MAST Documentation

Release 20131216.canoe

University of Wisconsin-Madison Computational Materials Group

December 23, 2013

Welcome to the MAST documentation site!

CHAPTER

ONE

INTRODUCTION

Welcome to the MAterials Simulation Toolkit (MAST)!

link to whitepaper here?? MAST is intended to be an easy-to-use wrapper to facilitate complex sequences of calculations.

1.1 The MAST Kitchen

MAST uses kitchen terminology to organize the materials simulation workflow.

- An *Ingredient* is a single calculation, like a single VASP calculation resulting in a relaxed structure and energy.
- A *Recipe* is a collection of several ingredients and information about how the ingredients are combined together.
  - As in a cooking recipe, ingredients may need to be addressed in a certain logical order. This temporal order of how ingredients work together is the workflow.
  - The `Recipe Template` and *Input File* together describe the order of the ingredients and the way they are combined together.

For more information, see the MAST introductory powerpoint **need to add ppt here**

1.2 Computing in the MAST Kitchen

1. Plan your workflow.
   - What are the single calculations you will need (Ingredients)?
   - Which calculations depend on each other and should be grouped into a Recipe?
   - What are all of the conditions for each calculation (e.g. which ones can have volume change, and which ones should be at fixed volume? How fine a kpoint mesh does each calculation need? etc.)?
2. Start with some of the standard recipes in your $MAST_RECIPE_PATH directory or use a new template.
3. Create an input file, for example, test.inp.
4. Run the command `mast -i test.inp` to parse the input file.
5. Under `$MAST_SCRATCH`, MAST creates a timestamped recipe directory.
6. Within the recipe directory:
   (a) Each ingredient gets its own directory within the system_recipe_timestamp directory.
   (b) Additional files are created, including:

i. `personal_recipe.txt`, which is your recipe template file filled in with information gathered from the input `.inp` file.

ii. `archive_input_options.txt`, so you can see what the input options originally were

iii. `archive_recipe_plan.txt`, which tells you how MAST interpreted the recipe file. You can check this file to see which ingredients are considered parents of which other ingredients, for troubleshootin

iv. `status.txt`, which tells the status of all the ingredients.

v. `input.inp`, which is a copy of the input file (or an individual loop of a looped input file)

vi. `metadata.txt`, which stores metadata information

vii. `mast_recipe.log`, which stores recipe-level logging information.

7. Run the command `mast` to start the MAST scheduling arm. The MAST scheduler will get information from the personal_recipe.txt, input.inp, and status.txt file in the recipe folder.
8. When all ingredients in the recipe are complete, the recipe directory is moved into a `$MAST_ARCHIVE` directory.

For general examples on how to use MAST, see the presentations here **need presentations** Please check your output carefully, especially when setting up a new workflow using MAST.

CHAPTER

TWO

INSTALLATION

2.1 Installation on bardeen

2.1.1 1. Environment variables

MAST is installed in `//share/apps/MAST`.

Set the MAST environment variables. Add the following lines to your setup profile, such as `//home/username/.bashrc`, where `username` is your username. Replace all instances of `//home/username` with your actual username, like `//home/janedoe`.

The environment variables are:

```
export MAST_INSTALL_PATH=//share/apps/MAST
export MAST_RECIPE_PATH=//home/username/MAST/recipe_templates
export MAST_SCRATCH=//home/username/MAST/SCRATCH
export MAST_ARCHIVE=//home/username/MAST/ARCHIVE
export MAST_CONTROL=//home/username/MAST/CONTROL
export PYTHONPATH=$PYTHONPATH://share/apps/MAST
export VASP_PSP_DIR=//share/apps/MAST/vasp_pps
export PATH=$PATH://share/apps/MAST/bin
export PATH=//share/apps/EPD_64bit/epd_free-7.3-2-rh5-x86_64/bin:$PATH
```

An explanation of each variable appears below.

MAST_INSTALL_PATH: This variable should be set to the installation directory.

```
export MAST_INSTALL_PATH=//share/apps/MAST
```

MAST_RECIPE_PATH: MAST looks for recipe templates in this folder. You may want to copy recipes from the `$MAST_INSTALL_PATH/recipe_templates` directory into this folder and modify them.

```
export MAST_RECIPE_PATH=//home/username/MAST/recipe_templates
```

MAST_SCRATCH: This variable may be set to any directory. MAST will look for recipes in this directory.

```
export MAST_SCRATCH=//home/username/MAST/SCRATCH
```

MAST_ARCHIVE: This variable may be set to any directory. MAST will move completed recipes from `$MAST_SCRATCH` into this directory.

```
export MAST_ARCHIVE=//home/username/MAST/ARCHIVE
```

MAST_CONTROL: This variable may be set to any directory. MAST monitor log files, MAST monitor error files, and other MAST monitor output will be written to this directory.

```
export MAST_CONTROL=//home/username/MAST/CONTROL
```

PYTHONPATH: If this environment variable already exists, the MAST installation directory should be appended. Otherwise, this variable can be set to the installation directory. Assuming PYTHONPATH already has some value (use env to see a list of environment variables):

```
export PYTHONPATH=$PYTHONPATH://share/apps/MAST
```

VASP_PSP_DIR: This variable is necessary if VASP and VASP pseudopotential files are being used. See the documentation for the **:ref:'Materials Project's <http://materialsproject.org>'_ :ref:'pymatgen <http://pymatgen.org>'_** code. The VASP_PSP_DIR should be set to a path which contains folder such as POT_GGA_PAW_PBE (for functional PBE, or mast_xc PBE in Ingredients) or POT_GGA_PAW_PW91 (for functional PW91).

```
export VASP_PSP_DIR=//share/apps/MAST/vasp_pps
```

PATH: This variable should be appended with the `$MAST_INSTALL_PATH/bin` directory, for example:

```
export PATH=$PATH://share/apps/MAST/bin:PATH
```

2.1.2 2. Python version

Make sure that the correct version of python is defaulted to be used first. If you already use python for something else and this next line interferes with your other python calls (for example, you routinely use Python 2.4.3 instead and your other programs break if called from python 2.7.3), please see Tam.

```
export PATH=//share/apps/EPD_64bit/epd_free-7.3-2-rh5-x86_64/bin:$PATH
```

This version of python has pymatgen, numpy, and scipy in the appropriate versions.

- Type `which python` and you should get: `/share/apps/EPD_64bit/epd_free-7.3-2-rh5-x86_64/bin/pytho`
- Type `which mast` and you should get: `/share/apps/MAST/bin/mast`

2.1.3 3. Additional setup

1. Create all directories which do not yet exist (e.g., `mkdir //home/username/MAST`, `mkdir //home/username/MAST/recipe_templates`, ARCHIVE, CONTROL, and SCRATCH)
2. Make an empty file at `//home/username/MAST/CONTROL/submitlist`
3. Log out of all bardeen terminals and log back in. (You may also run `source ~/.bashrc`, but sometimes this doesn't quite set everything.)
4. (There are some additional Platform Support steps which have already been taken: Queue and submission script commands are in $MAST_INSTALL_PATH/submit and may need to be heavily modified depending on the platform used. To customize the queue submission behavior, copy the appropriate files out of $MAST_INSTALL_PATH/submit/platforms and into $MAST_INSTALL_PATH/submit, omitting the platform name, and modify the new queue_commands.py, script_commands.py, and submit.sh accordingly. This has already been done on bardeen. No step here.)

2.2 Test that MAST can run on bardeen

1. Copy the test recipe template to your recipe_templates folder:

```
cp //share/apps/MAST/recipe_templates/phonon_test_neb.txt //home/username/MAST/recipe_templates/
```

2. Make a test directory, like `//home/username/MAST/test`
3. Copy the test input file to your test folder:

   ```
   cp //share/apps/MAST/test/phononreorgtest/phonon_with_neb.inp //home/username/MAST/test/test.inp
   ```

4. Go to your test directory, `cd //home/username/MAST/test`
5. Try to parse the input file, entering the following command as one line:

   ```
   nice -n 19 mast -i test.inp
   ```

   - The .nice -n 19. keeps this command low priority, since it is being run on the headnode (but it is not too intensive).
   - The -i signals to MAST that it is processing an input file.

6. Your `//home/username/MAST/SCRATCH` directory should now have a folder with a very long name in it (recipe directory), which contains several subfolders (ingredient directories).
7. Go to that long recipe directory. (PhononNebTest...)
   - To see the input options:
     - `cat input.inp` (should be identical to test.inp since no looping was used)
       - Note that you can use other viewing commands, not just .cat., but be careful not to edit any of these files.
     - `cat archive_input_options.txt` (should show Al instead of element X1)
   - To see information about the ingredient relationships MAST detected from the recipe template:
     - `cat personal_recipe.txt`
     - `cat archive_recipe_plan.txt`
   - To see ingredient statuses at a glance:
     - `cat status.txt`
8. Run mast once: `nice -n 19 mast`
9. You should see a *mastmon* job appear on morganshort.
10. MAST should have detected that the first ingredient was ready to run, so when that process disappears, run mast again: `nice -n 19 mast`
11. Now you should see `perfect_opt1` appear on the queue.
12. `status.txt` in the recipe directory in `$MAST_SCRATCH` should show that `perfect_opt1` is queued.
13. If you forgot some step above (like you forgot to create the submitlist file) and are running into strange problems, delete the PhononNebTest... folder from `$MAST_SCRATCH` and start again from the beginning of this section.
14. The `$MAST_CONTROL` folder gives you error messages and other information. See *Running MAST* for tips.

