The LSST Metrics Analysis Framework (MAF)

R. Lynne Jones[a], Peter Yoachim[a], Srinivasan Chandrasekharan[b], Andrew P. Connolly[a], Kem H. Cook[c], Željko Ivezić[a], K. Simon Krughoff[a], Catherine Petry[d], Stephen T. Ridgway[b].

[a]University of Washington, Address, Seattle, USA;
[b]National Optical Astronomy Observatory, Address, Tucson, USA
[c]Eureka Scientific, Inc., Address, City, USA
[d]University of Arizona, Address, Tucson, USA

ABSTRACT

We describe the Metrics Analysis Framework (MAF), an open-source python framework developed to provide a user-friendly, customizable, easily-extensible set of tools for analyzing data sets. MAF is part of the Large Synoptic Survey Telescope (LSST) Simulations effort. Its initial goal is to provide a tool to evaluate LSST Operations Simulation (OpSim) simulated surveys to help understand the effects of telescope scheduling on survey performance, however MAF can be applied to a much wider range of datasets. The building blocks of the framework are Metrics (algorithms to analyze a bit of data), Slicers (subdividing the overall data set into smaller data slices as relevant for each Metric), and Database classes (to access the dataset and read data into memory). We describe how these building blocks work together, and provide an example of using MAF to evaluate different dithering strategies. We also outline how users can write their own custom Metrics and use these within the framework.

Keywords: Large Synoptic Survey Telescope, LSST, Metrics Analysis Framework, Operations Simulation

1. INTRODUCTION

As LSST moves toward construction, significant effort is being invested in understanding the effect of telescope scheduling on the overall performance of the survey. The LSST Operations Simulations group is working towards this goal by producing sets of simulated surveys created with a range of observing strategies. These simulated surveys are created by the Operations Simulator (OpSim).[?,?,?] Each survey includes a detailed model of the telescope pointing and movement capability, weather conditions, scheduled and unscheduled downtime, and a scheduler algorithm which can be programmed with different sets of requested observations. The output of each simulated survey run is a 10-year simulated pointing history for LSST, consisting of approximately 3 million visits. LSST plans currently call for each visit to consist of two 15-second back-to-back exposures. The OpSim output includes the location of these visits, together with the observing conditions (airmass, seeing, sky brightness, limiting magnitude, etc.) for each visit, along with records of the telescope state over the 10-year simulation and information relevant to the internal state of the simulator itself. To help evaluate these simulated survey outputs, we have built the LSST Metrics Analysis Framework (MAF).

In broad overview, MAF is an open-source python software framework that provides a user-friendly, easily-extensible, easily-customizable set of tools to

- read the OpSim simulated survey data from a database,
- slice the data according to the values of single or multiple columns within the data or the spatial location of the data points,
- apply metrics to each data slice, saving the results,
- and visualize the metric results,

Further author information: (Send correspondence to R.L. Jones)
R.L. Jones.: E-mail: ljones@astro.washington.edu, Telephone: 1-206-543-9487

while also preserving metadata about the source of the data and how the data was sliced. MAF provides utilities which make it possible to automatically generate reasonable plot labels and titles, while maintaining the ability for the user to override and provide custom labels, titles, axes ranges, etc. Additional utilities make it easy to add new data columns 'on the fly' for run-time extension of existing database tables. More details on the overall design are described in Section 2.

An example of analyzing an OpSim simulated survey using MAF is described in Section 3, including a brief comparison of how a user can evaluate different potential dithering methods.

MAF is intended to be extensible, in that it is designed to allow a user to easily write their own metrics, which can then be plugged into the framework and used in the same manner as any of the provided metrics. Users can also write new data slicing classes if needed, as well as point the framework to any kind of database. Base classes demonstrate the APIs for each of these aspects of the framework, and documentation (including tutorials) is provided online at `https://confluence.lsstcorp.org/display/SIM/MAF+documentation`. While the primary goal of MAF is to analyze the LSST OpSim outputs, we have built the framework to be easily applicable to more general use-cases as well.

We anticipate that the most common extension of MAF will be users writing their own Metrics, specifically to address their science concerns. A demonstration of the Metric API with a view towards the requirements when writing new Metrics is provided in Section 4.

All code for LSST MAF can be git cloned from or browsed online at the LSST Stash repository, `https://stash.lsstcorp.org/projects/SIM/repos/sims_maf/`.

2. FRAMEWORK DESIGN

The heart of the MAF framework are the Metrics and Slicers. These basic code classes correspond to defining "what algorithm is being calculated with the data" (the Metric) and "what unit of data is being evaluated" (the Slicer). As some examples: if we want to analyze "the mean airmass of all visits", the Slicer would simply pass all visits to a Metric which calculates the mean of the airmass values; if we want to analyze "the mean airmass as a function of RA/Dec", the Slicer would identify visits which overlap a series of RA/Dec points and pass each of those subsets to a Metric which calculates the mean of those airmass values. In these two examples, the Slicers (defining the unit of data being evaluated at a time) are different, but the Metric itself (what calculation is being done) is the same. Some more examples: if we want to analyze "the mean single visit limiting magnitude as a function of RA/Dec" as well as "the total number of visits as a function of RA/Dec", we would use the same Slicer but different Metrics. These examples emphasize a fundamental point about the framework: Metrics and Slicers have been built to be modular and interchangeable.

Metrics are very familiar sorts of things: they are simply algorithms to analyze a bit of data. MAF provides a set of Metrics that include a number of very simple algorithms, as well as some more complex Metrics focused on analyzing some aspects of the cadence of observations or the technical performance of an OpSim run. The simple Metrics include Mean, Minimum, Maximum, Median, RMS, RobustRMS, FullRange, Percentile, and Count. These can be applied to any column in the data, making the Metrics themselves flexible and easily reusable pieces of code. A full list of included Metrics is provided in the MAF documentation, although we anticipate that users will wish to write new Metrics specific to their science.

The Slicers are perhaps less familiar. The idea is that the Slicer subdivides the larger OpSim output data into well-defined units, such as 'all visits', 'all visits overlapping a particular HealPixel RA/Dec grid point', 'all visits to a particular OpSim FieldID', or 'all visits with a particular data value within a given interval (of airmass, seeing, night, or any other column in the data)'. We can iterate through all of these subdivisions using methods defined in the Slicer, thus iterating through each Healpixel grid point or each OpSimFieldID or each interval. The Slicer defines what data is passed to a Metric.

MAF provides a set of Slicers that include:

- UniSlicer: returns a single data slice, containing all visits in the input data. This could also be thought of as the identity operator for slicing.

