Kamodo Onboarding Instructions Mar 7, 2022

Rebecca Ringuette (Editors: Lutz Rastaetter, and Darren De Zeeuw)

Section 1: Introduction

Kamodo provides a fast, intuitive interface for functional programming, memory intensive visualization, and unit conversions. At CCMC, we are building several tools based on Kamodo to provide additional services to our users, including a satellite flythrough tool, a data reconstruction tool, and a line-of-sight calculation tool. We will develop additional Kamodo-based tools as community interest and time allow. Installation instructions for Kamodo on your personal computer are given on the Github repository (https://github.com/nasa/Kamodo) in a file named "*Kamodo_CCMC_InstallationInstructions.md*" in the top directory. Since you will be contributing a new model reader to Kamodo, the last line of the installation instructions should be altered to "*python -m pip install -e ./Kamodo*" to install Kamodo in development mode. All other commands should be used as given. Examples of how to use the tools built on top of Kamodo are in the notebooks directory on GitHub. The remainder of this document focuses on what is needed for your model to be incorporated into these tools.

For your new model to be accessible through these tools, a 'model reader' specific to the model's output data format must be written in python in the format described in these instructions. Section 2 describes the necessary background details, such as filename conventions and variable definitions, followed by a detailed description of the necessary reader components in Section 3. Instructions on what code to add to the flythrough are given in Section 4. Section 5 is the summary, which includes our contact information, and is followed by a description of a few example model readers in Section 6. Snippets of code in this document are indicated with white text on a black blackground. For ease of access, a high-level video tour of a simple model reader is available at https://www.youtube.com/watch?v=nvl61pklEuU.

Section 2: Documentation and Format Requirements

In this section, we outline the basic format and metadata requirements needed to support the reader described in the next section. We expect modelers to generate a short document based on the requirements presented in this section and submit it to the Kamodo team prior to model submission. Portions of that document will also likely be necessary for other CCMC tasks and may be shared with other CCMC teams for collaboration.

Naming and formatting conventions:

Model output filenames must contain the full date in UTC corresponding to the data in the file, including the year, and be the same format for each new run (e.g. *abcd_YYYYMMDD.nc* or similar). This UTC time value should also be included in each model output file's content with the corresponding attribute name indicated in documentation. If the model outputs data into one file per timestep, then modelers should also include the hours, minutes, and seconds as necessary in the filename to completely capture the corresponding time in the file name. For example, if the model can at most produce output every minute, then the seconds information is not necessary, but the filename convention should still be identical between runs. In cases where some variables are stored in different files covering the same time interval, the filenames should contain the same time string but different prefixes (e.g. *2D_YYYYMMDD.nc* and *3D_YYYYMMDD.nc*, which would hold all 2D and 3D variable data separately for

the same time interval). If outputs for a given day are produced in multiple files, including outputs with more than one file per time value, then a file listing should arrange files of the same content in correct temporal order when listed in lexical or numerical order (e.g. no *output.9.dat* followed by an *output.10.dat* file). Modelers will need to include a description of the filename convention in the documentation to be submitted to CCMC.

Although we do not request any particular data format convention, modelers should consider the data output format carefully. Writing the model reader described in the next section is easier and faster for data stored in a netCDF4 file or similar formats. However, this does not exclude other file types. Modelers planning to output their data to a different file type must provide a script separate from the reader to read the data into the programming language they plan to perform the interpolation in (typically python). The names of all scripts must be included in documentation, along with a brief description of the script's purpose (e.g. model reader or file reader). Model reader script names typically begin with the model's name in all lower case letters (no version indicators), followed by an underscore and any descriptive indicators (e.g. '*4D*' for time+spatial interpolation, 'cdf' to indicate a conversation to netCDF4, or 'int' for a custom interpolator script). Any accompanying scripts should be similarly named and described in the documentation (e.g. for file conversion or custom interpolation functions).

Variable metadata:
We also need a set of information for each quantity or variable calculated by the model and included in the model output, apart from the metadata required for the CCMC metadata registry described elsewhere. In the documentation, modelers will associate each quantity with their choice from a list of standardized variable names and include a description of the variable. Variables in the output data without a mapping to a standardized variable will not be accepted. However, if none of the options are suitable for a particular quantity, then please contact us to discuss other possible names. This does not place any restriction on what modelers call the quantity in their code or model output, but rather places the responsibility upon the modeler to associate their variable names with commonly accepted variable names for better representation in Kamodo.

