Building Matlab Standalone Package from Java for Differential Dependence Network Analysis Bioinformatics Toolkit

Lu Jin

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the
degree of

Master of Science
In
Computer Engineering

Yue Wang, Chairman
Jason Xuan
Chang-Tien Lu

May 26, 2010
Arlington, Virginia

Keywords: Matlab Compiler, Java Native Interface, Differential Dependence Network, C shared library, C driver, Java driver

Building Matlab Standalone Package from Java for Differential Dependence Network Analysis Bioinformatics Toolkit

Lu Jin

Abstract

This thesis reports a software development effort to transplant Matlab algorithm into a Matlab license-free, platform dependent Java based software. The result is almost equivalent to a direct translation of Matlab source codes into Java or any other programming languages. Since compiled library is platform dependent, an MCR (Matlab Compiler Runtime environment) is required and has been developed to deploy the transplanted algorithm to end user. As the result, the implemented MCR is free to distribution and the streamline transplantation process is much simpler and more reliable than manually translation work. In addition, the implementation methodology reported here can be reused for other similar software engineering tasks.

There are mainly 4 construction steps in our software package development. First, all Matlab *.m files or *.mex files associated with the algorithms of interest (to be transplanted) are gathered, and the corresponding shared library is created by the Matlab Compiler. Second, a Java driver is created that will serve as the final user interface. This Java based user interface will take care of all the input and output of the original Matlab algorithm, and prepare all native methods. Third, assisted by JNI, a C driver is implemented to manage the variable transfer between Matlab and Java. Lastly, Matlab mbuild function is used to compile the C driver and aforementioned shared library into a dependent library, ready to be called from the standalone Java interface.

We use a caBIG^{TM} (Cancer Biomedical Informatics Grid) data analytic toolkit, namely, the DDN (differential dependence network) algorithm as the testbed in the software development. The developed DDN standalone package can be used on any Matlab-supported platform with Java GUI (Graphic User Interface) or command line parameter. As a caBIG^{TM} toolkit, the DDN package can be integrated into other information systems such as Taverna or G-DOC. The major benefits provided by the proposed methodology can be summarized as follows. First, the proposed software development framework offers a simple and effective way for algorithm developer to provide novel bioinformatics tools to the biomedical end-users, where the frequent obstacle is the lack of language-specific software runtime

environment and incompatibility between the compiled software and available computer platforms at user's sites. Second, the proposed software development framework offers software developer a significant time/effort-saving method for translating code between different programming languages, where the majority of software developer's time/effort is spent on understanding the specific analytic algorithm and its language-specific codes rather than developing efficient and platform/user-friendly software. Third, the proposed methodology allows software engineers to focus their effort on the quality of software rather than the details of original source codes, where the only required information is the inputs and outputs of the algorithm. Specifically, all used variables and functions are mapped between Matlab, C and Java, handled solely by our designated C driver.

Key words: Matlab Compiler, Java Native Interface, Differential Dependence Network, C shared library, C driver, Java driver

Acknowledgements

First I would like to thank my beloved family for their loving considerations and great confidence in me through all these years.

My deepest gratitude goes to Dr. Yue Wang, my supervisor, for his constant encouragement and guidance. He provides me the thesis topic and walked me through all the steps of the writing of this thesis. Without his consistent and illuminating instruction, this thesis could not have reached its present form.

Second, I would like to express my heartfelt gratitude to Dr. Huai Li, who led me into the heart of this project, continually provide constructive suggestions which helped me to conquer the technical difficulties one by one. I am also greatly indebted to Dr. Jianhua Xuan, who have instructed and helped me a lot in the past two years.

I want to express my sincere gratitude to the other committee members, Dr. Chang-Tien Lu. He has provided important insights that are very precious to me and his suggestions have significantly improved this thesis.

Last my thanks would go to Dr. Robert Clarke who supported me all through this project. I also owe my sincere gratitude to my friends and my fellow lab mates in the Computational Bioinformatics and Bio-imaging Laboratory and Dr. Clarkes Lab who gave me their help and time in listening to me and helping me work out my problems during the difficult course of the thesis.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Motivation

In today's biomedical research, tons of new data are being generated every day. A high throughput efficient tool is needed for researchers to extract useful information from such data pool. With the help of various toolboxes, Matlab is a widely used programming language for developing bioinformatics analytic tools. In particular, the vector semantics and powerful visualization capabilities of Matlab are powerful features for efficient algorithm development, testing, and applications. However, the major limitation associated with Matlab-based software is that the actual use of the developed Matlab software requires an Matlab license runing on every single computer due to the close-source nature of Matlab. This requirement significantly limits the distribution and utility of any newly developed data analytic algorithm written in Matlab. It makes no sense for every user to have a Matlab license if the user simply wants to try a reported new algorithm. In order to eliminate the restrictions imposed by Matlab licensing, one frequent solution used by Matlab programmers is to create a source code of the same algorithm in other programming language, where the original algorithm is compiled into an executable application that can be assisted by a bit of interfacing code. No matter which language is used for translation, manually translating algorithmic model to other code can be a laborious task. As a result, developers tend to avoid making this translation until late in the design process, or other people will do this extension work instead later. This increases the length of the deployment process. Manual translation is also an error-prone process. This makes it difficult to determine if functional bugs are the result of a poor algorithm or a translation error. Matlab compiler provide a friendly feature called Matlab Compiler Runtime (MCR), with which user can build standalone executables and software components from Matlab. Also, some related work have been presented [1-3]. [1]shows an open-source Matlab-to-Python compiler. [2] provides a one-button Matlab-to-C conversion tool called Catalytic MCS, which converting Matlab source code into C code automatically. [3] uses a library in R called Rmatlab package as an interface to Matlab.

It is known that C-based languages have a system-related portable problem. In C and C++, each implementation decides the precision and storage requirements for basic data types (short, int, float, double, etc.). This is a major source of porting problems when moving from one kind of system to another, since changes in numeric precision can affect calculations and assumptions about the size of structures can be violated. Java defines the size of basic types for all implementations: an int on one system is the same size as the one on every other system. Although C and C++ are supported on all platforms that support Java, these languages are not supported in a platform-independent manner. C and C++ applications moving between OS platforms require recompilation as a minimum, in most cases, significant

redesign. Java programs are compiled into Java virtual machine code called bytecode. The bytecode is machine independent and is able to run on any machine that has a Java interpreter (JRE).

Python, as another programming language, also has great application potential. It almost overlaps with every aspect of Java, but with less support, slower runtime speed and higher system resources consumption. Java is more mature overall, and still the most popular programming language according to Tiobe programming community index 2010, while Python as a developing language is at seventh. Moreover, the integration platform called Taverna, which we will be using currently only open doors to Java script or R but not Python. Regarding the applicability of R language, according to the information provided on the Omega project website, Rmatlab package works the same way as transplant Matlab to Java, while has much less ability of creating high quality user interface as compared to Java.

In our effort, with the purpose of accommodating various Matlab's advantages appreciated by algorithm developers and also offering the end-users state-of-the-art bioinformatics software tools, we have chosen Java to be the development language of the standalone application. The overall goal of this thesis is to create a method, which can be used repeatedly to make algorithm developed within Matlab run all cross the platform and independently from Matlab. This method promises to have the lowest translation risk among all the other choices and the outcome package can be easily integrated with other applications.

1.2 Background

The main concept behind this thesis comes from the design principle of caBIG (cancer biomedical informatics grid) research plan. As a funded caBIG developer, in order to move the outcomes of our novel breast cancer classification algorithms through high-throughput, pathway-centric annotation, and ultimately into *in silico* drug discovery, we have developed an integrated workflow (**Fig 1**) that brings together the multidisciplinary expertise from medical oncology, basic cellular and molecular research, caBIG technology development, bioinformatics, biostatistics and project management. caBIG initiative was launched by the National Cancer Institute as a 21st century information system that will transform the way we do cancer research. They are creating a network that will freely connect the entire cancer community. In doing so, caBIG is leveraging valuable resources and saving precious time toward new discoveries. This is an open source network that enabling members of the cancer community – researchers, physicians, and patients to share data and knowledge. Nowadays, the component of caBIG is widely applicable beyond cancer as well. To help with the standardization process, we provide details of the research workflow that connects the various pieces of the proposed research focusing on breast cancer. While this workflow

currently depicts tasks that are application-specific, it can be adapted for additional/alternative studies in future and for other similar applications.

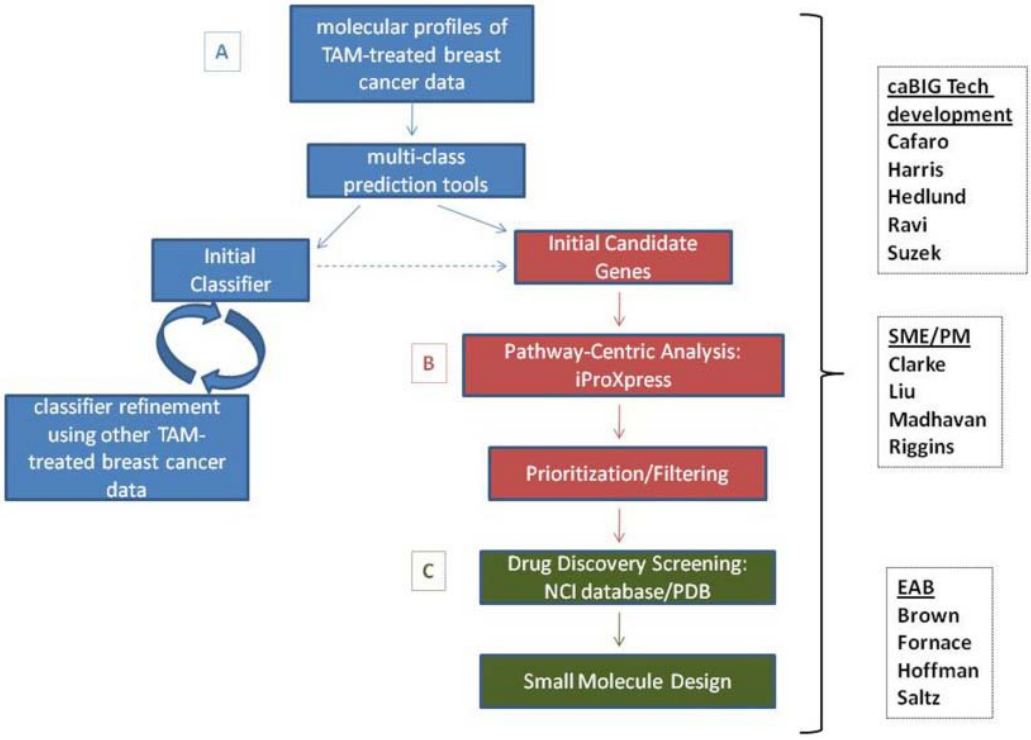


Figure 1 Overall Study Workflow (from caBIG ISRCE research plan)

A variety of new computational methodologies have been developed at Computational Bioinformatics and Bio-imaging Laboratory (CBIL) to build predictive models for breast cancer diagnosis, prognosis and therapy using public data. The Georgetown caBIG in silico Research Center for Excellence (ISRCE) project aim1 is to develop breast cancer classification algorithms. The workflows of the proposed methods are shown in Fig 2. Data to be used include published breast cancer gene expression datasets on resistance and recurrence of estrogen receptor-positive breast cancer treated with TAM. These are primarily Affymetrix U133 Plus2.0 data, and non-Affymetrix data will be mapped across the platforms. In the workflow of gene expression data analysis, we will acquire all array datasets from G-DOC server and use G-DOC web interface to perform data pre-processing and further analysis. G-DOC is an informatics framework that allows researchers, using unified data portals, to access and analyze clinical and research data across multiple trials and studies. The framework can be used to import data from multiple studies, to access biomedical research data, to perform analysis, and generate ad hoc queries and customized reports. Moreover it provides a mechanism for integrating and aggregating biomedical research data and provides

access to variety of data type (e.g. tissue annotations, gene expression, clinical trials data, health service related population data, etc.) in an integrated fashion. For the further analysis We mentioned above, as shown in Fig 2.

There are 4 new classification and prediction approaches will be implemented in this project: (1) the union of phenotype up-regulated genes(PUGs) within a one-versus –rest support vector machine (OVRSVM) framework (PUG-OVRSVM); (2) the support vector machine with recursive feature elimination (SVM-RFE); (3) the differential dependence network (DDN); and (4) the fused margin regression (FMR) method. This thesis uses DDN Matlab algorithm as the testbed in the planned software development. DDN extracts the network structure information from microarray gene expressions and provides an alternative means of biomarker identification. Under two different conditions, the topology of the underlying gene regulatory network changes and the genes that undergo the most network topological changes are identified as candidate biomarkers or drug targets. Here, we will first construct a differential dependence network model, represented by a set of local conditional probabilities and later use the model to perform topology-based classification for resistance and recurrence after tamoxifen treatment.

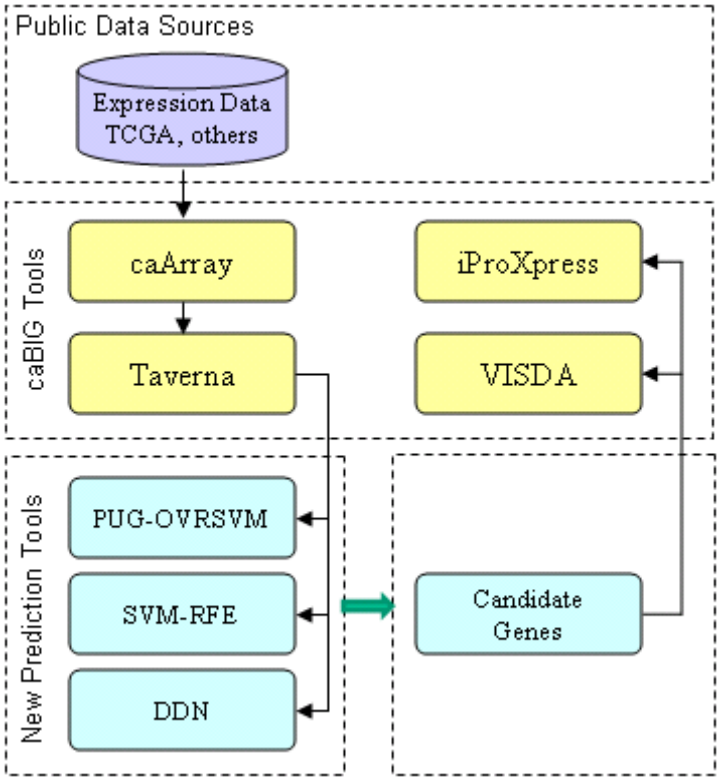


Figure 2 Algorithm Development Workflow (from caBIG ISRCE research plan)

1.3 Organization of the thesis

In order to transplant Matlab algorithm into a Matlab license-free, platform dependent Java based software, we need to manage the variable mapping between Matlab, C and Java. Since compiled C library is platform dependent, an MCR (Matlab Compiler Runtime environment) is required and has been developed to deploy the transplanted algorithm to end user. As a result, the implemented MCR is free to distribution and the streamline transplantation process is much simpler and more reliable than manually translation work. In addition, the implementation methodology reported here can be reused for other similar software engineering tasks.

In chapter 2, we address the transplantation requirements and methods in detail. There are 4 major steps we need to mention which mostly common in every transplantation job. They are shared library generation, Java driver development, C driver development and jni stub library compilation. The first and fourth steps are managed by mcc and mbuild functions within Matlab. We will briefly talk about that and then focus on the developments of two drivers, because those are the core parts of this method.

In chapter 3, based on method presented in chapter 2, we use differential dependency network package (DDN) as an example to perform an actual transplantation. Within this chapter, you can see clearly why a random Matlab algorithm can be integrated into Java without knowing any specific variable passes between each m-function. The only information we need to acquire is the input and output variable type. At the same time, the whole transplantation process is described more into detail. If the whole thesis structure is considered as a 5 layer network model, chapter 2 is can be seen as transport layer and chapter 3 is at the application level.

After showing the development of the DDN package, in chapter 4, we will present the deployment of transplanted DDN package. There are major changes for the environment variable settings in different platform deployments (Windows, Linux). Then we will show you a demo of breast cancer research, during which we will explain the potential error or warning messages for this method. Finally we will show the overhead of this method and its efficiency compare to the original Matlab source code.

At last, we will discuss the future work in chapter 5. After freed from Matlab license, the java standalone package (*.jar) has many choices. It can be used independently as described in chapter 4. Also, with affordable changes, it can be integrated into many popular platforms or web servers, such as Georgetown database of cancer (GDOC) and the standard user interface for caBIG --Taverna.