2.3 Installation on another cluster

2.3.1 0. Pre-steps

- If you are on ACI/HPC, make sure you are using the compile node for all installation tasks. (aci-service-2 as of Dec. 2013) Use the submit node only to submit jobs.
- Have the owner of //tmp/pip-build remove the directory if it exists (see **:ref:'pip issue 729 <https://github.com/pypa/pip/issues/729>'_**):

```
cd //tmp
rm -r pip-build
```

2.3.2 1. Verify your Python version

Check your version of python: `python --version` If your version of python is not 2.7.3, try to locate an existing version of python 2.7.3.

- On platforms with modules, it is probably something like `module load python`, but get the correct version (`module avail` to see available modules). Type `which python` to make sure you have the right version, or `python --version`.
- DLX has python 2.6.6 normally. `module load Python`, even though it is 2.7.3, has some difficulties installing pymatgen, possibly because of the way the module system works. Follow the `install python` directions instead.
- On bardeen it is //share/apps/EPD_64bit/epd_free-7.3-2-rh5-x86_64

If you do not have python 2.7.3, install it.

Installing python

The EPD/Canopy version is preferred because it includes numpy and scipy already. Download this version from here: **:ref:'EPD Free Canopy <https://www.enthought.com/downloads/>'_**

- Version 2.7.5 is okay
- On DLX, go into interactive setup with the command `srun -u bash -i`
- `bash ./canopy-1.0.3-rh5-64.sh`
- Follow the prompts (use spacebar to scroll through the license file)

Add lines to your profile to make this python installation your default python:

```
vi ~/.bashrc
#EPD (Canopy) python
export PATH=//home/tma249/Canopy/appdata/canopy-1.0.3.1262.rh5-x86_64/bin:$PATH
```

- Do not just use the .Canopy/bin. directory - python modules will not load properly
- Log out and log in

Check your version of python: `python --version`

The version given must be the correct version. If not, for all subsequent commands that say *python*, give the full path to your version of python, e.g. `//share/apps/EPD_64bit/epd_free-7.3-2-rh5-x86_64/bin/python`

2.3.3 2. Verify setuptools (easy_install) and pip

Check if easy_install and pip are available:

- `which pip`
- `which easy_install`

Example:

```
[username@aci-service-2 ~]$ which pip
//home/username/Canopy/appdata/canopy-1.0.3.1262.rh5-x86_64/bin/pip
[username@aci-service-2 ~]$ which easy_install
//home/username/Canopy/appdata/canopy-1.0.3.1262.rh5-x86_64/bin/easy_install
```

pip must be version 1.3 or later (pip –version)

If either easy_install or pip is missing, install them as follows.

Get setuptools (easy_install)

- **:ref:'setuptools <https://pypi.python.org/pypi/setuptools>'_**
- `wget https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py`
- `python ez_setup.py`

Get pip

- **:ref:'pip <https://pypi.python.org/pypi/pip>'_**
- `curl -O https://raw.github.com/pypa/pip/master/contrib/get-pip.py`
- `python get-pip.py`

easy_install and pip should now be located wherever your installed python is. Check their locations and the pip version again.

2.3.4 3. Verify numpy and scipy

Check if numpy and scipy available:

```
python
import numpy
import scipy
```

If numpy is not available, try pip installation:

```
pip install --user numpy
```

(Use the pip in the bin directory of the correct version of python)

If pip does not work, follow Quick install of numpy here. This will install Numpy without external library support. It is a quick and easy way to install Numpy, and will suite you for the purposes of running MAST.

- Grab the most recent stable release of numpy from **:ref:'<http://www.scipy.org/install.html>'_**
- Untar with command `tar -zxvf numpy-<version>.tar.gz`
- `cd numpy-<version>`
- Put the following in your command line:

```
BLAS=None LAPACK=None ATLAS=None python setup.py config build install --prefix=<location where y
```

- Get something to drink; this'll take about 5-10 minutes.
- Add to your .bashrc:

```
NUMPY=<location you specified above>
export PYTHONPATH=$NUMPY:$PYTHONPATH
```

- source $HOME/.bashrc

2.3.5 4. Install pymatgen and custodian

Make sure you explicitly use the correct pip and easy_install, e.g. //home/username/.local/bin/pip and //home/username/.local/bin/easy_install or other such paths, corresponding to the correct version of python

Use the `--user` tag if you are not using the easy_install and pip from your own installation of python. Otherwise, you can omit this tag.

Upgrade the *distribute* package. You **MUST** upgrade this package, even if it is freshly installed. (8/9/13)

```
nice -n 19 easy_install --user --upgrade distribute
```

pip install pymatgen and custodian:

```
nice -n 19 pip install --user pymatgen
nice -n 19 pip install --user custodian
```

If pip does not work, try making your own temp directory.

```
mkdir //home/<username>/tmp
export TMPDIR=.//home/<username>/tmp.
```

Then try running the pip commands again.

Remove any pip directory if it exists.

```
cd //tmp
rm -r pip-build
```

2.3.6 5. Set up the pymatgen VASP_PSP_DIR

Locate the VASP pseudopotentials

- On bardeen, this is `//share/apps/vasp_pseudopotentials`
- On DLX it is `//home/adozier/VASP`
  - On DLX, SKIP TO THE NEXT NUMBERED STEP

Run pymatgen's python setup tool. This tool should be located wherever pymatgen was installed, either `~/.local/bin/potcar_setup.py` if you installed it with `--user`, or wherever python is, otherwise.

```
python .local/bin/potcar_setup.py or python potcar_setup.py or simply potcar_setup.py
```

(Remember to use the correct version of python, determined in step 2, e.g. //share/apps/EPD_64bit/epd_free-7.3-2-rh5-x86_64/bin/python .local/bin/potcar_setup.py)

Take the paw directory if you are using PAW. Do not take the top directory, or the GGA/LDA/etc folders will overwrite.

Example of running the python setup tool:

```
Please enter full path where the POT_GGA_PAW_PBE, etc. subdirs are present. If you obtained the PSPs
Please enter the fullpath of the where you want to create your pymatgen resources directory:
//home/<username>/.local/vasp_pps
```

Rename the folders under `//home/<username>/.local/vasp_pps`:

- Rename the PBE folder POT_GGA_PAW_PBE to correspond to mast_xc pbe
- Rename the GGA folder POT_GGA_PAW_PW91 to correspond to mast_xc pw91

Add a line to your .bashrc file exporting the environment variable VASP_PSP_DIR to this VASP directory.

- On bardeen, it should look something like:

  ```
  export VASP_PSP_DIR=//home/<username>/.local/vasp_pps
  ```

- On DLX, use the directories already created:

  ```
  export VASP_PSP_DIR=//home/adozier/VASP/resources
  export VASP_PSP_DIR=<whichever path you used in the potcar_setup.py script>
  ```

- Remember to save your .bashrc file. Test the change:

  ```
  source ~/.bashrc
  cd $VASP_PSP_DIR
  ```

- Make sure you are getting to the right directory, which has POT_GGA_POW_PBE etc. folders inside it.

2.3.7 6. Get MAST

Get the MAST.tar.gz file from MaterialsHub.org and untar it:

```
nice -n 19 tar -xzvf MAST.tar.gz
```

(or run this command over interactive submission, which is better)

Make the bin executables runnable. Supposing the uncompressed path was //home/username/MAST, then run the following command:

```
chmod -R a+x //home/username/MAST/bin
```

Modify the submission details for your platform:

```
cd //home/username/MAST/submit
cp platforms/script_commands_<yourplatform>.py script_commands.py
cp platforms/queue_commands_<yourplatform>.py queue_commands.py
cp platforms/submit_<yourplatform>.sh submit.sh
```

Modify submit.sh as necessary for your platform.

- The submit.sh script should be set up to run mastmon.py on the shortest wallclock, fastest-turnaround queue on your system (e.g. a serial queue, morganshort, etc.)
- Examples of special modifications for submit.sh:
  - ACI/HPC, add line: `#SBATCH --partition=univ`
  - Bardeen, add a line to tell control where to run the monitor: `#PBS -q morganshort`

Modify script_commands.py as necessary for your platform.

- ACI/HPC: in script_commands.py, near line 95, add line: `myscript.data.append("#SBATCH --partition=univ " + "\n")`
- Bardeen: in script_commands.py near line 95 add line: `myscript.data.append("#PBS -q " + mast_queue + "\n")`

Modify queue_commands.py as necessary for your platform. (On DLX, ACI, and bardeen, no modification should be necessary.)

Figure out the correct mast_exec calls for your system, to be used in the *Input File*. Examples are below.

