Kamodo Onboarding Instructions
Draft v0: Rebecca Ringuette, July 8, 2021

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 below. Examples of how to use the tools built on top of Kamodo are in the documentation on GitHub. The remainder of this document focus on what is needed in order for your model to be incorporated into these tools.

For your new model to be accessible through these tools, a 'reader' specific to the model data must be written in python in the format described in these instructions. First, the necessary background details will be described, such as filename conventions and variable definitions. Then, a detailed description of the necessary reader components will follow. Sections are indicated with bold-faced text, with subsection titles in italics. Text highlighted in yellow in the third section emphasize additional documentation requirements not referenced in the next section. Snippets of code are indicated with white text on a black blackground.

Installation Instructions:
The installation instructions below assume you already have the capability to create a new conda environment on your machine and an internet connection. Also, the instructions assume you will be working in python. If you intend to also work with C++/C code, there are a few additional instructions at the end of this section. For a Windows machine, use Anaconda prompt to execute the commands below. Otherwise, use a Linux terminal (or equivalent) in which the 'conda' command has been enabled.

1. Create the appropriate conda environment
    - conda create -n Kamodo_env python=3.7.9
    - conda install -n Kamodo_env -c conda-forge plotly sympy=1.5.1 scipy pytest pandas hydra-core requests ipython netCDF4
    - conda activate Kamodo_env
    - pip install python-forge
    - pip install --upgrade spacepy
    - (If you have trouble installing spacepy, see the spacepy website: https://spacepy.github.io/install.html)
    - (If you plan to call Kamodo from either C++ or C, you will also need to include cmake and pybind11: conda install -n Kamodo_env -c conda-forge cmake pybind11)
2. Install Kamodo in the directory of choice
    - Create a directory in which to install Kamodo (mkdir Kamodo).
    - Download Kamodo from https://github.com/RebeccaRinguette/Kamodo into the Kamodo directory. (This version has the satellite flythrough and reconstruction tools and will soon also have the C++/C interface codes.)

- While in the ./Kamodo directory, install Kamodo in the ./Kamodo directory (pip install -n Kamodo_env ./Kamodo-Master)
- Test your Kamodo installation by executing the following commands with the Kamodo_env environment activated:
  ipython
  import Kamodo
  If no error occurs, your installation is good. You may exit ipython (exit()) and continue.

3. Use it! See the Github repository for examples on how to use the model readers and the tools built on top of Kamodo in the notebooks folder.

If you plan to call Kamodo or one of the tools from C++ or C, there are additional instructions that are operating system dependent. Regardless of the operating system, you will need to download CMakeLists.txt, main.cpp, and the functions.cpp files into the Kamodo_env directory and create an additional directory named 'build' in the same directory. Not all features will be available from other languages, but most functions can be called from the command line when the environment is activated. The following set of commands to call Kamodo or one of the tools from C++ are categorized by operating system, currently including Windows 10 and from a Linux command terminal.

Windows 10:

- conda activate Kamodo_env
- cmake -B build -A x64
- cmake --build build
- **set PYTHONHOME=(*path* to python.exe in env)**
- build\Debug\main.exe
- set PYTHONHOME = C:\ProgramData\Anaconda3
- conda deactivate

The path of python.exe is printed by the message command in the CMakeLists.txt file after the find_package python command. You must execute the second set PYTHONHOME command before attempting to deactivate the environment, using the location of the main python.exe file on your system, or the conda command will not be recognized.

Linux terminal:

- conda activate Kamodo_env
- cmake -B build
- cmake --build build
- chmod +x build/main
- ./build/main
- conda deactivate

The interfaces to C++, C, and Fortran are under currently development and are subject to change.

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.

Data filenames:
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). 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. For example, if the model can at best produce output every minute, then the seconds information is not necessary, but the filename convention should still be identical between runs. Modelers will need to include a description of the filename convention in the documentation to be submitted to CCMC.

Variable metadata:
We will 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 metadata registry described elsewhere. In the documentation, modelers will associate each quantity with their choice from a list of standardized variable names (currently in development). 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, 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. The last two columns should give the list of coordinates that variable depends on, and whether the coordinate grid varies in time.