Chapter 2 Development Method

2.1 Requirements and specifications

Since the Matlab compiler only compiles a library to be run from C/C++ code [4], to run the library from Java, the Java Native Interface is used to convert Java variables to and from C/C++ code. A C/C++ driver is created to map the Java variables into a form usable by the Matlab library. As a highly popular programming language with ample mathematical toolboxes, Matlab has a fast developing speed, and has at least two major updates every year (e.g., many bugs are fixed and also small feature changed). For this thesis, the software is developed based on Matlab 2009b, along with version 4.11 Matlab Compiler. For the Java software development, Java Development Kit (JDK)version is 1.6.0 or newer, where NetBean 6.8 is used as the development tool. It shall be noted that all these environment settings is important due to the requirements of Matlab Compiler. Specifically, Matlab Compiler uses the Matlab Compiler Runtime (MCR), which is a standalone set of shared libraries that enable the execution of M-files. In every Matlab Compiler, there is a deployed version of MCR, which is version-specific. MCR provides complete support for all features of Matlab without Matlab GUI. When a software application is packaged and distributed to users, the supporting files generated by the builder as well as the Matlab Compiler Runtime (MCR) are included. User can only run the applications with the specific version of the MCR that is created by the specific version of Matlab Compiler. The final form of the integrated work will be in jar or at least a .class file, which also required the user to have the same 1.6.0_xx version of JRE in their machines.

According to NCI, Taverna, a caBIG standard platform, only opens doors to Java and R programming language. If we choose R to be our major development language, which means we need to translate all of our existing algorithms, no matter build in C or Matlab or Java to R. Obviously, it is a tedious and error-prone process. It is preferred that Taverna 2 will provide a way for Ruby, Python etc. script author to link their services or module into the standard workflow. If so, OMPC will become a popular translation tool for later integration development. However, for now we need to figure out a way to “translate” Matlab code into Java. The major challenge is that the Matlab compiler only compiles a library to be run from C/C++ code. To run the library from Java, the Java Native Interface (JNI) is used to convert Java variables to and from C/C++ code. This powerful feature is desirable or necessary to work closely with native code written in other languages when Java platform is deployed on top of host environments. The JNI can support two types of native code: native libraries and native applications. For this thesis, we mainly use the JNI to write native methods that allow Java applications to call functions implemented in native libraries. This works the same way

as call methods implemented in the Java programming language. JNI have such a useful feature but at the same time it loses some benefits of Java platform. First, Java applications that depend on JNI can no longer readily run on multiple host environments, although the interface or application part written in Java programming language is portable across platform, it will be necessary to recompile the code written in native programming language. In our case, since Matlab variables do not talk to Java variables directly, native C libraries play an important role in this process. This is the only part of code that is platform-dependent and needs to be recompiled under different hardware architecture. The flowchart of the entire software development process is summarized in Fig. 3 that shows the variable flow between Java, C and Matlab.

Java Variable ←→ C variable ←→ Matlab variable

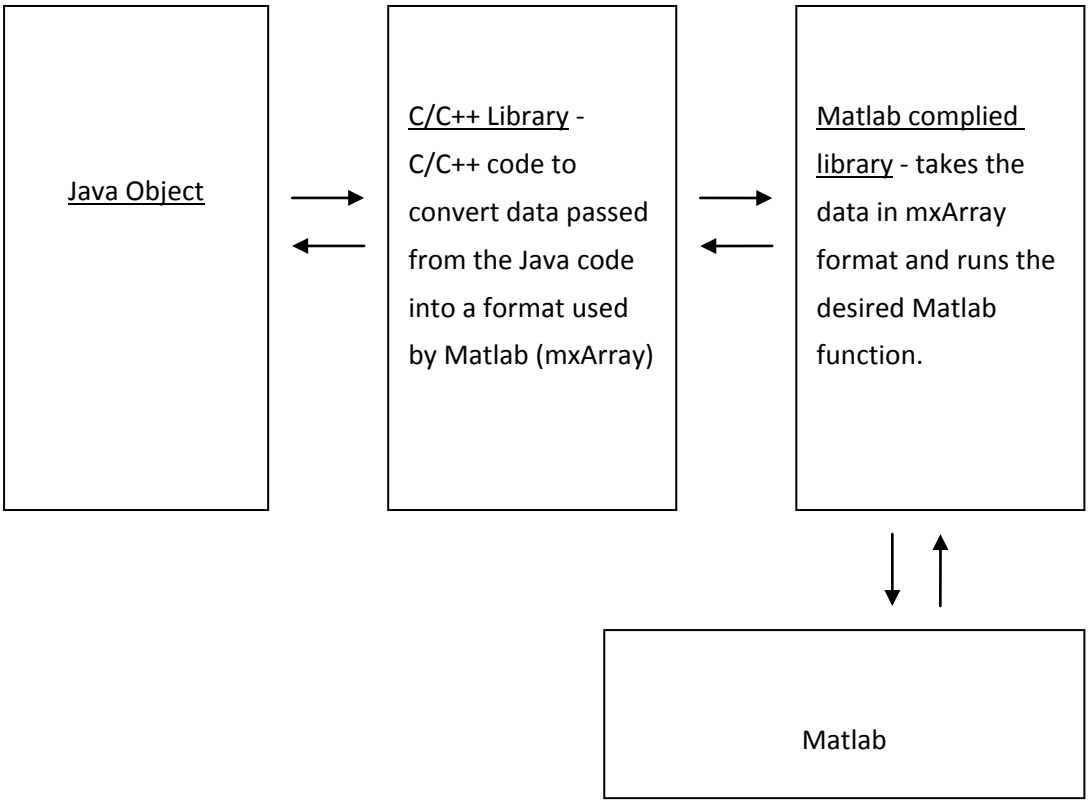


Figure 3 Variable flow chart for developing standalone package

The C/C++ code maps the Java variables to C variables, then the C variables to Matlab variables (mxArray). Specifically, there is a set of JNI methods that perform the conversion to C variables and a set of Matlab methods that perform the conversion to mxArray. Only mxArray is a valid variable in Matlab. All primitive type data (C variables types), like int and double are treated as a 1×1 matrix in Matlab.

2.2 Architectural design

The integration package contains 4 major components. (1) C shared library created by Matlab compiler (2) self-written Java Driver with GUI (3) self-written C Driver (4) Matlab compiled JNI shared library depended on C shared library.

2.2.1 C shared library

Matlab Compiler allows the user to create a shared library from any sets of Matlab files. mcc is the Matlab command that invokes Matlab Compiler. It generates a .dll wrapper file (for windows), a header file and an export list. The header file contains all of the entry points for all the compiled Matlab functions. The command line is like this:

mcc -vB csharedlib:libfilename main.m file1.m file2.m file3.m …

Where option v means Verbose which display the compilation steps, including Matlab Compiler version number, the source file names as they are processed, the names of the generated output files as they are created and the invocation of mbuild. Option B means specify bundle file, which provide a convenient way to group sets of Matlab Compiler options and recall them as needed, for a C shared library –B equals –W lib:<shared_library_name> -T link:lib

There are 7 files created after this command:

libfilename.c The library wrapper C source file containing the exported functions of the library representing the C interface to all the Matlab functions (main.m, file1.m, file2.m file3.m, …) as well as library initialization code.

libfilename.h The library wrapper header file. This file is included by applications that call the exported functions of libfilename.

libfilename_mcc_component_data.c C source file containing data needed by the MCR to initialize and use the library. This data includes path information, encryption keys, and other initialization for the MCR

libfilename.exports The exports file used by mbuild to link the library.

libfilename.dll The shared library binary file. This is the format for windows. In Linux the extension is so.

libfilename.exp Exports file used by the linker. The linker uses the export file to build a program that contains exports, usually a dynamic-link library (.dll). The import library is used to resolve references to those exports in other programs.

libfilename.lib An import library is used to validate that a certain identifier is legal, and will be present in the program when the .dll is loaded. The linker uses the information from the import library to build the lookup table for using identifiers that are not included in the

.dll. When an application or .dll is linked, an import library may be generated, which will be used for all future .dll s that depend on the symbols in the application or .dll.

According to Matlab help, when invoking the Matlab Compiler, it examines the input Matlab files and the external dependency database to determine the complete list of all functions used by the application or component. As part of this process, Matlab compiler determines which files are not able to be compiles and automatically excludes them.

2.2.2 Java Driver

Java driver contains all the native methods corresponding to the functions in the library create in 2.2.1. All the programs that call Matlab Compiler generated libraries have roughly the same structure:

1. It is a requirement to call applicationInitialize() once at the beginning of the driver application. Before calling any other functions in DDN algorithm or accessing any type of Matlab variables, this function set up the global MCR state and enables the construction of MCR instances

   eg. public static native void applicationInitialize();

2. Call libInitialze() once to create the MCR instance required by the library.
   eg. public static native void libInitialize();
3. Invoke functions in the library, and process the results, this is the main body of the program
   eg. public native void filename(String a, String b, String c, double d, double e, double f, String g);
4. Call libTerminate() once after step 2.2.3 is completely done, to destroy the associated MCR. There is one thing to notice though, after the library was terminated, any further libInitialize() call will generate error message says the library cannot be initialized again, because in current session the library association has already been destroyed. This mechanism causes a problem. If a Java GUI can only be run once every time open it, it is definitely not a user-friendly tool. To solve this problem we separate these functions into 3 parts, we put all the initialize functions together with UI initialize method, put the main body function into OKActionPerformed method, which actually execute the algorithm and put all Terminate functions into exitActionperformed method, make sure the memory and resources are freed after the application exit. After the separation, Java GUI would work repeatedly as intended. If the user prefer or force to have a text based application, the situation is much simpler then GUI case. No matter the parameters are given along with the command line or input manually after each prompt, the application

always return to prompt state after runs a full process. Recall the initialize() after terminate() won't be a case. It is mentioned above that the whole variable transfer process goes from Java variables to C variables then to Matlab variables for the input stream. For the output stream, the process is exactly backwards.

Except the native methods, Java driver also contains the other methods that handle all the interfaces, data inputs and outputs as normal Java class. This part is different depends on the integration. Eventually, the Matlab algorithm will be presented by this Java class.

The most important thing in Java driver is the class loader. Java virtual machine must locate and load the implementation of the native methods. Usually these native methods are contained in a native library. Class loader helps to load native library and link classes in the Java virtual machine. Each class loader maintains a set of native libraries, which is created with Matlab mbuild function in 2.2.4. Native libraries are loaded by System.loadLibrary method, if the library location is in one of the default system library path, such as bin folder under JRE directory. Otherwise, System.load method can be used to load the library from an absolute path.

The last thing before finish this step is to use the javah utility to create a header file filename.h, which contains the definition of all C formed mapping functions. For most cases, this file should contains the declaration of applicationInitialize(), applicationTerminate(), all libInitialize() and libTerminate() plus the main method functions in the Matlab algorithm.

2.2.3 C Driver

Without this part, 2.2.2 makes no sense at all. Matlab do not recognize native methods in Java, C driver takes charge of the variable and function mapping between Matlab and Java by applying JNI. Basically the C driver can be separated into two parts as well as the Java Driver, one part takes care of the variable type mapping, the other takes care of the native method implementation.

For variable type mapping from Java to C, there are two kinds of types in the Java programming language, one is primitive types such as int, double, char and the other is reference types such as classes, instances, and arrays. String type is considered as instance from java.lang.string class, which is necessary to import at the beginning of the code. For primitive types, The JNI treats it in a straight forward way. Adding a letter "j" before each primitive type in Java programming language creates its one to one mapping to C type. For example, int to jint, double to jdouble, char to jchar. These mapping functions are defined in a header file called jni.h in Java. For reference types, all JNI references have a type jobject. But for convenience and enhanced type safety, JNI have some subtypes of jobject, such as jstring denotes strings, jobjectArray denotes an array of objects. At the same time, Matlab provide a variety of API to communicate with C. It can be seen clearly from the header file generated

by Matlab (libfilename.h), all types of input variables have been defined to mxArray type, which is the only variable type Matlab use to communicate with C. Output type depends on Matlab algorithms output type. These definitions were generated automatically by Matlab Compiler. It can be considered as a rule for how to deal with C programming language and to write a C Driver. All kinds of variables in C can be considered as mxArray, a fundamental variable type in Matlab. Now, it is clear that what we need to do is to create several methods to handle some most common types of variable transition. They are javamatrix, javadouble, and javastring. My first priority is to keep the mapping process as simple as possible. That's why all int, float are categorized as double, charater can be consider as a string, and matrix contains all kinds of arrays. If the Matlab algorithm has a return value, then we need to add three more methods to handle the mxArray type backward mapping, separately into javamatrix, javadouble and javastring, which depends on the return variable type.

For the other part of the C Driver, libfilename.h and filename.h guides the way of function implementation. The former one generated by Matlab Compiler, the later one generated by Javah. The second header file may differ between different native languages. The examples shown above are using C programming language as the native language. Usually, there are at least two pairs of functions control the library and application initialization and termination. The exact JNI function and method definition will be given separately in libfilname.h and filename.h. After all the preparation above, we add implementation of the core function, which will be the very function that actually execute the Matlab algorithm when called by Java Driver, its definition and JNI function are also available at these two header files. There are several tasks need to be covered in this function, declare variables, process input parameters or in other words process input arrays, because all variables should have the mxArray* format now, call core function, process output arrays and at last destroy all arrays and free the memory. Main thread will walk through this pipeline when Java Driver calls the native function. One thing need to notice though, it is extremely important to check, handle and clear any pending exceptions before calling any subsequent JNI function. If not, most of them will throw an undefined exception.

2.2.4 Compile library with C driver

This step is mostly handled by Matlab. Use the mbuild tool in Matlab to compile the JNI stub library into a shared library and link against the Matlab Compiler generated library in step (1). According to Matlab help, mbuild can also create shared libraries from C source code. If a file with the extension .exports is passed to mbuild, a shared library is built. The .exports file must be a text file, with each line containing either an exported symbol name, or starting with a # or * in the first column (in which case it is treated as a comment line). If multiple .exports files are specified, all symbol names in all specified .exports files are exported.

Chapter 3 DDN integration

3.1 DDN algorithm

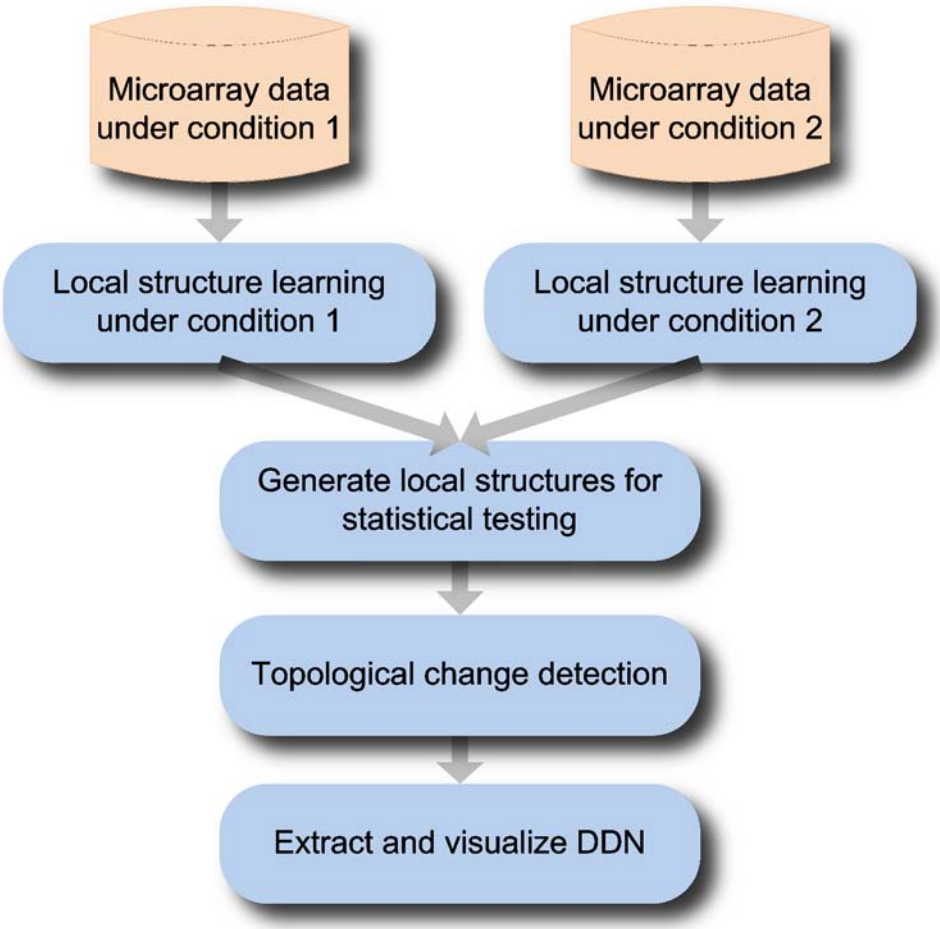


Figure 4 Flowchart of differential dependency network analysis (ref. DDN book chapter)

Before integration, we discuss differential dependency network (DDN) analysis as a new method to model and detect the statistically significant topological changes in transcriptional networks between two conditions. This discussion is based on the work proposed in [5]. The workflow is shown in Fig.4. We use local dependency models to characterize the dependencies of genes in the network and extract and represent local network substructures. Local dependency models decompose the entire network into a series of local networks, which serve as the basic network elements for subsequent statistical testing. Local dependency models select the number of dependent variables automatically by the Lasso method [6], and thereby learn the local network structures. Subsequently, we perform permutation tests on the local dependency models under two conditions and assign the p-values to the local structures. It may seem straightforward to construct an entire network

under each condition and compare the differences between the two networks. However, in realistic applications this approach runs into the difficulty that the network structure learning can be inconsistent with a limited number of data samples. When applied to the very high dimensional data produced by gene expression microarrays, the properties of the data impose additional constraints and complications [7]. The detection procedure proposed here assures the statistical significance of the detected network topological changes by performing a permutation test on individual local structures. We also pinpoint "hot spots" in the network where the genes exhibit network topological changes between two conditions above a given significance level. Lastly, we extract and visualize the DDN, i.e., the sub-networks exhibiting the most significant topological changes.