- Bardeen: `mast_exec //opt/mpiexec/bin/mpiexec //share/apps/bin/vasp5.2_par_opt1` (or any of the other vasp executables)
- DLX: `mast_exec //home/username/bin/vaspmpirun`, where vaspmpirun is the following script:

```
#!/bin/bash
export PERL5LIB=/opt/moab/lib/perl5
export MIC_LD_LIBRARY_PATH=/share/cluster/RHEL6.2/x86_64/apps/intel/ict/composer_xe_2013.0.079/c
export LD_LIBRARY_PATH=/share/cluster/RHEL6.2/x86_64/apps/openmpi/1.6.2/lib:/share/cluster/RHEL6
export INTEL_MKL_LIBS=/share/cluster/RHEL6.2/x86_64/apps/intel/ict/composer_xe_2013.0.079/mkl/li
export QTLIB=/usr/lib64/qt-3.3/lib
PATH=$PATH:$HOME/bin:$HOME/bin/convaspTest
export PATH
VaspPath=//home/adozier/VASP/vasp.5.2
export OMP_NUM_THREADS=1
ulimit -s unlimited
ulimit -l unlimited
#mpirun $VaspPath/vasp
//share/cluster/RHEL6.2/x86_64/apps/openmpi/1.6.2/bin/mpirun $VaspPath/vasp
```

Modify ~/.bashrc if necessary

- ACI/HPC, add line: `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH://opt/intel/lib/intel64`

To ensure recipes are created correctly, add python whitespace tab stops to your ~/.vimrc file:

```
" VIM settings for python in a group below:
set tabstop=4
set shiftwidth=4
set smarttab
set expandtab
set softtabstop=4
set autoindent
```

Follow the environment variable setup in a similar fashion to *Installation on bardeen* Follow the testing instructions from *Test that MAST can run on bardeen*

2.4 Platform Support

Queue and submission script commands are in `//home/user/topmast/submit` and may need to be heavily modified depending on the platform used. To customize the queue submission behavior, copy a queue_commands.py, script_commands.py and submit.sh from `$MAST_INSTALL_PATH/submit/platforms` to `$MAST_INSTALL_PATH/submit/`. Remove the platform name from the file names.

The out-of-the-box PBS submission script is built using the following input file keywords (see *Ingredients*):

- mast_processors or a combination of mast_ppn and mast_nodes
- mast_queue

- mast_exec
- mast_walltime
- mast_memory
- the ingredient name

CHAPTER

THREE

INGREDIENTS

Each ingredient is a separate calculation. Ingredients make up recipes.

Each ingredient is responsible for updating its child ingredients through an update_children method.

Each ingredient is given:

- A name, which is the full path to the ingredient.s directory and is automatically generated using the system name and the recipe template. (Do not use parentheses in ingredient names.) Some ingredient names must be structured specifically. For examples of naming conventions, see the *Recipe*. In particular:
- An ingredient which is supposed to correspond to values given by the `$defects` section of the *Input File* should always be named with `inducedefect_` (for the structural creation of the defect) or `defect_` (for an actual defect calcualtion)
- **An ingredient which is supposed to correspond to values in the `$neb` section, such as a nudged elastic band (NEB) calcula**
  - A phonon calculation should always be named with `phonon_`, and a subsequent calculation of phonon results should be named with `phonon_...parse`
- The letters .q=. are reserved (generated automatically by the recipe template in some cases) and should not otherwise be put in an ingredient name
- A dictionary of program-specific keywords, which come from each ingredient.s section in the `$ingredients` section of the *Input File*.
- A pymatgen structure object representing the very first structure created from the `$structure` section in the input file.
- A type, which is specified in the recipe, next to the ingredient name, in parentheses. The ingredient type corresponds to the ingredient type subsection in the `$ingredients` section of the input file. The information given in these subsections includes:
  - Program-specific keywords
- Other MAST keywords, including:
  - **The .write. method: which files the ingredient should write out before running (e.g., create the INCAR)**
    - The .ready. method: how MAST can tell if the ingredient is ready to run (often, in addition to writing its own files, an ingredient must also wait for data from its parent ingredient(s)).
    - The .run. method: what MAST should do to run the ingredient (e.g. submit a submission script to a queue, or perform some other action)
    - The .complete. method: how MAST can tell if the ingredient is considered complete

    * The .update children. method: what information an ingredient passes on to its children, and how this information is passed on

The same ingredient in a recipe may be listed more than once, with several different ingredient types. In this case, the first four methods and all the ingredient keywords are given by the first ingredient type encountered. Only the .update_children. method is changed for all subsequent positions. This situation indicates that the ingredient has many children, which must be updated in different ways and thus needs different update_children methods for those different situations.

More detail on ingredients is given in the `$ingredients` section of the *Input File*.

CHAPTER

FOUR

INPUT FILE

4.1 Introduction to the Input File

The MAST program is driven by two main files: an input file which contains all the various keywords required for setting up the recipe (i.e. workflow), and a *Recipe Template* which organizes all the ingredients (i.e. calculations) in the recipe. In this section, we will discuss the input file

The input file contains several sections and subsections. Bounds of sections are denoted by `$sectionname` and `$end`. Bounds of subsections within a section are denoted by `begin subsectionname` and `end`.

Comments in the input file are allowed only as separate lines starting with .#.. A comment may not be appended to a line.

Example of the `$structure` section, with three subsections, .elementmap., .coordinates., and .lattice.:

```
$structure
coord_type fractional

begin elementmap
X1 Ga
X2 As
end

begin coordinates
X1 0.000000 0.000000 0.000000
X1 0.500000 0.500000 0.000000
X2 0.250000 0.250000 0.250000
X2 0.750000 0.750000 0.250000
end

begin lattice
6.0 0.0 0.0
0.0 6.0 0.0
0.0 0.0 6.0
end
$end
```

Each section is described in detail below.

4.2 The MAST section

The `$mast` section contains this keyword:

- system_name: Specify a single descriptive word here, like EpitaxialStrain. This keyword will become part of the recipe directory.s name and allow you to spot the recipe in the `$MAST_SCRATCH` directory:

  ```
  system_name EpitaxialStrain
  ```

4.3 The Structure section

The `$structure` section contains the coordinate type, coordinates, and lattice, or, optionally, the name of a structure file (either CIF or VASP POSCAR-type).

4.3.1 Structure by file

Using the keyword `posfile`, a VASP POSCAR-type file or a CIF file can be inserted here in this section:

```
$structure
posfile POSCAR_fcc
$end
```

The file should be located in the same directory as the input file.

A CIF file should end with *.cif.

A POSCAR-type filename must start with `POSCAR_` or `CONTCAR_` in order for pymatgen to recognize it. The elements will be obtained from the POSCAR unless you also have a POTCAR in the directory, in which case, check your output carefully because the elements might be given by the POTCAR instead, no matter what elements are written in the POSCAR file.

4.3.2 Structure by specification

To specify a structure, use the following subsections:

‘‘coord_type‘‘: This keyword specifies fractional or cartesian coordinates. Only fractional coordinates have been thoroughly tested with most MAST features.

‘‘lattice‘‘: The lattice subsection specifies lattice basis vectors on a cartesian coordinate system.

‘‘elementmap‘‘: The elementmap subsection allows you to create a generic lattice and interchange other elements onto it. This is useful when looping over other elements (discussed later).

The elementmap subsection works in conjunction with the coordinates subsection.

‘‘coordinates‘‘: The coordinates subsection specifies the coordinates in order.

Fractional coordinates are fractional along each lattice basis vector, e.g. .0.5 0 0. describes a position 0.5 (halfway) along the first lattice basis vector.

Each fractional coordinate must be preceded by either an element symbol or an X# symbol corresponding to the symbols assigned in the elementmap section.

Example:

begin $structure

coord_type fractional

begin lattice 6.0 0.0 0.0 0.0 6.0 0.0 0.0 0.0 6.0 end

begin elementmap X1 Ga X2 As end

begin coordinates X1 0.000000 0.000000 0.000000 X1 0.500000 0.500000 0.000000 X1 0.000000 0.500000 0.500000 X1 0.500000 0.000000 0.500000 X2 0.250000 0.250000 0.250000 X2 0.750000 0.750000 0.250000 X2 0.250000 0.750000 0.750000 X2 0.750000 0.250000 0.750000 end

$end

4.4 The Ingredients section

The `$ingredients` section contains a section for global ingredient keywords and then a section for each ingredient type.

Program-specific keywords such as VASP INCAR keywords are included in these sections. All other keywords are prefaced with `mast_`.

Each ingredient type in the recipe should have a subsection denoted by

begin ingredient_type (keywords here)

end

even if there are no keywords within that section, in which case the `end` line directly follows the `begin` line.

4.4.1 Ingredients that are VASP calculations

VASP keywords such as `IBRION`, `ISIF`, `LCHARG`, `LWAVE`, and so on, can be specified under each ingredient type in the `$ingredients` section of the input file.

Such program-specific keywords are only allowed if they are listed in the program-specific file located in the `$MAST_INSTALL_PATH/MAST/ingredients/programkeys/` folder, for example, `$MAST_INSTALL_PATH/MAST/ingredients/programkeys/vasp_allowed_keywords.py`.

