Develop a Galaxy-based Graphic User Interface for QIIME

Author: Jose Antonio Navas Molina
External supervisor: Rob Knight
Internal supervisor: Beatriz Otero Calviño
Index

Index.................................................................................................................................1

1. Introduction ....................................................................................................................3
   1.1. Motivation ................................................................................................................3
   1.2. Objectives ................................................................................................................4

2. Problem statement ........................................................................................................5
   2.1. Introduction to QIIME ............................................................................................5
   2.2. Analysis of QIIME ..................................................................................................14
   2.3. GUI characteristics .................................................................................................14

3. Previous work ..............................................................................................................16
   3.1. QIIME database .....................................................................................................16
   3.2. QIIME integration on Galaxy ................................................................................16

4. Proposed solution .........................................................................................................18
   4.1. What is Galaxy? .......................................................................................................18
   4.2. Why Galaxy? ...........................................................................................................22
       4.2.1. Generic GUI requirements ..............................................................................22
       4.2.2. QIIME-specific GUI requirements .................................................................23
   4.3. Galaxy: disadvantages ............................................................................................26
   4.4. Case study: FastUniFrac ........................................................................................29
       4.4.1. FastUniFrac functionalities .............................................................................29
       4.4.2. Developed software .........................................................................................40
       4.4.3. FastUniFrac integration ...............................................................................48
       4.4.4. Deploy package ..............................................................................................61
   4.5. QIIME automated integration ................................................................................61
       4.5.1. QIIME modifications .......................................................................................62
       4.5.2. Design ..............................................................................................................63
       4.5.3. Implementation ...............................................................................................64
       4.5.4. Testing .............................................................................................................69

5. Galaxy on the Cloud: Galaxy CloudMan .......................................................................72
   5.1. QIIME + Galaxy CloudMan AMI .............................................................................73

6. Conclusions ...................................................................................................................83
   6.1. Future work .............................................................................................................83

7. Financial report ...........................................................................................................84
   7.1. Planning ...................................................................................................................84
   7.2. Budget .....................................................................................................................85

Annex A – FastUniFrac pseudo-code ..............................................................................86
<table>
<thead>
<tr>
<th>Libraries</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>make_heatmap.py</td>
<td>86</td>
</tr>
<tr>
<td>make_html_heatmap.py</td>
<td>86</td>
</tr>
<tr>
<td>newick_to_asciiArt.py</td>
<td>87</td>
</tr>
<tr>
<td>parse_beta_significance_output.py</td>
<td>88</td>
</tr>
<tr>
<td>parse_jackknife_support.py</td>
<td>92</td>
</tr>
<tr>
<td>Scripts</td>
<td>Pages</td>
</tr>
<tr>
<td>category_map_to_id_map_parser.py</td>
<td>93</td>
</tr>
<tr>
<td>sample_id_map_to_otu_table_parser.py</td>
<td>93</td>
</tr>
<tr>
<td>copy_data.py</td>
<td>96</td>
</tr>
<tr>
<td>modify_qiime_parameters.py</td>
<td>96</td>
</tr>
<tr>
<td>make_cluster_samples_file.py</td>
<td>97</td>
</tr>
<tr>
<td>make_pcoa_html.py</td>
<td>97</td>
</tr>
<tr>
<td>make_beta_significance_heatmap.py</td>
<td>99</td>
</tr>
<tr>
<td>make_unifrac_significance_each_sample.py</td>
<td>100</td>
</tr>
<tr>
<td>make_distance_matrix_heatmap.py</td>
<td>101</td>
</tr>
<tr>
<td>make_jackknife_html.py</td>
<td>103</td>
</tr>
<tr>
<td>Annex B – FastUniFrac XML files</td>
<td>Pages</td>
</tr>
<tr>
<td>cluster_samples.xml</td>
<td>104</td>
</tr>
<tr>
<td>PCoA.xml</td>
<td>105</td>
</tr>
<tr>
<td>p_test_significance.xml</td>
<td>106</td>
</tr>
<tr>
<td>unifrac_significance.xml</td>
<td>107</td>
</tr>
<tr>
<td>sample_counts.xml</td>
<td>108</td>
</tr>
<tr>
<td>sample_distance_matrix.xml</td>
<td>109</td>
</tr>
<tr>
<td>jackknife_sample_clusters.xml</td>
<td>110</td>
</tr>
<tr>
<td>Annex C – FastUniFrac deploy package pseudo-code</td>
<td>Pages</td>
</tr>
<tr>
<td>integrate.py</td>
<td>111</td>
</tr>
<tr>
<td>Annex D – QIIME automated integration pseudo-code</td>
<td>Pages</td>
</tr>
<tr>
<td>galaxy_integration.py</td>
<td>112</td>
</tr>
<tr>
<td>xml_generator.py</td>
<td>114</td>
</tr>
<tr>
<td>tgz_manager.py</td>
<td>122</td>
</tr>
<tr>
<td>Bibliography</td>
<td>Pages</td>
</tr>
</tbody>
</table>

