

KINC v0.1 Specification



Knowledge Independent Network Construction

Joshua Burns*, Stephen Ficklin*

December 8, 2015

*Dept. of Horticulture, Washington State University

Contents

1	Introduction	3
2	Main Program	4
2.1	Console	4
2.1.1	ConsoleStream	5
2.2	Data	6
2.3	Analytic	9
2.4	Exceptions	10
2.4.1	DataException	10
2.4.2	AnalyticException	10
	Appendices	11
A	Data Classes	12
A.1	Expression	12
A.1.1	Properties	12
A.1.2	Constructor	12
A.1.3	Virtual Functions	12
A.1.4	File Structure	12
A.2	Correlation	13
A.3	Network	14
A.4	Annotation	14
B	Analytic Classes	15
B.1	Pearson	15
B.2	Spearman	15

1 Introduction

KINC is designed for use in construction of biological networks, specifically, gene co-expression networks.

The main program of KINC is broken up into three major classes. The [Console](#) class is responsible for controlling the program and presenting the user input and output interface with data or analytic objects. The abstract [Data](#) class is an interface that can be used to implement a data object which is responsible for storing certain types of biological data. The abstract [Analytic](#) class is an interface that can be used to implement an analytic object which is responsible for taking data object inputs and creating new data object from them using statistical methods.

2 Main Program

The main program consists of a console management class which acts as the program's controller and two abstract classes that create a common interface to the program. Any implementation data or analytic object must inherit and follow this interface.

2.1 Console

The `Console` class has a single instance within the main program and is designed to be given control through its `run()` function. This class creates a terminal console for the user, creates and manages all `Data` objects, and handles all `Analytic` execution.

This class also interfaces with the plugin object factory functions that generate new data or analytic interface object based off their unique names. These functions are implemented in a common source file which requires editing and recompiling whenever a new plugin is added to the program.

The KINC program upon loading presents the user with a console. [Figure 1](#) shows the basic commands this console will support.

Command	Description
history	Shows the history of a given data file.
load	Loads a new data file with the given type and name.
export	Exports a given data file to an ASCII export file.
query	Query a given data file for information.
merge	Merge two or more data files into one, if possible.
list	List all loaded data files.
gpu	GPU commands.

Figure 1: Basic Console Commands

The `history` command is handles by this class and prints out the history of the given data file that is loaded.

The `load` command is partially handled by this class. This command has two or three arguments. The first argument is always the data object type to be loaded. The next argument is the binary file of the data object. The third optional argument is a human readable text file that will be used as data to import into the data object if it is newer than the binary file.

The `export` command is handled by a data object. The first argument is the name of the data object to export. The second argument is the file name that it will be exported to as human readable text. Any other arguments are optional and passed along to the data object.

The `query` command is handled by a data object. The first argument is the name of the data object to query. All other arguments are optional and passed along to the data object.

The `merge` command is handled by the new data object that is merging two or more other data objects into a single data set. All arguments before the last argument are other data objects that will be merged into this new data object. The very last argument is the new data object that will contain the data of all merged inputs. Some data types cannot be merged and using this command will do nothing. All input data types must also be of the same type.

The `list` command is handled by this class and prints out a list of all currently loaded data objects.

The `gpu` command has a list of subcommands for querying and setting up any GPGPU device to be used with analytic objects. [Figure 2](#) shows all subcommands pertaining to GPGPU setup through OpenCL.

Command	Description
list	List all available OpenCL compatible devices.
set	Set a specific OpenCL device to be used for accelerated computation.
clear	Clear any previously set OpenCL device.

Figure 2: GPU Console Subcommands

Any other command will be treated as the name of an analytic object to load and execute. All arguments of an analytic command will be passed to the analytic. If no analytic is found with the given name nothing is done.

Figure 3 shows the public functions this class implements.

```

Console();
void run(int, char*[]);
bool add(Data*, std::string&);
bool del(std::string&);
Data* find(std::string&);
Data* new_data(std::string&);
Analytic* new_analytic(std::string&);

```

Figure 3: Functions for Console Class

The `run(...)` function takes control of the process and runs a console for the user or executes a script, depending on the command line arguments supplied. Once the user exits the console or the script has finished executing this function returns control to the caller.

The `add(...)` function adds a new `Data` object to the list of available `Data` objects within the console with the name string supplied. The name must be unique from all other loaded `Data` objects.

The `del(...)` function removes the `Data` object with the name string specified if it exists. If it exists and it was removed it returns `TRUE` else it returns `FALSE`.

The `find(...)` function finds a loaded `Data` object with the given name. If no object is found with that name then `NULL` is returned, else a pointer to the found object is returned.

