The dataChest For Dummies

Alexander Opremcak*

December 27, 2016

Contents

*electronic address: opremcak@wisc.edu

1 Introduction

The `dataChest` is a Python library written for the storage and retrieval data. Datasets are stored in the HDF5 file format so this library is just a glorified wrapper over the h5py library. The `dataChest` was written to enforce lab-wide consistency, but remains flexible enough to handle data of virtually any type or shape. In the next section, I describe all of the public methods this class has to offer. I will then formalize our definitions of 1D and 2D datasets in order to provide examples of how each of these types is encountered in a laboratory setting.

2 Overview of functionality

The `dataChest` has a small number of public methods which can be categorized as follows: i) Folder Creation and Navigation methods, ii) Dataset methods, and iii) Parameter methods. The Folder Creation and Navigation methods give users basic tools for navigating and creating directories. The Dataset methods allow users to create new datasets, open existing datasets, and add/fetch data. Parameter methods allow users to attach additional information, 'parameters', to a dataset.

2.1 Setting up environment variables

Before we begin with any examples of how to use this library, we must set up two environment variables: i) `DATA_CHEST_ROOT` and ii) `PYTHONPATH`. The `DATA_CHEST_ROOT` provides the `dataChest` with the starting location (root directory) in which all folders and datasets will be contained. Make sure that this location exists before you attempt to run the Python code examples. The `PYTHONPATH` enviroment variable points to the location of the `dataChest` folder which contains all code relevant to this library. By creating this environment variable, you will be able to import `dataChest` from any location on your computer.

2.1.1 For Mac

If you are working on a mac, open up terminal and type:

```
cd $HOME
ls -a
```

This will list all of the contents within your home directory, including hidden files. You should see a file called `.bash_profile` (or `.profile` for older macs). Now type

```
nano .bash_profile
```

which will open a text editor for modifying this file. Once opened enter the following two lines:

```
DATA_CHEST_ROOT=$HOME/Desktop/Data
export DATA_CHEST_ROOT
PYTHONPATH=${PYTHONPATH}:$HOME/Desktop/Repositories/servers/dataChest
export PYTHONPATH
```

After you save these changes, run

```
source .bash_profile
```

to make your changes take effect.

2.1.2 For Windows

If you are working with windows, you will need to create the environment variables in a slightly different manner. Here are the steps:

1. Open the control panel and type "Environment Variable" into the search menu.
2. Click on the "Edit the system environment variables" link.
3. In the System Properties menu under the advanced tab click on the "Environment Variables" button.
4. Under the "User Variables" portion of the menu, click the new button to create "New..." environment variable or "Edit..." if it already exists.
5. Let `DATA_CHEST_ROOT` = C:/Data (or wherever you want data to be stored on your computer, in any case **note the direction of the slash** which is contrary to the usual windows backslash).
6. Let `PYTHONPATH` = C:\RepositoryLocationPath\servers\dataChest (if the variable already exists, append to it in a semicolon delimited fashion).

2.2 Creating a `dataChest` object

Now that we are done setting up environment variables, let's cover one last preliminary step: instantiation. First open up terminal, then type `python` and create a `dataChest` object as follows:

```
from dataChest import *
d = dataChest("DataChestTutorial") # defines the root directory
```

The folder `"DataChestTutorial"` will be created if it does not exist.

Hereafter I will assume the `dataChest` library has been imported, the object `d` above exists, and that each snippet of code has been executed in the order that it was presented.

