Kepler Mission's Focal Plane Characterization Models Implementation

Christopher Allen[a], Todd Klaus[a], and Jon Jenkins[b]

[a]Orbital Sciences Corporation, NASA Ames Research Center, MS 244-30, PO Box 1, Moffett Field, CA 94035

[b]SETI Institute, NASA Ames Research Center, M/S 244-30, Moffett Field, CA, USA 94035

ABSTRACT

The *Kepler Mission* photometer is an unusually complex array of CCDs. A large number of time-varying instrumental and systematic effects must be modeled and removed from the *Kepler* pixel data to produce light curves of sufficiently high quality for the mission to be successful in its planet-finding objective. After the launch of the spacecraft, many of these effects are difficult to remeasure frequently, and various interpolations over a small number of sample measurements must be used to determine the correct value of a given effect at different points in time. A library of software modules, called Focal Plane Characterization (FC) Models, is the element of the *Kepler* Science Processing Pipeline that handles this. FC, or products generated by FC, are used by nearly every element of the Science Operations Center (SOC) processing chain. FC includes Java components: database persistence classes, operations classes, model classes, and data importers; and MATLAB code: model classes, interpolation methods, and wrapper functions. These classes, their interactions, and the database tables they represent, are discussed. This paper describes how these data and the FC software work together to provide the pipeline with the correct values to remove non-photometric effects caused by the photometer and its electronics from the *Kepler* light curves. The interpolation mathematics is reviewed, as well as the special case of the sky-to-pixel/pixel-to-sky coordinate transformation code, which incorporates a compound model that is unique in the SOC software.

Keywords: Kepler Mission, Focal Plane Characteristics, Calibration, NASA, Ames, Kepler, CCD, transit photometry, infrastructure, software, Java, MATLAB, extrasolar, space telescope

1. INTRODUCTION

This article describes the end-to-end use of the *Kepler Mission*'s[1,2] Focal Plane Characterization (FC) Models in the Kepler Science Processing Pipeline[3,4] (hereafter "pipeline"). The FC models consist of a set of database tables, persistence classes, and associated handling code that determine the correct values of a variety of instrumental effects for the purposes of data calibration at any time during the mission. Calibration of the *Kepler* pixel data is necessary to produce light curves of sufficiently high quality for the mission to be successful in its planet-finding objective.

The general structure of each FC model is a triad of database tables. The first table, referred to as the Data Table, contains measured values of the model at one or more points in time. The second table contains a set of "History" objects, used to determine which set of values from the Data Table were valid at a given time during the mission life and to maintain a complete history of the best-knowledge values of each FC model over time during the mission. The third table links data entries in the first table to the History objects in the second table in a one-to-many relationship (*i.e.*, one History object can be linked to many entries in a given Data Table). The linking table also enables the *Kepler* Science Office (SO) to fine-tune their understanding of the model values over time by allowing the insertion of updated or corrected values into the Data Tables.

The FC models are used by *Kepler* software in many ways. A few examples are:

Further author information: (Send correspondence to C.A.)
C.A.: E-mail: Kester.Allen@nasa.gov, Telephone: 650-604-2412

1. Observation planning makes extensive use of the pixel-to-sky coordinate transformation provided by the RaDec2Pix model and the Pixel Response Function (PRF) model.
2. Pixel-level data calibration uses the gain, flat field, two-d black, and linearity models, among others.
3. Photometer performance assessment uses many of the models to determine the health of the *Kepler* focal plane.

A user-friendly construction of the FC model data is provided by Model objects (see Section 2.3). FC model objects contain no internal database representation data; they only contain the *Kepler* focal plane information that users would be interested in. The model objects are also annotated with a set of data accountability information. This data accountability set records which database account the model was created from, what location in the data change control repository the data originally came from, what time the data was ingested into the FC system (see Section 2.4), and how the data was described by the pipeline operator who performed the ingest.

2. JAVA CODE

Java code handles the business logic and pipeline integration of the FC models. The FC Java components are database persistence classes, operations classes, model classes, and data importers.

