MAST Documentation

Release 1.3.0.17

University of Wisconsin-Madison Computational Materials Group

July 08, 2015

CHAPTER

ONE

WHAT'S NEW IN VERSION 1.3.0

Additions:

- STEM image simulation functionality has been added to structopt. See StructOpt.
- A charge-density-based pathfinding method has been added as an alternative to linear interpolation for NEBs. See The Ingredients section.
- An optional atom indexing feature has been implemented for structures which are expected to undergo significant atomic relaxation when relaxed and/or defected. See The Structure section.
- A 14-frequency concentrated diffusion model has been added to the Diffusion Coefficient post-processing tool. See MAST post-processing utilities.
- The new MAST "modify recipe" function allows existing recipes to be modified. See Running MAST for real.

Fixes:

Changes for users:

- Scaling is its own section in the input file, and is no longer under the structure section. See The Scaling section for format changes.
- Instead of a mast_recipe.log file in each recipe directory, there is now one large mast.log file in $MAST_CONTROL.
- Instead of recipe-level trapped errors causing MAST to fail, MAST will create a MAST_ERROR document in the recipe directory and log a warning to `$MAST_CONTROL/mast.log`
- Standalone grain-boundary diffusion and diffusion analyzer tools must be unzipped from the MAST tar.gz file downloaded from github (see Programming for MAST). They will not be installed through pip.

Changes for programmers:

- In order to see some logged messages that are now logged to the DEBUG level, set environment variable MAST_DEBUG to any value.

CHAPTER

TWO

WHAT'S NEW IN VERSION 1.2.X

Additions:

- Finite size scaling support (the <S> tag) has been added. See The Structure section, The Recipe section, and MAST post-processing utilities.

Fixes:

- The Effective Grain Boundary Diffusivity Calculator and Particle Trajectory Diffusion Analysis packages are now properly included in the installation directory after running `setup.py`.

Changes for users:

- The `$recipe` section of the input file now requires the recipe to be entered directly.
  - Do not use a text file name any more.
  - Do not start with a recipe name line.
  - The `MAST_RECIPE_PATH` environment variable is no longer necessary.
- When the input file is processed, it will create a `$personal_recipe` section directly in the input file.
  - There is no longer a `personal_recipe.txt` file in the recipe directory.
  - If copying an input file for use in a new recipe, delete the `$personal_recipe` section from the new copy of the input file.
- MAST will now tell you where it was installed when you run `mast`.
- Platform support is now all under `<MAST installation directory>/submit/platforms/`.
  - The `platforms` folder is no longer copied to `$MAST_CONTROL`. See Installation for creating and modifying platforms.

Changes for programmers:

- Automatic citation support files are no longer copied to `$MAST_CONTROL`. They are located in `<MAST installation directory>/summary/citations`.
- Program key files are no longer copied to `$MAST_CONTROL`. They are located in `<MAST installation directory>/ingredients/programkeys`.
- Optimizer.py is no longer copied to `$MAST_CONTROL`. It is located in `<MAST installation directory>/structopt`
- mastmon_submit.sh is now copied from `<MAST installation directory>/submit/platforms/<platform name>` into `$MAST_CONTROL` each time that MAST is run. Edits should therefore be made to `<MAST installation directory>/submit/platforms/<platform name>/mastmon_submit.sh` if they are necessary.
- The `$MAST_CONTROL/set_platform` file is no longer used and references to it have been removed.

CHAPTER

THREE

INTRODUCTION

Welcome to the MAterials Simulation Toolkit (MAST)!

MAST is an automated workflow manager and post-processing tool.

MAST focuses on diffusion and defect workflows that use density functional theory. It interfaces primarily with the Vienna Ab-initio Simulation Package (VASP).

However, MAST can be generalized to other workflows and codes.

MAST is available from the Python Package Index.

Additional tools and unit tests are available through the latest MAST tar.gz file.

3.1 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, including information about how the ingredients are combined together.
  - As in a cooking recipe, ingredients may need to be addressed in a logical order, with some ingredients depending on other ingredients.
  - The Recipe section defines this order, or workflow.

When MAST reads an input file, it creates a recipe in the `$MAST_SCRATCH` directory.

- Many recipes can reside in `$MAST_SCRATCH`.
- MAST will check and update the recipes in alphanumeric order.

When MAST finds that a recipe is complete, it will move the recipe from `$MAST_SCRATCH` to `$MAST_ARCHIVE`.

3.2 Computing in the MAST Kitchen

1. Install MAST (see Installation).
2. 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?

  - Which calculations have a volume change?
  - Which calculations should be run at fixed volume?
  - How fine a kpoint mesh does each calculation need?
  - Etc...

3. Run an example file (see Trying out MAST) to get a feel for how MAST works.
4. Copy and modify an example file for your own workflow.

Please check your output carefully, especially when setting up a new workflow.

CHAPTER

FOUR

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.

The ingredient directory will contain:

- Any input files written by MAST or delivered by the parent ingredients.
- Any output files generated by the calculation
- A `metadata.txt` file, which stores important information for MAST
- A `jobids` file, which stores job ID numbers that the ingredient has had on the queue.

An ingredient object (created by MAST from the input file, and accessible to MAST while MAST is running) will have:

- A name, which is the full path to the ingredient's directory and is automatically generated from information in the input file.
- A dictionary of keywords, which come from the ingredient's **ingredient type** in The Ingredients section.
  - Program-specific keywords
  - 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
- A pymatgen Structure object representing the very first structure created from The Structure section.

CHAPTER FIVE

RECIPES

Each recipe is a collection of ingredients.

The recipe directory will contain:

- An `input.inp` file, which is a copy of the original input file and is used by MAST when checking the recipe. The original input file is not used. This copy also contains The Personal Recipe section, which is not in the original input file.
- Archive files from the initial setup of the recipe directory
- Ingredient directories
- A top-level `metadata.txt` file, which stores important information for MAST
- A `status.txt` file listing the status of each ingredient
- If an error is detected, a `MAST_ERROR` file will be created using the error text.

For other logging information, see the `$MAST_CONTROL/mast.log` file.

A recipe object (created by MAST from the input file, and accessible to MAST while MAST is running) will have:

- A name, which is the full path to the recipe's directory
- Several dictionaries which specify:
  - Which ingredient directories exist
  - Which ingredients have parents, and the names of those parent ingredients
  - Which method(s) each ingredient should run for each mast_xxx_method (see The Ingredients section)
    - Which method(s) each ingredient should run for its mast_update_children_method, depending on the name of the child ingredient

CHAPTER

SIX

INSTALLATION

6.1 Do pre-installation steps

6.1.1 Locate your user profile

Your user profile will set up environment variables like `$PATH` when you log in.

This installation will ask you to modify your user profile several times.

If you are comfortable modifying your user profile, please skip to *Use your cluster correctly*.

For others:

- Your user profile is probably located in your home directory as `//home/<username>/<user profile name>`, for example, `//home/<username>/.bashrc`
- Common user profile names are `.bashrc`, `.bash_profile`, `.profile`, and `.profile_user`
  - These names usually start with a dot.
  - You may need to use the command `ls -a` to see these "hidden" files.
  - Sometimes you may need to create your own user profile file.
    - For example, you may have a `.profile` file listed, but when you look at it, it tells you to create and modify a `.profile_user` file.
- **After you save your changes to the user profile, you need to log out and then log back in, in order to see the changes take effect.**
  - Alternately, you can `source <user profile name>`, but occasionally this command will produce complications, for example, in path order.

If you cannot locate your user profile, please contact your system administrator.

6.1.2 Use your cluster correctly

For this installation, please follow the correct procedures in order to avoid excessive headnode use on your cluster.

- For example, you may want to preface every command with `nice -n 19` in order to reduce headnode load.
- Or, your cluster may have a dedicated compile node, or it may support interactive queue submission.

Please check with your cluster administrator if you are unsure of the correct procedures.

6.2 Install local python version and dependencies

MAST requires the following dependencies, some of which have additional dependencies:

- numpy
- scipy
- matplotlib
- pymatgen
- custodian
- pandas
- ase

Currently, the Anaconda python package is preferred because it:

- automatically installs to a user directory and does not need root privileges
- does not have the environment complications that the previously-recommended Enthought Canopy python has
- can easily install numpy, scipy, and matplotlib.

6.2.1 Install Anaconda python

Download the free installer from Anaconda. Use a Python 2.X version.

- Run the setup script. (e.g. `bash ./Anaconda-<version>.sh`)
- Follow the prompts and specify a local installation (use spacebar to scroll through the license file).

When prompted, agree for the installer to add a line to your user profile to make this python installation your default python, for example:

```
export PATH=//home/<username>/anaconda/bin:$PATH
```

- Log out and log back in. Verify that your python version is now the anaconda python version.:

```
    which python
```

Use the conda command to install the following packages:

```
conda install numpy scipy matplotlib
```

If you are developing for MAST, install the following packages:

```
conda install numpy scipy matplotlib nose sphinx
```

6.2.2 Install the MAST package

Install the MAST package from the python package index:

```
pip install MAST
```

This command should install MAST, pymatgen, custodian, and pandas.

If pymatgen cannot be installed because gcc cannot be found in order to compile spglib, then please see your system administrator.

If you need additional standalone tools (see Standalone Tools) or unit tests, then also get and unzip the MAST tar.gz file (see Programming for MAST).