2.3 Folder creation and navigation methods

2.3.1 `dataChest.mkdir`

`dataChest.mkdir(directoryToMake)`

- Description: Creates a directory within the current working directory.
- Parameters:
  - `directoryToMake` : `str`
    The name of the directory you wish to create.
- Returns:
  - `None`

`dataChest.mkdir` does not automatically change the working directory when a new directory is created, nor would it in Linux/Unix. If the directory already exists or the directory name is invalid, an `OSError` is raised. Note that directory names are case-insensitive on MAC and Windows. Here is an example of how it works:

```
d.mkdir("DummyDirectory")
```

2.3.2 `dataChest.ls`

`dataChest.ls()`

- Description: Returns a list composed of two lists. The first element being a list of all files in the current working directory. The second being a list of all folders in the current working directory
- Parameters:
  - Accepts no input parameters.
- Returns:
  - `[filesList, foldersList]` : `list` composed of two `lists`

Here is how its used:

```
contents = d.ls()
files = contents[0] # list of all files names
folders = contents[1] # list of all folders names

print "DummyDirectory" in folders # ==> True
print "NonexistentDirectory" in folders # ==> False

print "contents=", contents # ==> [[],["DummyDirectory"]]
print "files=", files # ==> [] no files created thusfar
print "folders=", folders # ==> ["DummyDirectory"]
```

2.3.3 `dataChest.pwd`

`dataChest.pwd()`

- Description: Returns a `str` path of the current working directory.
- Parameters:
  - Accepts no input parameters.
- Returns:
  - `currentWorkingDirectory` : `str`
    The path of the current working directory.

This method will be used in the example that follows.

2.3.4 `dataChest.cd`

`dataChest.cd(relativePath)`

- Description: Makes changes relative to the current working directory.
- Parameters:
  - `relativePath`: `str` or `list`
- Returns:
  - `None`

Try running this:

```
print "working directory =", d.pwd()

d.cd("DummyDirectory") # str style input
print "working directory =", d.pwd()

d.mkdir("DummySubDirectory")
d.cd("..") # bumps up one
print "working directory =", d.pwd()

d.cd(["DummyDirectory", "DummySubDirectory"]) # list style input
print "working directory =", d.pwd()

d.cd("") # moves home
print "working directory =", d.pwd() # root
```

Note that the root directory is `DATA_CHEST_ROOT` + "/" + "DataChestTutorial", not simply the `DATA_CHEST_ROOT` environment variable. This was done to avoid cluttering the root directory in which all of our datasets are stored.

2.4 Dataset methods

2.4.1 dataChest.createDataset

```
dataChest.createDataset(datasetName, indepVarsList,
                        depVarsList, dateStamp=None)
```

- Description: Creates an hdf5 file in location `d.pwd()`.
- Parameters:
  - `datasetName`: `str`
  - `indepVarsList`: `list`
    Each list element is a `tuple` with entries
    `(varNameStr, varShapeList, dtypeStr, varUnitsStr)`
  - `depVarsList`: `list`
    List elements are the same as for `indepVarsList`.
  - `dateStamp`: `str`
    An optional argument that allows you to submit your own datestamp. Should be used for backfilling purposes only.
- Returns:
  - `None`

Here is a quick example of how it is used:

```
d.createDataset("timeSeries",
                [("time", [1], "float64", "s")],
                [("voltage", [1], "float64", "V")]
                )
```

Let us briefly discuss the arguments. The first argument is the `datasetName`. The second argument is the `indepVarsList` which describes all pertinent independent variable information to the dataChest. It contains a single `tuple`, meaning there is only one independent variable. Let us pick apart the entries of this `tuple`. The first element is the independent variable's name. The second element is a list describing the variable's shape. We shall assume that a variable never changes its shape. The third element, the `dtypeStr`, is a `str` describing the variable's data type. Again we will assume that a variable never changes its data type. For a list of acceptable data types, see Table 1. The last element is a `str` describing the variable's units. The same holds for the `depVarsList`.

While scalar data is very common, it pays to allow for other shapes of data to be stored. Shapes are just lists which specify the shape of some multi-dimensional array. For example 3x3 matrix would have shape `[3,3]` whereas a 1000 point waveform (1D array) has shape `[1000]`.