2.1 Database Classes

Database classes handle the object-relational mapping (ORM) work of persisting the data parsed by the Importer classes (see Section 2.4) to the *Kepler* database. The *Kepler* project uses the Hibernate Java libraries for ORM. These classes are not used directly by users. Objects of the Database classes contain the actual data, and the objects' accessors are used by other code to extract values. The objects' methods are fairly low level; most of the complicated work of interpolation or model construction is done by the Operations classes (see Section 2.2).

2.2 Operations Classes

The purpose of the Operations classes is to hide the low-level database operations from users. These classes present a high-level API which performs the high-level tasks that users of the FC library will want to carry out. For example, the GeometryOperations class method `retrieveGeometryModel` returns a GeometryModel object that is valid for the time range specified in the input arguments. The method `retrieveGainModel(double mjdStart, double mjdEnd)` returns a GainModel object that gives the correctly interpolated gain values for the *Kepler* field of view (FOV) for the time range between the two Modified Julian Date (MJD) times given as input arguments.

The low-level database operations corresponding to the first example include:

1. Instantiate GeometryCrud class.
2. Query for the latest History Object.
3. Query for the GeometryHistoryModel objects that are linked to the History object.
4. Retrieve the Geometry objects that are linked to the GeometryHistoryModel objects.
5. Copy the contents of the Geometry objects into a GeometryModel object (retaining the data, but discarding the persistence metadata).

Figure 1 is a class diagram of the ReadNoiseOperations class, and illustrates the typical group of methods available to a Operations object. The getHistory method returns the History object that is associated with the current dataset, the retrieveReadNoise/persistReadNoise methods perform the database read/write operations, and the retrieveReadNoiseModel and retrieveReadNoiseModelAll methods extract Model objects.

The FC CRUD class, FcCrud, is used by the Operations classes to perform the actual database operations, *i.e.*, adding rows to or extracting rows from existing database tables. CRUD (Create, Read, Update, and Delete) is an ORM term that indicates low-level database use. tables. The FcCrud methods are used directly only in unusual circumstances or for debugging.

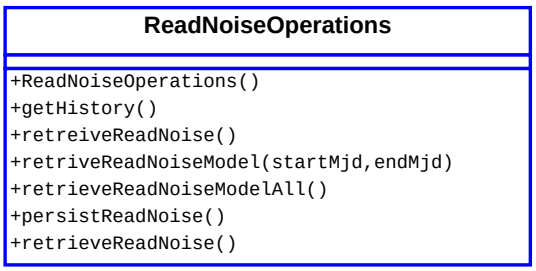

Figure 1. A class diagram for a representative Operations class (ReadNoiseOperations).

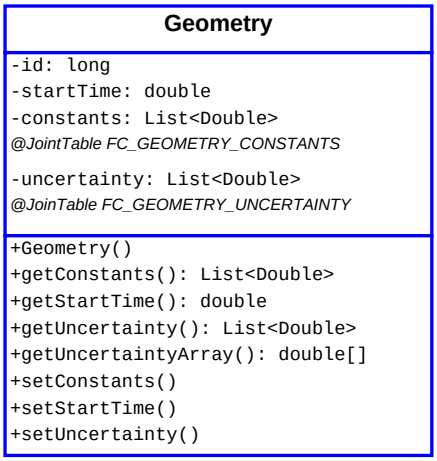

Figure 2. A representative class diagram for the Geometry Hibernate class.

2.2.1 Data Classes

The Data classes define the objects that are actually stored in the database by Hibernate. In addition to the model data, they contain database metadata which is used to persist and depersist the information into database tables.

The class diagram for the Geometry class in Figure 2 illustrates the structure of a data class. The member variables `startTime`, `constants`, and `uncertainty` are the actual stored data, as supplied by the SO. The member variable `id` is used for database persistence. The methods are standard accessors for getting or setting the data contents.

2.2.2 History Class

The History class provides a single database reference to the entire dataset of one FC Model at a given time. It has a one-to-many relationship to the HistoryData mapping classes, which in turn have a one-to-one relationship to the Data classes.

