1	STREAMS	61
8.2.2	READ AHEAD	61
8.3	Useful lines to include in the perl header	61
8.3.1	Checking for correct userids	61
9	User hooks and Variables	63
9.1	Variables	63
9.2	Hooks	64
10	Libraries	67
10.1	B lattice defaults	68
10.2	C lattice defaults	69
10.3	D lattice defaults	70
10.4	Control hooks	71
10.5	Email hooks	73
10.6	Signal hooks	75
10.7	Standard log directory hooks	76
10.8	Standard read ahead	77

10.9	Standard limit hooks	78
10.10	Standard hooks	79
11	An example runfile	81
	Concept Index	87