Table 1: Data Types

Data Types	Description#1
"bool_"	Boolean
"int8"	8-bit Signed Integer (-2^7 to 2^7-1)
"int16"	16-bit Signed Integer (-2^{15} to $2^{15}-1$)
"int32"	32-bit Signed Integer (-2^{31} to $2^{31}-1$)
"int64"	64-bit Signed Integer (-2^{63} to $2^{63}-1$)
"uint8"	8-bit Unsigned integer (0 to 2^8-1)
"uint16"	16-bit Unsigned integer (0 to $2^{16}-1$)
"uint32"	32-bit Unsigned integer (0 to $2^{32}-1$)
"uint64"	64-bit Unsigned integer (0 to $2^{64}-1$)
"int_"	Default integer type (normally either "int64" or "int32")
"float16"	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
"float32"	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
"float64"	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
"float_"	Shorthand for "float64"
"complex64"	Complex number, represented by two 32-bit floats
"complex128"	Complex number, represented by two 64-bit floats
"complex_"	Shorthand for "complex128"
"utc_datetime"	UTC Datetime Float
"string"	String

The data type "utc_datetime" shown in Table 1 deserves some special attention. UTC, which stands for Coordinated Universal Time, is the primary time standard by which the world regulates clocks and time. Here is how you generate the current "utc_datetime":

```
from dateStamp import * # helper class for dataChest
dStamp = dateStamp() # instantiation
utcFloat = dStamp.utcNowFloat()
print "utcFloat=", utcFloat # ==> current UTC time as a float
```

Note this has microsecond timing resolution.

2.4.2 dataChest.getDatasetName

`dataChest.getDatasetName()`

- Description: Returns the name of the current dataset.
- Parameters:
  - Accepts no input parameters.
- Returns:
  - `currentDatasetName : str`

Here is an example of how to use it:

```
datasetName = d.getDatasetName()
print "datasetName=", datasetName #dateStamp+"_"+"timeSeries.hdf5"
```

The dateStamp plus the name provided to `dataChest.createDataset` make up the actual name of the dataset on disk. The dateStamp makes the filename unique while preserving the chronological order in which the dataset was created.

2.4.3 `dataChest.getVariables`

`dataChest.getVariables()`

- Description: Returns a list of lists containing the `indepVarsList` and `depVarsList` provided to `dataChest.createDataset`.
- Parameters:
  - Accepts no input parameters.
- Returns:
  - `[indepVarsList, depVarsList]` : `list`

Try the following:

```
varsList = d.getVariables()
indepVarsList = varsList[0]
depVarsList = varsList[1]

print indepVarsList # list of independent vars
# ==>  [('time', [1], 'float64', 's')]
print depVarsList # list of dependent vars
# ==>  [('voltage', [1], 'float64', 'V')]
```

2.4.4 `dataChest.addData`

`dataChest.addData(data)`

- Description: Adds data to a newly[1] created dataset.
- Parameters:
  - data: `list`
    A list whose entries are composed of individual 'rows' of data.
- Returns:
  - `None`

[1] How to add data to an existing dataset will be discussed with `dataChest.openDataset`.

Run the following:

```
d.createDataset("someDataset",
                [("ind1", [1], "float64", "s")],
                [("dep1", [1], "complex128", "V")]
                )
d.addData([[1.0, 2.0+1j*2.0]]) #add 1 row
d.addData([[2.0, 4.0+1j*4.0],[3.0, 6.0+1j*6.0]]) #add 2 rows
```

It is worth noting that each dataChest object is associated with at most one dataset at any given time. If you wanted to write to multiple datasets at once, each dataset will require its own dataChest object. Here is an example of how to do this:

```
d1 = dataChest("ParallelWrites")
d2 = dataChest("ParallelWrites")

d1.createDataset("Dataset1",
                [("LoggerTime", [1], "utc_datetime", "s")],
                [("ColdStageTemp", [1], "float64", "K")]
                )

d2.createDataset("Dataset2",
                [("time", [1], "float64", "s")],
                [("voltage", [1], "float64", "V")]
                )

d1.addData([[dStamp.utcNowFloat(), 21.2e-3]]) # write to one
d2.addData([[1.0, 2.0]]) # then to the other

d1.addData([[dStamp.utcNowFloat(), 19.4e-3]])  # repeat
d2.addData([[2.0, 4.0]])
```

2.4.5 `dataChest.getNumRows`

`dataChest.getNumRows()`

- Description: Returns the number of rows added to a dataset.
- Parameters:
  - Accepts no input parameters.
- Returns:
  - `numRows` : `int`

This will be used in the next example.