2
1. Introduction

1.1. Motivation

Advances in DNA sequencing technology have allowed us to comprehensively characterize microbial communities in a variety of environments, including those associated with the human body. Our body hosts a wide variety of microbial organisms that outnumber human cells by a ratio of 10 to 1 \cite{1} and perform critical roles in our well-being \cite{2}. Changes in these microbial communities (collectively known as the microbiome) are associated with different diseases, such as obesity and Crohn’s disease \cite{3}.

Analysis of microbial communities has been traditionally limited by the fact that we could only cultivate a minority of the microorganisms present in a sample. Culture-independent methods and the introduction of high-throughput sequencing technology have allowed us to characterize with higher precision the contents of the microbiome \cite{4}. These changes in methodology have produced in a drastic increase in the number of DNA fragments that we now obtain in the analysis of each sample, which in turn has resulted in a need for efficient and scalable tools that allow us to make sense of this vast amount of data. QIIME (Quantitative Insights Into Microbial Ecology) is one such tool tailored for the analysis of high-throughput data in microbial studies \cite{5}. QIIME provides an integrated pipeline for the analysis and visualization of similarities and differences across large number of samples, incorporating third party tools commonly used for the analysis of microbial data.

QIIME is used by a large user base of biologist and bioinformaticians to perform analysis of microbial communities, and has been extensively cited in the literature (e.g. \cite{6, 7}). However, QIIME currently does not incorporate a Graphic User Interface (GUI), which makes it difficult to utilize for users not accustomed to the command line interface. The lack of a GUI makes QIIME less accessible to researchers, due to the high learning curve associated with using the command line. As Stephenson said “...hard things can be made easy, and complicated things simple, by putting the right interface on them.” \cite{8}.
1.2. Objectives

The main objective of this project is to make QIIME more accessible to researches by giving them a user-friendly interface. Furthermore, we use an interface that the users are used to work with. This way, we integrate QIIME on Galaxy project, which already includes tools used by researchers to analyze genomic data. The innovative aspect of this project is the fact that this integration on Galaxy is done automatically through a Python software.

Specifically, the following tasks are in the scope of the project:

- To analyze QIIME in order to get the characteristics which the GUI must satisfy.
- To analyze the GUI in order to get the potential limitations when integrating QIIME.
- To replicate the FastUnifrac website using QIIME scripts and the GUI.
- To generate the GUI for the most updated version of QIIME.
- To test the GUI deeply.
- To configure Galaxy with QIIME on Amazon EC2.
2. Problem statement

2.1. Introduction to QIIME

QIIME is a Python-based open source software pipeline built using the PyCogent toolkit [9]. QIIME provides an integrated pipeline for the analysis and visualization of similarities and differences across large number of samples. In this section, we will describe the core analysis steps performed using QIIME.

The first step in a QIIME analysis is to filter the raw DNA sequences (i.e. the sequences unprocessed, as obtained from the sequencing instrument) for quality and de-multiplex these reads. Multiplexing is a process by which a short DNA fragment, denominated “barcode”, is attached to each read, with the peculiarity that there is a unique barcode for each sample. In this way, after sequencing we can identify to which sample a sequence belongs just by checking the barcode attached to it (see Figure 2.1). Filtering is achieved by utilizing a quality score file, also provided as an output of the sequencer, and which determines the overall quality of each nucleotide in a DNA fragment.