6.2.3 Install additional dependencies

- Do a quick installation of ASE following the instructions on the ASE website

6.3 Set up the pymatgen VASP_PSP_DIR

This step is necessary if you are running VASP with MAST. If you are not running VASP with MAST, skip to install-mast.

6.3.1 Set up the pseudopotential folders

Locate the VASP pseudopotentials. If you cannot locate the VASP pseudopotentials, contact your system administrator or another person who uses VASP on the cluster.

`which potcar_setup.py` should return the pymatgen utility for setting up your pseudopotential directories in the way that pymatgen requires. If this command does not return a file location, then probably `$HOME/.local/bin` or `<python installation directory>/bin` is missing from your `$PATH` environment variable. See add-local-bin.

Run `potcar_setup.py`:

```
potcar_setup.py
```

- The first directory address that you give to the utility is the directory that contains a few subdirectories, for example: potpaw_GGA, potpaw_LDA.52, potpaw_PBE.52, potUSPP_LDA, potpaw_LDA, potpaw_PBE, potUSPP_GGA.
  - These subdirectories themselves contain many sub-subdirectories with element names like Ac, Ac_s, Zr_sv, etc.
- The second directory address that you give should be a new directory that you create.

Once the new pymatgen-structured folders have been created, rename the GGA PBE folder to `POT_GGA_PAW_PBE`.

Later on, ingredients with a value of `pbe` for the ingredient keyword `mast_xc` will draw pseudopotentials out of this folder (see Input File).

Rename the GGA PW91 folder to `POT_GGA_PAW_PW91`. Ingredients with a value of `pw91` for the ingredient keyword `mast_xc` will draw pseudopotentials out of this folder.

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 directly from VASP, this should
typically be the directory that you untar the files to :

//share/apps/vasp_pseudopotentials/paw

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`:

```
mv //home/<username>/.local/vasp_pps/<pbe_name> //home/<username>/.local/vasp_pps/POT_GGA_PAW_PBE

mv //home/<username>/.local/vasp_pps/<pw91_name> //home/<username>/.local/vasp_pps/POT_GGA_PAW_PW91
```

For assistance with potcar_setup.py, please contact the Pymatgen support group

6.3.2 Add the VASP_PSP_DIR to your user profile

Add a line to your user profile exporting the environment variable `$VASP_PSP_DIR` to the new pseudopotential directory created above.

For example:

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

Log out and log back in.

Test the change:

```
cd $VASP_PSP_DIR
```

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

6.4 Set the MAST environment variables

The pip installation of MAST should have set up a `MAST` directory in your home directory, that is, `//home/<username>/MAST`.

- This directory is primarily for storing calculations, and should not be confused with the python module directory, which is where the actual MAST python code resides.

Inside `$HOME/MAST` there should be:

1. A `SCRATCH` folder:
   - Each time an input file is given to MAST, MAST will create a recipe directory inside this folder.
   - Each recipe directory will itself contain ingredient, or calculation, directories. Calculations will be submitted to the queue from inside these ingredient directories.
   - Multiple recipes may reside in `SCRATCH` at the same time, and MAST will evaluate them alphabetically.
2. An `ARCHIVE` folder:
   - When a recipe directory is complete, MAST will move it from `SCRATCH` to `ARCHIVE`.
3. A `CONTROL` folder:
   - MAST requires some control files in order to run. It also does some higher-level logging, and stores that output here.

- On some clusters, like Stampede, the home directory is not where you actually want to store calculations. Instead, there may be a separate "work" or "scratch" directory. In this case, move the entire `$HOME/MAST` directory into the work or scratch directory, for example:

```
    mv $HOME/MAST $WORK/.
```

In this case, the environment variables below should therefore say `$WORK` instead of `$HOME`.

- You can also move the MAST directory anywhere else, as long as you set the environment variables correcty.

Copy and paste the environment variables into your user profile, setting the paths correctly if you have moved the `$HOME/MAST` directory:

```
export MAST_SCRATCH=$HOME/MAST/SCRATCH
export MAST_ARCHIVE=$HOME/MAST/ARCHIVE
export MAST_CONTROL=$HOME/MAST/CONTROL
export MAST_PLATFORM=<platform_name>
```

For platform_name, choose from one of the following:

```
aci
bardeen
dlx
korczak
no_queue_system
pbs_generic
sge_generic
slurm_generic
stampede
turnbull
```

For example:

```
export MAST_PLATFORM=stampede
```

- If your platform is available by name (not _generic), then:
  - Add the four environment variable lines to your user profile as above.
  - Log out and log back in.
  - Go to *Additional setup*.
- If your platform is not matched exactly, or you would choose one of the generic choices:
  - Set the three other environment variables (MAST_SCRATCH, MAST_ARCHIVE, and MAST_CONTROL) in your user profile.
  - Log out and log back in.
  - Go to *Make a custom platform, if necessary*.

6.4.1 Make a custom platform, if necessary

Run the following command. It should produce some errors, but ignore those and just see where MAST is installed:

```
mast -i none
```

For example, output may be:

```
------------------------------------------------------
Welcome to the MAterials Simulation Toolkit (MAST)
Version: 1.1.5
Installed in: .local/lib/python2.7/site-packages/MAST
------------------------------------------------------
```

and then some errors.

Go to the "installed in" directory, and then:

```
cd submit/platforms
```

Identify the closest-matching directory to your actual platform (for example, if you have an SGE platform, this directory would be sge_generic)

Copy this directory into a new directory inside the `platforms` folder, for example:

```
cp -r sge_generic my_custom_sge
```

Then, inside your new folder, like `my_custom_sge`, modify each of the following files as necessary for your platform:

```
submit_template.sh
mastmon_submit.sh
queue_commands.py
```

Explanations for each file are given in the following sections. Modify and test each file in your new custom platform folder.

Then, in your user profile, use your new custom folder for the platform name of `$MAST_PLATFORM`:

```
export MAST_PLATFORM=my_custom_sge
```

Log out and log back in.

submit_template.sh

`submit_template.sh` is the generic submission template from which ingredient submission templates will be created.

- MAST will replace anything inside question marks, for example `?mast_ppn?` with the value of the appropriate keyword.

The following keywords may be used; see Input File for more information on each keyword.

- mast_processors
- mast_ppn
- mast_nodes
- mast_queue
- mast_exec
- mast_walltime
- mast_memory
- mast_name (the ingredient name)

Examine the template carefully, as an error here will prevent your ingredients from running successfully on the queue.

- The provided template should be a good match for its platform.
  - Otherwise, you can take one of your normal submission templates and substitute in `?mast_xxx?` fields where appropriate.
- Or, vice versa, you can take the provided template, replace the `?mast_xxx?` fields with some reasonable values, and see if filled-in submission template will run a job if submitted normally using `qsub`, `sbatch`, etc.

mastmon_submit.sh

mastmon_submit.sh is the submission template that will submit the MAST Monitor to the queue every time mast is called.

The MAST Monitor will check the completion status of every recipe and ingredient in the $MAST_SCRATCH folder.

- If you have a recipe you would like to skip temporarily, manually put a file named MAST_SKIP inside that recipe's folder in $MAST_SCRATCH. MAST_SKIP can be an empty file, or it can contain notes; MAST does not check its contents.
- mastmon_submit.sh should be set to run on the shortest-wallclock, fastest-turnaround queue available, e.g. a serial queue

The mastmon_submit.sh script is copied into the $MAST_CONTROL directory every time you run mast.

If you see that after you type mast, no "mastmon" process appears on the queue, then test the submission script directly:

```
cd $MAST_CONTROL
qsub mastmon_submit.sh (or use sbatch for slurm, etc.)
```

- Modify the $MAST_CONTROL/mastmon_submit.sh file until it the "mastmon" process successfully runs on the queue.
- Copy your changes into the <MAST installation directory>/submit/platforms/<platform>/mastmon_su file so that your changes will be reflected the next time that you run MAST.

queue_commands.py

These queue commands will be used to submit ingredients to the queue and retrieve the job IDs and statuses of ingredients on the queue.

- For a custom platform, modify the <MAST installation directory>/submit/platforms/<your custom platform>/queue_commands.py file.
- Do not modify the <MAST installation directory/submit/queue_commands.py file.