This approach utilizes the network structure information and provides an alternative way for biomarker identification. In addition, as knowledge of cellular networks accumulates, many biological databases will expand to contain more useful information. The proposed approach is an open framework, into which biological knowledge in specific applications can be easily incorporated as the local structure learning constraints. Some issues are worth further exploration. Currently, only linear relationships are considered. How nonlinear relationships should be modeled efficiently and correctly, remains a difficult problem. Second, since many cellular reactions take place in the genome, transcriptome, and proteome, it is essential to construct pathways by integrating data from heterogeneous sources.

In summary, DDN analysis presents a new approach to extract knowledge of a biological network by emphasizing the dynamic nature of cellular networks and utilizing a network's structural information. It also provides an alternative and promising approach to identify possible biomarkers and drug targets.

3.2 DDN integration structure

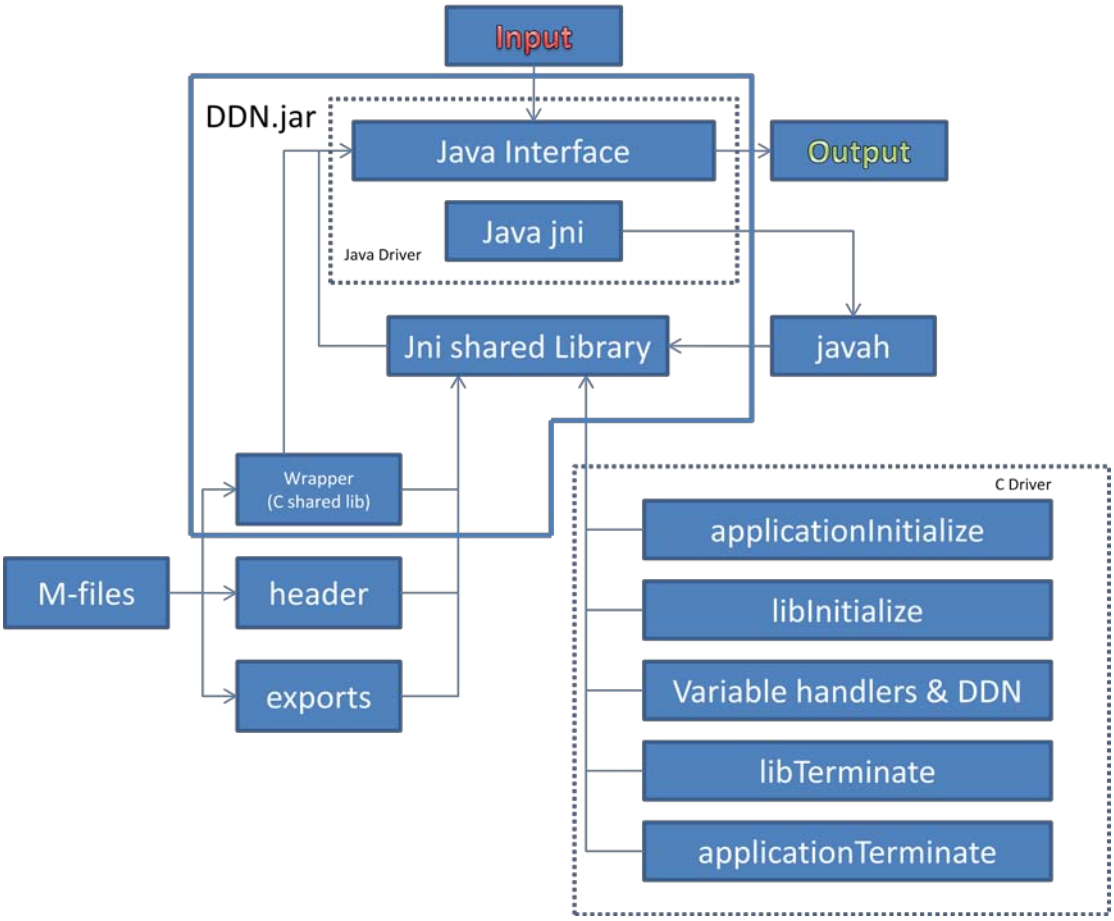


Figure 5 Standalone DDN jar package work flow

Before the integration, there are some editing to do about the Matlab source code of DDN, there are totally nine .m files for DDN algorithem. DDN.m contains the main function ddn(data1, data2, max_K, geneNames, threshold, p_value_cutoff, expTitle), the other 8 are the subfunction for this algorithm. In ddn(), Data1 represent the microarray data under condition1, usually it is a M*N matrix, M denotes the number of gene names, N denotes for the sample size. Data2 is the microarray data under condition2, which is exactly the same size as data1. Max_k is the maximum size of set Zi, an integer between 1 and M-1. GeneNames is the list of M gene names from both datasets. Threshold is the threshold of the fit of the local dependency model. P_value_cutoff is the p value cutoff for selecting the statistically significant local structures. Finally expTitle is the experiment title set by user. There is no diagram drawing involve for the whole process. It can be seen from main function, there are four variable types, two double matrix, three double value, one string matrix and one string. Worth a notice here, string matrix is considered as cell array in Matlab which will need an additional javastringarray to mxArray variable type handler in the C driver. This complicate the variable mapping process a little more. One of the purpose of this thesis is to simplify the

programming language translation work. Fortunately there is a work around, for developing purpose and keeping the original Matlab source code unchange, we create a m file called ddnwrapper to handle all the data input. Since the three input files are all text data files which can be read into Matlab by simply using csvread or xlsread funtion inside Matlab. Howerver .xls files are only limited within Windows operating system, we choose csvread funtion for the input data read for Matlab wrapper file.

Here is the wrapper file code for DDN:

```
function ddnwrapper(a,b,c,max_k,threshold,p_value_cutoff,expTitle)

fid=fopen(c,'r');
InputText=textscan(fid,'%s','delimiter',',');
geneList=InputText[8];
dataset1=csvread(a,1,1);
dataset2=csvread(b,1,1);
ddn(dataset1,dataset2,max_k,geneList,threshold,p_value_cutoff,expTitle);
fclose(fid);
```

3.3 C shared library created by Matlab compiler

There are 9 .m files for DDN algorithm, they are cod.m, cv_lars.m, dataPermutation.m, ddn.m, ddnwrapper.m, dependencyModel.m, extractDDN.m, extractDDN_simple.m, lars_larsso.m and learnLocalStructure.m, plus one wrapper file, ddnwrapper.m. Totally 10 .m files are necessary for integration work.

Use the command:

```
mcc -vB csharedlib:libddnwrapper ddnwrapper.m ddn.m cod.m cv_lars.m
dataPermutation.m dependencyModel.m extractDDN.m extractDDN_simple.m
lars_lasso.m learnLocalStructure.m
```

where libddnwrapper is the library name we choose. The first .m file will be considered as the one which controls the input and output of the algorithm by MCC.

Figure 6 9 files generated by this command

3.4 Java Driver

As mentioned above, this driver can be considered as two parts. One is to define these native methods, the other takes care of the input, output and interface. The name of DDN Java Driver is ddnwrapperDriver, which will be shown as a class in Java programming language.

There are six native methods need to declare in Java driver.

public static native void applicationInitialize();
public static native void libInitialize();
public native void ddnwrapper(String a, String b, String c, double max_k, double threshold, double p_value_cutoff, String expTitle);
public static native void libTerminate();
public static native void applicationTerminate();
public ddnwrapperDriver()

Native methods corresponding to these functions that we need to call in the compiler-generated shared library.

Below is the copyFromJar function. As its name, this method manages to copy the file inside a .jar file out.

```
public static void copyFromJar(String fileUrl, String dest)
        throws IOException {
    int BUFF_SIZE = 100000;
    byte[] buffer = new byte[BUFF_SIZE];
```

```
        InputStream in = null;
        OutputStream out = null;
        URL url = new URL(fileUrl);
        try {
            in = url.openStream();
            out = new FileOutputStream(dest);
            while (true) {
                synchronized (buffer) {
                    int amountRead = in.read(buffer);
                    if (amountRead == -1) {
                        break;
                    }
                    out.write(buffer, 0, amountRead);
                }
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
```

The purpose we have this method here is for deployment. Because finally there will be an executable jar file generated for deployment, inside contains all .class files and library files. Classloader cannot locate a shared library inside any compressed file. When running, the two shared library will be extract to the current folder, details will be shown in chapter 4.

Unzip and classloder part in the main function for Java Driver:

```
/* unzip */
        String curDir = System.getProperty("user.dir");
        String a = "jar:file:" + curDir + "/ddn.jar!/jniddnwrapperDriver.so";
        String b = curDir + "/jniddnwrapperDriver.so";
        String c = "jar:file:" + curDir + "/ddn.jar!/libddnwrapper.so";
        String d = curDir + "/libddnwrapper.so";
        try {
```

```
                    ddnwrapperDriver.copyFromJar(a, b);
                    ddnwrapperDriver.copyFromJar(c, d);
                } catch (IOException ex) {
Logger.getLogger(ddnwrapperDriver.class.getName()).log(Level.SEVERE, null, ex);
            }
/* set path */
                    File lib = new File(new File(curDir), "jniddnwrapperDriver.so");
                    System.load(lib.getAbsolutePath());
```

Notice: *User.dir* catches the current path. The shared library files in the code have extension .so, which is for Linux deployment. For windows, it is .dll. We choose to use the absolute path for classloader which benefit to individual deployment. It is not user friendly to put this step in the manual and require every user to copy all shared libraries into a certain path.

The command line parameter input, output code for main:

```
/* parameters conversion & check */
        try {
            String dataset1 = args[0];
            String dataset2 = args[1];
            String geneList = args[2];
            double max_k = Double.parseDouble(args[3]);
            double threshold = Double.parseDouble(args[4]);
            double p_value_cutoff = Double.parseDouble(args[5]);
            String expTitle = args[6];

            /* Call the application initialization routine. */
            applicationInitialize();
            /* Call the library initialization routine. */
            libInitialize();
            /* Call the library functions */
            ddnwrapperDriver md = new ddnwrapperDriver();
            md.ddnwrapper(dataset1, dataset2, geneList, max_k, threshold,
p_value_cutoff, expTitle);
            System.out.println("done!");
```

```
        } catch (ArrayIndexOutOfBoundsException ex) {
            System.out.println("Usage: java -jar ddn.jar <dataset1> <dataset2>
<geneList> <max_k> <threshold> <p_value_cutoff> <output_file_prefix>");
            System.out.println("Example: java -jar ddn.jar dataset1.csv dataset2.csv
geneList.csv 1 0.25 0.05 TEST");
            System.exit(0);
        }
            /* Call the library termination routine. */
            libTerminate();
            /* Call the application termination routine. */
            applicationTerminate();
```

The most important step in this part begins at *applicationInitialize();* from which the Java virtual machine start to work and calling native functions. The instance of *ddnwrapperDriver* class *md.ddnwrapper* must be called after *applicationInitialize() and libInitialize(),* before *libTerminate() and applicationTerminate().* Also, those native functions cannot be called before classloader finished its job. Once *libTerminate()* has been called, any further *Initialize()* call will no longer be allowed within this specific running. The *System.out.println("done!")* is to make sure the main native method runs successfully. Since all input parameter reading jobs have been passed to Matlab, and there is no output parameter either. The main function without any array reading or printing process looks much clearer than usual.

If the user chooses to have GUI, which we believe 90% of people will do, there are some management to do in the code because of the calling order constrain, such as when to trigger which native method, when to show the GUI. Consider a user friendly feature like this (fig.7):

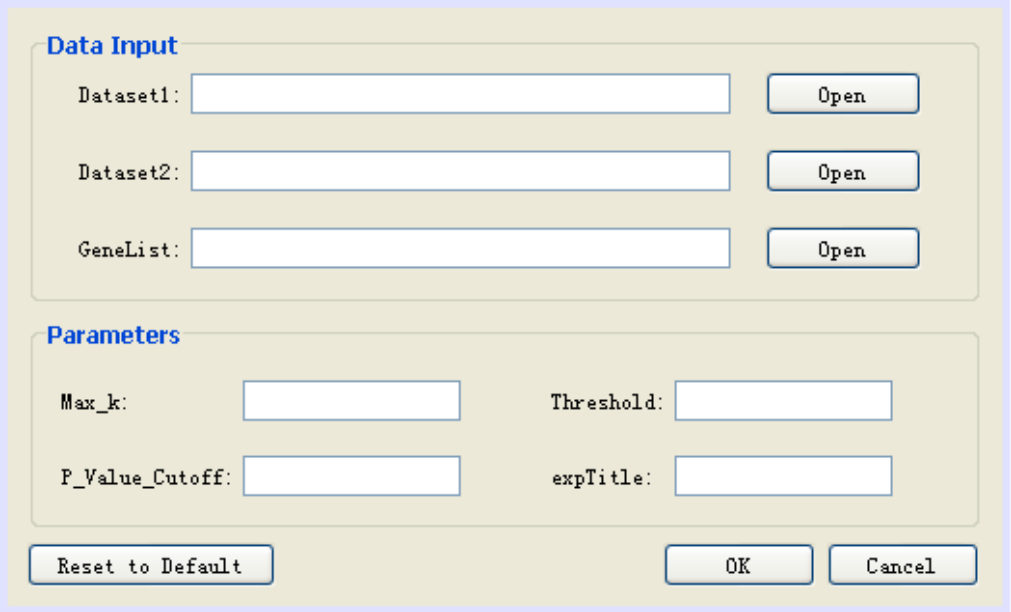


Figure 7 GUI of DDN standalone package

From the appearance we can see that it provides all input parameters space same as the command line way, “OK” button is an alternative way to type “Enter”. First we try to put all the main function code in *OKActionPerformed* method, all seems right for the first time run, but second time press the “OK” button will cause the error “mclInitializeApplication may not be called once mclTerminateApplication has been called”. (Note: mcl+* is the function in C code form which will be discussed in 3.4) Just as described before, the main thread ends at press cancel or X in the top right corner. Within the same thread, the second time call *applicationInitialize()* is consider as a call after *applicationTerminate()*. The problem is solved by separate main function code into several parts and put into different button action performed method as follow:

```
private void OKActionPerformed(java.awt.event.ActionEvent evt) {
    setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));

    String a = Dataset1Location.getText();
    String b = Dataset2Location.getText();
    String c = GenelistLocation.getText();
    double max_k = Double.parseDouble(Maxk.getText());
    double threshold = Double.parseDouble(Threshold.getText());
    double p_value_cutoff = Double.parseDouble(Pvaluecutoff.getText());
    String expTitle = Exptitle.getText();

    ddnwrapperDriver md = new ddnwrapperDriver();
    md.ddnwrapper(a, b, c, max_k, threshold, p_value_cutoff, expTitle);
    setCursor(null);
}
```

OKActionPerformed method passes the parameter to Java virtual machine and call *ddnwrapper* function, at the same time set the cursor to waiting form suggest that the thread is running.

```
/* Creates new form DDNUI */
public DDNUI() {
    initComponents();
    Dataset1Location.setEditable(false);
    Dataset2Location.setEditable(false);
    GenelistLocation.setEditable(false);
    String curDir = System.getProperty("user.dir");
```

```
        File lib = new File(new File(curDir), "jniddnwrapperDriver.dll");
        String[] args = new String[4];
        args[0] = "jar:file:/" + curDir + "\\DDN
UserInterface.jar!/jniddnwrapperDriver.dll";
        args[1] = curDir + "\\jniddnwrapperDriver.dll";
        args[2] = "jar:file:/" + curDir + "\\DDN UserInterface.jar!/libddnwrapper.dll";
        args[3] = curDir + "\\libddnwrapper.dll";
        try {
            ddnwrapperDriver.copyFormJar(args[0], args[1]);
            ddnwrapperDriver.copyFormJar(args[2], args[3]);
        } catch (IOException ex) {
            Logger.getLogger(DDNUI.class.getName()).log(Level.SEVERE, null, ex);
        }
        System.load(lib.getAbsolutePath());
        ddnwrapperDriver.applicationInitialize();
        ddnwrapperDriver.libInitialize();
    }
```

Put all path setting, application and library initializing code in the GUI initialize part. It means that once user choose to have GUI, the main thread will be passed the “Set” and ” Ready” phase waiting for the “GO” action. This work around the calling order limitation and provide a user friendly GUI as usual. Different running situation will be shown in Chapter 4.