![Figure 2.1 – De-multiplexing process](image)

The next step in the workflow is clustering the DNA reads into Operational Taxonomic Units (OTUs). OTUs are cluster of sequences intended to represent some degree of taxonomic relatedness. By default, QIIME uses UCLUST [10] to compute the clustering. The idea behind this algorithm is to group the sequences by a certain percentage of global identity (typically 97%). The global identity is computed as the number of matching nucleotides divided by the length of the shorter sequence (see Figure 2.2).
UCLUST uses a representative sequence (called seed) to define a cluster. In a cluster, each sequence matches the seed over the global identity threshold, but a pair of non-seed sequences may have identities below the threshold. To perform clustering, UCLUST starts with an empty database of seeds and takes each sequence to be clustered in input order. Then, the sequence is compared with the seeds of the database. If the sequence matches to a seed in the database, it is added to that cluster, otherwise the sequence becomes a new seed in the database and, therefore, a new cluster is created (see Figure 2.3).

![Figure 2.2 - Global identity computation](image)

\[
\text{Identity} = \frac{\text{matches}}{\text{length shorter sequence}} = \frac{8}{9} = 88.89\%
\]

**Figure 2.2 – Global identity computation**

![Figure 2.3 - UCLUST clustering algorithm](image)

**Figure 2.3 – UCLUST clustering algorithm**
Once the sequences have been clustered, one sequence from each OTU is picked as a representative. The representative sequence is defined as the most abundant sequence present in an OTU, and will be used in downstream analysis for taxonomic assignment and phylogenetic reconstruction. Taxonomic assignment is a process in which a sequence (in this case, a sequence representing an OTU) is compared against a reference dataset of sequences for which a trusted taxonomic classification has already been established. The sequence is then assigned its own taxonomy based on similarity to reference sequences. Notice how it is possible that different reads in the same dataset get assigned at different taxonomic depths and with varying levels of confidence in the assignment (see Figure 2.4).

Reference tree:

```
X
  /   \
 /     \   
Y       Z
```

Sequences:
- Seq1: GGGG
- Seq2: AATT
- Seq3: CCCC

Assigned taxonomy:
- X:Z:c
- X:Y:-
- X:-:-

Figure 2.4– Taxonomy assignment

After taxonomy assignment, the representative sequences are aligned in order to construct a phylogeny. This alignment is done using the PyNAST tool [11]. PyNAST aligns a candidate sequence to a template alignment. First of all, PyNAST uses BLAST [12] to find the sequence most similar to the candidate sequence in the template alignment; this will be the template sequence. Next, the gaps of the template sequence are removed and it is pairwise-aligned with the candidate sequence. The gaps previously removed from the template sequence are then reintroduced into the pairwise alignment. This step can produce a pairwise alignment longer than the template alignment, and so in order to avoid this and guarantee that the length is the same as the template alignment, gaps introduced during pairwise alignment are removed and the nearest gap character in the aligned candidate sequence is also removed in order to maintain the alignment. Thereby, local misalignments have been introduced but they are permitted so that the global alignment is achieved and the length of the template alignment is not disrupted (see Figure 2.5).
Once the alignment is completed, QIIME performs a filtering step to facilitate the generation of a phylogenetic tree. This filtering will remove positions that only contain gaps in every sequence. The user can additionally supply a lamemask file defining what positions should be included when building the tree and which should be ignored. After this filtering, the phylogenetic tree is built using FastTree, which infers an approximately-maximum-likelihood tree from the filtered alignment [13].