The `new_data(...)` function creates a new data object of the type specified in the supplied string. If the string is not a valid data type then `NULL` is returned, else a pointer to the new data object is returned.

The `new_analytic(...)` creates a new analytic object of the type specified in the supplied string. If the string is not a valid analytic type then `NULL` is returned, else a pointer to the new analytic object is returned.

The `new_data(...)` and `new_analytic(...)` functions are designed to be plugin object factories which return data and analytic objects with the type given to them as a string. These two functions along with the list of all available plugins containing their unique string names and unique number identifiers are all contained in the same source and header file. These two files will represent where new static plugins that implement either a data or analytic interface can be added.

2.1.1 ConsoleStream

The `ConsoleStream` class is a common output interface to be used by all data and analytic objects for printing any output to the user. This class is responsible for taking the output of objects and directing it to the user interface.

Figure 4 shows the public functions this class implements. The constructor takes a single argument which tells an instance of its output is general, warning, or error output. An enumerated list will be defined in the class for each possible output.

```

ConsoleStream(int);
void print(short);
void print(unsigned short);
void print(int);
void print(unsigned int);
void print(long);
void print(unsigned long);
void print(float);
void print(double);
void print(const char*);
void print(const std::string&);
Console& operator<<(short);
Console& operator<<(unsigned short);
Console& operator<<(int);
Console& operator<<(unsigned int);
Console& operator<<(long);
Console& operator<<(unsigned long);
Console& operator<<(float);
Console& operator<<(double);
Console& operator<<(const char*);
Console& operator<<(const std::string&);
void flush();

```

Figure 4: Functions for ConsoleStream Class

The `print` functions takes the given output variable and prints it to its objects output type. The overloaded stream operators do the same thing.

The `flush` function will flush any output given to the stream and write it to the user interface.

This class will not be instantiated by any data object. There will be three objects of this class as public variables of the `Console` class. Figure 5 shows the three public objects of this class the `Console` class will possess and what they are.

Object	Description
out	Standard output used for normal notifications.
warn	Output used for warning messages.
err	Output used for error messages.

Figure 5: ConsoleStream Objects for Console Class

2.2 Data

The abstract `Data` class creates a common data object interface to the console program and provides file input/output for any data class implementing it. This class is responsible for implementing the basic file input and output operations along with specifying a common interface with the console program.

Additional functions can and should be added to any implementation of this interface class that deals with manipulating that specific type of data.

Figure 6 shows the public functions this class defines. Most of these functions are pure virtual functions that any class inheriting this class are required to implement. The constructor takes a single argument which is the binary file location where the data for an object is stored.

```
Data();  
void __history();  
virtual uint32_t type() = 0;  
virtual bool __load(std::string&) = 0;  
virtual bool __load(std::string&,std::string&) = 0;  
virtual bool __export(std::string&,std::vector<std::string>&) = 0;  
virtual bool __query(std::vector<std::string>&) = 0;  
virtual bool __merge(std::vector<std::string>&) = 0;  
virtual bool flush() = 0;
```

Figure 6: Functions for Data Class

The `__history()` function prints the entire history of its object to standard output.

The virtual `type()` function must return the unique identifier of the type of data this object represents. The list of unique identifiers are stored in a common header file.

The virtual `__load(...)` function is called when a load console command is issued on a new data object. The first argument is the file name of the binary file to load. The optional second argument is the location of a human readable file to import if it is newer than the binary file. The data object instance can assume that it is empty and was just created by the console. This function is responsible for loading data from a human readable ASCII file and encoding it into its binary format in a new file of the same name. This function returns `TRUE` if the command was successful else it returns `FALSE`.

The virtual `__export(...)` function is called when a export console command is issued on a given data object. The first argument is the location of the output file where the human readable format will be written to. The second argument is the list of remaining optional arguments. This function is responsible for exporting its internal data stored in binary format and decoding it into a human readable file. This function returns `TRUE` if the command was successful else it returns `FALSE`.

The virtual `__query(...)` function is called when a query console command is issued on a given data object. The single argument is the list of optional arguments. This function is responsible for giving information about the data this object holds to the user based off arguments given. This function returns `TRUE` if the command was successful else it returns `FALSE`.

The virtual `__merge(...)` function is called when a merge console command is issued on a given data object. The single argument is the list of data object names that will be merged into this new data object. This function is responsible for merging two or more separate data objects of the same type into a new single data object. The data object this command is called on is the new data object that will merge all data from all input data objects. This function does not have to merge the given data objects if not appropriate. This function returns `TRUE` if the command was successful else it returns `FALSE`.