These program-specific keywords will be turned into uppercase keywords. The values will not change case, and should be given in the case required by the program. For example, `lwave False` will be translated into `LWAVE False` in the VASP INCAR file.

One exception for VASP keywords is the `IMAGES` keyword, which signals a nudged elastic band run, and should instead be set in the `$neb` section of the input file.

For VASP ingredients, please include

lcharg False lwave False

in your ingredient global keywords in order to avoid writing the large VASP files CHGCAR and WAVECAR, unless you really need these files.

Any keyword that starts with `mast_` is considered a special keyword utilized by MAST and will not be written into the VASP INCAR file.

4.4.2 Special MAST ingredient keywords:

Some of these special MAST keywords are only appropriate for VASP calculations.

mast_program: Specify which program to run (.vasp., .vasp_neb., or .phon. are currently supported)

```
mast_program vasp
```

- This keyword must be in lowercase (.vasp., .phon.)

mast_kpoints: Specify k-point instructions in the form of kpoints along lattice vectors a, b, and c, and then a designation M for Monkhorst-Pack or G for Gamma-centered.

mast_kpoints = 3x3x3 G

- Either this keyword or `mast_kpoint_density` is required for VASP calculations.

mast_kpoint_density: A number for the desired kpoint mesh density. Only works with `mast_write_method` of `write_singlerun_automesh`

- Either this keyword or `mast_kpoints` is required for VASP calculations.

mast_pp_setup: Specify which pseudopotential goes to which element:

mast_pp_setup La=La Mn=Mn_pv O=O_s

mast_xc: Specify an exchange correlation functional; for VASP, follow the conventions of pymatgen (e.g. pw91, pbe)

- This keyword is required for VASP calculations.

mast_multiplyencut: Specify a number with which to multiply the maximum ENCUT value of the pseudopotentials. Volume relaxations in VASP often take 1.5; otherwise 1.25 is sufficient.

- Default is 1.5
- If `encut` is given as a program keyword, then that value will be used and `mast_multiplyencut` should have no effect

mast_setmagmom: Specify a string to use for setting the initial magnetic moment. A short string will result in multipliers. For example, `mast_setmagmom 1 5 1` will produce `2*1 2*5 8*1` for a 12-atom unit cell with 2A, 2B, and 8C atoms. A string of the number of atoms in the `POSCAR` file will be printed as entered, for example, `mast_setmagmom 1 -1 1 -1 1 -1 1 -1`.

mast_charge: Specify the charge on the system (total system)

- -1 charge means the ADDITION of one electron. For example, O2- has two more electrons than O neutral.
- A positive charge is the REMOVAL of electrons. For example, Na+ with a +1 charge has one FEWER electron than Na neutral.

mast_coordinates: For a non-NEB calculation, allows you to specify a single POSCAR-type of CIF structure file which corresponds to the relaxed fractional coordinates at which you would like to start this ingredient. ONLY the coordinates are used. The lattice parameters and elements are given by the $structure section of the input file. The coordinates must be fractional coordinates.

```
mast_coordinates coordposcar
```

- For an NEB calculation, use a comma-delimited list of poscar files corresponding to the correct number of images. Put no spaces between the file names. Example for an NEB with 3 intermediate images:

mast_coordinates im1poscar,im2poscar,im3poscar

- The structure files must be found in the directory from which the input file is being submitted when initially inputting the input file (e.g. the directory you are in when you run `mast -i test.inp`); once the `input.inp` file is created in the recipe directory, it will store a full path back to these poscar-type files.
- This keyword cannot be used with programs other than VASP, cartesian coordinates, and special ingredients like inducedefect-type ingredients, whose write or run methods are different.

mast_strain: Specify three numbers for multiplying the lattice parameters a, b, and c. Only works with `mast_run_method` of `run_strain`

mast_strain 1.01 1.03 0.98

This example will stretch the lattice along lattice vector a by 1%, stretch the lattice along lattice vector b by 3%, and compress the lattice along lattice vector c by 2%

mast_scale: A number for which to scale all dimensions of a supercell. Only works with `mast_run_method` of `run_scale` or `run_scale_defect`

mast_frozen_seconds: A number of seconds before a job is considered frozen, if its output file has not been updated within this amount of time. If not set, 21000 seconds is used.

mast_auto_correct: Specify whether mast should automatically correct errors.

- The default is True, so if this keyword is set to True, or if this keyword is not specified at all, then MAST will attempt to find errors, automatically correct the errors, and resubmit the ingredient.
- If set to False, MAST will attempt to find errors, then write them into a `MAST_ERROR` file in the recipe folder, logging both the error-containing ingredient and the nature of the error, but not taking any corrective actions. The recipe will be skipped in all subsequent MAST runs until the `MAST_ERROR` file is manually deleted by the user.

The following queue-submission keywords are platform dependent and are used along to create the submission script:

mast_exec: The command used in the submission script to execute the program. Note that this is a specific command rather than the .class. of program, given in `mast_program`, and it should include any MPI commands.

```
mast_exec //opt/mpiexec/bin/mpiexec ~/bin/vasp_5.2
```

mast_nodes: The number of nodes requested.

mast_ppn: The number of processors per node requested.

mast_queue: The queue requested.

mast_walltime: The walltime requested, in whole number of hours

mast_memory: The memory per processor requested.

The following keywords have individual sections:

mast_write_method: The .write. method, which specifies files the ingredient should write out before running (e.g., create the INCAR)

mast_ready_method: The .ready. method, which specifies how MAST can tell if the ingredient is ready to run (often, in addition to writing its own files, an ingredient must also wait for data from its parent ingredient(s)).

mast_run_method: The .run. method, which specifies what MAST should do to run the ingredient (e.g. submit a submission script to a queue, or perform some other action)

mast_complete_method: The .complete. method, which specifies how MAST can tell if the ingredient is considered complete

mast_update_children_method: the .update children. method, which specifies what information an ingredient passes on to its children, and how it does so.

mast_write_method keyword values

write_singlerun

- Write files for a single generic run.
- Programs supported: vasp, phon (phon assumes vasp-type output given by one of the .give_phonon. update children methods)

write_singlerun_automesh

- Write files for a single generic run.
- Programs supported: vasp

- Requires the `mast_kpoint_density` ingredient keyword

write_neb

- Write an NEB ingredient. This method writes interpolated images to the appropriate folders, creating 00/01/.../0N directories.
- Programs supported: vasp

write_neb_subfolders

- Write static runs for an NEB, starting from a previous NEB, into image subfolders 01 to 0(N-1).
- Programs supported: vasp

write_phonon_single

- Write files for a phonon run.
- Programs supported: vasp

write_phonon_multiple

- Write a phonon run, where the frequency calculation for each atom and each direction is a separate run, using selective dynamics. CHGCAR and WAVECAR must have been given to the ingredient previously; these files will be softlinked into each subfolder.
- Programs supported: vasp

mast_ready_method keyword values

ready_singlerun

- Checks that a single run is ready to run
- Programs supported: vasp (either NEB or regular VASP run), phon

ready_defect

- Checks that the ingredient has a structure file
- Programs supported: vasp

ready_neb_subfolders

- Checks that each 01/.../0(N-1) subfolder is ready to run as its own separate calculation, following the ready_singlerun criteria for each folder
- This method is used for NEB static calculations rather than NEB calculations themselves.

ready_subfolders * Checks that each subfolder is ready to run, following the ready_singlerun criteria. * Generic * This method is used for calculations whose write method includes subfolders, and where each subfolder is a calculation, as in `write_phonon_multiple`.

mast_run_method keyword values

run_defect

- Create a defect in the structure; not submitted to queue
- Generic
- Requires the `$defects` section in the input file.

run_singlerun

- Submit a run to the queue.
- Generic

run_neb_subfolders

- Run each 01/.../0(N-1) subfolder as run_singlerun
- Generic

run_subfolders

- Run each subfolder as run_singlerun
- Generic

run_strain

- Strain the structure; not submitted to queue
- Generic
- Requires the `mast_strain` ingredient keyword

run_scale

- Scale the structure (e.g. a 2-atom unit cell scaled by 2 becomes a 16-atom supercell)
- Generic
- Requires the `mast_scale` ingredient keyword, and must not be run on the starting ingredient (for VASP, the ingredient must already have been given a smaller POSCAR file, like the POSCAR for a 2-atom unit cell)

run_scale_defect

- Scale the structure and defect it (e.g. a single defect at 0.5 0.5 0.5 in the original structure becomes a single defect at 0.25 0.25 0.25 in the structure scaled by 2)
- Generic
- Requires the `mast_scale` ingredient keyword, and must not be run on the starting ingredient

mast_complete_method keyword values

complete_singlerun

- Check if run is complete
- Programs supported: vasp, phon (only entropy calculation)
- Note that for VASP, the line .reached required accuracy. is checked for, as well as a .User time. in seconds. The exceptions are:
  - NSW of 0, NSW of -1, or NSW not specified in the ingredients section keywords is taken as a static calculation, and .EDIFF is reached. is checked instead of .reached required accuracy.
  - IBRION of -1 is taken as a static calculation, and .EDIFF is reached. is checked instead of .reached required accuracy.
  - IBRION of 0 is taken as an MD calculation, and only user time is checked
- IBRION of 5, 6, 7, or 8 is taken as a phonon calculation, and only user time is checked

complete_neb_subfolders

- Check if all NEB subfolders 01/.../0(N-1) are complete, according to complete_singlerun criteria.