The FastTree algorithm proceeds as follows. First, it implements a heuristic Neighbor-Joining variant. Unlike the original Neighbor-Joining [14], FastTree stores a sequence profile for each node. A profile is defined by a frequency vector for each position. The profile of an internal node is the weighted average of its children’s profiles (see Figure 2. 6). Like original Neighbor-Joining, FastTree starts with the tree as a star topology and it refines the topology by joining the best pair of nodes, which means create a new internal node and append this pair of nodes from it. In order to reduce the number of comparisons between nodes, FastTree uses a combination of heuristics to select the best two nodes to join. Second, the resulted topology is improved using nearest neighbor interchanges (NNI). NNI takes each split of the tree and computes the alternate topologies. Then, it selects the best topology based on log-corrected profile distances (see Figure 2. 7). Finally, FastTree computes a local bootstrap for each internal node in order to estimate the tree’s reliability. Bootstrapping is counting the fraction of column resamples of the profiles that supports the original topology over the alternate topologies. The resulted phylogenetic tree will be later used during diversity analysis to calculate phylogenetically-aware metrics [15, 16].
At this point, the core object on which all subsequent analysis is based is constructed: a contingency table with columns representing samples, rows representing OTUs, and OTU counts per sample as entries of the table. This OTU table can contain metadata associated both with the OTUs (e.g. taxonomy calculated as described above.) and/or with the samples [17].

Utilizing an OTU table that includes taxonomic information, a summary of taxonomic groups found in a study can be constructed. OTUs are grouped at different taxonomic levels to produce a series of barplots that summarize what bacterial groups can be observed in each sample (see Figure 2.8). Since the samples can be sorted in the horizontal axis of this graph, it is straightforward to visualize differences according to groups of samples, e.g. controls versus treatments (Figure 2.9).
In order to assess the diversity observed within a given sample (alpha diversity), QIIME provides 24 different diversity metrics. However, and in order to eliminate sampling bias, rarefaction is usually performed first on the OTU table so diversity values are not inflated in samples with OTUs having larger number of observations (i.e. sequences) (see Figure 2. 10). Rarefaction is performed several times to obtain a distribution of diversity values so that, at each rarefaction level, an average and a standard deviation can be provided. From these results, a plot of alpha diversity versus simulated sequencing effort is generated (see Figure 2. 11).
Figure 2. 10 – Example of rarefaction
While alpha diversity is concerned with diversity within a sample, beta diversity provides an explicit estimation of diversity between samples. In order to reduce variable sequencing effort across samples, rarefaction is again performed before diversity is calculated. Several metrics for beta diversity are currently implemented in QIIME, most notably UniFrac [15], a phylogeny-aware metric that calculates the distance between two communities (i.e. the OTUs present in each of the samples) as the fraction of the total branch length that leads to members of either community, but not to both. A distance matrix with all-versus-all distances is thus calculated which can then be used as the input of different ordination techniques such as PCoA. This techniques will represent samples as points in a three dimensional space, which can be used to visualize differences between different groups of samples.

*Figure 2. 11 – Alpha diversity plot*
The steps detailed above describe the core functionality of QIIME, although there are several other methods that can be applied to any dataset, such as jackknifed beta diversity, hierarchical clustering, or construction of OTU/sample networks. Each of these steps can also be modified to run with a wide variety of parameter values, which are set by the user depending on the nature of the dataset being analyzed and the particular hypotheses to be tested. All these features make QIIME an extremely powerful tool, at the price of having a steep learning curve for users, and particularly so given the lack of a user interface.
2.2. Analysis of QIIME

To better understand what characteristics the QIIME GUI should fulfill, we need first to analyze how QIIME itself is designed and utilized. Depending on the user needs, QIIME can be used on platforms ranging from a small laptop to large cluster systems, including cloud computing. QIIME is implemented in a highly modular fashion, incorporating several third-party tools used at different steps of the analysis of microbial data. As its user base expands, new functionality keeps being constantly added to QIIME, which has resulted in a linear growth both in number of scripts and in lines of code, as seen in the Figure 2.13:

![Figure 2.13 – Growth of QIIME](image)


2.3. GUI characteristics

The main goal of a GUI is to make software more usable. Nielsen [18] defines five main usability characteristics:

- **Learnability**: the GUI must be easy for the user to accomplish tasks the first time they encounter it.

