Common Community Physics Package Single Column Model (SCM)

User and Technical Guide
v6.0.0

June 2022

Grant Firl
CIRA/CSU, NOAA GSL, and DTC

Dustin Swales, Laurie Carson, Michelle Harrold
NCAR and DTC

Ligia Bernardet
NOAA GSL and DTC

Dom Heinzeller
CU/CIRES, NOAA/GSL, and DTC
currently JCSDA

Acknowledgement

If significant help was provided via the helpdesk, email, or support forum for work resulting in a publication, please acknowledge the Developmental Testbed Center team.

For referencing this document please use:

Firl, G., D. Swales, L. Carson, L. Bernardet, D. Heinzeller, and M. Harrold, 2022. Common Community Physics Package Single Column Model v6.0.0 User and Technical Guide. 40pp. Available at https://dtcenter.org/sites/default/files/paragraph/scm-ccpp-guide-v6-0-0.pdf.

Contents

Preface

Meaning of typographic changes and symbols

Table 1 describes the type changes and symbols used in this book.

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.bashrc` Use `ls -a` to list all files. `host$ You have mail!`.
`AaBbCc123`	What you type, contrasted with on-screen computer output	`host$ su`
`AaBbCc123`	Command line placeholder: replace with a real name or value	To delete a file, type `rm` *`filename`*

Table 1: Typographic Conventions

1 Introduction

A single column model (SCM) can be a valuable tool for diagnosing the performance of a physics suite, from validating that schemes have been integrated into a suite correctly to deep dives into how physical processes are being represented by the approximating code. This SCM has the advantage of working with the Common Community Physics Package (CCPP), a library of physical parameterizations for atmospheric numerical models and the associated framework for connecting potentially any atmospheric model to physics suites constructed from its member parameterizations. In fact, this SCM serves as perhaps the simplest example for using the CCPP and its framework in an atmospheric model. This version contains all parameterizations of NOAA's evolved operational GFS v16 suite (implemented in 2021), plus additional developmental schemes. The schemes are grouped in six supported suites described in detail in the CCPP Scientific Documentation (GFS_v16, GFS_v17p8, RAP, HRRR, and RRFS_v1beta, and WoFS_v0).

This document serves as both the User and Technical Guides for this model. It contains a Quick Start Guide with instructions for obtaining the code, compiling, and running a sample test case, an explanation for what is included in the repository, a brief description of the operation of the model, a description of how cases are set up and run, and finally, an explanation for how the model interfaces with physics through the CCPP infrastructure.

Please refer to the release web page for further documentation and user notes:
https://dtcenter.org/community-code/common-community-physics-package-ccpp/download

1.1 Version Notes

The CCPP SCM v6.0.0 contains the following major and minor changes since v5.0.

Major

- Inclusion of regression testing functionality
- Combine single- and multi-run capabilities into one script

Minor

- Add RUC LSM support

- Add the GFS_v17p8, HRRR, RRFS_v1beta, and WoFS_v0 suites
- Update the vertical coordinate code to better match latest FV3 vertical coordinate code
- Simplify the case configuration namelists
- Add greater flexibility for output location (outside of bin directory)

1.1.1 Limitations

This release bundle has some known limitations:

- In the output file, temperature tendency variables all mistakenly have the same description, although their variable names are correct. This has been fixed in the development code.
- Using the RRFS_v1beta, HRRR, and WoFS_v0 suites for cases where deep convection is expected to be active will likely produce strange/unreliable results, unless the forcing has been modified to account for the deep convection. This is because forcing for existing cases assumes a horizontal scale for which deep convection is subgrid-scale and is expected to be parameterized. The suites without convection are intended for use with regional UFS simulations with horizontal scale small enough not to need a deep convection parameterization active, and it does not contain a deep convective scheme. Nevertheless, these suites are included with the SCM as-is for research purposes.
- The provided cases over land points cannot use an LSM at this time due to the lack of initialization data for the LSMs. Therefore, for the provided cases over land points (ARM_SGP_summer_1997_* and LASSO_*, where sfc_type = 1 is set in the case configuration file), prescribed surface fluxes must be used:
  - surface sensible and latent heat fluxes must be provided in the case data file
  - sfc_flux_spec must be set to true in the case configuration file
  - the surface roughness length in cm must be set in the case configuration file
  - the suite defintion file used (physics_suite variable in the case configuration file) must have been modified to use prescribed surface fluxes rather than an LSM.
  - NOTE: If one can develop appropriate initial conditions for the LSMs for the supplied cases over land points, there should be no technical reason why they cannot be used with LSMs, however.
- As of this release, using the SCM over a land point with an LSM is possible through the use of UFS initial conditions (see section **??**). However, advective forcing terms are unavailable as of this release, so only short integrations using this configuration should be employed. Using dynamical tendencies (advective forcing terms) from the UFS will be part of a future release.
- There are several capabilities of the developmental code that have not been tested sufficiently to be considered part of the supported release. Those include additional parameterizations. Users that want to use experimental capabilities should refer to Subsection 2.1.2.

2 Quick Start Guide

This chapter provides instructions for obtaining and compiling the CCPP SCM. The SCM code calls CCPP-compliant physics schemes through the CCPP framework code. As such, it requires the CCPP framework code and physics code, both of which are included as submodules within the SCM code. This package can be considered a simple example for an atmospheric model to interact with physics through the CCPP.

Alternatively, if one doesn't have access or care to set up a machine with the appropriate system requirements but has a working Docker installation, it is possible to create and use a Docker container with a pre-configured computing environment with a pre-compiled model. This is also an avenue for running this software with a Windows PC. See section 2.5 for more information.

2.1 Obtaining Code

The source code for the CCPP and SCM is provided through GitHub.com. This release branch contains the tested and supported version for general use, while a development branch is less stable, yet contains the latest developer code. Instructions for using either option are discussed here.

2.1.1 Release Code

Clone the source using

```
git clone --recursive -b v6.0.0 https://github.com/NCAR/ccpp-scm
```

Recall that the `recursive` option in this command clones the main ccpp-scm repository and all subrepositories (ccpp-physics and ccpp-framework). Using this option, there is no need to execute **`git submodule init`** and **`git submodule update`**.

The CCPP framework can be found in the ccpp/framework subdirectory at this level. The CCPP physics parameterizations can be found in the ccpp/physics subdirectory.

2.1.2 Development Code

If you would like to contribute as a developer to this project, please see (in addition to the rest of this guide) the scientific and technical documentation included with this release:

https://dtcenter.org/community-code/common-community-physics-package-ccpp/documentation

There you will find links to all of the documentation pertinent to developers.

For working with the development branches (stability not guaranteed), check out the **main** branch of the repository:

```
git clone --recursive -b main https://github.com/NCAR/ccpp-scm
```

By using the `recursive` option, it guarantees that you are checking out the commits of ccpp-physics and ccpp-framework that were tested with the latest commit of the SCM main branch. You can always retrieve the commits of the submodules that were intended to be used with a given commit of the SCM by doing the following from the top level SCM directory:

```
git submodule update --init --recursive
```

You can try to use the latest commits of the ccpp-physics and ccpp-framework submodules if you wish, but this may not have been tested (i.e. SCM development may lag ccpp-physics and/or ccpp-framework development). To do so:

1. Navigate to the ccpp-physics directory.
   ```
   cd ccpp-scm/ccpp/physics
   ```
2. Check out main.
   ```
   git checkout main
   ```
3. Pull down the latest changes just to be sure.
   ```
   git pull
   ```
4. Do the same for ccpp-framework
   ```
   cd ../framework
   git checkout main
   git pull
   ```
5. Change back to the main directory for following the instructions in section 2.3 assuming system requirements in section 2.2 are met.
   ```
   cd ../..
   ```

2.2 System Requirements, Libraries, and Tools

The source code for the SCM and CCPP components is in the form of programs written in FORTRAN, FORTRAN 90, and C. In addition, the I/O relies on the NetCDF libraries.

Beyond the standard scripts, the build system relies on use of the Python scripting language, along with cmake, GNU make and date.

The following software stacks have been tested with this code. Other versions of various components will likely still work, however.

- gfortran 12.1.0, gcc 12.1.0, cmake 3.23.2, NetCDF 4.7.4, Python 3.9.12
- GNU compilers 10.1.0, cmake 3.16.4, NetCDF 4.8.1, Python 3.7.12
- GNU compilers 11.1.0, cmake 3.18.2, NetCDF 4.8.1, Python 3.8.5
- Intel compilers 2022.0.2, cmake 3.20.1, NetCDF 4.7.4, Python 3.7.11
- Intel compilers 2022.1.0, cmake 3.22.0, NetCDF 4.8.1, Python 3.7.12

Because these tools are typically the purview of system administrators to install and maintain, they are considered part of the basic system requirements. The Unified Forecast System (UFS) Short-Range Weather Application release v1.0.0 of March 2021 provides software packages and detailed instructions to install these prerequisites and the hpc-stack on supported platforms (see section 2.2.3).

Further, there are several utility libraries as part of the hpc-stack package that must be installed with environment variables pointing to their locations prior to building the SCM.

- bacio - Binary I/O Library
- sp - Spectral Transformation Library
- w3emc - GRIB decoder and encoder library

The following environment variables are used by the build system to properly link these libraries: `bacio_ROOT`, `sp_ROOT`, and `w3emc_ROOT`. Computational platforms in which these libraries are prebuilt and installed in a central location are referred to as preconfigured platforms. Examples of preconfigured platforms are most NOAA high-performance computing machines (using the Intel compiler) and the NCAR Cheyenne system (using the Intel and GNU compilers). The machine setup scripts mentioned in section 2.3 load these libraries (which are identical to those used by the UFS Short and Medium Range Weather Applications on those machines) and set these environment variables for the user automatically. For installing the libraries and its prerequisites on supported platforms, existing UFS packages can be used (see section 2.2.3).

2.2.1 Compilers

The CCPP and SCM have been tested on a variety of computing platforms. Currently the CCPP system is actively supported on Linux and MacOS computing platforms using the Intel or GNU Fortran compilers. Windows users have a path to use this software through a Docker container that uses Linux internally (see section 2.5). Please use compiler versions listed in the previous section as unforeseen build issues may occur when using older versions. Typically the best results come from using the most recent version of a compiler. If you have problems with compilers,

please check the "Known Issues" section of the release website (https://dtcenter.org/community-code/common-community-physics-package-ccpp/download).