Modify the following python functions as necessary:

- queue_submission_command:
  - This function should return the correct queue submission command,
  - For example, this function should return qsub on PBS/Torque, or sbatch on slurm.
- extract_submitted_jobid:
  - This function should parse the job ID, given the text that returns to screen when you submit a job.
  - For example, it should return 456789 as the jobid for the following job submission and resulting screen text:

```
login2.mycluster$ sbatch submit.sh
-----------------------------------------------------------------
          Welcome to the Supercomputer
-----------------------------------------------------------------
--> Verifying valid submit host (login2)...OK
--> Verifying valid jobname...OK
--> Enforcing max jobs per user...OK
--> Verifying job request is within current queue limits...OK
Submitted batch job 456789
```

- On a different cluster, it would return `456789` as the jobid for the following submission and resulting screen text:

```
[user1@mycluster test_job]$ qsub submit.sh
456789.mycluster.abcd.univ.edu
```

- `queue_snap_command`:
  - This function should show a summary of your current submitted jobs, which we call the `queue_snapshot`.
  - For example, the queue snapshot command should return something like the following (platform-dependent):

```
JOBID   PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
456789      normal test1 user1 PD       0:00      4 (Resources)
456788      normal test2 user1 PD       0:00      1 (Resources)
456774      normal test3 user1  R    6:14:53      1 c123-124
456775      normal test4 user1  R    6:15:34      1 c125-126
```

- `queue_status_from_text`:
  - This function should return the status of a specific job, based on the job number.
  - For example, job 456789 in the queue snapshot above, with status “PD” should correspond to a “Q” status (queued status) for MAST.
  - Job 456775 in the queue snapshot above, with status “R”, should correspond to an “R” status (running status) for MAST.
- `get_approx_job_error_file`:
  - This function should return the name of the job error file.
  - The name of this file will depend on what is specified in `submit_template.sh` and is usually something like `slurm.<jobnumber>` or `<jobname>.e<jobnumber>`

6.5 Additional setup

You may need to do any or all of the following:

- Identify the correct `mast_exec` call for your system.
  - For example, suppose you run VASP like this:

```
//opt/mpiexec/bin/mpiexec //share/apps/bin/vasp5.2_par_opt1
```

  - Then, in your input files, the `mast_exec` keyword would be specified like this:

```
mast_exec //opt/mpiexec/bin/mpiexec //share/apps/bin/vasp5.2_par_opt1
```

- Add additional lines to your user profile which allow you to run VASP, including any modules that need to be imported, additions to your library path, unlimiting the stack size, and so on.
- Modify your text editor settings so that tabs become four spaces (or so that you have such an option readily available). This setting is very important to ensure that MAST can read the input file, especially the recipe section of the input file.
  - If you use VIM (`vi`), add the following lines 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
```

Once you have completed any additional setup and have identified what `mast_exec` should be, go to Trying out MAST.

CHAPTER SEVEN

TRYING OUT MAST

1. Go to `$HOME/MAST/examples` (or `$WORK/MAST/examples` or a similar folder, if you moved the `$HOME/MAST` folder from its default location.)
2. Select one of the examples. The fastest one is `simple_optimization.inp`
3. Copy that file:

```
cp simple_optimization.inp test.inp
```

4. Modify the test.inp file with the correct `mast_exec`, `mast_ppn`, `mast_queue`, `mast_walltime`, and other settings described in Input File
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 `$MAST_SCRATCH` directory should now have a recipe directory in it.
   - The recipe directory will have a name corresponding to the elements and the input file, and ending with a timestamp of YYYYMMDD"T"hhmmss.
   - The recipe directory will contain several subfolders, which are ingredient directories.
7. Go to that recipe directory.
   - To see the input options:
     - `cat input.inp` (should be identical to test.inp since no looping was used)
       - Note that you can use other viewing commands, not just `cat`, but be careful not to edit any of these files.
     - `cat archive_input_options.txt` (should show Al instead of element X1)
   - To see information about the ingredient relationships MAST detected from the recipe template:
     - `cat archive_recipe_plan.txt`
     - Look at the `$personal_recipe` section in the `input.inp` file
   - 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 the queue specified in `$MAST_CONTROL/mastmon_submit.sh`

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 `$MAST_SCRATCH` should show that `perfect_opt1` has a status of "Proceed to Queue", or "P".
13. When the queued `perfect_opt1` job starts running, you should be able to see output files inside `$MAST_SCRATCH/<recipe directory>/perfect_opt1`
14. If you forgot some step above, or you encounter some errors, remove the recipe folder from `$MAST_SCRATCH` and start again from the beginning of this section.
15. The `$MAST_CONTROL` folder gives you error messages and other information. See Running MAST for tips.

7.1 Unit testing

Unit tests are available through the MAST tar.gz file. See Programming for MAST. (Unit tests are not installed by default using pip.)

To check the validity of the MAST source code, navigate to `<MAST installation directory>/MAST/test` and run the unit tests with:

```
nosetests --exe
```

Some tests may have been designated to be skipped. Errors should be reported to the MAST development team as an issue on the github site (see Programming for MAST).

CHAPTER

EIGHT

INPUT FILE

When you use the command `mast -i <inputfile>.inp`, MAST does the following:

- Reads the input file
- Creates a recipe directory in `$MAST_SCRATCH`
- Creates the ingredient directories under that recipe directory
- Creates all the necessary metadata.txt files for that recipe and its ingredients.

MAST will then copy the input file into that recipe directory, as `input.inp`.

MAST will refer to this recipe-local `input.inp` file for all subsequent contact with the recipe.

8.1 General structure of the input file

The input file has many sections. Sections are denoted by `$<section name>` and `$end`:

```
$section

section_text
section_keyword keyword_value

$end
```

Within each section there may also be subsections, with keywords and values. Subsections are denoted by `begin <subsection name>` and `end`.:

```
$section

section_text
section_keyword keyword_value

begin subsection
subsection_text
subsection_keyword subsection_keyword_value
end

$end
```

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

8.2 Sections of the input file

See Sections of the Input File

8.3 Looping in the input file

If special looping tags are present in the input file, MAST can read in a single input file and create several permutated recipes in `$MAST_SCRATCH`.

The looping tag `indeploop` may be used to create combinatorial permutations.

- `indeploop` may be used once at the beginning of a line (that is not a section or subsection header or "end" line).
- `indeploop` may be used multiple times in an input file.

When `indeploop` is present at the beginning of the line, input file permutations will be created depending on the values in parentheses.

```
indeploop keyword1 (k1value1,k1value2)
```

The previous line would create two input files and corresponding recipes. On the line that used to have `indeploop` on it, one input file would have:

```
keyword1 k1value1
```

The other input file would have:

```
keyword1 l1value2
```

If indeploop tags are present multiple times in the recipe, input files are created combinatorially:

```
indeploop keyword1 (k1value1,k1value2)
indeploop keyword2 (k2value1,k2value2)
```

The previous two lines in an input file would create four input files and corresponding recipes. One input file would have:

```
keyword1 k1value1
keyword2 k2value1
```

Another would have:

```
keyword1 k1value1
keyword2 k2value2
```

A third would have:

```
keyword1 k1value2
kewyord2 k2value1
```

A fourth would have:

```
keyword1 k1value2
keyword2 k2value2
```

Sometimes, instead of combinatorial looping, some loops are meant to go together. In this case, the `pegloop1` and `pegloop2` tags may be used.

- There are only two pegged looping tags allowed, `pegloop1` and `pegloop2`.
- Each tag may be used only once on a line.
- Each tag may be used on multiple lines.

Every line that starts with `pegloop1` (the same will apply for `pegloop2`) will loop over keyword values, much like `indeploop`. However, the point of the pegged loops is to have two or more keywords loop together.

For example:

```
pegloop1 keyword1 (k1value1,k1value2)
pegloop1 keyword2 (k2value1,k2value2)
```

Using the `pegloop1` tag, the lines above would not produce four input files and corresponding recipes, as they would when using the `indeploop` tag. Instead, they would produce only two input files and corresponding recipes.

One input file would have:

```
keyword1 k1value1
keyword2 k2value1
```

The other input file would have:

```
keyword1 k1value2
keyword2 k2value2
```

The number of items in parentheses should be equal for all instances of the `pegloop1` (or, separately, the `pegloop2`) tag.

`pegloop1`, `pegloop2`, and all instances of `indeploop` will work combinatorially with each other.

Complex example (for looping only - many other necessary lines in the input file are skipped):

```
$mast
pegloop1 system_name (strain1,strain2,strain3)
$end

$structure
begin lattice
pegloop1 (3,4,5) 0 0
pegloop1 0 (3,4,5) 0
pegloop1 0 0 (3,4,5)
end

begin elementmap
pegloop2 X1 (Cr,Mn)
end
$end

$ingredients
begin ingredients_global
indeploop mast_xc (pw91,pbe)
LDAUJ 1
pegloop2 LDAUU (4.5,5)
end
$end
```

The above example would create 3*2*2 = 12 input files and corresponding recipes. The input file for the one of the recipes would look like:

```
$mast
system_name strain2
$end

$structure
begin lattice
4 0 0
0 4 0
0 0 4
end

begin elementmap
X1 Mn
end
$end

$ingredients
begin ingredients_global
indeploop mast_xc pbe
LDAUJ 1
LDAUU 5
end
$end
```

CHAPTER

NINE

SECTIONS OF THE INPUT FILE

9.1 The MAST section

The `$mast` section contains this keyword:

system_name: Specify a single descriptive word here.

This keyword will become part of the recipe directory's name, and allow you to spot the recipe more easily in the `$MAST_SCRATCH` directory.

This keyword comes in handy with pegged looping, in order to help you identify loops.

- Loops are otherwise differentiated by elements, if you were looping over elements, or simply by a 1-second timestamp difference.

Example:

```
$mast
system_name epitaxialstrain
$end
```

Example for pegged loop:

```
$mast
pegloop1 system_name (strain1,strain2,strain3)
$end
```

9.2 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).

9.2.1 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 at the time you call MAST, and should not be moved until the recipe is complete.

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.

9.2.2 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 in Input File).

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
```

9.2.3 Structure indexing (beta)

Some atomic positions in structures may change significantly after structural relaxation.

Without structure indexing, current operation of MAST is as follows: just before a defect, NEB, or phonon is about to be run, MAST decides which atoms to remove, single out for phonons, etc., based on a coordinate which is guessed at ahead of time, for example, 0.5 0.5 0.5, and sometimes a tolerance.

However, the relaxed atom may be quite far from that coordinate, so the search may fail.

When set to True, the **use_structure_index** keyword in the `$structure` section turns on atomic position indexing in the structure.:

```
$structure
use_structure_index True
...
$end
```

The coordinates in the `$defect`, `$neb`, etc. sections should exactly match the coordinates in the initial structure given to MAST. Scaling sizes will be handled automatically.