Table 1: Example of Variable Information

Variable name in output	Standardized variable name	Variable units in data	Coord. Sys.	Dependent coordinate grids	Time-dependent spatial grid?
Electron_temp	T_e	K	GDZ, SPH	time, latitude, longitude, height	No

Total_mass_density	rho_Total	g/cm**3	GSE, CAR	time, x, y, z	No
Numberdensity_molecO	N_O2	1/kg**3	GEO_plev, SPH	time, latitude, longitude, press. lev.	No

Coordinates

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. 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.

The coordinate ranges for each coordinate listed must be included with units in the documentation. Additionally, each variable's coordinate grid should be aligned with a SpacePy coordinate grid, unless it is defined by an internal parameter such as a magnetic field custom to the model. For example, the latitude range for the SPH spherical coordinate grid is from -90 to 90 degrees and the longitude range is from 0 to 360 degrees, both inclusive. These ranges should be the ranges of the coordinate data in the model output, which may not be the same as the assumed range in the modeling code. For example, the coordinate values may be placed in the center of a cell, and so only cover from -87.5 to 87.5 degrees for a 5-degree grid spacing in latitude. While the modeling code understands this to cover the entire range (-90 to +90), the given grid will not be sufficient for accurate interpolation over the same range. An additional grid step on both ends will be needed in the latitude grid data *and* in the corresponding variable data. This issue can either be addressed in the model output data as proposed here or in the reader script described in the next section. In the example reader codes, we typically refer to this as 'wrapping' in either latitude or longitude. See tiegcm_4D.py (found at https://github.com/rebeccaringuette/Kamodo/tree/master/kamodo/readers) for an example where wrapping in both latitude and longitude was needed (the wrap_3Dlatlon and wrap_4Dlatlon functions). If data for multiple times are stored in the same output file, then modelers should also consider placing one time from a previous data output file in the same model run into the next file to prevent time gaps between files. Otherwise, additional logic will be needed in the model reader (see the fulltime keyword behavior description in the Model Reader section), which will increase the execution time, typically by the amount of time it takes to read another file.

It is common for models to provide variable outputs depending upon more than one coordinate system. For instance, some ionosphere-thermosphere models have some variables that depend on a variation of geographical spherical coordinates (time, longitude, latitude, height) and other variables that depend on pressure level instead of height. In these cases, it is imperative for the model data output to include a variable allowing the Kamodo-based tools we are developing to convert between the two coordinate conventions, such as a variable called height that depends on pressure level. This is not necessary for the coordinate systems included in SpacePy (https://spacepy.github.io/autosummary/spacepy.coordinates.Coords.html). Modelers should provide an acceptable coordinate conversion method if the SpacePy conversions are not considered sufficient or do not apply to the situation, 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.

Finally, modelers should also 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. 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 below to read the data into the programming language they plan to perform the interpolation in (typically python).

Model Reader Description

This section describes the required components of the model reader. Examples of currently-used model readers can be found at https://github.com/rebeccaringuette/Kamodo/tree/master/kamodo/readers and have '4D' in the reader names. The reader script names begin with the model name in all lower case letters (no version indicators), followed by an underscore and the characters '4D' (to indicate interpolation in both time and space). If the reader script refers to a secondary file conversion script that converts the model data file format to netCDF4, then we typically add the letters 'cdf' to the reader script name. We expect modelers to follow the given reader script naming convention, and supply the name of the reader script in the documentation. Other ending characters can be chosen to better reflect the script, such as 'int' if the reader script refers to a custom interpolator code. Note that the 4D examples in this directory all depend on functions in the scripts beginning with 'reader_', and the plotting script is only accessed by the test notebooks in the separate notebook directory. Modelers are welcome to consult these notebooks for executed examples of how a reader script should behave, but note that the notebook cannot be re-executed without the data file referenced.

