Plugin Based Microbiome Analysis (PLUMA ) Version 1.1 - User Guide

Trevor Cickovski and Giri Narasimhan
School of Computing and Information Sciences
Florida International University
tcickovs@fiu.edu, giri@fiu.edu

http://biorg.cis.fiu.edu/pluma/

Other Contributors from Florida International University:

Vanessa Aguiar-Pulido
Guillermo Barquero
Cesia Bulnes
Michael Campos
Bhavyta Chauhan
Mark Fajet
Mitch Fernandez
Wenrui Huang
Shamsed Mahmoud
Veronica Parra
Jingan Qu
Joseph Quinn
Juan Daniel Riveros
Victoria Suarez-Ulloa
Rishabh Vaidya
Camilo Valdez

July 10, 2020
Abstract

We present PLUMA, a lightweight and flexible package for constructing general software pipelines. PLUMA is designed to be infinitely extensible, allowing researchers to select a specific set of dynamically loaded plugins as sequential stages of their pipelines. These plugins can be implemented by either themselves or other users, in their language of choice.

We begin by introducing the key features of PLUMA, and follow with a discussion of how to download and install the latest version, compile, and run the software. We also include information on setting up configuration files that specify desired plugins for a pipeline, and how to extend PLUMA with new plugins in various programming languages. Finally, we conclude with a full pipeline example and a brief discussion of our envisioned future of PLUMA.

We distribute PLUMA under the MIT Software License, copyrighted by Florida International University. Any professional work that uses PLUMA should provide the following citation:

Acknowledgements

We acknowledge support from the following on the PLUMA project:

- Department of Defense (Contract W911NF-16-1-0494)
- National Institute of Health (Grant 1R15AI128714-01)
- National Institute of Justice (Grant 2017-NE-BX-0001)
- Florida Department of Health (FDOH 09KW-10)
- Alpha-One Fundation
- NVIDIA (CUDA Teaching Center Program)
- The College of Engineering and Computer Science, Florida International University
- The Natural Sciences Collegium, Eckerd College (Faculty Development Grant)
# Contents

1 Introduction 5

2 Availability and Installation of PLUMA 7
   2.1 How to Download PLUMA 7
   2.2 Compiling PLUMA 7
      2.2.1 Dependencies 8
      2.2.2 Environment Variables 8
      2.2.3 Compiling Plugins 9

3 Getting Started 10
   3.1 Command Line 10
      3.1.1 Plugin Location(s) 11
      3.1.2 Other Options 11
   3.2 Configuration File 11
      3.2.1 Comments 12
      3.2.2 Prefix 12
      3.2.3 Plugins 12
      3.2.4 Pipelines 12
      3.2.5 Kitties 13
   3.3 Online Tools 14
      3.3.1 PLUMA Plugin Pool 14
      3.3.2 PLUMA Pipeline Pool 15
      3.3.3 Alpha GUI 16

4 Extending PLUMA 18
   4.1 Converting an Existing Tool to a Plugin 18
      4.1.1 Option 1: Single Command (No Arguments) 18
      4.1.2 Option 2: Input and Output File Arguments 19
      4.1.3 Option 3: Fully Customized (Multiple Arguments) 19
   4.2 Building a New Plugin From Scratch 20
      4.2.1 C++ 20
      4.2.2 CUDA 21
      4.2.3 Python 24
      4.2.4 R 25
      4.2.5 Perl 25
4.3 Building a New Plugin using Existing Plugins and Computational Core 26
4.4 Testing New Plugins 27

5 PLUMA Full Pipeline 30
5.1 Stage 1: Mothur 30
5.2 Stage 2: CountTableProcessing 32
5.3 Stage 3: CSVNormalize 32
5.4 Stage 4: Spearman 32
5.5 Stage 5: CSVPad 33
5.6 Stage 6: GPUATria 33
5.7 Stage 7: CSV2GML 34
5.8 Stage 8: Cytoscape 34

6 The Future of PLUMA 37

A PLUMA Software License 38
A.1 Conditions and Regulations 38
A.2 Contact Information 38
List of Figures

1.1 An example metagenomics analysis pipeline. Each stage gets executed sequentially, with the output of a specific stage serving as input to a later stage of the pipeline. .................. 5
1.2 Conceptual design of PLUMA. Users interact with the software through the user layer, where they can assemble pipelines using plugin extensions in a variety of languages. Python, Perl and R plugins interface to the scripted layer of PLUMA, and compiled plugins in C++ or CUDA to the computational layer. From [5]. .................. 6

3.1 PLUMA plugin pool. Available at http://biorg.cs.fiu.edu/pluma/plugins. .. 15
3.2 PLUMA pipeline pool. Available at http://biorg.cs.fiu.edu/pluma/pipelines. 16
3.3 PLUMA alpha GUI. Available at http://github.com/movingpictures83/PluMA-GUI. 17

5.1 Conceptual description of a full metagenomics pipeline in PLUMA, with plugins in a variety of supported languages and two generated by PluGen. .................. 31
5.2 The equivalent signed and weighted correlation network corresponding to the matrix in Program 13. .................. 33
5.3 Our network visualized with Cytoscape upon executing Stage 8 of our pipeline. .................. 35
5.4 Our Cytoscape visualization after importing our NOA file from Stage 6. .................. 36
Chapter 1

Introduction

Software pipelines are applicable to any field of study and involve control flow execution as a series of stages, with an output of a given stage serving as an input to the next. As an example, Figure 1.1 shows a common metagenomics analysis pipeline, with an initial set of DNA sequences passing through a denoising stage that improves read quality, followed by clustering through some similarity or compositional metric, and finally labelling sequence clusters with the closest matching taxonomic unit [2].

Many pipelines have been developed by independent teams as standalone tools, although stages of individual pipelines could potentially be reused amongst one another. Particularly in the field of metagenomics analysis, there are many software pipelines that perform similar tasks and may even share stages that are still constructed independently, because there is no standardized framework for developing, testing and particularly integrating these stages. PLUMA is designed to address this need by providing a lightweight and generic back end that can execute these stages as dynamically loaded plugins, specified by a user through a configuration file or GUI.

We show the conceptual design of PLUMA in Figure 1.2, which follows a Problem-Solving Environment (PSE, [13]) using three conceptual tiers [8]: a compiled machine layer, a middle scripted layer, and an upper user layer. The user layer consists of plugins [3, 14] which can be developed by various users in the PLUMA community in multiple languages, and subsequently integrated to form pipelines. A user currently performs this integration through a configuration file or the GUI (Alpha version).

Plugins can be developed standalone using either the scripted or computational (compiled) interfaces to PLUMA. The scripted interface currently supports Python, Perl and R, and the compiled interface supports C++ for the CPU and CUDA for the GPU. We also provide a PluginGenerator (PluGen) module which can produce plugins that wrap existing tools on a user’s machine (we will later show examples with Mothur [27] and Cytoscape [10]). Scripted plugins can also refer to each other, and we have a future goal of creating wrappers to the computational layer using SWIG [4]. We also will be expanding our set of supported languages, with Java as our next target.

![Figure 1.1: An example metagenomics analysis pipeline. Each stage gets executed sequentially, with the output of a specific stage serving as input to a later stage of the pipeline.](image)
Figure 1.2: Conceptual design of PLUMA. Users interact with the software through the user layer, where they can assemble pipelines using plugin extensions in a variety of languages. Python, Perl and R plugins interface to the scripted layer of PLUMA, and compiled plugins in C++ or CUDA to the computational layer. From [5].
Chapter 2

Availability and Installation of PLUMA

2.1 How to Download PLUMA

PLUMA is hosted by the Bioinformatics Research Group (BioRG) at Florida International University at http://biorg.cis.fiu.edu/pluma, with source code available through GitHub:

https://github.com/FIUBioRG/PluMA

PLUMA users must agree to MIT software license regulations, part of the Open Source Initiative (OSI). We provide a copy of this license at the end of this User Guide. We have available a plugin pool of dynamically loadable plugin libraries implemented in various languages at:

http://biorg.cs.fiu.edu/pluma/plugins

which contains links to repositories of various PLUMA plugins developed by external users. Pipelines can be assembled using combinations of these plugin extensions. We have a pipeline pool of example pipelines here:

http://biorg.cs.fiu.edu/pluma/pipelines

Finally, an Alpha version of a GUI for PLUMA is available here:

http://github.com/movingpictures83/PluMA-GUI

Once you have all plugins that you need, the GUI can be used to drag and drop these plugins into pipelines.