2.4.6 dataChest.getData

```
dataChest.getData(startIndex=np.nan, stopIndex=np.nan)
```

- Description: Returns data from the current dataset.
- Parameters:
  - `startIndex` : `int`, optional
  - `stopIndex` : `int`, optional
- Returns:
  - `numRows` : `int`

Let's run the following code:

```
numRows = d.getNumRows()
print numRows # ==> 3
data = d.getData() # fetch all data from "someDataset"
print data # all entries
```

There may be situations in which a user wishes to fetch a subset of the available data as opposed to the all or nothing style shown above. Here is how to do it:

```
d.getData(1) # second element
d.getData(-1) # last element
d.getData(-2) # second to last element
d.getData(None, 2) # equivalent to [:2]
d.getData(1, None) # equivalent to [1:]
```

`None` as the first (last) argument plays the role of blank followed (preceded) by a colon in the usual list slicing syntax for Python.

2.4.7 dataChest.openDataset

```
dataChest.openDataset(filename, modify=False)
```

- Description: Opens an existing dataset for the retrieval of data and parameters.
- Parameters:
  - `fileName` : `str`
  - `modify` : `bool`, optional
    An optional argument which can be used to modify an existing dataset's parameters and to add data.
- Returns:
  - `None`

Let's run the following:

```
currentDataset = d.getDatasetName() # get current dataset name
print currentDataset
d.createDataset("someName",  # this is now the current dataset
                [("yee", [1], "float64", "s")],
                [("haw", [1], "float64", "V")]
                ) # create new dataset
print d.getDatasetName() # name differs from currentDataset
d.openDataset(currentDataset) # reopens the previous dataset
print d.getDatasetName() # currentDataset
d.addData([[4.0, 8.0+1j*8.0]]) # raises Warning
print d.getData() # prints data from old dataset
```

`dataChest.openDataset` does not allow users to use add data by default. This was done so that datasets are not unintentionally modified. If a user would like to add data to an existing dataset, they must set the optional parameter `modify = True` as shown below[2]:

```
d.openDataset(currentDataset, modify = True) # write/read access
d.addData([[4.0, 8.0+1j*8.0]]) # add to an old dataset with modify = True
```

2.5 Parameter methods

2.5.1 `dataChest.addParameter`

```
dataChest.addParameter(paramName, paramValue, overwrite=False)
```

- Description: Adds a parameter to a dataset.
- Parameters:
  - `paramName` : `str`
    The parameter's name.
  - `paramValue` : `int`, `long`, `float`, `complex`, `bool`, `list`, `str`, `unicode` or any numpy type shown in Table 1.
  - `overwrite` : `bool`
    An optional argument that allows for parameters to be overwritten when set to `True`.
- Returns:
  - `None`

[2]Parameters obey the same rules.

Here is how it works:

```
d.addParameter("Who", "Mike Jones")
d.addParameter("Time", "Goon Time")
d.addParameter("Base Temp", -13.789) # et cetera
```

If a user attempts to add a parameter that already exists, this method will raise an exception. If a user would like to overwrite an existing parameter with a new value[3], then they must opt for overwrite privileges using the following syntax:

```
d.addParameter("DummyParam", 11)
d.addParameter("DummyParam", 14)  # raises an exception
d.addParameter("DummyParam", 13, overwrite = True)
d.addParameter("DummyParam", "blah", overwrite = True)
```

2.5.2 `dataChest.getParameter`

```
dataChest.getParameter(paramName, bypassIOError=False)
```

- Description: Opens an existing dataset for the retrieval of data and parameters.
- Parameters:
  - `paramName` : `str`
  - `bypassIOError` : `bool`, optional
    An optional argument which can be used to bypass an `IOError` in the event that a parameter is not found.
- Returns:
  - `paramValue` : with a data type corresponding to what was given to `dataChest.addParameter`.