- **Efficiency**: once the GUI use has been mastered, the users should be able to perform tasks quickly.

- **Ability of infrequent users to return to the system without having to learn it all over**: the users can reestablish proficiency easily after a period of not using the GUI.
• Frequency and seriousness of user errors: the users do not make errors frequently, and if they do, the errors are not critical and can be easily recovered from.

• Subjective user satisfaction: the users feel confident and it is pleasant for them to use the GUI.

These five characteristics are generic for any GUI. Specifically, we would like the following characteristics to be incorporated into the QIIME GUI:

• System-independent: the main QIIME code base can be installed in systems ranging from a laptop to large cluster systems; the GUI must therefore be able to work in such systems.

• Capacity to leverage cloud computing resources: the GUI should be able to work within a cloud computing environment.

• Automated generation: given the quick growth of the QIIME code base, the GUI should be automatically generated in order to keep up with such growth. Manual implementation would require constant modifications with each new functionality or update introduced in the main code base.
3. Previous work

There have been some attempts to create a GUI for QIIME in the past, but they did not support all the functionalities included in QIIME or have not been maintained up-to-date. In this section, we will discuss two such efforts.

3.1. QIIME database

The QIIME database has been designed to serve as a common platform for the analysis of microbial community data [19], and it can be utilized de facto to access QIIME functionality through a simplified user interface. The QIIME database allows users to compare their datasets against previously characterized samples and already uploaded in the database, thus facilitating cross-studies analyses for users with no computational skills.

A user willing to compare his dataset to existing studies will start by uploading his raw DNA sequences to the DB. He can then execute a QIIME analysis workflow to pre-process his data and generate an OTU table. This workflow is composed of several steps, but the user is not allowed to modify the different options involved during this pre-processing, as this part of the analysis must be common across different studies to reduce bias.

Once the OTU table is generated, the user can perform some of the downstream analyses offered by QIIME that take an OTU table as an input. It should be noticed that the user is allowed to modify the parameter values utilized during these analyses. Currently, the QIIME database only supports taxa summaries, OTU table heatmaps, and alpha and beta diversity. Finally, results are presented as a list of files for the user to retrieve.

The main advantage of the QIIME database is that it can automatically update the GUI according to changes in the underlying QIIME scripts.

On the other hand, the QIIME database is not system-independent, and users cannot have a local copy of the database. Additionally, the QIIME database pre-processes all the raw data using a common set of parameters. For instance, DNA reads produced using Roche’s 454 sequencer with Titanium chemistry are trimmed to FLX length, so that read length does not affect downstream analysis when comparing across studies. However, a user might be interested in pre-processing its dataset in different way, and particularly so when using the standardized pre-processing of the QIIME database leads to loss of information. Furthermore, the QIIME database does not offer all the functionalities currently present in QIIME, such as jackknifed beta diversity (jackknifed_beta_diversity.py) or hierarchical clustering (upgma_cluster.py).

3.2. QIIME integration on Galaxy

QIIME 1.2.1 was partially integrated into Galaxy [20-22], and the code is still available on the Galaxy tool shed [23]. Galaxy is a web-based open platform for high-throughput data analyses. Galaxy allows integrating custom tools by defining XML configuration files and it generates automatically a simple consistent interface for the tool.
In our specific case, due to some Galaxy’s characteristics, there was necessary to develop a QIIME wrapper. The QIIME wrapper is a python script which acts as a command line interface between Galaxy and the QIIME scripts (see Figure 3. 1). The main objective of the QIIME wrapper is to allow Galaxy to keep track of the output files resulting from running the QIIME scripts.

![Diagram of Galaxy and QIIME wrapper](image)

**Figure 3. 1 – QIIME wrapper functionality**

Although this implementation achieves system independence and capacity to leverage cloud computing, it presents several drawbacks. For instance, this solution does not allow for automated generation of the GUI, one of the requirements it must fulfill, as previously discussed in Chapter 2. As the XML configuration files must be generated manually, every change in the script interface would imply a subsequent manual change on the XML file linked with this script. Additionally, adding new scripts to QIIME would also result in a required change of the QIIME wrapper to incorporate code specific for the added script. This solution is therefore non-scalable.
4. Proposed solution

After analyzing the previous projects on the QIIME GUI and expose their weaknesses and strengths, we will design and develop a QIIME GUI taking advantage of the knowledge acquired from the previous projects. Our approach consist of automatically integrate QIIME on Galaxy.