In addition to the variable name mapping, we also need the correct units for each variable as represented in the data, and a list of the coordinate grids each variable is dependent upon. We recommend a table, such as the one presented in Table 1, to communicate this information efficiently. The first column should include the name of the variable in the output data, followed by the associated standardized variable name in the second column and the units of the variable in the output data in the third column. The fourth column should contain the string '*CAR*' or '*SPH*' to indicate whether the coordinate system is cartesian or spherical, and the string representing the coordinate system (from SpacePy's coordinate module). If the coordinate system upon which the variable depends upon is intrinsic to the model, such as dependent upon an internal magnetic field, then a method should be offered to convert from the given coordinate system to the coordinate system internal to the model or the data should be interpolated to a gridded coordinate system already defined in SpacePy. (See the *SF_Traj_Coords_Plots.ipynb* file in the notebook directory on Github for more details.) We note the choice of coordinate system also implies the coordinate units (e.g. *['hr', 'deg', 'deg', 'km']*' for the spherical GDZ coordinate system) and any unit conversions necessary should be performed in the model

reader. The last two columns should give the list of coordinates that variable depends on, and whether the coordinate grid varies in time.

Coordinate Dependencies:
The order of the coordinates in the table should match either the (time, x, y, z) convention or the (time, longitude, latitude, height) convention. If another convention is needed, please contact us to discuss this. Modelers are responsible for the accuracy of the information in the table and will be asked for clarification if needed. If there is more than one grid of a dimension type, such as pressure level, then those grids must be distinguished from each other in the same table (e.g. press. lev. 1 for a secondary pressure level grid). Modelers should also take care to differentiate between similar coordinates, such as cartesian geographical coordinates and other cartesian coordinate systems, for instance, in these associations. We also recommend that modelers choose either cartesian or spherical coordinates for all the variable data, and preferably in the same reference frame (e.g. GSE or GSM, not both) to simplify the reader script and its interaction with higher software layers. This does not preclude coordinate grids in the same system with different values, such as when models use center, left, and right sided cartesian coordinates. As shown in the last column of the table, modelers should also indicate whether the spatial grid for each variable changes in time. This does not refer to a grid with non-uniform spacing, or grids that are different in different model runs, but to a grid that changes from one time-step to another.

Table 1: Example of Variable Information

Variable name in output	Standard variable name	Description	Variable units in data	Coord. Sys.	Coordinate grids	Time dependent spatial grid?
Electron_temp	T_e	Electron temperature	K	GDZ, SPH	time, latitude, longitude, height	No
Total_mass_density	rho_Total	Total mass density	g/cm**3	GSE, CAR	time, x, y, z	No
Numberdensity_molecO	N_O2	Number density of molecular oxygen	1/kg**3	GEO_plev, SPH	time, latitude, longitude, press. lev.	No

Some models provide variable data depending upon a custom coordinate system (e.g. pressure level). In these cases, it is imperative for the model data output to include a variable allowing the Kamodo-based tools to convert between the two coordinate conventions for each custom coordinate, such as a variable called height that depends on pressure level. If the SpacePy conversions are not considered sufficient or do not apply to the situation, then modelers should provide an acceptable coordinate conversion method and clearly note this preference in the documentation. The name of this conversion function in the model reader script has a special role in higher layers, and so requires collaboration between the Kamodo team and the modelers.

Section 3: Model Reader Description
A given model reader consists of the same progression of code blocks: the *model_varnames* dictionary, a call to the file converter script (if needed), calculation and storage of various time information, logic to perform interpolation between files, logic to determine which variable datasets to retrieve, and logic to

functionalize the chosen variable datasets, which includes the definition of an interpolator for each variable dataset. These code blocks are described generally in the following subsections. The requirements described for each code block in the following subsections were determined to ensure all model readers function in the same way (e.g. a model-agnostic capability). We recommend the developer begin the model reader development by copying the various blocks of code from the current model reader scripts. Although we must require certain capabilities for our users, we recognize that each model reader must address the situations inherent in that model data. Some model readers contain additional code blocks to address such special cases, which we expect to also be useful to the developer (see Section 5). If a particular special case is needed but not described in this section, consult the descriptions in Section 5 to find the best match and follow the code example indicated.

The model_varnames dictionary:

A dictionary called '*model_varnames'* must be placed at the beginning of the script, which serves as a variable metadata directory and variable name translator for the model's output data. Each key of the dictionary must be a string identical to the name of each variable in the model data. The associated value should be a list with the following elements at the position indicated:

0. A string equal to the standardized representation of the variable name,
1. A string containing the variable description, which can be unique to the model,
2. An integer for that variable,
3. A string equal to the standardized name of the coordinate system the variable depends on (e.g. '*SM'* or '*GDZ*'),
4. A string equal to either '*car'* or '*sph'* (indicating either cartesian or spherical coordinates),
5. A list of strings equal to the coordinate names that variable depends on and in the agreed upon conventional order (e.g. *['time','lon','lat','height']*),
6. A string representing the proper representation of the units (e.g. '*erg/cm/s*' or '*kg/m**3'*).