Using the History class as a reference for a model's dataset, additional data can be appended to refine the understanding of *Kepler* focal plane responses. The process involves adding additional HistoryData Mapping objects to the database, which are linked to the appropriate pre-existing History object, and then adding Data objects which are linked one-to-one to these HistoryData mapping objects.

2.2.3 HistoryData Mapping Classes

The HistoryData classes provide a mapping between the History objects and the Data objects. They are not created or used except in conjunction with the Data class objects, and are not seen by users. They serve to link the History objects to the actual data associated with them. History objects are created by Importers during importation of data (see Section 2.4).

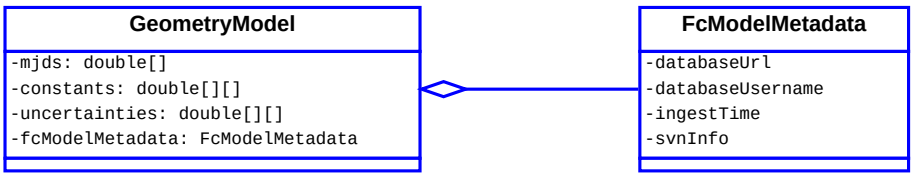

Figure 3. Class diagram for the GeometryModel class and the aggregated class FcModelMetadata. The getter/setter methods are excluded from this view.

2.3 Model Classes

Model classes contain the persisted information (model values), but do not include the metadata necessary to map the objects into a relational database. Internal database details (metadata like table key IDs, etc.) are not needed by FC end users and are excluded from the Model objects. The data types in the Model are all types that can be passed between Java and MATLAB. Data passing from Java to MATLAB is the primary use of the Model classes. Figure 3 shows the linkage between model data and the FcModelMetadata class. The FcModelMetadata object provides an easy way for users to view which database their data came from, when it was ingested and by whom, and the SVN address of the original import data.

Model objects are constructed for a time range, which is given by the user as a pair of start/stop arguments. If the time range is not specified, the Model object is constructed with the largest possible amount of data and widest possible time range. This behavior guarantees the correct interpolation between actual measurements of model value, since interpolation for a given History is always either done between two measurements or extrapolated from the same two measurements. If additional measurements need to be added to a model between two existing entries, the SO will deliver a new set of blessed data files for the Importers (see Section 2.4) to process, and a new History will be created. The Kepler SOC's data accountability requirements are also met by using the individual History objects to track the individual model components that are used in any processing operation.

If a user requests a time range that does not include more than one model measurement, the bracketing measurements are includes in the output Model object. Results that are generated within a valid time range are guaranteed to be the same for any other object with a different valid time range at the same time, if the time range for that object is also within the valid time range.

2.4 Importer Classes

Importer classes provide command-line tools to persist formatted text files into the database as FC models. An agreement between the SOC and the SO defines the text file format for each type of data. Human operators execute the Importer tools after the SO approves new data; in practice, this happens infrequently. An example of a triggering event for an importer run would be analysis that indicated the readout gain for the *Kepler* FOV had changed, or that the electronics undershoot had become more severe. Each Importer parses in data from the text file, constructs an object (*e.g.*, a Geometry object), and persists the object into the database. History and HistoryModel entries are also created and persisted as appropriate.

Two broad categories of importer data exist: pixel-based and nonpixel-based. Pixel-based models include flat field and two-d black; the data volume for these models is very large compared to nonpixel-based models. The import files for pixel-based models define a value for each pixel in the focal plane. Nonpixel-based models contain information that describes per-CCD parameters (*i.e.*, geometry,[5] read noise), per-readout chain parameters (*i.e.*, readout linearity), and spacecraft-based parameters (*i.e.*, roll times, spacecraft pointing). Nonpixel-based import files define only the parameters necessary to calculate the relevant effect.

The importers are implemented as command-line tools and execute the Java `main()` method. The command-line arguments include a directory, which is where the importer files are located. An SO-SOC agreement defines a filename format required by importer files; the Importer classes determine which files are valid based on a regular expression matching operation using the filename definition.

