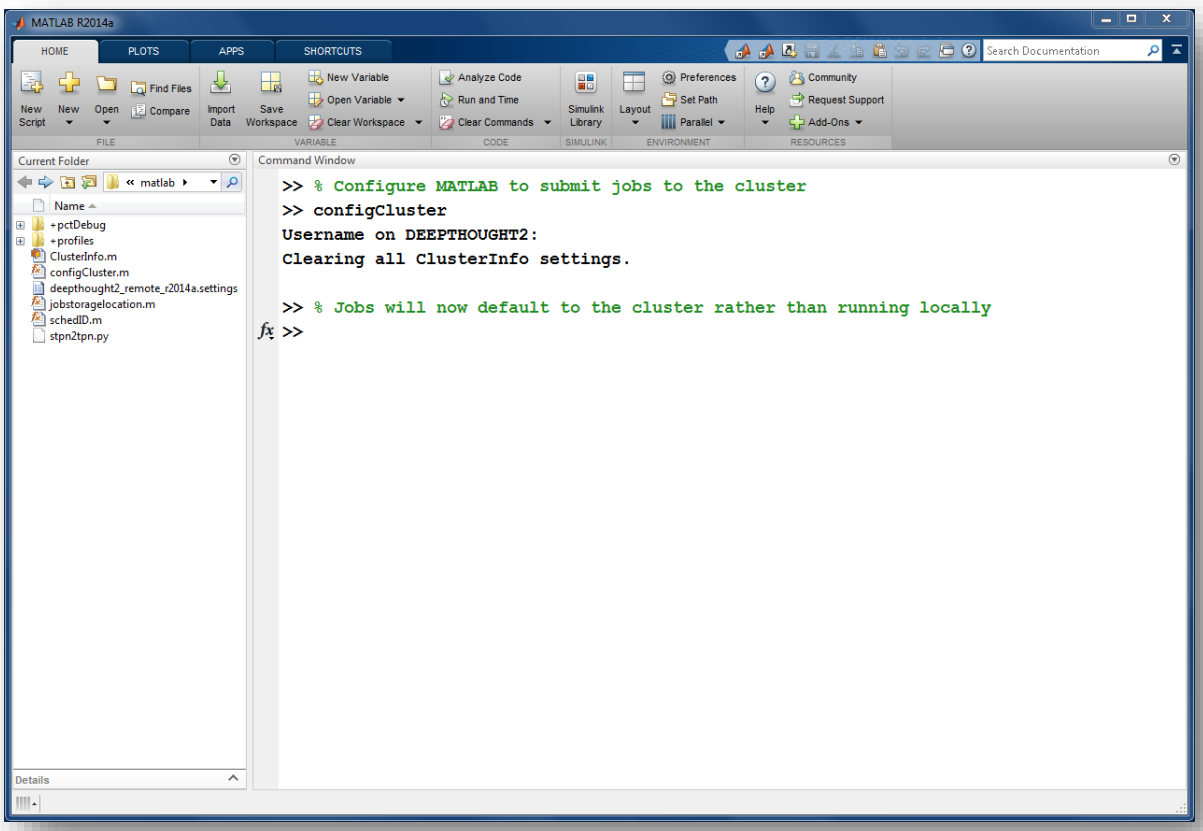
Getting Started with Serial and Parallel MATLAB on Deepthought2

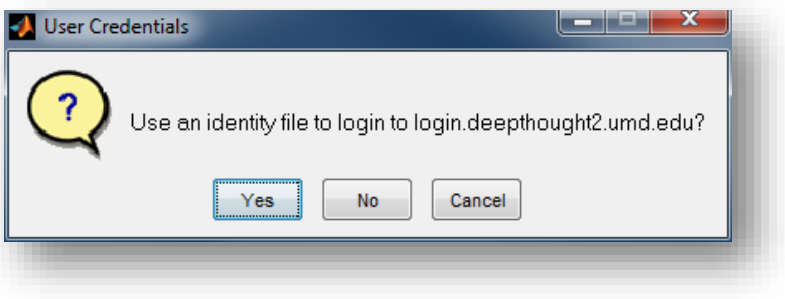
CONFIGURATION

- Download either `deepthought2.remote.r2014a.zip` (Windows) or `deepthought2.remote.r2014a.tar` (Linux/Mac)
- For Windows users, unzip the download and place the contents into the folder returned by `userpath` (for example `My Documents\MATLAB` or `Documents\MATLAB`).
- For Linux users, untar the download and place the contents into `$matlab/toolbox/local`
- Start MATLAB. Configure MATLAB to run parallel jobs on the Deepthought2 cluster by calling `configCluster`.