The `__load(...)`, `__export(...)`, `__merge(...)`, and `flush()` functions will not return control to the caller of the function until all write operations to the binary file the data object represents have completed.

Figure 7 shows the protected functions this class implements that a class inheriting this abstract class can use for file input and output of its binary data. These functions are designed to hide the raw file from an implementation class so it is impossible to overwrite the header part of the file that this abstract class is responsible for maintaining.

```

bool fopen(const std::string&);
uint64_t fsize();
bool fgrow(uint64_t);
void fseek(uint64_t);
template<class T> bool fread(T*, uint64_t);
template<class T> bool fwrite(T*, uint64_t);

```

Figure 7: Protected Functions for Data Class

The `fopen(...)` function opens the file given as the binary file for this data object. It returns `TRUE` if the file was successfully opened or `FALSE` if it failed.

The `fsize()` function returns the total size of the binary file in bytes. This does not include the header information that is hidden from any implementation class.

The `fgrow()` function will increase the size of the binary file by the number of bytes given. If this was successful it returns `TRUE` else if it could not grow the size of the file it returns `FALSE`.

The `fseek()` function will move the file position's indent to the number given in bytes. This number must be within range of the total size of the binary file.

The `fread(...)` function reads the number of elements given and writes them to the pointer of that element type given, starting at the current file position and incrementing by the number of bytes read. The new file position cannot exceed the total size of the file.

The `fwrite(...)` function overwrites the number of elements given from the given pointer to the binary file, starting at the current file position and incrementing by the number of bytes overwritten. The new file position cannot exceed the total size of the file.

This abstract class is responsible for reading in the header information of any data file since it is generic to any specific data type. This part of the binary file is hidden from any implementation class.

Figure 8 shows the binary format for the beginning of a data file.

Name	Value	Description	Type
header	KINC	Special header tag specifying this is a KINC binary data file.	char[4]
type		Number that defines Data type for a file.	uint32_t
historySize		Total size of all history items in bytes.	uint32_t
history		Array of history items.	byte[historySize]

Figure 8: Binary File Format of Header

The `header` field is a special tag that specifies this is a KINC data file. The `type` field represents the specific type of data this file represents. The `historySize` field represents the total size of all history data in bytes.

Figure 9 shows the format for a single history item within the history buffer. History items are nested inside one another, the highest history element being the history of the current file, and all subhistories being the history for all input files.

Name	Description	Type
elemSize	The total size of this history item and all subhistory items in bytes.	uint64_t
fileLen	Length of file name string in bytes.	uint16_t
objectLen	Length of object name string in bytes.	uint16_t
commandLen	Length of command string in bytes.	uint16_t
date	Linux time-stamp of when file was last modified.	uint64_t
subHistoryAmt	Number of input history items.	uint16_t
subHistorySize	Size of all nested subhistory data in bytes.	uint32_t
file	File name string.	char[fileLen]
object	Name of object that built file in bytes.	char[objectLen]
command	Command line used in console to construct data file.	command[commandLen]
subHistory	Array of input history items.	byte[subHistorySize]

Figure 9: Binary File Format of Individual History Item

The `elemSize` field represents the total size of this history item, including all subhistory items nested within it.

The `file`, `object`, and `command` fields represent the name of the files, the object that created the data, and the specific KINC console command that invoked the creation of the file, respectively. The `fileLen`, `objectLen`, and `commandLen` give the length of the proceeding character strings.

The `date` field represents the date when this data file was last modified.

The `subHistoryAmt` field gives the total number of input history items for this history item. This does not include any input history nested within those history items. The `subHistorySize` represents the total size of all nested history items in bytes, respectively. The `subHistory` field contains the list of all nested subhistory items contained within this history item.

2.3 Analytic

The abstract `Analytic` class creates a common analytic interface to the console program. This class is almost a pure virtual class and only supplies a pointer to an OpenCL context from its constructor, provided to it from the main console.

Figure 10 shows the virtual functions an implementation of this class must define along with its constructor. The constructor takes a pointer to an OpenCL context that can be used for accelerated computation of the data, but it is not required and can be given `NULL`. An implementation of this class is required to provide a means to compute its data with or without an OpenCL device.

```
Analytic(cl::Context*);
virtual uint32_t type() = 0;
virtual bool execute(std::vector<std::string>&) = 0;
```

Figure 10: Functions for Analytic Class

The virtual `type()` function must return the unique identifier of the type of analytic this object represents. The list of unique identifiers are stored in a common header file.

The virtual `execute(...)` function is called for this analytic to execute the task it is designed to implement. The argument passed to it is a list of additional arguments given to the specific command issued by the user on the console. This command must be blocking until all file input/output has completed.

