**Motivation**

Although Pandas has a spot-on interface and it is full of useful functionalities, it lacks performance and scalability. For example, it is hard to decipher intensive intraday data such as Options data or S&P500 constituents tick-by-tick data using Pandas.

Another issue that I have encountered is, often, the research is done using Python, because it has such tools as Pandas, but the execution in production is in C++ for its efficiency, reliability and scalability. Therefore, there is this translation, or sometimes a bridge, between research and executions.

Also, in this day and age, C++ needs a heterogeneous data container. Mainly because of these factors, I implemented the C++ DataFrame.

**Code structure**

The DataFrame library is almost a header-only library with one source file exception, *BaseContainer.cc*.

Starting from the root directory;

*DMScu* is a helper module that contains a few objects. One is a stack-based string object and the other is a Linux-only mmap file interface. These objects are used by the DataFrame library and therefore must be compiled beforehand (see build instructions below).

*include* directory contains most of the code. It includes *.h* and *.tcc* files. The later are C++ template code files. The main header file is *DataFrame.h*. It contains the entire DataFrame object and its interface. There are comprehensive comments for each interface call in that file. The rest of the files there will show you how the sausage is made.

*src* directory has the only source file for the library, make-files, and a test program source file. The test source file is *datasci\_tester.cc*. It contains test cases for almost all functionalities of DataFrame. It is not in a very organized structure. I plan to make the test cases more organized.

**Build Instructions**

Using plain make and make-files

Go to the root of the repository, where license file is, and execute *build\_all.sh*. This will build the library and test executables for Linux flavors.

Using cmake

Coming soon for Linux, Windows, Mac, …

**General**

This library is based on a heterogenous vector. The heterogeneity is achieved by using static STL or STL-like vectors. Since C++ is a strongly typed language, you still have to know your column types per container at compile time. You can add more columns with different types at any time to your container, but when analyzing the data at any given time you must know the column types.

This is an example of how to create a DataFrame, load data, and run an operation on it:

// Defines a DataFrame with unsigned long index type that used std::vector

typedef DataFrame<unsigned long, std::vector> MyDataFrame;

MyDataFrame df;

std::vector<int> intvec = { 1, 2, 3, 4, 5 };

std::vector<double> dblvec = { 1.2345, 2.2345, 3.2345, 4.2345, 5.2345 };

std::vector<double> dblvec2 = { 0.998, 0.3456, 0.056, 0.15678, 0.00345, 0.923, 0.06743, 0.1 };

std::vector<std::string>. strvec = { "Col\_name", "Col\_name", "Col\_name", "Col\_name", "Col\_name" };

std::vector<unsigned long> ulgvec = { 1UL, 2UL, 3UL, 4UL, 5UL, 8UL, 7UL, 6UL }

std::vector<unsigned long> xulgvec = ulgvec;

// This is only one way of loading data into the DataFrame. There are

// many different ways of doing it. Please see *DataFrame.h* and *datasci\_tester.cc*

int rc = df.load\_data(std::move(ulgvec),

std::make\_pair("int\_col", intvec),

std::make\_pair("dbl\_col", dblvec),

std::make\_pair("dbl\_col\_2", dblvec2),

std::make\_pair("str\_col", strvec),

std::make\_pair("ul\_col", xulgvec));

// Sort the Frame by index

df.sort<MyDataFrame::TimeStamp, int, double, std::string>();

//Sort the Frame by column “dbl\_col\_2”

df.sort<double, int, double, std::string>("dbl\_col\_2");

// A functor to calculate mean, variance, skew, kurtosis, defined in *DFVisitors.h* file

StatsVisitor<double> stats\_visitor;

// Calculate the stats on column “dbl\_col”

df.visit<double>("dbl\_col", stats\_visitor);

|  |
| --- |
|  |
|  |
|  |

For more examples see file *datasci\_testr.cc*