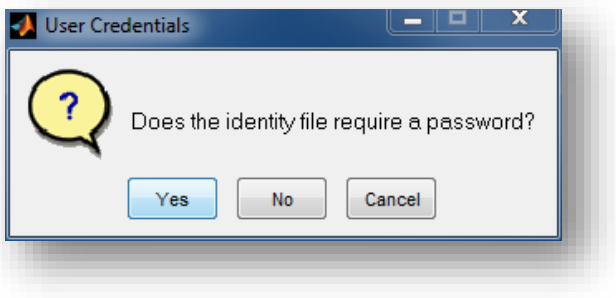
CREDENTIALS

The first time a user submits a job to Deepthought2, the user will be prompted whether to supply a password or a private key for their SSH credentials



If the user chooses a private key, the user will be prompted for the location of the file. The private key is stored with MATLAB so that they are not prompted for it at a later time.

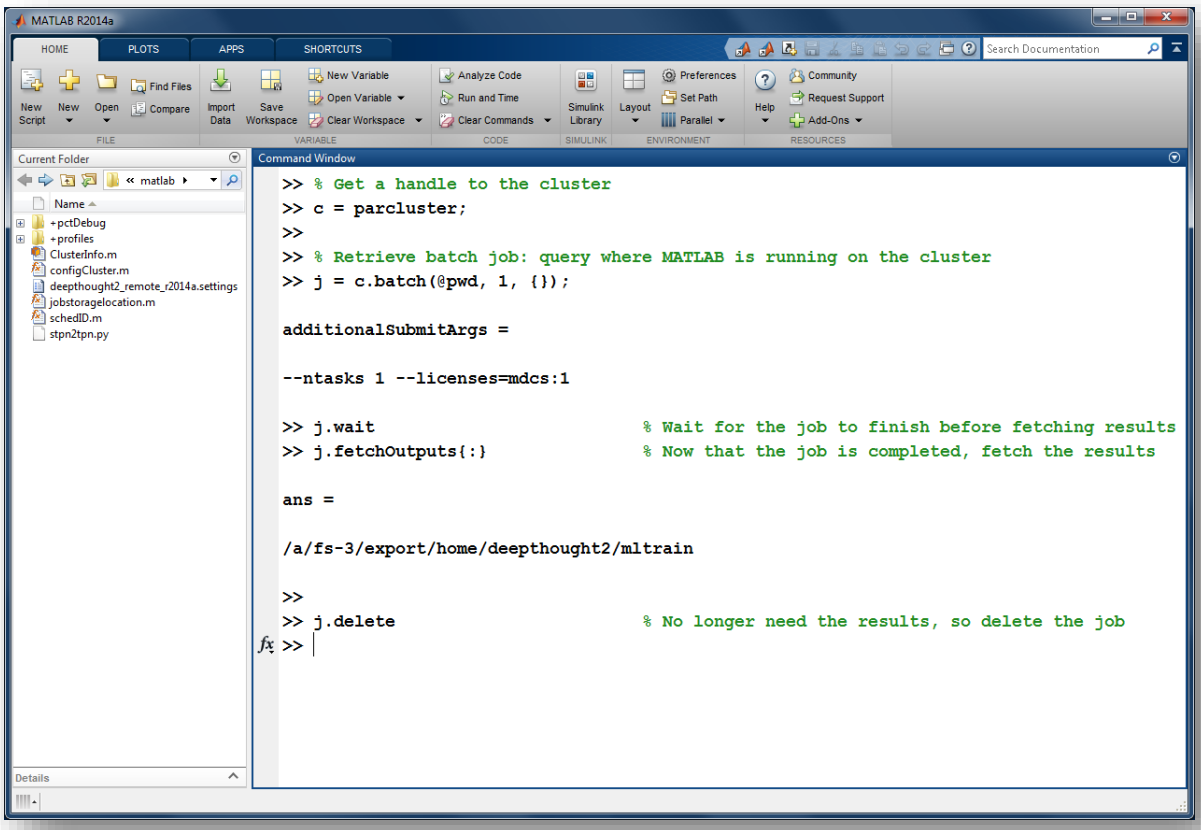
If using a private key, the user will also be prompted if the key requires a passphrase.