First, we will describe deeply how Galaxy works. Then, we will manifest how it fulfills the requirements for the QIIME GUI. Next, we will present the disadvantages of using Galaxy as a solution. After, we will show the case study used to understand the logic behind Galaxy. Finally, we will present the software generated to automatically integrate QIIME on Galaxy.

4.1. What is Galaxy?

Galaxy project [20-22] is an open-source framework which offers interactive tools for perform accessible, reproducible and transparent large-scale genome analysis. Before going into the details of Galaxy, we will define the accessibility, reproducibility and transparency problems.

The accessibility problem consists in the fact that the scientists without programming or informatics expertise are not able to use the computational tools tailored for researching in the life sciences. The have to overcome challenges ranging from the tools installation to combine different tools in an analysis chain efficiently, including determining the best parameter values to use in such tools.

The reproducibility of the experimental results is a key aspect in the scientific field. By reproducing the results, the scientific community can understand the process carried out. Consequently, others researches can integrate such process in their analyses and extend the results, providing new discoveries on the field. In the case of genetic researches, the reproducibility of the results has become a serious problem due to the lack of standards, the use of complex tools and the large size of the used datasets. An investigation has showed that less than a half of microarray gene expression analyses published in *Nature Science* could be reproduced [24]. In order to achieve reproducibility, the concept of Reproducible Research System (RRS) has been introduced [25]. An RRS is a system which offers an execution environment which automatically records all the metadata needed to reproduce an experiment. This metadata include the used methods, the origin of the data and the obtained results, among others. By recording this metadata, any researcher can successfully reproduce the analysis performed.

The fact that an experiment is reproducible does not ensure that the experiment is understandable and/or reportable. That is the transparency problem. In other words, transparency is the capacity of promote accountability and collaboration through open sharing and communication of experimental results. The transparency should be achieved through the RRS systems. An RRS system should provide a document-preparation system which allows embedding directly the analyses into a paper. Hence, a researcher can describe his investigation joining text and direct links to the execution environment, providing all the
information about computational results, analyses and methods needed for achieving transparency.

At this point, we have the accessibility, reproducibility and transparency problems defined, but how Galaxy overcomes these Challenges? Galaxy is an RRS which uses the following approaches to fulfill the accessibility, reproducibility and transparency requirements.

In order to guarantee accessibility, Galaxy offers a unified point-and-click web-based interface for getting the data and applying developed tools to analyze the data (see Figure 4.1). This approach avoids the necessity of using the command line interface, which is averse for most of the biomedical researches.

![Galaxy's point-and-click web-based interface.](image)

**Figure 4.1 - Galaxy’s point-and-click web-based interface. It is divided in 4 parts: on the top there is the navigation bar; in the left column there is the tool panel, which contains a list of the available tools; in the middle column there is the detail panel, where the user can select the parameter values for the tool; and, in the right column, there is the history panel, which shows the available datasets and the analyses step taken.**

Since Galaxy implementation is highly modular, it allows rapid integration of new developed tools. Thereby, it needs some mechanism to ensure a unified look and feel and functionality of the tools interface. This mechanism lies in automatically generating the interfaces from abstract descriptions of the tools (see Figure 4.2). The only constraint on the tool is that it should be callable from the command line. In order to integrate a new tool in Galaxy, the developer should write a configuration file which has a description of how to run the tool and a detailed specification of its input and output parameters. In spite of losing flexibility on the interface definition, this approach allows Galaxy to work with the tool as an abstract object, ensuring accessibility.
Galaxy provides reproducibility by using datasets and automatically storing metadata for each analysis step. Datasets are the Galaxy's abstraction of files and are where the data is stored in. Apart from storing the data, datasets also include metadata which describes the data stored. This metadata includes the type of the data (file format), name of the dataset, additional information by the user and additional values specific for the type of data. Galaxy manages datasets using a local database (see Figure 4.3). All the dataset's metadata is stored in the database. Since most of the tools tailored for genetic analyses takes files as an input, Galaxy does not stores the data in the database. It stores the path to the local file.