complete_subfolders

- Check if all subfolders are complete, according to complete_singlerun criteria.
- Generic

complete_structure

- Check if run has an output structure file written
- Programs supported: vasp (looks for CONTCAR)

mast_update_children_method keyword values

give_structure

- Forward the relaxed structure
- Programs supported: vasp (CONTCAR to POSCAR)

give_structure_and_energy_to_neb

- Forward the relaxed structure and energy files
- Programs supported: vasp (CONTCAR to POSCAR, and copy over OSZICAR)

give_neb_structures_to_neb

- Give NEB output images structures as the starting point image input structures in another NEB
- Programs supported: vasp (01/.../0(N-1) CONTCAR files will be the child NEB ingredient.s starting 01/.../0(N-1) POSCAR files.

give_phonon_single_forces_and_displacements(self, childname)

- Forward force and displacement information
- Programs supported: vasp, for vasp-to-phon transition (DYNMAT, XDATCAR)

give_phonon_multiple_forces_and_displacements

- Combine individual phonon forces and displacements and forward this information
- Programs supported: vasp, for vasp-to-phon transition (DYNMAT, XDATCAR)

give_saddle_structure

- Forward the highest-energy structure of all subfolder structures
- Programs supported: vasp

give_structure_and_restart_files (same as give_structure_and_restart_files_softlinks)

- Forward the relaxed structure and additional files
- Programs supported: vasp (CONTCAR to POSCAR, and softlinks to parent.s WAVECAR and CHGCAR files)

give_structure_and_restart_files_full_copies

- Forward the relaxed structure and additional files
- Programs supported: vasp (CONTCAR to POSCAR, and full copies of parent.s WAVECAR and CHGCAR files)

give_structure_and_charge_density_full_copy

- Forward the relaxed structure and charge density file; copies the file
- Programs supported: vasp (CONTCAR to POSCAR, and copy over CHGCAR)

give_structure_and_charge_density_softlink

- Forward the relaxed structure and charge density file as a softlink
- Programs supported: vasp (CONTCAR to POSCAR, and softlink to CHGCAR)

give_structure_and_wavefunction_full_copy * Forward the relaxed structure and wavefunction file; copies the file * Programs supported: vasp (CONTCAR to POSCAR, and copy over WAVECAR)

give_structure_and_wavefunction_softlink

- Forward the relaxed structure and wavefunction file as a softlink
- Programs supported: vasp (CONTCAR to POSCAR, and softlink to WAVECAR)

Example Ingredients section

Here is an example ingredients section:

```
$ingredients
begin ingredients_global
mast_program    vasp
mast_nodes      1
mast_multiplyencut 1.5
mast_ppn        1
mast_queue      default
mast_exec       mpiexec //home/mayeshiba/bin/vasp.5.3.3_vtst_static
mast_kpoints    2x2x2 M
mast_xc PW91
isif 2
ibrion 2
nsw 191
ismear 1
sigma 0.2
lwave False
lcharg False
prec Accurate
mast_program   vasp
mast_write_method           write_singlerun
mast_ready_method           ready_singlerun
mast_run_method             run_singlerun
mast_complete_method        complete_singlerun
mast_update_children_method  give_structure
end

begin volrelax_to_singlerun
isif 3
end

begin singlerun_to_phonon
ibrion -1
nsw 0
mast_update_children_method  give_structure_and_restart_files
mast_multiplyencut 1.25
lwave True
lcharge True
end

begin inducedefect
mast_write_method           no_setup
mast_ready_method           ready_defect
```

```
mast_run_method             run_defect
mast_complete_method        complete_structure
end

begin singlerun_vac1
mast_coordinates            vac1poscar
end

begin singlerun_vac2
mast_coordinates            vac2poscar
end

begin singlerun_to_neb
ibrion -1
nsw 0
mast_update_children_method  give_structure_and_energy_to_neb
lwave True
lcharge True
end

begin neb_to_neb_vac1-vac2
mast_coordinates            nebim1poscar,nebim2poscar,nebim3poscar
mast_write_method           write_neb
mast_update_children_method  give_neb_structures_to_neb
mast_nodes                  3
mast_kpoints                1x1x1 G
ibrion 1
potim 0.5
images 3
lclimb True
spring -5
end

begin neb_to_neb_vac1-vac3
mast_coordinates            nebim1poscar,nebim2poscar,nebim3poscar
mast_write_method           write_neb
mast_update_children_method  give_neb_structures_to_neb
mast_nodes                  3
mast_kpoints                1x1x1 G
ibrion 1
potim 0.5
images 3
lclimb True
spring -5
end

begin neb_to_nebstat
mast_write_method           write_neb
mast_update_children_method  give_neb_structures_to_neb
mast_nodes                  3
ibrion 1
potim 0.5
images 3
lclimb True
spring -5
end

begin nebstat_to_nebphonon
```

```
ibrion -1
nsw 0
mast_write_method           write_neb_subfolders
mast_ready_method           ready_neb_subfolders
mast_run_method             run_neb_subfolders
mast_complete_method        complete_neb_subfolders
mast_update_children_method  give_saddle_structure
end

begin phonon_to_phononparse
mast_write_method           write_phonon_multiple
mast_ready_method           ready_subfolders
mast_run_method             run_subfolders
mast_complete_method        complete_subfolders
mast_update_children_method  give_phonon_multiple_forces_and_displacements
ibrion 5
nfree 2
potim 0.01
istart 1
icharg 1
end

begin phononparse
mast_program                phon
lfree .True.
temperature 273
ptemp 10 110
nd 3
qa 11
qb 11
qc 11
lnosym .True.
ldrift .False.
lsuper .False.
mast_exec $MAST_INSTALL_PATH/bin/phon_henry
mast_multiplyencut 1.25
end

$end
```

4.5 The Recipe section

The `$recipe` section contains the recipe template to be used.

```
$recipe
```

recipe_file myrecipefile.txt $end

4.6 The Defects section (optional)

The `$defects` section includes the defect type of vacancy, interstitial, substitution, or antisite (which is the same as substitution), the defect coordinates, and the defect element symbol.

- Note that if an `elementmap` subsection is given in the `$structure` section of the input file, the mapped designations `X1`, `X2`, and so on can be given instead of an element symbol.

The `coord_type` keyword specifies fractional or cartesian coordinates for the defects.

The `threshold` keyword specifies the absolute threshold for finding the defect coordinate, since relaxation of the perfect structure may result in changed coordinates.

Example `$defects` section:

```
$defects
```

coord_type fractional threshold 1e-4

vacancy 0 0 0 Mg vacancy 0.5 0.5 0.5 Mg interstitial 0.25 0.25 0 Mg interstitial 0.25 0.75 0 Mg

$end

The above section specifies 4 point defects (2 vacancies and 2 interstitials) to be applied separately and independently to the structure. When combined with the correct *recipe*, four separate ingredients, each containing one of the defects above, will be created.

Multiple point defects can be also grouped together as a combined defect within a .begin/end,. with a label after the .begin,. such as:

```
$defects
```

coord_type fractional threshold 1e-4

begin doublevac vacancy 0.0 0.0 0.0 Mg vacancy 0.5 0.5 0.5 Mg end

interstitial 0.25 0.25 0 Mg interstitial 0.25 0.75 0 Mg

$end

In this case, there will be three separate .defect. ingredients: one ingredient with two vacancies together (where the defect group is labeled .doublevac.), one interstitial, and another interstitial.

Charges can be specified as `charge=0,10`, where a comma denotes the lower and upper ranges for the charges.

Let.s say we want a Mg vacancy with charges from 0 to 3 (0, 1, 2, and 3):

```
vacancy 0 0 0 Mg charge=0,3
```

Let.s say we want a dual Mg vacancy with a charge from 0 to 3 and labeled as Vac@Mg-Vac@Mg:

```
begin Vac@Mg-Vac@Mg
vacancy 0.0 0.0 0.0 Mg
vacancy 0.5 0.5 0.5 Mg
charge=0,3
end
```

For a single defect, charges and labels can be given at the same time:

Let.s say we have a Mg vacancy with charges between 0 and 3, and we wish to label it as Vac@Mg:

vacancy 0.0 0.0 0.0 Mg charge=0,3 label=Vac@Mg

The charge and label keywords are interchangeable, i.e. we could also have typed:

```
vacancy 0 0 0 Mg label=Vac@Mg charge=0,3
```

If you use charges in the defects section like this, then you should use a *recipe* template with a free-form defect_<N>_<Q> format.