2.2 Compiling PLUMA

PLUMA uses the SCons (http://www.scons.org) open source software construction tool to compile its back end. Please download and install SCons before compiling PLUMA. Once SCons is installed, the steps to compile PLUMA are:
1. Change to the main pluma directory.

2. Run the command `scons /`. If you run only `scons` and pass no flags, PLuMA will assume you would like compatibility with plugin extensions for all supported languages (currently C++, CUDA, Perl, Python, and R). If you do not need plugins in some of these languages and/or do not wish to install their facilities, you can turn off their compilation by setting the flags `cuda`, `perl`, `python` and `r` to zero (note you cannot turn off C++ since the back end uses it). So for example, `scons cuda=0 /` will build PLuMA without the capability of running CUDA plugins, which should be done if the user does not have an NVIDIA graphics card.

### 2.2.1 Dependencies

Compiling with R support additionally requires installation of the `RInside` package, which can be done using the standard R installation process using the command `install.packages`. This will automatically install another package `Rcpp`, which you will also need.

Perl and Python also assume the user has PThreads installed, though these now come standard on most *nix systems.

### 2.2.2 Environment Variables

Additionally, it is possible that your language tool installation locations are non-standard and you may need to set environment variables so that `scons` will include and link them properly. We have outlined each of these below:

<table>
<thead>
<tr>
<th>Language</th>
<th>Environment Variable</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>PYTHON_INCLUDE_DIR</td>
<td>Directory with Python.h</td>
<td>Location of Python, concatenated with include/python followed by version</td>
</tr>
<tr>
<td>Python</td>
<td>PYTHON_LIB_DIR</td>
<td>Directory with libpython*</td>
<td>Location of Python, concatenated with lib/python, followed by version, followed by config</td>
</tr>
<tr>
<td>Perl</td>
<td>PERL_INCLUDE_DIR</td>
<td>Directory with perl.h</td>
<td>/usr/lib/perl, followed by version, followed by CORE</td>
</tr>
<tr>
<td>Perl</td>
<td>PERL_LIB_DIR</td>
<td>Directory with libperl*</td>
<td>Same as <code>PERL_INCLUDE_DIR</code></td>
</tr>
<tr>
<td>Python/Perl</td>
<td>PTHREAD_LIB_DIR</td>
<td>Directory with libpthread*</td>
<td>/usr/local/lib</td>
</tr>
<tr>
<td>Python/Perl</td>
<td>PTHREAD_INCLUDE_DIR</td>
<td>Directory with pthread.h</td>
<td>/usr/local/include</td>
</tr>
<tr>
<td>R</td>
<td>R_INCLUDE_DIR</td>
<td>Directory with R.h</td>
<td>/usr/share/R/include</td>
</tr>
<tr>
<td>R</td>
<td>R_LIB_DIR</td>
<td>Directory with libR*</td>
<td>/usr/local/lib/R</td>
</tr>
<tr>
<td>R</td>
<td>RINSIDE_INCLUDE_DIR</td>
<td>Directory with RInside.h</td>
<td>R_LIB_DIR, followed by site-library/RInside/include</td>
</tr>
<tr>
<td>R</td>
<td>RINSIDE_LIB_DIR</td>
<td>Directory with libRInside*</td>
<td>R_LIB_DIR, followed by site-library/RInside/lib</td>
</tr>
<tr>
<td>R</td>
<td>RCPP_INCLUDE_DIR</td>
<td>Directory with Rcpp.h</td>
<td>R_LIB_DIR, followed by site-library/Rcpp/include</td>
</tr>
</tbody>
</table>

Note: CUDA will work as long as the NVIDIA compiler `nvcc` is in the system PATH.
2.2.3 Compiling Plugins

Running scons will also automatically compile plugins inside the plugins/ folder of the PLUMA source tree, which comes included (empty) within a PLUMA installation. We recommend installing any desired plugins in this folder; however other plugin directories can also be included in the compilation by specifying them in the environment variable PLUMA_PLUGIN_PATH. As an example, setting PLUMA_PLUGIN_PATH in this way:

```bash
export PLUMA_PLUGIN_PATH=/usr/local/bin/plugins:/home/johndoe/myplugins
```

would automatically include the folders: /home/johndoe/myplugins, /usr/local/bin/plugins, and plugins/ in the scons compilation. At the time of this release, the supported compiled languages for PLUMA are C++ and CUDA. However, if the user passes the flag cuda=0 to scons, CUDA plugins will automatically be excluded from the compilation.

After everything has compiled successfully, the user may run PLUMA from its root directory, which is described in the next chapter.
Chapter 3

Getting Started

We now introduce the commands needed to run PLUMA, including command line formats on *nix machines and configuration file formats. We will also demonstrate some useful features of PLUMA such as log and restart files. In the next section, we will show some real example configuration files to help further illustrate concepts. Although only *nix machines (UNIX, Linux and Mac) are currently supported, we are currently developing a Windows release for PLUMA.

3.1 Command Line

We conveniently name the application file pluma, which can be typed at a UNIX prompt followed by a configuration file and, optionally, a restart point:

```
./pluma (configuration file) (optional restart point)
```

The configuration file can be specified using an absolute or relative path. The restart point is a plugin name from this same configuration file, which if specified causes PLUMA to start from its stage, rather than the first plugin in the configuration file (the default).

As an example, see the three PLUMA executions below and their meanings. These executions assume that PLUMA has been installed in the location /home/johndoe and that the file /home/johndoe/examples/myconfig.txt specifies three plugins Stage1, Stage2 and Stage3, in this order. Note that Stage1, Stage2 and Stage3 immediately become valid possible restart points:

```
./pluma /home/johndoe/examples/myconfig.txt
Run using the configuration file /home/johndoe/examples/myconfig.txt

./pluma examples/myconfig.txt
Same

./pluma examples/myconfig.txt Stage1
Same

./pluma examples/myconfig.txt Stage2
Same, but start with second stage

./pluma examples/myconfig.txt Stage3
Same, but only run last stage
```

Restart points work conveniently with log files, which PLUMA automatically outputs on every pipeline execution into a folder labelled with the date and time. These include each plugin that executed and particularly for a pipeline with many stages, can be a convenient way to determine a point of failure or an interesting intermediate result [26]. The user can subsequently start their pipeline using this intermediate stage as their restart point, rather than rerunning all stages before it (some may be expensive) just to reproduce the same outputs.
3.1.1 Plugin Location(s)

Dynamic loading of PLUMA plugins takes a similar approach to compiling PLUMA plugins. As before, the default location that PLUMA will assume for plugins is the plugins/ folder in the PLUMA root directory. PLUMA will also search for plugins in any directory within the environment variable PLUMA_PLUGIN_PATH. The order is also the same for compilation, though becomes more significant here if two plugins have the same name. Once again, if the variable was set using:

```bash
export PLUMA_PLUGIN_PATH=/usr/local/bin/plugins:/home/johndoe/myplugins
```

PLUMA will first search the default plugins/ directory, followed by /usr/local/bin/plugins, followed by /home/johndoe/myplugins. **If two plugins have the same name, PLUMA will use the second plugin.** Therefore PLUMA_PLUGIN_PATH can be used to define default versions of individual plugins, which can be overridden by local changes from a user.