2.2.2 Using Existing Libraries on Preconfigured Platforms

Platform-specific scripts are provided to load modules and set the user environment for preconfigured platforms. These scripts load compiler modules (Fortran 2008-compliant), the NetCDF module, Python environment, etc. and set compiler and environment variables. From the top-level code directory (`ccpp-scm` by default), source the correct script for your platform and shell. For *t/csh* shells,

```
source scm/etc/Hera_setup_intel.csh
source scm/etc/Cheyenne_setup_gnu.csh
source scm/etc/Cheyenne_setup_intel.csh
```

For bourne/bash shells,

```
. scm/etc/Hera_setup_intel.sh
. scm/etc/Cheyenne_setup_gnu.sh
. scm/etc/Cheyenne_setup_intel.sh
```

2.2.3 Installing Libraries on Non-preconfigured Platforms

For users on supported platforms such as generic Linux or macOS systems that have not been preconfigured, the `hpc-stack` project is suggested for installing prerequisite libraries. Visit https://github.com/NOAA-EMC/hpc-stack for instructions for installing prerequisite libraries via `hpc-stack` in their docs directory. UFS users who already installed libraries via the `hpc-stack` package only need to set the compiler (`CC, CXX, FC`), NetCDF (`NetCDF_ROOT`), and `bacio`, `sp` and `w3emc` (`bacio_ROOT`, `sp_ROOT`, `w3emc_ROOT`) environment variables to point to their installation paths in order to compile the SCM.

The SCM uses only a small part of the UFS `hpc-stack` package and has fewer prerequisites (i.e. no `ESMF` or `wgrib2` needed). Users who are not planning to use the UFS can install only NetCDF/NetCDF-Fortran manually or using the software package manager (`apt`, `yum`, `brew`).

The Python environment must provide the `f90nml` module for the SCM scripts to function. Users can test if f90nml is installed using this command in the shell:

```
python -c "import f90nml"
```

If `f90nml` is installed, this command will succeed silently, otherwise an `ImportError: No module named f90nml` will be printed to screen. To install the `f90nml` (v0.19) Python module, use the install method preferred for your Python environment (one of the following):

- `easy_install f90nml==0.19`
- `pip install f90nml==0.19`
- `conda install f90nml=0.19`

or perform the following steps to install it manually from source:

```
cd /directory/with/write/priveleges
git clone -b v0.19 https://github.com/marshallward/f90nml
cd f90nml
python setup.py install [--prefix=/my/install/directory or --user]
```

The directory `/my/install/directory` must exist and its subdirectory `/my/install/directory/lib/python[version]/site-packages` (or `lib64` instead of `lib`, depending on the system) must be in the `PYTHONPATH` environment variable.

2.3 Compiling SCM with CCPP

The first step in compiling the CCPP and SCM is to properly setup your user environment as described in sections 2.2.2 and 2.2.3. The second step is to download the lookup tables and other large datasets (large binaries, <1 GB) needed by the physics schemes and place them in the correct directory: From the top-level code directory (`ccpp-scm` by default), execute the following scripts:

```
./contrib/get_all_static_data.sh
./contrib/get_thompson_tables.sh
```

If the download step fails, make sure that your system's firewall does not block access to GitHub. If it does, download the files `comparison_data.tar.gz`, `phyiscs_input_data.tar.gz`, `processed_case_input.tar.gz`, `raw_case_input.tar.gz` from the GitHub release website using your browser and manually extract its contents in the directory `scm/data`. Similarly, do the same for `thompson_tables.tar.gz` and `MG_INCCN_data.tar.gz` and extract to `scm/data/physics_input_data/`.

Following this step, the top level build system will use `cmake` to query system parameters, execute the CCPP prebuild script to match the physics variables (between what the host model – SCM – can provide and what is needed by physics schemes in the CCPP for the chosen suites), and build the physics caps needed to use them. Finally, `make` is used to compile the components.

1. From the top-level code directory (`ccpp-scm` by default), change directory to the top-level SCM directory.

   ```
   cd scm
   ```

2. Make a build directory and change into it.

```
mkdir bin && cd bin
```

3. Invoke **cmake** on the source code to build using one of the options below. This step is used to identify for which suites the ccpp-framework will build caps and which suites can be run in the SCM without recompiling.
   - Default mode

     ```
     cmake ../src
     ```

     By default, this option uses all supported suites. The list of supported suites is controlled by `scm/src/suite_info.py`.
   - All suites mode

     ```
     cmake -DCCPP_SUITES=ALL ../src
     ```

     All suites in `scm/src/suite_info.py`, regardless of whether they're supported, will be used. This list is typically longer for the development version of the code than for releases.
   - Selected suites mode

     ```
     cmake -DCCPP_SUITES=SCM_GFS_v16,SCM_RAP ../src
     ```

     This only compiles the listed subset of suites (which should still have a corresponding entry in `scm/src/suite_info.py`
   - The statements above can be modified with the following options (put before `../src`):
     - Use threading with openmp (not for macOS with clang+gfortran)

       ```
       -DOPENMP=ON
       ```

     - Debug mode

       ```
       -DCMAKE_BUILD_TYPE=Debug
       ```

   - One can also save the output of this step to a log file:

     ```
     cmake [-DCMAKE_BUILD_TYPE ...] ../src 2>&1 | tee log.cmake
     ```

   CMake automatically runs the CCPP prebuild script to match required physics variables with those available from the dycore (SCM) and to generate physics caps and makefile segments. It generates software caps for each physics group defined in the supplied Suite Definition Files (SDFs) and generates a static library that becomes part of the SCM executable.
   If necessary, the CCPP prebuild script can be executed manually from the top level directory (`ccpp-scm`). The basic syntax is

   ```
   ./ccpp/framework/scripts/ccpp_prebuild.py --config=./ccpp/config/
     ccpp_prebuild_config.py --suites=SCM_GFS_v16,SCM_RAP[...] --
     builddir=./scm/bin [--debug]
   ```

   where the argument supplied via the `--suites` variable is a comma-separated list of suite names that exist in the `./ccpp/suites` directory. Note that suite names are the suite definition filenames minus the **`suite_`** prefix and **`.xml`** suffix.
4. Compile. Add `VERBOSE=1` to obtain more information on the build process.

   ```
   make
   ```

   - One may also use more threads for compilation and/or save the output of the compilation to a log file:

     ```
     make -j4 2>&1 | tee log.make
     ```

The resulting executable may be found at `./scm` (Full path of `ccpp-scm/scm/bin/scm`).

Although `make clean` is not currently implemented, an out-of-source build is used, so all that is required to clean the build directory is (from the `bin` directory)

```
pwd #confirm that you are in the ccpp-scm/scm/bin directory before
    deleting files
rm -rfd *
```

Note: This command can be dangerous (deletes files without confirming), so make sure that you're in the right directory before executing!

If you encounter errors, please capture a log file from all of the steps, and start a thread on the support forum at: https://dtcenter.org/forum/ccpp-user-support/ccpp-single-column-model

2.4 Run the SCM with a supplied case

There are several test cases provided with this version of the SCM. For all cases, the SCM will go through the time steps, applying forcing and calling the physics defined in the chosen suite definition file using physics configuration options from an associated namelist. The model is executed through a Python run script that is pre-staged into the `bin` directory: `run_scm.py`. It can be used to run one integration or several integrations serially, depending on the command line arguments supplied.

2.4.1 Run Script Usage

Running a case requires four pieces of information: the case to run (consisting of initial conditions, geolocation, forcing data, etc.), the physics suite to use (through a CCPP suite definition file), a physics namelist (that specifies configurable physics options to use), and a tracer configuration file. As discussed in chapter 5, cases are set up via their own namelists in `../etc/case_config`. A default physics suite is provided as a user-editable variable in the script and default namelists and tracer configurations are associated with each physics suite (through `../src/suite_info.py`), so, technically, one must only specify a case to run with the SCM when running just one integration. For running multiple integrations at once, one need only specify one argument (`-m`) which runs through all permutations of supported suites from `../src/suite_info.py` and cases from `../src/supported_cases.py`. The run script's options are described below where option abbreviations are included in brackets.

- `--case [-c]`
  - **This or the `--multirun` option are the minimum required arguments.** The case should correspond to the name of a case in `../etc/case_config` (without the `.nml` extension).

- `--suite [-s]`
  - The suite should correspond to the name of a suite in `../ccpp/suites` (without the `.xml`) extension that was supplied in the `cmake` or `ccpp_prebuild` step.
- `--namelist [-n]`
  - The namelist should correspond to the name of a file in `../ccpp/physics_namelists` (WITH the `.nml` extension). If this argument is omitted, the default namelist for the given suite in `../src/suite_info.py` will be used.
- `--tracers [-t]`
  - The tracers file should correspond to the name of a file in `../etc/tracer_config` (WITH the `.txt` extension). If this argument is omitted, the default tracer configuration for the given suite in `../src/suite_info.py` will be used.
- `--multirun [-m]`
  - **This or the `--case` option are the minimum required arguments.** When used alone, this option runs through all permutations of supported suites from `../src/suite_info.py` and cases from `../src/supported_cases.py`. When used in conjunction with the `-- file` option, only the runs configured in the file will be run.
- `--file [-f]`
  - This option may be used in conjunction with the `--multirun` argument. It specifies a path and filename to a python file where multiple runs are configured.
- `--gdb [-g]`
  - Use this to run the executable through the `gdb` debugger (if it is installed on the system).
- `--docker [-d]`
  - Use this argument when running in a docker container in order to successfully mount a volume between the host machine and the Docker container instance and to share the output and plots with the host machine.
- `--runtime`
  - Use this to override the runtime provided in the case configuration namelist.
- `--runtime_mult`
  - Use this to override the runtime provided in the case configuration namelist by multiplying the runtime by the given value. This is used, for example, in regression testing to reduce total runtimes.
- `--levels [-l]`
  - Use this to change the number of vertical levels.
- `--npz_type`
  - Use this to change the type of FV3 vertical grid to produce (see `src/scm_vgrid.F90` for valid values).
- `--vert_coord_file`
  - Use this to specify the path/filename of a file containing the a_k and b_k coefficients for the vertical grid generation code to use.
- `--bin_dir`
  - Use this to specify the path to the build directory.
- `--run_dir`
  - Use this to specify the path to the run directory.

- `--case_data_dir`
  - Use this to specify the path to the directory containing the case data file (useful for using the DEPHY case repository).
- `--n_itt_out`
  - Use this to specify the period of writing instantaneous output in timesteps (if different than the default specified in the script).
- `--n_itt_diag`
  - Use this to specify the period of writing instantaneous and time-averaged diagnostic output in timesteps (if different than the default specified in the script).
- `--timestep [-dt]`
  - Use this to specify the timestep to use (if different than the default specified in `../src/suite_info.py`).
- `--verbose [-v]`
  - Use this option to see additional debugging output from the run script and screen output from the executable.