Typically this class takes input from one or more data files and creates a new data file as a result. This class is responsible for communicating to these individual data objects and creating a new data object if one is to be created. This class must know all pertinent functions involved in all data types it is responsible for reading and writing to.

2.4 Exceptions

Exception classes are defined for data and analytic objects. Any class implementing a data or analytic type must use their respective exception classes when throwing any exception. If a data or analytic object throws an exception the console will catch it and if it is a data object from it from the list of data objects. In either case the console will report an error to the user about the object throwing an exception and failing.

Figure 12 shows the public functions this class implements. The constructor is given 4 arguments. These arguments are the name of the file the exception occurred, the function where it occurred, the line number where it occurred, and a textual description of the type of exception, respectively.

```
DataException(const char*,const char*,int ,const char*);  
const char* file();  
const char* func();  
int line();  
const char* what();
```

Figure 11: Functions for Exception Class

2.4.1 DataException

Inherits from `Exception` Class.

The `DataException` class must be used for all data classes when reporting errors.

Figure 12 shows the public functions this class implements. The constructor is given 5 arguments. These arguments are the name of the file the exception occurred, the function where it occurred, the line number where it occurred, a pointer to the data object and threw it, and a textual description of the type of exception, respectively.

```
DataException(const char*,const char*,int ,Data*,const char*);  
Data* who();
```

Figure 12: Functions for DataException Class

The `who()` function returns a pointer to the data object and threw an exception.

2.4.2 AnalyticException

Inherits from `Exception` Class.

The `AnalyticException` class must be used for all data classes when reporting errors.

Figure 13 shows the public functions this class implements. The constructor is given 5 arguments. These arguments are the name of the file the exception occurred, the function where it occurred, the line number where it occurred, a pointer to the data object and threw it, and a textual description of the type of exception, respectively.

```
AnalyticException(const char*,const char*,int ,Analytic*,const char*);  
Analytic* who();
```

Figure 13: Functions for AnalyticException Class

The `who()` function returns a pointer to the analytic object and threw an exception.

A Data Classes

A.1 Expression

The Expression class is responsible for mangling gene expression-level data.

A.1.1 Properties

Do we need any properties?

I don't think we need properties. I am also unsure how to implement them in C++. My thought is all interactions between the classes will be defined in the Abstract Classes section using virtual functions?

A.1.2 Constructor

```
Data(int argc, char *argv[])
```

We need to design how the functions of the class will receive arguments. will we have a constructor that receives, parses and responds to errors for all functions? Or should each function be responsible for checking it's own arguments. I know we can't do that in the abstract class, but we need to accomdate the behavior we settle on in our design so plugins are consistent.

I almost completely agree. These functions and interactions will all be defined in the abstract classes section if that is OK? It is standard C++ to define everything you are talking about in the abstract interface class with virtual functions. It is usually a good idea to have a default constructor only for implemenation classes, and have any additional configuration added into additional virtual functions that any implementation must handle.

A.1.3 Virtual Functions

The following functions should be implemented by any plugin that creates classes that inherits the Data class.

```
virtual void import() = 0
```

This function reads a tab-delimited file. Each line of this file represents the gene expression levels of a single gene, transcript or probeset. Each tab-separated value in a single line indicates the gene expression level for each sample. The expression level of a samples must be in the same order for every line. The first line of the file may contain a tab-delimited list of sample names, and a file may contain as many samples and genes as desired.

A.1.4 File Structure

Figure 14 shows the binary format of expression data and how it is stored on file. `geneAmt` and `sampleAmt` give the total number of genes and samples in the data, respectively. `geneNames` is the list of all gene names as a string who's length and partitioning is defined by `geneNameLen` and `geneNameSize`. `sampleNames` is the list of all sample names as a string who's length and partitioning is defined by `sampleNameLen` and `sampleNameSize`. Lastly, `samples` is 2 dimensional matrix of all samples for each gene, where the matrix is sorted by gene major order.

Name	Description	Type
geneAmt	Total number of genes.	uint32_t
sampleAmt	Number of samples per gene.	uint32_t
geneNameLen	Length of each string identifying genes.	uint16_t
geneNameSize	Total size of gene name list in bytes.	uint64_t
sampleNameLen	Length of each string identifying samples.	uint16_t
sampleNameSize	Total size of sample name list in bytes.	uint64_t
geneNames	List of gene string identifiers.	char[geneNameSize]
sampleNames	List of sample string identifiers.	char[sampleNameSize]
sampleTotal	Total number of samples for all genes.	uint64_t
samples	List of all samples per gene.	float[sampleTotal]

Figure 14: Binary File Format of Expression Data

A.2 Correlation

This is responsible for storing correlation data between genes.