Figure 4 shows the structure of two importer tools. The gain importer `ImporterGain` inherits from `ImporterNonImageParent`, which contains code to handle importing non-image-based models. The 2D-black

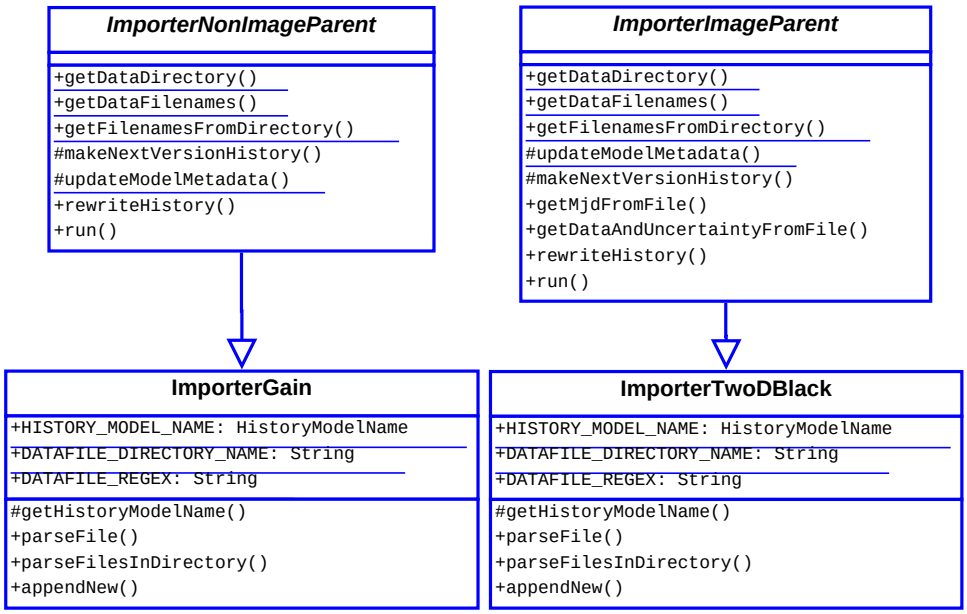

Figure 4. Class diagrams of the representative Importer classes ImporterGain and ImporterTwoDBlack, and their respective parent classes.

importer `ImporterTwoDBlack` inherits from the image-based importer, which handles the larger-sized image-based models.

3. MATLAB CODE

3.1 Model Parsing

The MATLAB classes mirror the Java Model classes in most respects. A significant difference is that MATLAB pixel-based classes (two-d black and flat field) can be constructed in two ways, as either:

1. an entire image, or
2. a subset of requested pixels.

In this way, the user is given the option to avoid downloading and processing a model dozens or even hundreds of times larger than necessary if only a small subsection of the entire dataset is needed. Since the number of pixels on the focal plane used to image *Kepler* targets is a relatively small fraction of the total number of pixels, this technique can provide significant improvements in runtime and decreases in data volume for algorithms that are operating on target pixels.

Each MATLAB FC class contains standard accessor methods, and additionally the method that performs the interpolation of the data to the time or times requested by the user.

3.2 Model Interpolation

The type of interpolation that is performed in FC models between SO-supplied data is handled on a model-by-model basis. For most models the interpolation is linear between measurements, but in the electronics readout linearity model (LinearityClass), pixel undershoot (UndershootClass), and pixel response function (PrfClass), more complex methods are used. This is because the internal representations of these models involve polynomial components, linear filter components, or solutions to centroiding operations.