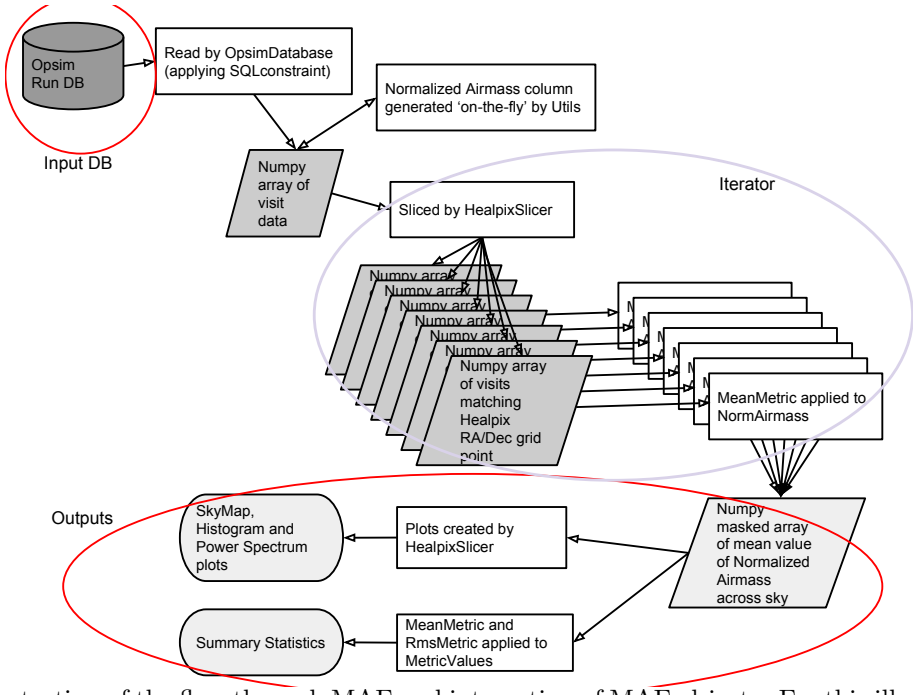

Figure 1: An illustration of the flow through MAF and interaction of MAF objects. For this illustration, we have chosen the use-case of calculating the mean value of the 'normalized airmass' across the sky. The normalized airmass is the airmass of each visit divided by the minimum airmass that the field would achieve, if it was observed at zenith, given the latitude of the telescope. The normalized airmass is not one of the columns provided by the OpSim database, so we calculate this for each visit using the MAF utility to add columns 'on-the-fly' and merge it into the numpy array describing the properties of each visit. Because we want to calculate the mean value of the normalized airmass at each RA/Dec value across the sky, we use a HealpixSlicer to determine the visits which overlap each Healpixel RA/Dec grid point - these are the 'data slices' passed to the MeanMetric. The MeanMetric is configured to operate on the 'normalized airmass' column, and returns the mean value of this column in each data slice, as the slicer iterates through all data slices. The values at each Healpixel are combined to generate the mean values of the normalized airmass across the sky. This metric data is saved to disk and used to generate visualizations of the metric data by the HealpixSlicer: a sky map, a histogram and a power spectrum plot (the types of visualizations are slicer-dependent). Summary statistics, such as the mean of all metric data values across the sky and the RMS of these values, can also be calculated.

- OneDSlicer: returns slices of data where the value of a user-specified column is within a given interval (and iterates through a series of intervals). The user can specify the intervals defining each slice directly, specify the overall number of intervals, specify the size of each interval, or let the Slicer choose the number of intervals using the Freedman-Diaconis rule. This Slicer, when combined with a Count Metric, acts as a histogram and the intervals are defined in the same manner as numpy's histogram function.
- NDSlicer: returns data slices where the values of multiple user-specified columns are within given N-dimensional interval (and iterates through a series of intervals). This is the OneDSlicer, but in N-dimensions.
- OpSimFieldSlicer: returns data slices where the FieldID matches a specified FieldID (and iterates through a set of FieldIDs). This Slicer is most useful for technical metrics involving the performance of the OpSim simulator itself.
- HealpixSlicer: returns data slices where the RA/Dec of the visit overlaps (based on a user-defined radius) the RA/Dec of the Healpix grid point (and iterates through all Healpixels at a user-defined resolution level). This Slicer allows calculation of metrics with resolution of field overlaps. The HealpixSlicer uses the HEALpix tesselation of a sphere,[?] making it possible for the user to set a spatial resolution and rapidly compute power spectra.

Each Slicer provides methods to iterate through all slices and understands the definition of the slices. For example, the HealpixSlicer understands the underlying HEALpix grid and the location ('slicePoint') of each data slice in the overall grid, and the OneDSlicer knows the data values defining the limits of each interval in its series of 'slicePoints'. Each Slicer therefore also provides methods to visualize the metric values generated by iterating through the Slicer and applying a Metric at each slice. The OneDSlicer provides methods to plot the one-dimensionally sliced metric values and the NDSlicer provides methods to plot the N-dimensional metric values along either one or two user-defined axes. The OpSimFieldSlicer and HealpixSlicer provide methods to generate sky maps of the metric values as well as histograms of the resulting metric values; the HealpixSlicer also provides a method to plot the power spectrum of the metric values.

Further under the hood, in order to do the data slicing efficiently, each Slicer also indexes the visit data according to the slice definitions. The best example of this is with the HealpixSlicer. In order to efficiently find the visits which overlap a particular Healpixel, the Slicer first builds a kd-tree on the visit RA and Dec values. Then for each Healpixel, it searches the kd-tree for visits within a specified radius (the radius of the LSST field of view) of the Healpixel RA and Dec value.

There are some additional special-use Slicers within MAF, written to help evaluate particular technical issues. Although the provided Slicers cover a wide phase-space of potential data slicing, it is also easy for users to write new Slicers to create custom subdivisions of the input data or to create new visualizations of the metric values.

To couple together Slicers and Metrics, we provide the SliceMetric class. This class provides methods to take a single Slicer object and multiple Metric objects and then iterate through the Slicer, applying the multiple Metrics at each slicePoint. It allocates and saves the metric values as numpy masked arrays and handles masking the metric values when there is either no data at a particular slicePoint or the Metric returns a flagged 'bad value'. The SliceMetric also provides convenience methods to plot all metric values, save all metric values to disk and read previously saved metric values back from disk. The SliceMetric also keeps track of the relevant metadata (e.g., which simulation is being analyzed) for each Metric + Slicer combination and adds this to the plot titles and saved files. This class is provided as a convenience for users interacting with MAF from within a python shell or from their own custom python scripts; it is also used by the MAF Driver interface, which allows users to run MAF from relatively simple configuration files.

Data comes into MAF from a database via the Database classes. The real workhorse here is the Table class, which provides the necessary tools to connect to a database table, execute queries on that table, and returns the results of the queries in a numpy recarray. The Table class depends on database tools developed for another LSST software package used to generate simulated catalogs. Underlying these tools is SQLAlchemy,[?] thus MAF's Table class can connect to any SQL database and is agnostic about the specific type.

On top of the Table class, the OpSim-specific OpsimDatabase class carries more information about the full set of database tables in the sqlite databases generated by OpSim. OpsimDatabase includes methods to connect to the various tables within the sqlite database file, fetch and parse the configuration tables used to run a particular simulated survey, retrieve the number of years of operations the simulation was intended to represent, get and identify various proposal IDs, and most importantly: fetch the records representing each visit and its observing conditions, allowing the user to specify which columns to retrieve from the database and to apply a SQL constraint to the selection of visits.

The basic flow through the framework to evaluate an OpSim simulated survey is then:

- Connect to an OpSim sqlite database file using an OpsimDatabase object.
- Instantiate the Metrics objects to be used to evaluate the simulated survey. This sets up a registry containing the columns needed from the OpSim outputs (such as airmass and seeing, etc.).
- Instantiate the Slicer to be used with these Metrics, which may add some additional columns needed from the OpSim output (such as RA and Dec).
- Cross-reference the necessary columns for the Slicer and Metrics against a MAF utility which provides definitions for the source of each column. Columns coming directly from the OpSim outputs are indicated as coming from the database. However this utility also provides a way to generate new columns 'on-the-fly' by calculating values for each visit using methods (called Stackers) which can be added to the framework.
- Retrieve the necessary data columns from the OpSim output using OpsimDatabase, limited by a user-defined SQL constraint if desired. This data is then stored as a numpy recarray in memory.
- Use the methods defined by the Stackers to generate the on-the-fly columns and add into the numpy recarray.
- Set up the Slicer to be ready for slicing, indexing the necessary information from the visit data.
- Instantiate a sliceMetric object to make it easy to couple the Metrics and Slicer, and add the Metrics and Slicer to the sliceMetric.
- Use the sliceMetric to iterate over the Slicer and apply all the Metrics at each slice, calculating all of the desired metric values.
- Use the sliceMetric to save the metric values to disk, along with relevant metadata.
- Use the sliceMetric to generate all plots for all metric value and save these to disk.
- Calculate any user-specified summary statistics: these are just Metrics which are applied to the metric values instead of the visit data. A typical usage would be to calculate the mean and RMS of metric values calculated at all points over the sky using a HealpixSlicer.

This can then be repeated for different Slicers and different SQL constraints as desired. An illustration is provided in Figure 1.

There is some subtlety to determining the boundary between the Slicer and the SQL constraint. As a simplified example, if we want to calculate the number of visits for each year of the survey, we could apply a series of SQL constraints limiting the retrieval of data from the database to visits falling within each year, and then use a UniSlicer with a Count Metric to simply count how many visits were in each year. The results would all be single, separate numbers. We could also apply no SQL constraint, select visits from all years, then use a OneDSlicer set up with slice sizes of a year, together with a Count Metric. The result would then be the same set of values, but linked by the Slicer and easily plotted as a function of year. The Slicer and a series of SQL constraints act similarly, but the Slicer allows us to keep better track of the relationship between each slice. Thus, in general, when using MAF, use SQL constraints for large subdivisions (e.g., observations in a single

filter), especially if only a single subsection of the visits is desired, and use Slicers if a series of subdivisions (and a sense of their relationship) is desired (e.g., how a Metric value changes over space or time).