3.1.2 Other Options

PLUMA also provides command-line options that can provide a user specific information about their software and plugins, as specified in the table below.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>./pluma help</td>
<td>Help with PLUMA command line</td>
</tr>
<tr>
<td>./pluma (with no arguments)</td>
<td>Same</td>
</tr>
<tr>
<td>./pluma version</td>
<td>Version number of PLUMA that is running</td>
</tr>
<tr>
<td>./pluma plugins</td>
<td>List of all currently installed plugins (will reference PLUMA_PLUGIN_PATH as well)</td>
</tr>
</tbody>
</table>

3.2 Configuration File

We now show an example PLUMA configuration file (config.txt), which runs a sample eight-stage metagenomics analysis pipeline that includes some downstream analysis. In the next section, we will illustrate how to build PLUMA plugins in various languages by building some of these plugins:

```
config.txt:
# Metagenomics analysis pipeline
Prefix pipelines/Mouse
Plugin Mothur inputfile input.mothur outputfile none
Plugin CountTableProcessing inputfile mouse.trim.abund.pick.an.unique_list outputfile abund.csv
Plugin CSVNormalize inputfile abund.csv outputfile abund.norm.csv
Plugin Spearman inputfile abund.norm.csv outputfile spearman.csv
Plugin CSVPad inputfile spearman.csv outputfile network.csv
Plugin GPUATria inputfile network.csv outputfile ATria.noa
# Plugin ATria inputfile network.csv outputfile ATria.noa
Plugin CSV2GML inputfile network.csv outputfile network.gml
Plugin Cytoscape inputfile network.visualization.txt outputfile none
```
This configuration file uses three core PLUMA configuration file components: comments, prefixes and plugins. We now discuss each of these.

3.2.1 Comments

PLUMA implements comments by ignoring all characters on a line that follow a pound (#) sign. Comments are useful for writing well-documented configuration files, or also to quickly swap out a plugin and test a different one. For example in the case above we included a plugin for the *Ablatio Triadum* (ATria, [6]) algorithm, but then implemented a more efficient version on the GPU. To test for accuracy and performance improvement, we could simply comment out the CPU version of ATria and add the GPU version.

3.2.2 Prefix

The *Prefix* keyword specifies a relative or absolute path for input and output data files. This saves having to retype this path for multiple input and output files when specifying them for plugins. PLUMA will automatically use the last value of *Prefix* if more than one is specified, and the configuration file is read sequentially (so a user can specify one prefix for part of the configuration file and a different one for the rest). A user can also specify no *Prefix* at all, but then must include paths when specifying input and output files for plugins (or just have them reside in their current working directory).

3.2.3 Plugins

The fundamental component of the PLUMA configuration file is a sequential collection of unique plugin identifiers (one for each stage of the pipeline), their input files, and their output files:

<table>
<thead>
<tr>
<th>Plugin</th>
<th>plugin1 inputfile</th>
<th>inputfilename1</th>
<th>outputfile</th>
<th>outputfilename1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plugin</td>
<td>plugin2 inputfile</td>
<td>inputfilename2</td>
<td>outputfile</td>
<td>outputfilename2</td>
</tr>
<tr>
<td>Plugin</td>
<td>plugin3 inputfile</td>
<td>inputfilename3</td>
<td>outputfile</td>
<td>outputfilename3</td>
</tr>
</tbody>
</table>

Plugins (*plugin1, plugin2, plugin3, etc.*) must be installed within the PLUMA_PLUGIN_PATH or the PLUMA plugins/ folder. Output files are written by their respective plugins, and input files are read. If a particular plugin does not read and/or write a file, specify none as the value of *inputfile* and/or *outputfile*, respectively. A plugin’s input files must exist when it executes, but not necessarily at the start of the pipeline – since very often with pipelines an input file will be an output file from a prior plugin, and probably most often the one immediately before. We had one exception in our above example, with both GPUATria (Stage 6) and CSV2GML (Stage 7) taking the same file *network.csv* as input, which was output by CSVPad (Stage 5).

3.2.4 Pipelines

If the configuration file above (*config.txt*) were applied to microbiome analysis, it really contains three parts:
1. Taking raw sequences and determining which taxa are present in which amounts (plugins Mothur and CountTableProcessing). These amounts are stored in a file abund.csv (for ‘abundances’).

2. Downstream analysis that involves normalizing the data, producing co-occurrence networks and determining important taxa (the rest).

3. Visualizing the results.

   The configuration file could therefore also broken into parts. The first, which we could name config.seq.txt, would contain the first two plugins:

   ```
   config.seq.txt:
   Plugin Mothur inputfile input.mothur outputfile none
   Plugin CountTableProcessing inputfile mouse.trim.abund.pick.an.unique outputfile abund.csv
   ```

   And the second, config.downstream.txt, would contain the rest:

   ```
   config.downstream.txt:
   Plugin CSVNormalize inputfile abund.csv outputfile abund.norm.csv
   Plugin Spearman inputfile abund.norm.csv outputfile spearman.csv
   Plugin CSVPad inputfile spearman.csv outputfile network.csv
   Plugin GPUATria inputfile network.csv outputfile ATRia.noa
   # Plugin ATRia inputfile network.csv outputfile ATRia.noa
   Plugin CSV2GML inputfile network.csv outputfile network.gml
   ```

   config.txt could then act as a ‘parent’ file, which uses the Pipeline keyword to invoke both:

   ```
   config.txt:
   # Metagenomics analysis pipeline
   Prefix pipelines/Mouse
   Pipeline config.seq.txt
   Pipeline config.downstream.txt
   Plugin Cytoscape inputfile network.visualization.txt outputfile none
   ```

   Separating pipelines in this way can be a good way to organize stages and pinpoint potential bugs. They also work well with kitties, which we describe next.

3.2.5 Kitties

   A PLUMA Kitty is a lighter form of a PLUMA Prefix which is temporarily conCATenated to all input and output filenames of ONE Pipeline execution (thus the name ‘kitty’, a softer form of the *nix ‘cat’).

   To see the usefulness of kitties, take the above configuration file. Suppose sequence analysis had already been done for several other organisms: say Human, E.Coli, Fruit Fly, and Oyster. We have abund.csv files
for all of them, that contain the amounts of taxa present as with the Mouse example above. Now, we must perform downstream analysis on all of them. These files are contained in directories pipelines/Human, pipelines/EColi, pipelines/FruitFly and pipelines/Oyster, also similar to the Mouse.

Up to this point, we would need to create a config.txt for each one of these, despite the fact that they only differ in the Prefix. Additionally, they would require completely separate runs of PLUMA on these different configuration files. Finally, they would be almost identical to the Mouse except for two small differences (sequence analysis and visualization), creating potential copy-paste errors.

Kitties would allow us to do all of this within one config.txt, since they only temporarily set the prefix for one Pipeline execution:

```
cfg.txt:
# Metagenomics analysis pipeline
Prefix pipelines/Mouse
Pipeline config.seq.txt
Pipeline config.downstream.txt
Kitty pipelines/Human
Pipeline config.downstream.txt
Kitty pipelines/EColi
Pipeline config.downstream.txt
Kitty pipelines/FruitFly
Pipeline config.downstream.txt
Kitty pipelines/Oyster
Pipeline config.downstream.txt
Plugin Cytoscape inputfile network.visualization.txt outputfile none
```

Now in the case above, the first two Pipelines are done on the Mouse data. We then switch the prefix to pipelines/Human and run downstream analysis on the Human data. Once this finishes, the prefix is back to pipelines/Mouse. We then repeat the process for E.Coli, Fruit Fly, and Oyster. Once all have finished, the prefix is back to pipelines/Mouse again. From which point we can run our visualization, which only should be done for the Mouse.

Particularly for large pipelines with multiple data sets, Kitties can save a great deal of code repetition in configuration files and reduce the likelihood of bugs significantly.

### 3.3 Online Tools

#### 3.3.1 PLUMA Plugin Pool

Existing PLUMA plugins developed by the PLUMA community can be downloaded from the PLUMA plugin pool at:

http://biorg.cs.fiu.edu/pluma/plugins

Figure 3.1 shows the current PLUMA plugin pool, which as of this release has 225 plugins. The plugin
Figure 3.1: PLUMA plugin pool. Available at http://biorg.cs.fiu.edu/pluma/plugins.

The pool specifies each plugin along with a link to its source code repository, the source language, and a short description of its functionality. Many of these plugins are also file format converters, which can be seamlessly integrated between two pipeline stages for input/output file compatibility. To assemble a PLUMA pipeline a user can install the plugins they need from the pool inside the PLUMA plugins directory or any directory within their PLUMA PLUGIN PATH. A user can also develop their own PLUMA plugins and run them alongside other plugins in the pool, which we describe next. If you develop a PLUMA plugin and would like to add your repository to the pool for other users in the community, please contact Trevor Cickovski at tcickovs@fiu.edu and provide a link to your repository and a description of your plugin.

