Materia: A Data Quality Control Embedded Domain Specific Language in Python

Connor Scully-Allison

University of Arizona, Tucson AZ, USA
`cscullyallison@email.arizona.edu`

Abstract. Current solutions for data quality control (QC) in the environmental sciences are locked within propriety platforms or reliant on specialized software. This can pose a problem for data users when attempting to integrate QC into their existing workflows. To address this limitation, we developed an embedded domain specific language (EDSL), Materia, that provides functions, data structures, and a fluent syntax for defining and executing quality control tests on data. Materia enables developers to more easily integrate QC into complex data pipelines and makes QC more accessible for students and citizen scientists. We evaluate Materia via two metrics: productivity and a quantitative performance analysis. Our productivity examples show how Materia can simplify complex descriptions of tests in Pandas and mirror natural language descriptions of common QC tests. We also demonstrate that Materia achieves satisfactory performance with over 200,000 floating-point values processed in under three seconds.

1 Introduction

Quality Control (QC) on scientific data is a critical step in the data preprocessing pipeline and is essential to performing good science. This "QC process" describes any systematic approach undertaken by data experts to check for errors in datasets. It can be done manually or automatically. In the manual case an expert scientist scans through columns of records and notes any problems they find. For example, they may discount a temperature value for being too cold for the season or when visualizing data they will see that a value spikes up dramatically compared to its preceding measurements, or drifts over time.

If these anomalies are not recorded and properly associated with their corresponding dataset, the utility of collected scientific data is significantly limited. Errenous values in a dataset do not accurately reflect the ground truth they are supposed to measure and, accordingly, cannot be used in scientific models to enhance our understanding of natural phenomena. In recent years, automation has transformed both the data collection and QC process but has not fully solved the QC problem [11, 12].

Increased automation in the data collection process has introduced novel vectors for the propagation of data errors. Old sensors, dying batteries, poor wiring, network failures, and more, can contribute to a missed data point or an incorrect value being logged. In the environmental sciences, these opportunities for failure are compounded by the relative remoteness and wild characteristics that pervade data collection sites [2]. Everything from weather events to wild animals can cause sensors to log incorrect data or stop data feeds altogether.

To combat this problem of quality issues introduced by autonomous data collection, a handful of technologies have been devised and deployed into modern workflows to provide quality control. Among them are the GCE Toolbox [7], Loggernet [9], ArcGIS [4], and Pandas [8]. The GCE Toolbox was produced to provide a dedicated toolkit for managing QC processes in the earth and environmental sciences. Loggernet's support software and ArcGIS provide tools and modules which enable users to perform limited Quality Control on data. Pandas, by contrast, is a general purpose statistics and data analytics library embedded in Python which was not built for data Quality Control but provides functionality which supports QC tasks very well, like binding metadata to data frames and allowing users to define time-series indices.

Together these libraries, software packages and frameworks give data managers a variety of options to choose from for their QC needs. Unfortunately, they have drawbacks that hinder their utility to specific individuals or prevent easy integration into existing data collection workflows. For the GCE Toolbox, Loggernet and ArcGIS, each of these solutions are reliant on proprietary or expensive software to run. For the GCE Toolbox, a subscription to the Matlab language is required. The ArcGIS software requires a licence costing a minimum of 200 dollars per year. Loggernet QC solutions only work with Campbell Scientific sensors: which are expensive, enterprise-level instruments used for collecting data, and are often eschewed for cheaper "iButton" devices, which are used in projects as prominent as the NSF funded McMurdo Dry Valleys Long Term Ecological Research Network (LTER) in Antarctica [1].

```
dfcopy['repeat-ids'] = (dfcopy[self.column] !=
                        dfcopy[self.column].shift(1))
                        .cumsum()
counts = dfcopy[['repeat-ids',self.column]]
            .groupby(['repeat-ids']).agg('count')
counts_less = counts.loc[counts[self.column] > 1]
dfcopy = dfcopy
            .join(counts_less,
                  on='repeat-ids',
                  lsuffix='_caller',
                  rsuffix='_other')
dfcopy[self.column] = dfcopy[self.column + '_other']
                           .map(lambda x: x >= 3)
```

Listing 1.1. Code which checks for repeat values in Pandas and returns a Boolean array indicating if a value is a repeat or not if it exceeds the repeat threshold of "3".

Pandas does not suffer from these limitations being a free, open-source library implemented in a free programming language: Python. Instead, Pandas' problem is one of domain expressiveness. The abstractions provided in that library do not always map well to defining tests in the quality-control space. An example of such a mapping can be seen in Listing 1.1, which depicts a test that checks for values that do not vary across multiple time steps. This implementation, while efficient, can be difficult to parse even as an experienced coder, making it that much more difficult for a domain scientist.

To fill these gaps which are unmet by existing solutions we developed Materia: a domain specific language embedded in Python. This language provides several data structures which simplifies the development effort required to read in tabular time-series data, manage quality control flags and perform the logical comparisons required for QC workflows. Furthermore, Materia leverages Python's built in method chaining to support a fluent syntax for managing tests. Finally, Materia provides a functional syntax which abstracts away iterative operations and enables tests to be defined over an "arbitrary" value at a single point in time. And, although the default assumption is that a function will be testing a single value, these tests also keep track of the context around a single value enabling more complex operations on one series or multiple.

The remainder of this paper is organized as follows: related and prior work are presented in Section 2; design and implementation details on Materia are presented in Section 3; evaluations of Materia and discussions of results are presented in Section 4. Finally, conclusions and future work are presented in Section 5.

2 Related Work

Relevant to the motivating problem of this research, there are several papers that discuss the importance of data quality control in the environmental sciences. Recent scholarship on this topic has emphasized an increased need for programmatic solutions to this problem. In the paper "Quantity is Nothing Without Quality: Automated QA/QC Streaming for Environmental Sensor Data," the motivation and execution of automateable quality control processes are discussed at length[2]. This paper informs domain expectations of general quality control processes and helps circumscribe the domain problem space.

In addition to this, several other works in the environmental science domain describe what standards are expected from quality controlled data [14, 6]. Specifically, the handbook by Gouldman et al. provides examples of flag codes in use by the National Oceanic and Atmospheric Administration (NOAA). This resource exemplifies what a QC-focused programming language should support. It also details many types of automated tests which are standard for their organization: "Rate of change in time", "Spike Test", "Regional Range Test", "Stuck Value Test". These tests map directly to several of the tests we will evaluate for performance and brevity, and reinforces the validity of our chosen tests.

The development of the GCE toolkit, which represents the closest prior implementation of an EDSL for QC is detailed in Sheldon et al [13]. In this work, Sheldon provides implementation and design details for the GCE toolkit which were leveraged to build up Materia. Specifically, the Dataset data structure used in Materia can be thought of as a simplified version of the GCE toolkit's "Dataset" object with some implementation features drawn from Pandas. This work also provides context about the use and impact of the GCE toolkit in the earth science domain and provides insight into the specific communities which could benefit from Materia. Sheldon indicates that it is used by various Long Term Ecological Research (LTER) sites, the United States Geologic Survey (USGS), and by GCE itself in addition to many others.

In addition to the above paper, the paper associated with the Pandas library was also mined for insight on how to construct an efficient data management/-analytics library in Python [8]. This work helped us understand how to organize the columnar data in our datasets and use numpy arrays most efficiently. Specifically, we organized our arrays into numpy matrices with shared types mirroring their "block manager" and provide support for "label based data access." Using Pandas as a reference in this way enables us to build upon state-of-the-art work and provide interfaces people are familiar with.

The concept of an embedded domain specific language (EDSL) has been steadily growing in attention and scholarship over the past two decades. For an overview of embedded domain specific languages, see Gill [5]. This classification describes programming languages built within and using an existing programming language. This construct allows for ease of development on the part of the language developer and allows language users to use syntax and supporting libraries which are familiar to them when working on domain problems.

3 Materia

Materia is a domain specific language (DSL) shallowly embedded in Python. The term, "shallowly-embedded," means that it extends the host programming language with additional functionality and makes-use of language provided constructs. It provides data structures, an abstraction which removes loop-definition clutter from function definitions, and a fluent syntax. Together this functionality enables users to quickly define and execute QC tests on tabular, time-series data. In addition to these features, Materia was designed to work with a wide variety of datasets and arbitrary flag codes to enhance its utility to various organizations. The following subsections will describe in detail these specific aspects of Materia. For more details on the usage and implementation of Materia, source code and documentation can be found at [10].

3.1 Data Structures

Materia is comprised of 2 key data structures: a DataSet and a TimeSeries. Both of these structures were designed to mitigate overhead of array creation and management, flag storage and data alignment on the part of the user.

The first data structure, a DataSet, can be thought of as a lightweight Pandas dataframe. Compared to it's bulkier cousin, it provides some domain-specific functionality which the dataframe lacks. First, it stores header metadata found in many tabular data files in a dedicated row-major matrix. This header metadata can be used for reference at any point in the development of a quality control script and can also be used to produce unique and semantically meaningful header names for label-based array access of specific data columns. An abridged example of a tabular data file which would work with Materia can be seen in Table 1.

Second, a DataSet operates under the assumption that data will be indexed by a series of datetime values. Accordingly, it performs automatic detection and conversion of any datetime columns found in the provided data file. It also automatically sets one of these columns as the primary index for the DataSet. This timeseries index is used in Quality Control tests to find missing values and align time series for comparison when they have the same temporal range but may not have the same number of array elements. All vectors in the DataSet object are are numpy arrays and are stored in column major format.

To provide a simple and familiar interface for extracting an individual TimeSeries object from a DataSet, the DataSet class overloads the "[]" operators to enable label-based access of individual columns. The returned TimeSeries object manages the execution of tests and storage of resultant flags. A time series object is comprised of three numpy arrays: the global time series index which was defined at the DataSet level, the values array for this series and an array which keeps track of quality flags resulting from tests. The TimeSeries object tracks what tests were run on it and other related metadata like its label in the DataSet and flag codes which were originally defined over the DataSet object.

Site Name:	Rockland Summit	Rockland Summit
Deployment:	Air temperature	Air temperature
Monitored System:	Climate	Climate
Measured Property:	Temperature	Temperature
Vertical Offset from Surface:		
Units:	degC	degC
Measurement Type:	Maximum	Minimum
Measurement Interval:	00:01:00	00:01:00
Time Stamp (UTC-08:00)		
2019-04-01T16:41:00.0000000-08:00	-9999	5.235
2019-04-01T16:42:00.0000000-08:00	5.124	4.97
2019-04-01T16:44:00.0000000-08:00	5.165	5.046

Table 1. An example of a tabular dataset, downloaded from the Nevada Research Data Center. Materia was designed for datasets like this one.

3.2 Managing Flag Codes

Contrary to some popular conceptions of the purpose of Quality Control, the main output of the initial phases of a quality control process is not repaired data. Instead, it is only metadata. This metadata comes in the form of "flags" – codes that indicate to a data user what the quality of particular values in a dataset may be. Flags are typically stored in an array. They are aligned alongside values in a tabular format making them easy to cross-reference when managing data.

While flags are a simple concept they can sometimes be difficult to work with in practice. This difficulty is due to the general standardization problem in scientific organizations. Like many other aspects of data, flags are not standardized across organizations, with some orgs using string values to convey the quality of their data (“bad,” “good,” “suspect”) while others use integers (0,1,2) [3]. This diversity of flag codes requires that Materia support flag standards which have differing numbers of flags, datatypes and names. In order to accommodate this plurality, Materia provides a function which allows users to affix a dictionary of their own flags to a DataSet object, visible in listing 1.2.

```
DataSet.flagcodes()
        .are({
        "None":"OK",
        "Repeat Value":"Repeat Value",
        "Missing Value": "Missing",
        "Range": "Exceeds Range",
        "SI": "Incosistent (Spatial)",
        "LI": "Inconsistent (Logical)",
        "Sp": "Spike"
        })
```

Listing 1.2. How to define flag codes on a Materia DataSet.

After setting this dict on our dataset, subsequently extracted TimeSeries objects keep track of these key-value pairs so that users need only provide the key when executing a quality control test. An example of this can be seen in listing 1.5. By mapping the flag codes to strings in this way, Materia allows users to define their own keys for flags which enables them to access them in a more succinct way. Users can use ”SI” instead of ”Inconsistent (Spatial).”

3.3 Defining Tests

```
def rv_test(value):
    n = 3
    if not value.isnan():
        if value == value.prior(n):
            return True
        return False
```

Listing 1.3. A QC test definition to check for multiple repeat values in a row using the Materia language.

The Materia language itself was designed around a functional paradigm, whereby users can define tests which operate on an abstract datapoint. Instead of developing functions which operate on specific arrays of values or contain internal iterations, users instead define Python functions with a single argument: “value.” This argument represents a single datapoint in an arbitrary time series. This abstraction enables users to define tests in fundamental terms which mirror real test specifications. An example of a test definition using this syntax can be seen in Listing 1.3.

In this example, we can see how a series agnostic test definition syntax supports natural expression of a data quality control test. Instead of working with whole vectors or iterating over a vector passed in as an argument, we are instead expressing our repeat value test as "If a value is equal to the prior 3 values in this series, return that it failed the test (true)." We also see how the "value" argument provides significant functionality to support fluent test definitions.

```
def spatial_inconsistency(value):
    comp_val = series_max_10.value().at(value)
    diff = value - comp_val
    avg = (value + comp_val) / 2
    threshold_p = 75
    exceeds_threshold = (abs(diff/avg) * 100.0 >
                              threshold_p)
    if exceeds_threshold:
        return True
    return False
```

Listing 1.4. A QC test definition in Materia which checks if the values within two related time series diverge beyond a specific threshold. series_max_10 refers to a time series pulled from the dataset prior to this function definition.

"Value" is itself an object which contains a scalar by default but can contain a vector of values. In order to support comparisons between vectors and scalars and maintain fluent test definition syntax it overloads the following binary conditional operators: "==," "! =," "<=," ">=," ">=," ">," "<". As can be seen with the "value.prior(n)" call, this object also keeps track of the context of the time series from which it was called. As many tests need to support comparisons between values preceding or following individual data points in a time series, its essential that Materia's test definition function provide an interface for retrieving contextually relevant data points.

```
series.datapoint()
        .flag('Repeat Value')
        .when(rv_test)
```

Listing 1.5. How to execute a test on a time series object in Materia.

One further example of Materia's test definition syntax can be seen in Listing 1.4. In this test definition, we can see that our "value" abstraction also overloads mathematical operators to support binary operations between our value object, numeric constants, and other value objects. In order to support a diverse array of mathematical operations on various data types, these overloaded binary operators support standard operations between individual scalars and heterogeneous operations between vectors and scalars. Finally, we also see in Listing 1.4, that a method is provided by time series objects which allows us to pass in a particular value and get out a temporally aligned value from that series. This method enables enables users to define tests between related series of values within the same temporal range.

3.4 Calling Tests and the when() function

In order to execute these tests within the context of a particular set of values, each TimeSeries object provides several methods which are chained together. These methods execute the test which is passed to "when()" as an argument and affix flags depending on the results of the provided tests. The syntax of these methods can be seen in listing 1.5.

This part of Materia implements a fluent syntax through chained method calls. This is done so that a test execution can mirror a natural language statement: "Flag a datapoint in [this] series with 'repeat value' when the repeat value test fails." This syntax for calling tests in Materia further reinforces the mental model that tests are being called on singular data points in our TimeSeries.

In listing 1.5, we can see how the when() method operates from a user perspective; specifically, it is just a higher-order function which executes the "rv_test" argument passed to it. Internally, this when() function performs several tasks to execute passed tests and manage their results.

Immediately upon being called, the "when" function uses the Python library, inspect, for method reflection. Using getsource(), it stores the source code of the passed function so that this information can be later used to provide additional context to our flags. Second, it uses getargspec() to determine if an optional iterator argument was declared as part of our test definition. If there was, "when" will invoke the function with two parameters and not one. This secondary argument can be used for debugging or checking specific errors in the provided dataset.

After performing these reflective operations, "when" then declares a loop over the internal values array. At each iteration of this loop, a new "Value" object is created and passed in as an argument to the provided test function. This Value object is constructed with the contextual information of its surrounding values in the calling TimeSeries, its offset from the beginning of the array and its datetime index. By managing our loop inside of the "when" function and providing users with a robust object for testing values this simplifies test definitions significantly.

4 Evaluation and Discussion

Materia was formally evaluated within the scope of two categories which are commonly used to evaluate EDSLs: productivity and performance. The first, productivity, encompasses measures of how a language may improve a users ability to write effective programs in their domain. For Materia we first examined how closely a test definition in a language conforms to a natural language description of a test found in earth science handbooks and literature. Second, we compared specific implementations of QC tests in Pandas against functionally equivalent definitions in Materia. This enabled us to compare ease of mapping the domain problem space to the language of implementation. For performance, we compared the runtimes of test definitions in Materia against implementations in Pandas. For these tests we used 6 commonly found quality control tests as our benchmarks

4.1 Benchmark Tests

The six tests used as our benchmarks and foundations of syntactic comparison come from Campbell et al. missing-measurements, range, persistence, spatial inconsistency, internal inconsistency and change in slope [2]. Missing-measurements checks the difference between two date-times in chronological order, if they exceed a specified time interval that means a measurement is missing and a row must be inserted. Range tests seek to identify if a value exceeds some normal range in values. Persistence checks if a given value in a dataset is a repeat of one or more data points preceding it. With spatial inconsistency, two or more time series measuring the same data type in close proximity to one-another are compared. If one diverges it usually indicates a logging error. Internal consistency evaluates a break in a logical condition between two time series. For example: a minimum variable measurement at a certain time index cannot exceed a maximum reading from the same sensor for the same variable at the same time index. Finally, a change in slope describes a dramatic upward or downward change in our time series' slope over a short time period.

4.2 Productivity

For the first metric of productivity, natural-language similarity, we selected the following five descriptions of individual quality control tests:

- "No values less than a minimum value or greater than the maximum value the sensor can output are acceptable" – Range Test [6]
- "This test compares the present observation n to a number . . of previous observations." – Persistence [6]
- "A sharp increase or decrease [in slope] over a very short time interval (i.e., a spike or step function)" – Change in slope [2]
- "ensuring that the minimum air temperature is less than maximum air temperature " – Internal Inconsistency [2]
- ". . . data from one location are compared with data from nearby identical sensors" – Spatial Inconsistency [2]

These particular tests were chosen because they reflect the six cardinal tests enumerated by Cambell et al [2], with the exception of "missing measurements." "Missing measurements" was omitted from this examination because it was implemented as static method in the TimeSeries class. These descriptions were compared by the developers of Materia, through and informal side-by-side comparison. (These implementations are not included here for space reasons however they can be found at [10].)

From this examination, no clear consensus was found. Quantifying similarity on a four step range, from "very similar" to "similiar" to "dissimilar" to "very dissimilar" we determined that a Materia-based implementation of a persistence test, visible in Listing 1.3, was "very similar." We considered the very complex "Change in slope" test to be, by contrast, "very dissimilar". We found internal inconsistency and range tests "similar" and spatial inconsistency to be "dissimilar."

Overall, the results of similarity or dissimilarity seem to be most significantly related to how simple the mathematical or boolean operation used in a test may be. For the persistence test, syntactical similarity is bolstered by the fact that an natural language description explicitly mentions a comparison and and that Materia provides a "prior" function which was designed to mimic natural language comparisons like this. On the other end of the spectrum, a Materia-based implementation of a slope test is over 30 lines of code and reflects the multiple calculations required to compare two slopes. The natural language description of this does not capture the complexity of these operations at all.

For the second metric gauging productivity, we compared Materia-based implementations of code against Pandas-based implementations. For Spatial Inconsistency, Spike and Range Tests, the details of implementation were approximately the same; although Pandas based implementations occasionally required the use of helper functions like np.logical_and to support vector-wise Boolean operations. With a logical inconsistency test we see more deviation with Materia's more simple definition where a user can simply declare "**return max_value < min_value**". In Pandas we require the use of a helper function, "np.where" to return an array of boolean values. Although not a significant addition, this does introduce some visual noise which can be hard to parse. Finally, we see significant divergence between implementations of a persistence test in Pandas and Materia. As can be seen in Figure 1.1, Pandas requires several different statements to identify and filter a time series down to identified repeat values. It requires further statements to express the result as a Boolean array. Without knowing what the code does, it's hard to understand even with a few minutes of exposure. By comparison, it takes mere seconds to understand what the persistence test in Materia (Listing 1.3) is doing.

4.3 Performance

Due to space limitations we cannot include a comprehensive breakdown of our performance measurements. It should be noted however, that across all tests, Materia did perform more poorly than the highly optimized Pandas. In general, Pandas, ran it's operations faster by an order of magnitude compared to Materia. We argue that this is not a fatal mark against this EDSL as the absolute runtimes of Materia never exceeded 2.5 seconds for more than 200,000 data points. When this represents 6 months of data and Quality Control is often done on a month by month basis or as part of prepossessing steps we deemed this an acceptably low runtime for practical use.

4.4 Additional Considerations

In addition to the above evaluation metrics, it should also be noted that Materia provides several features which support productivity but were not formally evaluated and are not found in more general purpose languages like Pandas. Specifically, Materia provides to users the automatic binding of flags to data, the automatic detection of column datatypes, alignment with a time series index, the graceful management of multivariate data stored in a tabular format, and the ability to handle nonstandard, highly variable header metadata.

In addition to these features, Materia also uniquely supports tracking Quality Control methods by storing the specific functions which were used to generate flags. This level of provenance is extremely important to creating comprehensive data products. With this feature, data producers are able to not only express that a datapoint may be suspect or bad but also *under what conditions* it was found to be suspect or bad.

5 Conclusions and Future Work

Over the course of this paper we introduced a topology of modern data quality control, existing solutions and the limitations of those solutions which motivate this work. We further introduced our embedded domain specific language: Materia, and detailed the design and implementation of it. This language was not shown to be superior in terms of performance compared to its most similar counterpart on the Python platform: Pandas. However, it was argued that it should be sufficiently performant for most typical quality control use-cases. In terms of productivity, Materia was developed to be more usable than Pandas for defining quality control specific tests and functions. Additionally, it was shown that Materia aggregates many features into one language which can positively impact the quality of life for developers building quality control applications and scripts.

There are many opportunities for future development of Materia. First and foremost, with the functional structure that was implemented for this prototype, its evident that many quality control tests are embarrassingly parallel. Accordingly, this language would significantly benefit from a deep embedding which could be exported to a GPU computing language. Many of these functions map 1 to 1 with a kernel that could be deployed to individual threads on a GPU architecture. In addition to this, with a deep embedding, Materia could support a deeper and richer syntax than is currently provided in this implementation.

Acknowledgements

We thank Michelle Strout for her invaluable input throughout the development of this project and editing provided for this paper. We would also like to thank Kate Isaacs for her input editing this paper and Chase Carthen for his input on the design of this language.

References

[1] Lars Brabyn et al. "Accuracy assessment of land surface temperature retrievals from Landsat 7 ETM + in the Dry Valleys of Antarctica using iButton temperature loggers and weather station data". In: *Environmental Monitoring and Assessment* 186.4 (Apr. 2014), pp. 2619–2628. ISSN: 1573-2959. DOI: `10.1007/s10661-013-3565-9`. URL: `https://doi.org/10.1007/s10661-013-3565-9`.

[2] John L. Campbell et al. "Quantity is Nothing without Quality: Automated QA/QC for Streaming Environmental Sensor Data". In: *BioScience* 63.7 (July 2013), pp. 574–585. ISSN: 1525-3244. DOI: `10.1525/bio.2013.63.7.10`. URL: `https://academic.oup.com/bioscience/article-lookup/doi/10.1525/bio.2013.63.7.10`.

[3] ESIP Envirosensing Cluster. *Sensor Data Quality*. Last Accessed on 05/28/20. 2019. URL: `http://wiki.esipfed.org/index.php/Sensor_Data_Quality`.

[4] ESRI. *ArcGIS*. `http://resources.arcgis.com/en/communities/data-reviewer/`. Oct. 2017.

[5] Andy Gill. "Domain-specific languages and code synthesis using Haskell". In: *Communications of the ACM* 57.6 (2014), pp. 42–49.

[6] Carl C. Gouldman, Kathleen Bailey, and Julianna O. Thomas. "Manual for Real-Time Oceanographic Data Quality Control Flags". In: *IOOS* (2017).

[7] Georgia Coastal Ecosystems LTER. *GCE Data Toolbox for MATLAB*. `http://gce-lter.marsci.uga.edu/public/im/tools/data_toolbox.htm`. 2017.

[8] Wes Mckinney. "pandas: a Foundational Python Library for Data Analysis and Statistics". In: *Python High Performance Science Computer* (Jan. 2011).

[9] Campbell Scientific. *LoggerNET*. `https://www.campbellsci.com/loggernet`. Dec. 2017.

[10] Connor Scully-Allison. *Materia*. Version 0.11. May 2020. DOI: `10.5281/zenodo.3870396`. URL: `https://github.com/cscully-allison/Materia`.

[11] Connor Francis Scully-Allison. "Keystone: A Streaming Data Management Model for the Environmental Sciences". PhD thesis. 2019.

[12] Connor Scully-Allison et al. "Near real-time autonomous quality control for streaming environmental sensor data". In: *Procedia Computer Science* 126 (2018), pp. 1656–1665.

[13] Wade M Sheldon. "Dynamic, Rule-based Quality Control Framework for Real-time Sensor Data". In: *Proceedings of the Environmental Information Management Conference* (2008), pp. 145–150. URL: `https://lternet.edu/wp-content/uploads/2010/12/eim-2008-proceedingssmall.pdf`.

[14] Mark D. Wilkinson et al. "The FAIR Guiding Principles for scientific data management and stewardship". In: *Scientific Data* (2016). DOI: `10.1038/sdata.2016.18`.