4.6.1 Phonons for defects

Phonon calculations are described by a .phonon center site. coordinate and a .phonon center radius. in Angstroms. Atoms within the sphere specified by these two values will be included in phonon calculations.

For VASP, this inclusion takes the form of selective dynamics T T T for the atoms within the sphere, and F F F otherwise, in a phonon calculation (IBRION = 5, 6, 7, 8)

If the phonon center radius is 0, only the atom found at the phonon center site point will be considered.

To use phonons in the defects section, use the subsection keyword .phonon. followed by a label for the phonon, the fractional coordinates for the phonon center site, and a float value for the phonon center radius. Multiple separate phonon calculations may be obtained for each defect, for example:

```
begin int1
interstitial 0.25 0.25 0.25 X2
phonon host3 0.3 0.3 0.4 2.5
phonon solute 0.1 0.1 0.2 0.5
end
```

In the example above, .host3. is the label for the phonon calculation where (0.3, 0.3, 0.4) is the coordinate for the phonon center site, and 2.5 Angstroms is the radius for the sphere inside which to consider atoms for the phonon calculation. In the example above, .solute. is the label for the phonon calculation bounded within a 0.5 Angstrom radius centered at (0.1, 0.1, 0.2) in fractional coordinates.

The recipe template file for phonons may include either the explicit phonon labels and their charge and defect label, or <N>_<Q>_<P> (defect label _ charge label _ phonon label).

Because phonons are cycled with the defects, a new parent loop must be provided for the phonons, for example:

{begin} defect_<N>_<Q>_stat (static)

phonon_<N>_<Q>_<P> (phonon) phonon_<N>_<Q>_<P>_parse (phononparse)

{end}

4.7 The chemical potentials section

The $chemical_potentials section lists chemical potentials, used for defect formation energy calculations using the defect formation energy tool. Currently, chemical potentials must be set ahead of time. Each chemical potential set may be labeled.

```
$chemical_potentials

begin Ga rich
Ga -3.6080
As -6.0383
Bi -4.5650
end

begin As rich
Ga -4.2543
As -5.3920
Bi -4.5650

end
```

$end

4.8 The NEB section

The `$neb` section includes a list of nudged-elastic-band hops. Each neb hop should be a subsection labeled with the starting and ending .defect group. as specified in the `$defects` section, and then also indicate the movement of elements, and their closest starting and ending positions. These explicit positions disambiguate between possible interpolations.

- Note that if an `elementmap` subsection is given in the `$structure` section of the input file, the mapped designations `X1`, `X2`, and so on can be given instead of an element symbol.

Again, the `$neb` section is tied to specific defect labels. The NEB ingredients must be able to find defects or defect groups with those labels.

The `images` keyword specifies the number of intermediate images, which must currently be the same in all NEBs in the recipe.

Phonons may be specified within each NEB grouping, as in the defects section. The presumed saddle point in an NEB is usually taken; use the `mast_update_children give_saddle_structure` to give that saddle point structure to the phonon calculation. If, in an NEB, the frequencies for the moving atom are desired for the phonon calculations, and if that atom is anticipated to pass from fractional coordinate 0 0 0 to fractional coordinate 0.5 0 0, then the phonon_center_site should be 0.25 0 0 (assuming a straight path), and the phonon_center_radius is probably about 1 Angstrom.

Example defect and NEB section together:

```
$defects

    coord_type fractional threshold 1e-4

    vacancy 0.0 0.0 0.0 Mg label=vac1 vacancy 0.0 0.5 0.5 Mg label=vac2 interstitial 0.25 0.0 0.0 Al label=int1 interstitial 0.0 0.25 0.0 Al label=int2

    $end

    $neb

    begin vac1-vac2 images 1 Mg, 0 0 0, 0 .5 0.5
end

    begin int1-int2 Al, 0.25 0 0, 0 0.25 0 images 3 phonon movingatom 0.125 0.125 0.0 1.0
end

$end
```

4.9 Creating several input files at once: the looped input file

One input file may be able to spawn several nearly-identical input files, which differ in small ways.

4.9.1 Independent loops

The special looping keyword `indeploop` may be used to signify a line which indicates that spawned input files should cycle through these values.

```
indeploop mast_xc (pw91, pbe)
```

In this example, two input files will be created. One input file will contain the line `mast_xc pw91`. The other input file will contain the line `mast_xc pbe`.

- Any text within parentheses and separated by a comma will be looped. * Lines which normally include commas, like the .charge. line in the `$defects` section, may not be looped.
- This keyword may only be used once on a line.

If there is more than one `indeploop` keyword in the input file, a combinatorial spawn of input files will be created.

For example, this excerpt would generate four input files: one with iron using pw91, one with iron using pbe, one with copper using pw91, and one with copper using pbe:

```
$structure
```

begin elementmap indeploop X1 (Fe, Cu) end ... $end

$ingredients begin ingredients_global indeploop mast_xc (pw91, pbe) ... end

$end

4.9.2 Dependent, or pegged, loops

Sometimes looped lines should really be looped together at the same time, rather than with each value looped over each other value.

For example, if you want to create a single input file, but signify that it should be copied into three input files, one for each element, but with different GGA+U U-values, you would use a pegged loop like this:

```
$structure
begin elementmap
pegloop1 X1 (Es, Fm, Md)
end
```

... $end

$ingredients begin ingredients_global pegloop1 ldauu (5.3, 6.5, 8.0) ... end

$end

In this case, three input files will be created. In the first input file, Es will be paired with a U-value of 5.3. In the second input file, Fm will be paired with a U-value of 6.5. In the third input file, Md will be paired with a U-value of 8.0.

There are two pegged loops allowed, specified by `pegloop1` and `pegloop2`.

Each pegged loop and independent loop will be combinatorially combined. For example, if a separate line `indeploop mast_xc (pw91, pbe)` were included in the `ingredients_global` subsection above, then six input files would be created: one pw91 and one pbe input file for Es with +U 5.3, another pair for Fm, and another pair for Mn.

In the example below, four input files would be created, corresponding to four different lattices, [(6.0,0.0,0.0),(0.0,6.0,0.0),(0.0,0.0,2.0)], [(6.0,0.0,0.0),(0.0,6.0,0.0),(0.0,0.0,3.0)], [(4.0,0.0,0.0),(0.0,4.0,0.0),(0.0,0.0,2.0)], and [(4.0,0.0,0.0),(0.0,4.0,0.0),(0.0,0.0,3.0)]

```
begin lattice
pegloop1 (6.0,4.0) 0.0 0.0
pegloop1 0.0 (6.0,4.0) 0.0
indeploop 0.0 0.0 (2.0,3.0)
end
```

4.10 Full example

Recipe:

```
Recipe OptimizeWorkflow
perfect_opt1 (lowmesh)
    perfect_opt2
        perfect_stat (static)
        {begin}
        inducedefect_<N> (inducedefect)
            defect_<N>_<Q>_opt1 (lowmesh_defect)
                defect_<N>_<Q>_opt2 (defect_relax)
                    defect_<N>_<Q>_stat (static)
        {end}
{begin}
defect_<N>_<Q>_stat (static)
    phonon_<N>_<Q>_<P> (phonon)
        phonon_<N>_<Q>_<P>_parse (phononparse)
{end}
{begin}
defect_<B>_<Q>_stat (static_to_neb), defect_<E>_<Q>_stat (static_to_neb)
    neb_<B-E>_<Q>_opt1 (neb_to_neb)
        neb_<B-E>_<Q>_opt2 (neb_to_nebstat)
            neb_<B-E>_<Q>_stat (nebstat_to_phonon)
    neb_<B-E>_<Q>_opt2 (neb_to_nebstat)
    neb_<B-E>_<Q>_stat (nebstat_to_phonon)
{end}
{begin}
neb_<B-E>_<Q>_stat (nebstat_to_phonon)
    phonon_<B-E>_<Q>_<P> (phonon)
        phonon_<B-E>_<Q>_<P>_parse (phononparse)
{end}
```