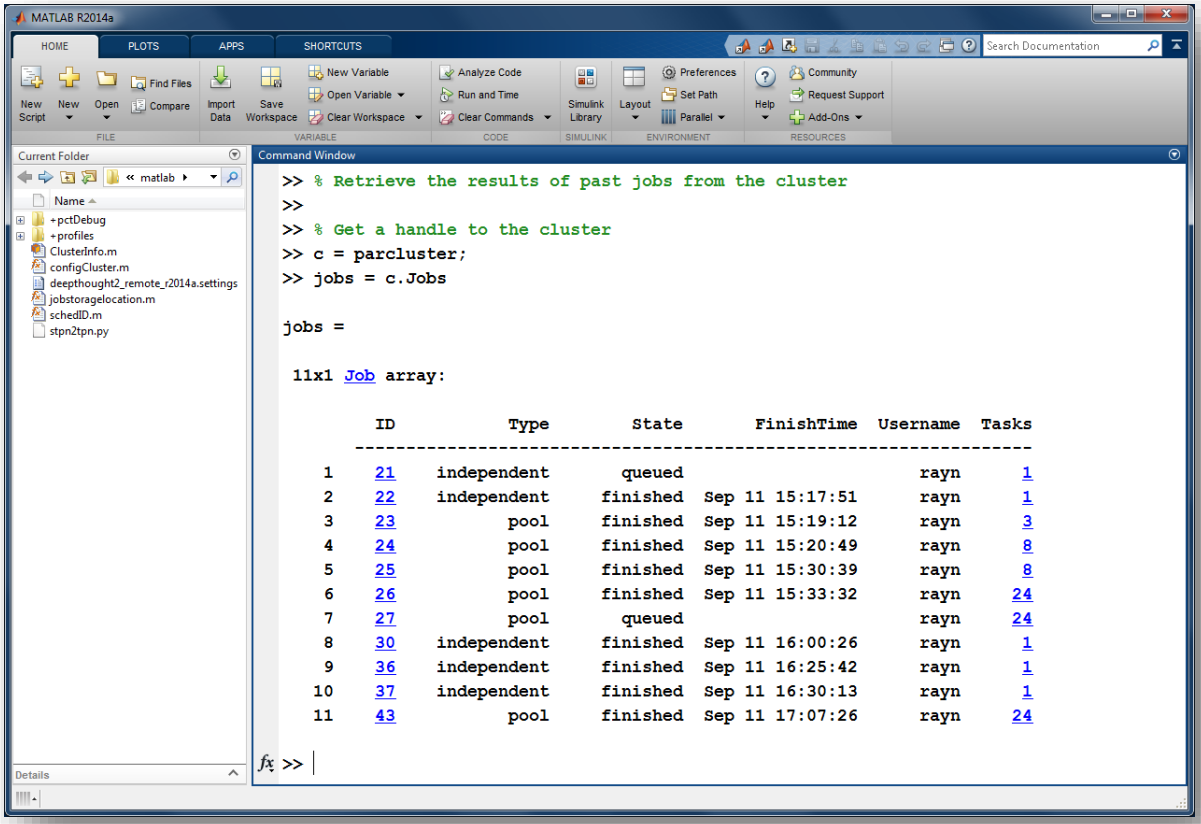
SERIAL JOBS

Use the `batch` command to submit asynchronous jobs to the cluster. The batch command will return a job object which is used to access the output of the submitted job. See the example below and see the MATLAB documentation for more help on `batch`.

Note: In the example below, wait is used to ensure that the job has completed before requesting results. In regular use, one would not use wait, since a job might take an elongated period of time, and the MATLAB session can be used for other work while the submitted job executes.