The MAF Driver provides a simple way to go through this entire flow automatically. When using the Driver, the user specifies the database address, the output directory, the desired Metrics, the desired Slicers to go with these sets of Metrics, and the desired SQL constraints to apply. The Driver uses these specifications to run through the steps listed above, looping through the unique SQL constraints and different Slicers. It also saves the configuration used to run MAF, to make it easy to recreate the analysis.

The MAF Driver takes as input a single configuration file, which is itself a python script, making it easy to configure and combine large numbers of Metrics, Slicers, and SQL constraints. Thus, MAF can generate detailed reports on each OpSim simulated survey.

3. MAF APPLICATIONS

This section demonstrates the application of MAF to a particular OpSim run, 'Opsim3.61'. While we can only show a few Metrics and Slicers here, we have attempted to illustrate the power and range of MAF in these choices, as well as the flexibility in the configuration scripts for the Driver.

Let's start with a very simple analysis: suppose we want to calculate the mean and RMS of the seeing distribution for visits which were taken in r band, and then also in i band. To find the mean seeing for all visits in r band, we would use a SQL constraint to select r bands from the OpSim output database, then use a UniSlicer to sub-select all visits in the dataSlice, and then use a MeanMetric applied to the 'seeing' column. To find the RMS of the seeing in r band, it is similar but using an RmsMetric instead. In order to calculate the equivalent values in i band, we do the same but use a SQL constraint where we select i band visits instead. Note that MAF would do two queries of the database (not four), one for visits in r band and one for visits in i band.

With this translation into MAF Metrics and Slicers in mind, it is then fairly simple to write a configuration file for the Driver to generate this output. We can loop over the two filters desired, and configure the Metrics (MeanMetric and RmsMetric), configure the Slicer (linking the Metrics to the Slicer), and send this information back from our configuration script into the driver script itself. A full example driver configuration file that recreates all the plots in this section is attached in Appendix A. However, the relevant lines that set up and configure these Metrics and Slicer (including looping over the two filters) are shown in Figure 2. We find that for Opsim3.61 the results are:

- Mean seeing in r band: $0.81''$, RMS of seeing in r band: $0.21''$
- Mean seeing in i band: $0.79''$, RMS of seeing in i band: $0.20''$.

Of course, this analysis is very simple and could easily be done in any number of ways. The interesting thing is that it is very easy to write new Metrics (see Section 4) and that the database was only scanned twice to get the data to calculate these numbers.

Moving on to a slightly more complicated analysis that includes visualization; suppose we also want to generate and plot a histogram of the airmass values in r and i bands visits. Translating this to MAF Slicers and Metrics, we would again use a SQL constraint to first select visits in r and i band. Then we would use a OneDSlicer to slice on the values of the airmass column (in each band), together with a CountMetric to return the number of visits in each slice, to generate the histograms. It is worth noting that since these SQL constraints also match the ones in the UniSlicer example above (e.g., 'where filter = "r"' and 'where filter = "i"'), we will still only do two queries of the database when combining this analysis with the UniSlicer calculations. That is, for each SQL constraint in the driver configuration file, the Driver evaluates what columns are needed for all the Slicers and Metrics being run with the same constraint, and retrieves all of these columns in one query. The Driver has hooks when configuring each Metric to allow the user to customize each plot: set the title, x and y labels, x and y ranges, and colors. It also provides the option to combine the outputs from multiple OneDSlicers (such as we have done here) into one plot. The resulting combined plot is shown in Figure 3, and the additional configuration parameters for the driver configuration file are shown in Figure 4.

```
filters = ['r', 'i']
for f in filters:
    # Configure a metric that will calculate the mean of the seeing
    #  Adding the 'IdentityMetric' to the summaryStats means it will print the output to a file.
    m1 = makeMetricConfig('MeanMetric', params=['seeing'], summaryStats={'IdentityMetric':{}})
    # Configure a metric that will calculate the rms of the seeing
    m2 = makeMetricConfig('RmsMetric', params=['seeing'], summaryStats={'IdentityMetric':{}})
    # Combine these metrics with the UniSlicer and a SQL constraint based on the filter, so
    #  that we will now calculate the mean and rms of the seeing for all r band visits
    #  (and then the mean and rms of the seeing for all i band visits).
    slicer = makeSlicerConfig('UniSlicer', metricDict=makeDict(m1, m2),
                              constraints=['filter = "%s"' %(f)])
    # Add this configured slicer (carrying the metric information and the sql constraint) into a list.
    configList.append(slicer)
```

Figure 2: These are the only lines needed in a driver configuration file to calculate the mean and rms of the seeing in all visits in r and i, respectively. One for each Metric, one for the Slicer, and then a few others to loop over the different filters. Comments in-line above provide additional description.

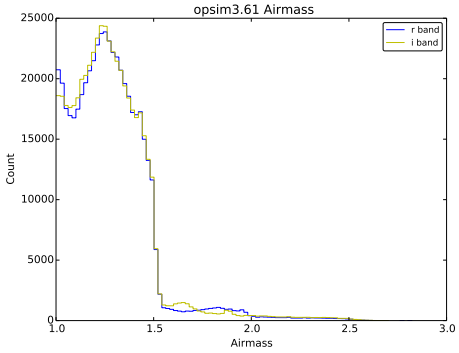

Figure 3: Example of using a OneDSlicer together with a Count Metric to generate a histogram of the airmass distribution in r and i bands, and merging the result using the driver configuration file.

```
for f in filters:
    # Configure a metric + a OneDSlicer so that we can count how many visits
    #  are within in each interval of the seeing value in the OneDSlicer.
    m1 = makeMetricConfig('CountMetric', params=['Airmass'], kwargs={'metricName':'Airmass'},
                          # Set up a additional histogram so that the outputs of these count metrics in each
                          #   filter get combined into a single plot (with both r and i band).
                          histMerge = {'histNum':1, 'legendloc':'upper right', 'label':'%s band' %(f),
                                       'xlabel':'Airmass', 'color':colors[f]})
    # Set up the OneDSlicer, including setting the interval size for slicing.
    slicer = makeSlicerConfig('OneDSlicer', kwargs={'sliceColName':'Airmass', 'slicesize':0.02},
                              metricDict=makeDict(m1), constraints=['filter = "%s"' %(f)])
    configList.append(slicer)
```

Figure 4: These configuration lines let us loop over several filters, and create a histogram of the airmass distribution in each filter. The 'histMerge' line tells the driver to merge the histograms into a single plot, which is shown in Figure 3.