When invoking the run script, the only required argument is the name of the case to run. The case name used must match one of the case configuration files located in `../etc/case_config` (*without the .nml extension!*). If specifying a suite other than the default, the suite name used must match the value of the suite name in one of the suite definition files located in `../../ccpp/suites` (Note: not the filename of the suite definition file). As part of the sixth CCPP release, the following suite names are valid:

1. SCM_GFS_v16
2. SCM_GFS_v17p8
3. SCM_RAP
4. SCM_HRRR
5. SCM_RRFS_v1beta
6. SCM_WoFS_v0

Note that using the Thompson microphysics scheme requires the computation of look-up tables during its initialization phase. As of the release, this process has been prohibitively slow with this model, so it is HIGHLY suggested that these look-up tables are downloaded and staged to use this scheme as described in section 2.3. The issue appears to be machine/compiler-specific, so you may be able to produce the tables with the SCM, especially when invoking `cmake` with the `-DOPENMP=ON` option.

Also note that some cases require specified surface fluxes. Special suite definition files that correspond to the suites listed above have been created and use the `*_prescribed_surface` decoration. It is not necessary to specify this filename decoration when specifying the suite name. If the `spec_sfc_flux` variable in the configuration file of the case being run is set to `.true.`, the run script will automatically use the special suite definition file that corresponds to the chosen suite from the list above.

If specifying a namelist other than the default, the value must be an entire filename that exists in `../../ccpp/physics_namelists`. Caution should be exercised when modifying physics namelists since some redundancy between flags to control some physics parameterizations and scheme entries in the CCPP suite definition files currently exists. Values

of numerical parameters are typically OK to change without fear of inconsistencies. If specifying a tracer configuration other than the default, the value must be an entire filename that exists in `../../scm/etc/tracer_config`. The tracers that are used should match what the physics suite expects, lest a runtime error will result. Most of the tracers are dependent on the microphysics scheme used within the suite. The tracer names that are supported as of this release are given by the following list. Note that running without `sphum`, `o3mr`, and `liq_wat` may result in a runtime error in all supported suites.

1. sphum
2. o3mr
3. liq_wat
4. ice_wat
5. rainwat
6. snowwat
7. graupel
8. hailwat
9. cld_amt
10. water_nc
11. ice_nc
12. rain_nc
13. snow_nc
14. graupel_nc
15. hail_nc
16. graupel_vol
17. hail_vol
18. ccn_nc
19. sgs_tke
20. liq_aero
21. ice_aero
22. q_rimef

A NetCDF output file is generated in an output directory located named with the case and suite within the run directory. If using a Docker container, all output is copied to the `/home` directory in container space for volume-mounting purposes. Any standard NetCDF file viewing or analysis tools may be used to examine the output file (ncdump, ncview, NCL, etc).

2.4.2 Batch Run Script

If using the model on HPC resources and significant amounts of processor time is anticipated for the experiments, it will likely be necessary to submit a job through the HPC's batch system. An example script has been included in the repository for running the model on Hera's batch system (SLURM). It is located in `ccpp-scm/scm/etc/scm_slurm_example.py`. Edit the `job_name`, `account`, etc. to suit your needs and copy to the `bin` directory. The case name to be run is included in the `command` variable. To use, invoke

```
./scm_slurm_example.py
```

from the `bin` directory.

Additional details regarding the SCM may be found in the remainder of this guide. More information on the CCPP can be found in the CCPP Technical Documentation available at https://ccpp-techdoc.readthedocs.io/en/v6.0.0/.

2.5 Creating and Using a Docker Container with SCM and CCPP

In order to run a precompiled version of the CCPP SCM in a container, Docker will need to be available on your machine. Please visit https://www.docker.com to download and install the version compatible with your system. Docker frequently releases updates to the software; it is recommended to apply all available updates. NOTE: In order to install Docker on your machine, you will be required to have root access privileges. More information about getting started can be found at https://docs.docker.com/get-started

The following tips were acquired during a recent installation of Docker on a machine with Windows 10 Home Edition. Further help should be obtained from your system administrator or, lacking other resources, an internet search.

- Windows 10 Home Edition does not support Docker Desktop due to lack of "Hyper-V" support, but does work with Docker Toolbox. See the installation guide (https://docs.docker.com/toolbox/toolbox_install_windows/).
- You may need to turn on your CPU's hardware virtualization capability through your system's BIOS.
- After a successful installation of Docker Toolbox, starting with Docker Quickstart may result in the following error even with virtualization correctly enabled: `This computer doesn't have VT-X/AMD-v enabled. Enabling it in the BIOS is mandatory`. We were able to bypass this error by opening a bash terminal installed with Docker Toolbox, navigating to the directory where it was installed, and executing the following command:

  ```
  docker-machine create default --virtualbox-no-vtx-check
  ```

2.5.1 Building the Docker image

The Dockerfile builds CCPP SCM v6.0.0 from source using the GNU compiler. A number of required codes are built and installed via the DTC-supported common community container. For reference, the common community container repository can be accessed here: https://github.com/NCAR/Common-Community-Container.

The CCPP SCM has a number of system requirements and necessary libraries and tools. Below is a list, including versions, used to create the the GNU-based Docker image:

- gfortran - 9.3
- gcc - 9.3
- cmake - 3.16.5
- NetCDF - 4.6.2
- HDF5 - 1.10.4
- ZLIB - 1.2.7
- SZIP - 2.1.1
- Python - 3
- NCEPLIBS subset: bacio v2.4.1_4, sp v2.3.3_d, w3emc v2.9.2_d

A Docker image containing the SCM, CCPP, and its software prerequisites can be generated from the code in the software repository obtained by following section 2.1 by executing the following steps:

NOTE: Windows users can execute these steps in the terminal application that was installed as part of Docker Toolbox.

1. Navigate to the `ccpp-scm/docker` directory.
2. Run the `docker build` command to generate the Docker image, using the supplied Dockerfile.

   ```
   docker build -t ccpp-scm .
   ```

   Inspect the Dockerfile if you would like to see details for how the image is built. The image will contain SCM prerequisite software from DTC, the SCM and CCPP code, and a pre-compiled executable for the SCM with the 6 supported suites for the SCM. A successful build will show two images: dtcenter/common-community-container, and ccpp-scm. To list images, type:

   ```
   docker images
   ```

2.5.2 Using a prebuilt Docker image from Dockerhub

A prebuilt Docker image for this release is available on Dockerhub if it is not desired to build from source. In order to use this, execute the following from the terminal where Docker is run:

```
docker pull dtcenter/ccpp-scm:v6.0.0
```

To verify that it exists afterward, run

```
docker images
```

2.5.3 Running the Docker image

NOTE: Windows users can execute these steps through the Docker Quickstart application installed with Docker Toolbox.

1. Set up a directory that will be shared between the host machine and the Docker container. When set up correctly, it will contain output generated by the SCM within the container for manipulation by the host machine. For Mac/Linux,

   ```
   mkdir -p /path/to/output
   ```

   For Windows, you can try to create a directory of your choice to mount to the container, but it may not work or require more configuration, depending on your particular Docker installation. We have found that Docker volume mounting in Windows can be difficult to set up correctly. One method that worked for us was to create a new directory under our local user space, and specifying the volume mount as below. In addition, with Docker Toolbox, double check that the mounted directory has correct permissions. For example, open VirtualBox, right click on the running virtual machine, and choose "Settings". In the dialog that appears, make sure that the directory you're trying to share shows up in "Shared Folders" (and add it if it does not) and make sure that the "auto-mount" and "permanent" options are checked.
2. Set an environment variable to use for your SCM output directory. For *t/csh* shells,

   ```
   setenv OUT_DIR /path/to/output
   ```

   For bourne/bash shells,

   ```
   export OUT_DIR=/path/to/output
   ```

   For Windows, the format that worked for us followed this example: `/c/Users/my username/path/to/directory/to/mount`
3. To run the SCM, you can run the Docker container that was just created and give it the same run commands as discussed in section 2.4.1. **Be sure to remember to include the `-d` option for all run commands**. For example,

   ```
   docker run --rm -it -v ${OUT_DIR}:/home --name run-ccpp-scm ccpp-
       scm ./run_scm.py -c twpice -d
   ```

   will run through the TWPICE case using the default suite and namelist and put the output in the shared directory. NOTE: Windows users may need to omit the curly braces around environment variables: use `$OUT_DIR` instead of `${OUT_DIR}`. For running through all supported cases and suites, use

   ```
   docker run --rm -it -v ${OUT_DIR}:/home --name run-ccpp-scm ccpp-
       scm ./run_scm.py -m -d
   ```

   The options included in the above `run` commands are the following:
   - `--rm` removes the container when it exits
   - `-it` interactive mode with terminal access
   - `-v` specifies the volume mount from host directory (outside container) to inside the container. Using volumes allows you to share data between the host machine and container. For running the SCM, the output is being mounted from `/home` inside the container to the `OUT_DIR` on the host machine. Upon exiting the container, data mounted to the host machine will still be accessible.

- `--name` names the container. If no name is provided, the daemon will auto-generate a random string name.

NOTE: If you are using a prebuilt image from Dockerhub, substitute the name of the image that was pulled from Dockerhub in the commands above; i.e. instead of `ccpp-scm` above, one would have `dtcenter/ccpp-scm:v6.0.0`.

4. To use the SCM interactively, run non-default configurations, create plots, or even develop code, issue the following command:

```
docker run --rm -it -v ${OUT_DIR}:/home --name run-ccpp-scm ccpp-
    scm /bin/bash
```

You will be placed within the container space and within the `bin` directory of the SCM with a pre-compiled executable. At this point, one could use the run scripts as described in previous sections (remembering to include the `-d` option on run scripts if output is to be shared with the host machine). NOTE: If developing, since the container is ephemeral, one should push their changes to a remote git repository to save them (i.e. a fork on GitHub.com).