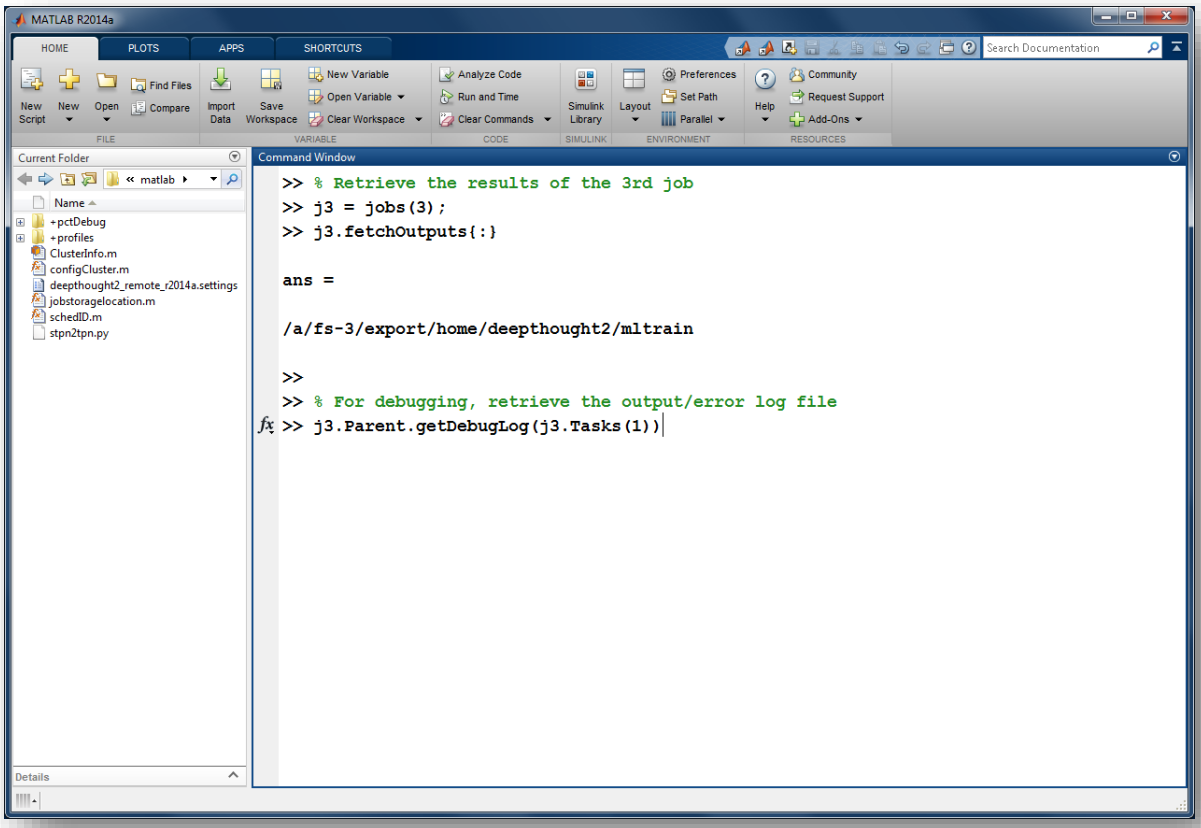


To retrieve a list of currently running or completed jobs, call `parcluster` to retrieve the cluster object. The cluster object stores an array of jobs that were run, are running, or are queued to run. This allows us to fetch the results of completed jobs. Retrieve and view the list of jobs as shown below.



Once we've identified the job we want, we can retrieve the results as we've done previously. If the job produces an error, we can call the `getDebugLog` method to view the error log file. The error log can be lengthy and is not shown here. The example below will retrieve the results of job #3.

NOTE: `fetchOutputs` is used to retrieve function output arguments. Data that has been written to files on the cluster needs be retrieved directly from the file system.



PARALLEL JOBS

Users can also submit parallel workflows with batch. Let's use the following example for a parallel job.