### 3.3.2 PLUMA Pipeline Pool

We have also provided an empty pipelines/ folder in the PLUMA root directory, to install any PLUMA pipelines that have been made publicly available. For example, our sample pipeline that we reference in this userguide can be found at: https://github.com/movingpictures83/Mouse. This pipeline contains scripts that will automatically clone the appropriate PLUMA plugins into the plugins directory.

A pool of available pipelines is also available online, at:

http://biorg.cs.fiu.edu/pluma/pipelines

Figure 3.2 shows a screenshot of this site. Pipelines developed and tested by the PLUMA community are available under the Public table. Private pipelines are currently under development by the BioRG, and will be released in the future. If you successfully develop and test a PLUMA pipeline you may also contact Trevor Cickovski (tcickovs@fiu.edu) with a link to your repository and description of your pipeline to be added to the public table.
3.3.3 Alpha GUI

As an alternative to the text-based configuration file interface to PLUMA, one can opt for a Graphical User Interface (GUI). An Alpha version of a GUI for PLUMA is available here:

http://github.com/movingpictures83/PluMA-GUI

The GUI is built with Electron JS and obtains a list of available plugins, allowing a user to drag and drop them into pipelines and run PLUMA as shown in Figure 3.3.

Full documentation, including a user manual, installation guide, and poster/video descriptions of this GUI are available on the github site.
PluMA: Plugin-based Microbiome Analysis

### Installed Plugins

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Input:</th>
<th>Output:</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIOM2CSV</td>
<td>BIOM file</td>
<td>CSV file</td>
</tr>
<tr>
<td>CalcMeanStd</td>
<td>NOA (nodes and values)</td>
<td>none (mean and standard deviation printed to the screen)</td>
</tr>
<tr>
<td>ClusterCSV2NOA</td>
<td>CSV (clusters)</td>
<td>NOA (nodes and cluster ID)</td>
</tr>
<tr>
<td>CountTableProcessing</td>
<td>prefix (for abundances and OTUs produced by Mothur)</td>
<td>CSV (abundance matrix)</td>
</tr>
<tr>
<td>Tab2CSV</td>
<td>Tab-delimited CSV file</td>
<td>CSV file</td>
</tr>
</tbody>
</table>

### Add and Install more Plugins

![Diagram showing the flow of data through PluMA plugins](image)

**Figure 3.3:** PLUMA alpha GUI. Available at [http://github.com/movingpictures83/PluMA-GUI](http://github.com/movingpictures83/PluMA-GUI).
Chapter 4

Extending PLUMA

With plugin-based analysis the ultimate goal is infinite extensibility and flexibility, therefore we now illustrate this important property of PLUMA through some sample plugin extensions in various languages. We first show how to use our tool PluGen to automatically generate a plugin for an existing software package, followed by a discussion on how to build a plugin in each programming language supported by PLUMA, and finally an example of how to take components of an existing plugin and use them to build a new one. All plugins mentioned in this section are currently part of the PLUMA plugin pool.

4.1 Converting an Existing Tool to a Plugin

It is very possible for a user to have installed an existing software package that they would like to convert to a PLUMA plugin to run alongside other stages. A perfect example of this is our Mothur plugin from our above example, which references the metagenomics software package Mothur [27]. In this case we are using Mothur to take a sample of raw gut microbiome sequences from a mouse and produce a set of abundances for each microbial taxon in the sample. We now use PluGen to produce a PLUMA plugin for Mothur. All plugins generated by PluGen will be placed inside the default PLUMA plugins/ directory. Note this also automatically includes them in the scons compilation as well.

PluGen is also automatically compiled by scons. To run PluGen, first change to the PluGen/ subdirectory.

4.1.1 Option 1: Single Command (No Arguments)

If your software package does not accept command line arguments or its command line arguments are predefined somehow, you can generate a PLUMA plugin as follows:

```
./plugen plugin_name command_to_run
```

Where `plugin_name` will be the name of the new PLUMA plugin (be sure it is unique) and `command_to_run` is the command to run the software (with any predefined command line arguments). Although this was not applicable for our Mothur plugin since Mothur requires an input file, since its executable name is mothur we could have produced a plugin to run Mothur with no input parameters by running `./plugen Mothur mothur`. Most software tools, like Mothur, will require some type of input (and possibly output) parameters, so we now outline how to generate plugins with these possibilities.
4.1.2 Option 2: Input and Output File Arguments

If a plugin only requires an input and output file, then its structure can be mapped directly to that of a PLUMA plugin, where in the configuration file we provide an inputfile and outputfile. This option was applicable for our Mothur plugin, since Mothur accepts a command line parameter for an input file (it also outputs files, but those are specified in the input file and not on the command line). It would then make sense to allow the input file to be the same inputfile specified by a user in the PLUMA configuration file, and similar for outputfile. To do this, PluGen allows these same placeholders in its command line arguments, to represent user-specified configuration file values. For example, to generate our Mothur plugin we used the command:

```
./PluGen Mothur mothur inputfile
```

Now in our configuration file, we specified:

```
Plugin Mothur inputfile input.mothur outputfile none
```

Mothur would then be executed by our new plugin using the command mothur input.mothur. If we later ran the same plugin but with a different value for inputfile in the configuration file, that value would then be substituted as the argument to mothur.

4.1.3 Option 3: Fully Customized (Multiple Arguments)

Other software tools may accept multiple input parameters. For example, a common method for running Cytoscape [10] is to provide on the command line a network file to visualize, and a Cytoscape (*.cys) file for visualization properties. For our pipeline, we would have executed on the command line:

```
./cytoscape -N network.gml -s myproperties.cys
```

Since there are now multiple configuration file parameters, PluGen will generate a plugin that accepts a single input file with multiple keyword-value pairs. Therefore in our configuration file above, we provide:

```
Plugin Cytoscape inputfile network.visualization.txt outputfile none
```

In network.visualization.txt, we then provide as its contents:

```
networkfile network.gml
sessionfile myproperties.cys
```

When running PluGen we use the command line as a template. The keywords in the single input file then become placeholders for their corresponding values. Our command to generate the Cytoscape plugin was:

```
./plugen Cytoscape cytoscape -N networkfile -s sessionfile
```
As before, if we were to run the Cytoscape plugin with a different input file and provide different values for networkfile and sessionfile, those values would automatically be substituted into the command line parameters above. Note we also provide flags like -N and -s in addition to the name of the cytoscape executable, as arguments to PluGen.

4.2 Building a New Plugin From Scratch

We now describe how to build a new plugin using one of PLUMA’s supported languages. These languages have a variety of syntax and semantics, some are object-oriented and some are not, some are compiled and others scripted, some run on the CPU and others on the GPU. As one can imagine, the process will vary with the language. However, from the perspective of running PLUMA nothing will be different. Independent of its source language, each plugin uses the same configuration file specification outlined above. The one requirement of all PLUMA plugins is three procedures (these can be empty if desirable):

1. An input() procedure that accepts one parameter (an input file). Generally this will reads that file and initialize the plugin.

2. A run() procedure that accepts zero parameters, and runs the plugin.

3. An output() procedure that accepts one parameter (an output file). Generally this will finalize the plugin and write that file.

We now review PLUMA’s supported languages, and show through examples how to write plugins in each language. Some of these plugins were in our example configuration file from the previous session, and all are currently available in the PLUMA plugin pool.