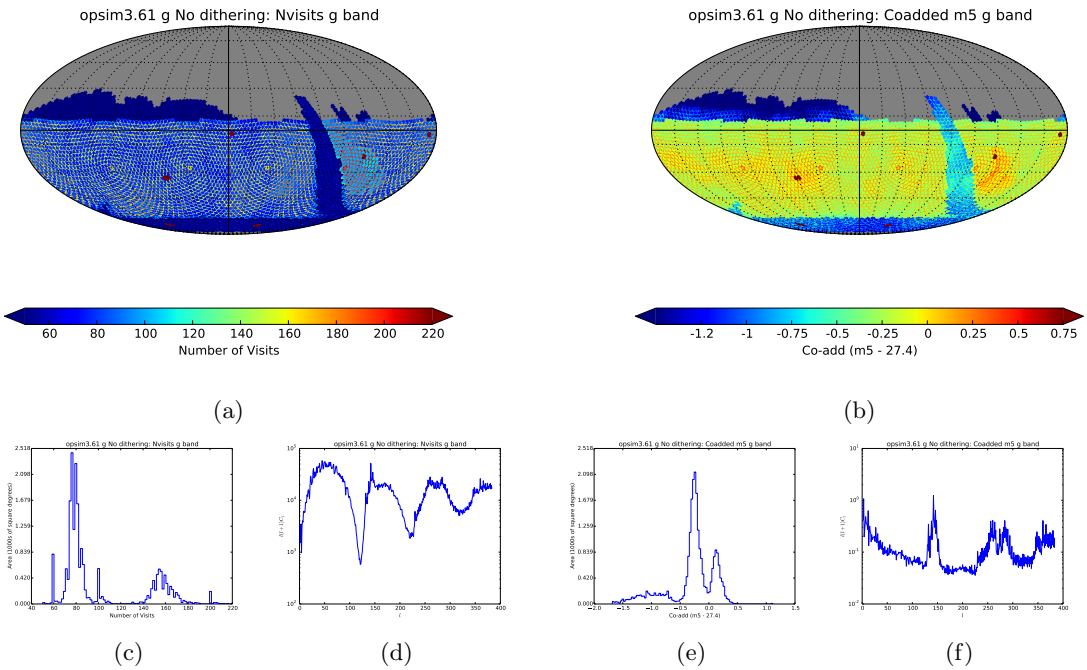

Figure 5: Example of the plots resulting from using the HealpixSlicer to calculate the number of visits per healpixel and the coadded limiting magnitude per healpixel. Panels 5a and 5b show the skymap of the number of visits and coadded depth (respectively) in g band. Panels 5c and 5d show the additional visualizations available from the HealpixSlicer: a histogram of the area corresponding to a particular number of visits, and the angular power spectrum of the number of visits across the sky. Panels 5e and 5f show these additional plots for the coadded limiting magnitude metric.

Next, let us suppose that we want to evaluate the number of visits and the coadded limiting magnitude over the sky, this time in g band, and we want to do this at a higher resolution than the size of an LSST field of view. At the time of this publication, the LSST OpSim uses fixed field pointings*, returning to the same RA and Dec for each field every time – however, these fields do overlap slightly, on the order of 100 arcminutes. By using the HealpixSlicer, with a value of NSIDE of at least 64, we can begin to resolve the field overlaps. Coupling this with a Count Metric, we can evaluate the number of visits to each RA and Dec point in the Healpix grid. We can also calculate the coadded depth at each point in the Healpix grid using the provided Coaddm5Metric, which uses the individual visit 5-sigma limiting magnitude output by OpSim and combines them according to

$$\mathrm{Coadd\,m5} = 1.25 \log_{10} \sum (10^{0.8\,\mathrm{m5_i}}). \quad (1)$$

The resulting metric values will be two arrays, containing the number of visits and the coadded m5 values evaluated at the RA/Dec point defining each healpixel. The HealpixSlicer then provides the following methods to visualize these metric values: a SkyMap (see Figures 5a and 5b for examples), a Histogram, where the area of each healpixel is used to convert the histogram counts into area on the sky (see Figures 5c and 5d), and a Power Spectrum (see Figures 5e and 5f). The HealpixSlicer uses healpy's anafast function to calculate the angular power spectrum, and then plots the resulting *cl* and *ell* values, optionally removing the spherical harmonic dipole.

We can also calculate 'summary statistics' on any calculated metric value. Summary statistics are intended to take metric values calculated across the sky or over many intervals and extract a scalar value as a 'summary'. Any Metric can be used to calculate a summary statistic, as long as it only requires a single column of data (the

*In the future, the OpSim will likely adopt a dithering pattern of some kind, which will be based on evaluations of various dither patterns by the LSST community.

```
# Configure Coadded depth metric, specifying m5 column
#  and 'metricName' (which is placed at top of plots)
m2 = makeMetricConfig('Coaddm5Metric', kwargs={'m5col':'5sigma_modified',
                                               'metricName':'Coadded m5 g band'},
                      # Specify some options for plotting - zp removes a zeropoint from plotted values
                      plotDict={'zp':mag_zpoint,
                                'plotMin':-1.5, 'plotMax':0.75,
                                'units':'Co-add (m5 - % .1f)'%mag_zpoint},
                      # Calculate summary statistics "mean" and "rms"
                      summaryStats={'MeanMetric':{}, 'RmsMetric':{}}}
```

Figure 6: Example of configuring a metric where specific plotting options are desired (in particular, here a zeropoint is subtracted from all metric values before plotting; a normalization value is also supported), and a mean and rms summary statistics are specified.

metric values). We previously calculated the number of visits and coadded depth at each Healpix in g band. In order to more easily compare many different OpSim simulated surveys, we would want to know the mean and rms of the distribution of these values across the sky. We can easily do that via the Driver, by simply adding these lines to the Metric configuration, as shown in Figure 6. For this simulation, the mean of the g band number of visits was 93.8 and the mean of the coadded depth was 26.9, while the RMS was 106.3 visits and 0.63 mags, respectively.

Thus far we have demonstrated how to evaluate properties of all visits (or a subset selected by a SQL constraint) using the UniSlicer, how to evaluate properties of visits defined by a series of intervals using the OneDSlicer, and how to evaluate properties of visits at all points across the sky (resolving field overlaps) using the HealpixSlicer, as well as how to summarize the results of these evaluations into a single scalar number using summary statistics. We showed how this can be done simply using the MAF Driver. Next, let's consider a more complicated analysis that includes the MAF utility to generate data columns 'on the fly' in order to evaluate some dithering strategies.

As mentioned above, the existing OpSim simulated surveys uses a fixed tesselation of the sky when scheduling visits. In the current OpSim simulated survey outputs, the 'fieldRA' and 'fieldDec' columns refer to these fixed field centers. The OpSim database outputs also provide an example of a dithering strategy, referred to as 'hexdither'. The hexdither strategy offsets all the original fieldRA and fieldDec values in a night by a consistent amount in RA and Dec; the size of the offset is defined by a series of vertices, packed in a triangular pattern into a hexagon inscribed in LSST field of view. The resulting RA and Dec values are recorded in the 'hexdithra' and 'hexdithdec' columns in the OpSim outputs. Figure 7 illustrates the pattern of the hexdither vertices within the LSST field of view; when all 217 vertices have been visited, the pattern repeats again from the beginning. Evaluating other potential dithering strategies is straightforward with MAF.

When using the HealpixSlicer, each data slice consists of visits overlapping the Healpix RA/Dec point. Determining the relevant visits is based on the visit 'fieldRA' and 'fieldDec' by default; however, the HealpixSlicer can also be configured to use other columns – such as the 'hexdithra' and 'hexdithdec' columns. Thus, it is extremely easy to run a set of Metrics on the original, non-dithered field pointings and then run the same set of Metrics using a HealpixSlicer configured to use the hexdithered pointings. The same is true of any RA/Dec-like columns which are added straight into the OpSim database file. However, MAF also provides users the capability of adding new columns to the OpSim visit data after the data has been queried from the database. We can use this to evaluate dithering strategies without altering the OpSim database, instead simply writing a small amount of additional python code and adding it to the MAF utils.addCols module. Example code written to add a random RA and Dec dither to each pointing is shown in Figure 8. Using each of these different columns is simple and in the driver configuration file there is no differentiation between columns retrieved directly from the database or columns generated by utils.addCols – the name of the column desired for 'spatialkey1' and 'spatialkey2' is simply entered into the makeSlicerConfig kwargs in the driver config. This can be seen in the driver configuration file shown in Appendix A.

Previously we evaluated the number of visits and coadded depth in g band, using a HealpixSlicer. We can now repeat this evaluation, but apply each of these different dithering strategies as well. The results are shown in Figure 9. As can be seen, the peaks in the number of visits and coadded depth due to field overlaps are

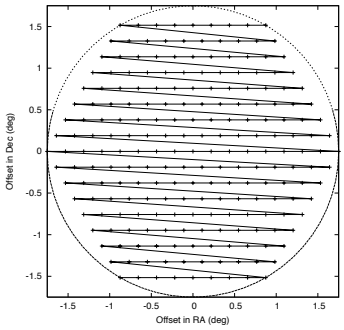

Figure 7: The hexdither dithering strategy offsets the fixed OpSim field centers by a given amount each night. The offset corresponds to the vertexes of a triangular tesselation of the hexagon inscribed in the LSST field of view. The solid line shows the progression through the vertices by night, from the lower left to the upper right. The offsets repeat after the entire pattern is completed.

dramatically smoothed by both the hexdither and random dither strategies. We can also generate histograms showing the area achieving a given number of visits and the area achieving a given coadded depth (see Panels 9g and 9h), which show that the double peaks in coadded depth corresponding to field centers vs. field overlaps are smoothed into a single peak. Similarly, we can create plots to compare the power spectra; Panel 9i shows the dramatic difference that dithering makes. In the non-dithered case, the peaks in the power spectra of the coadded depth can print through to galaxy number counts, causing problems for large scale structure cosmological analysis.[?]