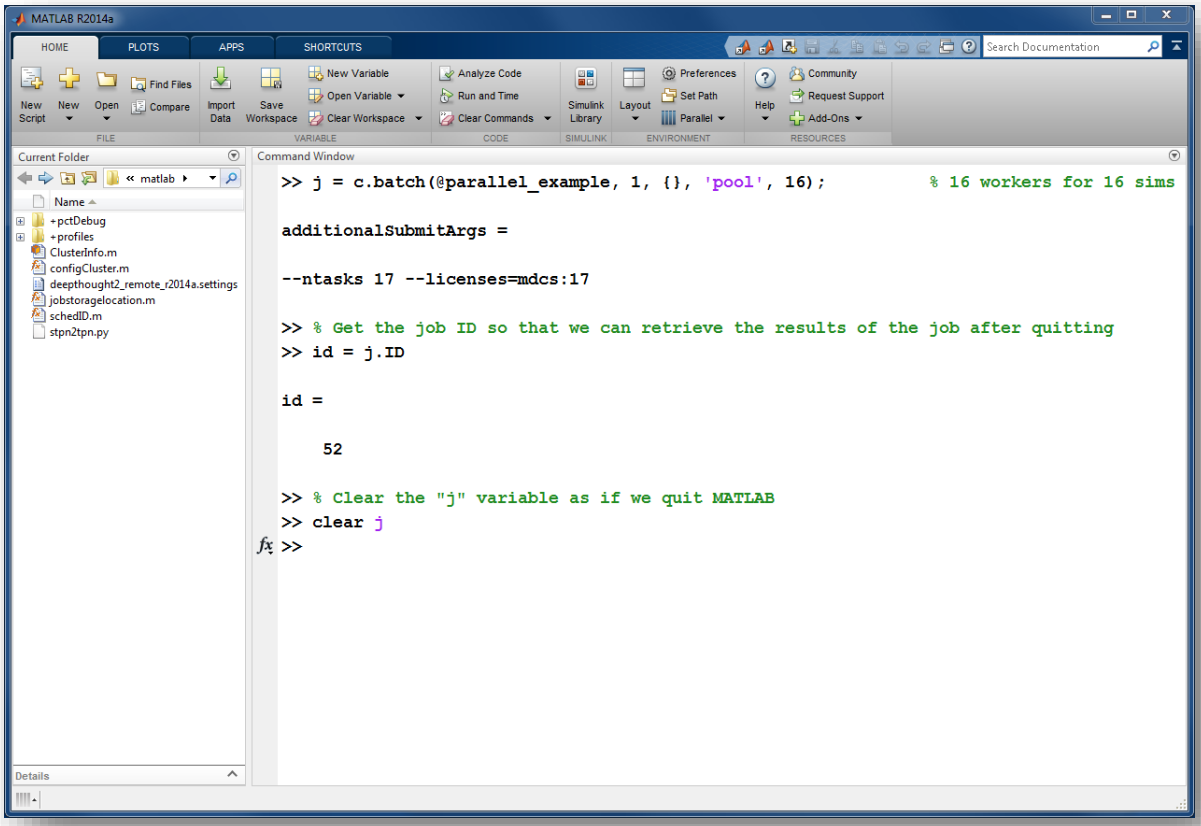
```
function t = parallel_example

t0 = tic;
parfor idx = 1:16
    A(idx) = idx;
    pause(2)
end
t = toc(t0);
```

We'll use the `batch` command again, but since we're running a parallel job, we'll also specify a MATLAB Pool.

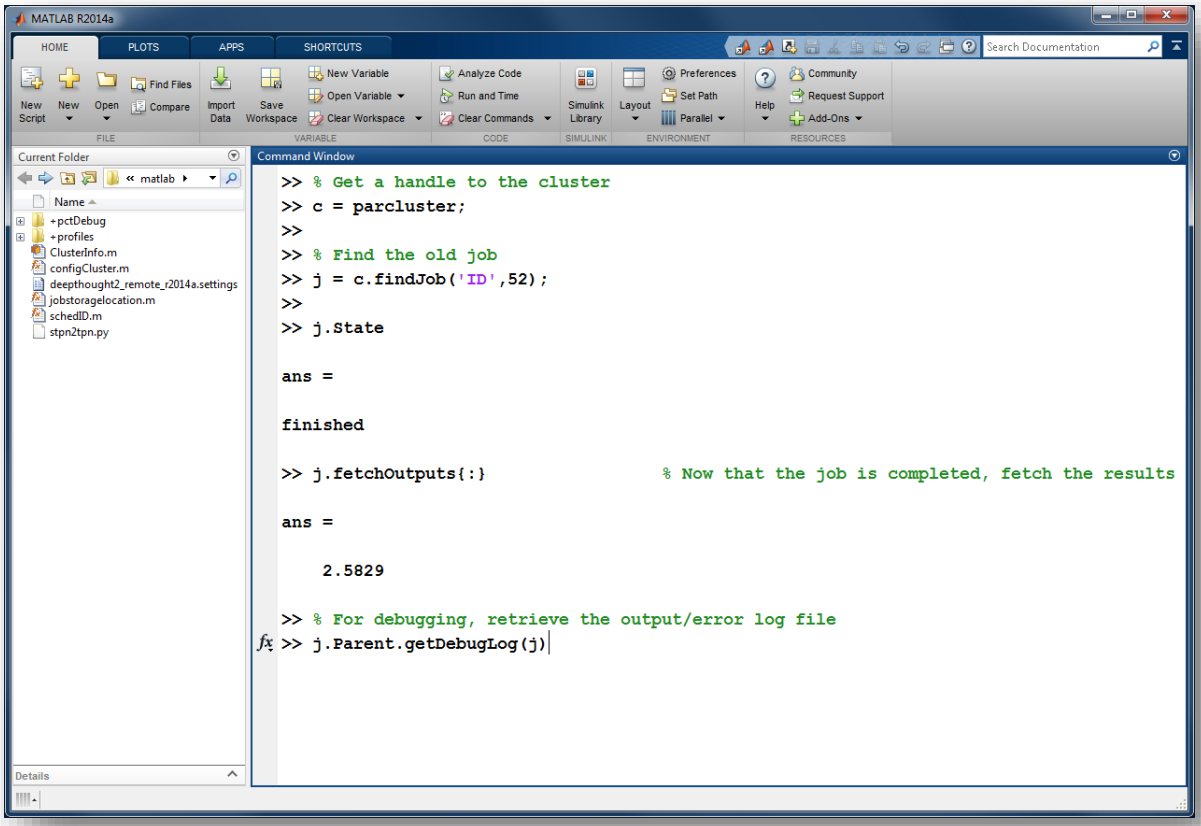
```
>> j = c.batch(@parallel_example, 1, {}, 'pool', 8);          % 8 workers for 16 sims

additionalSubmitArgs =

--ntasks 9 --licenses=mdcs:9

>> j.wait                                  % Wait for the job to finish before fetching results
>> j.fetchOutputs{:}                       % Now that the job is completed, fetch the results

ans =

          5.52

>>
```