4.2.1 C++

For C++ we show the ATria plugin, which accepts a signed and weighted network in CSV format, computes node centrality (importance) using the Ablatio Triadum algorithm (ATria, [7]), and produces a list of the most central nodes in NOde Attribute (.noa) format, which Cytoscape can use to color nodes based on centrality. New plugins should be constructed in the PLUMA plugins folder and another directory within your PLUMA_PLUGIN_PATH. A subdirectory will then uniquely identify the plugin, in which C++ source files will use that same name followed by Plugin. For example, assuming we used the PLUMA plugins folder our directory tree structure looked like this:

```
plugins/ATria/ATriaPlugin.h
plugins/ATria/ATriaPlugin.cpp
```

The file name should also be the same name as a new C++ class that they define. This class will then inherit from a parent class Plugin, defined with PLUMAin the file Plugin.h, which should be included as shown in Program 1, which defines the header file ATriaPlugin.h. Note all three required methods are present, and input and output each accept a single parameter of type string for the input and output files, respectively. Therefore the Standard Template Library [20] string class should also be included. All other included headers, member procedures and variables are at the user’s discretion.

We now show a template for the corresponding source file ATriaPlugin.cpp in Program 2. This source file will include definitions for the three required procedures, along with any other procedures defined
Program 1 Plugin header file ATria/ATriaPlugin.h.

```cpp
#ifdef ATRIAPLUGIN_H
#define ATRIAPLUGIN_H

#include "Plugin.h"
#include <string>
// Other necessary includes...

class ATriaPlugin : public Plugin
{
    public:
    // These are required
    void input(std::string file);
    void run();
    void output(std::string file);

    // Other member procedures...

    private:
    float* OrigGraph;
    std::string* bacteria;
    // Other member variables...
};

#endif
```

by the user. As is typical, the corresponding header ATriaPlugin.h should be included, along with two PLUMA headers: PluginManager.h and PluginProxy.h. The proxy [3] and manager [9] work together to interface this new plugin to the PLUMA computational core. Note when assembling the proxy, we also specify the name (ATria) which PLUMA uses to reference this plugin in the configuration file. Upon this reference, code for this plugin will be dynamically loaded at runtime, keeping the software lightweight. Note that the plugin class ATriaPlugin should also be passed into the proxy template as shown.

### 4.2.2 CUDA

In addition to C++, CUDA can be incorporated into a plugin to take advantage of GPU parallelism, assuming you have an NVIDIA graphics card installed. We will demonstrate CUDA functionality by taking the C++ ATria plugin from the previous section and converting it to a version GPUATria which runs on the GPU. We have previously used this plugin to increase the speed of ATria by an order of magnitude [7] for large networks, allowing us to analyze a 3000-node fruit fly network in a few hours as opposed to a few days.

Since CUDA is an extension of C and the NVIDIA compiler (NVCC) now accepts C++ code, building a CUDA plugin will work similarly to building a C++ plugin. The source file must now end in .cu as opposed to .cpp so that scons will compile the code with nvcc. CUDA kernel functions can then be added standalone as shown in Program 3.

CUDA uses the `__global__` keyword to preface kernel declarations. These can in turn be invoked from the C++ class functions in the source file, as shown in Program 4. For our particular implementation we used kernels for the all-pairs shortest path algorithm [12], [22] and a final pass to compute centrality.

CUDA kernels will most likely be invoked from the `run` procedure, though not necessarily. The amount
Program 2 Plugin source file ATRia/ATriaPlugin.cpp.

```cpp
#include "ATriaPlugin.h"
#include "PluginManager.h"
#include "PluginProxy.h"
// Other necessary includes...

void ATRiaPlugin::input(std::string file) {
    // Read file, and initialize member variables...
}

void ATRiaPlugin::run() {
    // Run the algorithm...
}

void ATRiaPlugin::output(std::string file) {
    // Perform any final operations, and write file...
}

// Other member procedure definitions...

// Required, connects the plugin to the PluMA back end
PluginProxy<ATriaPlugin> ATRiaPluginProxy
    = PluginProxy<ATriaPlugin>("ATria", PluginManager::getInstance());
```

Program 3 GPUATria/GPUATriaPlugin.h

```cpp
#ifndef GPUATRIAPLUGIN_H
#define GPUATRIAPLUGIN_H

// Same includes as C++...

class GPUATriaPlugin : public Plugin {
public:
    // These are still required
    void input(std::string file);
    void run();
    void output(std::string file);

    // Other member procedures...

private:
    // Member variables...
};

__global__ void GPU_Floyd_kernel(int k, float *G, int N);
__global__ void GPU_Pay_kernel(float *D, float *P, int N);
// Other GPU kernels...

#endif
```
Program 4  GPUATria/GPUATriaPlugin.cu

// Same includes as C++...

void GPUATriaPlugin::input(std::string file) {
    // Read file, and initialize member variables...
}

void GPUATriaPlugin::run() {
    // ...
    // First kernel
    _GPU_Floyd_kernel<<<dimGrid,BLOCK_SIZE>>>(k,dG,N);
    // ...
    // Second kernel
    _GPU_Pay_kernel<<<numblocks,BLOCK_SIZE>>>(dG,dPay,(N/2));
    // ...
}

void GPUATriaPlugin::output(std::string file) {
    // Perform any final operations, and write file...
}

// Other member procedure definitions...

__global__ void _GPU_Pay_kernel(float* D, float* P, int N) {
    // GPU code...
}

__global__ void _GPU_Floyd_kernel(int k, float*G, int N) {
    // GPU code...
}

// Other kernel procedure definitions...

// Still required...
PluginProxy<GPUATriaPlugin> GPUATriaPluginProxy
    = PluginProxy<GPUATriaPlugin>("GPUATria", PluginManager::getInstance());
of CUDA cores and threads you allocate for each kernel (specified within the <<< and >>>) can be critical to its efficiency [17]. For more information, please view the CUDA Programmer’s Guide [21].

4.2.3 Python

The remaining PLuMA-supported languages are all scripted. The first major difference when constructing a plugin in one of these languages is that the entire plugin should be encapsulated in one file. For example, in this section we will build a template for a Python plugin PageRank that runs Google’s PageRank [23] centrality algorithm. The filename should follow similar conventions to the compiled plugins; using the plugin name followed by Plugin. We thus name our file PageRankPlugin.py. Note that because these plugins are scripted, they are not detected in the scons PLuMA compilation. However, they still should be installed in either plugins/ or a location in the PLuMA_PLUGIN_PATH to be recognized at execution time. In addition, we will not need to include a proxy to interface properly to the PLuMA back end, unlike C++ and CUDA.

We show our plugin template for PageRank in Program 5. Our particular implementation imports various Python libraries [15, 19, 28] to aid in the computation. Since Python is object-oriented we once again will build the plugin as a class and as with C++ and CUDA, the classname (in our case, PageRankPlugin) should be the same as the filename. We then declare the three required procedures as member functions of this Python class, with input and output accepting their file parameter. Since Python is dynamically typed, as is the case for most scripting languages, no specification is required for the types of these parameters.

Program 5 PageRank/PageRankPlugin.py

```python
import numpy
import networkx
from pythonds.graphs import PriorityQueue, Graph, Vertex
# Other imports ...

numdiff = 0
ALPHA=0.5
def buildNetworkXGraph(filename):
    # Read filename, build and return graph

# Other global variables/procedures ...

class PageRankPlugin:
    def input(self, file):
        self.bacteria, self.graph = buildNetworkXGraph(file)
    def run(self):
        self.U = networkx.pagerank(self.graph, alpha=ALPHA, max_iter=100)
    def output(self, file):
        UG = []
        for key in self.U:
            UG.append((self.U[key], key))
        UG.sort()
        UG.reverse()

        # Write file...
```
4.2.4 R

Since the remaining PLUMA-supported language, R and Perl, are not by default object-oriented (although there are importable packages that facilitate object-oriented programming), PLUMA does not use classes for plugins in either of these languages but instead assumes that the three required procedures will be implemented standalone. Program 6 shows a plugin Spearman that we developed for calculating Spearman correlations, following similar conventions with regard to the filename.

Program 6 Spearman/SpearmanPlugin.R

```r
p_value <- 0.01;
libs <- c('Hmisc');
lapply(libs, require, character.only=T);
# Other global variables and libraries...

# Required
input <- function(inputfile) {
  pc <- read.csv(inputfile, header = TRUE);
}

# Required
run <- function() {
  # Some preprocessing...
  correlations <- rcorr(pc[,], type=c("spearman"));
  pc <- as.matrix(correlations$r);
  # Post-processing and p-value thresholding...
}

# Required
output <- function(outputfile) {
  write.table(pc, file=outputfile, sep=' ', append=FALSE,
              row.names=unlist(cn), col.names=unlist(cn), na='');
}

# Other helper procedures...
```

We declare any global variables and import any necessary libraries at the top of the file before the three procedure definitions. This particular plugin takes input and output in CSV format and computes correlations using the `rcorr` method from the R Hmisc package following some preprocessing. We perform some final p-value thresholding before output.