3 Repository

3.1 What is included in the repository?

The repository contains all code required to build the CCPP SCM and scripts that can be used to obtain data to run it (e.g. downloading large initialization tables for the Thompson microphysics schemes discussed in subsection 2.4.1 and processed case data). It is functionally separated into 3 subdirectories representing the SCM model infrastructure (`scm` directory), the CCPP infrastructure (`ccpp/framework` directory), and the CCPP physics schemes (`ccpp/physics` directory). The entire `ccpp-scm` repository resides on Github's NCAR space, and the `ccpp/framework` and `ccpp/physics` directories are git submodules that point to repositories `ccpp-framework` and `ccpp-physics` on the same space. The structure of the entire repository is represented below. Note that the `ccpp-physics` repository also contains files needed for using the CCPP with the UFS Atmosphere host model that uses the Finite-Volume Cubed-Sphere (FV3) dynamical core.

```
ccpp-scm/
├── ccpp/
│   ├── config/ ..... contains the CCPP prebuild configuration file
│   ├── framework/
│   │   └── See https://github.com/NCAR/ccpp-framework for contents
│   ├── physics/ ..... contains all physics schemes
│   │   └── See https://github.com/NCAR/ccpp-physics for contents
│   ├── physics_namelists ..... contains physics namelist files associated with suites
│   └── suites/ ..... contains suite definition files
├── CMakeModules/ ..... contains code to help cmake find other software
│   └── See https://github.com/noaa-emc/CMakeModules for contents
├── CODEOWNERS ..... list of GitHub users with permission to merge
├── contrib/
│   ├── get_all_static_data.sh .... script for downloading/extracting the processed SCM
│   │                               case data
│   ├── get_thompson_tables.sh . script for downloading/extracting the Thompson lookup
│   │                            tables
│   └── get_mg_inccn_data.sh . script for downloading/extracting the Morrison-Gettelman
│                              data
├── docker/
│   └── Dockerfile ..... contains Docker instructions for building the CCPP SCM image
├── README.md
└── scm/
```

- `bin/` ..... build directory (initially empty; populated by cmake)
- `data/` ..... build directory (most data directories populated by contrib/get_all_static_data.sh)
  - `comparison_data/` .. initially empty; contains data with which to compare SCM output
  - `physics_input_data/` ..... initially empty; contains data needed by the CCPP physics
  - `processed_case_input/` initially empty; contains initialization and forcing data for cases
  - `raw_case_input/` . initially empty; contains case data to be processed by scripts
  - `vert_coord_data/` ..... contains data to calculate vertical coordinates (from GSM-based GFS only)
- `doc/` ..... contains this User's/Technical Guide
  - `TechGuide/` ..... contains LaTeX for this User's Guide
- `etc/` ..... contains case configuration, machine setup scripts, and plotting scripts
  - `case_config/` ..... contains case configuration files
  - `CENTOS_docker_setup.sh` ..... contains machine setup for Docker container
  - `Cheyenne_setup_gnu.csh` ..... setup script for Cheyenne HPC for csh, tcsh
  - `Cheyenne_setup_gnu.sh` ..... setup script for Cheyenne HPC for sh, bash
  - `Cheyenne_setup_intel.csh` ..... setup script for Cheyenne HPC for csh, tcsh
  - `Cheyenne_setup_intel.sh` ..... setup script for Cheyenne HPC for sh, bash
  - `Desktop_setup_gfortran.csh` ..... setup script for Mac Desktop for csh, tcsh
  - `Desktop_setup_gfortran.sh` ..... setup script for Mac Desktop for sh, bash
  - `Hera_setup_intel.csh` ..... setup script for Theia HPC for csh, tcsh
  - `Hera_setup_intel.sh` ..... setup script for Theia HPC for sh, bash
  - `scm_qsub_example.py` ..... example QSUB run script
  - `scm_slurm_example.py` ..... example SLURM run script
  - `scripts/` ..... Python scripts for setting up cases and plotting
    - `plot_configs/` ..... plot configuration files
  - `tracer_config` ..... tracer configuration files
- `LICENSE.txt`
- `run/` ..... initially empty; populated by run_scm.py
- `src/` source code for SCM infrastructure, Python run script, CMakeLists.txt for the SCM, example multirun setup files, suite_info.py

4 Algorithm

4.1 Algorithm Overview

Like most SCMs, the algorithm for the CCPP SCM is quite simple. In a nutshell, the SCM code performs the following:

- Read in an initial profile and the forcing data.
- Create a vertical grid and interpolate the initial profile and forcing data to it.
- Initialize the physics suite.
- Perform the time integration, applying forcing and calling the physics suite each time step.
- Output the state and physics data.

In this chapter, it will briefly be described how each of these tasks is performed.

4.2 Reading input

The following steps are performed at the beginning of program execution:

1. Call `get_config_nml()` in the `scm_input` module to read in the `case_config` and `physics_config` namelists. This subroutine also sets some variables within the `scm_state` derived type from the data that was read.
2. Call `get_case_init()` (or `get_case_init_DEPHY()` if using the DEPHY format) in the `scm_input` module to read in the case input data file. This subroutine also sets some variables within the `scm_input` derived type from the data that was read.
3. Call `get_reference_profile()` in the `scm_input` module to read in the reference profile data. This subroutine also sets some variables within the `scm_reference` derived type from the data that was read. At this time, there is no "standard" format for the reference profile data file. There is a `select case` statement within the `get_reference_profile()` subroutine that reads in differently-formatted data. If adding a new reference profile, it will be required to add a section that reads its data in this subroutine.

4.3 Setting up vertical grid and interpolating input data

The CCPP SCM uses pressure for the vertical coordinate (lowest index is the surface). The pressure levels are calculated using the surface pressure and coefficients (a_k and b_k), which are taken directly from FV3 code (`fv_eta.h`). For vertical grid options, inspect `scm/src/scm_vgrid.F90` for valid values of `npz_type`. The default vertical coordinate uses 127 levels and sets `npz_type` to the empty string. Alternatively, one can specify the (a_k and b_k) coefficients via an external file in the `scm/data/vert_coord_data` directory and pass it in to the SCM via the `--vert_coord_file` argument of the run script. If changing the number of vertical levels or the algorithm via the `--levels` or `--npz_type` run script arguments, be sure to check `src/scm/scm_vgrid.F90` and `fv_eta.h` that the vertical coordinate is as inteneded.

After the vertical grid has been set up, the state variable profiles stored in the `scm_state` derived data type are interpolated from the input and reference profiles in the `set_state` subroutine of the `scm_setup` module.

4.4 Physics suite initialization

With the CCPP framework, initializing a physics suite is a 3-step process:

1. Initialize variables needed for the suite initialization routine. For suites originating from the GFS model, this involves setting some values in a derived data type used in the initialization subroutine. Call the suite initialization subroutine to perform suite initialization tasks that are not already performed in the `init` routines of the CCPP-compliant schemes (or associated initialization stages for groups or suites listed in the suite definition file). Note: As of this release, this step will require another suite intialization subroutine to be coded for a non-GFS-based suite to handle any initialization that is not already performed within CCPP-compliant scheme initialization routines.
2. Associate the `scm_state` variables with the appropriate pointers in the `physics` derived data type.
3. Call `ccpp_physics_init` with the `cdata` derived data type as input. This call executes the initialization stages of all schemes, groups, and suites that are defined in the suite definition file.

4.5 Time integration

The CCPP SCM uses the simple forward Euler scheme for time-stepping.

During each step of the time integration, the following sequence occurs:

1. Update the elapsed model time.
2. Calculate the current date and time given the initial date and time and the elapsed time.
3. Call the `interpolate_forcing()` subroutine in the `scm_forcing` module to interpolate the forcing data in space and time.
4. Recalculate the pressure variables (pressure, Exner function, geopotential) in case the surface pressure has changed.
5. Call `do_time_step()` in the `scm_time_integration` module. Within this subroutine:
   - Call the appropriate `apply_forcing_*` subroutine from the `scm_forcing` module.
   - For each column, call `ccpp_physics_run()` to call all physics schemes within the suite (this assumes that all suite parts are called sequentially without intervening code execution)
6. Check to see if output should be written during the current time step and call `output_append()` in the `scm_output` module if necessary.

4.6 Writing output

Output is accomplished via writing to a NetCDF file. If not in the initial spin-up period, a subroutine is called to determine whether data needs to be added to the output file during every timestep. Variables can be written out as instantaneous or time-averaged and there are 5 output periods:

1. one associated with how often instantaneous variables should be written out (controlled by the `-- n_itt_out` run script variable).
2. one associated with how often diagnostic (either instantaneous or time-averaged) should be written out (controlled by the `-- n_itt_diag` run script variable)
3. one associated with the shortwave radiation period (controlled by `fhswr` variable in the physics namelist)
4. one associated with the longwave radiation period (controlled by the `fhlwr` variable in the physics namelist)
5. one associated with the minimum of the shortwave and longwave radiation intervals (for writing output if any radiation is called)

Further, which variables are output and on each interval are controlled via the `scm/src/scm_output.F90` source file. Of course, any changes to this file must result in a recompilation to take effect. There are several subroutines for initializing the output file (`output_init_*`) and for appending to it (`output_append_*`) that are organized according to their membership in physics derived data types. See the `scm/src/scm_output.F90` source file to understand how to change output variables.

5 Cases

5.1 How to run cases

Only two files are needed to set up and run a case with the SCM. The first is a configuration namelist file found in `ccpp-scm/scm/etc/case_config` that contains parameters for the SCM infrastructure. The second necessary file is a NetCDF file containing data to initialize the column state and time-dependent data to force the column state. The two files are described below.

5.1.1 Case configuration namelist parameters

The `case_config` namelist expects the following parameters:

- `case_name`
  - Identifier for which dataset (initialization and forcing) to load. This string must correspond to a dataset included in the directory `ccpp-scm/scm/data/processed_case_input/` (without the file extension).
- `runtime`
  - Specify the model runtime in seconds (integer). This should correspond with the forcing dataset used. If a runtime is specified that is longer than the supplied forcing, the forcing is held constant at the last specified values.
- `thermo_forcing_type`
  - An integer representing how forcing for temperature and moisture state variables is applied (1 = total advective tendencies, 2 = horizontal advective tendencies with prescribed vertical motion, 3 = relaxation to observed profiles with vertical motion prescribed)
- `mom_forcing_type`
  - An integer representing how forcing for horizontal momentum state variables is applied (1 = total advective tendencies; not implemented yet, 2 = horizontal advective tendencies with prescribed vertical motion, 3 = relaxation to observed profiles with vertical motion prescribed)
- `relax_time`
  - A floating point number representing the timescale in seconds for the relaxation forcing (only used if `thermo_forcing_type` = `3` or `mom_forcing_type` = `3`)
- `sfc_flux_spec`
  - A boolean set to `.true.` if surface flux are specified from the forcing data (there is no need to have surface schemes in a suite definition file if so)

- `sfc_roughness_length_cm`
  - Surface roughness length in cm for calculating surface-related fields from specified surface fluxes (only used if `sfc_flux_spec` is True).
- `sfc_type`
  - An integer representing the character of the surface (0 = sea surface, 1 = land surface, 2 = sea-ice surface)
- `reference_profile_choice`
  - An integer representing the choice of reference profile to use above the supplied initialization and forcing data (1 = "McClatchey" profile, 2 = mid-latitude summer standard atmosphere)
- `year`
  - An integer representing the year of the initialization time
- `month`
  - An integer representing the month of the initialization time
- `day`
  - An integer representing the day of the initialization time
- `hour`
  - An integer representing the hour of the initialization time
- `column_area`
  - A list of floating point values representing the characteristic horizontal domain area of each atmospheric column in square meters (this could be analogous to a 3D model's horizontal grid size or the characteristic horizontal scale of an observation array; these values are used in scale-aware schemes; if using multiple columns, you may specify an equal number of column areas)
- `model_ics`
  - A boolean set to `.true.` if UFS atmosphere initial conditions are used rather than field campaign-based initial conditions
- `C_RES`
  - An integer representing the grid size of the UFS atmosphere initial conditions; the integer represents the number of grid points in each horizontal direction of each cube tile
- `input_type`
  - 0 => original DTC format, 1 => DEPHY-SCM format.