Before the diving into the model reader description, it is necessary to consider the best interpolation method for your model output. The chosen interpolation method *must* interpolate in time *and* space for all dimensions the variable depends on. For time-independent spatial grids, we recommend using SciPy's RegularGridInterpolator as done in the examples in the github repository (see reader_utilities.py). The interpolator performs linear interpolation on the given grid and does not require uniform spacing between grid points, but does require the grid to not change from the initial grid values. For example, if a variable depends on time, longitude, latitude, and altitude, then those coordinate grids cannot be time-dependent if SciPy's interpolator is to be used. Typically, the interpolator is given the model data output for the chosen variable for an entire day for more efficient calculations. However, smaller segments of data are possible if the total set of variable data for a day is too large for memory (see the

readers for GITM and SWMF_IE). An alternate solution is to only allow the reader to register an interpolator for the one variable requested for an entire day, allowing more memory for the operation to occur. Modelers must indicate which method they choose in the model reader documentation.

For cases where the model's spatial grid changes in time, the interpolator is typically written in C or Fortran, and then called from python. We consider it the modeler's responsibility to verify the accuracy of the interpolation method, but encourage modeler's to also consider interpolation speed. Typical interpolation times for current 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.

The content of the model reader script is described below in the typical order the attributes are coded. If the model's spatial grid is time-independent, then the following subsection provides the best description of the necessary components of the model reader to be written in python. The required structure outlined in the following sections applies to both cases. Exceptions to this are given at the end of the last subsection, named *Interpolation*.

Model_varname dictionary

In both the time-independent and time-dependent case, the model reader should be written in python and must subclass Kamodo. A dictionary called 'model_varnames' should follow the import statements, which will serve as a variable directory for the model 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. The first value in the list should be a string equal to the standardized representation of the variable name, followed by a string containing the standardized variable description, an integer for that variable, and two strings equal to the standardized name of the coordinate system the variable depends on (e.g. 'SM','car' or 'GDZ','sph'). The third value should be 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']). The final element in the main list for the key value pair should be a string representing the proper python 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. This dictionary is followed by any functions needed for the execution of the remainder of the script, typically consisting of various time conversion functions.

Class call signature and behavior

The main portion of the python reader script is the definition of the MODEL class. This class must subclass Kamodo, and have the same arguments and keywords and defaults as shown in Figure 1. The class should be returned by a function also named MODEL, in which Kamodo should be imported. This is to decrease the number of times Kamodo is imported in the execution of upper level software. The filename argument must include the full path to the file. With the given keyword defaults as shown, users should be able to import the MODEL class from the script and return the entire functionalized data set by executing the call 'kamodo_object = MODEL(*filename*)' in an ipython or notebook session, as shown in the reader notebooks found in the github repository (https://github.com/rebeccaringuette/Kamodo/tree/master/kamodo/notebooks/).

```
def MODEL():
    from numpy import array, zeros, abs, NaN, unique, insert, diff, where
    from time import perf_counter
    from os.path import isfile, basename
    from kamodo import Kamodo
    print('KAMODO IMPORTED!')
    from netCDF4 import Dataset
    from kamodo.readers.reader_utilities import regdef_4D_interpolators, regdef_3D_interpolators

    #main class
    class MODEL(Kamodo):
        '''CTIPe model data reader'''
        def __init__(self, full_file_prefix, variables_requested = [], filetime=False,
                     runname = "noname", printfiles=False, gridded_int=True,
                     fulltime=True, verbose=False, **kwargs):
```

Figure 1: The required arguments and keywords of the MODEL class object. The import statements vary between readers.

Each keyword governs different behavior of the class object necessary to interact with higher layers of software. The variables_requested variable is by default an empty list, which acts to functionalize all possible variables in the model data output. Modelers can code in variables to be explicitly excluded as desired unless the variable name is in the variables_requested list. If the user specifies a non-empty list for the variables_requested variable (e.g. variables_requested=['rho','TEC']), then the reader should use the model_varnames dictionary to translate from the standardized variable name(s) given in the list to the associated variable name in the model data. In this case only variables requested should be returned, with one exception. If a variable depends on an alternate coordinate system (e.g. pressure level instead of height or an internally defined magnetic longitude and magnetic latitude instead of the geographic versions), then the accepted conversion function should also be returned. This could be either a functionalized height function for the pressure level case or some other user defined function to be used for the coordinate conversion. Examples of reader scripts with this behavior include the tiegcm_4D.py and ctipe_4D.py scripts.