It is customary, but not required, for each key value pair to be placed on its own line in the code (see Figure 1 for an example).

```
model_varnames = {'Ne':['N_e','electron number density',0,'SPH','sph',['time','lon','lat','radius'],'1/m**3'],
                  'Te':['T_e','electron temperature',1,'SPH','sph',['time','lon','lat','radius'],'K'],
                  'Ti':['T_i','ion temperature',2,'SPH','sph',['time','lon','lat','radius'],'K'],
```

Figure 1: Example of a *model_varnames* dictionary showing the first three key-value pairs. See text for details. The example shown is from https://github.com/nasa/Kamodo/blob/master/kamodo_ccmc/readers/iri_4D.py.

Class call signature and behavior:

The typical code needed to properly initialize a Kamodo class object is shown in Figure 2. The import statements needed vary between model readers and are shown here only as an example. The required components are the placement of the Kamodo sub-classed object inside a function, the class name and function name (*MODEL*), the placement of the from kamodo import Kamodo statement inside the function before the class definition, and the argument, keywords, and default values. The only argument in the class call signature must be the filename, called '*full_filename3d*' in this case, which must include the full path to the file in addition to the filename.

Each keyword governs different behavior of the class object necessary to interact with higher layers of software or to provide useful functionalities to the user. The *variables_requested* keyword accepts a list of standardized variable names that the user wants functionalized. By default, the *variables_requested* keyword is an empty list, which acts to functionalize all possible variables in the selected model data output. Modelers can exclude variables as desired by simply excluding them from the *model_varnames* dictionary. Logic must be included to print out relevant messages to the user if a variable requested is not recognized (e.g. not in the *model_varnames* dictionary) or if a variable requested is not in the given file (e.g. in the *model_varnames* dictionary but not in the data).

Several of the keywords are simplistic in use. The *verbose* keyword is False by default for quiet execution, but can be set to *True* for more detailed feedback. The *gridded_int* keyword is set to *True* by default to functionalize the variables on the given coordinate grid and on a gridded version of the same coordinate grid. This 'gridified' version of the functionalized data is useful for slicing through the data in one of the dependent coordinate grids (e.g. at a constant time and altitude)[1]. The *printfiles* keyword is set to *False* by default to reduce output. Users can set this to *True* to print the list of associated data files to the screen during execution. Otherwise, the list of files can be accessed afterwards by accessing the *filename* class attribute.

```
def MODEL():
    from kamodo import Kamodo
    from netCDF4 import Dataset
    from os.path import basename
    from numpy import array, transpose, NaN, unique, append, zeros, abs, diff, where
    from time import perf_counter
    from astropy.constants import R_earth
    from kamodo_ccmc.readers.reader_utilities import regdef_4D_interpolators, regdef_3D_interpolators

    class MODEL(Kamodo):
        '''IRI model data reader.'''
        def __init__(self, full_filename3d, variables_requested = [],
                     printfiles=False, filetime=False, gridded_int=True, fulltime=True,
                     verbose=False,**kwargs):
            super(MODEL, self).__init__(**kwargs)
```

Figure 2: The required structure, arguments, keywords and default values of the MODEL class object. The import statements vary between readers. See text for more details. The example shown is from the IRI model reader (https://github.com/nasa/Kamodo/blob/master/kamodo_ccmc/readers/iri_4D.py).

The *filetime* and *fulltime* keywords work together to accomplish a few goals, described here in the same order presented in Table 2. Given the default values (*filetime=False, fulltime=True*), the scripted behavior is to fully execute the reader code (*filetime=False*) and handle interpolation between files (*fulltime=True*) (Table 2, row 1). This capability requires the model reader to call itself recursively to (1) determine if there are any files with timestamps within the calculated time resolution of the beginning or end of the file (Table 2, row 2), and if so (2) retrieve the data from that file (Table 2, row 3), all while avoiding infinite recursive calls. Higher layers of code also require a method to quickly determine what time range each model reader instance covers, which must include the time between files if

[1]A non-gridded interpolator requires as input a list of position tuples (specifying positions in a 1-dimensional arrangement). A gridded interpolator accepts one or more coordinate positions to extract a lower dimensional slice or a hyperslab. Both kinds of interpolators return the variable's dataset when given no arguments. See the script referenced in the text and the *ModelReaderTutorial_TIEGCM_4D* notebook for more details and examples.

interpolation between the files is possible (Table 2 row 4). These requirements motivate the majority of the model reader code structure and are nearly identical between model readers.