Here is how it works:

```
print d.getParameter("Who") # ==> "Mike Jones"
print d.getParameter("Time") # ==> "Goon Time"
print d.getParameter("Base Temp") # ==> -13.789
print d.getParameter("DummyParam") # ==> "blah"
```

[3] A parameters data type can also be changed.

2.5.3 `dataChest.getParameterList`

`dataChest.getParameterList()`

- Description: Retrieves a list of all parameter names that are attached to the dataset.
- Parameters:
  - `fileName` : `str`
  - `modify` : `bool`, optional
    An optional argument which can be used to modify an existing dataset's parameters and to add data.
- Returns:
  - `None`

Here is how it works:

```
print d.getParameterList()
# ==> ['Who', 'Time', 'Base Temp', 'DummyParam']
```

3 Canonical types of datasets

While the overview of user-end functionality may have been helpful, it will prove useful to go over some examples of how to create and add to some of the most commonly encountered types of datasets. For each type of data, I will provide motivation for the definition and then an example of how to add it to the dataChest.

3.1 1D data

By 1D data, we mean a dataset with one independent variable and m dependent variables where $m \geq 1$. Loosely speaking, this captures what it means to be a 1D dataset entirely. In what follows, I will motivate the two most commonly encountered types of 1D data.

3.1.1 Arbitrary 1D data

The most general type of 1D data has an arbitrary spacing between consecutive data points along the independent variable axis, the x-axis. By fixing the independent variable at some value, call it v, and measuring all of the m dependent quantities $Q_1(v), \ldots, Q_m(v)$ of interest, we obtain a single row of data

$$\text{row} = \Big[v, Q_1(v), \ldots, Q_m(v)\Big] \tag{1}$$

Note the size brackets that we use here to define a row. For each value of v, we repeat this process and eventually our data looks like

$$\text{data} = \Big[\Big[v_0, Q_1(v_0), \ldots, Q_m(v_0)\Big], \Big[v_1, Q_1(v_1), \ldots, Q_m(v_1)\Big], \ldots \Big[v_N, Q_1(v_N), \ldots, Q_m(v_N)\Big]\Big] \tag{2}$$

where we have $N+1$ rows of data with $m+1$ columns of scalar[4] data in each row. Let us call data of this format **Arbitrary Type 1 Data**. Here is an example of how this data entered:

```
import numpy as np
d.createDataset("MyFavoriteTimeSeries",
                [("indepName1", [1], "float64", "s")],
                [("depName1", [1], "float64", "V")]
                )
d.addParameter("X Label", "Time")
d.addParameter("Y Label", "Digitizer Noise")
d.addParameter("Plot Title", "Random Number Generator")
net = []
for ii in range(0, 100):
    net.append([float(ii),np.random.rand()])
d.addData(net) #add 100 rows of data at once
d.getData() #single row
```

Note that the shape of each variable is `[1]` which is really the definition of **Arbitrary Type 1 Data** as far as the dataChest is concerned.

Alternatively we could group this data like so

$$\text{data} = \Big[\Big[[v_0, \ldots, v_N], [Q_1(v_0), \ldots, Q_1(v_N)], \ldots [Q_m(v_0), \ldots, Q_m(v_N)]\Big]\Big] \tag{3}$$

Note that each column ('variable') has the same length and the implicit functional dependence of the Q_i's as a function of index that we have assumed. Let us call data of this format **Arbitrary Type 2 Data**. Here is how it is entered:

```
res = 1e-4
timeAxis = np.arange(0.0, 1.0, res)
```

[4]String data is also allowed.

```
d.createDataset("DampedOscillations",
                [("time", [len(timeAxis)], "float64", "s")],
                [("Oscillation", [len(timeAxis)], "float64", "V")]
                )
d.addParameter("X Label", "Time")
d.addParameter("Y Label", "Voltage")
d.addParameter("Plot Title", "Damped Oscillations")
d.addData([ [timeAxis, np.sin(2 * np.pi * timeAxis)] ])
d.getData()
```

We use the term 'arbitrary' for these types of data because it is not necessary for the v_i's to be related by any closed form expression as of function of array index. So there you have it. All 1D data fall under this category. So were done ... not quite!