In main function, Show GUI when called:

```
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new DDNUI().setVisible(true);
            }
        });
```

Terminate the library and application when exit GUI:

```
    private void CancelActionPerformed(java.awt.event.ActionEvent evt) {
        ddnwrapperDriver.libTerminate();
        ddnwrapperDriver.applicationTerminate();
        System.exit(0);
}
```

3.5 C Driver

This is the most important part of the DDN integration, it is considered as a bridge between ddnwrapperDriver.java and DDN Matlab source code.

Native code accesses Java virtual machine by JNIEnv interface pointer [8]. The first argument received by the function that implement a native method is JNIEnv interface pointer. Once the native method is being called, virtual machine will be guaranteed to pass the exact interface pointer to the native method.

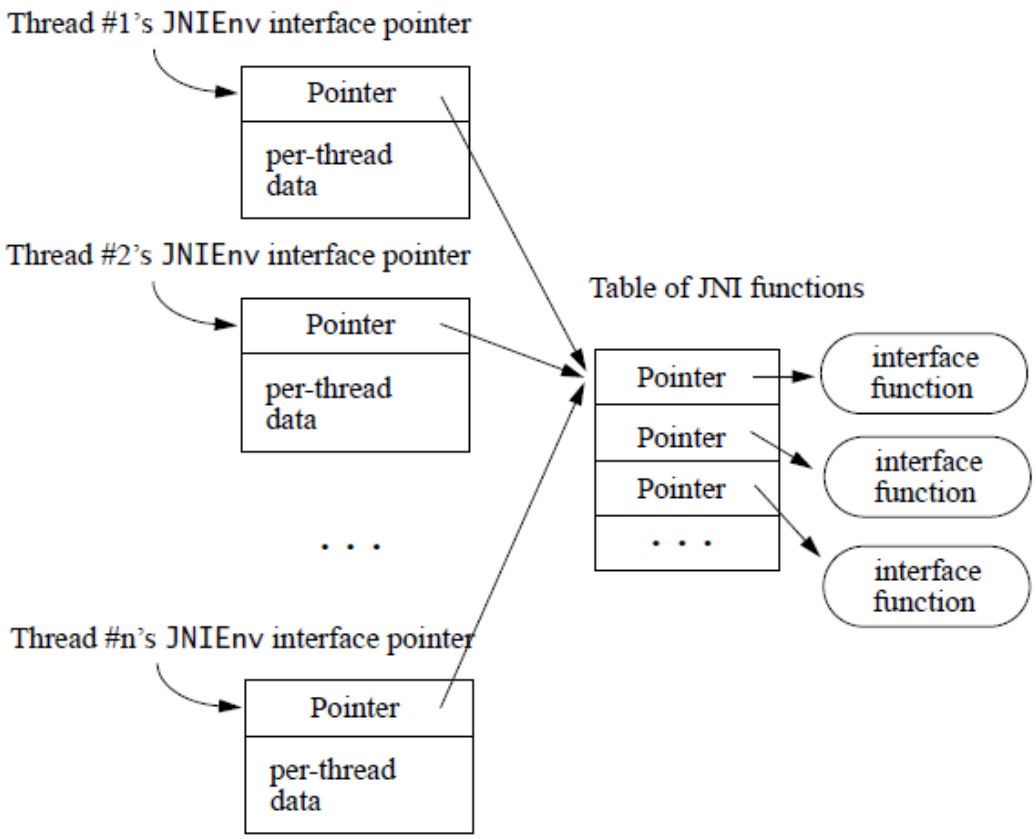


Figure 8 JNIEnv functional (ref. JNI guide)

Three parts consist of the C Driver: Java exception handler, variable mapping handler and JNI function handler.

There are two header files that are generated by Matlab Compiler and javah, which we treated as guide files need to be included at the beginning of the first part. Then, create a function handles the exception in JNI, take specific exception name and message as input, throw them out in Java exception. This is a vital preparation step for the further development, as mentioned in Chapter 2.2, calling a JNI function with any kind of exception in pending will lead to an undefined results.

#include "ddnwrapperDriver.h"

```
#include "libddnwrapper.h"
static void throw_java_exception(JNIEnv *env, const char* name, const char* msg)
{
	jclass cls = (*env)->FindClass(env, name);
	if (cls != NULL)
	{
		(*env)->ThrowNew(env, cls, msg);
		(*env)->DeleteLocalRef(env, cls);
	}
}
```

Second part, since ddnwrapper.m is the first .m file entered in mcc function, it takes charge of the input and output of the main algorithm. There are seven variables for the input and none output variables for ddnwrapper.m. The seven variables are categorized into three types: four strings, three doubles, which equals four jstring and three jdouble in JNI. Thus two variable mapping functions are needed here, they are javadouble to mxArray and javastring to mxArray.

```
	/* Helper function to marshal a Java double into an mxArray.*/
	static mxArray* javadouble_to_mxArray(JNIEnv* env, jdouble arr)
	{
		mxArray *px;
		double arr2 = arr;	/* get data from java variable */
		px = mxCreateDoubleScalar(arr2); /* copy data to mxArray */
		return px;
	}

/* Helper function to marshal a Java string into an mxArray. */
static mxArray* javastring_to_mxArray(JNIEnv* env, jstring arr)
{
	mxArray *px;
	const char* arr2 = (*env) -> GetStringUTFChars(env,arr, 0); /* get data from java
variable */
	px = mxCreateString(arr2); /* copy data to mxArray */
	return px;
}
```

As you can see, the methods begin with mx are the API MX Matrix library provided by Matlab to facilitate communication with C programming language. mxCreateDoubleScalar is to create scalar, double-precision array initialized to specified value. mxCreateString is to

create 1-by-N string mxArray initialized to specified string. Variable type double can be directly passed from Java to C, because the mx scalar creator we use. Variable type string use java method GetStringUTFChar to pass the physical location character by character into pointer. If developer likes to have Java to read the input array, then the following javamatrix to mxArray mapping handler function is needed:

```
static mxArray* javamatrix_to_mxArray(JNIEnv* env, jobjectArray arr)
{
    mxArray* px;            /* mxArray to return */
    int m;                  /* Number of rows */
    int n;                  /* Number of columns */
    int i,j;                /* Loop variables */
    jdoubleArray row;       /* Stores current row */

    if (arr == NULL)
        return NULL;
    /* Get number of rows and columns */
    m = (int)(*env)->GetArrayLength(env, arr);
    if ((row = (jdoubleArray)(*env)->GetObjectArrayElement(env, arr, 0)) == NULL)
        return NULL;
    n = (int)(*env)->GetArrayLength(env, row);
    (*env)->DeleteLocalRef(env, row);
    /* Create mxArray */
    px = mxCreateDoubleMatrix(m, n, mxREAL);
    /* Loop over rows */
    for (i=0; i<m; i++)
    {
        int subs[2];            /* Stores subscripts for mxArray indexing */
        int offset;             /* Offset in mxArray of value */
        double* pval;           /* Current value to copy */
        subs[0] = i;
        row = (jdoubleArray)(*env)->GetObjectArrayElement(env, arr, (jsize)i);
        /* Loop over columns */
        for (j=0; j<n; j++)
        {
            subs[1] = j;
            offset = mxCalcSingleSubscript(px, 2, subs);
            pval = mxGetPr(px) + offset;
```

```
            (*env)->GetDoubleArrayRegion(env, row, (jsize)j, 1, pval);
        }
        (*env)->DeleteLocalRef(env, row);
    }
    return px;
}
```

Apparently, it is much more complicated then the single value mapping. mxCreateDoubleMatrix is to create 2-D, double-precision, floating-point mxArray initialized to 0. mxCalcSingleSubscript is used to calculate the offset from first element to desired element. mxGetPr is for real data elements in mxArray of type double. This is another reason I am highly preferred not to have a return value in the original Matlab code. Because it not only doubles the C drivers mapping handler functions but also delays the output through backward variable mapping. So, we create a Matlab wrapper file to handle all the output as well as input. Luckily, DDN algorithm has already got the output write into three separated text files. Therefore, only input is taken care of in ddnwrapper.m. This also makes the implementation of the native core function simpler.

From these variable mapping methods, we can see that, primitive data type is copied between Java virtual machine and native code. On the other hand, Object data type is copied by reference. The reason of using reference instead of pointer is because the garbage collector. After the object is moved, Java virtual machine can follow the register of the reference to acquire the object. This is surely an advantage to do so. However, DeleveLocalRef method is necessary to be dressed.

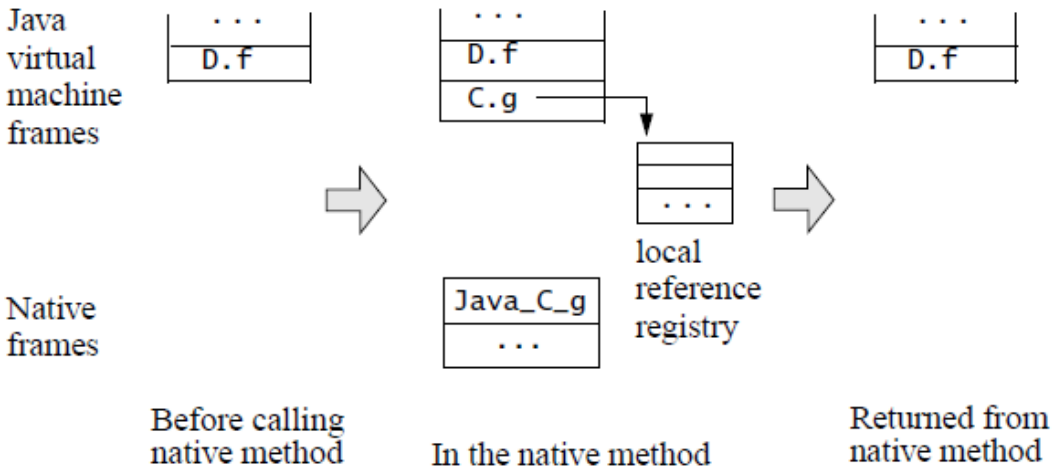


Figure 9 Local reference registry (ref. JNI guide)

As shown in fig.9, before calling the next native method, current local reference must be deleted, allow it to be garbage collected. There is a register mapping all the local references

to the object pointers, objects which pass the native method will automatically get one. When the function has a return value, current local reference is deleted from the registry table.

The third part of C Driver is implementation of native methods. The following code is a pair of application and library initialization handler from ddnwrapperDriver Java class. mclInitializeApplication() and libddnwrapperInitialize() are the function names provided by Matlab Compiler, at the same time, the actual initialization code is also generated in libddnwrapper.c. mclInitializeApplication() setup the global MCR state and enables the construction of MCR instance. libddnwrapperInitialize() create the MCR instance required by the library. These two methods call the actual method and provide self made exception message for development purpose, it give me the warning when application or library malfunction.

```
JNIEXPORT
void JNICALL Java_ddnwrapperDriver_applicationInitialize(JNIEnv *env, jclass cls)
{
    (void)env;
    (void)cls;
    if (!mclInitializeApplication(NULL, 0))
    {
        throw_java_exception(env,
            "java/lang/RuntimeException",
            "Could not initialize the application.");
    }
}
JNIEXPORT
void JNICALL Java_ddnwrapperDriver_libInitialize(JNIEnv *env, jclass cls)
{
    (void)env;
    (void)cls;
    if (!libddnwrapperInitialize())
    {
        throw_java_exception(env,
            "java/lang/RuntimeException",
            "Could not initialize the library.");
    }
}
```

The other pair of application and library termination handler has basically the same purpose as the above two. The only difference is function name. They are

libddnwrapperTerminate() and mclTerminateApplication(), which destroy the associated MCR and free resources associated with the global MCR state.

Between the two pairs of environment setup method, there is the core function of DDN algorithm.

```
JNIEXPORT
void JNICALL Java_ddnwrapperDriver_ddnwrapper(JNIEnv *env, jobject obj, jstring a,
jstring b, jstring c, jdouble d, jdouble e, jdouble f, jstring g)
{
    mxArray* pa = NULL;
    mxArray* pb = NULL;
    mxArray* pc = NULL;
    mxArray* pd = NULL;
    mxArray* pe = NULL;
    mxArray* pf = NULL;
    mxArray* pg = NULL;
    mxArray* ph = NULL;
    /* Process input arrays */
    if ((pa = javastring_to_mxArray(env, a)) == NULL)
        goto EXIT;
    if ((pb = javastring_to_mxArray(env, b)) == NULL)
        goto EXIT;
    if ((pc = javastring_to_mxArray(env, c)) == NULL)
        goto EXIT;
    if ((pd = javadouble_to_mxArray(env, d)) == NULL)
        goto EXIT;
    if ((pe = javadouble_to_mxArray(env, e)) == NULL)
        goto EXIT;
    if ((pf = javadouble_to_mxArray(env, f)) == NULL)
        goto EXIT;
    if ((pg = javastring_to_mxArray(env, g)) == NULL)
        goto EXIT;
    /* Call ddnwrapper */
    mlfDdnwrapper(1, &ph, pa, pb, pc, pd, pe, pf, pg);
    /* Process output array */
    EXIT:
    if (pa != NULL)
        mxDestroyArray(pa);
```

```
    if (pb != NULL)
        mxDestroyArray(pb);
    if (pc != NULL)
        mxDestroyArray(pc);
    if (pd != NULL)
        mxDestroyArray(pd);
    if (pe != NULL)
        mxDestroyArray(pe);
    if (pf != NULL)
        mxDestroyArray(pf);
    if (pg != NULL)
        mxDestroyArray(pg);
    if (ph != NULL)
        mxDestroyArray(ph);
    return 0;
}
```

After declare and process the input variables, main function is invoked, mlf prefix is coming from the header file libddnwrapper.h generated by Matlab Compiler. The benefit for not having an output variable can also be seen here. No need to deal with variable backward mapping any more. At last, clean up all mxArray variables and close files to free allocated memories.