Table 2: Behavior of *fulltime* and *filetime* keywords

filetime	***fulltime***	**Code Behavior**
False	True	(default) Fully execute the reader code and interpolate between files.
True	False	Returns the Kamodo class object with only the time-related attributes. Used in recursive calls for interpolation between files.
False	False	Returns the utc timestamp and all variable data for the beginning or end of the time range in the neighboring file. Used in recursive calls for interpolation between files.
True	True	Returns the Kamodo class object with only the time-related attributes *after* including the beginning or end of the time range from the neighboring file. Used in higher code layers.

Changing the values of both keywords from the default values to (*filetime=True, fulltime=False*) returns a Kamodo class object with only the time-related attributes called *filedate*, *datetimes*, *filetimes*, and *dt* Table 2, row 2). The *self.filedate* attribute is a datetime object corresponding to the model data time at midnight. The *self.datetimes* attribute is a list of two strings of format 'YYYY-MM-DD HH:MM:SS' indicating the start and end times of the model data in UTC. The *self.filetimes* attribute is a list containing the UTC timestamps corresponding to the two date-time strings in the *self.datetimes* attribute. Finally, the *dt* attribute is the time resolution of the data set in seconds.

Setting both keywords to *False* returns a *MODEL* object for the given file without adding time from a neighboring file (Table 2, row 3). In this case, the time related attributes previously mentioned and a *self.short_data* attribute are returned as attributes without functionalizing the data. The *short_data* attribute is a dictionary containing the variable data and units for each requested variable assigned to keys corresponding to the standardized variable names, with an additional key value pair for the timestamp of the needed time in UTC. This dictionary is used to append the time value and the corresponding variable data to the original *MODEL* object before functionalizing the data. Some model data outputs require this to be added to the end of the time range, while others need this at the beginning of the time range. The deciding factor of which is necessary for your particular model reader is simply which end of the time range best completes the dataset for the day (or hour). Finally, setting both the *filetime* and *fulltime* keywords to *True* returns a *MODEL* object with only the time attributes, but with the added time value from the neighboring file, if available (Table 2, row 4). Otherwise, the time attributes are returned unchanged.

There are multiple code blocks in which the beginning/ending time convention is affected. Using the IRI reader as the simplest case (as in the video tutorial), these code blocks begin on lines 83 (the *if fulltime:* statement), 167 (the *if not fulltime:* statement), 175 (the *if filecheck:* statement), and 200 (the *for varname in varlist:* statement). Comparing these blocks to the similar blocks in other readers reveals that the differences between the two conventions reduces to a choice of either zeros or ones (or -1s) and either an *append* or *insert* statement. We recommend the developer simply copy the relevant code blocks from a model reader with the same beginning/ending time convention into their own, paying detailed attention to where these code blocks are placed relative to others (see section 6 for a guide to examples in a selection of model readers).

Secondary file conversion scripts:
If the developer implements a file conversion script, that script must be called from the model reader before the converted file is accessed. In this case, the given *filename* variable should be interpreted as the full file path plus the naming convention pattern. The related logic should first look for a converted file for the requested filename pattern. If no converted file is found, the script should then import the file conversion script and call the script to perform the file conversion. If the file conversion is not successful, then the *self.conversion_test* attribute should be assigned a *False* value, followed by an empty return statement to end execution. Otherwise, the script should continue in its execution.

Full execution class attributes:
When the full reader script is executed, additional attributes are required. The *self._time* attribute must be an one dimensional numpy array of the times in hours since midnight on the day saved in the *self.filedate* attribute. The *self.variables* attribute is a nested dictionary of the format:

```
self.variables[variable_name] = dict(units=variable_units, data=variable_data, xvec =
xvec_dependencies)
```

where *self.variables* is a dictionary, *variable_name* is a string identical to the standardized variable name from the *model_varnames* dictionary at the top of the script, *variable_units* is a string giving the units for the variable (also from the *model_varnames* dictionary), *variable_data* is a numpy array of the variable's data (typically a 4D array), and *xvec_dependencies* is a dictionary of the coordinate dependencies for the given variable (e.g. `{'time':'hr', 'x':'R_E','y':'R_E','z':'R_E'}` for a cartesian grid). The keys of this dictionary should match the coordinate list given in the *model_varnames* dictionary for each variable and the MODEL attribute names to which the given coordinate arrays are assigned (e.g. *self._x* as described below*)*. The *variable_data* numpy array should be ordered such that the 0^{th} row represents the time coordinate, and the remaining rows are in the same order as the conventional choice (e.g. longitude, latitude, height or x, y, z).