MAST will create a separate structure index file, a "manifest" file, for the initial structure and each scaling size and defect configuration. Each NEB endpoint will have a separate manifest file, and each phonon calculation will have a separate manifest for detailing selective dynamics information.

These files are stored in the `structure_index_files` directory within the recipe directory.

The manifest file consists of atomic index numbers, which correspond to `atom_index_<index number>` files. Each atomic index file is updated with the appropriate coordinates from a completed calculation.

When a calculation is complete, its relaxed atomic coordinates will be saved back to the appropriate atom_index file.

NEB notes for structure indexing

Special care should be taken when defining NEB calculations in the `$neb` section of the input file.

Again, vacancy and substitutional defects should have coordinates that exactly match those in the initial structure.

Coordinates in the `$neb` section should exactly match coordinates in the `$defects` section.

Defect manifests have their defects sorted to the bottom, except for vacancies. NEB manifests are created from defect manifests.

Atoms indicated in the `$neb` section are pulled out of their order in the manifest and put in order at the bottom.

As long as vacancies are accounted for in the `$neb` section, and grouped defects are entered in the same order in the defect endpoints in the `$defects` section, this process ensures lineup.

Grouped defects that are intended to persist but not move should be placed in the same order, that is, if defect1 and defect2 are going to be in an NEB and both have unmoving Sr antisite defects, then the Sr antisite defects should be in the same order in the begin defect1 and begin defect2 sections.

Pymatgen's interpolate function resorts atoms, so a temporary manifest is made for pymatgen, and the interpolation is resorted back to match the NEB endpoint manifests.

Also, all image atoms are ordered following the endpoint1 manifest. The final endpoint atoms are ordered following the endpoint2 manifest.

Therefore, after completion, coordinates for image atoms are recorded in the atom index files indicated by the endpoint 1 manifest.

9.3 The Scaling section

The `$scaling` section supports finite-size scaling through different supercell sizes.

9.3.1 Finite-size scaling

Finite size scaling is supported with a special “scaling” subsection.

Defect positions will be automatically scaled.

- For example, `0.25 0.0 0.0` in the original supercell would become `0.125 0.0 0.0` in a 2x1x1 cell.

Special notes:

- The Ingredients section should include an “inducescaling” ingredient with a `mast_run_method` of `run_scale`
- The Recipe section should include `inducescaling_<S>` and `defect_<S>` ingredients.
  - The “<S>” tags will correspond to the scaling sizes and labels.

For each scaling size, create a subsection with `begin <labelname>`.

Within the subsection, include keywords:

- **mast_size** with a scaling matrix of integers `[M, N, P]` or `[M1 M2 M3, N1 N2 N3, P1 P2 P3]`
- **mast_kpoints** with a Kpoint mesh in the form `QxRxS`, followed by a
- Kpoint mesh type, M for Monkhorst-Pack and G for Gamma-point centered
- (Optional) Kpoint mesh shift, in floats, e.g. `0.1 0.2 0.3`
- **WILL OTHER KEYWORDS NOW WORK AS WELL, FOR OVERRIDING?**

Example:

```
$scaling
begin 2xhigh
mast_size [2, 2, 2]
mast_kpoints 4x4x4 M
end
begin 4xhigh
mast_size [4, 4, 4]
mast_kpoints 2x2x2 M
end
$end
```

NEED TO VERIFY MADELUNG POTENTIAL CHANGES

In order to figure out which scaling sizes to use for finite-size scaling, MAST includes a Madelung potential utility.

This utility generates a distribution of cell sizes for best scaling, according to the method in:

```
Hine, N. D. M., Frensch, K., Foulkes, W. M. C. & Finnis, M. W. Supercell size scaling of density func
```

Run this utility as follows in order to generate a cut-and-paste for the scaling section.

```
mast_finite_size_scaling_sizes perfDir defDir minDefDist maxNumAtoms numStructAsked
```

- **perfDir**: perfect primordial (small) cell directory, which should already have run and include VASP CONTCAR, OSZICAR, etc. files.

- **defDir**: defected primordial cell directory, which should already have run and include VASP CONTCAR, OSZICAR, etc. files.
- **minDefDist** (default 3): minimum defect-defect distance between periodic images, in Angstroms.
- **maxNumAtoms** (default 600): maximum number of atoms for supercell size evaluations
- **numStructAsked** (default 5): number of structures to return in the distribution
- Note that you will have to manually adjust the kpoint mesh in your cut-and-paste.

9.4 The Ingredients section

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

Each ingredient type in the recipe should have a subsection denoted by `begin <ingredient type>`.

Example `$ingredients` section:

```
$ingredients

begin ingredients_global
keyword1 k1value1
end

begin ingredient_type1
keyword1 k1value2
keyword2 k2value1
end

begin ingredient_type2
keyword2 k2value2
end

$end
```

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

If there are no changes from the ingredients_global section, just add an empty subsection for that ingredient type:

```
begin ingredient_type
end
```

For a specific ingredient type, if a keyword is not specified in that ingredient type's subsection but is specified in the **ingredients_global** subsection, then, the value for that keyword will be taken from ingredients_global.

- In the example above, `ingredient_type2` would inherit `keyword1 k1value1` from ingredients_global.

9.4.1 Program-specific keywords

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 installation directory>/MAST/ingredients/programkeys/` folder, for example, `<MAST installation directory>/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.

9.4.2 Special MAST keywords

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

Submission script keywords

The following queue submission keywords are platform-dependent and are used along to create the submission script (see Installation).

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.

MAST control flow keywords

mast_program: Specify which program to run (`vasp`, `vasp_neb`, or `None` for a generic program, are currently supported)

```
mast_program vasp
```

- This keyword must be in lowercase

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.

VASP-specific keywords

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 POSCAR_initialize
```

- 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 POSCAR_im1,POSCAR_im2,POSCAR_im3
```

- 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.

Structure manipulation keywords

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_xxx_method keywords

The following keywords have individual sections:

mast_write_method: Specifies what the ingredient should write out before running (e.g., create the INCAR)

mast_ready_method: 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: 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: Specifies how MAST can tell if the ingredient is considered complete

mast_update_children_method: Specifies what information an ingredient passes on to its children, and how it does so.

Specific available values for each keyword are given in the accompanying sections, and require no arguments, e.g.:

```
mast_write_method write_singlerun
```

They depend on having an appropriate program set in `mast_program`.

However, you may choose to specify arguments where available, e.g.:

```
mast_complete_method file_has_string myoutput "End of Execution"
```

You may also choose to specify multiple methods. These methods will be performed in the order listed. For `mast_ready_method` or `mast_complete_method`, all methods listed must return True in order for the ingredient to be considered ready or complete, respectively. Use a semicolon to separate out the methods:

```
mast_complete_method file_has_string myoutput "End of Execution"; file_exists Parsed_Structures
```

In the example above, the file “myoutput” must exist and contain the phrase “End of Execution”, and the file “Parsed_Structures” must exist, in order for the ingredient to be considered complete.

Update-children methods will always get the child name appended as the end of the argument string. For example,

```
mast_update_children_method copy_file EndStructure BeginStructure
```

will copy the file EndStructure of the parent ingredient folder to a new file BeginStructure in the child ingredient folder. There is no separate argument denoting the child ingredient.

All arguments are passed as strings. Arguments in quotation marks are kept together.

Some common open-ended methods are:

- **file_exists <filename>**
- **file_has_string <filename> <string>**
- **copy_file <filename> <copy_to_filename>**
- **softlink_file <filename> <softlink_to_filename>**
- **copy_fullpath_file <full path file name> <copy_to_filename>**: This method is for copying some system file like //home/user/some_template, not an ingredient-specific file
- **write_ingred_input_file <filename> <allowed file> <uppercase keywords> <delimiter>**: The allowed file specifies an allowed keywords file name in `<MAST installation directory>/MAST/ingredients/programkeys`.
  - Use "all" to put any non-mast keywords into the input file.
  - Use 1 to uppercase all keywords, or 0 to leave them as entered.
  - Leave off the delimiter argument in order to use a single space.
  - Examples:

```
write_ingred_input_file input.txt all 0 =
write_ingred_input_file input.txt phon_allowed_keys.py 1
```

- **no_setup**: Does nothing. Useful when you want to specifically specify doing nothing.
- **no_update**: Does nothing (but, does accept the child name it is given). Useful when you want to specify doing nothing for a child update step.
- **run_command: <command string, including all arguments>**: This method allows you to run a python script.
  - The python script may take in only string-based arguments
  - Please stick to common text characters.
  - Example:

```
mast_run_method run_command "//home/user/myscripts/my_custom_parsing.py 25 0.01"
```

  - In the example above, the numbers 25 and 0.01 will actually be passed into sys.argv as a string.
  - This method is intended to allow you to run short custom scripts of your own creation, particularly for `mast_write_method` when setting up your ingredient.
  - **For long or complex execution steps where you want the output tracked separately, do not use this method. Instead,**
    # Use `write_submit_script` in your `mast_write_method`, along with any other write methods # Use `mast_run_method run_singlerun` # Put your script in the `mast_exec` keyword

mast_write_method keyword values

write_singlerun

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

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 and uses linear interpolation between images.
- Programs supported: vasp

write_pathfinder_neb

- Write an NEB ingredient using a charge-density-based pathfinding method.
- Programs supported: vasp
- This method takes the argument of an ingredient name for the ingredient from which to take the charge density.
  - The ingredient name must be fully specified, e.g. no <S>, <N>, etc. tags.
  - The ingredient must have a CHGCAR file written. A gamma-point calculation is sufficient for this purpose.
  - The ingredient should have both endpoints removed. For example, for vacancy migration, the ingredient should have neither a vacancy at the initial position, nor a vacancy at the final position. However, it should have all other non-migrating defects that are common to both the initial and final state.
- This method only works with `use_structure_index True` in the `$structure` section of the input file.