The following describes the format of the KINC correlation file. All multi-byte numbers are little-endian, regardless of the machine endianness.

I like this type of table for describing the file format. I borrowed it from the BAM file specification

So do I! I was actually going to convert these definitions to a tabular format after your first review. :)

Figure 15 shows the binary format of correlation data and how it is stored on file. `geneAmt`, `sampleAmt`, and `corrAmt` give the number of genes, number of samples per gene, and number of correlations per gene, respectively. `geneNames` is the list of all gene names that are correlated who's length and partitioning is defined by `geneNameLen` and `geneNameSize`. `sampleNames` is the list of all sample names used for correlation between genes who's length and partitioning is defined by `sampleNameLen` and `sampleNameSize`. `corrTypes` is the list of all correlation types listed for all gene pairs who's length and partitioning is defined by `corrTypeLen` and `corrTypeSize`. Lastly, `correlations` is a special diagonal matrix where all correlations for gene pairs are stored using gene major order.

Name	Description	Type
geneAmt	Total number of genes.	uint32_t
sampleAmt	Number of samples per gene.	uint32_t
corrAmt	Number of correlations per gene relationship.	uint8_t
geneNameLen	Length of each string identifying genes.	uint16_t
geneNameSize	Total size of gene name list in bytes.	uint64_t
sampleNameLen	Length of each string identifying samples.	uint16_t
sampleNameSize	Total size of sample name list in bytes.	uint64_t
corrTypeLen	Length of each string identifying correlation type.	uint16_t
corrTypeSize	Total size of correlation type list in bytes.	uint16_t
geneNames	List of gene string identifiers.	char[geneNameSize]
sampleNames	List of sample string identifiers.	char[sampleNameSize]
corrTypes	List of correlation type strings.	char[corrTypeSize]
corrTotal	Total number of correlations for all gene relationships.	uint64_t
correlations	Diagonal matrix list of all gene correlations for all relationships.	float[corrTotal]

Figure 15: Binary File Format of Correlation Data

A.3 Network

This is responsible for storing network data between genes.

Figure 16 shows the binary format of network data and how it is stored on file. `geneAmt` give the number of genes in the network. `geneNames` is the list of all gene names that are correlated who's length and partitioning is defined by `geneNameLen` and `geneNameSize`. Lastly, `network` is a special diagonal matrix where all network edges for gene pairs are stored using gene major order.

Name	Description	Type
geneAmt	Total number of genes.	uint32_t
geneNameLen	Length of each string identifying genes.	uint16_t
geneNameSize	Total size of gene name list in bytes.	uint64_t
geneNames	List of gene string identifiers.	char[geneNameSize]
netTotal	Total number of edges, true or false, in network data.	uint64_t
network	Diagonal matrix list of all possible edges in gene network.	bool[netTotal]

Figure 16: Binary File Format of Network Data

A.4 Annotation

This is responsible for storing additional information for genes.

Figure 17 shows the binary format of annotation data and how it is stored on file. `geneAmt` and `annotAmt` give the number of genes and the number of annotations, respectively. `geneNames` is the list of all gene names who's length and partitioning is defined by `geneNameLen` and `geneNameSize`. `annotNames` is the list of all annotation names who's length and partitioning is defined by `annotNameLen` and `annotNameSize`. `annotValSize` is a list of all string lengths for each annotation value per each gene. Lastly, `annotations` is a 2 dimensional matrix that lists all annotations for all genes using gene major order.

Name	Description	Type
geneAmt	Total number of genes.	uint32_t
annotAmt	Total number of annotations per gene.	uint32_t
geneNameLen	Length of each string identifying genes.	uint16_t
geneNameSize	Total size of gene name list in bytes.	uint64_t
annotNameLen	Length of each string identifying the name of a annotation.	uint16_t
annotNameSize	Total size of annotation name list in bytes.	uint64_t
annotNames	List of annotation string identifiers.	char[annotNameSize]
geneNames	List of gene string identifiers.	char[geneNameSize]
annotValLens	List of numbers that identify the length of each value string for each annotation.	uint16_t[annotAmt]
annotValSize	Total size of all annotation values.	uint64_t
annotations	List of all annotations per gene.	char[annotValSize]

Figure 17: Binary File Format of Annotation Data

B Analytic Classes

B.1 Pearson

This takes an Expression BioData object and produces a Correlation BioData object. It uses the Pearson correlation statistical method for giving correlation values.

B.2 Spearman

This takes an Expression BioData object and produces a Correlation BioData object. It uses the Spearman correlation statistical method for giving correlation values.