4.2.5 Perl

For our Perl plugin we show a simple example of a plugin CytoViz in Program 7 that calls Cytoscape and visualizes a network represented in the Graph Modeling Language (GML), assuming the path to Cytoscape is defined in the environment variable CYTOSCAPE_HOME. Following naming conventions, we implement this plugin in a file CytoVizPlugin.pl and include all three required procedures. Note Perl procedure parameters are implicitly stored in the array @_, so there is no need to specify parameters for input and output. Perl also requires global variables to be preceded with the my keyword, which we use for the network file. In this case our plugin will either invoke Cytoscape and visualize the network using its -N flag, or produce an error.
Program 7 CytoViz/CytoVizPlugin.pl

my $gmlfile;
# Any other global variables...

# Required
sub input {
    $gmlfile = @[0];
    return;
}

# Required
sub run {
    $cytohome = $ENV{'CYTOSCAPE_HOME'};
    length($cytohome) != 0 or die "Please set CYTOSCAPE_HOME
";
    @args = ($cytohome . '/cytoscape.sh', '-N', $gmlfile);
    system(@args) == 0 or die "system @args failed: $?\n"
    return;
}

# Required
sub output {
    return;
}

4.3 Building a New Plugin using Existing Plugins and Computational Core

PLUMA plugins can also use functionality from other plugins, which can be very useful. For example, Google’s PageRank can be artificially biased [29] to favor nodes with certain properties. PageRank internally involves a random walker that constantly moves between neighboring network nodes and determines centrality based on the amount of times they land on a node. We could thus tailor our centrality values to leader nodes of tightly connected network components by biasing PageRank to make the walker more likely to visit nodes in the same cluster (enabling cluster leaders to get hit the most). This does assume that we have already run a clustering algorithm of some kind on our network, and there are a few currently available as PLUMA plugins.

We now design our BiasedPageRank plugin and import the PageRank plugin as a Python module, as shown in Program 8. For this to work properly, the directories plugins and plugins/PageRank must be available as importable Python packages. This can be done by inserting an empty __init__.py file in each of these folders (the default PLUMA plugins/ directory includes one by default). We recommend any user designing Python plugins to include this file in their specific installation directory, since this increases potential for reusing or building upon their code to produce new and creative ideas, fundamental to the purpose of PLUMA.

BiasedPageRankPlugin can then inherit from PageRankPlugin, as shown in the class header. We do this because the source code of our new plugin is almost exactly the same as PageRankPlugin, except for one small change. PageRank uses a variable alpha to represent the likelihood of the random walker advancing to a particular neighbor node; in this case we use two different values alpha1 (larger) and alpha2 (smaller), depending on whether or not the candidate neighbor is in the same cluster.

To obtain this knowledge, we also borrow functionality from a plugin Clusterize, which contains functions to both read a cluster file (assumed to be in CSV format mapping node names to cluster identifiers), and to determine given this data if two parameter nodes are in the same cluster. As that is also a Python plugin, we import it in a similar fashion. Our input method thus changes to read two CSV files (one for
the network, and one for the clusters), but since all other procedures are the same as PageRank we do not need to redefine them.

Finally, the PyPluMA module is a SWIG-wrapped interface to the PluMA computational core, providing the ability to call C++ functionality from the PluMA PluginManager class from scripted interfaces (R, Python or Perl). In this case, we use the PyPluMA log() function to record the number of clusters in the network as a part of the current PluMA log. Another useful function in this interface is prefix(), which returns the current Prefix or Kitty as a string.

Program 8 BiasedPageRank/BiasedPageRankPlugin.py

```python
import PyPluMA
import numpy
import networkx as nx
import plugins.Clusterize.ClusterizePlugin
import plugins.PageRank.PageRankPlugin

def biasedpagerank(G, clusters, alpha1=0.5, alpha2=0.35, max_iter=100,
                   tol=1.0e-8, nstart=None):
    # ...
    # Use appropriate alpha value
    for nbr in W[n]:
        if (ClusterizePlugin.inSameCluster(n, nbr, clusters)):
            x[nbr]+=alpha1*xlast[n]+W[n][nbr][\'weight\']
        else:
            x[nbr]+=alpha2*xlast[n]+W[n][nbr][\'weight\']
    # ...

class BiasedPageRankPlugin(PageRankPlugin.PageRankPlugin):
    def input(self, filename):
        PageRankPlugin.input(self, filename+'\,.csv')
        self.clusters = ClusterizePlugin.readClusterFile(filename+'\,.clusters.csv')
        PyPluMA.log("Running\_BiasedPageRank\_with\_\"+str(len(self.clusters))+\"\_clusters.\")
    def run(self):
        self.U = biasedpagerank(self.graph, self.clusters, alpha1=0.5, alpha2=0.35,
                                 max_iter=100)
```

4.4 Testing New Plugins

Before a plugin is added to the plugin pool, it should be thoroughly tested. PluMA now comes equipped with a Python script in its root directory testPluMA.py, which can be run in this way to test all installed plugins:

```
python testPluMA.py
```

Or a specific plugin (i.e. Mothur):

27
We ask all developers who are interested in registering their plugin with the PLUMA plugin pool to first verify that their plugin is compatible with this test script, and generates a result of **PASS**.

Compatibility with this script is a simple process. Suppose a user has developed a new plugin X. The new plugin folder (X) should contain:

1. The plugin source code, named as XPlugin.ext, where ext is the appropriate extension for their language of choice (py, pl, R, cpp or cu).

2. An example/ directory, containing:
   - The parent configuration file, named config.txt.
   - Any other input files necessary to run the configuration file.
   - Expected output files, named appropriately concatenated with the suffix .expected (for example, if we were testing CountTableProcessing in the Mouse pipeline above, we would provide a file abund.csv.expected).
   - A README.md file, beginning with the following lines:
     ```
     # (plugin name)
     # Language: (programming language used)
     # Input: (expected input file format, i.e. TXT or CSV)
     # Output: (expected output file format, or none if screen)
     # Tested with: (version of PluMA (i.e. PluMA 1.1)), version of language tools (i.e. gcc 4.8.4)
     # Dependency: (if any, i.e. Mothur or Cytoscape)
     ```

The testPluMA.py script will then run PluMA on example/config.txt, and produce output files. It will look for any output files for which the developer provided a file with the suffix .expected, but will chop off .expected. It then will compare these with the corresponding .expected file for accuracy. If any of these are not produced or do not exactly match the expected output, the script will error.

There are some exceptions to this rule:

- **Screen output.** Some plugins only produce screen output. If that is the case, the developer should provide a file screen.expected in the example/ folder, which contains the exact expected screen output produced by PluMA for their configuration file. This will then be compared to the screen output generated by PLUMA on their test.

- **Interactive plugins.** Other plugins are user-interactive (i.e. Cytoscape), requiring the user to point and click on some items. These are not currently compatible with the test script, and so we allow an empty interactive file in the example directory to signify this. *This should be the exception, not the rule.* If at all possible, it is better to at least generate a text file with some data that can be used to signify proper functionality.