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 (see The Defects section).

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)
- Requires the `$scaling` subsection in the input file (see The Structure section).
- Must not be run on the starting ingredient.

mast_complete_method keyword values

complete_singlerun

- Check if run is complete
- Programs supported: vasp
- Note that for VASP, the phrase `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.
- This method is not for checking the completion of NEBs! An NEB ingredient should have `mast_program vasp_neb` and `mast_complete_method complete_singlerun`.
- An NEB static calculation, or a static calculation for each image, would use this keyword as `mast_complete_method complete_neb_subfolders` but have `mast_program vasp` instead of vasp_neb.

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_saddle_structure

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

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            POSCAR_vac1
end

begin singlerun_vac2
mast_coordinates            POSCAR_vac2
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            POSCAR_nebim1,POSCAR_nebim2,POSCAR_nebim3
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            POSCAR_nebim1_set2,POSCAR_nebim2_set2,POSCAR_nebim3_set2
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

$end
```

9.5 The Recipe section

The `$recipe` section of the input file contains information about how the ingredients are related to each other.

- This information complements the `mast_update_children_method` keyword given for each ingredient.

An ingredient in the recipe is referred to by:

```
<ingredient name> (ingredient type in $ingredients section)
```

For example:

```
perfect_opt1 (lowmesh_relaxation)
```

If no ingredient type is given, then only settings from the ingredients_global ingredient type of the input file will be used.

The ingredient name has some restrictions:

- For a simple workflow, the ingredient name may be fully and arbitrarily specified for the user.
- In most complex workflows, however, tags may be used as shortcuts to ingredient names. These tags will be filled in from information in the input file.
  - **<S>**: The `scaling` subsection of The Structure section
  - **<N>**: The Defects section
  - **<Q>**: The `charge` keyword in The Defects section
  - **<P>**: The `phonon` keyword in The Defects section and *3_1_6_neb*
  - **<B>, <E>, <B-E>**: The NEB section
  - The filled-in tags will be evident in The Personal Recipe section of the `input.inp` file in the recipe directory, once MAST has read the input file and set up the recipe directory.
- When tags are used, certain conventions must be followed:
  - Inducing scaling must use an `inducescaling_<S>` ingredient.
  - Inducing defects must use an `inducedefect_<N>` or `inducedefect_<S>_<N>` ingredient.
  - Defects must start with `defect`, and if tags are used, they must follow the order <S>, <N, B, or E>, <Q>, depending on which tags are used.

```
        defect_<S>_<N>_<Q>_arbitrarysuffix
```

  - Phonons must start with `phonon`, and if tags are used, they must follow the order <S>, <N or B-E>, <Q>, <P>
  - NEBs must start with `neb`, and if tags are used, they must follow the order <S>, <B-E>, <Q>

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

- See Installation for setting up text editors.
- Also make sure that the recipe you are working with has not somehow been converted to tabs.

9.5.1 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. 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:

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

- Those different ingredient types should have different mast_update_children_method keyword values in the input file.
- They should have all the same other keywords.

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 size-scaling labels <S>, defect labels <N, B, or E>, neb labels <B-E>, charge labels <Q>, and phonon labels <P>.

- 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.
- Some post-processing utilities may also rely on folder names.

The <S> tag The <S> tag will correspond to labels in the `scaling` subsection of The Structure section.

The <N>, <B>, <E>, and <B-E> tags For defects, the <N> tag will correspond to labels in The Defects section.

The same labels will be matched up and should be used as <B> and <E> labels (beginning and ending states) to correspond with NEBs, which are labeled <B-E>.

The NEB labels will correspond to labels in The NEB section

NEB label names must match up exactly with defect label names. For example, defect_vac1 and defect_vac2 must match up with neb_vac1-vac2.

Use <N> in a recipe unless specifying that a defect is a parent of an NEB, in which case use <B> or <E>:

```
{begin}
defect_<N>_opt1 (relax)
    defect_<N>_stat (static)
{end}

{begin}
defect_<B>_stat (static_to_neb), defect_<E>_stat (static_to_neb)
    neb_<B-E>_opt1 (neb)
{end}
```

The <Q> tag The <Q> tag will correspond to charges given in The Defects section.

- Charges are given as
  - q=p0 for no charge
  - q=nX for negative charge X (addition of electrons)
  - q=pX for positive charge X (removal of electrons)

{begin} and {end}

In the recipe, {begin} and {end} will loop over, match up, and fill in scaling labels <S>, defect labels <N, B, and E>, NEB labels <B-E>, charges <Q>, and phonons <P>

- Only charges in the charge range of both the <B> and <E> defect parents of an NEB will produce an charged NEB.
- Use a new {begin} and {end} when you have a new tree branch or unindentation in the recipe that switches between <N> and <B> or <E>
- Note that defect endpoints need to be the parents of all NEB optimizations and NEB static calculations. Therefore, the endpoint-neb parent-child block may look like the following:

```
{begin}
defect_<B>_stat (static_to_neb), defect_<E>_stat (static_to_neb)
    neb_<B-E>_opt1 (neb)
        neb_<B-E>_opt2 (neb)
            neb_<B-E>_stat (neb_static)
    neb_<B-E>_opt2 (neb)
    neb_<B-E>_stat (neb_static)
{end}
```

Full example:

```
$recipe
perfect_opt1 (lowmesh)
    perfect_opt2
        perfect_stat (static)
        {begin}
        inducescaling_<S>
            inducedefect_<S>_<N> (inducedefect)
                defect_<S>_<N>_<Q>_opt1 (lowmesh_defect)
                    defect_<S>_<N>_<Q>_opt2 (defect_relax)
                        defect_<S>_<N>_<Q>_stat (static)
        {end}
{begin}
defect_<S>_<N>_<Q>_stat (static)
```

```
    phonon_<S>_<N>_<Q>_<P> (phonon)
{end}
{begin}
defect_<S>_<B>_<Q>_stat (static_to_neb), defect_<S>_<E>_<Q>_stat (static_to_neb)
    neb_<S>_<B-E>_<Q>_opt1 (neb_to_neb)
        neb_<S>_<B-E>_<Q>_opt2 (neb_to_nebstat)
            neb_<S>_<B-E>_<Q>_stat (nebstat_to_phonon)
    neb_<S>_<B-E>_<Q>_opt2 (neb_to_nebstat)
    neb_<S>_<B-E>_<Q>_stat (nebstat_to_phonon)
{end}
{begin}
neb_<S>_<B-E>_<Q>_stat (nebstat_to_phonon)
    phonon_<S>_<B-E>_<Q>_<P> (phonon)
{end}
$end
```

9.6 The Personal Recipe section

The `$personal_recipe` section of the input file is generated by MAST and appears in the recipe-local `input.inp` file in the recipe directory.

Here, any <S>, <N>, <B>, <E>, <B-E>, <Q>, and <P> tags are extracted from the input file and substituted into the recipe.

The personal recipe is a good way to check exactly which ingredients MAST is going to run.

If errors are spotted in this section once the recipe directory has been created, it is best to:

1. Remove the entire new recipe directory from `$MAST_SCRATCH`
2. Examine the original input file's `$recipe` section for errors and edit and save it
3. Re-run `mast -i <inputfile>.inp` to create a new recipe directory in `$MAST_SCRATCH`

9.7 The Defects section

The `$defects` section specifies defects by:

- **defect type:**
  - vacancy
  - interstitial
  - substitution or antisite
- defect coordinates
- **defect element symbol**
  - Note that if an `elementmap` subsection is given in The Structure section, then the mapped designations `X1`, `X2`, and so on can be given instead of an element symbol.

ATTENTION:

- Elements in the initial structure, given in The Structure section, will appear in order as entered, by posfile keyword or through the coordinates and/or elementmap subsections.

- However, once a defect is formed, structures are RESORTED by element ELECTRONEGATIVITY. Therefore, if you are using substitutional defects or non-self-interstitials, you may find that later element-specific keywords (mast_setmagmom, LDAUU, LDAUJ) may be OUT OF ORDER FOR YOUR DEFECTED STRUCTURE.
- Please check your files carefully! You may want a separate input file for each chemical system (possibly created through looping (see Input File) in order to synchronize the elements and element-specific keywords.

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 in The Recipe section, 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 `<defect label>` subsection 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.

9.7.1 Charges for defects

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 must use a tagged `defect_<N>_<Q>` type recipe in The Recipe section.

9.7.2 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
- A float value for the phonon center radius
- An optional float value for the tolerance-matching threshold for matching the phonon center site (if this last value is not specified, 0.1 is used).

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 0.01
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. Points within 0.01 of fractional coordinates will be considered for matching the phonon center site.

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. As no threshold value was given, points within 0.1 (default) of fractional coordinates will be considered for matching the phonon center site.

The recipe template file for phonons may include either the explicit phonon labels and other labels, or <S>, <N>, <Q>, <P>. See The Recipe section.

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}
```

9.8 The NEB section

The `$neb` section includes a subsection for each nudged-elastic-band hops.