3.1.2 1D scans

Suppose we rip a time series from an ADC with some fixed sampling rate, say 1GS/s. Then our independent axis is specified uniquely by 3 numbers, namely the initial time t_0, the final time t_f, and the number of samples N. The time at the j^{th} point in our time series is given by

$$t[j] = t_0 + \left(\frac{t_f - t_0}{N-1}\right) \cdot j \tag{4}$$

Let us call this style of data a **Linear 1D Scan** along the x-axis. In light of this equation, let us store our data in slightly more efficient manner

$$\begin{aligned} \text{data} = \Big[\Big[& [t_0, t_f], [V_0(t_0), V_0(t_1), \ldots, V_0(t_f)], \\ & [V_1(t_0), V_1(t_1), \ldots, V_1(t_f)], \ldots \\ & [V_m(t_0), V_m(t_1), \ldots, V_m(t_f)]\Big]\Big] \end{aligned} \tag{5}$$

where $[t_0, t_f]$ is simply shorthand an N dimensional array whose values are determined by equation (4). Here is how its entered:

```
length =1e7
mu, sigma = 1, 0.1
gaussian = mu + sigma*np.random.randn(length)
t0 = 0.0
tf = 100.0
d.createDataset("LinearWaveform",
                [("indepName1", [2], "float64", "s")],
                [("depName1", [int(length)], "float64", "V"),
                 ("depName2", [int(length)], "float64", "V")])
```

```
shorthandTime = [t0, tf]
d.addParameter("Scan Type", "Lin")
d.addData([[shorthandTime,gaussian,gaussian]])
d.getData()
```

As another example, suppose we are doing reflection and transmission measurements with a 2-port VNA from 20 MHz to 20 GHz. A linear frequency sweep will place a very low point density in the "low" frequency region with a majority of the points falling in the GHz region of frequency space. One way to get around this is by linearizing your frequency data on a logarithimic scale, i.e. a log sweep. Again, the independent axis, frequency, is determined by 3 numbers, that is the starting frequency f_0, the stopping frequency f_f, and the number of points swept in frequency space N. Then the frequency at the m^{th} point is given by

$$\log(f[m]) = \log(f_0) + \left(\frac{\log(f_f) - \log(f_0)}{N-1}\right) \cdot m \tag{6}$$

Lets call this style of data a **Logarithmic 1D Scan** along the x-axis. Following the above paragraph our data will look like

$$\begin{aligned} \text{data} = \Bigg[\Bigg[& [f_0, f_f], [S_{11}(f_0), S_{11}(f_1), \ldots, S_{11}(f_f)], \\ & [S_{12}(f_0), S_{12}(f_1), \ldots, S_{12}(f_f)], \\ & [S_{21}(f_0), S_{21}(f_1), \ldots, S_{21}(f_f)], \\ & [S_{22}(f_0), S_{22}(f_1), \ldots, S_{22}(f_f)]\Bigg]\Bigg] \end{aligned} \tag{7}$$

where $[f_0, f_f]$ is simply shorthand an N dimensional array whose values are determined by equation (6).

Since 1D scans are really just specific cases of arbitrary 1D datasets, you may wonder why this discussion is justified. However if the distinction being made here seems artificial, consider the fact that data returned from most ADCs, VNAs, and Spectrum Analyzers contains only the y values, the x-axis is built in software and returned for your convenience.

3.2 2D data

By 2D dataset, we mean a dataset with 2 independent variables and m dependent variables. The example that jumps to mind is 2D bias sweep over some rectangular grid but there are many others.