The actual interpolation operation is performed by the MATLAB built-in function `interp1`. For almost all models, the interpolation method is linear interpolation. Exceptions include the two-d black model, which uses

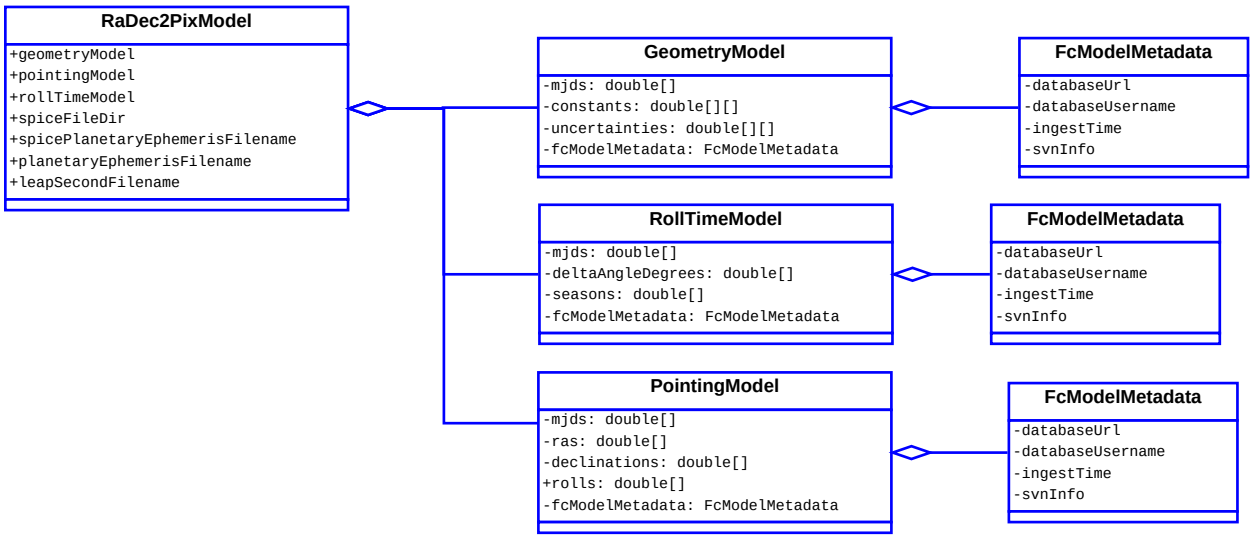

Figure 5. Partial class diagram of the RaDec2PixModel class to illustrate its composite nature.

an interpolate-between-points and extrapolate-beyond-points method (the 'extrap' option for `interp1`), and the Pixel Response Function (PRF) model, which uses an opaque internal data format (a MATLAB object generated by the SO), and is not interpolated.

3.3 RaDec2Pix: A Composite Model

The RaDec2Pix model defines the sky-to-pixel and pixel-to-sky coordinate transformations, *i.e.*, where a star falls on the *Kepler* FOV at any given time. It is unique in that it contains three other models as submodels; it includes a Geometry model, a Pointing model, and a Roll Time model as internal components. It also includes the path to several locally cached ephemerides files, which are retrieved from the datastore if there is no local cache or if the local cache does not contain the most current ephemerides.

The compound nature of the RaDec2Pix model places some limitations on its use. For example, to guarantee correct interpolation, the model's valid time range must begin at the latest start time of the three submodels and end at the earliest end time of the three submodels. If this were not the case, results from times that were inside one submodel's valid range but outside that of another's would be incorrect. The compound nature of RaDec2Pix is illustrated in Figure 5.

SPICE ephemeris[6] files are used by the *Kepler* mission to determine the position and velocity of the spacecraft. This information is needed by the RaDec2Pix model to perform velocity aberration calculations as part of the sky-to-pixel/pixel-to-sky transformation. The RaDec2PixOperations Java class uses the EphermerisOperations class to make a locally cached copy of the SPICE ephemeris files. EphermerisOperations verifies if a local copy of the ephemerides exists and if it is the same as the most current database version; if it is not, a new copy is downloaded and cached to the local disk.

3.4 Science User Tools

The science user tools (also called wrappers) are a set of tools written in MATLAB. The purpose of these tools is to provide the SO users, who are in general MATLAB- but not Java-literate, with a method to retrieve Model objects for the FC models. The science user tools encapsulate all the Java work and database management. The Model objects that are returned are translated into MATLAB data types, *e.g.*, Java String objects are converted into MATLAB char arrays.