- Each neb hop should be a subsection labeled with a name composed of a starting and ending defect group, connected with a dash, like `vac1-vac2`.
  - These labels should correspond exactly to the labels given in The Defects section.
- The subsection should also include the movement of each primary moving atom, including:
  - The atom's element symbol: if an `elementmap` subsection is given in The Structure section, then the mapped designations `X1`, `X2`, and so on can be given instead of an element symbol.
  - 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.
  - The `images` keyword, which specifies the number of intermediate images.

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.

9.8.1 Charges for NEBs

To enable charged-supercell NEBs, use <Q> tags for the defect and NEB ingredients in The Recipe section and also specify charges for the defects in The Defects section.

The NEB ingredients will only be run for charges in the charge ranges of both parent endpoints.

For example, if defect parent vac1 has a charge range of charge=-2,0 and defect parent vac2 has a charge range of charge=-1,3, then the NEB with the label vac1-vac2 will only run with supercell charges -1 and 0.

9.8.2 Phonons for NEBs

Phonons may be specified within each NEB grouping, as in The Defects section.

The presumed saddle point in an NEB is usually taken.

- To give the saddle point structure to the phonon calculation, in The Ingredients section, use `mast_update_children give_saddle_structure` for the NEB ingredient type of the NEB parent to the phonon calculation.
- If the frequencies of a 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
```

9.9 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 subsection may be labeled descriptively.

```
$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
```

9.10 The Summary section

The `$summary` section of the input file will cause a `SUMMARY.txt` file to be printed into the recipe directory, once the recipe is complete.

Each line in the summary section follows the format:

```
<ingredient name search string> <summary information>
```

- <search string> is a search string for matching ingredient names.
- <summary information> refers to a python file in `<MAST installation directory>/summary` which is supposed to extract information from a given ingredient directory.
- For example, the following section would extract energies from each ingredient matching `vac` in its name.

  $summary vac energy $end

CHAPTER

TEN

RUNNING MAST FOR REAL

10.1 General notes

Depending on your cluster, you might find it necessary to *nice* your processes:

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

Nice-ing allows the headnode to put its regular functions before the MAST processes. MAST should start running within several seconds.

10.2 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 the same path as the original input file.
- Those files may not be moved until the recipe is complete.

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 `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.

10.3 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 platform chosen when you set $MAST_PLATFORM. See Installation.
- The exact commands can be found in your MAST installation path under submit/platforms/<platform name>/queue_commands.py.

Individual ingredients' submission scripts are created automatically through a combination of The Ingredients section in the input file, and your the template submission script for your platform

- The template submission script is found in your MAST installation path under submit/platforms/<platform name>/submit_template.sh).

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

- The mastmon_submit.sh and runmast.py files, originally located in your MAST installation path submit/platforms/<platform name> and submit folders, respectively, and then copied into $MAST_CONTROL when you first run mast, are are responsible for submitting this process.
- The script 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).
- You may make changes directly in $MAST_CONTROL/mastmon_submit.sh

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

10.3.1 The MAST monitor

The MAST monitor, or mastmon, process goes through the $MAST_SCRATCH directory.

- It looks at the recipe directories under $MAST_SCRATCH.
- For each recipe directory, the MAST monitor builds a Recipes plan object from information in the recipe directory, using a combination of the input.inp and status.txt files in the recipe directory.
- MAST then uses the recipe plan object to assess the next steps appropriate for the recipe, creating objects for the separate Ingredients and evaluating them.

10.3.2 Troubleshooting in a recipe directory

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_correct` 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 non-complete 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`

Errors in a recipe directory

Errors in a recipe which cause the recipe to fail out completely are logged to a `MAST_ERROR` file.

These errors will need to be addressed manually. Until then, MAST will skip over the recipe directory and log a warning to the mast.log file.

Once the error has been addressed, delete the `MAST_ERROR` file, and the recipe should be picked up on the next `mast` command.

To get more information about why the error may have been generated, set the `MAST_DEBUG` environment variable, e.g. `export MAST_DEBUG=1`, delete the `MAST_ERROR` file, and rerun MAST.

The error should be re-logged, and the `$MAST_CONTROL/mast.log` file will now also contain DEBUG-level information.

10.3.3 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. The default setting is INFO level. To also see DEBUG level information, set environment variable MAST_DEBUG, for example, `export MAST_DEBUG=1`.

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.

10.3.4 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.

- Occasionally, MAST may report that it is locked. If there is no *mastmon* process running or queued on the queue, you may delete the `mast.write_files.lock` file manually.

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 do not 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.

10.3.5 The ARCHIVE folder

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

10.4 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
```

10.5 Modifying recipes

Occasionally it is convenient to add additional ingredients onto an existent, completed or nearly-completed recipe.

For example, it may be helpful to add an additional charge state, or calculate phonons, make additional defects on a relaxed structure, or calculate additional NEBs.

The MAST "modify recipe" functionality allows new ingredient branches to be added onto an existing recipe in an existing recipe directory.

Instructions are as follows:

- In the recipe directory in $MAST_SCRATCH, modify the input file as you would want it. (If the recipe directory is not in $MAST_SCRATCH, move it there.)
  - For example, if the `$recipe` section uses the <N> <S> <Q> etc. tags, then the `$defects` section could add an additional `begin defectname ... end` subsection, or a charge designation within a defect subsection could be expanded.
- Remove the `$personal_recipe` section of the input file. (That is, remove the `$personal_recipe` line, all lines in between, and the `$end` line).
- From within the recipe directory, run the command `mast -m modifyrecipe`

These steps may be accomplished over multiple recipes using a shell script, but with caution.

10.5.1 Example

My charged supercell isn't charged! What happened?

My input file had charge=2,2 in the $defects section, but it did not have the charge tag <Q> in the $recipe section

The metadata.txt file wasn't getting written correctly, and the checker wasn't looking for a charge label, either.

Remove the $personal_recipe section. Redo the $recipe section to have the <Q> tags.

Run mast -m modifyrecipe

The uncharged supercell calculations were fine; move their data to folders with a <Q> tag for q=p0 (no charge).

Run mast (especially. mast -m monitoronly) until the status.txt file catches up Now mast will rerun a new arm of charged supercell calculations.

10.5.2 Caveats

- If ingredient names in the $recipe section are changed, some data may need to be moved around (see the example above).
- An already-complete ingredient is not necessarily rerun, depending on how its completion is evaluated. It may not get any new parent information from a newly added ingredient.
- The recipe's status.txt file is reset so that all ingredients are at status Initialized.
  - Each ingredient, whether previously completed or not, gets its state re-evaluated when MAST is called (using the normal `mast` command).
  - This procedure may require several `mast` calls until the recipe is caught up again.
  - This procedure is necessary in order to update all parent-child relationships and to establish the correct data transfer among the existing and new ingredients.

CHAPTER

ELEVEN

MAST POST-PROCESSING UTILITIES

These utilities are meant to be used as part of a MAST workflow. See example files in `$HOME/MAST/examples` or wherever you may have moved the initally-created `$HOME/MAST/examples` folder for examples on how to use them.

These utilities should have been copied into your bin or .local/bin directory (see Installation).

11.1 Defect formation energy with finite-size scaling

Initially determining the sizes for finite-size scaling is covered in The Structure section with the utility `mast_finite_size_scaling_sizes`.

The `mast_madelung_utility` utility runs as the last ingredient in a finite-size scaling defect workflow (see `$HOME/MAST/examples/finite_size_scaling.inp`).

Run the utility as `mast_madelung_utility`. All inputs are derived from the recipe-local `input.inp` file in the recipe directory.

- The utility should generate a series of tables and .png plots that display the finite-size-scaling-corrected and original defect formation energies for different chemical potentials.
- The Chemical Potentials section of the input file should be set in order for the utility to work.

11.2 Defect formation energy versus Fermi energy

The `mast_defect_formation_energy` tool plots defect formation energy versus Fermi energy.

The defect formation energy tool is intended to be run as another ingredient folder in the recipe directory.

If you do not have such an ingredient in the recipe directory, you may manually create the ingredient folder and give it a `dfe_input.txt` file.

The `dfe_input.txt` file for a manually-created or embedded workflow ingredient (see `//home/<username>/MAST/example/defect_formation_energy.inp`) should contain the following information:

```
dfe_label1=perfect_label defected_label
dfe_label2=perfect_label defected_label
dfe_label3=perfect_label defected_label
(etc. for more defects)
bandgap_lda_or_gga=<float>
bandgap_hse_or_expt=<float>
plot_threshold <float>: Plotting threshold value
```

- <perfect_label> and <defected_label> are the ingredient names of the perfected and corresponding defected cells.
- bandgap_lda_or_gga should be a float value indicating a DFT-calculated bandgap, usually expected to be underestimated.
- bandgap_hse_or_expt should be a float value indicating an experimental or more accurate bandgap, e.g. from a hybrid calculation.
- plot_threshold should be a float value indicating the threshold for transitions.
- In addition, The Chemical Potentials section should exist in the `input.inp` input file inside the recipe directory.

Run the utility as:

```
mast_defect_formation_energy dfe_input.txt
```

A directory named `dfe_results` should be created within the ingredient directory. Inside that directory:

- The two-column printout for each chemical potential-labeled text file contains Fermi energy on the left, and defect formation energy on the right.
- The `dfe.txt` printout contains defect formation energy information for each charge state.