Input file:

```
# Small demo for NEB workflow
$mast
system_name PhononNebTest
$end

$structure
coord_type fractional

begin elementmap
X1 Al
X2 Mg
end

begin lattice
3.5 0 0
0 3.5 0
0 0 3.5
end

begin coordinates
X1 0.0000000000 0.0000000000 0.0000000000
X1 0.5000000000 0.5000000000 0.0000000000
X1 0.0000000000 0.5000000000 0.5000000000
```

```
X1 0.5000000000 0.0000000000 0.5000000000
end

$end

$defects
threshold 1e-4
coord_type fractional

begin int1
interstitial 0.25 0.25 0.25 X2
phonon host 0.0 0.5 0.5 0.5
charge=-3,-2
end

begin int2
interstitial 0.25 0.25 0.75 X2
phonon host 0.0 0.0 0.0 0.5
phonon int 0.25 0.25 0.75 0.5
charge=-2,-2
end

begin int3
interstitial 0.75 0.25 0.25 X2
phonon host 0.0 0.0 0.0 0.5
phonon int 0.75 0.25 0.25 0.5
charge=-3,-3
end

$end

$ingredients
begin ingredients_global
mast_nodes         1
mast_multiplyencut 1.5
mast_ppn           1
mast_queue         default
mast_exec          //share/apps/vasp5.2_cNEB
mast_kpoints       2x2x2 M
mast_xc            PBE
isif 3
ibrion 2
nsw 191
ismear 1
sigma 0.2
lwave False
lcharg False
prec Accurate
mast_program   vasp
mast_write_method            write_singlerun
mast_ready_method            ready_singlerun
mast_run_method              run_singlerun
mast_complete_method         complete_singlerun
mast_update_children_method  give_structure
end

begin inducedefect
mast_write_method            no_setup
```

```
mast_ready_method            ready_defect
mast_run_method              run_defect
mast_complete_method         complete_structure
end

begin lowmesh
mast_kpoints 1x1x1 G
end

begin lowmesh_defect
mast_kpoints 1x1x1 G
isif 2
end

begin defect_relax
isif 2
end

begin static
ibrion -1
nsw 0
mast_multiplyencut 1.25
mast_update_children_method give_structure
end

begin static_to_neb
ibrion -1
nsw 0
mast_multiplyencut 1.25
mast_update_children_method give_structure_and_energy_to_neb
end

begin phonon
ibrion 5
mast_write_method write_phonon_single
mast_update_children_method give_phonon_single_forces_and_displacements
end

begin phononparse
mast_program phon
lfree .True.
temperature 1173
nd 3
qa 11
qb 11
qc 11
lsuper .False.
mast_exec //home/tam/tammast/bin/phon_henry
end

begin neb_to_neb
mast_kpoints 1x1x1 G
mast_program   vasp_neb
mast_write_method            write_neb
mast_update_children_method  give_neb_structures_to_neb
end

begin neb_to_nebstat
```

```
mast_program   vasp_neb
mast_write_method            write_neb
mast_update_children_method  give_neb_structures_to_neb
end

begin nebstat_to_phonon
mast_program   vasp
mast_write_method            write_neb_subfolders
mast_ready_method            ready_neb_subfolders
mast_run_method              run_neb_subfolders
mast_complete_method         complete_neb_subfolders
mast_update_children_method  give_saddle_structure
end

$end

$neb
begin int1-int2
X2, 0.25 0.25 0.25, 0.25 0.25 0.75
images 1
phonon int 0.25 0.25 0.5 0.5
phonon host 0.0 0.0 0.0 0.5
end
begin int1-int3
X2, 0.25 0.25 0.25, 0.75 0.25 0.25
images 1
phonon int 0.5 0.25 0.25 0.5
phonon host 0.0 0.0 0.0 0.5
end
$end

$recipe
recipe_file phonon_test_neb.txt
$end
```

CHAPTER

FIVE

THE RECIPE

5.1 Introduction to the Recipe

The recipe defines the relationships between ingredients, or which ingredients need to be run before which other ingredients.

Out-of-the-box recipes are stored in `$MAST_INSTALL_PATH/recipe_templates`. You may copy them into your `$MAST_RECIPE_PATH` directory (see *Installation*). If you create new recipes, they should also go in the `$MAST_RECIPE_PATH` directory

The full recipe name goes in the `$recipe` section of the input file:

```
$recipe
recipe_file neb.txt
```

$end

Important: when creating or editing recipes, do not use the Tab key. Instead, use 4 spaces to indent.

Also make sure that the recipe you are working with has not somehow been converted to tabs.

If you use vi as your code editor, consider adding the following settings to your `~/.vimrc file`, in order to use python four-space tab stops instead of the Tab character.:

```
set tabstop=4
set shiftwidth=4
set smarttab
set expandtab
set softtabstop=4
set autoindent
```

5.1.1 Syntax

Each indentation level marks a parent-child relationship.

perfect_opt1 (volrelax_lowmesh)

> **perfect_opt2** perfect_opt3

The ingredient type of an ingredient is specified in parentheses after the ingredient.

The ingredient type should correspond to ingredient subsections within the $ingredients section of the *input file*. If no ingredient type is specified, the ingredient gets all default values from the ingredients_global subsection.

In the recipe:

```
perfect_opt1 (volrelax_lowmesh)
```

In the input file:

$ingredients

begin volrelax_lowmesh mast_run_method run_singlerun ... end

$end

If the parent needs to update several children in different ways, create new trees where the originating parent is the same parent name, but with a different ingredient type. * Those different ingredient types should have different mast_update_children_method keyword values in the input file. * Only the first ingredient type specified per parent, going from the top of the file to the bottom of the file, will be used for all program keywords (run method, write method, INCAR settings, etc.) except for mast_update_children_method. The mast_update_children_method will be taken from the ingredient type specified between the parent and that child.

```
perfect_stat (stat_to_defect)
    defect_opt
perfect_stat (stat_to_phonon)
    phonon_opt1
```

If two children need to be the parent of one ingredient, also create a new tree:

```
perfect_stat
    defect_1_opt
    defect_2_opt
defect_1_opt, defect_2_opt
    neb_1-2_opt
```

Parent-child relationships are name-based, and the name must also include correct formats for defect labels (defect_XXX), charge labels (q=XX), neb labels (neb_XXX-XXX), and phonon labels (phonon_XXX). These names are important for following the tree structure and for setting the metadata file. Parent-child relationships are specified by these particular folder names. However, once all runs have been completed, post-processing utilities should only look at the metadata file within each run folder, and not at the folder name.

For defects, the labels must correspond to labels in the `$defects` section:

```
defect_<label>
```

Defect charges are given as q=p0 for no charge, q=nX for negative charge X (remember that negative charge means more electrons), and q=pX for positive charge X. (Please note that inducedefect ingredients should be labeled with inducedefect rather than with induce_defect, which will confuse them with defect ingredient labels.)

For nebs, the labels must correspond to labels in the `$neb` section:

```
neb_<label>
```

For phonons, the labels must correspond to labels in the `$phonon` section:

```
phonon_<label>
phonon_<label>_parse
```

You may create a fully-specified recipe in which you write out the labels, and also the charges, if necessary, for example:

defect_opt1_q=n2 (lowmesh)

However, in many cases it is more convenient to use abbreviations within the recipe. `{begin}` and `{end}` tags specify sections that can be looped over for as many defect labels `<N>` are specified in the `$defects` section of the input file and NEB labels `<B-E>`, where `<B>` and `<E>` are also defect labels, as specified in the `$neb` section of the input file.

Charges <Q> are given by the charge range in the $defects section. Available charges are carried into the <B-E>_<Q> labels based on which charges are available to both the <B> and the <E> defect in the label.

Note that defect endpoints need to be the parents of all NEB optimizations and NEB static calculations.

Example:

```
Recipe NEBtest
perfect_opt1 (lowmesh)
    perfect_opt2
        perfect_stat (static)
        {begin}
        inducedefect_<N> (inducedefect)
            defect_<N>_<Q>_opt1 (lowmesh_defect)
                defect_<N>_<Q>_opt2 (defect_relax)
                    defect_<N>_<Q>_stat (static)
        {end}
{begin}
defect_<N>_<Q>_stat (static)
    phonon_<N>_<Q>_<P> (phonon)
        phonon_<N>_<Q>_<P>_parse (phononparse)
{end}
{begin}
defect_<B>_<Q>_stat (static_to_neb), defect_<E>_<Q>_stat (static_to_neb)
    neb_<B-E>_<Q>_opt1 (neb_to_neb)
        neb_<B-E>_<Q>_opt2 (neb_to_nebstat)
            neb_<B-E>_<Q>_stat (nebstat_to_phonon)
    neb_<B-E>_<Q>_opt2 (neb_to_nebstat)
    neb_<B-E>_<Q>_stat (nebstat_to_phonon)
{end}
{begin}
neb_<B-E>_<Q>_stat (nebstat_to_phonon)
    phonon_<B-E>_<Q>_<P> (phonon)
        phonon_<B-E>_<Q>_<P>_parse (phononparse)
{end}
```

CHAPTER

SIX

RUNNING MAST

6.1 General notes

Depending on your cluster, you might find it polite to .nice. your processes:

```
nice -n 19 mast -i input.inp
```

nice -n 19 mast

This allows the headnode to put its regular functions before the mast processes. MAST should start running within several seconds.

6.2 Inputting an input file

To parse an input file, use

mast -i input.inp

or

mast -i //full/path/to/input/file/myinput.inp

If your input file specifies any POSCAR or CIF files, those files must be in your current working directory at the time you call MAST.

The input file will be parsed and a recipe directory should be created inside the `$MAST_SCRATCH` directory, with the appropriate ingredient subdirectories.

Look at the personalized_recipe.txt, input.inp, archive_input_options.txt, and archive_recipe_plan.txt files in the recipe directory to see if the setup agrees with what you think it should be.

6.3 Running MAST

Running MAST is separate from inputting input files. Use this command:

mast

This command will do two things:

1. Submit all ingredient runs listed in the `$MAST_CONTROL/submitlist` list to the queue

The submission command (sbatch, qsub, etc.) is based on the command in your `$MAST_INSTALL_PATH/submit/queue_commands.py` script.