Optional variables (that may be overridden via run script command line arguments) are:

- `vert_coord_file`
  - File containing FV3 vertical grid coefficients.
- `n_levels`
  - Specify the integer number of vertical levels.

5.1.2 Case input data file (CCPP-SCM format)

The initialization and forcing data for each case is stored in a NetCDF (version 4) file within the `ccpp-scm/scm/data/processed_case_input` directory. Each file has at least

two dimensions (`time` and `levels`, potentially with additions for vertical snow and soil levels) and is organized into 3 groups: scalars, initial, and forcing. Not all fields are required for all cases. For example the fields `sh_flux_sfc` and `lh_flux_sfc` are only needed if the variable `sfc_flx_spec` = `.true.` in the case configuration file and state nudging variables are only required if `thermo_forcing_type` = `3` or `mom_forcing_type` = `3`. Using an active LSM (Noah, NoahMP, RUC) requires many more variables than are listed here. Example files for using with Noah and NoahMP LSMs are included in `ccpp-scm/scm/data/processed_case_input/fv3_model_point_noah[mp].nc`.

Listing 5.1: example NetCDF file (CCPP-SCM format) header for case initialization and forcing data

```
netcdf arm_sgp_summer_1997 {
dimensions:
 time = UNLIMITED ; // (233 currently)
 levels = UNLIMITED ; // (35 currently)
variables:
 float time(time) ;
  time:units = "s" ;
  time:description = "elapsed time since the beginning of the simulation" ;
 float levels(levels) ;
  levels:units = "Pa" ;
  levels:description = "pressure levels" ;

// global attributes:
  :description = "CCPP SCM forcing file for the ARM SGP Summer of 1997 case" ;

group: scalars {
  } // group scalars

group: initial {
  variables:
   float height(levels) ;
    height:units = "m" ;
    height:description = "physical height at pressure levels" ;
   float thetail(levels) ;
    thetail:units = "K" ;
    thetail:description = "initial profile of ice-liquid water potential temperature" ;
   float qt(levels) ;
    qt:units = "kg kg^-1" ;
    qt:description = "initial profile of total water specific humidity" ;
   float ql(levels) ;
    ql:units = "kg kg^-1" ;
    ql:description = "initial profile of liquid water specific humidity" ;
   float qi(levels) ;
    qi:units = "kg kg^-1" ;
    qi:description = "initial profile of ice water specific humidity" ;
   float u(levels) ;
    u:units = "m s^-1" ;
    u:description = "initial profile of E-W horizontal wind" ;
   float v(levels) ;
    v:units = "m s^-1" ;
    v:description = "initial profile of N-S horizontal wind" ;
   float tke(levels) ;
    tke:units = "m^2 s^-2" ;
    tke:description = "initial profile of turbulence kinetic energy" ;
   float ozone(levels) ;
    ozone:units = "kg kg^-1" ;
    ozone:description = "initial profile of ozone mass mixing ratio" ;
  } // group initial

group: forcing {
  variables:
   float lat(time) ;
    lat:units = "degrees N" ;
    lat:description = "latitude of column" ;
   float lon(time) ;
    lon:units = "degrees E" ;
    lon:description = "longitude of column" ;
```

```
    float p_surf(time) ;
     p_surf:units = "Pa" ;
     p_surf:description = "surface pressure" ;
    float T_surf(time) ;
     T_surf:units = "K" ;
     T_surf:description = "surface absolute temperature" ;
    float sh_flux_sfc(time) ;
     sh_flux_sfc:units = "K m s^-1" ;
     sh_flux_sfc:description = "surface sensible heat flux" ;
    float lh_flux_sfc(time) ;
     lh_flux_sfc:units = "kg kg^-1 m s^-1" ;
     lh_flux_sfc:description = "surface latent heat flux" ;
    float w_ls(levels, time) ;
     w_ls:units = "m s^-1" ;
     w_ls:description = "large scale vertical velocity" ;
    float omega(levels, time) ;
     omega:units = "Pa s^-1" ;
     omega:description = "large scale pressure vertical velocity" ;
    float u_g(levels, time) ;
     u_g:units = "m s^-1" ;
     u_g:description = "large scale geostrophic E-W wind" ;
    float v_g(levels, time) ;
     v_g:units = "m s^-1" ;
     v_g:description = "large scale geostrophic N-S wind" ;
    float u_nudge(levels, time) ;
     u_nudge:units = "m s^-1" ;
     u_nudge:description = "E-W wind to nudge toward" ;
    float v_nudge(levels, time) ;
     v_nudge:units = "m s^-1" ;
     v_nudge:description = "N-S wind to nudge toward" ;
    float T_nudge(levels, time) ;
     T_nudge:units = "K" ;
     T_nudge:description = "absolute temperature to nudge toward" ;
    float thil_nudge(levels, time) ;
     thil_nudge:units = "K" ;
     thil_nudge:description = "potential temperature to nudge toward" ;
    float qt_nudge(levels, time) ;
     qt_nudge:units = "kg kg^-1" ;
     qt_nudge:description = "q_t to nudge toward" ;
    float dT_dt_rad(levels, time) ;
     dT_dt_rad:units = "K s^-1" ;
     dT_dt_rad:description = "prescribed radiative heating rate" ;
    float h_advec_thetail(levels, time) ;
     h_advec_thetail:units = "K s^-1" ;
     h_advec_thetail:description = "prescribed theta_il tendency due to horizontal
         advection" ;
    float v_advec_thetail(levels, time) ;
     v_advec_thetail:units = "K s^-1" ;
     v_advec_thetail:description = "prescribed theta_il tendency due to vertical
         advection" ;
    float h_advec_qt(levels, time) ;
     h_advec_qt:units = "kg kg^-1 s^-1" ;
     h_advec_qt:description = "prescribed q_t tendency due to horizontal advection" ;
    float v_advec_qt(levels, time) ;
     v_advec_qt:units = "kg kg^-1 s^-1" ;
     v_advec_qt:description = "prescribed q_t tendency due to vertical advection" ;
  } // group forcing
}
```

5.1.3 Case input data file (DEPHY format)

The Development and Evaluation of Physics in atmospheric models (DEPHY) format is an internationally-adopted data format intended for use by SCM and LESs. The initialization and forcing data for each case is stored in a NetCDF (version 4) file, although these files are not by default included in the CCPP SCM repository. To access these

cases you need to clone the DEPHY-SCM repository, and provide the DEPHY-SCM file location to the SCM. For example:

```
cd [...]/ccpp-scm/scm/data
git clone https://github.com/GdR-DEPHY/DEPHY-SCM DEPHY-SCM
cd [...]/ccpp-scm/scm/bin
./run_scm.py -c MAGIC_LEG04A --case_data_dir [...]/ccpp-scm/scm/data/
   DEPHY-SCM/MAGIC/LEG04A -v
```

Each DEPHY file has three dimensions (`time`, `t0`, `levels`) and contains the initial conditions (`t0`, `levels`) and forcing data (`time`, `levels`). Just as when using the CCPP-SCM formatted inputs, 5.1.2, not all fields are required for all cases. More information on the DEPHY format requirements can be found at DEPHY.

Listing 5.2: example NetCDF file (DEPHY format) header for case initialization and forcing data

```
netcdf MAGIC_LEG04A_SCM_driver {
dimensions:
 t0 = 1 ;
 time = 214 ;
 lev = 2001 ;
variables:
 double t0(t0) ;
  t0:standard_name = "initial_time" ;
  t0:units = "seconds since 2012-10-20 18:00:00" ;
  t0:calendar = "gregorian" ;
 double time(time) ;
  time:standard_name = "forcing_time" ;
  time:units = "seconds since 2012-10-20 18:00:00" ;
  time:calendar = "gregorian" ;
 double lev(lev) ;
  lev:standard_name = "height" ;
  lev:units = "m" ;
 float zh(t0, lev) ;
  zh:standard_name = "height" ;
  zh:units = "m" ;
  zh:coordinates = "t0 zh lat lon" ;
 float pa(t0, lev) ;
  pa:standard_name = "air_pressure" ;
  pa:units = "Pa" ;
  pa:coordinates = "t0 zh lat lon" ;
  ta:coordinates = "t0 zh lat lon" ;
 float theta(t0, lev) ;
  theta:standard_name = "air_potential_temperature" ;
  theta:units = "K" ;
  theta:coordinates = "t0 zh lat lon" ;
 float thetal(t0, lev) ;
  thetal:standard_name = "air_liquid_potential_temperature" ;
  thetal:units = "K" ;
  thetal:coordinates = "t0 zh lat lon" ;
 float qv(t0, lev) ;
  qv:standard_name = "specific_humidity" ;
  qv:units = "1" ;
  qv:coordinates = "t0 zh lat lon" ;
 float qt(t0, lev) ;
  qt:standard_name = "mass_fraction_of_water_in_air" ;
  qt:units = "1" ;
  qt:coordinates = "t0 zh lat lon" ;
 float ps_forc(time) ;
  ps_forc:standard_name = "forcing_surface_air_pressure" ;
  ps_forc:units = "Pa" ;
  ps_forc:coordinates = "time lat lon" ;
 float ug(time, lev) ;
  ug:standard_name = "geostrophic_eastward_wind" ;
  ug:units = "m s-1" ;
  ug:coordinates = "time zh_forc lat lon" ;
 float vg(time, lev) ;
```

```
  vg:standard_name = "geostrophic_northward_wind" ;
  vg:units = "m s-1" ;
  vg:coordinates = "time zh_forc lat lon" ;
 float tnta_adv(time, lev) ;
  tnta_adv:standard_name = "tendency_of_air_temperature_due_to_advection" ;
  tnta_adv:units = "K s-1" ;
  tnta_adv:coordinates = "time zh_forc lat lon" ;

// global attributes:
  :case = "MAGIC/LEG04A" ;
  :title = "Forcing and initial conditions for MAGIC Leg04A case - SCM-enabled version"
      ;
  :reference = "J. McGibbon, C. Bretherton (JAMES 2017)" ;
  :author = "M. Ahlgrimm" ;
  :version = "Created on Wed Jan 11 20:24:24 2023" ;
  :format_version = "DEPHY SCM format version 1" ;
  :modifications = "" ;
  :script = "DEPHY-SCM/MAGIC/LEG04A/driver_SCM.py" ;
  :comment = "" ;
  :start_date = "2012-10-20 18:00:00" ;
  :end_date = "2012-10-25 05:00:00" ;
  :forcing_scale = -1 ;
  :adv_ta = 1 ;
  :adv_theta = 1 ;
  :adv_thetal = 1 ;
  :radiation = "on" ;
  :adv_qv = 1 ;
  :adv_qt = 1 ;
  :adv_rv = 1 ;
  :adv_rt = 1 ;
  :forc_wa = 1 ;
  :forc_wap = 0 ;
  :forc_geo = 1 ;
  :surface_type = "ocean" ;
  :surface_forcing_temp = "ts" ;
  :surface_forcing_moisture = "none" ;
  :surface_forcing_wind = "none" ;
}
```