3.6 Compiling the library using C Driver

Although Java is a once compiled, cross platform usable programming language, Matlab Compiler is not. For Windows system, mbuild command is as follow:

```
mbuild -I"C:\Program Files\Java\jdk1.6.0_10\include" -I"C:\Program Files\Java\jdk1.6.0_10\include\win32" jniddnwrapperDriver.c jniddnwrapperDriver.exports libddnwrapper.lib -link shared -output jniddnwrapperDriver
```

The name for output c shared library is jniddnwrapperDriver.dll

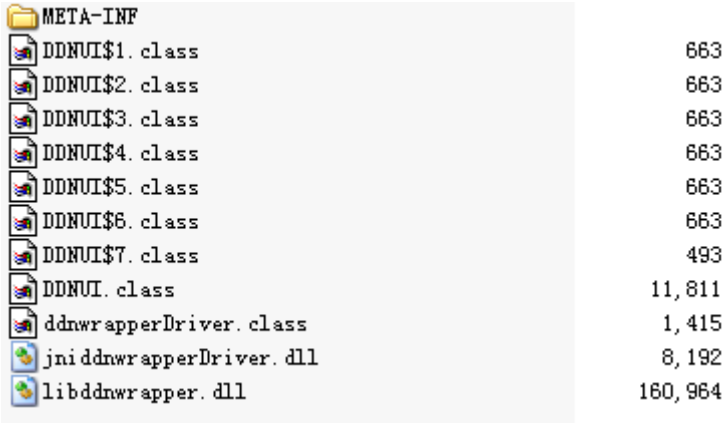
For Linux system, mbuild command is as follow:

```
mbuild -I<jdk-dir>/include -I<jdk-dir>/include/<arch> jniddnwrapperDriver.c jniddnwrapperDriver.exports libddnwrapper.so -L. -lddnwrapper -link shared -output jniddnwrapperDriver
```

<jdk-dir> is Java JDK path

<arch> is the architecture of the Linux system usually (i386 or amd64)

jniddnwrapperDriver.exports is a binary file which suppose to contain the output variable name. In DDN case, it is a blank file. The name for output c shared library is jniddnwrapperDriver.so



META-INF	
DDNUI$1.class	663
DDNUI$2.class	663
DDNUI$3.class	663
DDNUI$4.class	663
DDNUI$5.class	663
DDNUI$6.class	663
DDNUI$7.class	493
DDNUI.class	11,811
ddnwrapperDriver.class	1,415
jniddnwrapperDriver.dll	8,192
libddnwrapper.dll	160,964

Figure 10 Content of DDN jar package

Figure 10 shows the content of final standalone DDN jar package for Window operation system. You can see two library files and eight .class files. First seven .class files are for Java GUI, ddnwrapperDriver.class is the Java Driver..dll extension libraries are for Windows system only. If compiled in Linux system, they will have .so extension. One thing need to mention though, we use Matlab 2009b compile the C shared library. In early version, there is an additional file call *.ctf generated along with libddnwrapper.dll. If any developer uses the early version, they should put this file inside the standalone package too, and use copyfromJar method in the Java Driver to extract it from the jar package before running any initialization functions. In Matlab 2009b this file is embedded into library file libddnwrapper. Use –C option in mbuild function if you still want to have a separate CTF file.

Chapter 4 DDN Deployments

Will it disobey my intention to make the package platform independent as described in JNI section? Unfortunately the answer is yes. However, there are three major platforms that MCR can work on: windows, Linux and MacOS. Take 32/64 bits structure into consideration. Compile six different sets of libraries will be enough for the cross platform distribution. In fact, 64 bits operating system also support 32bit libraries, thus three sets of library will meet the minimum requirements. If there is a operating system that Matlab do not support, then there is no way to run this tool on that machine, because the fundamental of this process is Matlab Runtime environment.

4.1 Windows deployment

(1). Prerequisites for Deployment

* Verify the Matlab Compiler Runtime (MCR) is installed and ensure you have installed version 7.11.

* If the MCR is not installed, run MCRInstaller, located in:

\Matlab\R2009b\toolbox\compiler\deploy\win32\MCRInstaller.exe or download the correct version MCRInstaller.exe from Mathworks.

On the target machine, add the MCR directory to the system path, specified by the target system's environment variable.

i. Locate the name of the environment variable to set, using the table below:

Operating System	Environment Variable
Windows	PATH

ii. Set the path by doing one of the following:

NOTE: <mcr_root> is the directory where MCR is installed on the target machine.

* Add the MCR directory to the environment variable by opening a command prompt and issuing the DOS command:

set PATH=<mcr_root>\v711\runtime\win32;%PATH%

Alternately, for Windows, add the following path name:

<mcr_root>\v711\runtime\win32

to the PATH environment variable, by doing the following:

1. Select the My Computer icon on your desktop.
2. Right-click the icon and select Properties from the menu.
3. Select the Advanced tab.

4. Click Environment Variables.

NOTE: On Windows, the environment variable syntax utilizes backslashes (\), delimited by semi-colons (;).

* JRE_1.6.0+ is needed, unless you want to use the JRE inside Matlab compiler Runtime, the path is Matlab Compiler Runtime\v711\sys\java\jre\win32\jre\bin.

The is the most important part in deployment. UnsatisfiedLinkError is a common exception throw by Java if system path does not set correctly.

(2). Description

* max_K: the maximum size of set i Z , an integer between 1 and M-1
* threshold: threshold of the fit of the local dependency model
*p_value_cutoff: p value cutoff for selecting the statistically significant local structures
* expTitle: the experiment title, a string.
* Input data format is .CSV. (eg.Dataset1.csv Dataset2.csv geneList.csv)
* The outputs of this program are three files saved in the current directory.

One output file is a ".sif" file, which specifies the network topology of the differential dependency network and can be imported in Cytoscape. The second file is a ".NA" file, which is the node attribute file for visualization in Cytoscape. The third file is a ".txt" file, which includes the "hot spots" identified by the DDN analysis, together with their fold-changes and p-values.

Here is a demo:

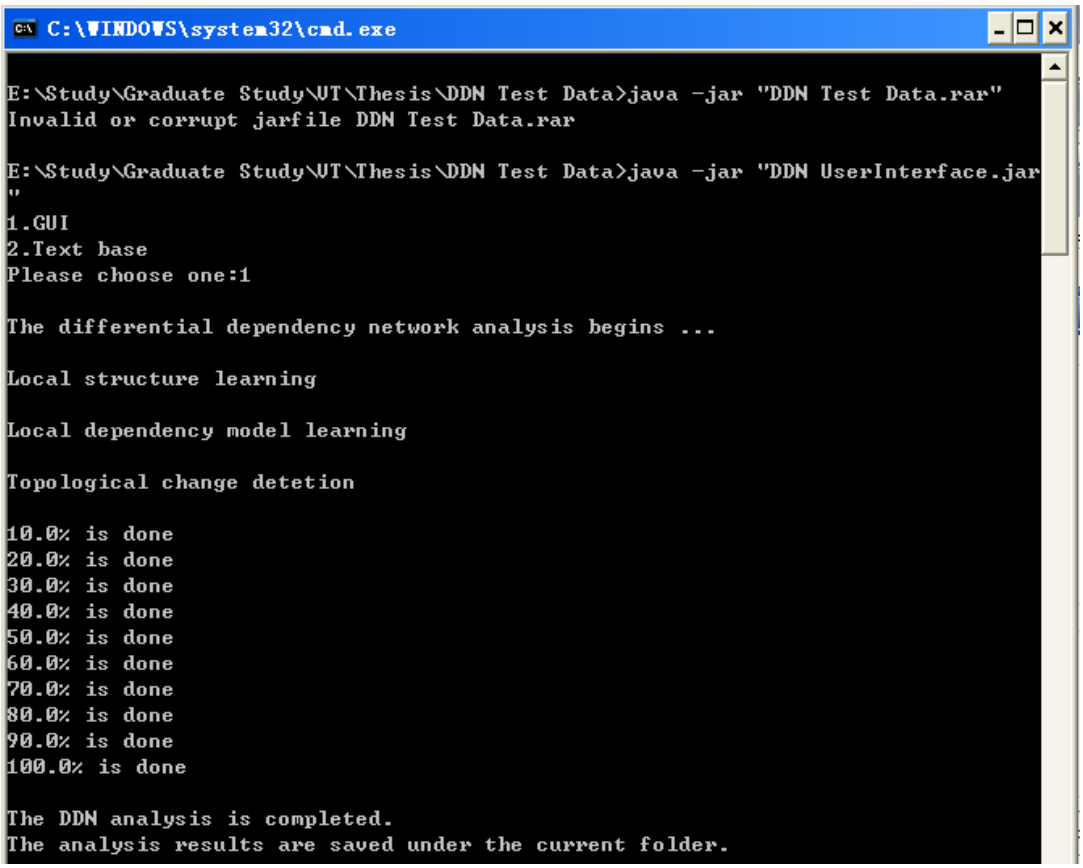


Figure 11 DDN standalone jar package running example

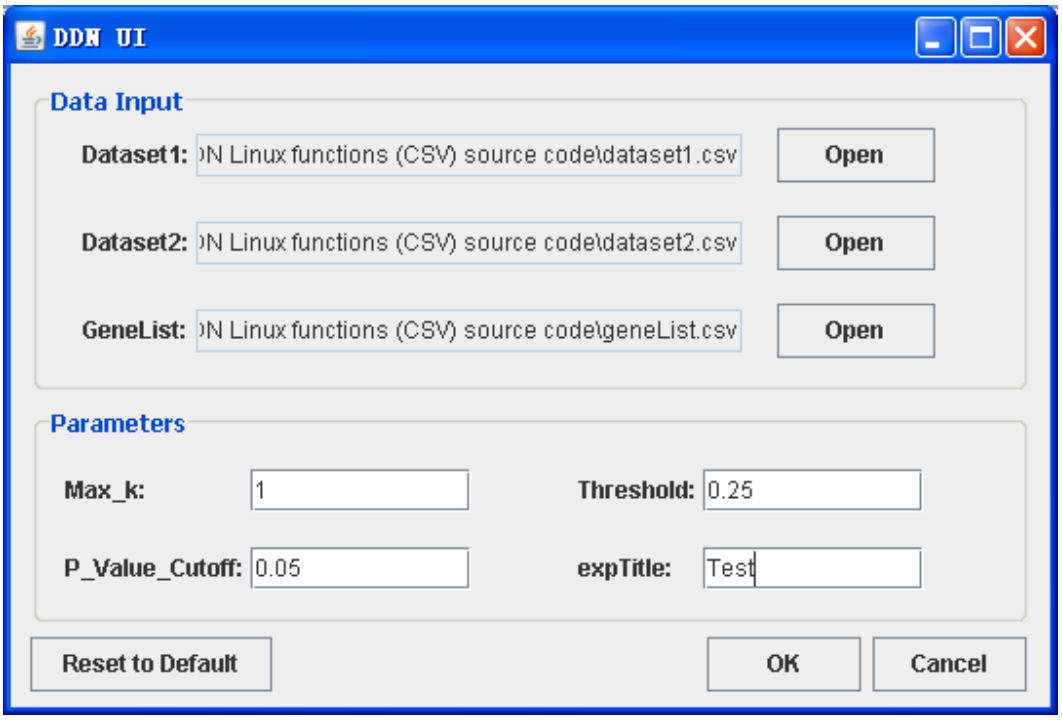


Figure 12 DDN GUI demo

User choose 1 for GUI shown in fig.11, then DDN UI pop up (fig.12), after providing the necessary datasets and parameter, press “OK” to execute the algorithm. As I mentioned in Chapter 3, any initialization function cannot be called after calling termination function in Java Driver. If developer didn’t notice this problem, a second time execution will end up error, as shown in fig.13. The correct situation is shown in fig.14.

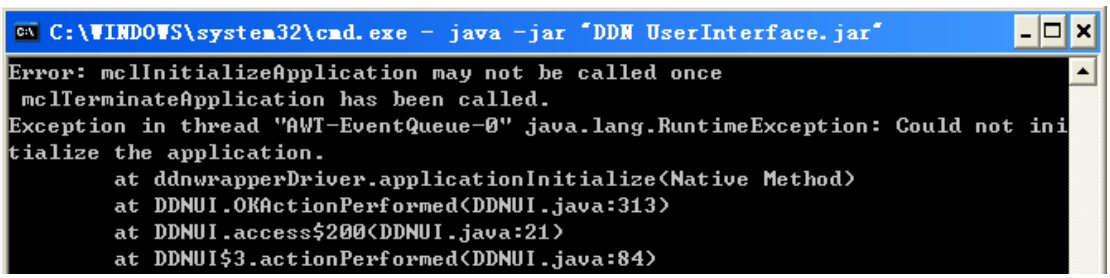


Figure 13 Error message because of violation

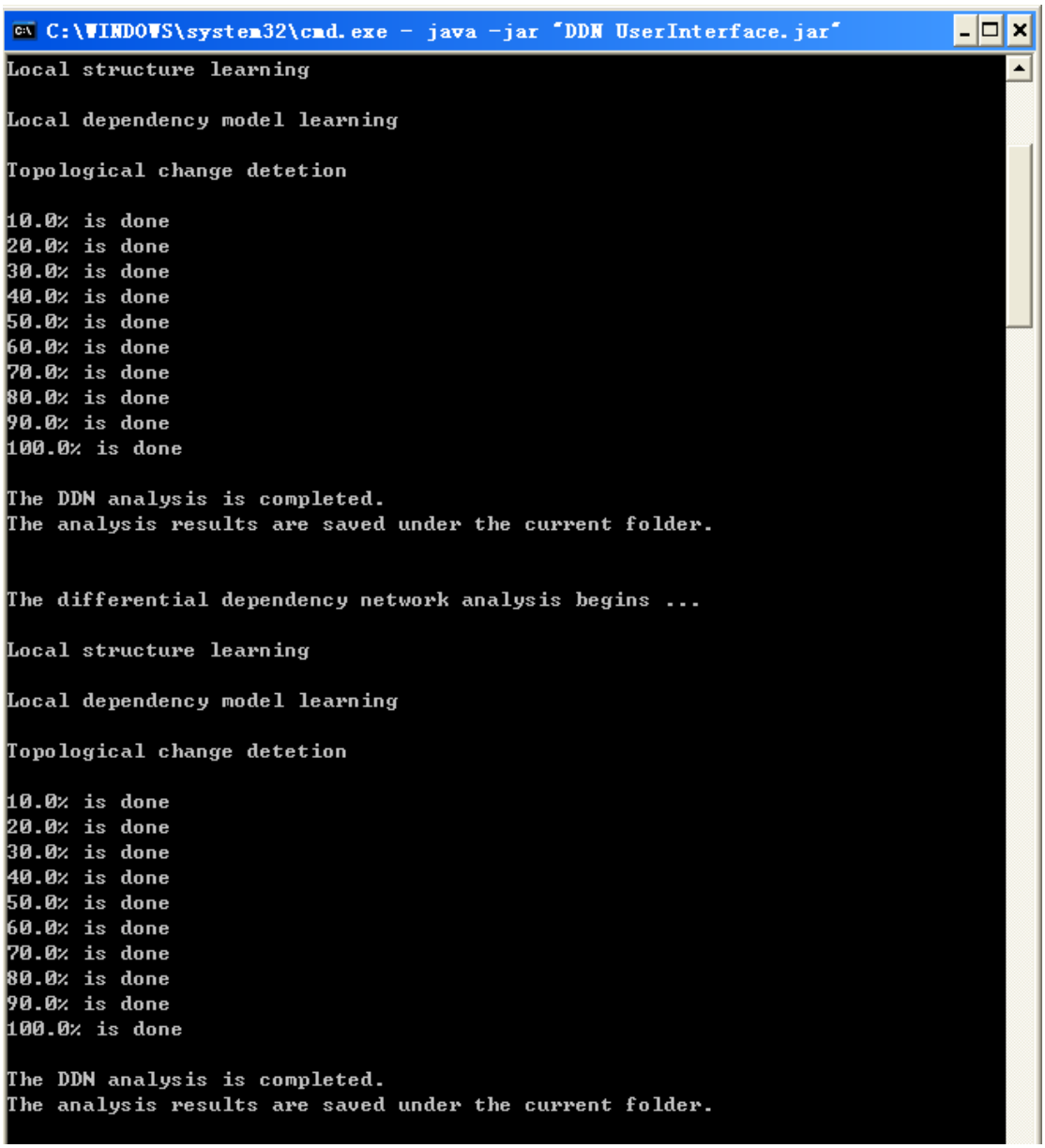


Figure 14 Successful multiple running situations

4.2 Linux deployment

(1). Prerequisites for Deployment

* Verify the Matlab Compiler Runtime (MCR) is installed and ensure you have installed version 7.11

* If the MCR is not installed, run MCRInstaller, located in:

/Matlab/toolbox/compiler/deploy/i386/MCRInstaller.bin

(2). Files to Deploy and Package

Files to package for Shared Libraries

==========

-libddnwrapper.so

-MCRInstaller.bin

- include when building component by selecting "include MCR" option

(3) On the target machine, add the MCR directory to the system path specified by the target system's environment variable. Different from window system, Linux system do not provide a GUI control panel for setting PATH. Even if user set the LD_LIBRARY_PATH, it only works once. Instead enter all the commands before every single run, we provide a bash file contains all presetting materials. If preferred, user can also run the DDN package inside the bash file. All six LD_LIBRARY_PATH setting is a must for development.