- **Large input files.** Some plugins require large input files, particularly if they are processing sequence data. Github has limits regarding the size of files that can be committed, requiring in some cases to commit compressed (i.e. tar or gz) versions of these files. These then need to be extracted before the plugin can be tested. If this is the case, the developer can supply a file pretest.txt with any commands that should be run before config.txt.
• **Outputs that vary slightly each time.** This can happen in the case where a tool outputs a file containing the date, or execution time (which will change every time the program runs). In this case, the developer can provide a `test.py` file in `examples/`. This file should contain a function `test(inputfile, outputfile)` which returns `true` (for PASS) or `false` (for FAIL). If `testPluMA.py` sees this file, instead of comparing expected outputs to actual it will call `test` and pass actual output as the first and expected output as the second. The developer could then write a Python function that ignores the date/execution time when checking for equivalence.
Chapter 5

PLUMA Full Pipeline

We now return to our earlier example of a complete metagenomics pipeline that we used to illustrate the PLUMA configuration file:

```
# Metagenomics analysis pipeline
Prefix pipelines/Mouse
Plugin Mothur inputfile input.mothur outputfile none
Plugin CountTableProcessing inputfile mouse.trim.abund.pick.an.unique_list outputfile abund.csv
Plugin CSVNormalize inputfile abund.csv outputfile abund.norm.csv
Plugin Spearman inputfile abund.norm.csv outputfile spearman.csv
Plugin CSVPad inputfile spearman.csv outputfile network.csv
Plugin GPUATria inputfile network.csv outputfile ATria.noa
# Plugin ATRia inputfile network.csv outputfile ATria.noa
Plugin CSV2GML inputfile network.csv outputfile network.gml
Plugin Cytoscape inputfile network.visualization.txt outputfile none
```

Figure 5.1 provides a conceptual description of this pipeline, which consists of plugins in all PLUMA-supported languages except Perl, although CytoViz could easily be substituted for Cytoscape above to execute our Perl example. There are also two plugins (Motur and Cytoscape) that were generated by PluGen. Some stages are simple file converters and others are more involved, but the setup is uniform in PLUMA configuration file.

We now outline the details of each pipeline stage.

5.1 Stage 1: Mothur

For the first stage, we provide a file input.mothur to our Mothur plugin. This file executes an example from the Mothur Mealybugs tutorial [16] and uses Illumina mouse genome data from MiSeq [25]. This tutorial removes noise from the reads including chimeric sequences before performing similarity-based clustering [30] and finally mapping each cluster to the closest Operational Taxonomic Unit (OTU) through lookups in the SILVA [24] database. While Mothur produces a series of output files for our pipeline we are most interested in OTUs and their abundances, which are provided in *.shared (Program 9) and *.cons.taxonomy (Program 10) files with the same prefix. Both will be necessary to run our next stage.
Figure 5.1: Conceptual description of a full metagenomics pipeline in PLUMA, with plugins in a variety of supported languages and two generated by PluGen.

**Program 9** OTU abundances produced by Mothur in a (.shared file).

<table>
<thead>
<tr>
<th>bel</th>
<th>Group</th>
<th>numOtus</th>
<th>Otu001</th>
<th>Otu002</th>
<th>Otu003</th>
<th>Otu004</th>
<th>Otu005</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.03</td>
<td>F3D0</td>
<td>528</td>
<td>500</td>
<td>306</td>
<td>394</td>
<td>408</td>
<td>636</td>
<td>...</td>
</tr>
<tr>
<td>0.03</td>
<td>F3D1</td>
<td>528</td>
<td>352</td>
<td>311</td>
<td>189</td>
<td>64</td>
<td>73</td>
<td>...</td>
</tr>
<tr>
<td>0.03</td>
<td>F3D141</td>
<td>528</td>
<td>388</td>
<td>335</td>
<td>300</td>
<td>482</td>
<td>427</td>
<td>...</td>
</tr>
</tbody>
</table>

**Program 10** OTUs classifications produced by Mothur in a (.cons.taxonomy file).

<table>
<thead>
<tr>
<th>OTU</th>
<th>Size</th>
<th>Taxonomy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Otu001</td>
<td>12307</td>
<td>Bacteria (100); ‘‘ Bacteroidetes ’’ (100); ‘‘ Bacteroidia ...</td>
</tr>
<tr>
<td>Otu002</td>
<td>8904</td>
<td>Bacteria (100); ‘‘ Bacteroidetes ’’ (100); ‘‘ Bacteroidia ...</td>
</tr>
<tr>
<td>Otu003</td>
<td>7799</td>
<td>Bacteria (100); ‘‘ Bacteroidetes ’’ (100); ‘‘ Bacteroidia ...</td>
</tr>
<tr>
<td>Otu004</td>
<td>7485</td>
<td>Bacteria (100); ‘‘ Bacteroidetes ’’ (100); ‘‘ Bacteroidia ...</td>
</tr>
<tr>
<td>Otu005</td>
<td>7459</td>
<td>Bacteria (100); ‘‘ Bacteroidetes ’’ (100); ‘‘ Bacteroidia ...</td>
</tr>
<tr>
<td>Otu006</td>
<td>6633</td>
<td>Bacteria (100); ‘‘ Bacteroidetes ’’ (100); ‘‘ Bacteroidia ...</td>
</tr>
<tr>
<td>Otu007</td>
<td>6307</td>
<td>Bacteria (100); ‘‘ Bacteroidetes ’’ (100); ‘‘ Bacteroidia ...</td>
</tr>
</tbody>
</table>

...
5.2 Stage 2: CountTableProcessing

Now that we have the OTU abundances, we are ready to perform downstream analysis in our pipeline. However, many of our latter stages use the Comma-Separated-Value (CSV) format for data. Therefore, we include a file conversion plugin as the second stage of our pipeline. This R plugin, CountTableProcessing, will take the above two files from Mothur and produce an abundance matrix in CSV format. This file will include samples as rows and OTUs as columns, with entry \([i, j]\) corresponding to the abundance of OTU \(j\) in sample \(i\). Since CountTableProcessing needs both Mothur output files, we provide their common prefix as input and allow the plugin to attach their respective extensions. Program 11 shows the output of this plugin.

Program 11 OTU abundance matrix produced by Stage 2 of our pipeline.

```
' ' ' ' Family . Porphyromonadaceae . 0001 ', ' ' Family . Porphyromonadaceae ...
' ' F3D0 ', 500.306.394.408.636.356.168.164.135.19.27.110.52.104.87.78...
' ' F3D1 ', .352.311.189.64.73.117.127.174.83.111.53.35.115.253.288.0.7...
' ' F3D141 ', 388.335.300.482.427.279.205.327.120.175.112.101.8.15.28.95...
' ' F3D142 ', 243.258.142.152.253.192.201.81.72.43.51.82.96.6.8.45.24.5...
' ' F3D143 ', 189.152.172.206.309.183.116.92.63.76.69.43.37.2.7.36.26.0...
' ' F3D144 ', 347.243.260.316.469.282.132.36.140.255.96.84.13.7.10.100.38...
...
```

5.3 Stage 3: CSVNormalize

For our third stage, we use a Python plugin to normalize this abundance matrix, so that each sample’s OTU counts sum to 1. Normalizing guards against artifacts produced by variants such as sample quality. Measuring each OTU as a percentage of the total OTUs in the sample creates a uniform metric across all samples. Program 12 shows the normalized abundances produced by CSVNormalize.

Program 12 Normalized abundance matrix produced by Stage 3 of our pipeline.

```
' ' ' ' Family . Porphyromonadaceae . 0001 ', ' ' Family . Porphyromonadaceae ...
' ' F3D0 ', .003987779386.0.0492040520984.0.0633542370156.0.06560540...
' ' F3D1 ', .0075455198285.0.0666666666667.0.0405144694534.0.01371918...
' ' F3D141 ', .0083333333333.0.0719501718213.0.0644329896907.0.103522...
' ' F3D142 ', .0.10028889806.0.10647957078.0.0586050350805.0.0627321502...
' ' F3D143 ', .0.0788485607009.0.0634125990822.0.0717563621193.0.085940...
' ' F3D144 ', .0.100696459663.0.070516540917.0.0754497968659.0.09170052...
...
```

5.4 Stage 4: Spearman

Normalized abundances enable us to produce a correlation matrix where OTUs occupy both rows and columns, and entry \((i, j)\) is the correlation value between the abundances of OTU \(i\) and OTU \(j\) over all samples. This can also be represented as a correlation network [], where OTUs are network nodes and edges are weighted with their correlation values (a correlation of zero would mean no edge in the network). Here
we use our same Spearman plugin that we used to demonstrate how to construct an R plugin for PLUMA, producing the output shown in Program 13 which equates to the network in Figure 5.2. Here we use green edges to represent positive correlations and red edges for negative correlations, with strength indicated by edge thickness.