To continue this evaluation of dithering strategies a little further, let us add a few additional Metrics. While there are endless possibilities, for reasons of space here we will consider only the ProperMotionMetric and the QuickRevisitMetric. The ProperMotionMetric calculates the expected final precision in the proper motion measurement at a given RA/Dec, based on the astrometric uncertainties in each observation (estimated from the user-specified star's magnitude plus the 5-sigma limiting magnitude of each visit, including an error floor) and the times of the observations (visits spread further apart in time will have a lower error in the proper motion), and assuming no parallax. The QuickRevisitMetric simply counts the number of nights that have more than a user-defined number of visits. This metric is intended to be a simple way to start looking at the effect of dithering on short time-scale revisits (as this dithering can affect the field overlap region, which can act as a source for significant numbers of quick revisits). We ran the ProperMotionMetric with stars at 20th and 24th magnitude (to evaluate high and low SNR regimes), and looked for nights with more than 10 visits with the QuickRevisitMetric. Figure 10 shows the results with each of the different dithering strategies – no dithering, hex dithering, and random dithering. At 20th magnitude, the ProperMotionMetric shows very little difference between any of these observing strategies, but once we reach 24th magnitude, it is apparent that dithering smoothes the accuracy of the proper motion measurements across the sky in a similar fashion to how dithering smoothed the number of visits and coadded depth in g band, above. However, it is only when we start to consider the QuickRevisitMetric that the differences between the hexdither and random dither start to become apparent. Looking at Panels 10g, 10h, 10i, and particularly 10l, we can see that in the case of no dithering, a small area of the sky receives a large number of nights with more than 10 visits per night but most of the sky receives much fewer than 5 nights with this many visits. Meanwhile, random dithering (where each visit is independently and randomly dithered) spreads visits around the sky, so no significant portion of sky has more than 20 nights with more than 10 visits but more of the sky gets more nights with this many revisits; i.e., the median number of nights with this many revisits is higher. Hexdither, on the other hand, because it keeps a constant offset from the original fixed field pointings for an entire night (thus preserving field overlaps within a night, while still smoothing the location of the overlaps from one night to the next), has an even higher median number of nights with more than 10 revisits. This is reflected in the summary statistics for this metric, see Table 1.

By creating Metrics tied closely to science goals, we can use MAF to analyze the effects of different dithering

```
class RandomDither(object):
    """Randomly dither the RA and Dec pointings up to maxDither degrees from center."""
    def __init__(self, raCol='fieldRA', decCol='fieldDec', maxDither=1.8, randomSeed=None):
        # Instantiate the RandomDither object and set internal variables.
        self.raCol = raCol
        self.decCol = decCol
        self.maxDither = maxDither * np.pi / 180.0
        self.randomSeed = randomSeed
        # self.units used for plot labels
        self.units = 'rad'
        # Values required for framework operation: this specifies the names of the new columns.
        self.colsAdded = ['randomRADither', 'randomDecDither']
        # Values required for framework operation: this specifies the data columns required from the database
            .
        self.colsReq = [self.raCol, self.decCol]

    def run(self, simData):
        # Generate random numbers for dither, using defined seed value if desired.
        if self.randomSeed is not None:
            np.random.seed(self.randomSeed)
        dithersRA = np.random.rand(len(simData[self.raCol]))
        dithersDec = np.random.rand(len(simData[self.decCol]))
        # np.random.rand returns numbers in [0, 1) interval.
        # Scale to desired +/- maxDither range.
        dithersRA = dithersRA*np.cos(simData[self.decCol])*2.0*self.maxDither - self.maxDither
        dithersDec = dithersDec*2.0*self.maxDither - self.maxDither
        # Add to RA and Dec and wrap back into expected range.
        randomRADither = simData[self.raCol] + dithersRA
        randomRADither = randomRADither % (2.0*np.pi)
        randomDecDither = simData[self.decCol] + dithersDec
        # Wrap dec back into +/- 90 using truncate
        randomDecDither = np.where(randomDecDither < -np.pi/2.0, -np.pi/2.0, randomDecDither)
        randomDecDither = np.where(randomDecDither > np.pi/2.0, np.pi/2.0, randomDecDither)
        stackerInput = np.core.records.fromarrays([randomRADither, randomDecDither],
                                                  names=['randomRADither', 'randomDecDither'])
        # Add the new columns into the opsim simulated survey data.
        simData = _opsimStack([simData, stackerInput])
        return simData
```

Figure 8: Code snippet written to add a random RA and Dec dither to each visit pointing. With this in place in the MAF utils.addCol module, a HealpixSlicer can be easily configured to use these 'new columns' generated on the fly within MAF by just referring to their names.

strategies in detail as above. We can also use MAF to analyze multiple OpSim simulated surveys created with different simulation parameters such as: varying the exposure time, obtaining visits in pairs (or singletons or triples), changing the footprint of the survey, varying the airmass, seeing or skybrightness limits for obtaining observations, or otherwise changing the cadence goals. This can be done simply by changing the name of the database in the driver configuration file (also settable via the command line). In this way, MAF can be used to evaluate different observing strategies.

4. WRITING A NEW METRIC

We anticipate that many users will want to write their own Metrics. The Metric class API is minimal to help accommodate this. A base class provides the framework overhead, such as creating a column registry (so that the framework can determine what columns to fetch from the database) and determining units for each column if not provided by the user (so that plots can be properly labeled). This leaves the user free to concentrate on the code relevant to their particular analysis.

Each Metric must have *__init__* and *run* methods. The *run* method is given only the data slice and slice location, applies the algorithm to calculate the metric value, and returns only that metric value for that data slice. For the simplest metrics, operating on a single column and returning a scalar value for each data slice, the user only needs to write the *run* method. An example of a simple Metric, providing only a new *run* method is shown in Figure 11. Simple Metrics like these inherit from SimpleScalarMetric, which is a subclass of the BaseMetric that provides some additional error checking about column names and returned data types.