Before closing the data file, the script should assign the various coordinate grids to properly named attributes. For instance, if one of the variables in the model data depends on latitude, longitude, and pressure level, then the 1D numpy arrays containing the grid values should be stored in the *self._lon, self._lat*, and *self._ilev* attributes (see Table 3 for the current list). Some of the coordinate descriptions are currently only available for three dimensional variables (e.g. time + 2D spatial) as indicated in the last column. The attribute name should be preceded by an underscore as shown, and should otherwise match the coordinate list given in the *model_varnames* dictionary at the top of the script (the keys in the *xvec_dependencies* dictionary must also match the same list). While we expect to add variations of the coordinate attribute names to the example list given in Table 3, we prefer modelers to use the current list of possible names as additions may require more logic elsewhere. Note the grid values should be converted to match the units given in the last column of the table, which are case-sensitive.

Interpolation:
Once all the variable and coordinate data are stored in their proper attributes, the variable data are then functionalized by assigning an interpolator to each variable name. Once this process is completed, the *self.T_e* attribute of the *MODEL* class becomes an interpolator for the electron temperature data which

can be executed with the line MODEL.T_e([0.5,0.,0.,400.]), where the values in brackets are the time in hours since midnight, the longitude in degrees, the latitude in degrees, and the altitude in kilometers, assuming the variable depends on those coordinates (e.g. the GDZ spherical coordinate system). If the coordinate grids are time-independent (e.g. constant in time) and location-independent (e.g. the same grid spacing throughout the domain), then the model reader script can simply call the *regdef_4D_interpolators* or *the regdef_3D_interpolators* functions from the *read_utilities.py* script, depending on how many coordinates the variable depends on, with the required arguments to create and assign the interpolator to the correct class attribute. If the *gridded_int* keyword is *True* when the *MODEL* class object is called, then a gridded interpolator is also created for the same variable data.

Table 3: Coordinate Naming Examples

Coordinate Description	**Attribute Name**	**Units**	**Dimensionality***
Time in hours since midnight	*self._time*	hr	All
Longitude	*self._lon*	deg	All
Latitude	*self._lat*	deg	All
Radius	*self._radius*	R_E	4D only
Altitude	*self._height*	km	4D only
Pressure level (primary grid)	*self._ilev*	(none)	4D only
Pressure level (secondary grid)	*self._ilev1*	(none)	4D only
X for a cartesian grid	*self._x*	R_E	All
Y for a cartesian grid	*self._y*	R_E	All
Z for a cartesian grid	*self._z*	R_E	4D only
Longitude in electrodynamics grid	*self._Elon*	deg	3D only
Latitude in electrodynamics grid	*self._Elat*	deg	3D only
Magnetic longitude	*self._mlon*	deg	4D only
Magnetic latitude	*self._mlat*	deg	4D only
Magnetic pressure level	*self._milev*	(none)	4D only

*Dimensionality: 'All' means the coordinate name is allowed for both 3D (time + 2D spatial) and 4D (time + 3D spatial) variables, '4D only' means only allowed for 4D variables, and '3D only' means only allowed for 3D variables.

The interpolator called by these two functions, SciPy's *RegularGridInterpolator*, performs linear interpolation on the given grid. The function does not require uniform spacing between grid points, but does require the grid to not change from the initial grid values and to be the same grid-spacing throughout the domain. We also advise the developer to determine if the *RegularGridInterpolator* function performs interpolations with sufficient accuracy, particularly for unusual coordinate systems. Typically, the interpolator is given the model data output for the chosen variable for an entire day; however, smaller segments of data (e.g. for one hour) can be used if the total size of the variable data for a day is larger than ~15 GB (the typical memory limitation of a laptop computer).

If the modeler prefers to use their own interpolation code for their time-independent data, then we recommend they copy the logic executed by the appropriate example function and replace the rgi = RegularGridInterpolator(… calls with calls to their own interpolating function. This logic must include the generation of a non-gridded interpolator, as demonstrated in the *reader_utilities.py* code. A gridded interpolator is preferred, but not required, for irregular or complex coordinate grids and for coordinate grids that change in time. We encourage modelers to use the given code to compare with their own interpolator results. If the interpolator code is written in a language other than python, the developer is

responsible for ensuring the call to the interpolating function from python behaves correctly. We consider it the modelers' responsibility to verify the accuracy of the interpolation method, but encourage modelers to also consider interpolation speed. Typical interpolation times for current model readers are a few seconds or less for 43200 locations (one day of satellite locations with a two-second cadence) for a time-independent grid. Once this code is transferred to CCMC, we expect to coordinate with modelers to ensure the interpolating functions' execution is stable and returns the expected values.