Kepler's science user tools are explicitly designed to return model objects that are identical to those delivered to the MATLAB modules by the Java interfaces. This requirement introduces MATLAB structures that appear

unusual, including an ".array" substructure which is designed to mimic a Java array-of-arrays. This substructure is what the Java module interface produces; the science user tools produce the same structure to ensure compatibility for module interface testing and scientific module programming.

Additional tools are provided for convenience purposes: a tool to list the available time ranges of cadence data (`retrieve_available_data_ranges`), routines to convert between short cadence number and long cadence number (`convert_short_cadence_to_long_cadence convert_long_cadence_to_short_cadence`), and a retriever for general filestore time-series data (`retrieve_ts`).

A psuedocode illustration of the general pattern of a science user tool is as follows:

```
operations = GainOperations();
model = operations.retrieveGainModel(mjdStart, mjdEnd);

data.mjds = model.getMjds();
constants = model.getConstants();
for i = 1:length(constants)
    data.constants(i).array = constants(i,:);
end
data.fcModelMetadata = model.getFcModelMetdata();

return data;
```

The tool instantiates an Operations (Java) object, and retrieves a Model (Java) object. The data in the Model (in this case, the MJD and values of the gain measurements) are parsed out into a MATLAB data structure. Finally, the model's metadata is copied. (In this illustration, preallocation, argument parsing, and Java imports are excluded in order to increase clarity.)

4. USECASE WALKTHROUGH

The sections below illustrate a general usecase for software users who interact with the FC Models.

4.1 Java-based Walkthrough

A programmer using the FC Models library in a Java context would perform the following sequence of operations to retrieve one or more FC Models for pipeline or debugging use:

1. Instantiate appropriate Operations object.
2. Execute the appropriate `operations.retrieve*Model` method, returning a Model object.
3. Package the Model object and

   pass it through the module interface, or

   perform local operations.

4.2 MATLAB-based Walkthrough

A programmer using the FC Models library in a MATLAB context would use the following sequence of operations to produce a working copy of FC Models data:

1. Retrieve model either through

   module interface or

   Science User Tool (the appropriate `retrieve_*_model` tool).

2. Instantiate a MATLAB Model object.

3. Execute the appropriate `matlabObject.getModelValue(time)` method to calculate the value of the model at the given time.

4. Apply these values to *Kepler* data.

It is worth noting that since the MATLAB-side code is well-insulated from the details of the FC Models' implementation, the *Kepler* scientific programmers working in MATLAB do not need to know anything about the actual database (configuration, setup, or environment). This improves the efficiency of programmers working on scientific modules, and increases the maintainability of the codebase.

5. CONCLUSION

The FC Models provide the pipeline and SO users access to complete information about the state of the *Kepler* focal plane. This information is used to plan observations, calibrate data, and assess photometer performance. The correctness, usability, and speed of these data are critical components of the *Kepler* processing and analysis software. The FC Models use both the Java and MATLAB SOC infrastructures, and are accessible by both the SO and pipeline. The history and data accountability of model data is managed via the described History and HistoryModel objects.

ACKNOWLEDGMENTS

Special thanks to William Borucki, David Koch, and Miles Cote for assistance with this work, and to the *Kepler* team, without whom this work would not have been possible.

Funding for the *Kepler Mission* is provided by NASA's Science Mission Directorate.

REFERENCES

[1] Borucki, W. J., *et al.*, "Kepler planet-detection mission: introduction and first results" *Science* **327**, 977–980 (2010).

[2] Koch, D. G., *et al.*, "Kepler Mission design, realized photometric performance, and early science" *ApJL* **713(2)**, L79–L86 (2010).

[3] Middour, C, *et al.* "Kepler Science Operations Center Architecture" *Proc. SPIE* **7740**, in press (2010).

[4] Klaus, T. C., *et al.*, "The Kepler Science Operations Center science pipeline configuration and execution" *Proc. SPIE* **7740**, in press (2010).

[5] Tenenbaum, P., and Jenkins, J. M., "Focal plane geometry characterization of the Kepler Mission" *Proc. SPIE* **7740**, in press (2010).

[6] JPL, http://naif.jpl.nasa.gov/naif/toolkit.html, (2010).