5.2 Included Cases

Several cases are included in the repository to serve as examples for users to create their own and for basic research. All case configuration namelist files for included cases can be found in `ccpp-scm/scm/etc/case_config` and represent the following observational field campaigns:

- Tropical Warm Pool – International Cloud Experiment (TWP-ICE) maritime deep convection
- Atmospheric Radiation Measurement (ARM) Southern Great Plains (SGP) Summer 1997 continental deep convection
- Atlantic Stratocumulus Transition EXperiment (ASTEX) maritime stratocumulus-to-cumulus transition
- Barbados Oceanographic and Meteorological EXperiment (BOMEX) maritime shallow convection
- Large eddy simulation ARM Symbiotic Simulation and Observation (LASSO) for May 18, 2016 (with capability to run all LASSO dates - see 5.4) continental shallow convection

For the ARM SGP case, several case configuration files representing different time periods of the observational dataset are included, denoted by a trailing letter. The LASSO case may be run with different forcing applied, so three case configuration files corresponding to these different forcing are included. In addition, two example cases are included for using UFS Atmosphere initial conditions:

- UFS initial conditions for 38.1 N, 98.5 W (central Kansas) for 00Z on Oct. 3, 2016 with Noah variables on the C96 FV3 grid (`fv3_model_point_noah.nc`)
- UFS initial conditions for 38.1 N, 98.5 W (central Kansas) for 00Z on Oct. 3, 2016 with NoahMP variables on the C96 FV3 grid (`fv3_model_point_noahmp.nc`)

See 5.5 for information on how to generate these files for other locations and dates, given appropriate UFS Atmosphere initial conditions.

5.3 How to set up new cases

Setting up a new case involves preparing the two types of files listed above. For the case initialization and forcing data file, this typically involves writing a custom script or program to parse the data from its original format to the format that the SCM expects, listed above. An example of this type of script written in Python is included in `/ccpp-scm/scm/etc/scripts/twpice_forcing_file_generator.py`. The script reads in the data as supplied from its source, converts any necessary variables, and writes a NetCDF (version 4) file in the format described in subsections 5.1.2 and 5.1.3. For reference, the following formulas are used:

$$\theta_{il} = \theta - \frac{\theta}{T}\left(\frac{L_v}{c_p}q_l + \frac{L_s}{c_p}q_i\right) \tag{5.1}$$

$$q_t = q_v + q_l + q_i \tag{5.2}$$

where θ_{il} is the ice-liquid water potential temperature, θ is the potential temperature, L_v is the latent heat of vaporization, L_s is the latent heat of sublimation c_p is the specific heat capacity of air at constant pressure, T is absolute temperature, q_t is the total water specific humidity, q_v is the water vapor specific humidity, q_l is the suspended liquid water specific humidity, and q_i is the suspended ice water specific humidity.

As shown in the example NetCDF header, the SCM expects that the vertical dimension is pressure levels (index 1 is the surface) and the time dimension is in seconds. The initial conditions expected are the height of the pressure levels in meters, and arrays representing vertical columns of θ_{il} in K, q_t, q_l, and q_i in kg kg^{-1}, u and v in m s^{-1}, turbulence kinetic energy in m^2 s^{-2} and ozone mass mixing ratio in kg kg^{-1}.

For forcing data, the SCM expects a time series of the following variables: latitude and longitude in decimal degrees [in case the column(s) is moving in time (e.g., Lagrangian column)], the surface pressure (Pa) and surface temperature (K). If surface fluxes are specified for the new case, one must also include a time series of the kinematic surface sensible heat flux (K m s^{-1}) and kinematic surface latent heat flux (kg kg^{-1} m s^{-1}).

The following variables are expected as 2-dimensional arrays (vertical levels first, time second): the geostrophic u (E-W) and v (N-S) winds (m s^{-1}), and the horizontal and vertical advective tendencies of θ_{il} (K s^{-1}) and q_t (kg kg^{-1} s^{-1}), the large scale vertical velocity (m s^{-1}), large scale pressure vertical velocity (Pa s^{-1}), the prescribed radiative heating rate (K s^{-1}), and profiles of u, v, T, θ_{il} and q_t to use for nudging.

Although it is expected that all variables are in the NetCDF file, only those that are used with the chosen forcing method are required to be nonzero. For example, the following variables are required depending on the values of `mom_forcing_type` and `thermo_forcing_type` specified in the case configuration file:

- `mom_forcing_type` = `1`
  - Not implemented yet
- `mom_forcing_type` = `2`
  - geostrophic winds and large scale vertical velocity
- `mom_forcing_type` = `3`
  - u and v nudging profiles
- `thermo_forcing_type` = `1`
  - horizontal and vertical advective tendencies of θ_{il} and q_t and prescribed radiative heating (can be zero if radiation scheme is active)
- `thermo_forcing_type` = `2`
  - horizontal advective tendencies of θ_{il} and q_t, prescribed radiative heating (can be zero if radiation scheme is active), and the large scale vertical pressure velocity
- `thermo_forcing_type` = `3`
  - θ_{il} and q_t nudging profiles and the large scale vertical pressure velocity

For the case configuration file, it is most efficient to copy an existing file in `ccpp-scm/scm/etc/case_config` and edit it to suit one's case. Recall from subsection 5.1.1 that this file is used to configure the SCM framework parameters for a given case. Be sure to check that model timing parameters such as the time step and output frequency are appropriate for the physics suite being used. There is likely some stability criterion that governs the maximum time step based on the chosen parameterizations and number of vertical levels (grid spacing). The `case_name` parameter should match the name of the case input data file that was configured for the case (without the file extension). The `runtime` parameter should be less than or equal to the length of the forcing data unless the desired behavior of the simulation is to proceed with the last specified forcing values after the length of the forcing data has been surpassed. The initial date and time should fall within the forcing period specified in the case input data file. If the case input data is specified to a lower altitude than the vertical domain, the remainder of the column will be filled in with values from a reference profile. There is a tropical profile and mid-latitude summer profile provided, although one may add more choices by adding a data file to `ccpp-scm/scm/data/processed_case_input` and adding a parser section to the subroutine `get_reference_profile` in `-scm/scm/src/scm_input.f90`. Surface fluxes can either be specified in the case input data file or calculated using a surface scheme using surface properties. If surface fluxes are specified from data, set `sfc_flux_spec` to `.true.` and specify `sfc_roughness_length_cm` for the surface over which the column resides. Otherwise, specify a `sfc_type`. In addition, one must specify a `column_area` for

each column.

To control the forcing method, one must choose how the momentum and scalar variable forcing are applied. The three methods of Randall and Cripe (1999, JGR) have been implemented: "revealed forcing" where total (horizontal + vertical) advective tendencies are applied (type 1), "horizontal advective forcing" where horizontal advective tendencies are applied and vertical advective tendencies are calculated from a prescribed vertical velocity and the calculated (modeled) profiles (type 2), and "relaxation forcing" where nudging to observed profiles replaces horizontal advective forcing combined with vertical advective forcing from prescribed vertical velocity (type 3). If relaxation forcing is chosen, a `relax_time` that represents the timescale over which the profile would return to the nudging profiles must be specified.

5.4 Using other LASSO cases

In order to use other LASSO cases than the one provided, perform the following steps:

1. Access http://archive.arm.gov/lassobrowser and use the navigation on the left to choose the dates for which you would like to run a SCM simulation. Pay attention to the "Large Scale Forcing" tab where you can choose how the large scale forcing was generated, with options for ECMWF, MSDA, and VARANAL. All are potentially valid, and it is likely worth exploring the differences among forcing methods. Click on Submit to view a list of simulations for the selected criteria. Choose from the simulations (higher skill scores are preferred) and check the "Config Obs Model Tar" box to download the data. Once the desired simulations have been checked, order the data (you may need to create an ARM account to do so).
2. Once the data is downloaded, decompress it. From the `config` directory, copy the files `input_ls_forcing.nc`, `input_sfc_forcing.nc`, and `wrfinput_d01.nc` into their own directory under `ccpp-scm/scm/data/raw_case_input/`.
3. Modify `ccpp-scm/scm/etc/scripts/lasso1_forcing_file_generator_gjf.py` to point to the input files listed above. Execute the script in order to generate a case input file for the SCM (to be put in `ccpp-scm/scm/data/processed_case_input/`):

   ```
   ./lasso1_forcing_file_generator_gjf.py
   ```

4. Create a new case configuration file (or copy and modify an existing one) in `ccpp-scm/scm/etc/case_config`. Be sure that the `case_name` variable points to the newly created/processed case input file from above.