The majority of the remaining keywords are simplistic but necessary. The runname keyword is simply for metadata purposes. At CCMC, all model runs are given an unique run name. If known, the user is expected to assign the run name as a string to the runname keyword at the class call, which is then saved as a class attribute for internal documentation (self.runname = runname). The verbose keyword has its typical application: to print various messages to the screen for troubleshooting. This keyword default is set to False to decrease the amount of printed output when the reader is called by higher software levels. The gridded_int keyword is set to True by default to functionalize not only the variables on the given coordinate grid, but also an alternate form of the functionalizes variables, called a gridded interpolator, which in some cases is easier to interact with.

The printfiles keyword is set to True by default, which simply means the complete list of model data files used to generate the class object is printed to the screen. Otherwise, the list of files can be accessed by the filename class attribute (self.filename=filename). Modelers will have to write code to incorporate data from multiple files if their data is stored in one file per timestep and/or in multiple files per timestep or per day (e.g. ctipe_4D.py or gitm_4Dcdf.py). Typically, such a complication is handled by using a second script layer to be called by the reader script. This secondary script layer collects all the

model data files given the file prefix, reads in the coordinate and variable data, performs any coordinate wrapping necessary, and writes all of the output to a netCDF4 file named with the file prefix given and the corresponding file extension. This file conversion script is only be called the first time the given files are called, as shown in the indicated reader scripts, and greatly decreases the execution time on subsequent calls. Also, the netCDF4 file size is typically smaller than the total size of the contributing files and so is the preferred data format if possible. This approach has also proved useful to convert from ascii output to netCDF4, also reducing the execution time after the first call (see swmfie_4Dcdf.py). If the modeler chooses to write their own interpolator in a different language, then a different file format will likely be needed. The scripts in the GitHub repository serving as file converters have '_tocdf.py' in the script names.

The filetime and fulltime keywords work together to accomplish a few goals. Given the default values, the scripted behavior is to fully execute the reader code (filetime=False) and add a time value from a neighboring file, if available, to better handle the time interpolation between files (fulltime=True). Setting both keywords to True returns a Kamodo class object with only three time-related attributes called filedate, datetimes, and filetimes. 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. Given the stated values of the keywords, these two time ranges will include a time step from a neighboring data file, if available. No other attributes should be assigned for this keyword combination to minimize the execution time. This behavior is required by the higher layers of code to assign the requested data times to the correct reader MODEL object.

Setting the filetime and fulltime keywords to True and False, respectively, returns a similar MODEL object, but without the added time value. This combination is used by the reader script to call itself recursively to find a neighboring model data file to add the appropriate time value. On the other hand, setting both the filetime and fulltime keywords to False returns a MODEL object for the given file without adding time from a neighboring file with the three time related attributes previously mentioned and a self.short_data attribute. This 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 reader is simply which end of the time range best completes the dataset for the day (or hour).

Secondary file conversion scripts

Before discussing the necessary attributes for a full execution, we pause here to explain the necessary logic needed when a secondary file converter script is included. In these cases, the filename variable in Figure 1 should be replaced by a file_prefix variable, which should include the full file path as before, but only the first portion of the filename pattern instead of an entire filename. (See gitm_4Dcdf for an example.) Modelers must include the format of the file prefix in the documentation. The file conversion