3.2.1 Arbitrary 2D data

Following the section on Arbitrary 1D Data, the most general type of 2D data has an arbitrary spacing between consecutive points with respect to each of

the dependent variables. Meaning you could move along a spiral, raster over some arbitrary 2D shape, or sweep over a rectangle. Suppose we can vary two independent quantities, v and p. We are interested in studying how m dependent quantities, $Q_1(v,p),\ldots,Q_m(v,p)$, vary as a function of v and p. For each combination $(v,p) \in \{(v_0,p_0),(v_1,p_1),\ldots,(v_N,p_N)\}$, we measure each of the Q_i's. The data will look something like

$$\begin{aligned}\text{data} = \Bigg[\Big[&v_0, p_0, Q_1(v_0,p_0),\ldots,Q_m(v_0,p_0)\Big],\\ &\Big[v_1, p_1, Q_1(v_1,p_1),\ldots,Q_m(v_1,p_1)\Big],\ldots\\ &\Big[v_n, p_n, Q_1(v_n,p_n),\ldots,Q_m(v_n,p_n)\Big]\Bigg]\end{aligned} \tag{8}$$

Again lets call data of this format **Arbitrary Type 1 Data** where we now have two independent quantities, v and p, instead of one.

Alternatively we could format it as

$$\begin{aligned}\text{data} = \Bigg[\Big[&[v_0,v_1,\ldots,v_n],[p_0,p_1,\ldots,p_n],\\ &[Q_1(v_0,p_0),Q_1(v_1,p_1),\ldots,Q_1(v_n,p_n)],\\ &[Q_2(v_0,p_0),Q_2(v_1,p_1),\ldots,Q_2(v_n,p_n)],\ldots\\ &[Q_m(v_0,p_0),Q_m(v_1,p_1),\ldots,Q_m(v_n,p_n)]\Big]\Bigg]\end{aligned} \tag{9}$$

Again we call data of this format **Arbitrary Type 2 Data**.

3.2.2 2D scans

While sweeping around in 2D space in an arbitrary manner is fun, it sometimes makes sense to sweep in a more systematic way. Suppose we are interested in rastering over some rectangular grid, namely all pairs of points (v,p) such that $v \in \{v_0,v_1,\ldots,v_m\}$ and $p \in \{p_0,p_1,\ldots,p_n\}$ where each of these sets follows a formula similar to equation (4) or (6). By measuring each of the Q_i's for all pairs, we map out a 2D surface over some rectangular domain that we decide upon in software. Lets fix a point along the v-axis and perform a 1D Scan along the p-axis, then move to the next point along the v-axis and repeat, we obtain

data that looks like

$$\text{data} = \Bigg[\Big[v_0, [p_0, p_n], [Q_1(v_0, p_0), Q_1(v_0, p_1), \ldots, Q_1(v_0, p_n)]\Big] \\ \Big[v_1, [p_0, p_n], [Q_1(v_1, p_0), Q_1(v_1, p_1), \ldots, Q_1(v_1, p_n)]\Big], \ldots \\ \Big[v_m, [p_0, p_n], [Q_1(v_m, p_0), Q_1(v_m, p_1), \ldots, Q_1(v_m, p_n)]\Big]\Bigg] \tag{10}$$

where $[p_0, p_n]$ is compact notation for a linear or logarithmic scan. Let us call this a **2D Scan** with scan direction along the y-axis.

Alternatively, we could have fixed a point along the p-axis and performed a 1D scan along the v-axis, then stepped to the next point on the p axis and so on which would give the data

$$\text{data} = \Bigg[\Big[[v_0, v_m], p_0, [Q_1(v_0, p_0), Q_1(v_1, p_0), \ldots, Q_1(v_m, p_0)]\Big] \\ \Big[[v_0, v_m], p_1, [Q_1(v_0, p_1), Q_1(v_1, p_1), \ldots, Q_1(v_m, p_1)]\Big], \ldots \\ \Big[[v_0, v_m], p_n, [Q_1(v_0, p_n), Q_1(v_1, p_n), \ldots, Q_1(v_m, p_n)]\Big]\Bigg] \tag{11}$$

Again we call this a **2D Scan** with scan direction along the x-axis.

These two forms encapsulate how nearly all buffered 2D scans are actually carried out with instruments. Fix one independent variable, sweep the other while measuring some quantity, step and repeat.