For time-dependent, irregular or complex coordinate grids, the model reader script attributes must change. The 'data' value of the *self.variables* dictionary should be an empty list (self.variables[variable_name]['data'] = []) and the coordinate grid arrays do not need to be stored unless they are constant throughout the time+spatial domain. However, the array of time values must be stored in the manner described previously. In these special situations, we also do not require a gridded interpolating function, as mentioned before, or for an empty interpolator call to return the variable's dataset, but do recommend the capabilities be included if reasonably possible. If the modelers choose to not supply a gridded interpolator, then the modelers must change the default value of the *gridded_int* keyword in the model reader script to *False* and add logic to either raise an error when the user sets *gridded_int* to *True*, or print a message for the user and change the value to *False*.

Section 4: How to Add a New Model Reader to the Flythrough

Once a new model reader is completed and tested, it can be added to the flythrough function via the *model_wrapper.py* script in the */kamodo_ccmc/flythrough/* directory through a few simple steps. First, add the new model acronym as a key-value pair to the end of the *model_dict* dictionary located directly underneath the import statements near the top of the script (e.g. 6: 'ENLIL'). Second, add the new model reader name to the *Choose_Model* function (the second function in the script). This should be done by adding a new '*elif*' block at the bottom of the sequence directly before the '*else'* statement. For example, a model reader named *enlil_4D.py* written for the ENLIL model, the new code block would be written as follows:

```
elif model==ENLIL:
    import kamodo_ccmc.readers.enlil_4D as module
    return module
```

Finally, add the file naming convention to the *FileSearch* function using an 'elif' statement similar to the others in the function. The logic for this block should be nearly identical to that used near the beginning of the '*if fulltime:*' code block from the reader to search for files (e.g. line 88 from the IRI model reader for the simplest case). Compare the similar lines of code from other model readers to their associated '*elif*' statements in the *model_wrapper.py* script for more complex examples. **All other changes to the *model_wrapper.py* script and any other scripts will be rejected** unless previously discussed**.** We expect the developer to test that their new model reader works correctly in the flythrough function and recommend using the various tutorial notebooks in the notebook directory for this purpose. To submit new changes, including a new model reader script, create a pull request on NASA's Kamodo repository on Github and send us some sample model data. We will test the new scripts and changes using the

model data, and will work with the developer to address any concerns before including the changes in the repository.

Section 5: Summary

This document describes the components model reader script required to add new models to the Kamodo-based tools we are developing at CCMC. Examples of reader scripts for time-independent coordinate grids can be found at the GitHub repository (https://github.com/nasa/Kamodo) in the */kamodo_ccmc/readers/* directory. Once a final version of the model output data, the model reader script, and other ccompanying scripts are completed, please send them to the Kamodo team for testing. If modelers have any questions about items discussed in this document, please contact the Kamodo team at CCMC. The contact information for each team member and a list of their focus areas are below.

Rebecca Ringuette: Rebecca.ringuette@nasa.gov
Model reader elements and testing, integrating model readers into the flythrough function, standardized variable names and descriptions, unit string formats and coordinate systems.
Lutz Rastaetter: Lutz.rastaetter@nasa.gov
Calling C/Fortran functions from python, time-dependent and specialized interpolators, standardized variable names and descriptions.
Darren De Zeeuw: Darren.dezeeuw@nasa.gov
Visualization, metadata requirements.

Section 6: Library of Examples

The model readers and file conversion Python scripts are located in NASA's Kamodo github repository (https://github.com/nasa/Kamodo), specifically in the */kamodo_ccmc/readers/* directory. A selection of the model readers and file converters are listed below by model name with the script names, followed by a basic description of the situations encountered in each. These are included below to aid developers in choosing the closest model reader (and file converter if needed) to their own situation. Particularly interesting aspects of each reader are emphasized with bold-faced text.