<mcr_root>/v711/runtime/glnx86:

<mcr_root>/v711/sys/os/glnx86:

<mcr_root>/v711/sys/java/jre/glnx86/jre/lib/i386/native_threads:

<mcr_root>/v711/sys/java/jre/glnx86/jre/lib/i386/server:

<mcr_root>/v711/sys/java/jre/glnx86/jre/lib/i386:

'pwd':

XAPPLRESDIR = $<mcr_root>/v711/X11/app-defaults

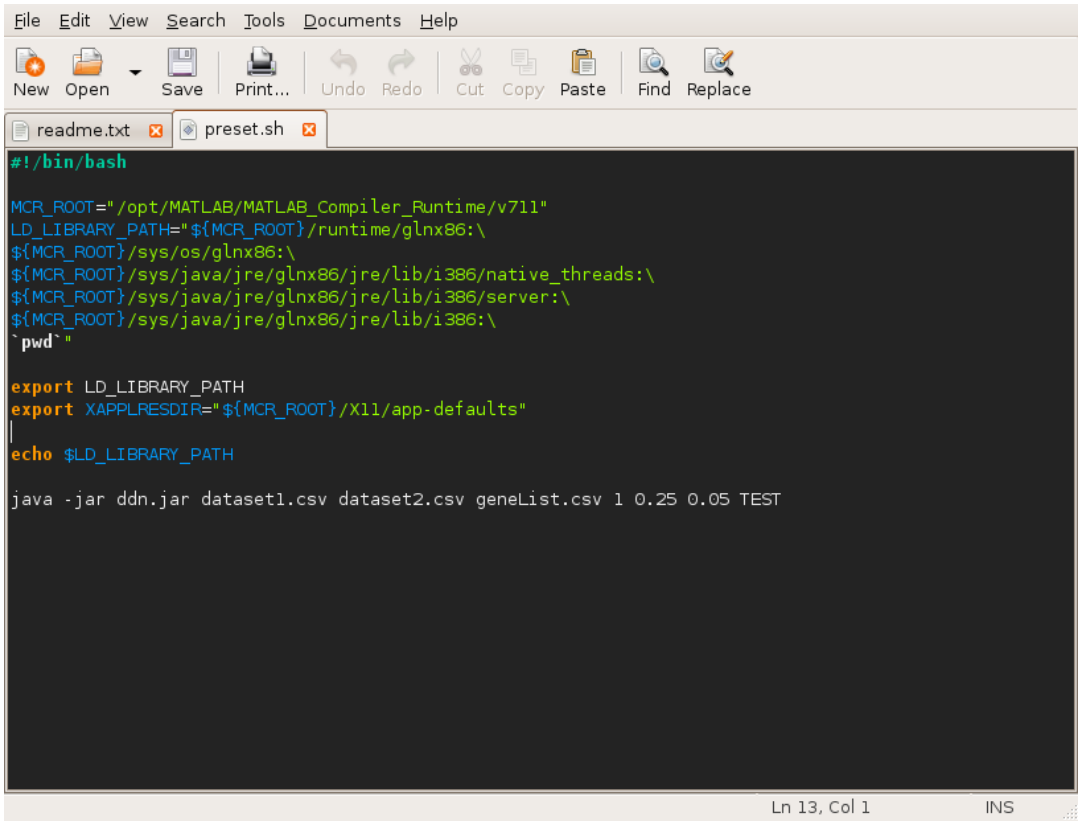


Figure 15 Bash file for Unix deployment

If user would like to make these changes persistent after logout on UNIX or Mac machines, they need to modify the .cshrc file to include these commands. On UNIX or Mac, the environment variable syntax utilizes forward slashes (/), delimited by colons (:). The system path may vary between different Linux platforms. The above instruction is for a 32-bit Ubuntu. For 64-bit Linux systems, simply change all seven presetting to

<mcr_root>/v711/runtime/glnxa64:

<mcr_root>/v711/sys/os/glnxa64:

<mcr_root>/v711/sys/java/jre/glnxa64/jre/lib/amd64/native_threads:

<mcr_root>/v711/sys/java/jre/glnxa64/jre/lib/amd64/server:

<mcr_root>/v711/sys/java/jre/glnxa64/jre/lib/amd64

XAPPLRESDIR=$ <mcr_root>/v711/X11/app-defaults

The following is a demo:

Successful running (fig.16)

```
File  Edit  View  Terminal  Tabs  Help
jinl@jinl-laptop:~/local/DDN/DDN Linux functions (CSV) source code 32bit$ ./preset.sh
/opt/MATLAB/MATLAB_Compiler_Runtime/v711/runtime/glnx86:/opt/MATLAB/MATLAB_Compiler_Ru
ntime/v711/sys/os/glnx86:/opt/MATLAB/MATLAB_Compiler_Runtime/v711/sys/java/jre/glnx86/
jre/lib/i386/native_threads:/opt/MATLAB/MATLAB_Compiler_Runtime/v711/sys/java/jre/glnx
86/jre/lib/i386/server:/opt/MATLAB/MATLAB_Compiler_Runtime/v711/sys/java/jre/glnx86/jr
e/lib/i386:/home/jinl/local/DDN/DDN Linux functions (CSV) source code 32bit

The differential dependency network analysis begins ...

Local structure learning

Local dependency model learning

Topological change detetion

10.0% is done
20.0% is done
30.0% is done
40.0% is done
50.0% is done
60.0% is done
70.0% is done
80.0% is done
90.0% is done
100.0% is done

The DDN analysis is completed.
The analysis results are saved under the current folder.

done!
jinl@jinl-laptop:~/local/DDN/DDN Linux functions (CSV) source code 32bit$
```

Figure 16 Linux demo I

Usage and example provided when running error (Fig.17. There will be usage and example available for this situation.

```
File  Edit  View  Terminal  Tabs  Help
jinl@jinl-laptop:~/local/DDN/DDN Linux functions (CSV) source code 32bit$ ./preset.sh
/opt/MATLAB/MATLAB_Compiler_Runtime/v711/runtime/glnx86:/opt/MATLAB/MATLAB_Compiler_Ru
ntime/v711/sys/os/glnx86:/opt/MATLAB/MATLAB_Compiler_Runtime/v711/sys/java/jre/glnx86/
jre/lib/i386/native_threads:/opt/MATLAB/MATLAB_Compiler_Runtime/v711/sys/java/jre/glnx
86/jre/lib/i386/server:/opt/MATLAB/MATLAB_Compiler_Runtime/v711/sys/java/jre/glnx86/jr
e/lib/i386:/home/jinl/local/DDN/DDN Linux functions (CSV) source code 32bit
Usage: java -jar ddn.jar <dataset1> <dataset2> <geneList> <max_k> <threshold> <p_value
_cutoff> <output_file_prefix>
Example: java -jar ddn.jar dataset1.csv dataset2.csv geneList.csv 1 0.25 0.05 TEST
jinl@jinl-laptop:~/local/DDN/DDN Linux functions (CSV) source code 32bit$
```

Figure 17 Linux demo II

4.3 Testing Results

Take breast cancer research as an example. There are three files generate every time. Their files name (which define by the user, include experiment title, data and parameters) are:

BC20091028T174032_DDN_pvalue0.01_threshold0.25_K2.sif

BC20091028T174032_node_attribute0.01_threshold0.25_K2.NA

BC20091028T174032_HotSpot_pvalue0.01_threshold0.25_K2.txt

"BC20091028T174032_HotSpot_pvalue0.01_threshold0.25_K2.txt" gives the "hot spots" identified by the DDN algorithm:

Hot spots	*Fold change*	*p-value for t-test*
ABCB11	0.205497	8.48E-01
AKT1	-4.02366	3.92E-01
BCAR3	-0.49242	9.37E-01
BCL2	-2.45542	2.62E-01
BIK	-2.75073	7.52E-01
BIRC1	0.598092	8.67E-01
BIRC3	2.659149	5.60E-01
CAV3	4.122603	7.92E-01
CGA	4.186385	7.00E-01
COX7A2L	3.937694	2.32E-01
EBAG9	2.043713	6.76E-01
ESR2	-0.37236	8.74E-01
HOXA10	4.13861	7.89E-01
MAPK1	-1.34566	6.09E-01
MAPK13	2.813102	5.01E-01

MAPK14	-0.11199	9.35E-01
MAPK3	0.242089	9.67E-01
MAPK4	1.552917	4.84E-01
MAPK8	-6.72935	1.65E-01
NFKB1	0.039079	9.88E-01
NFKB2	-0.09148	6.92E-01
NPM3	1.59144	6.85E-01
XBP1	-1.64618	2.23E-01

Table 1 Hotspot detected by DDN

"BC20091028T174032_DDN_pvalue0.01_threshold0.25_K2.sif" can be viewed in Cytoscape. To download Cytoscape, please visit

http://www.cytoscape.org/download.php?file=cyto2_6_0.

To install Cytoscape, please refer to

http://www.cytoscape.org/manual/Cytoscape2_6Manual.html.

After launching Cytoscape, select **File** -> **Import** -> **Network (multiple file types) …**, and then select the file

"BC20091028T174032_DDN_pvalue0.01_threshold0.25_K2.sif"

and import. Now the differential dependency network shows up. The menu "Layout" implements different layout methods, or simply drags the nodes in the network to obtain a proper layout.

"BC20091028T174032_node_attribute0.01_threshold0.25_K2.NA" is a node attribute file, which contains the fold changes of the gene expressions between two conditions (after base 2 logarithm).

In Cytoscape, select **File** -> **Import** -> **Node Attributes …**, and then select the file

"BC20091028T174032_node_attribute0.01_threshold0.25_K2.NA"

and import.

To better visualize the topological changes under two conditions, the Vizmap Property File "ddn.props" is designed to differentiate the network changes and represent expression changes under two conditions.

Select **File** -> **Import** -> **Vizmap Property File …**, and then select the file "ddn.props" and import. Then in the **Control Panel**, select Tab **VizMapperTM**, and choose "ddn" for **Current Visual Style**, as shown in Figure 18.

In the network, the red lines represent the connections (dependencies) that only exist under condition 1, and the green lines represent the connections (dependencies) that only exist under condition 2.

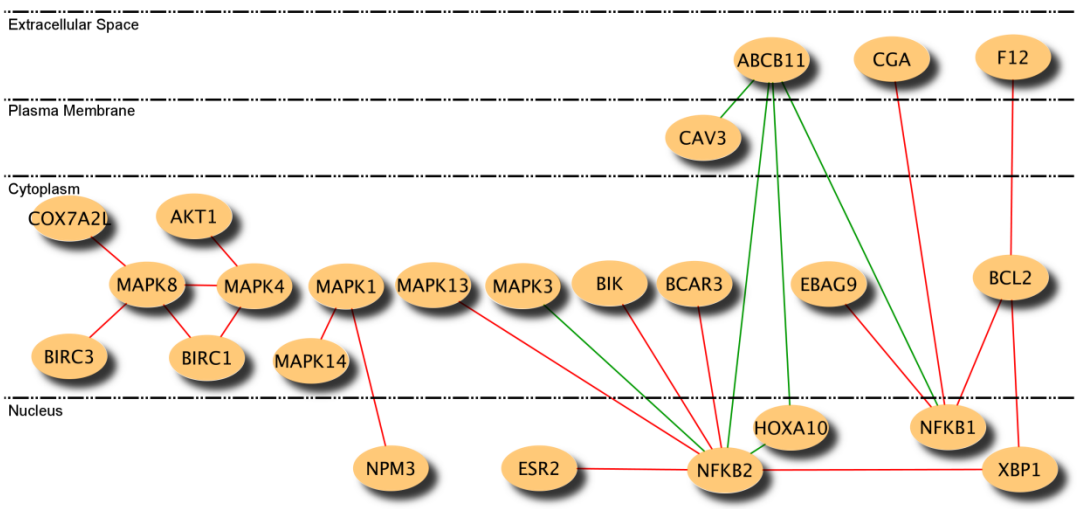


Figure 18 Sample network generated for breast cancer research, use Cytoscape to visualize the differential dependency network.

4.4 Runtime comparison

Standalone package running Speed is another concern for this transplantation method. Firstly, it has different situations, such as standalone jar package with GUI, standalone jar package with command line parameters. Also, it depends on several conditions, whether the data matrix has already cached, whether parameters are the same and whether the shared libraries need to be extracted from the jar package, etc.

The software testing platform has Intel® Core™2 CPU T5600 @1.83GHz, 2.00GB memory, running a Windows XP professional SP3 operating system. In order to do a fair runtime comparison between different situations, we take the simulation data sample as testing material. It has a 20*50 data matrix, denotes 20 interested genes and 50 samples for each gene. Meanwhile, we set all the parameters to Max_k = 1, threshold = 0.25, p_value_cutoff = 0.05. The shared libraries extraction time is negligible, because usually the size of a shared library is below 200KB, after applying pre-allocate buffer in Java, the extraction work will be completed in millisecond level. Table 2 presents a detail running time comparison between 3 situations.

Situations	*Startup time*	*Running time*	*Running time 2nd*
Matlab source code	20″ 29	49″ 93	49″ 71
Standalone Jar package (Graphic user interface)	12″ 48	59″ 95	52″ 18
Standalone Jar package (Command line parameters)	~0	59″ 34	59″ 76

Table 2 Runtime comparison between different situations

Note1: Startup time means the time it takes for a computer to open the software that execute the algorithm

Note2: Running time 2nd means the second time running the algorithm with the same parameters (cached information)

There are a few things need to be mentioned. Firstly, comparing to the other two situations, the ready time for Matlab software is bad, however this is only the first startup time for Matlab, the second time startup time will be shorten to around 10 seconds. Secondly, the reason for standalone package with GUI has more than 12 seconds startup time is not only because the initialization of graphic interface, but mainly because of setting up the MCR environment as mentioned in chapter 3 java driver. Finally, the second running time is basically the same compare to their first running time except for GUI. This is because for the second time, graphic interface initialization process and MCR environment setup time has been reduced to 0.

According to table 2, we can conclude that for small datasets and simple parameters the running time differences for standalone packages compare to Matlab source code are in an acceptable range. The time gap between those two is due to the variable mapping process, because for standalone package, the core algorithm is still running inside Matlab which running time should equal to the Matlab source code. Apparently, if we want to have easier transplantation, we cannot expect better performance. Comparing to hard translation the Matlab source code to the other programming language, efficiency is the biggest disadvantage of this method.

Chapter 5 Discussion and Future work

Matlab is a fast growing development tool with many useful functions and toolboxes. It is very convenient to complete the core algorithm design in Matlab. With the help of JNI, using Java GUI and data management service will greatly decrease the coding difficulty, shorten the development cycle and improve software reliability. In particular, by using Matlab Compiler calling dynamic link library, the standalone package eliminate the code dependency on Matlab main program, greatly expand the application area of Matlab. Like the DDN standalone jar package developed above, every similar standalone package can be distributed with a version specific MCRInstaller. It is easy to deploy and do not restricted by Matlab license any more. This benefit those developers who still love to use Matlab as their main developing tool but worried about Matlab license will limit their distribution of the work.

For future work, the DDN standalone jar package can be embedded into GDOC (Georgetown Database of Cancer) server, provide a web based tool for cancer research community. The architecture is shown in Fig.19. Also, it can be integrated into caBIG, an open source platform where researchers can share their tools. The standalone DDN jar package can be expected to fit into caBIG standard working flow as a service. Apparently, it will further widen the availability of DDN. Taverna is a good platform recommended by caBIG. It is an open source family of tools for designing and executing workflows, created by the myGrid project. Taverna allows for the automation of experimental methods through the use of a number of different services (such as Web services) from a very diverse set of domains – from biology, chemistry and medicine to music, meteorology and social sciences. Effectively Taverna allows a scientist with limited computing background and limited technical resources and support to construct highly complex analyses over public and private data and computational resources, all from a standard PC, UNIX box or Apple computer. The screenshot below shows the Taverna 2.1 Workbench in action during the design of a workflow.