**Program 13** OTU correlation matrix produced by Stage 4 of the pipeline.

```
'Family. Porphyromonadaceae.0001', 'Family. Porphyromonadaceae.0002', ...
'Family. Porphyromonadaceae.0001', 1.0, 0.743859648704529, 0.0, 0.0, 0.0, 0.0...
'Family. Porphyromonadaceae.0002', 0.743859648704529, 1.0, 0.0, 0.0, 0.0, 0.0...
'Family. Porphyromonadaceae.0003', 0.0, 0.1, 1.0, 0.0, 0.0, 0.0, 0.752631604671478...
'Family. Porphyromonadaceae.0004', 0.0, 0.0, 0.1, 0.624561429023743, 0.766666...
'Family. Porphyromonadaceae.0005', 0.0, 0.0, 0.624561429023743, 1.0, 0.81052...
'Family. Porphyromonadaceae.0006', 0.0, 0.0, 0.766666650772095, 0.8105263...
...```

### 5.5 Stage 5: CSVPad

Note that the `write.table` function of R produces a CSV file with no initial empty string at the upper left corner, meaning that the columns do not line up exactly with their headers. We constructed our plugin for Stage 6 assuming this would be present. Rather than modify our Stage 4 plugin for compatibility which would oppose the goals of PLUMA, we insert a simple file conversion plugin CSVPad that takes care of this without modifying Stage 4 or Stage 6, producing the output in Program 14.

### 5.6 Stage 6: GPUATria

Our network from Figure 5.2 is sometimes referred to as a *microbial social network* [1], since correlations (or edges) represent how often two OTUs tend to appear together (positive) or apart (negative). This opens the door to perform downstream analysis on such networks using social networking algorithms such as centrality [11] which finds important nodes. *Ablatio Triadum* (ATria, [7]) is one such algorithm that finds important nodes in signed and weighted networks, which we used for our C++ and CUDA examples earlier.
Program 14 Padded OTU correlation matrix produced by Stage 5 of the pipeline.

```
' ' ' ' ' 'Family . Porphyromonadaceae . 0001' ', ' 'Family . Porphyromonadaceae . 0002' ', ...
' ' ' ' ' 'Family . Porphyromonadaceae . 0001' ', 1.0 .743859648704529 .0 .0 .0 .0 .0 .0 ... 0 .0 .0 .0 .0 .0 .0 ...
' ' ' ' ' 'Family . Porphyromonadaceae . 0002' ', 0 .743859648704529 .1 .0 .0 .0 .0 .0 .0 ... 0 .0 .0 .0 .0 .0 .0 ...
' ' ' ' ' 'Family . Porphyromonadaceae . 0003' ', 0 .0 .1 .0 .0 .0 .0 .0 .752631604671474 ... 0 .0 .0 .0 .0 .0 .0 ...
' ' ' ' ' 'Family . Porphyromonadaceae . 0004' ', 0 .0 .0 .1 .0 .624561429023743 .0 .76666 ... 0 .0 .0 .0 .0 .0 .0 ...
' ' ' ' ' 'Family . Porphyromonadaceae . 0005' ', 0 .0 .0 .0 .0 .624561429023743 .1 .0 .81052 ... 0 .0 .0 .0 .0 .0 .0 ...
' ' ' ' ' 'Family . Porphyromonadaceae . 0006' ', 0 .0 .0 .0 .7666666650772095 .0 .8105263 ...
...
```

We use the faster CUDA version in this pipeline as stage 6, and show its output in Program 15. The output takes the form of a NOde Attribute (NOA) file, which we will provide to Cytoscape for visualizing nodes differently based on centrality value or rank.

Program 15 NOA file of nodes, centrality values, and ranks produced by Stage 6.

<table>
<thead>
<tr>
<th>Name</th>
<th>Centrality</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phylum . Bacteroidetes . 0001</td>
<td>116.068</td>
<td>536</td>
</tr>
<tr>
<td>Kingdom . Bacteria . 0012</td>
<td>86.5733</td>
<td>535</td>
</tr>
<tr>
<td>Class . Clostridia . 0001</td>
<td>72.5422</td>
<td>534</td>
</tr>
<tr>
<td>Escherichia . Shigella . 0001</td>
<td>67.3395</td>
<td>533</td>
</tr>
<tr>
<td>Family . Enterobacteriaceae . 0002</td>
<td>57.8472</td>
<td>532</td>
</tr>
<tr>
<td>Family . Porphyromonadaceae . 0057</td>
<td>52.0528</td>
<td>531</td>
</tr>
</tbody>
</table>

5.7 Stage 7: CSV2GML

Cytoscape will also need our network, but cannot visualize networks in CSV format. We thus include one final stage before visualization CSV2GML that accepts a CSV file and converts it to the Graph Modeling Language (GML) which is recognizable by Cytoscape. Program 16 shows the equivalent GML file to our CSV network in Program 14, which we will provide to Cytoscape along with the NOA output from Program 15.

5.8 Stage 8: Cytoscape

For our final stage we will use our plugin for Cytoscape generated by PluGen. Recall that this plugin accepts as input a plaintext file (we called this network.visualization.txt that includes both a properties file (.cys) and a network file (.gml). We provide our output GML file from Program 16 as the latter, and include our own properties file that colors nodes based on an attribute "Centrality". Figure 5.3 shows Cytoscape initially opening when our plugin executes, although no nodes are colored yet because we must input our NOA file from Program 15 manually through the Cytoscape menu bars. Once we input this file our node colors change based on centrality value. Figure 5.4 shows this network, with red (violet) indicating high (low) centrality. Zero centrality is indicated by white nodes.
Program 16 Our correlation network from Program 14 in GML format, produced by Stage 7 of our pipeline.

```
graph [ 
  node [ 
    id 0
    label "Family: Porphyromonadaceae.0001"
  ] 
  node [ 
    id 1
    label "Family: Porphyromonadaceae.0002"
  ] 
  edge [ 
    source 0
    target 1
    weight 0.743859648705
  ] 
  edge [ 
    source 0
    target 27
    weight -0.601754367352
  ] 
  ...
```

Figure 5.3: Our network visualized with Cytoscape upon executing Stage 8 of our pipeline.
Figure 5.4: Our Cytoscape visualization after importing our NOA file from Stage 6.
Chapter 6

The Future of PLUMA

As mentioned, our vision with PLUMA is to allow users to develop and new and creative analysis algorithms as plugins in their language of choice, and run them seamlessly alongside other plugins within a lightweight and flexible software framework. With 225 fully tested, functional plugins in an array of various languages, PLUMA 1 provides a solid foundation for assembling a wide range of software pipelines.

While PLUMA 1 is primarily a command line tool, PLUMA 2.0 will be emphasizing usability. The alpha version of the GUI will continue to be developed, with a focus on supporting more interactive PLUMA executions as opposed to just generating configuration files. Additionally, it must be extended to support the newer Pipeline and Kitty features of PLUMA that were recently added in version 1.1.

Also at the moment PLUMA has only been verified on Linux and Mac systems. We are currently developing a Windows version of the software, which we expect to be release in version 2.0. Additionally we would like to smooth out the installation and execution of plugins, most of which have some external dependency (i.e. a Python or R library). We are exploring options such as Homebrew and Docker [18], though will keep lightweightness in mind as much as possible while pursuing such solutions. Finally, we are working to develop cloud-based interfaces to the plugin pool and pipeline executions. Particularly the latter will be beneficial to users who simply want to assemble pipelines with existing plugins, since it will provide an environment with all necessary dependencies for the PLUMA plugin pool installed.
Appendix A

PLUMA Software License

A.1 Conditions and Regulations

Copyright 2017 Florida International University

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

A.2 Contact Information

The best contact path for licensing issues is by e-mail to lead developer Trevor Cickovski at tcickovs@fiu.edu or send correspondence to:

PLUMA Team
C/o Prof. Giri Narasimhan
Bioinformatics Research Group (BioRG)
School of Engineering and Computer Science
Florida International University
11200 SW 8th Street Miami, FL 33199 USA
Bibliography