CTIPe: *ctipe_4D.py* (model reader) and *ctipe_tocdf.py* (file converter)

- *Data:* CTIPe data is divided into three basic files that each cover an entire day of data. Each file contains data on possibly different grids of spatial coordinates. To increase the model reader execution speed, we implemented a file converter script to combine the files into one file per day, handling the various grid dependencies and the array transpositions needed to get the dimension order correct for the interpolators. The final filesizes are well within the computational capabilities of a typical laptop, so no special considerations were needed.
- *Time grid:* CTIPe has a single time grid with spacings that are constant in time. No special code is needed for this.
- *Interpolating between data files:* No files were encountered with only one timestep per file. This data has an open time at the beginning of the day, so the final timestep from the file for the previous day is retrieved. See the logic beginning near line 120 (beginning with *if fulltime:*).
- *Variable dependencies:* CTIPe has two possible pressure level grids in addition to three spatial grids. The *H_ilev* or *H_ilev1* function must be automatically included in the functionalized output

if a variable depending on the respective pressure level grid is requested. **This model reader is a great example of how to handle this in code** (see logic near line 170, 193, and 243).
- *Coordinate grid choice and coverage:* The spherical coordinate grid does not include the end of the longitude range (especially at 360 degrees), so some **additional code was needed to 'wrap' or copy the value at 0 degrees to also be at 360 degrees** (see block beginning at line 54 in the file converter). The given longitude range in the data is from 0 to 360 degrees, so the SPH sph coordinate system in SpacePy was chosen.

GITM: *gitm_4Dcdf.py* (model reader) and *gitm_tocdf.py* (file converter)

- *Data:* GITM data is provided in a binary file with one timestep per file. A file converter was written based on work from the model developers to combine all files for one day into one netCDF4 file. Final file sizes are well within the memory limits of a typical laptop computer. Special logic was used in the model reader to **avoid calculating TEC, NmF2 and HmF2 (time+2D variables)** from the '3D' files if the 2D files are available (which typically contain these variables), which is are time-consuming calculations. (Search for the *flag_2D* variable in the model reader). Array transpositions are also performed in the file converter.
- *Time grid:* The time coordinate grid is constant in GITM data, but the number of time coordinate grids is unknown at the time of execution. See the *variable dependencies* section below.
- *Interpolating between data files:* GITM has an open time at the end of the day, so the first timestep from the next converted file is retrieved. See the logic beginning near line 225 (beginning with *if fulltime:*).
- *Variable dependencies:* The number of file types (e.g. 2D and 3D) are unknown at the time of execution, meaning that the number of coordinate grids is unknown. **This model reader is a great example of how to handle that situation** (beginning near line 300).
- *Coordinate grid choice and coverage:* The spherical coordinate grid includes the poles, so no special code was needed. The given longitude range in the data is from 0 to 360 degrees, so the SPH sph coordinate system in SpacePy was chosen.

IRI: *iri_4D.py* (model reader)

- *Data:* IRI model data output is given in two files, each providing data for the included variables for an entire day. No file converter was deemed necessary.
- *Time grid:* A single time grid is used for both files with constant spacing in time. No special code was needed.
- *Interpolating between data files:* IRI has an open time at the end of the day, so the first timestep from the next converted file is retrieved. See the logic beginning near line 84 (beginning with *if fulltime:*).
- *Variable dependencies:* The number of filetypes is always the same and use the same coordinate grids between the files. **This reader is a great example of the basic logic needed for a model reader in general.**
- *Coordinate grid choice and coverage:* The spherical coordinate grid includes the poles, so no special code was needed. The given longitude range in the data is from 0 to 360 degrees, so the SPH sph coordinate system in SpacePy was chosen.

OpenGGCM_GM: *openggcm_gm_4Dcdf.py* (model reader for small data files), *openggcm_gm_4Dcdf_xarray.py* (model reader for large data files), and file converter (not currently on

github). Note: The current approach to OpenGGCM_GM data is not finalized due to the large data file sizes and related considerations.