The job ran in 5.52 seconds using eight workers. Note that these jobs will always request N+1 CPU cores, since one worker is required to manage the batch job and pool of workers. For example, a job that needs eight workers will consume nine CPU cores.

We'll run the same simulation, but increase the Pool size. Note, for some applications, there will be a diminishing return when allocating too many workers. This time, to retrieve the results at a later time, we'll keep track of the job ID.

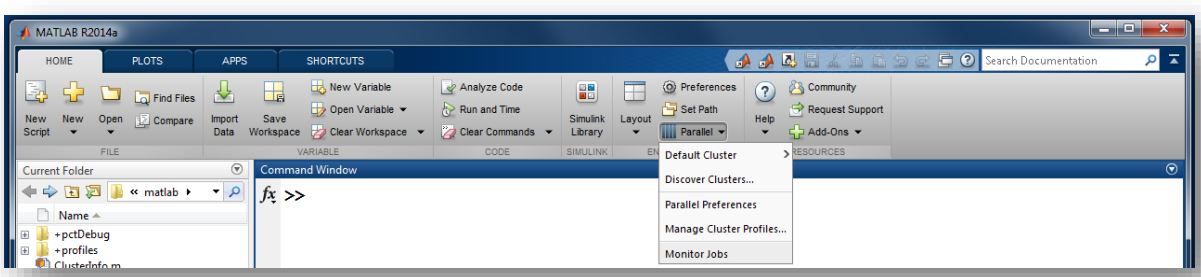


Once we have a handle to the cluster, we'll call the `findJob` method to search for the job with the specified job ID.



The job now runs in 2.58 seconds using 16 workers. Run code with different numbers of workers to determine the ideal number to use.

Alternatively, to retrieve job results via a graphical user interface, use the Job Monitor (Parallel > Monitor Jobs).