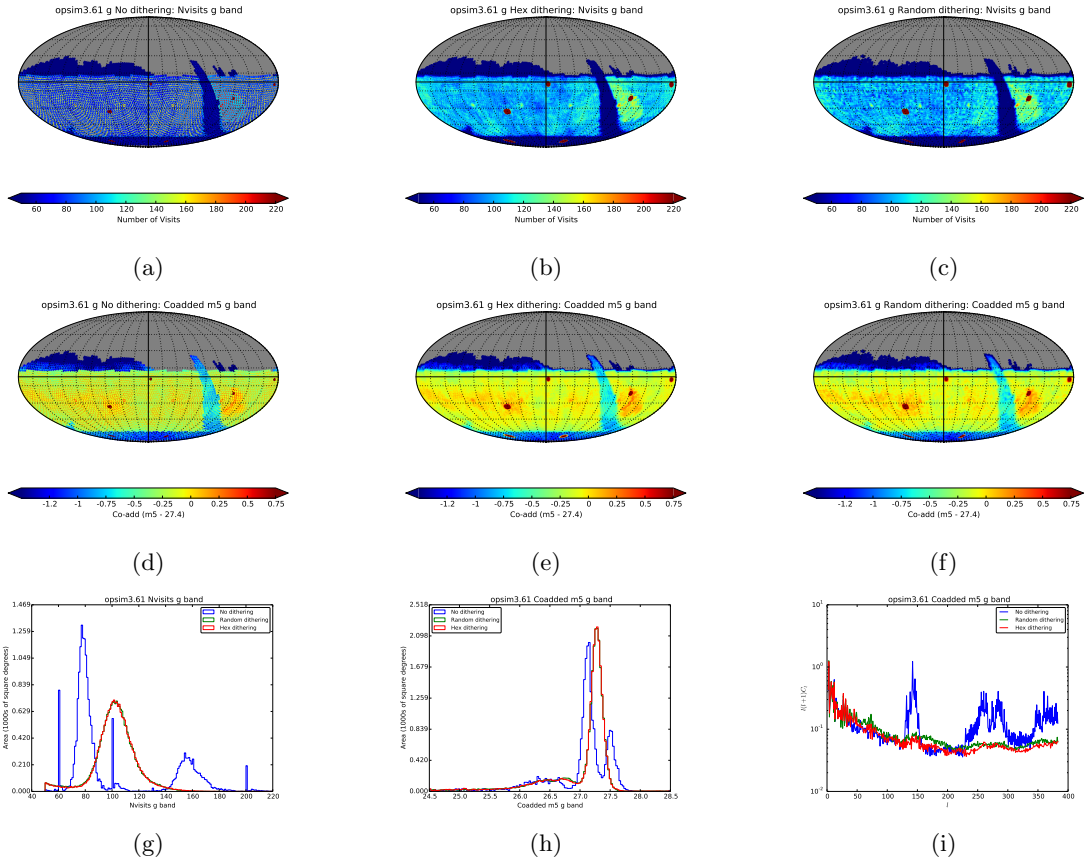

Figure 9: Evaluating the number of visits and coadded limiting magnitude in g band, in the case of no dithering (Panels 9a and 9d), hex dithering (Panels 9b and 9e) and random dithering (Panels 9c and 9f) applied to the OpSim FieldRA and FieldDec values. In the no dither and hex dither case, the relevant RA/Dec columns are provided by the OpSim database; the random dither RA/Dec columns were generated on the fly using MAF utilities. Panels 9g, 9h and 9i show the histogram and power spectrum plots created from these metric values. The areas with fewer visits (in the plane of the Milky Way, near the south celestial pole, and north of $\approx 10^{\circ}$) are areas outside the main footprint of LSST and are observed with a different set of requirements, including fewer visits. The six dark red circular areas visible in the number of visits and coadded depth plots represent the deep drilling fields, and are again observed with a different set of requirements than the main survey. Examining panels 9i, we see that dithering is extremely important for smoothing the peaks in the power spectrum of the coadded depth, which can print through to cosmological analysis such as galaxy clustering. It is also important for the coadded depth; although the mean value of the coadded limiting magnitude does not change significantly (26.98/26.99/26.98 in no dither/hex dither/random dither, respectively) the median does increase from 27.13 without dithering to 27.23 in both dithering options. (See Table 1).

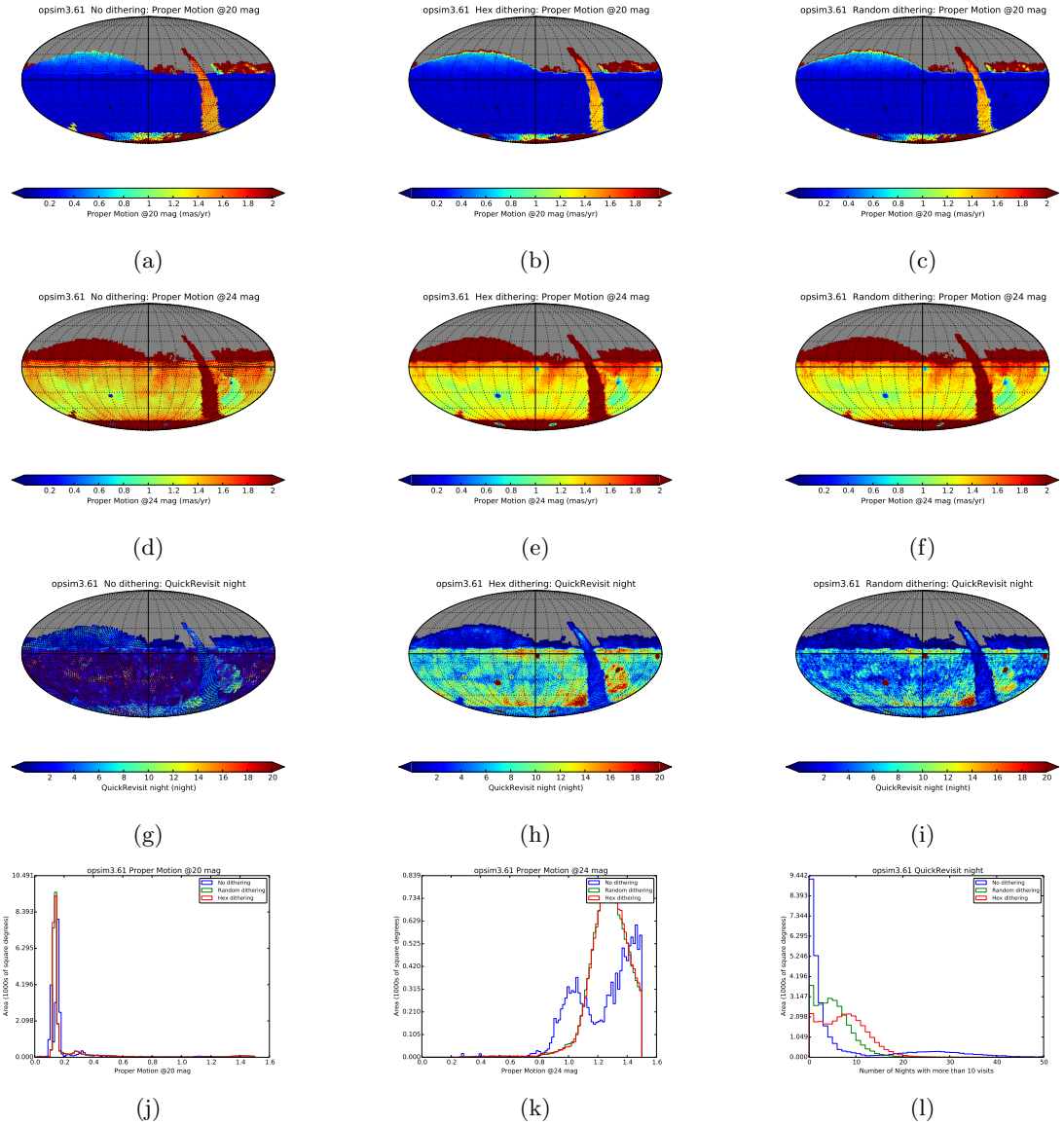

Figure 10: Analyzing the effects of dithering on proper motion measurements and the number of nights with more than 10 visits (i.e., 'QuickRevisits'). Panels 10a, 10b, and 10c show the on-sky distribution of expected errors in the measurement of proper motion for a 20th magnitude star in the case of no dithering, hex dithering and random dithering, respectively (all visits in all bands at a particular RA/Dec are used to calculate the proper motion; the Metric assumes there is no parallax to confuse the proper motion measurement). Panels 10d, 10e, and 10f show the same for a 24th magnitude star. By examining the skymaps and panel 10j, we can see that dithering is not very important for the proper motion measurement of high SNR stars, as generally each measurement is simply running up against the noise floor of the astrometric measurements (with the notable exception of the plane of the Milky Way, where fewer visits are obtained). However, as panel 10k shows, dithering is an important consideration for lower SNR stars, and improves the median error in the proper motion by a about a tenth of a mas/year, presumably due to the increase in the median number of visits and coadded depth. Panels 10g, 10h and 10i contain plots showing the result of the QuickRevisitMetric, counting the number of nights with more than 10 visits at each RA/Dec grid point. This Metric highlights some differences in the dither strategies, visible in panel 10l. Because the random dither strategy dithers each visit independently, it can decrease the number of revisits within a night by spreading these visits over a greater area and breaking field overlaps. The hex dither preserves relative field placement within a night, applying a constant offset for the pointings throughout an entire night.

Table 1: Summary statistics of metrics applied to non-dithered, hex dithered, and randomly dithered OpSim pointings.