11.3 Diffusion coefficient

The `mast_diffusion_coefficient` diffusion coefficient calculation tool supports the following models:

- FCC five-frequency model equation from R. E. Howard and J. R. Manning, Physical Review, Vol. 154, 1967.
- FCC concentrated fourteen-frequency model equation from Bocquet J.-L. and Le Claire A. D.
- HCP eight-frequency model equation from P. B. Ghate, Physical Review, Vol. 133, 1963.

The tool is designed to be used as a separate ingredient within the recipe directory. See `$HOME/MAST/examples/neb_with_phonons.inp` for an example input fileof a full workflow.

If the ingredient was not created within the workflow, an ingredient directory may be manually created for the tool.

The tool will use an input text file like `diffcoeff_input.txt`, which should contain the following lines. The order of the lines does not matter.

- Names of the directories of energies and attempt rates, which are specified with respect to different frequencies for the model:
  - **E** and **v** means energy and attempt rate, respectively. (There is no support for other characters such as w).
  - For 5-freq, **E0 through E4** should be used to specify the relations with certain directories
  - For 8-freq, **Ea, Eb, Ec, EX, Eap (p means prime), Ebp, Ecp, and EXp** should be used. Note they are all case sensitive and should be exactly the same as written here.
  - Generally speaking, each keyword (Exx or vxx) is followed by two ingredient names.
    - The first name indicates the ingredient name corresponding to the configuration of the starting point of NEB.
    - The second name indicates the ingredient name corresponding to the configuration of the saddle point of the NEB.
    - This order should not be changed.

        * For each name, the utility will expect two files to be present within the ingredient diretory of the diffusion coefficient tool:
            · <ingredient_name>_OUTCAR
            · <ingredient_name>_OSZICAR
            · If you are manually creating a diffusion coefficient tool ingredient, you will have to manually copy files from each of the completed ingredients specified.
    - The user can also type only one single float behind the keyword, and the code will then not refer to the directory for the related energy or attempting rate, but simply use the data given.

- **type** means which frequency model to choose. Either `5` or `fcc` tells the code that the five-frequency model should be applied, while either `8` or `hcp` tell the code that the eight-frequency model should be applied.
- **HVf** means the formation energy of the vacancy
    - Either 1 float or two ingredient names are expected after this keyword.
    - If ingredient names are used, in the order <perfect_ingredient> <defected_ingredient>, then the utility will expect two energy files to be present in the utility's ingredient directory:
        * <perfect_ingredient>_OSZICAR
        * <defected_ingredient>_OSZICAR
        * Charged defects are not currently supported.
- **HB** means the binding energy, and is only applicable for the 8-frequency model.
    - Either 1 float or four ingredient names are expected after this keyword.
    - If ingredient names are used:
        * Use the order <perfect ingredient> <vacancy and substitution> <substitution only> <vacancy only>
        * Supply an <ingredient_name>_OSZICAR file in the utility's ingredient directory.
- **lattice** indicates the ingredient name for the ingredient in which to find a lattice file.
    - This ingredient typically corresponds to an undefected supercell.
    - The utility expects to find a <lattice_ingredient_name>_POSCAR file inside the diffusion coefficient utility ingredient directory.
- **plotdisplay** indicates whether to use matplotlib.pyplot in order to create a plot, or whether to skip plotting.
    - Use "plotdisplay none" to skip plotting
    - Omit this keyword to use a default display
    - Use "plotdiplay tkagg" etc. or another display string to specify a matplotlib display.

Run as `mast_diffusion_coefficient -i <input>`

CHAPTER

TWELVE

STANDALONE TOOLS

12.1 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

12.2 Effective Grain Boundary Diffusivity Calculator

12.2.1 Effective Grain Boundary Diffusivity Calculator

Author: Jie Deng

Calculates the effective diffusivity in a grain boundary network with two types of randomly distributed grain boundaries.

Version 1.3 - published on 21 Feb 2014

Look for a new version in late 2014

- You must download and unzip the MAST tar.gz file from https://github.com/uw-cmg/MAST/releases in order to access the source code, which is in MAST/standalone/gbdiff, and will likely need to be recompiled for your machine.

This tool calculates the effective diffusivity in a grain boundary network represented by a three-dimensional Voronoi diagram. Two types of grain boundaries with different diffusivities are randomly distributed in the domain. The effective diffusivity is calculated using the mean squared displacement method, where periodic boundary conditions are applied in all directions. Users are free to choose the fraction of each grain boundary type as well as the activation energy and pre-factor for each grain boundary diffusivity.

12.2.2 Cite this work

Researchers should cite this work as follows:

```
Jie Deng (2014), "Effective Grain Boundary Diffusivity Calculator," https://materialshub.org/resource

@misc { 30,
    title = {Effective Grain Boundary Diffusivity Calculator},
    month = {Jan},
    url = {https://materialshub.org/resources/30},
    year = {2014},
    author = {Deng , Jie}
}
```

12.3 Particle Trajectory Diffusion Analyzer

12.3.1 Particle Trajectory Diffusion Analysis

Author: Leland Barnard Acknowledgments to: Amy Bengtson, Saumitra Saha

Computes mean squared displacements and tracer diffusion coefficients from particle position data as a function of time.

Version 1.13 - published on 28 Mar 2014

- You must download and unzip the MAST tar.gz file from https://github.com/uw-cmg/MAST/releases in order to access the source code, which is in MAST/standalone/diffanalyzer, and will likely need to be recompiled for your machine.

This tool takes as input particle position data from methods such as molecular dynamics or kinetic Monte Carlo and computes the mean squared displacement for all particles as a function of time. For a system with multiple types of particles, the mean squared displacement is computed for each particle type. The tracer diffusion coefficient is then calculated from the slope of the mean squared displacement vs time curve.

The tool is based on *The Working Man's Guide to Obtaining Self Diffusion Coefficients from Molecular Dynamics Simulations* by Professor David Keffer from UT Knoxville.

Input file format:

This tool reads in atomic position data in the VASP XDATCAR format. This file format begins with the following set of lines:

```
Name
C1 C2 C3 ...
N1 N2 N3 ...
Direct
```

- The first line is a name or description of the file. It is not read by the tool.
- The second line are the names of the components in the system. These will be element names in the case of an atomic simulation.
- The third line are the number of particles of each component in the system.
- The final line is a VASP generated line that specifies direct atomic coordinates.
- Following these 4 lines, the file must have 1 blank line, and then the particle position data begins on line 6. Particle positions must be in fractional or direct coordinates, and a single line must separate the blocks of particle positions at each time step throughout the file.

Calculation of error on the diffusion coefficient:

The error bars on the mean squared displacements represent a single standard deviation in the measurements of the squared displacements over all time origins.

The error in the diffusion coefficient represents the standard error in the slope of the weighted least squares fit to the mean squared displacement, using the variance in the squared displacements as the error weight.

References

"The Working Man's Guide to Obtaining Self Diffusion Coefficients from Molecular Dynamics Simulations" by Professor David Keffer from UT Knoxville, which may be found here: http://www.cs.unc.edu/Research/nbody/pubs/external/Keffer/selfD.pdf

Cite this work

Researchers should cite this work as follows:

```
Leland Barnard (2014), "Particle Trajectory Diffusion Analysis," https://materialshub.org/resources/c

@misc { 31,
    title = {Particle Trajectory Diffusion Analysis},
    month = {Feb},
    url = {https://materialshub.org/resources/31},
    year = {2014},
    author = {Barnard , Leland}
}
```

12.4 Diffusion Connectivity

12.4.1 Diffusion Connectivity

A new section is introduced in the input file:

```
$site
int1
0.5 0.5 0.5
0.5 0 0
0 0.5 0
0 0 0.5
int2
0.25 0.25 0.25
0.75 0.25 0.25
0.25 0.75 0.25
0.25 0.25 0.75
0.75 0.75 0.75
0.25 0.75 0.75
0.75 0.25 0.75
0.75 0.75 0.25
$end
```

In this example, there are two types of local minimum (interstitial site) int1 and int2. The geometrically equivalent site coordinates are listed for each type.

The create_paths.py code first parses the perfect lattice with the defect sites, then finds the nth neighbor of possible pairs of both same and different site types, and detect if the path candidate crosses over the host lattice site or another defect site and delete it. If the lattice user provides is too small and not all the neighbors (up to nth) are found, the code will double, triple, etc. the size until all required neighbor pairs are found.

The NEBcheck.py code will generate the MAST-input style defect structure for the possible pairs found in the create_paths.py and write a new input file that calls MAST to generate NEB and phonon folders to check if these paths are appropriate. Currently the code ends at calling MAST and does not yet manage to handle the NEB and phonon results analysis.

The usage is: python NEBcheck.py -i <input> -n <up-to-nth-neighbor>

12.5 StructOpt

12.5.1 StructOpt

Author: Amy Kaczmarowski

Research and development: Min Yu, Hyunseok Ko

- StructOpt is a general optimizer for python, targeted at identifying stable atomic structures.
- StructOpt is available as a standalone tool located at `<MAST installation directory or unzipped MAST-<version>.tar.gz directory>/MAST/structopt`.
  - Integration with MAST and use with VASP is under development.
- For more information, please see the user guide at `<MAST installation directory or unzipped MAST-<version>.tar.gz directory>/MAST/structopt/StructOpt_User_Guide_v1.docx`
  - If the user guide is not present, please download and unzip the full tar.gz file from http://dx.doi.org/10.5281/zenodo.11917 and look under `MAST/structopt`

STEM supplement to code

Author: Min Yu

Structopt has been extended to be able to read in a structure and simulate a STEM image using a convolution method.