- *Data:* OpenGGCM_GM output data is given in compressed binary files with one file per timestep. Currently, we have decided to combine the files into a single netCDF4 file per hour of data, which sometimes results in **file sizes larger than a typical laptop's memory (~15 GB) (see next bullet point)**.
- *Memory handling:* Memory handling is an important consideration for the OpenGGCM_GM data due to the large file sizes of the converted data, even for one hour of data. Our initial approach to handle this was to only allow a few variables to be requested at a time. This was not specifically coded, but the user encountered a memory error. The drawback we encountered was the long execution time for reading in the data from each file while creating the kamodo object (on the order of a few minutes), but the interpolation time was similar to other readers and quite fast. Otherwise, this version of the model reader was typical (see *openggcm_gm_4Dcdf.py*). *Note*: this model reader is outdated compared to the current file conversion method and the other model reader.
  Our second approach uses dask and xarray to speed up the execution time taken to read in the data. The main advantages are a large decrease in execution time (down to ~0.1 seconds from a few minutes) and the ability to assign interpolators for all of the variables in the model data without encountering memory issues. However, the interpolation time through the xarray interface is much slower compared to the other readers (~700 seconds compared to ~0.5 second or less for other model reader's interpolations of the same size grid). (See *openggcm_gm_4Dcdf_xarray.py* for more details.) We are considering other options for this model data in hopes of achieving better execution speeds and better interpolation speeds.
- *Time grid:* A single time grid given with constant spacing in time. No special code was needed.
- *Interpolating between data files:* **Interpolation between data files is handled in the file conversion script**. This script copies one time from an end of one file into the opposite end of the neighboring file to avoid gaps in time between the files. We decided on this approach to avoid the additional execution time needed to open and read a second file of this size.
- *Variable dependencies:* Three spatial coordinate grids are used for the possible variables. The **coordinate grids included in the final object attributes are determined by what variables are requested** (see logic block beginning near line 154 in the xarray version and near line 233 in the normal version). This additional logic helps to save space in memory.
- *Coordinate grid choice and coverage:* The cartesian coordinate grid is associated with the 'GSE' coordinate system as advised by model developers and are already in Earth radii. The model outputs come in negative X and Y coordinates and are flipped before writing out the converted files (i.e., a 180 degree rotation around the GSE Z axis is performed).

SWMF_IE: *swmfie_4Dcdf.py* (model reader) and *swmfie_tocdf.py* (file converter)

- *Data:* SWMF_IE output data is given in .tec files (a specific version of an ascii file for use with Tecplot) with one file per timestep. Each file contains a separate portion for the northern and southern hemispheres. To increase execution speed, we created a **file converter to combine all the .tec files for one day into a single netCDF4 file**.
- *Time grid:* SWMF_IE has a single time grid with spacings that are constant in time. No special code is needed for this.

- *Interpolating between data files:* SWMF_IE output an open time at the end of the day, so the first timestep from the next converted file is retrieved. See the logic beginning near line 134 (beginning with *if fulltime:*).
- *Variable dependencies:* The variable dependencies in the SWMF_IE output are simplistic and all depend on the same time, longitude, and latitude grids. No special code is needed for this.
- *Coordinate grid choice and coverage:* The spherical coordinate grid is associated with the 'SM' coordinate system as advised by model developers. The SM sph longitude range is from -180 to +180, but the range in the data was from 0 to 360. This discrepancy required some custom logic in the file converter to **change the longitude range without changing the 0 degree longitude reference position** (see logic beginning near line 214). Latitude wrapping at the poles was also necessary and used logic similar to that in the TIEGCM reader for scalar averaging (see logic near line 218 in the SWMF_IE file converter). Transposition of the variable data arrays was also needed to put the coordinate dependencies in the correct order.

TIEGCM: *tiegcm_4D.py* (model reader)

- *Data:* TIEGCM data typically has either the entire dataset for one day in one file, or for a large section of the day (e.g. 8 hours). No file converter was implemented.
- *Time grid:* All of the variables in the TIEGCM data depend on a single time grid, which is constant in time. However, the 'time' variable in TIEGCM data is known to be incorrect, so some time wrangling was needed (beginning near line 160).
- *Interpolating between data files:* This data has an open time at the beginning of the day, so the final timestep from the file for the previous day is retrieved. See the logic beginning near line 183 (beginning with *if fulltime:*).
- *Variable dependencies:* This data has three pressure level grids and two lat/lon grids. The associated code is similar to that used in the CTIPe model reader. However, the variable needed by the flythrough function to convert from altitude or radius to **the required pressure level grid is not always included in the data output**. In this case, the pressure level grids are different by a simple offset, and the provided *H_ilev* function is copied to the required function name (see beginning near line 360). The *H_milev* **data also has at least one layer of values equal to 1e36**, which is sliced off to avoid influencing the interpolation output at the top of the magnetic pressure level range (see logic beginning near line 507).
- *Coordinate grid choice and coverage:* The spherical coordinate grids do not include the poles and are not 'wrapped' in longitude (e.g. a value for the last longitude is not provided). The longitude wrapping is handled as in the CTIPe reader, but the **values at the poles required special logic** (see the *wrap_3Dlatlon* and the *wrap_4Dlatlon* functions). For scalar variables, all the values immediately next to the pole at that timestep were averaged to determine the best value. More complex logic was required for proper treatment of the non-scalar variables (see the *vector_average4D* function). The longitude ranges from -180 to +180 degrees and the vertical variable (excluding the pressure level grids) are altitudes, so the GDZ sph coordinate system from SpacePy was chosen.