5.5 Using UFS Output to Create SCM Cases: UFS-Replay

5.5.1 UFS_IC_generator.py

A script exists in `scm/etc/scripts/UFS_IC_generator.py` to read in UFS history (output) files and their initial conditions to generate a SCM case input data file, in DEPHY format. Note that the script requires a few python packages that may not be found by default in all python installations: **`argparse`**, **`fnmatch`**, **`logging`**, **`NetCDF4`**, **`numpy`**, **`shapely`**, **`f90nml`**, and **`re`**.

```
./UFS_IC_generator.py [-h] (-l LOCATION LOCATION | -ij INDEX INDEX) -d
DATE -i IN_DIR -g GRID_DIR -f FORCING_DIR -n
CASE_NAME [-t {1,2,3,4,5,6,7}] [-a AREA] [-oc]
[-lam] [-sc] [-near]
```

Mandatory arguments:

1. **`--location (-l)`** OR **`--index (-ij)`**: Either longitude and latitude in decimal degrees east and north of a location OR the UFS grid index with the tile number
   - -l 261.51 38.2 (two floating point values separated by a space)
   - -ij 8 49 (two integer values separated by a space; this option must also use the **`--tile (-t)`** argument to specify the tile number)
2. **`--date (-d)`** YYYYMMDDHHMMSS: date corresponding to the UFS initial conditions
3. **`--in_dir (-i)`**: path to the directory containing the UFS initial conditions
4. **`--grid_dir (-g)`**: path to the directory containing the UFS supergrid files (AKA "fix" directory)
5. **`--forcing_dir (-f)`**: path to the directory containing the UFS history files
6. **`--case_name (-n)`**: name of case

Optional arguments:

1. **`--tile (-t)`**: if one already knows the correct tile for the given longitude and latitude OR one is specifying the UFS grid index (**`--index`** argument)
2. **`--area (-a)`**: area of grid cell in m^2 (if known or different than the value calculated from the supergrid file)
3. **`--old_chgres (-oc)`**: flag if UFS initial conditions were generated using older version of chgres (global_chgres); might be the case for pre-2018 data
4. **`--lam (-lam)`**: flag to signal that the ICs and forcing is from a limited-area model run
5. **`--save_comp (-sc)`**: flag to create UFS reference file for comparison
6. **`--use_nearest (-near)`**: flag to indicate using the nearest UFS history file gridpoint

5.5.2 UFS_forcing_ensemble_generator.py

There is an additional script in `scm/etc/scripts/UFS_forcing_ensemble_generator.py` to create UFS-replay case(s) starting with output from UFS Weather Model (UWM) Regression Tests (RTs).

```
UFS_forcing_ensemble_generator.py [-h] -d DIR -n CASE_NAME
(-lonl LON_1 LON_2 -latl LAT_1 LAT_2 -nens NENSMEMBERS |
-lons [LON_LIST] -lats [LAT_LIST])
[-dt TIMESTEP] [-cres C_RES] [-sdf SUITE] [-sc] [-near]
```

Mandatory arguments:

1. **`--dir (-d)`**: path to UFS Regression Test output
2. **`--case_name (-n)`**: name of cases
3. Either: (see examples bewlow)
   - **`--lon_limits (-lonl)`** AND **`--lat_limits (-latl)`** AND **`--nensmembers (-nens)`**: longitude range, latitude range, and number of cases to create
   - **`--lon_list (-lons)`** AND **`--lat_list (-lats)`**: longitude and latitude of cases

Optional arguments:

1. **`--timestep (-dt)`**: SCM timestep, in seconds
2. **`--C_res (-cres)`**: UFS spatial resolution
3. **`--suite (-sdf)`**: CCPP suite definition file to use for ensemble
4. **`--save_comp (-sc)`**: flag to create UFS reference file for comparison
5. **`--use_nearest (-near)`**: flag to indicate using the nearest UFS history file gridpoint

Examples to run from within the **`scm/etc/scripts`** directory to create SCM cases starting with the output from a UFS Weather Model regression test(s):

On the supported platforms Cheyenne (NCAR) and Hera (NOAA), there are staged UWM RTs located at:

- `Cheyenne /glade/scratch/epicufsrt/GMTB/CCPP-SCM/UFS_RTs`
- `Hera /scratch1/BMC/gmtb/CCPP-SCM/UFS_RTs`

5.5.3 Example 1: UFS-replay for single point

UFS regression test, `control_c192`, for single point.

```
./UFS_forcing_ensemble_generator.py -d /glade/scratch/epicufsrt/GMTB/
   CCPP-SCM/UFS_RTs/control_c192/ -sc --C_RES 192 -dt 360  -n
   control_c192 -lons 300 -lats 34
```

Upon successful completion of the script, the command to run the case(s) will print to the screen. For example,

```
./run_scm.py --npz_type gfs --file scm_ufsens_control_c192.py --
    timestep 360
```

The file **scm_ufsens_control_c192.py** is created in **ccpp-scm/scm/bin/**, where the SCM run script is to be exectued.

5.5.4 Example 2: UFS-replay for list of points

UFS regression test, `control_c384`, for multiple points.

```
./UFS_forcing_ensemble_generator.py -d /glade/scratch/epicufsrt/GMTB/
    CCPP-SCM/UFS_RTs/control_c384/ -sc --C_RES 384 -dt 225 -n
    control_c384 -lons 300 300 300 300 -lats 34 35 35 37
```

Upon successful completion of the script, the command to run the case(s) will print to the screen. For example,

```
./run_scm.py --npz_type gfs --file scm_ufsens_control_c384.py --
    timestep 225
```

The file **scm_ufsens_control_c384.py** contains **ALL** of the cases created. Each case created will have the naming convention **case_name_nXXX**, where the suffix **XXX** is the case number from 0 to the number of points provided. The contents of the file should look like:

```
run_list = [{"case": "control_c384_n000", "suite": "SCM_GFS_v16"},
            {"case": "control_c384_n001", "suite": "SCM_GFS_v16"},
            {"case": "control_c384_n002", "suite": "SCM_GFS_v16"},
            {"case": "control_c384_n003", "suite": "SCM_GFS_v16"}]
```

5.5.5 Example 3: UFS-replay for an ensemble of points

UFS regression test, `control_p8`, for an ensemble (10) of randomly selected points over a specified longitude ($300 - 320^oW$) and latitude ($40 - 50^oN$) range

But first, to use the `control_p8` test we need to rerun the regression test to generate UFS history files with a denser and constant output interval. First, in `control_p8/model_configure`, change **`--output_fh`** to `"interval -1"`, where `interval` is the UFS history file output frequency (in hours), see UFS Weather Model Users Guide for more details.

For the purposes of this example the `control_p8` test has already been rerun, but if starting from your own UWM RTs, you can rerun the UWM regression test, on Cheyenne for example, by running the following command in the RT directory: `qsub job_card`

Now the cases can be generated with the following command:

```
./UFS_forcing_ensemble_generator.py -d /glade/scratch/epicufsrt/GMTB/
   CCPP-SCM/UFS_RTs/control_p8/ -sc --C_RES 96 -dt 720 -n control_p8 -
   lonl 300 320 -latl 40 50 -nens 10 -sdf SCM_GFS_v17_p8
```

Upon successful completion of the script, the command to run the case(s) will print to the screen. For example,

```
./run_scm.py --npz_type gfs --file scm_ufsens_control_p8.py --timestep
   720
```

The file **scm_ufsens_control_p8.py** contains ten cases (n000-n009) to be run. The contents of the file should look like:

```
run_list = [{"case": "control_p8_n000", "suite": "SCM_GFS_v17_p8"},
            {"case": "control_p8_n001", "suite": "SCM_GFS_v17_p8"},
            {"case": "control_p8_n002", "suite": "SCM_GFS_v17_p8"},
            {"case": "control_p8_n003", "suite": "SCM_GFS_v17_p8"},
            {"case": "control_p8_n004", "suite": "SCM_GFS_v17_p8"},
            {"case": "control_p8_n005", "suite": "SCM_GFS_v17_p8"},
            {"case": "control_p8_n006", "suite": "SCM_GFS_v17_p8"},
            {"case": "control_p8_n007", "suite": "SCM_GFS_v17_p8"},
            {"case": "control_p8_n008", "suite": "SCM_GFS_v17_p8"},
            {"case": "control_p8_n009", "suite": "SCM_GFS_v17_p8"}]
```

6 CCPP Interface

Chapter 6 of the CCPP v6 Technical Documentation (https://ccpp-techdoc.readthedocs.io/en/v6.0.0/) provides a wealth of information on the overall process of connecting a host model to the CCPP framework for calling physics. This chapter describes the particular implementation within this SCM, including how to set up, initialize, call, and change a physics suite using the CCPP framework.

6.1 Setting up a suite

Setting up a physics suite for use in the CCPP SCM with the CCPP framework involves three steps: preparing data to be made available to physics through the CCPP, running the `ccpp_prebuild.py` script to reconcile SCM-provided variables with physics-required variables, and preparing a suite definition file.

6.1.1 Preparing data from the SCM

As described in sections 6.1 and 6.2 of the CCPP Technical Documentation a host model must allocate memory and provide metadata for variables that are passed into and out of the schemes within the physics suite. As of this release, in practice this means that a host model must do this for all variables needed by all physics schemes that are expected to be used with the host model. For this SCM, all variables needed by the physics schemes are allocated and documented in the file `ccpp-scm/scm/src/scm_type_defs.f90` and are contained within the `physics` derived data type. This derived data type initializes its component variables in a `create` type-bound procedure. As mentioned in section 6.2 of the CCPP Technical Documentation, files containing all required metadata was constructed for describing all variables in the `physics` derived data type. These files are `scm/src/GFS_typedefs.meta`, `scm/src/CCPP_typedefs.meta` and `scm_physical_constants.meta`. Further, `scm_type_defs.meta` exists to provide metadata for derived data type definitions and their instances, which is needed by the ccpp-framework to properly reference the data. The standard names of all variables in this table must match with a corresponding variable within one or more of the physics schemes. A list of all standard names used can be found in `ccpp/framework/doc/DevelopersGuide/CCPP_VARIABLES_SCM.pdf`.

6.1.2 Editing and running `ccpp_prebuild.py`

General instructions for configuring and running the `ccpp_prebuild.py` script can be found in chapter 8 of the CCPP Technical Documentation. The script expects to be run with a host-model-dependent configuration file, passed as argument `-config=path_to_config_file`. Within this configuration file are variables that hold paths to the variable definition files (where metadata tables can be found on the host model side), the scheme files (a list of paths to all source files containing scheme entry points), the auto-generated physics schemes makefile snippet, the auto-generated physics scheme caps makefile snippet, and the directory where the auto-generated physics caps should be written out to. As mentioned in section 2.3, this script must be run to reconcile data provided by the SCM with data required by the physics schemes before compilation – this is done automatically by `cmake`.

6.1.3 Preparing a suite definition file

The suite definition file is a text file read by the model at compile time. It is used to specify the physical parameterization suite, and includes information about the number of parameterization groupings, which parameterizations that are part of each of the groups, the order in which the parameterizations should be run, and whether subcycling will be used to run any of the parameterizations with shorter timesteps.

In addition to the six or so major parameterization categories (such as radiation, boundary layer, deep convection, resolved moist physics, etc.), the suite definition file can also have an arbitrary number of additional interstitial schemes in between the parameterizations to prepare or postprocess data. In many models, this interstitial code is not known to the model user but with the suite definition file, both the physical parameterizations and the interstitial processing are listed explicitly.

For this release, supported suite definition files used with this SCM are found in `ccpp-scm/ccpp/suites` and have default namelist, tracer configuration, and timesteps attached in `ccpp-scm/scm/src/suite_info.py`. For all of these suites, the physics schemes have been organized into 3 groupings following how the physics are called in the UFS Atmosphere model, although no code is executed in the SCM time loop between execution of the grouped schemes. Several "interstitial" schemes are included in the suite definition file to execute code that previously was part of a hard-coded physics driver. Some of these schemes may eventually be rolled into the schemes themselves, improving portability.