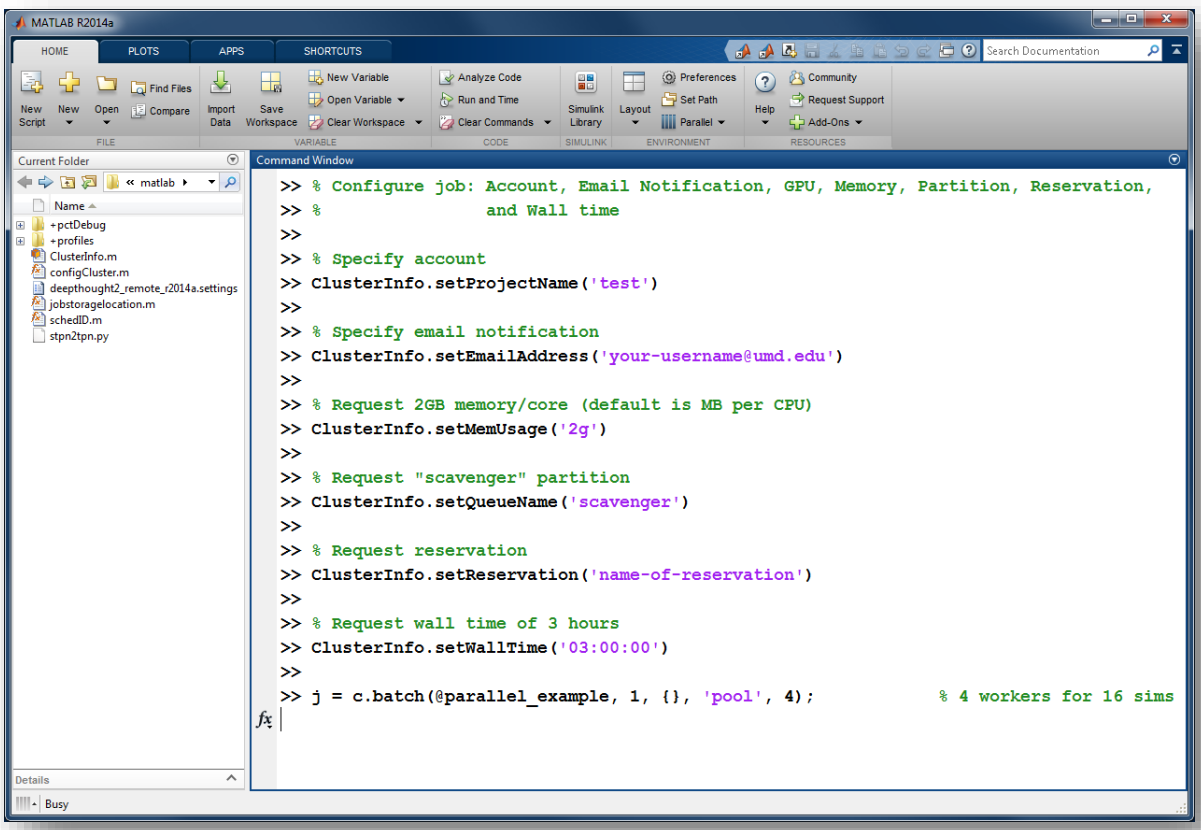
CONFIGURING JOBS

Prior to submitting the job, along with setting the wall time, we can also specify:

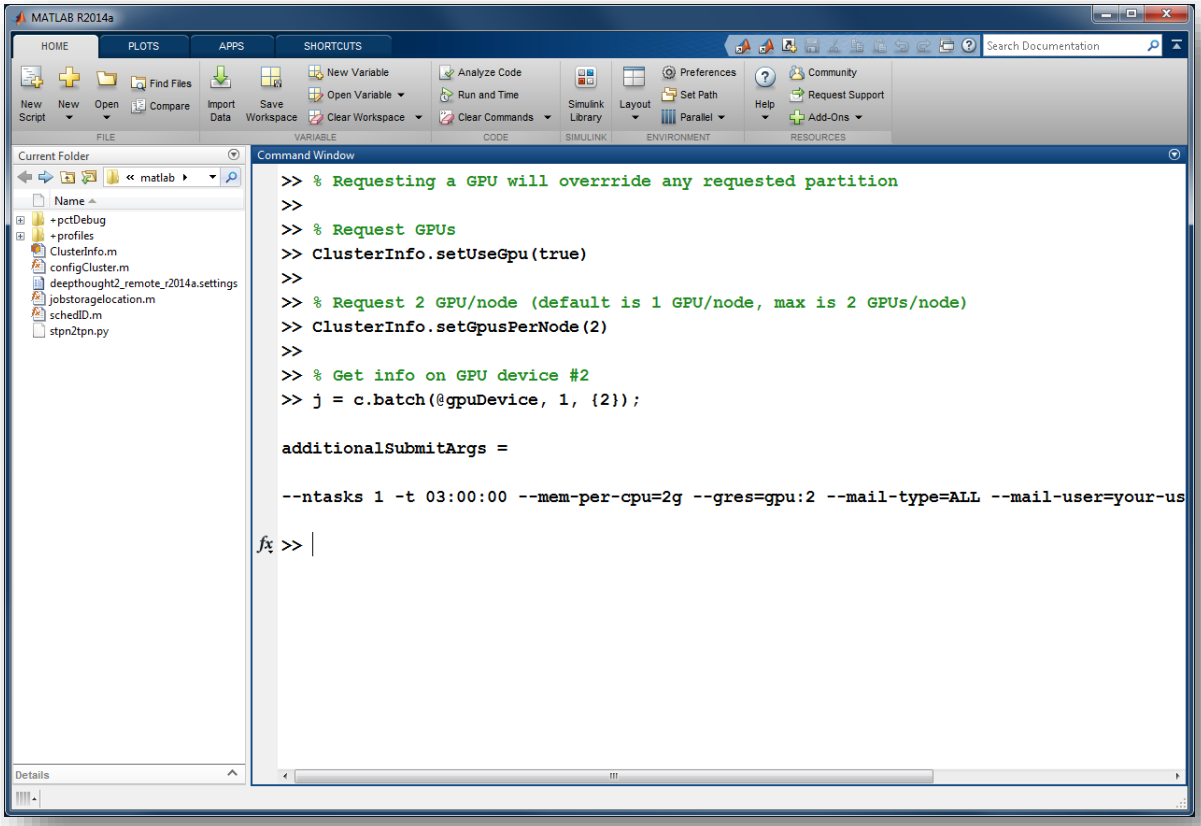
- Account
- Email Notification (when the job is running, exiting, or aborting)
- GPU
- Memory Usage
- Partition
- Reservation
- Wall time