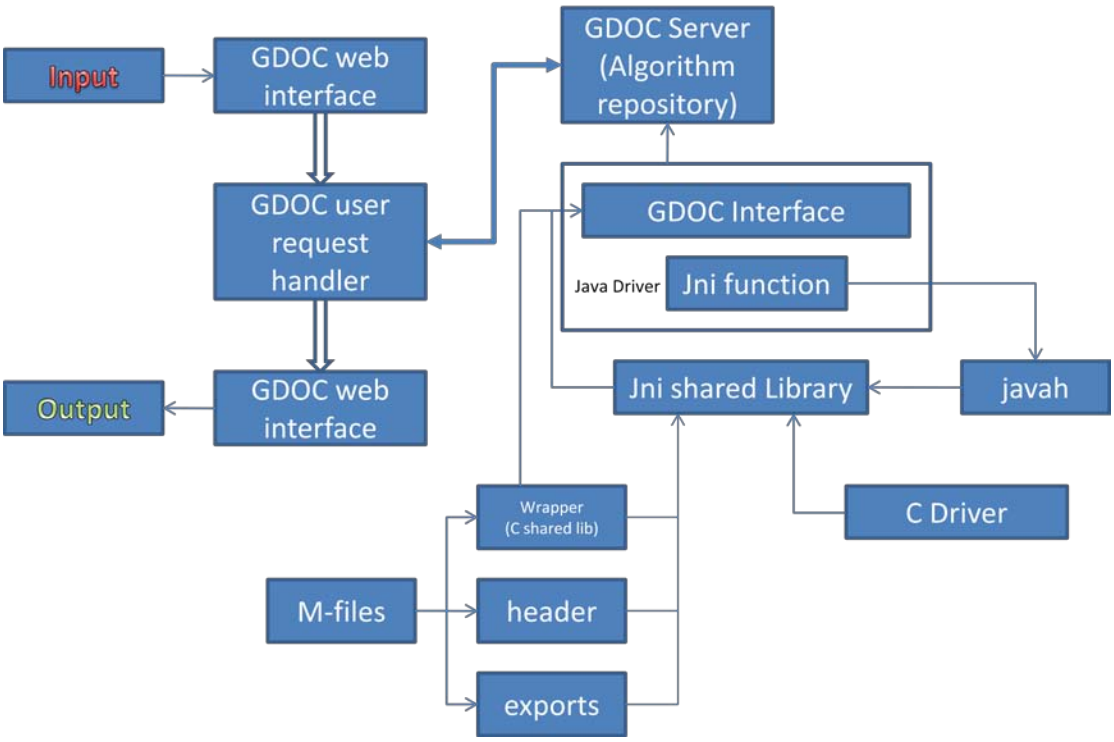


Figure 19 Architecture of DDN standalone package with G-DOC integration

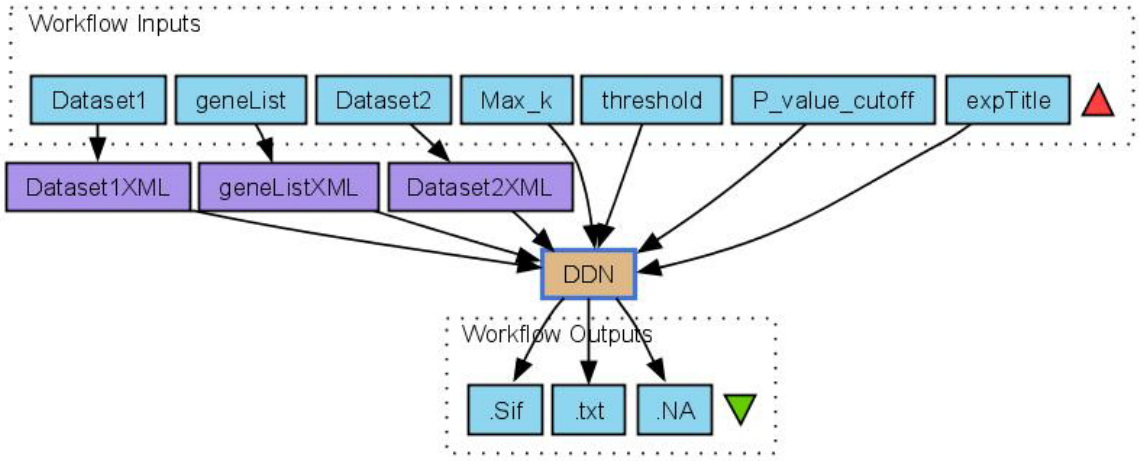


Figure 20 DDN integrated with Taverna Workbench 2.1.0 demo

Inside Taverna, user can choose or build their service from top left window, where they can find all different kinds of services available on the internet. Then with the help of drag-and-drop interface, user drags the required services and connects them together in the down left window by drawing arrows, as shown in Figure 19. A workflow diagram will form synchronically with those drag and draw in the workflow explorer. At last, press the green "start" button in the top menu to execute the workflow. Moreover, recently, "myExperiment" is integrated into Taverna workbench, which is a repository for existing workflow. Before

starting to build a workflow, user can look up in there and avoid the repetitive work. Taverna has already supported R shell in the work flow by directly copy the R source code into the R shell. I believe eventually it will support Matlab code. For now, bean shell with classloader may functional as well [9;10].

Although Matlab has the advantage in some aspect, the risen of lots of small and open source development tool is a threat to its position, such as R. Those development tools usually master in a specific research area, extremely convenient to use within that area. Most important, these tools are similar to Java, have platform independent quality. Building a standalone Matlab package is not very complicated. However, if the developer understands clearly about the orientation of his software, it is not a bad idea to create the software with those more targeted tools.

References

[1] Peter Jurica and Cees van Leeuwen, "OMPC: an open-source MATLAB-to-Python compiler," *Frontiers in Neuroinformatics*, vol. 3 Feb.2009.

[2] Robert Yu and Peter Jurica, "One-button MATLAB-to-C Conversion," Catalytic, Inc., 2007.

[3] Bitao Liu, "The Omega Project for Statistial Computing - The RMatlab Package," 2004.

[4] "Matlab Compiler Documentation," 2010.

[5] B. Zhang, H. Li, R. B. Riggins, M. Zhan, J. Xuan, Z. Zhang, E. P. Hoffman, R. Clarke, and Y. Wang, "Differential dependency network analysis to identify condition-specific topological changes in biological networks," *Bioinformatics.*, vol. 25, no. 4, pp. 526-532, Feb.2009.

[6] R Tibshirani, "Regression shrinkage and selection via the Lasso," *Journal of the Royal Statistical Society Series B-Methodological*, vol. 58, pp. 267-288, 1996.

[7] R. Clarke, H. W. Ressom, A. Wang, J. Xuan, M. C. Liu, E. A. Gehan, and Y. Wang, "The properties of high-dimensional data spaces: implications for exploring gene and protein expression data," *Nat. Rev. Cancer*, vol. 8, no. 1, pp. 37-49, Jan.2008.

[8] Sheng Liang, *Java Native Interface: Programmer's Guide and Reference* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1999.

[9] P Missier, S Soiland-Reyes, Stuart Owen, Wei Tan, Alexandra Nenadic, Ian Dunlop, Alan Williams, Tom Oinn, and Carole Goble, "Taverna Reloaded," 2010.

[10] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn, "Taverna: a tool for building and running workflows of services," *Nucleic Acids Res.*, vol. 34, no. Web Server issue, p. W729-W732, July2006.

Appendix A General Java source code

```
/*
 *
 * @author Lu Jin
 */
import java.io.*;
import java.net.URL;

public class ddnwrapperDriver {
	/*
	 * Native methods corresponding to the functions
	 * that we need to call in the compiler-generated
	 * shared library.
	 */

	public static native void applicationInitialize();

	public static native void libInitialize();

	public native void ddnwrapper(String a, String b, String c, double max_k, double
threshold, double p_value_cutoff, String expTitle);

	public static native void libTerminate();

	public static native void applicationTerminate();

	public ddnwrapperDriver() {
	}

	public static void copyFormJar(String fileUrl, String dest)
		throws IOException {
		int BUFF_SIZE = 100000;
		byte[] buffer = new byte[BUFF_SIZE];
		InputStream in = null;
		OutputStream out = null;
		URL url = new URL(fileUrl);
		try {
			in = url.openStream();
			out = new FileOutputStream(dest);
			while (true) {
				synchronized (buffer) {
					int amountRead = in.read(buffer);
```

```
                if (amountRead == -1) {
                    break;
                }
                out.write(buffer, 0, amountRead);
            }
        }
    } finally {
        if (in != null) {
            in.close();
        }
        if (out != null) {
            out.close();
        }
    }
}
```

Appendix B Java GUI (windows version) Source code

```
/*
 * DDNUI.java
 *
 * Created on Dec 15, 2009, 1:15:18 PM
 */
import java.io.*;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JFileChooser;

/**
 *
 * @author Lu Jin
 */
public class DDNUI extends javax.swing.JFrame {

    /** Creates new form DDNUI */
    public DDNUI() {
        initComponents();
        Dataset1Location.setEditable(false);
        Dataset2Location.setEditable(false);
        GenelistLocation.setEditable(false);
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated
Code">//GEN-BEGIN:initComponents
    private void initComponents() {

        openfile = new javax.swing.JFileChooser();
        Default = new javax.swing.JButton();
        Cancel = new javax.swing.JButton();
        OK = new javax.swing.JButton();
        jPanel1 = new javax.swing.JPanel();
        jLabel5 = new javax.swing.JLabel();
```

```
jLabel6 = new javax.swing.JLabel();
jLabel7 = new javax.swing.JLabel();
Dataset1Location = new javax.swing.JTextField();
Dataset2Location = new javax.swing.JTextField();
GenelistLocation = new javax.swing.JTextField();
jButton4 = new javax.swing.JButton();
jButton5 = new javax.swing.JButton();
jButton6 = new javax.swing.JButton();
jPanel2 = new javax.swing.JPanel();
jLabel1 = new javax.swing.JLabel();
jLabel2 = new javax.swing.JLabel();
jLabel3 = new javax.swing.JLabel();
jLabel4 = new javax.swing.JLabel();
Maxk = new javax.swing.JTextField();
Threshold = new javax.swing.JTextField();
Exptitle = new javax.swing.JTextField();
Pvaluecutoff = new javax.swing.JTextField();

setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
setTitle("DDN UI");

Default.setText("Reset to Default");
Default.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        DefaultActionPerformed(evt);
    }
});

Cancel.setText("Cancel");
Cancel.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        CancelActionPerformed(evt);
    }
});

OK.setText("OK");
OK.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        OKActionPerformed(evt);
    }
});
```

```
        jPanel1.setBorder(javax.swing.BorderFactory.createTitledBorder(null, "Data
Input", javax.swing.border.TitledBorder.DEFAULT_JUSTIFICATION,
javax.swing.border.TitledBorder.DEFAULT_POSITION, new java.awt.Font("Tahoma", 1,
12))); // NOI18N

        jLabel5.setText("Dataset1:");

        jLabel6.setText("Dataset2:");

        jLabel7.setText("GeneList:");

        jButton4.setText("Open");
        jButton4.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton4ActionPerformed(evt);
            }
        });

        jButton5.setText("Open");
        jButton5.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton5ActionPerformed(evt);
            }
        });

        jButton6.setText("Open");
        jButton6.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton6ActionPerformed(evt);
            }
        });

        javax.swing.GroupLayout jPanel1Layout = new
javax.swing.GroupLayout(jPanel1);
        jPanel1.setLayout(jPanel1Layout);
        jPanel1Layout.setHorizontalGroup(

jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(jPanel1Layout.createSequentialGroup()
                .addGap(19, 19, 19)

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILI
NG)
```

```
                                .addComponent(jLabel6)
                                .addComponent(jLabel7)
                                .addComponent(jLabel5))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADI
NG, false)
                                .addComponent(Dataset1Location,
javax.swing.GroupLayout.DEFAULT_SIZE, 280, Short.MAX_VALUE)
                                .addComponent(Dataset2Location)
                                .addComponent(GenelistLocation))
                            .addGap(18, 18, 18)

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADI
NG, false)
                                .addComponent(jButton5,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)
                                .addComponent(jButton6,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)
                                .addComponent(jButton4,
javax.swing.GroupLayout.DEFAULT_SIZE, 81, Short.MAX_VALUE))
                            .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE))
            );

            jPanel1Layout.linkSize(javax.swing.SwingConstants.HORIZONTAL, new
java.awt.Component[] {jLabel5, jLabel6, jLabel7});

            jPanel1Layout.setVerticalGroup(

jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
jPanel1Layout.createSequentialGroup()

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASEL
INE, false)
                                .addComponent(jLabel5)
                                .addComponent(Dataset1Location,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
```

```
                        .addComponent(jButton4))
                    .addGap(17, 17, 17)

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASEL
INE)
                        .addComponent(jLabel6,
javax.swing.GroupLayout.PREFERRED_SIZE, 14,
javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addComponent(Dataset2Location,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addComponent(jButton5))
                    .addGap(17, 17, 17)

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASEL
INE)
                        .addComponent(jLabel7,
javax.swing.GroupLayout.PREFERRED_SIZE, 14,
javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addComponent(GenelistLocation,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addComponent(jButton6))
                    .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE))
        );

        jPanel2.setBorder(javax.swing.BorderFactory.createTitledBorder(null,
"Parameters", javax.swing.border.TitledBorder.DEFAULT_JUSTIFICATION,
javax.swing.border.TitledBorder.DEFAULT_POSITION, new java.awt.Font("Tahoma", 1,
12))); // NOI18N

        jLabel1.setText("Max_k:");

        jLabel2.setText("Threshold:");

        jLabel3.setText("P_Value_Cutoff:");

        jLabel4.setText("expTitle:");
        jLabel4.setToolTipText("The beginning Letter for output files");

        javax.swing.GroupLayout jPanel2Layout = new
javax.swing.GroupLayout(jPanel2);
```

```
        jPanel2.setLayout(jPanel2Layout);
        jPanel2Layout.setHorizontalGroup(

jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .addGroup(jPanel2Layout.createSequentialGroup()
                        .addContainerGap()

.addGroup(jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADI
NG)
                                .addComponent(jLabel1)
                                .addComponent(jLabel3))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADI
NG, false)
                                .addComponent(Pvaluecutoff)
                                .addComponent(Maxk, javax.swing.GroupLayout.DEFAULT_SIZE,
113, Short.MAX_VALUE))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED, 40,
Short.MAX_VALUE)

.addGroup(jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADI
NG)
                                .addComponent(jLabel2)
                                .addComponent(jLabel4))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADI
NG)
                                .addComponent(Exptitle,
javax.swing.GroupLayout.PREFERRED_SIZE, 89,
javax.swing.GroupLayout.PREFERRED_SIZE)
                                .addComponent(Threshold,
javax.swing.GroupLayout.PREFERRED_SIZE, 106,
javax.swing.GroupLayout.PREFERRED_SIZE))
                        .addContainerGap(32, Short.MAX_VALUE))
        );

        jPanel2Layout.linkSize(javax.swing.SwingConstants.HORIZONTAL, new
java.awt.Component[] {Exptitle, Maxk, Pvaluecutoff, Threshold});
```

```
        jPanel2Layout.setVerticalGroup(

jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .addGroup(jPanel2Layout.createSequentialGroup()
                        .addContainerGap()

.addGroup(jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASEL
INE)
                                .addComponent(jLabel2)
                                .addComponent(jLabel1)
                                .addComponent(Maxk,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
                                .addComponent(Threshold,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE))
                        .addGap(18, 18, 18)

.addGroup(jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASEL
INE)
                                .addComponent(jLabel3)
                                .addComponent(Pvaluecutoff,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
                                .addComponent(jLabel4)
                                .addComponent(Exptitle,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE))
                        .addContainerGap())
        );

        javax.swing.GroupLayout layout = new
javax.swing.GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
                layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .addGroup(layout.createSequentialGroup()
                        .addContainerGap()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                                .addGroup(layout.createSequentialGroup()
```

```
.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                                        .addComponent(jPanel1,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)
                                        .addComponent(jPanel2,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE))
                                .addContainerGap())
                            .addGroup(layout.createSequentialGroup()
                                .addComponent(Default)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED, 187,
Short.MAX_VALUE)
                                .addComponent(OK,
javax.swing.GroupLayout.PREFERRED_SIZE, 79,
javax.swing.GroupLayout.PREFERRED_SIZE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                                .addComponent(Cancel,
javax.swing.GroupLayout.PREFERRED_SIZE, 75,
javax.swing.GroupLayout.PREFERRED_SIZE)
                                .addGap(12, 12, 12))))
            );

            layout.linkSize(javax.swing.SwingConstants.HORIZONTAL, new
java.awt.Component[] {Cancel, OK});

            layout.setVerticalGroup(
                layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .addGroup(layout.createSequentialGroup()
                    .addContainerGap()
                    .addComponent(jPanel1, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                    .addComponent(jPanel2, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                            .addComponent(Default)
```

```
                        .addComponent(OK)
                        .addComponent(Cancel))
                    .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE))
        );

        pack();
    }// </editor-fold>//GEN-END:initComponents

    private void DefaultActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_DefaultActionPerformed
        Maxk.setText("1");
        Threshold.setText("0.25");
        Pvaluecutoff.setText("0.05");
}//GEN-LAST:event_DefaultActionPerformed

    private void CancelActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_CancelActionPerformed
        System.exit(0);
}//GEN-LAST:event_CancelActionPerformed

    private void jButton4ActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_jButton4ActionPerformed
        int returnVal = openfile.showOpenDialog(this);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
            Dataset1Location.setText(openfile.getSelectedFile().getPath());
        }
    }//GEN-LAST:event_jButton4ActionPerformed

    private void jButton5ActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_jButton5ActionPerformed
        int returnVal = openfile.showOpenDialog(this);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
            Dataset2Location.setText(openfile.getSelectedFile().getPath());
        }
    }//GEN-LAST:event_jButton5ActionPerformed

    private void jButton6ActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_jButton6ActionPerformed
        int returnVal = openfile.showOpenDialog(this);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
            GenelistLocation.setText(openfile.getSelectedFile().getPath());
        }
```

```
    }//GEN-LAST:event_jButton6ActionPerformed

    private void OKActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_OKActionPerformed

        String curDir = System.getProperty("user.dir");
        String[] args = new String[4];
        args[0] = "jar:file:/" + curDir + "\\DDN
UserInterface.jar!/jniddnwrapperDriver.dll";
        args[1] = curDir + "\\jniddnwrapperDriver.dll";
        args[2] = "jar:file:/" + curDir + "\\DDN UserInterface.jar!/libddnwrapper.dll";
        args[3] = curDir + "\\libddnwrapper.dll";
        try {
            ddnwrapperDriver.copyFormJar(args[0], args[1]);
            ddnwrapperDriver.copyFormJar(args[2], args[3]);
        } catch (IOException ex) {
            Logger.getLogger(DDNUI.class.getName()).log(Level.SEVERE, null, ex);
        }

        File lib = new File(new File(curDir), "jniddnwrapperDriver.dll");
        System.load(lib.getAbsolutePath());
        String a = Dataset1Location.getText();
        String b = Dataset2Location.getText();
        String c = GenelistLocation.getText();
        double max_k = Double.parseDouble(Maxk.getText());
        double threshold = Double.parseDouble(Threshold.getText());
        double p_value_cutoff = Double.parseDouble(Pvaluecutoff.getText());
        String expTitle = Exptitle.getText();
        ddnwrapperDriver.applicationInitialize();
        ddnwrapperDriver.libInitialize();
        ddnwrapperDriver md = new ddnwrapperDriver();
        md.ddnwrapper(a, b, c, max_k, threshold, p_value_cutoff, expTitle);
        ddnwrapperDriver.libTerminate();
        ddnwrapperDriver.applicationTerminate();
    }//GEN-LAST:event_OKActionPerformed

    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) throws IOException {
        System.out.println("1.GUI");
        System.out.println("2.Text base");
        Scanner s = new Scanner(System.in);
```

```
System.out.print("Please choose one:");
double a = s.nextDouble();
if (a == 1) {
	java.awt.EventQueue.invokeLater(new Runnable() {

		public void run() {
			new DDNUI().setVisible(true);
		}
	});
}
else if (a == 2) {
	String curDir = System.getProperty("user.dir");
	args = new String[4];
	args[0] = "jar:file:/" + curDir + "\\DDN
UserInterface.jar!/jniddnwrapperDriver.dll";
	args[1] = curDir + "\\jniddnwrapperDriver.dll";
	args[2] = "jar:file:/" + curDir + "\\DDN
UserInterface.jar!/libddnwrapper.dll";
	args[3] = curDir + "\\libddnwrapper.dll";

	ddnwrapperDriver.copyFormJar(args[0], args[1]);
	ddnwrapperDriver.copyFormJar(args[2], args[3]);
	File lib = new File(new File(curDir), "jniddnwrapperDriver.dll");
	System.load(lib.getAbsolutePath());

	Scanner v = new Scanner(System.in);
	System.out.print("Please input the dataset1 file path/name: ");
	String b = v.next();
	System.out.print("Please input the dataset2 file path/name: ");
	String c = v.next();
	System.out.print("Please input the geneList file path/name: ");
	String d = v.next();
	System.out.print("Please input max_K: ");
	double max_k = v.nextDouble();
	System.out.print("Please input threshold: ");
	double threshold = v.nextDouble();
	System.out.print("Please input p value cutoff:");
	double p_value_cutoff = v.nextDouble();
	System.out.print("Please input expTitle: ");
	String expTitle = v.next();

	/* Call the application initialization routine. */
	ddnwrapperDriver.applicationInitialize();
```

```
            /* Call the library initialization routine. */
            ddnwrapperDriver.libInitialize();
            /* Call the library functions */
            ddnwrapperDriver md = new ddnwrapperDriver();
            md.ddnwrapper(b, c, d, max_k, threshold, p_value_cutoff, expTitle);
            System.out.println("done!");

            /* Call the library termination routine. */
            ddnwrapperDriver.libTerminate();
            /* Call the application termination routine. */
            ddnwrapperDriver.applicationTerminate();
        }
        else {
            System.out.println("Error input!");
        }
    }
    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JButton Cancel;
    private javax.swing.JTextField Dataset1Location;
    private javax.swing.JTextField Dataset2Location;
    private javax.swing.JButton Default;
    private javax.swing.JTextField Exptitle;
    private javax.swing.JTextField GenelistLocation;
    private javax.swing.JTextField Maxk;
    private javax.swing.JButton OK;
    private javax.swing.JTextField Pvaluecutoff;
    private javax.swing.JTextField Threshold;
    private javax.swing.JButton jButton4;
    private javax.swing.JButton jButton5;
    private javax.swing.JButton jButton6;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JLabel jLabel3;
    private javax.swing.JLabel jLabel4;
    private javax.swing.JLabel jLabel5;
    private javax.swing.JLabel jLabel6;
    private javax.swing.JLabel jLabel7;
    private javax.swing.JPanel jPanel1;
    private javax.swing.JPanel jPanel2;
    private javax.swing.JFileChooser openfile;
    // End of variables declaration//GEN-END:variables
}
```