Individual ingredients. submission scripts are created automatically through a combination of the `$ingredients` section in the input file, and your `$MAST_INSTALL_PATH/submit/script_commands.py` script.

2. Spawn a MAST monitor, or .mastmon., process on the queue.

Your `$MAST_INSTALL_PATH/submit/submit.sh` script is responsible for submitting this process, and should be set up to use the shortest, fastest turnover queue available (e.g. a serial queue with a maximum walltime of 4 hours, or morganshort on bardeen).

The mastmon process will generate additional entries on `$MAST_CONTROL/submitlist`, but these will not be submitted to the queue until MAST is called again.

6.3.1 The MAST monitor

The MAST monitor, or mastmon, process goes through the `$MAST_SCRATCH` directory. It looks at the folders there, which are recipe directories. For each recipe directory, the MAST monitor builds a .recipe plan. from a combination of the input.inp file, the personal_recipe.txt file, and the status.txt file. It then uses the recipe plan to assess the next steps appropriate for the recipe.

For human troubleshooting of a recipe, the archive_recipe_plan.txt file gives information about which ingredients are parents/children of which other ingredients, and which method each parent should use to update each of its child ingredients.

The status.txt files gives the status of each ingredient.

Ingredient statuses are: * I = initialized: The ingredient has just been created from inputting the input file, but nothing has been run. * W = waiting: The ingredient is waiting for parents to complete before it can be staged. * S = staged: All parents have updated this child, but the run is not yet ready to run * P = proceed: The ingredient has written its input files, all parents have updated it, and its run method has been called. The run method usually adds the ingredient to the list at `$MAST_CONTROL/submitlist`, to be submitted to the queue the next time mast is called. There is no MAST status change between an ingredient proceeding to the submitlist and being submitted to the queue off of the submitlist. However, `$MAST_CONTROL/submitted` can be used to see which ingredients were just submitted to the queue. * C = complete: The ingredient is complete * E = error: The ingredient has errored out, and ‘‘mast_auto_correc‘‘t was set to False in the input file (the default is True) * skip = skip: You can set ingredients to skip in the status.txt file by manually editing the file.

The MAST monitor checks the status of all ingredients whose status is not yet complete. The MAST monitor updates each ingredient status in the recipe plan.

Each ingredient is checked to see if it is complete (this is a redundant fast-forward check, since sometimes it is useful to copy over previously completed runs into a MAST ingredient directory.)

If complete, the ingredient updates its children and is changed to Complete

For each Initialized ingredient:

- If the ingredient has any parents, it is given status Waiting
- Otherwise, it is given status Staged

For each Proceed-to-run ingredient:

- If the ingredient is now complete, it updates its children and is changed to Complete

For each Waiting ingredient:

- If all parents are now marked complete, the ingredient is changed to Staged

For each Staged ingredient:

- If the ingredient is not already ready to run, its write method is called for it to write its input files.

- The ingredient.s run method is called, which usually adds its folder to `$MAST_CONTROL/submitlist`, except in the case of special run methods like run_defect (to induce a defect)
- The ingredient.s status is changed to Proceed.

When all ingredients in a recipe are complete, the entire recipe folder is moved from `$MAST_SCRATCH` to `$MAST_ARCHIVE`

6.3.2 The CONTROL folder

The `$MAST_CONTROL` folder houses several files:

- errormast: Contains any queue errors from running the MAST monitor on the queue
- mastoutput: Contains all queue output from running the MAST monitor on the queue, including a printout of the ingredient statuses for all recipes in the $MAST_SCRATCH directory
- submitlist: The list of all ingredient folders to be submitted to the queue
- submitted: A list of all ingredients submitted to the queue the last time the MAST monitor ran
- mast.log and archive.<timestamp>.log: contains MAST runtime information

Every file except `submitlist` can be periodically deleted to save space.

The `errormast` file is written when there is an error, and will need to be deleted for MAST to continue running.

6.3.3 The SCRATCH folder

The `$MAST_SCRATCH` folder houses all recipe folders. It also houses a mast.write_files.lock file while the MAST monitor is running, in order to prevent several versions of MAST from running at once and simultaneously checking and writing ingredients.

Skipping recipes or ingredients in the SCRATCH folder

If a certain recipe has some sort of flaw, or if you want to stop tracking it halfway through, you may have MAST skip over this recipe:

- Create an empty (or not, the contents don.t matter) file named MAST_SKIP in the recipe directory.
- Go through $MAST_CONTROL/submitlist and delete all ingredients associated with that recipe to keep them from being submitted during the next MAST run.

If you would like to skip certain ingredients of a single recipe, edit the recipe.s status.txt file and replace ingredients to be skipped with the status .skip. (use the whole word).

- To un-skip these ingredients, set them back to .W. for .waiting for parents. in status.txt. **Be careful if deleting any files for skipped ingredients. Do not delete the metadata.txt file. If deleting a file that was obtained from a parent, like a POSCAR file, also set the parent ingredient back to .P. when you un-skip the child ingredient.**
- No recipe can be considered complete by MAST if it includes skipped ingredients. However, if you consider the recipe complete, you can move the entire recipe directory out of `$MAST_SCRATCH` and into `$MAST_ARCHIVE` or another directory.

6.3.4 The ARCHIVE folder

When all ingredients in a recipe are complete, the entire recipe directory is moved from `$MAST_SCRATCH` to `$MAST_ARCHIVE`.

6.4 Running MAST repeatedly

The command `mast` needs to be run repeatedly in order to move the status of the recipe forward. In order to run mast automatically, use a crontab.

Important notes:

- Some clusters may not allow the use of cron. Please check the cluster policy before setting up cron.
- Be ready for a lot of notification emails. Crontab on a well-behaved system should send you an email each time it runs, giving you what would have been the output on the screen.
- Include `. $HOME/.bashrc` or a similar line to get your MAST environment variables and your usual path setup.

Crontab commands are as follows:

- `crontab -e` to edit your crontab
- `crontab -l` to view your crontab
- `crontab -r` to remove your crontab

This crontab line will run mast every hour at minute 15, and is usually suitable for everyday use:

```
15 * * * * . $HOME/.bashrc; nice -n 19 mast
```

This crontab line will run mast every 15 minutes and is ONLY suitable for short testing:

```
*/15 * * * * . $HOME/.bashrc; nice -n 19 mast
```

CHAPTER

SEVEN

MAST POST-PROCESSING UTILITIES

7.1 Defect formation energy

The defect formation energy tool goes through the output of finished recipes in $MAST_ARCHIVE and calculates defect formation energies. It is found in $MAST_INSTALL_PATH/tools.

The defect formation energy tool will create a `<recipe directory>_dfe_results` directory in the directory from which it is called.

To run without prompts:

```
python $MAST_INSTALL_PATH/tools/defect_formation_energy <DFT bandgap> <experimental bandgap>
```

where DFT bandgap is a float for an LDA or GGA bandgap, and experimental bandgap is a float for an experimental or more accurate hybrid calculation bandgap.

To run with prompts:

```
python $MAST_INSTALL_PATH/tools/defect_formation_energy prompt
```

- Select the desired recipe
- Follow the prompts for chemical potential conditions, band gap energy levels, and band gaps for adjustment

The two-column printout is Fermi energy on the left, and defect formation energy on the right.

7.2 Diffusion coefficient

Update this section with new diffusion coefficient help text

7.3 Defect finder

The defect finder takes a POSCAR file and finds vacancies and interstitials. The defect finder currently exists in a separate repository. You may test it online at materialshub.org > Resources > Tools > Defect Finder

CHAPTER

EIGHT

EXTERNAL PACKAGES

MAST is built using the following packages:

- pymatgen, pymatgen.org, Shyue Ping Ong, Anubhav Jai
- custodian, Shyue Ping Ong
- (Future plans) pymatgen-db, Anubhav Jain, Dan Gunter

MAST can interface with:

- PHON version 1.36, adapted, PHON webpage, Dario Alfe, Computer Physics Communications 180, 2622-2633 (2009)
- Vienna Ab-initio Simulation Package, VASP website, Jurgen Hafner, Georg Kresse, Doris Vogtenhuber, Martijn Marsman

CHAPTER

NINE

MAST PERSONNEL

PI: Professor Dane Morgan

Programmers by start date (+) indicates research performed using MAST):

- Tam Mayeshiba + (summer 2010 - present)
- Tom Angsten + (spring 2011 - summer 2013)
- Dr. Glen Jenness + (spring 2013 - summer 2013)
- Kumaresh Visakan Murugan (spring 2013, fall 2013-present)
- Hyunwoo Kim (spring 2013)
- Parker Sear (spring 2013 - summer 2013)
- Nada Alameddine (summer 2013)
- Jihad Naja (summer 2013)
- Dr. Henry Wu + (summer 2013 - present)

Additional team or project members:

- Ben Shrago (summer 2013)
- Amy Kaczmarowski (fall 2013 - present)
- Wei Xie (fall 2013 - present)
- Zhewen Song + (fall 2013 - present)

MAST users:

???

- *Method index*
- *Module index*
- *search*