MetricName	Metadata	Median	Mean	RMS
Nvisits r band	r No dithering	174.00	195.00	299.86
Nvisits r band	r Hex dithering	217.00	192.72	216.44
Nvisits r band	r Random dithering	217.00	190.76	215.37
Nvisits g band	g No dithering	79.00	93.80	148.39
Nvisits g band	g Hex dithering	98.00	92.17	106.27
Nvisits g band	g Random dithering	98.00	91.65	105.59
Coadded m5 g band (mag)	g No dithering	27.13	26.98	0.63
Coadded m5 g band (mag)	g Hex dithering	27.24	26.99	0.64
Coadded m5 g band (mag)	g Random dithering	27.24	26.98	0.65
Coadded m5 r band (mag)	r No dithering	27.34	27.04	0.84
Coadded m5 r band (mag)	r Hex dithering	27.45	27.07	0.81
Coadded m5 r band (mag)	r Random dithering	27.45	27.05	0.85
Proper Motion @20 mag (mas yr^{-1})	No dithering	0.16	-	-
Proper Motion @20 mag (mas yr^{-1})	Hex dithering	0.14	-	-
Proper Motion @20 mag (mas yr^{-1})	Random dithering	0.14	-	-
Proper Motion @24 mag (mas yr^{-1})	No dithering	1.54	-	-
Proper Motion @24 mag (mas yr^{-1})	Hex dithering	1.42	-	-
Proper Motion @24 mag (mas yr^{-1})	Random dithering	1.42	-	-
QuickRevisit (# nights)	No dithering	1.00	7.84	17.81
QuickRevisit (# nights)	Hex dithering	7.00	7.70	10.49
QuickRevisit (# nights)	Random dithering	4.00	6.36	20.42

```
class RobustRmsMetric(SimpleScalarMetric):
    """
    Use the inter-quartile range of the data to estimate the RMS.
    Robust since this calculation does not include outliers in the
    distribution.
    """
    def run(self, dataSlice, *args):
        iqr = np.percentile(dataSlice[self.colname], 75) -
              np.percentile(dataSlice[self.colname], 25)
        rms = iqr/1.349 #approximation
        return rms
```

Figure 11: Example of writing a new simple metric, operating on a single data column and returning a scalar value for each data slice. The class inherits from SimpleScalarMetric and uses this *__init__* method, but provides its own *run* method.

```
class PercentileMetric(SimpleScalarMetric):
    def __init__(self, colname, percentile=90, **kwargs):
        super(PercentileMetric, self).__init__(colname, **kwargs)
        self.percentile = percentile
    def run(self, dataSlice, *args):
        return np.percentile(dataSlice[self.colname], self.percentile)
```

Figure 12: Example of writing a new simple metric that includes a run-time configurable parameter (the percentile value). The class inherits from the SimpleScalarMetric, but extends the *__init__* method, while still providing its own *run* method.

By also extending the SimpleScalarMetric's *__init__* method using Python's *super*, users can provide configurable parameters to each metric at run time or add their own column definitions. An example is the PercentileMetric shown in Figure 12.

More complicated Metrics, such as those operating on more than one column or returning a complex data type (such as a dictionary or an array), inherit directly from the BaseMetric. In addition, those returning a complex data type can add 'reduce' methods, intended to take that complex data value and turn it into a scalar for each slice. This is especially useful for Metrics where a computationally expensive value must be computed but then could be interpreted in multiple ways to produce scalar values at each slicePoint for visualization. First the Metric *run* is called for each data slice; then all *reduce* methods are called for each data slice (but passed the metric value calculated by the *run* method at that point).

An example is the VisitGroupsMetric: this metric is intended to examine how visits are paired together as a preliminary exploration of how well an OpSim simulated survey might perform for solar system object discovery. First the metric calculates the number of visits within a given time interval on each night, for each data slice, returning a dictionary containing the nights that multiple visits were achieved and the total number of visits on each of those nights that were within the time interval. Then different *reduce* methods operate on each metric value (corresponding to each original data slice), calculating the median number of visits, the number of times that more than a threshold number of visits occurred within some larger time interval of number of nights, the number of lunations that received more than a threshold number of nights with multiple visits, and the number of successive lunations that received more than a threshold number of nights with multiple visits. The reduce methods provide multiple views into the underlying question (how visits are paired together for discovering solar system objects) and are all desirable; however, the first step is somewhat computationally expensive. By allowing multiple reduce methods, we can take the original data slice and end up examining it in multiple ways with minimal cost. Code for the VisitGroupsMetric can be found online in the LSST Stash repository at http://ls.st/rkw (shortened link), but excerpts illustrating the *run* and *reduce* methods are shown in Figure 13.

More information on writing and using your own metrics is available in the MAF documentation.

5. CONCLUSIONS

MAF has been designed to provide an easy to use, easy to extend framework to analyze LSST OpSim simulated surveys. It is a powerful tool for evaluating the effects of observing strategies, including dithering strategies. By collaborating with the wider LSST community to create new Metrics to cover a wide range of science cases, we hope to discover optimal strategies for the scheduling of LSST quantify both the overall efficiency of the surveys and their scientific potential.

Beyond analyzing LSST OpSim outputs, MAF can be applied to any dataset. Much of the necessary framework requirements are available in base classes (as well as an illustration of the required API), which can be extended for use with a particular dataset. For example, if a user writes a custom Metric and uses that with one of MAF's Slicers (or writes their own Slicer for a specific purpose), but then wants to analyze observing histories from other telescopes instead of (or as well as) LSST OpSim data, all that is required is a new Database class to access the new pointing history.

We intend to continue development on MAF, in particular improving the presentation of MAF outputs using a web interface. In addition, we welcome community contributions to MAF, including new Metrics, Slicers, or Database classes.

```
class VisitGroupsMetric(BaseMetric):
    """Count the number of visits per night within deltaTmin and deltaTmax."""
    def __init__(self, timesCol='expMJD', nightsCol='night',
                 deltaTmin=15.0/60.0/24.0, deltaTmax=90.0/60.0/24.0, minNVisits=2, window=30, minNNights=3,
                 **kwargs):
... <snip> ... # Removed lines setting internal variables to passed values
        super(VisitGroupsMetric, self).__init__([self.times, self.nights], **kwargs)

    def run(self, dataSlice):
        """
        Return a dictionary of:
         the number of visits within a night (within delta tmin/tmax of another visit),
         and the nights with visits > minNVisits.
        Count two visits which are within tmin of each other, but which have another visit
         within tmin/tmax interval, as one and a half (instead of two).
        """
        uniquenights = np.unique(dataSlice[self.nights])
        nights = []
        visitNum = []
 ... <snip> ... # Removed lines with details of calculation
        metricval = {'visits':visitNum, 'nights':nights}
        if len(visitNum) == 0:
            return self.badval
        return metricval

    def reduceMedian(self, metricval):
        """Reduce to median number of visits per night (2 visits = 1 pair)."""
        return np.median(metricval['visits'])

    def reduceNNightsWithNVisits(self, metricval):
        """Reduce to total number of nights with more than 'minNVisits' visits."""
        condition = (metricval['visits'] >= self.minNVisits)
        return len(metricval['visits'][condition])

    def reduceNVisitsInWindow(self, metricval):
        """Reduce to max number of total visits on all nights with more than minNVisits, within any 'window'
            (default=30 nights)."""
        maxnvisits = 0
        for n in metricval['nights']:
            vw, nw = self._inWindow(metricval['visits'], metricval['nights'], n, self.window, self.minNVisits
                )
            maxnvisits = max((vw.sum(), maxnvisits))
        return maxnvisits
```

Figure 13: Example of a more complex metric that returns a dictionary from the *run* method and then provides several *reduce* methods to evaluate different aspects of those calculated values. Both the *run* method and the *reduce* methods are called for each slice in the Slicer.