logic should be placed before the self.filedate, self.datetimes, and self.filetimes attributes are assigned to ensure it is executed for each call creating a MODEL class object. The logic should first look for a previously converted file for the file prefix requested. If no converted file is found, the script should then import the second 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 (filetime=False and fulltime=True), additional attributes are required, as listed below. The self._time attribute is 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)
```

In the example line of code above, self.variables is a dictionary, variable_name is a string identical to the standardized variable name, variable_units is a string giving the units for the variable, variable_data is a numpy array, typically a 4D array, of the properly wrapped and ordered data, and xvec_dependencies is also a dictionary. The variable_data numpy array should be ordered such that the 0^{th} row represents the time coordinate, and the remaining two or three rows are in the same order as the conventional choice (e.g. longitude, latitude, height or x, y, z). Finally, the xvec_dependencies dictionary is a simple dictionary of strings where the keys are strings for the coordinate names the given variable is dependent upon, and the values are the units of those coordinates. For example, the xvec_dependencies dictionary would be {'time':'hr','lon':'deg','lat':'deg'} for a variable that depends on time, longitude, and latitude but not on height. If a variable is dependent on geocentric solar ecliptic coordinates, the dictionary would be {'time':'hr', 'x':'R_E','y':'R_E','z':'R_E'} to indicate the values for each coordinate are given in earth radii. The standard name of the coordinate system for that variable is to be indicated in the model_varnames dictionary at the top of the script, as described at the beginning of the section. 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 coordinates are assigned.

Before closing the data file, the script should assign the various coordinate systems to properly named attributes. For instance, if one of the variables in the model data depends on geographical latitude, geographical 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. Other commonly-used attribute names are self._Elon and self._Elat for electric field longitude and latitude, self._mlon and self._mlat for magnetic longitude and latitude, self._ilev1 for a secondary pressure level grid, and self._x, self._y, and self._z for the primary cartesian grid. 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 (just as the keys in the xvec_dependencies dictionary must also match). While we expect to add variations of the coordinate attribute names to the example lists given, we prefer modelers to use the current list of possible names as additions will require more logic in the reader plotting code.

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 is 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 geographical longitude, the geographical latitude, and the altitude, assuming the variable depends on those coordinates (GDZ spherical). When the variables in the data depend on more than one coordinate grid, the logic to retrieve the correct coordinate arrays becomes slightly more complex. (Compare the register_4D_variable functions in the ctipe_4D.py script with the iri_4D.py script or the gitm_4Dcdfpy script.) If the coordinate grids are time-independent, the model script can simply call the regdef_4D_interpolators or the regdef_3D_interpolators function from the read_utilities.py script with the required arguments, depending on how many coordinates the variable depends on, to create the interpolator using the given coordinate grids and variable data, and assigned 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. Our users find this alternative version of the interpolator to be useful for faster interaction with Kamodo's visualization capabilities.

If the modeler prefers their own interpolation code for their time-independent data, then we recommend they copy the logic executed by the regdef_4D_interpolators and/or the regdef_3D_interpolators functions, replacing the rgi = RegularGridInterpolator(... calls with their own interpolating function. This logic must include the generation of a non-gridded and a gridded interpolator, as demonstrated in the reader_utilities.py code. We encourage modelers to use the given code to compare with their own interpolator results. Custom interpolator codes should be named with the model name in all lower-case letters followed by an underscore and the letters 'int'. The name of the interpolator code must be included in the documentation (e.g. tiegcm_int.py or tiegcm_int.f90). If the interpolator code is written in a language other than python, the modeler is responsible for ensuring the call to the interpolating function from python behaves correctly. 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 coordinate grids, the reader script attributes must change. The 'data' value of the self.variables dictionary should be an empty list (self.variables[variable_name]['data'] = []). In the time-dependent coordinate grid situation, we do not require a gridded interpolating function (e.g. the define_4d_gridded_interpolator and the define_3d_gridded_interpolator functions in reader_utilities). If the modelers choose to not supply this function for the time-dependent coordinate case, then the modelers should change the default value of the gridded_int keyword in the MODEL call to False, and add logic to raise an error or otherwise handle the case where the gridded_int is set to True.

Summary

This document describes the model reader script necessary to add new models to the Kamodo-based tools we are developing at CCMC. The components necessary to include in the documentation are described in the second section with additional comments highlighted in yellow in the third section. Examples of reader scripts for time-independent coordinate grids can be found at the GitHub repository in the reader directory (https://github.com/rebeccaringuette/Kamodo/tree/master/kamodo/readers).

Once a final version of the model output data, the model reader script, and other necessary scripts are completed, please send them to the Kamodo team for testing. If modelers have any questions about items discussed in this document, again please contact the Kamodo team at CCMC. The contact information for each team member and a list of their focus areas are given below.

Rebecca Ringuette: Rebecca.ringuette@nasa.gov

Model reader elements, standardized variable names and descriptions, unit string formats, bridging between C++ and python (either way), coordinate conversions and naming.

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.