Specification is done with `ClusterInfo`. The `ClusterInfo` class supports tab completion to ease recollection of method names.

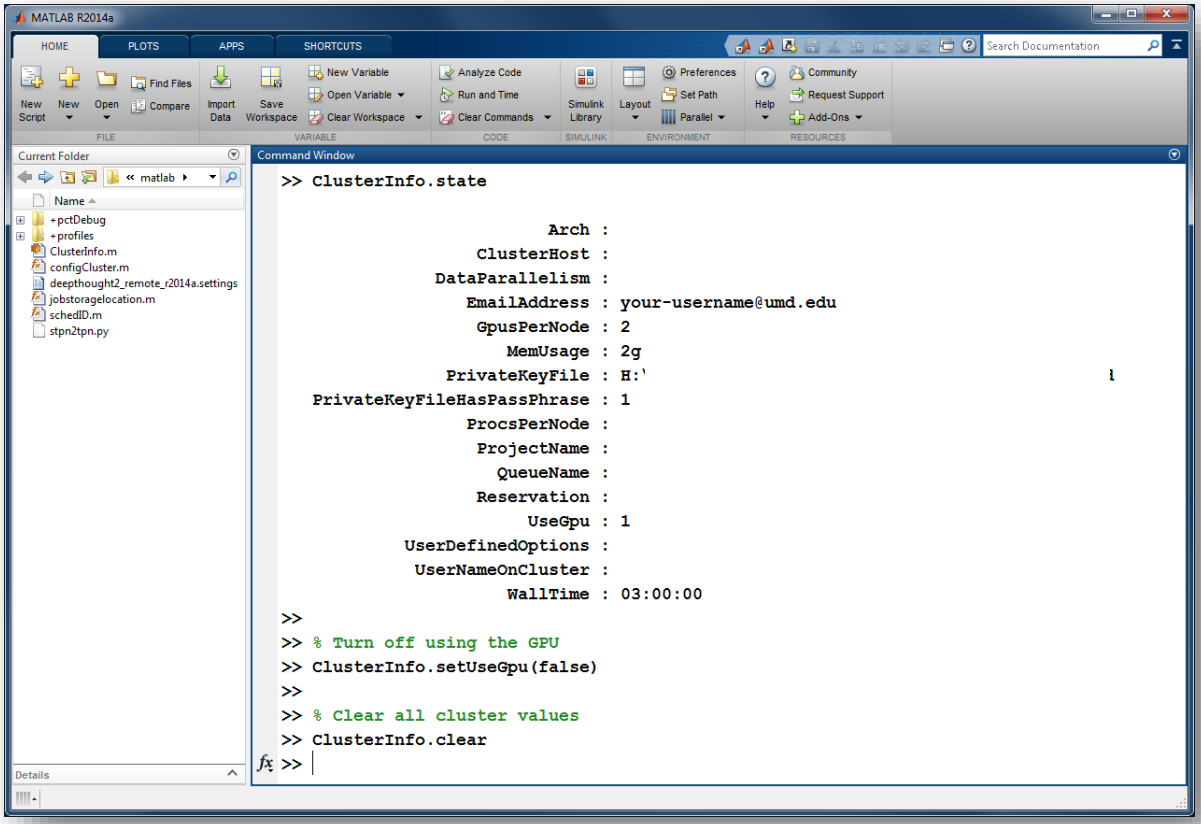
NOTE: Any parameters set with `ClusterInfo` will be persistent between MATLAB sessions.



An example of requesting a GPU



To see the values of the current configuration options, call the `state` method. To clear a value, assign the property an empty value (‘’, [], or false), or call the `clear` method to clear all values.



TO LEARN MORE

To learn more about the MATLAB Parallel Computing Toolbox, check out these resources:

- Parallel Computing Coding Examples
- Parallel Computing Documentation
- Parallel Computing Overview
- Parallel Computing Tutorials
- Parallel Computing Videos
- Parallel Computing Webinars