CHAPTER

THIRTEEN

PROGRAMMING FOR MAST

13.1 Source code

The MAST github repository is located at https://github.com/uw-cmg/MAST.

To report any issues, please create an issue in this repository.

To program with MAST:

1. Clone from the dev branch (see github's instructions for cloning) OR get the latest stable release from *https://github.com/uw-cmg/MAST/releases* and unzip it.
2. Prepend the clone directory to your `$PYTHONPATH` environment variable, and the clone directory's `MAST/bin` directory to your `$PATH` environment variable.
   - The command `mast` should reveal the clone directory instead of any other MAST installation directories.

To run unit tests and verify that the MAST code is sound, go to the test directory in `<clone directory>/MAST/test` and run the command:

```
nosetests -v --nocapture
```

The `nocapture` option allows print statements. The `verbose` option gives verbose results.

The development team may have designated some tests to be skipped. However, any errors should be reported to the development team as a github issue.

13.2 Object hierarchy

Several objects are created in MAST. The classes for these objects are in similarly named files, for example, class MyClass in file myclass.py.

- When the user types `mast` or when crontab executes `mast`, a **MAST monitor** object is created (class MAST-mon in MAST). This monitor is responsible for looking through the `$MAST_SCRATCH` directory for recipe folders.
- For each recipe folder,
  - An **Input Options** object is created from the `input.inp` file (class InputOptions in MAST/utility, parsed from the input file through class InputParser in MAST/parsers)
  - A **Recipe Plan** object is created from that Input Options object
- The status of the ingredients in the recipe is given by `status.txt`
  - Depending on the ingredient status, an **Ingredient** object is created using information from the Recipe Plan object (class ChopIngredient, inheriting from class BaseIngredient, in MAST/ingredients)

  - That Ingredient object may involve several **Checker** objects for different programs based on the `mast_program` keyword of its ingredient type in The Ingredients section (class XXXChecker, MAST/ingredients/checker)

13.3 Code hooks in the input file

The most common modifications to MAST are expected to be:

- Adding support for new programs, e.g. besides VASP
- Adding new parent-child information transfer methods, for example:
  - Giving additional information to a child ingredient, like number of pairs
  - Accommodating different run structures, for example, forward on the least symmetric structure among several folders in the parent ingredient

Both of these modifications are currently coded in `MAST/ingredients/chopingredient.py` and in `MAST/ingredients/checker`

In the input file, the `mast_xxxx_method` keywords are direct hooks to methods in the **ChopIngredient** class.

- Methods are separated by semicolons, and can include arguments (see The Ingredients section.)
- The method in the ChopIngredient class may involve a checker, if they are generic but require program-specific treatment, for example, `forward_final_structure`.
- Or, the method in the ChopIngredient class may not need a checker, if it is totally generic, for example, `copy_file OLDNAME NEWNAME`
- When used as an update method, please remember that the last argument to a method is going to be the child ingredient's directory, as determined by The Personal Recipe section in the recipe folder.

Support for using a new checker type as self.checker in a ChopIngredient class would need to be added at the top of `MAST/ingredients/baseingredient.py`. Alternately, a new checker instance may be initialized on-the-fly within a method, e.g. mychecker = VASPChecker(name=mydirectory)

13.4 Debugging

For classes which have a self.logger attribute, or functions in which a logger is defined, messages may be logged to the DEBUG level. (self.logger.debug("message"))

Set the MAST_DEBUG environment variable to any value so that the mast.log file will print debug messages.

CHAPTER

FOURTEEN

ACKNOWLEDGMENTS

14.1 The MAST Team

PI: Professor Dane Morgan

All inquiries should be directed to ddmorgan@wisc.edu

14.1.1 Current Team

The following team members are arranged by start date and then alphabetically by last name. (+) indicates research performed using MAST.

- Tam Mayeshiba + (summer 2010 - present)
- Dr. Henry Wu + (summer 2013 - present)
- Amy Kaczmarowski (fall 2013 - present)
- Zhewen Song + (fall 2013 - present)
- Ben Afflerbach (fall 2014 - present)

14.1.2 Alumni

- Tom Angsten + (spring 2011 - summer 2013)
- Dr. Glen Jenness + (spring 2013 - summer 2013)
- Kumaresh Visakan Murugan (spring 2013, fall 2013, spring 2014)
- Hyunwoo Kim (spring 2013)
- Parker Sear (spring 2013 - summer 2013, spring 2014)
- Nada Alameddine (summer 2013)
- Jihad Naja (summer 2013)
- Dr. Wei Xie (fall 2013 - spring 2015)
- Jesus Chavez (summer 2014)
- Saswati De (summer 2014)
- Chandana Hosamane Kabbali (summer 2014)

14.2 NSF

The MAterials Simulation Toolkit (MAST) was developed with funding from the National Science Foundation Grant 1148011. T. Mayeshiba gratefully acknowledges support from the National Science Foundation Graduate Research Fellowship Grant No. DGE-0718123.

14.3 pymatgen

Many underlying MAST functions are built using pymatgen (http://pymatgen.org), and the MAST team would especially like to thank pymatgen developers Shyue Ping Ong and Anubhav Jain for their assistance.

CHAPTER FIFTEEN

CITATIONS

15.1 Citing MAST

To properly cite MAST and its dependencies, go to your completed recipe directory in `$MAST_ARCHIVE` and locate the following file

```
CITATIONS.bib
```

For example:

```
cat $MAST_ARCHIVE/Optimization_Al_20140101T120000/CITATIONS.bib
```

This Bibtex-formatted file may be used directly with LaTeX or imported into a reference manager such as EndNote or Mendeley.

15.2 Full list of possible citations

MAST chooses from the following citations when writing the `CITATIONS.bib` file:

15.2.1 MAST

- MAST development team. MAterials Simulation Toolkit (MAST). (2015). at <http://pypi.python.org/pypi/MAST>
- Angsten, T., Mayeshiba, T., Wu, H. & Morgan, D. Elemental vacancy diffusion database from high-throughput first-principles calculations for fcc and hcp structures. New J. Phys. 16, 015018 (2014).
- Kaczmarowski, A., Yang, S., Szlufarska, I. & Morgan, D. Genetic algorithm optimization of defect clusters in crystalline materials. Computational Materials Science 98, 234-244, doi:10.1016/j.commatsci.2014.10.062 (2015).
- Yu, Min, STEM additions to structopt package, https://git@github.com/uw-cmg/MAST/structopt (2015).
- Kitchaev, Daniil, Charge-based pathfinder for nudged elastic band calculations, https://git@github.com/uw-cmg/MAST/utility/daniil_pathfinder.py, 2015

15.2.2 pymatgen

- Ong, S. P. et al. Python Materials Genomics (pymatgen): A robust, open-source python library for materials analysis. Comput. Mater. Sci. 68, 314-319 (2013).

15.2.3 spglib

- Togo, A. Spglib. (2009). at <http://spglib.sourceforge.net/>

15.2.4 VASP

VASP main program

- Kresse, G. & Furthmüller, J. Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set. Phys. Rev. B 54, 11169-11186 (1996).
- Kresse, G. & Furthmüller, J. Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set. Comput. Mater. Sci. 6, 15-50 (1996).
- Kresse, G. & Hafner, J. Ab initio molecular-dynamics simulation of the liquid-metal-amorphous-semiconductor transition in germanium. Phys. Rev. B 49, 14251-14269 (1994).
- Kresse, G. & Hafner, J. Ab initio molecular dynamics for liquid metals. Phys. Rev. B 47, 558-561 (1993).

VASP pseudopotentials in general

- Kresse, G. & Hafner, J. Norm-conserving and ultrasoft pseudopotentials for first-row and transition elements. J. Phys. Condens. Matter 6, 8245-8257 (1994).

VASP PAW pseudopotentials

- Kresse, G. & Joubert, D. From ultrasoft pseudopotentials to the projector augmented-wave method. Phys. Rev. B 59, 1758-1775 (1999).

Nudged Elastic Band Calculations with VASP

- Henkelman, G. & Jónsson, H. Improved tangent estimate in the nudged elastic band method for finding minimum energy paths and saddle points. J. Chem. Phys. 113, 9978 (2000).
- Henkelman, G., Uberuaga, B. P. & Jónsson, H. A climbing image nudged elastic band method for finding saddle points and minimum energy paths. J. Chem. Phys. 113, 9901 (2000).
- Jónsson, H., Mills, G. & Jacobsen, K. W. in Class. Quantum Dyn. Condens. Phase Simulations (Berne, B. J., Ciccotti, G. & Coker, D. F.) 385 (World Scientific, 1998).
- Sheppard, D. & Henkelman, G. Paths to which the nudged elastic band converges. J. Comput. Chem. 32, 1769-71; author reply 1772-3 (2011).
- Sheppard, D., Terrell, R. & Henkelman, G. Optimization methods for finding minimum energy paths. J. Chem. Phys. 128, 134106 (2008).
- Sheppard, D., Xiao, P., Chemelewski, W., Johnson, D. D. & Henkelman, G. A generalized solid-state nudged elastic band method. J. Chem. Phys. 136, 074103 (2012).

15.2.5 Contact us for corrections

If you feel that we have missed or mis-typed a citation, please contact us (Contact Us).

CHAPTER SIXTEEN

CONTACT US

All inquiries about MAST should be made to Dane Morgan at ddmorgan@wisc.edu

CHAPTER

SEVENTEEN

LICENSE

The MAterials Simulation Toolkit is released with the MIT license, reproduced below: