The MOSAIC Data-Set Explorer (MdsX): Initial Developer's Overview

Recent years have seen an increasing emphasis, in the academic and scientific worlds, on *data publishing* — sharing research data and experimental results/protocols via web portals complementing those that host scientific papers. Published data sets now take a position alongside books and articles as primary publicly-accessible outputs of scientific projects. Coinciding with this increased volume of raw data, there has also emerged an ecosystem of tools allowing researchers to find, view, explore, and reuse data sets. These tools enhance the value of published data, because they decrease the amount of effort which scientists need to make use of data sets in productive ways.

Unfortunately, however, this ecosystem of tools does not include extensive work on software *applications* for accessing and using published data sets. Prominent publishes (Elsevier, Springer, Wiley, de Gruyter, etc.) have all developed suites of components for manipulating data sets and data/code repositories, including APIs, search portals, Semantic Web ontologies and other forms of Controlled Vocabularies, and cloud-based computing or visualization engines, founded on technologies such as Jupyter, Docker, and WebGL. However, none of these publishers actually provide *applications* for accessing data sets outside of the online resources where data sets are indexed. While these online portals can provide a basic overview of the data sets, publishers do not provide tools to help researchers rigorously use any data sets once they are downloaded. Moreover, the ecosystem for manipulating published research is largely disconnected from the software applications which scientists actually use to do research. The ability to work with data-publishing tools has not been implemented within most scientific-computing environments.

These lacunae may be explained in part by publishers' and scientists' hopes of creating cloud-hosted environments that can themselves serve as fully featured scientific-computing frameworks, with the ability to run code, evaluate queries, interactively display 2D and 3D graphics, and maintain user and session state so that researchers can suspend and resume their work at different times. In these cloud environments, users can run computations and generate complex graphics on remote processing units, with relatively little data or code-execution stored or performed on their own computers. Such employment of remote, virtual programming environments is sometimes necessary when interacting with extremely large data repositories; and can be a convenient way to explore data sets in general, especially if a user is unsure whether or not a given data set is in fact germane to their research. Investigating data via cloud services spares the researcher from having to download the data set directly (along with the additional software and requirements which are often needed to make downloaded data functionally accessible). However, cloud-based data access is limited in

important ways, which makes relying solely on cloud services to provide the filaments of a research-data ecosystm a very bad idea. The first problem is that cloud services are, despite their technical features, essentially just web applications under the hood; as such, they are susceptible to the same User Expericne degradation as any other web service — subpar performance due to network latency, poor connectivity, and the simple fact that web-based graphics can never be as responsive or as compelling as desktop software, which can interact directly with the local operating system and react instantaneously to user actions. The second, more serious problem is that could-computing environments are computationally and architecturally different than the native-application contexts where scientific software usually operates. Insofar as researchers develop new analytic techniques, implement new algorithms, or write custom code to process the data generated by a new experiment, these computational resources are usually formulated in a local-processing environment that cannot be translated, without extra effort, to the cloud.

To be sure, scientists can sometimes "package" their experimental and analytic methods into a coherent framework, such as a Jupyter notebook, which serves as both a demonstration and a precis of their research work. Indeed, tools such as Jupyter (which packages code, data, and graphics into a self-contained Python-based programming environment) are useful in part because the content shared via these systems (e.g. Jupyter "notebooks") needs to be deliberately curated; building a notebook is a kind of summarial follow-up to actual research work. The intellectual discipline involved in packaging up one's research via such tools may be a valuable stage in the scientific process, but even then the programming environment where research code and data is publicly shared is fundamentally different than the environment where the research is actually carried out. As a consequence, sharing research indirectly via cloud services and or "notebook"-oriented frameworks like Jupyter is not really conducive to either reuse or replication. To actually replicate a course of investigation, it is more thorough to employ the same (or at least functionally equivalent) software for data acquisition, analysis, and validation as the original software; and to incorporate published data in new projects, the data should be shared in such a way that the original research data, code, and protocols can be absorbed into a new research context, including the software used by the research team. Cloud-based services, which provide only an overview of research data, with limited analytic and imaging/visualization functionality compared to actual scientific software, do not substantially promote data replication and reuse insofar as these cloud services are functionally disconnected from scientific applications themselves.

This is the motivation behind x, a *native*, *desktop-style* application for accessing research data of different kinds — within the overall space of published data sets we can find specific variations, such as *data repositories* comprising multiple data sets; *image corpora* designed as test beds for Machine Vision and diagnostic-imaging methods; *simulations* which involve not only raw data but digital experiments that can be re-run as a way to access the data; and so forth. Each of these various kinds of data sets present different sorts of interactive specifications which must be implemented by the data-set explorer software. While executed as a native application — not a cloud service —

MdsX nevertheless incoporates the important ideas from contemporary data publishing (including ideas originating in the cloud context): workflow models, notebooks, access to publishers' APIs, etc. Another feature of MdsX is that it can be run as a standalone application *or* embedded in other applications — such as the software which researchers are already using. In short, MdsX can be seen as akin to a cloud-based data-publishing platform where the "cloud" is replaced by a scientific-computing application. Instead of being hosted remotely ("on the cloud"), MdsX is hosted within a local desktop application. This host application may be pre-existing program, or a custom host implemented to allow MdsX data-sets to be explored in standalone fashion (with a default implementation that can, as desired, be modified for individual data sets/repositories).

The Structure of MdsX Notebooks

A common feature of software through which users study and reuse research data sets is some form of "interactive notebooks," or digital resources combining data, code, and graphics/visuals. The main feature of notebook-oriented design is the idea of interactive code editing, where changes in the code directly leads to changes in a visual display (such as a plot or diagram) which is viewed alongside the code. This setup allows developers to present or demonstrate data sets, and associated code, in an exploratory and interactive manner.

The exact details of how "notebooks" are designed and implemented varies between different technologies, although the concept is most clearly associated with JUPYTER, which is a coding and presentation environment based on PYTHON. Whatever the underlying programming environment, notebooks — or as MdsX uses the term, "interactive/digital notebooks" (IDNs) — have several software-engineering requirements, including a scripting environment and a data-visualization layer, wherein data sets or numeric models are transformed into 2D or 3D graphics (charts, diagrams, etc.). Moreover, the scripting layer needs to be connected to the data-visualization layer so that scripts can modify the data-to-graphics transformations. A further requirement is functionality to load pre-existing data sets from saved files or from a web resource.

Beyond these general features, IDN programming can take different forms and prioritize different styles of user interaction. The MdsX approach recognizes that it is often more convenient to interact with applications through GUI actions — buttons, tabs, context menus, and so forth — than by typing in commands (whether or not these are executed immediately in REPL, or "read-eval-print-loop", fashion, or are stored in scripts). As such, IDNs should not differ in design too noticeably from conventional GUI windows or dialog boxes — they should not be little more than "REPLs with plots." On the other hand, rigorous GUI programming calls for a carefully organized set of mappings from potential user actions to application responses. Whether on the scripting level or the GUI coding, in short, implementations need a level of abstraction more general than the underlying event-handling and procedure-calling logic which forms the application's concrete operational

behavior. This semi-abstracted layer can be described in terms of "meta-classes," "meta-objects," "tools," "transitions," "services," and so forth: the common denominator in different contexts is some notion of a structure which can be called a "meta-procedure," similar to an ordinary computational procedure in having inputs and outputs, but embodying a level of abstraction somewhat removed from concrete procedures. In particular, meta-procedures are not directly implemented; instead, some algorithm is necessary to determine, given a description of a meta-procedure with its outputs and context, what concrete procedure (or set of procedures) should actually run. Moreover, meta-procedures need some notion of delayed execution: there is a logical gap between "marking" (using the language of petri-net theory), i.e., fully specifying the input parameters consumed by a meta-procedure, and a meta-procedure's actual execution. As such, meta-procedural markings can be built up in stages, with input data coming from multiple sources (including scripts and GUI elements). For a concrete example, consider the process of filling out a web form, wherein entries typed in to the form fields are validated, one at a time, before the form can be submitted. In these cases, the step-by-step process of entering and validating individual fields corresponds to incremental marking, and "hitting the submit button" corresponds to meta-procedure execution.

In short — although different systems use different terminology — any IDN programming environment needs a mechanism to incrementally define and execute meta-procedure calls. The implementational foundations of that mechanism (hypergraphs, workflow engines, state monads, etc.) depend on the underlying programming environment. The MdsX approach borrows ideas primarily from HyperGraphDB and SeCo, which is a notebook-programming environment based on HyperGraphDB. As in SeCo, units of marking and execution are called *cells*. The main difference between MdsX and SeCo (apart from C++ instead of being the underlying programming language) is that MdsX cells are not intended, in the general case, to be typed in by programmers directly. Instead, MdsX cells are normally constructed behind the scenes, on the basis of GUI component state, user actions, or scripting input. However, once constructed, they can be manipulated like SeCo cells, both in terms of functionality and in terms of rationale: they can be used as a log of user actions, for undo/redo, for defining workflows, for generating scripts, and so on. In particular, the mappings from GUI actions to application handlers can be defined (and extended) by annotating the relevant GUI elements with meta-procedure cells. This also allows data sets to be annotated with micro-citations (which are discussed below).

As a C++ environment, MdsX uses an embedded "virtual machine" to interpret meta-procedure cells; application-level event handlers are not automatically exposed to a scripting interface as they would be in a or PYTHON environment. However, MdsX also supports scripting via a choice of languages, similar to SeCo. The primary scripting language used with MdsX is AngelScript, although other C/C++ based languages (Embeddable Common Lisp, , etc.) can work as well. To support various scripting languages, modules loaded into MdsX need to provide a meta-procedural interface declaration, and the desired scripting language also needs a bridge to work with these declarations (which is generally usable across all datasets and modules). Such a bridge will be

provided by default for AngelScript and (Embeddable Common Lisp), and similar tools could be implemented for other languages.

The typical **MdsX** notebook combines, at a minimum, some graphical element — such as an image to be analyzed and/or a plot/diagram to be populated with data — along with a user-interface "panel" for interacting with the graphics, and the overall application. This panel partially takes the place of a script-composition or **REPL** frame, although such a frame is implicitly present, normally behind the scenes (users can view it if desired). Notebooks can then load data files, and representations of the loaded data (e.g., text serializations) may then also become part of the notebook content, able to be visualized in their own frame. Notebooks in general then can have four varieties of frames (graphics views, interaction panels, data panels, and meta-procedure logs) although not every available frame may be explicitly constructed and/or visible at a given point in the user's session. There may also be multiple instances of graphics frames. In any case, the layout and state of these various frames — what frames are visible, and their current content — define notebook *state* which can be saved, restored, and shared. Loading a data set into a **MdsX** notebook therefore involves loading a particular initial state, defined as part of the data set, arranged in part to serve as a useful starting-point for users to explore and visualize the relevant data. Each of these kinds of frames corresponds to a particular aspect of software implementations, requiring its own strategies and paradigms. The following sections will review these various programming concerns one at a time.

Image Analysis and Data Visualization

The central graphical element of an **MdsX** notebook is either a **2D** or **3D** image loaded from an image file (in formats such as **PNG**, **JPG**, **DICOM**, **TIFF**, etc.), or else a **2D** or **3D** plot, chart, or diagram. The functionality of the notebook will therefore differ depending on whether the central graphics is an image loaded from a file (called an "image-based" notebook), or a data visualization constructed from a data set or some mathematical formulae (called a "diagram-based notebook"). A third option is an "object-based" notebook, whose central viewport contains a structured display of information, mostly in textual form (for instance, a table or tree view), but this section will focus on graphics-oriented notebooks.

Diagram-based **MdsX** notebooks can be implemented with different diagram-plotting engines; the default implementations support **Qt** Charts (a built-in **Qt** module) as well as the qtcustomplot and JKQCustomPlotter libaries. In short, diagram-based notebooks need to implement subclasses of **MdsX** frames for a navigation panel, graphics view, meta-procedure view, and data view, as well as a "**MdsX** diagram" subclass occupying the diagram view. In this case, the primary responsibility of the meta-procedural layer is to interface with functionality provided by the diagram/plotting engine. Since most coding details are derived from these engine's object models and classes, they lie mostly

outside the scope of this paper.

Image-based notebooks, on the other hand, need to integrate several different areas of functionality. As such, setting up the image view is only one step in constructing such a notebook; additional programming is needed to support image annotation, analysis, and feature extraction. In order to integrate these different layers of functionality, **MdsX** provides a "Data Structure Protocol for Image-Analysis Networking" (**D-SPIN**) which defines communication rules between image-related software subsystems in Object-Oriented terms. **D-SPIN** objects are comprised of four more specific objects or layers, describing different aspects of the shared image and how it should be processed. These four layers are defined as follows:

Metadata Layer This object presents metadata describing the image format and acquisition. If the surrounding **D-SPIN** object represents an image series (rather than a single image), the metadata object should also declare the size of the collection and how individual images should be referenced. The metadata should include a file path or resource identifier asserting where the image can be acquired from (which in the case of a series can be a zipped folder or a list of resource paths). More specific metadata depends on the image or images' graphical format; to properly load images in most formats (such as **PNG**, , **TIFF**, and **DICOM**) applications need to specify detail such as dimensions, resolution, and color depth. Of course, some of this information is stored internally within the image file (depending on its format), but certain formats require some metadata to be shared along with the image itself (moreover, it is often convenient to have basic information available without needing to extract it from binary image data). The details on which form of metadata are appropriate for which image format can be determined based on image-viewing code libraries, such as **libpng**, **libtiff**, or **DICOM** clients. If both end-points of a **D-SPIN** communication have the same libraries installed, the sending application will have a clear idea of how much supplemental data is needed over and above what will be read from image files directly. If there are uncertainties in library alignment between the two end-points, the sending application should consider serializing a more detailed summary of the image providing any information that would ordinarily be read from the image file.

Annotation Layer Almost all image analysis — whether done by humans or by softare — results in either some form of statistical representation of an image's properties, or a complex of data which presents information about (and may visually overlay) the image, particularly in the form of annotations. Image annotations are arrows, line segments, or **2D** closed shapes that call attention to some point or region inside the image, usually with some additional label or commentary. The basis of each annotation is therefore some zero-dimensional or two-dimensional region (or a set of zero-dimensional control points; or, occasionally, a one-dimensonsal line or curve), so annotations require a mechanism for designating regions. The same issues apply to asserting feature-vectors with respect to an image region rather than the image as a whole; accordingly, both annotations and feature vectors can be seen as equivalent varieties of constructions which

isolate and then define data structures on zero-, one-, and/or two-dimensional subimages (feature vectors on the entire image can accordingly be treated as a special case).

Contextual Layer Contextual information associated with an image can include metadata or supplemental details that are not directly relevant to the image, but convey facts about how the image connects to a broader context where it was obtained, and for what purpose. An example of contextual data would be the part of DICOM headers that include patient or clinical information, rather than metadata about image format or dimensions.

Procedural Layer

In MdsX, D-SPIN objects are associated with image-based notebooks in that the notebook components (the navigation panel and graphics, data, and meta-procedural controllers) jointly refer to a common D-SPIN object for all image-related data. Image-analysis routines conducted within the notebook then may yield additional data structures bundled into the overarching D-SPIN object. This overarching object can then be exported or saved, alongside (and as an extension of) notebook state.

Analytic operations available through an image-based notebook may be provided by the notebook itself, or by a host application where MdsX is embedded. In the latter case, the notebook needs to construct the proper calls to the host application, using the meta-procedural controller as a bridge to ambient capabilities. For instance, if an MdsX notebook is developed as a plugin to CaPTk, the notebook would interface with the host CaPTk application via the formats and programming constructs which CaPTk recognizes (specifically, the Common Workflow Language and the Qt signal/slot mechanism). This specific scenario — embedding MdsX in CaPTk — is employed as a demonstration and case-study for embedding notebooks in host application in general. The CaPTk workflow protocol also forms a basis for the meta-procedural view and controllers, discussed next.

MdsX Meta-Procedure Controllers

The meta-procedural layer of an MdsX notebook is responsible for handling events generated by a corresponding navigational panel, or at least those events which have a non-trivial impact on notebook/session state and data. The visual representation of meta-procedural commands and history is provided by a meta-procedural "view," which is normally invisible, but notebooks may choose to allow users to "unhide" this view. The meta-procedural controller is responsible for generating the meta-procedural view (if applicable) and responding to user events within this view; it is also responsible for maintaining an inventory of objects summarizing available metaprocedures, GUI elements, and the mappings between them.

In general, the GUI elements in these meta-procedural mappings are referred to as "visual objects," and are represented in the meta-procedural controller context via application-unique identifiers (not raw pointers). Similarly, "meta-procedural objects" encapsulate information about meta-procedures themselves.