6.2 Initializing/running a suite

The process for initializing and running a suite in this SCM is described in sections 4.4 and 4.5, respectively. A more general description of the process for performing suite

initialization and running can also be found in sections 6.4 and 6.5 of the CCPP Technical Documentation.

6.3 Changing a suite

6.3.1 Replacing a scheme with another

Prior to being able to swap a scheme within a suite, one must first add a CCPP-compliant scheme to the pool of available schemes in the CCPP physics repository. This process is described in chapter 2 of the CCPP Technical Documentation.

Once a CCPP-compliant scheme has been added to the CCPP physics repository, the process for modifying an existing suite should take the following steps into account:

- Examine and compare the arguments of the scheme being replaced and the replacement scheme.
  - Are there any new variables that the replacement scheme needs from the host application? If so, these new variables must be added to the host model cap. For the SCM, this involves adding a component variable to the `physics` derived data type and a corresponding entry in the metadata table. The new variables must also be allocated and initialized in the `physics%create` type-bound procedure.
  - Do any of the new variables need to be calculated in an interstitial scheme? If so, one must be written and made CCPP-compliant itself. The CCPP Technical Documentation will help in this endeavor, and the process outlined in its chapter 2 should be followed.
  - Do other schemes in the suite rely on output variables from the scheme being replaced that are no longer being supplied by the replacement scheme? Do these output variables need to be derived/calculated in an interstitial scheme? If so, see the previous bullet about adding one.
- Examine existing interstitial schemes related to the scheme being replaced.
  - There may be scheme-specific interstitial schemes (needed for one specific scheme) and/or type-generic interstitial schemes (those that are called for all schemes of a given type, i.e. all PBL schemes). Does one need to write analogous scheme-specific interstitial schemes for the replacement?
  - Are the type-generic interstitial schemes relevant or do they need to be modified?
- Depending on the answers to the above considerations, edit the suite definition file as necessary. Typically, this would involve finding the `<scheme>` elements associated with the scheme to be replaced and its associated interstitial `<scheme>` elements and simply replacing the scheme names to reflect their replacements. See chapter 4 of the CCPP Technical Documentation for further details.

6.3.2 Modifying "groups" of parameterizations

The concept of grouping physics in the suite definition file (currently reflected in the `<group name="XYZ">` elements) enables "groups" of parameterizations to be called with other computation (perhaps related to the dycore, I/O, etc.) in between. In the suite definition file included in this release, three groups are specified, but currently no computation happens between `ccpp_physics_run` calls for these groups. However, one can edit the groups to suit the needs of the host application. For example, if a subset of physics schemes needs to be more tightly connected with the dynamics and called more frequently, one could create a group consisting of that subset and place a `ccpp_physics_run` call in the appropriate place in the host application. The remainder of the parameterizations groups could be called using `ccpp_physics_run` calls in a different part of the host application code.

6.3.3 Subcycling parameterizations

The suite definition file allows subcycling of schemes, or calling a subset of schemes at a smaller time step than others. The `<subcycle loop = n>` element in the suite definition file controls this function. All schemes within such an element are called `n` times during one `ccpp_physics_run` call. An example of this is found in the `suite_SCM_GFS_v16.xml` suite definition file, where the surface schemes are executed twice for each timestep (implementing a predictor/corrector paradigm). Note that no time step information is included in the suite definition file. **If subcycling is used for a set of parameterizations, the smaller time step must be an input argument for those schemes. This is not handled automatically by the ccpp-framework yet.**

6.4 Adding variables

6.4.1 Adding a physics-only variable

Suppose that one wants to add the variable `foo` to a scheme that spans the depth of the column and that this variable is internal to physics, not part of the SCM state or subject to external forcing. Here is how one would do so:

1. First, add the new variable to the `physics` derived data type definition in `ccpp-scm/scm/src/scm_type_defs.f90`. Within the definition, you'll notice that there are nested derived data types (which contain most of the variables needed by the physics and are used for mainly legacy reasons) and several other integers/reals/logicals. One could add the new variable to one of the nested GFS derived data types if the variable neatly fits inside one of them, but it is suggested to bypass the GFS derived data types and add a variable directly to the `physics` type definition:

   ```
   real(kind=kind_phys), allocatable :: foo(:,:)
   ```

2. Second, within the `physics_create` subroutine, add an allocate and initialization statement.

   ```
   allocate(foo(n_columns, n_levels))
   physics%foo = 0.0
   ```

   Note that even though foo only needs to have the vertical dimension, it is also allocated with the `n_columns` dimension as the first dimension since this model is intended to be used with multiple independent columns. Also, the initialization in this creation subroutine can be overwritten by an initialization subroutine associated with a particular scheme.
3. At this point, these changes are enough to allocate the new variable (`physics%create` is called in the main subroutine of `scm.F90`), although this variable cannot be used in a physics scheme yet. For that, you'll need to add an entry in the corresponding metadata file. See section 2.2 of the CCPP Technical Documentation for more information regarding the format.
4. On the physics scheme side, there will also be a metadata file entry for **`foo`**. For example, say that scheme **`bar`** uses **`foo`**. If **`foo`** is further initialized in **`bar`**'s `_init` subroutine, a metadata entry for **`foo`** must be found in the corresponding section in the metadata file. If it is used in **`bar`**'s run subroutine, a metadata entry for foo must also appear in the metadata file section for `bar_run`. The metadata entry on the physics scheme side has the same format as the one on the host model side described above. The standard name, rank, type, and kind must match the entry from the host model table. Others attributes (local name, units (assuming that an automatic conversion exists in the ccpp-framework), long_name, intent) can differ. The local name corresponds to the name of the variable used within the scheme subroutine, and the intent attribute should reflect how the variable is actually used within the scheme.
   Note: In addition to the metadata file, the argument list for the scheme subroutine must include the new variable (i.e., **`foo`** must actually be in the argument list for `bar_run` and be declared appropriately in regular Fortran).

If a variable is declared following these steps, it can be used in any CCPP-compliant physics scheme and it will retain its value from timestep to timestep. A variable will ONLY be zeroed out (either every timestep or periodically) if it is in the `GFS_interstitial` or `GFS_diag` data types. So, if one needs the new variable to be 'prognostic', one would need to handle updating its value within the scheme, something like:

$$\text{foo}^{t+1} = \text{foo}^{t} + \Delta t * \text{foo\_tendency} \tag{6.1}$$

Technically, the host model can "see" foo between calls to physics (since the host model allocated its memory at initialization), but it will not be touching it.

6.4.2 Adding a prognostic SCM variable

The following instructions are valid for adding a passive, prognostic tracer to the SCM. Throughout these instructions, the new tracer is called 'smoke'.

1. Add a new tracer to the SCM state. In `ccpp-scm/scm/src/scm_type_defs.f90` do the following:
   - Add an index for the new tracer in the `scm_state_type` definition.
   - Do the following in the `scm_state_create` subroutine:
     - Increment `scm_state%n_tracers`
     - Set `scm_state%smoke_index` = **`(next available integer)`**
     - Set `scm_state%tracer_names(scm_state%smoke_index)` = **`'smoke'`**
     - Note: `scm_state%state_tracer` is initialized to zero in this subroutine already, so there is no need to do so again.
2. Initialize the new tracer to something other than zero (from an input file).
   - Edit an existing input file (in `ccpp-scm/scm/data/processed_case_input`): add a field in the 'initial' group of the NetCDF file(s) (with vertical dimension in pressure coordinates) with an appropriate name in one (or all) of the input NetCDF files and populate with whatever values are necessary to initialize the new tracer.
   - Create a new input variable to read in the initialized values. In `ccpp-scm/scm/src/scm_type_defs.f90`:
     - Add a new input variable in `scm_input_type`

       ```
       real(kind=dp), allocatable              :: input_smoke(:)
       ```

     - In `scm_input_create`, allocate and initialize the new variable to 0.
   - Read in the input values to initialize the new tracer. In `ccpp-scm/scm/src/scm_input.f90/get_case_init`:
     - Add a variable under the initial profile section:

       ```
       real(kind=dp), allocatable  :: input_smoke(:) !< smoke
          profile (fraction)
       ```

     - Add the new input variable to the allocate statement.
     - Read the values in from the file:

       ```
       call check(NF90_INQ_VARID(grp_ncid,"smoke",varID))
       call check(NF90_GET_VAR(grp_ncid,varID,input_smoke))
       ```

     - set `scm_input%input_smoke` = **`input_smoke`**
   - Interpolate the input values to the model grid. Edit `scm_setup.f90/set_state`:
     - Add a loop over the columns to call `interpolate_to_grid_centers` that puts **`input_smoke`** on grid levels in `scm_state%state_tracer`

       ```
       do i=1, scm_state%n_cols
              call interpolate_to_grid_centers(scm_input%
                 input_nlev, scm_input%input_pres, scm_input%
                 input_smoke, scm_state%pres_l(i,1,:), &
                 scm_state%n_levels, scm_state%state_tracer(i
                    ,1,:,scm_state%smoke_index,1),
                    last_index_init, 1)
       end do
       ```

   - At this point, you have a new tracer initialized to values specified in the input file on the model vertical grid, but it is not connected to any physics or changed by any forcing.
3. For these instructions, we'll assume that the tracer is not subject to any external forcing (e.g., horizontal advective forcing, sources, sinks). If it is, further work is required to:

- One needs to provide data on how tracer is forced in the input file, similar to specifying its initial state, as above.
- Create, allocate, and read in the new variable for forcing (similar to above).
- Add to `interpolate_forcing` (similar to above, but interpolates the forcing to the model grid and model time).
- Add statements to time loop to handle the first time step and time-advancing.
- Edit `apply_forcing_forward_Euler` in `ccpp-scm/scm/src/scm_forcing.f90`.

4. In order to connect the new tracer to the CCPP physics, perform steps 1-4 in section 6.4.1 for adding a physics variable. In addition, do the following in order to associate the `scm_state` variable with variables used in the physics through a pointer:
   - Point the new physics variable to `scm_state%state_tracer(:,:,:,scm_state%smoke_index)` in `ccpp-scm/scm/src/scm_type_defs.f90/physics_associate`.
5. There may be additional steps depending on how the tracer is used in the physics and how the physics scheme is integrated with the current GFS physics suite. For example, the GFS physics has two tracer arrays, one for holding tracer values before the physics timestep (`ccpp-scm/scm/src/GFS_typedefs.F90/GFS_statein_type/qgrs`) and one for holding tracer values that are updated during/after the physics (`ccpp-scm/scm/src/GFS_typedefs.F90/GFS_stateout_type/gq0`). If the tracer needs to be part of these arrays, there are a few additional steps to take. If you need help, please post on the support forum at: https://dtcenter.org/forum/ccpp-user-support/ccpp-single-column-model.