APPENDIX A. DRIVER CONFIGURATION FILE

```
from lsst.sims.maf.driver.mafConfig import makeBinnerConfig, makeMetricConfig, makeDict
import lsst.sims.maf.utils as utils

# Configure the output directory.
root.outputDir = './MAFOut'

# Configure our database access information.
root.dbAddress = {'dbAddress':'sqlite:///opsim3_61.db'}
root.opsimName = 'opsim3_61'

# Some parameters to help control the plotting ranges below.
nVisits_plotRange = {'g':[50,220], 'r':[150, 310]}

# Use a MAF utility to determine the expected design and stretch goals for number of visits, etc.
design, stretch = utils.scaleStretchDesign(10)
mag_zpoints = design['coaddedDepth']

# Some parameters we're using below.
nside = 128
nvisits = 10

# A list to save the configuration parameters for our Slicers + Metrics.
configList=[]

# Loop through r and i filters, to do some simple analysis.
filters = ['r', 'i']
colors = {'r':'b', 'i':'y'}
for f in filters:
    # Configure a metric that will calculate the mean of the seeing
    #  Adding the 'IdentityMetric' to the summaryStats means it will print the output to a file.
    m1 = makeMetricConfig('MeanMetric', params=['seeing'], summaryStats={'IdentityMetric':{}})
    # Configure a metric that will calculate the rms of the seeing
    m2 = makeMetricConfig('RmsMetric', params=['seeing'], summaryStats={'IdentityMetric':{}})
    # Combine these metrics with the UniSlicer and a SQL constraint based on the filter, so
    #  that we will now calculate the mean and rms of the seeing for all r band visits
    #  (and then the mean and rms of the seeing for all i band visits).
    slicer = makeSlicerConfig('UniSlicer', metricDict=makeDict(m1, m2),
                              constraints=['filter = "%s"' %(f)])
    # Add this configured slicer (carrying the metric information and the sql constraint) into a list.
    configList.append(slicer)
    # Configure a metric + a OneDSlicer so that we can count how many visits
    #  are within in each interval of the seeing value in the OneDSlicer.
    m1 = makeMetricConfig('CountMetric', params=['Airmass'], kwargs={'metricName':'Airmass'},
                          # Set up a additional histogram so that the outputs of these count metrics in each
                          #   filter get combined into a single plot (with both r and i band).
                          histMerge = {'histNum':1, 'legendloc':'upper right', 'label':'%s band' %(f),
                                       'xlabel':'Airmass', 'color':colors[f]})
    # Set up the OneDSlicer, including setting the interval size for slicing.
    slicer = makeSlicerConfig('OneDSlicer', kwargs={'sliceColName':'Airmass', 'slicesize':0.02},
                              metricDict=makeDict(m1), constraints=['filter = "%s"' %(f)])
    configList.append(slicer)

# Loop through the different dither options.
dithlabels = [' No dithering', ' Random dithering', ' Hex dithering']
slicerNames = ['HealpixSlicer' ,'HealpixSlicerRandom', 'HealpixSlicerDither']
for dithlabel, slicerName in zip(dithlabels, slicerNames):
    # Set up parameters for slicer, depending on what dither pattern we're using.
    if slicerName == 'HealpixSlicer':
        slicerName = 'HealpixSlicer'
        slicerkwargs = {'nside':nside}
        slicermetadata = 'No dither'
    elif slicerName == 'HealpixSlicerDither':
        slicerName = 'HealpixSlicer'
        slicerkwargs = {'nside':nside, 'spatialkey1':'hexdithra', 'spatialkey2':'hexdithdec'}
        slicermetadata = 'Hex dither'
    elif slicerName == 'HealpixSlicerRandom':
        slicerName = 'HealpixSlicer'
        slicerkwargs = {'nside':nside, 'spatialkey1':'randomRADither', 'spatialkey2':'randomDecDither'}
        slicermetadata = 'Random dither'
    # Configure QuickRevisit metric to count number times we have more than X visits within a night.
    m1 = makeMetricConfig('QuickRevisitMetric', kwargs={'nVisitsInNight':nvisits},
                          plotDict={'plotMin':0, 'plotMax':20, 'histMin':0, 'histMax':100},
                          summaryStats={'MeanMetric':{}, 'RmsMetric':{},'MedianMetric':{}},
                          # Add it to a 'merged' histogram, which will combine metric values from
                          #  all dither patterns.
                          histMerge={'histNum':2, 'legendloc':'upper right', 'label':dithlabel,
                                     'histMin':0, 'histMax':50,
```

```
                                'xlabel':'Number of Nights with more than %d visits' %(nvisits),
                                'bins':50})
    # Configure Proper motion metric to analyze expected proper motion accuracy.
    m2 = makeMetricConfig('ProperMotionMetric', kwargs={'m5Col':'5sigma_modified', 'seeingCol':'seeing',
                                                        'metricName':'Proper Motion @20 mag'},
                          plotDict={'percentileClip':95, 'plotMin':0, 'plotMax':2.0},
                          histMerge={'histNum':3, 'legendloc':'upper right', 'label':dithlabel,
                                     'histMin':0, 'histMax':1.5})
    # Configure another proper motion metric where input star is r=24 rather than r=20.
    m3 = makeMetricConfig('ProperMotionMetric', kwargs={'m5Col':'5sigma_modified', 'seeingCol':'seeing',
                                                        'rmag':24, 'metricName':'Proper Motion @24 mag'},
                          plotDict={'percentileClip':95, 'plotMin':0, 'plotMax':2.0},
                          histMerge={'histNum':4, 'legendloc':'upper right', 'label':dithlabel,
                                     'histMin':0, 'histMax':1.5})
    # Configure a Healpix slicer that uses either the opsim original pointing, or one of the dithered RA/Dec
        values.
    slicer = makeSlicerConfig(slicerName, kwargs=slicerkwargs,
                              metricDict=makeDict(m1, m2, m3), constraints=[''], metadata = dithlabel)
    # Add this configured slicer (which carries the metrics and sql constraints with it) to our list.
    configList.append(slicer)

    # Loop through filters g and r and calculate number of visits and coadded depth in these filters
    for f in ['g', 'r']:
        # Reset histNum to be 5 or 6 depending on filter
        #   (so same filter, different dithered slicers end in same merged histogram)
        if f == 'g':
            histNum = 5
        elif f == 'r':
            histNum = 7
        # Configure a metric to count the number of visits in this band.
        m1 = makeMetricConfig('CountMetric', params=['expMJD'], kwargs={'metricName':'Nvisits %s band' %(f)},
                              plotDict={'units':'Number of Visits', 'cbarFormat':'%d',
                                        'plotMin':nVisits_plotRange[f][0],
                                        'plotMax':nVisits_plotRange[f][1],
                                        'histMin':nVisits_plotRange[f][0],
                                        'histMax':nVisits_plotRange[f][1]},
                                summaryStats={'MeanMetric':{}, 'RmsMetric':{}},
                                histMerge={'histNum':histNum, 'legendloc':'upper right',
                                           'histMin':nVisits_plotRange[f][0],
                                           'histMax':nVisits_plotRange[f][1],
                                           'label':dithlabel,
                                           'bins':(nVisits_plotRange[f][1]-nVisits_plotRange[f][0])})
        histNum += 1
        # Configure a metric to count the coadded m5 depth in this band.
        m2 = makeMetricConfig('Coaddm5Metric', kwargs={'m5col':'5sigma_modified',
                                                       'metricName':'Coadded m5 %s band' %(f)},
                                plotDict={'zp':mag_zpoints[f],
                                          'percentileClip':95., 'plotMin':-1.5, 'plotMax':0.75,
                                          'units':'Co-add (m5 - %.1f)'%mag_zpoints[f]},
                                summaryStats={'MeanMetric':{}, 'RmsMetric':{}},
                                histMerge={'histNum':histNum, 'legendloc':'upper left', 'bins':150,
                                           'label':dithlabel, 'histMin':24.5, 'histMax':28.5})
        # Configure the slicer for these two metrics
        #  (separate from healpix slicer above because of filter constraint).
        slicer = makeSlicerConfig(slicerName, kwargs=slicerkwargs,
                            metricDict=makeDict(m1, m2), constraints=['filter="%s"'%(f)],
                            metadata = dithlabel)
        configList.append(slicer)

root.slicers=makeDict(*configList)
```

ACKNOWLEDGMENTS

We would like to thank Alex Kim, Michael Woods-Vasey, and Phil Marshall for participating in beta testing and providing feedback on early documentation for MAF.

LSST project activities are supported in part by the National Science Foundation through Cooperative Support Agreement (CSA) Award No. AST-1227061 under Governing Cooperative Agreement 1258333 managed by the Association of Universities for Research in Astronomy (AURA), and the Department of Energy under contract with the SLAC National Accelerator Laboratory. Additional LSST funding comes from private donations, grants to universities, and in-kind support from LSSTC Institutional Members.