Appendix C Java Driver for Linux (GDOC version) Source code

```
/*
  * Java class to the ddnwrapperDriver from Java.
  * This class assumes a shared lib called
  * (lib)jniddnwrapperDriver.(so,dll) that contains the
  * necessary gateway functions for initialization,
  * termination, and execution of the compiled m-functions.
  * This shared library in turn links against a compiler
  * generated shared lib (libddnwrapper) that contains the actual
  * m-functions.
  */

/**
  *
  * @author Lu Jin
  */
import java.io.*;
import java.net.URL;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ddnwrapperDriver {
        /*
          * Native methods corresponding to the functions
          * that we need to call in the compiler-generated
          * shared library.
```

```
  */

    public static native void applicationInitialize();

    public static native void libInitialize();

    public native void ddnwrapper(String a, String b, String c, double max_k, double
threshold, double p_value_cutoff, String expTitle);

    public static native void libTerminate();

    public static native void applicationTerminate();

    public ddnwrapperDriver() {
    }

    public static void copyFromJar(String fileUrl, String dest)
            throws IOException {
        int BUFF_SIZE = 100000;
        byte[] buffer = new byte[BUFF_SIZE];
        InputStream in = null;
        OutputStream out = null;
        URL url = new URL(fileUrl);
        try {
            in = url.openStream();
            out = new FileOutputStream(dest);
            while (true) {
                synchronized (buffer) {
                    int amountRead = in.read(buffer);
                    if (amountRead == -1) {
                        break;
                    }
                    out.write(buffer, 0, amountRead);
                }
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
```

```
    }

    public static void main(String[] args) {
        /* unzip */
        String curDir = System.getProperty("user.dir");
        String a = "jar:file:" + curDir + "/ddn.jar!/jniddnwrapperDriver.so";
        String b = curDir + "/jniddnwrapperDriver.so";
        String c = "jar:file:" + curDir + "/ddn.jar!/libddnwrapper.so";
        String d = curDir + "/libddnwrapper.so";
        try {
            ddnwrapperDriver.copyFromJar(a, b);
            ddnwrapperDriver.copyFromJar(c, d);
        } catch (IOException ex) {
            Logger.getLogger(ddnwrapperDriver.class.getName()).log(Level.SEVERE,
null, ex);
        }


        /* set path */
        File lib = new File(new File(curDir), "jniddnwrapperDriver.so");
        System.load(lib.getAbsolutePath());


        /* parameters conversion & check */
        try {
            String dataset1 = args[0];
            String dataset2 = args[1];
            String geneList = args[2];
            double max_k = Double.parseDouble(args[3]);
            double threshold = Double.parseDouble(args[4]);
            double p_value_cutoff = Double.parseDouble(args[5]);
            String expTitle = args[6];


            /* Call the application initialization routine. */
            applicationInitialize();
            /* Call the library initialization routine. */
            libInitialize();
            /* Call the library functions */
            ddnwrapperDriver md = new ddnwrapperDriver();
            md.ddnwrapper(dataset1, dataset2, geneList, max_k, threshold,
p_value_cutoff, expTitle);
            System.out.println("done!");
```

```
            } catch (ArrayIndexOutOfBoundsException ex) {
                System.out.println("Usage: java -jar ddn.jar <dataset1> <dataset2>
<geneList> <max_k> <threshold> <p_value_cutoff> <output_file_prefix>");
                System.out.println("Example: java -jar ddn.jar dataset1.csv dataset2.csv
geneList.csv 1 0.25 0.05 TEST");
                System.exit(0);
            }
                /* Call the library termination routine. */
                libTerminate();
                /* Call the application termination routine. */
                applicationTerminate();
        }
}
```

Appendix D C Driver for DDN

```
/*==========================================================
============
 *
 * JNIDDNWRAPPERDRIVER.C
 *              Driver code that calls the shared library created
 *              using Matlab Compiler from JNI. Refer to the documentation
 *              of Matlab Compiler for more information on this.
 *
 * This is the wrapper C code to call a shared library created
 * using Matlab Compiler.
 *
 *

*==========================================================
============*/

/*
 * Include the javah-generated header file and the library specific header file
 * as generated by Matlab Compiler.
 */
#include "ddnwrapperDriver.h"
#include "libddnwrapper.h"

/*
 * Helper function to throw a java exception.
 * This function takes the exception class name and
```

```
 * message as input, then throws a Java exception.
 */
static void throw_java_exception(JNIEnv *env, const char* name, const char* msg)
{
	jclass cls = (*env)->FindClass(env, name);
	if (cls != NULL)
	{
		(*env)->ThrowNew(env, cls, msg);
		(*env)->DeleteLocalRef(env, cls);
	}
}
/*
 * Helper function to marshal a Java double matrix into an mxArray.
 * This function assumes the input array is of type double[][] and
 * square. No check on the type is done. Column dimension is obtained
 * from size of first row.
 */
static mxArray* javamatrix_to_mxArray(JNIEnv* env, jobjectArray arr)
{
	mxArray* px;		/* mxArray to return */
	int m;			/* Number of rows */
	int n;			/* Number of columns */
	int i,j;		/* Loop variables */
	jdoubleArray row;	/* Stores current row */

	if (arr == NULL)
		return NULL;
	/* Get number of rows and columns */
	m = (int)(*env)->GetArrayLength(env, arr);
	if ((row = (jdoubleArray)(*env)->GetObjectArrayElement(env, arr, 0)) == NULL)
		return NULL;
	n = (int)(*env)->GetArrayLength(env, row);
	(*env)->DeleteLocalRef(env, row);
	/* Create mxArray */
	px = mxCreateDoubleMatrix(m, n, mxREAL);
	/* Loop over rows */
	for (i=0; i<m; i++)
	{
		int subs[2];		/* Stores subscripts for mxArray indexing */
		int offset;		/* Offset in mxArray of value */
		double* pval;		/* Current value to copy */
		subs[0] = i;
		row = (jdoubleArray)(*env)->GetObjectArrayElement(env, arr, (jsize)i);
```

```
		/* Loop over columns */
		for (j=0; j<n; j++)
		{
			subs[1] = j;
			offset = mxCalcSingleSubscript(px, 2, subs);
			pval = mxGetPr(px) + offset;
			(*env)->GetDoubleArrayRegion(env, row, (jsize)j, 1, pval);
		}
		(*env)->DeleteLocalRef(env, row);
	}
	return px;
}
/**
 * Helper function to marshal a Java double into an mxArray.
*/
static mxArray* javadouble_to_mxArray(JNIEnv* env, jdouble arr)
{
	mxArray *px;
	double arr2 = arr; /* get data from java variable*/
	px = mxCreateDoubleScalar(arr2); /*copy data to mxArray*/
	return px;
}
/**
 * Helper function to marshal a Java string into an mxArray.
*/
static mxArray* javastring_to_mxArray(JNIEnv* env, jstring arr)
{
	mxArray *px;
	const char* arr2 = (*env) -> GetStringUTFChars(env,arr, 0); /*get data from java
variable*/
	px = mxCreateString(arr2); /* copy data to mxArray*/
	return px;
}
/* Application level initialization */
JNIEXPORT
void JNICALL Java_ddnwrapperDriver_applicationInitialize(JNIEnv *env, jclass cls)
{
	(void)env;
	(void)cls;
	if (!mclInitializeApplication(NULL, 0))
	{
		throw_java_exception(env,
			"java/lang/RuntimeException",
```

```
                "Could not initialize the application.");
        }
}
/* Library level initialization */
JNIEXPORT
void JNICALL Java_ddnwrapperDriver_libInitialize(JNIEnv *env, jclass cls)
{
        (void)env;
        (void)cls;
        if (!libddnwrapperInitialize())
        {
                throw_java_exception(env,
                        "java/lang/RuntimeException",
                        "Could not initialize the library.");
        }
}

/* Calls DDN function */
JNIEXPORT
void JNICALL Java_ddnwrapperDriver_ddnwrapper(JNIEnv *env, jobject obj, jstring a,
jstring b, jstring c, jdouble d, jdouble e, jdouble f, jstring g)
{
        mxArray* pa = NULL;
        mxArray* pb = NULL;
        mxArray* pc = NULL;
        mxArray* pd = NULL;
        mxArray* pe = NULL;
        mxArray* pf = NULL;
        mxArray* pg = NULL;
        mxArray* ph = NULL;

        /* Process input arrays */
        if ((pa = javastring_to_mxArray(env, a)) == NULL)
                goto EXIT;
        if ((pb = javastring_to_mxArray(env, b)) == NULL)
                goto EXIT;
        if ((pc = javastring_to_mxArray(env, c)) == NULL)
                goto EXIT;
        if ((pd = javadouble_to_mxArray(env, d)) == NULL)
                goto EXIT;
        if ((pe = javadouble_to_mxArray(env, e)) == NULL)
                goto EXIT;
        if ((pf = javadouble_to_mxArray(env, f)) == NULL)
```

```
            goto EXIT;
        if ((pg = javastring_to_mxArray(env, g)) == NULL)
            goto EXIT;

        /* Call ddnwrapper */
        mlfDdnwrapper(1, &ph, pa, pb, pc, pd, pe, pf, pg);
        /* Process output array */
        return 1;
EXIT:
        if (pa != NULL)
            mxDestroyArray(pa);
        if (pb != NULL)
            mxDestroyArray(pb);
        if (pc != NULL)
            mxDestroyArray(pc);
        if (pd != NULL)
            mxDestroyArray(pd);
        if (pe != NULL)
            mxDestroyArray(pe);
        if (pf != NULL)
            mxDestroyArray(pf);
        if (pg != NULL)
            mxDestroyArray(pg);
        if (ph != NULL)
            mxDestroyArray(ph);
        return 0;
}

/* Library level termination */
JNIEXPORT
void JNICALL Java_ddnwrapperDriver_libTerminate(JNIEnv *env, jclass cls)
{
        (void)env;
        (void)cls;
        libddnwrapperTerminate();
}

/* Application level termination */
JNIEXPORT
void JNICALL Java_ddnwrapperDriver_applicationTerminate(JNIEnv *env, jclass cls)
{
        (void)env;
        (void)cls;
```

```
mclTerminateApplication();
}
```