As we mentioned above, Galaxy stores metadata for each analyses step. This metadata consist of the input datasets, the tool used to perform the analyses step, the used parameter values and the output datasets (results). All this metadata is also stored in the database. Finally, to fully guarantee reproducibility, the datasets and analyses step are grouped into histories, which can be copied and versioned. The history panel (see Figure 4.4) of the Galaxy interface allows viewing the stored metadata for each dataset and analysis step.
Finally, Galaxy offers transparency through three different ways: (1) a sharing model for datasets and histories; (2) a framework for viewing published datasets and histories; and (3) Galaxy Pages. The sharing model for datasets and histories allows Galaxy users to share their datasets and histories using a web link. The user is also allowed to publish their datasets and histories. A published dataset or history is included in the Galaxy’s public repository, giving them a readable web link editable by the user. The web links allows to other users to import and use the published datasets or histories. Using the framework for published datasets and histories, a user can search, sort and filter the published elements in order to find those of the user interest. The last but not least Galaxy’s mechanism to guarantee transparency is Galaxy Pages. Galaxy pages are customizable documents where the user can expose the details of his analyses by mixing text, graphics and embedded Galaxy’s histories and datasets. Galaxy pages were tailored to be an excellent publication supplement, ensuring the transparence of the experiment.

Summing up, all these characteristics make Galaxy an excellent solution for biomedical researchers since they can perform their analyses using a web browser and they do not have to be worried about the computation details or storage management, while they are using an available, reproducible and transparent framework.
4.2. Why Galaxy?

We have defined the Galaxy framework, but how it fulfills the requirements of the QIIME GUI? First, we will justify how Galaxy achieves the generic requirement for any GUI and then we show which mechanisms has Galaxy in order to fulfill the QIIME-specific requirements.

4.2.1. Generic GUI requirements

The Galaxy interface has the main advantage of being a web-based interface built following the standard web interface guidelines established by Nielsen [26]. This guidelines ensures that the built interface is consistent, provides visual feedback and it allows access to help and documentation. The most remarkable aspect is, because of following standard guidelines, the interface is not a completely new different interface from the functionality point of view. This fact guarantees the learnability of the GUI, since the user is accustomed to use similar interfaces. Furthermore, this characteristic also provides the ability of infrequent users to return to the system without having to learn it all over, due to it works similarly as other web interfaces that the user uses.

Once the user already knows how to use the interface, the simple and intuitive Galaxy point-and-click interface allows the user to perform task quickly, ensuring efficiency. As a very simple interface, the errors made by the user are more related with the lack of knowledge of the tools than with interface issues. Even so, Galaxy offers a mechanism to recover from the errors and to know what caused the error. This mechanism is provided by the history panel. The history panel shows each analysis step in four different ways, depending on the state of the analysis step (see Figure 4.5): in queue (the analyses step is waiting to be run), running (the analysis step is being executed), successful (the analysis step finished without any problem), or error (the analysis step did not finished successfully). In the case of error state, the history panel also includes a link to a more detailed description of the error, giving more feedback to the user.

Figure 4.5 – Different states of an analysis step in the history panel: (a) in queue, (b) running, (c) successful and (d) error (with the link to a more detailed description of the error).
Finally, the subjective user satisfaction is a hard requirement to quantify. One way to do this is looking at the number of Galaxy users and the number of jobs that the Galaxy’s public web server [27] process per day. Currently, Galaxy has more than 15,000 users worldwide and its public server process about 5,000 jobs per day [22]. This means that Galaxy is usable for many biologists. Moreover, there are 20 public Galaxy servers hosted by other organizations (such Genboree [28] or Nebula [29]), evidencing the users confidence.

4.2.2. QIIME-specific GUI requirements

The first specific requirement for the QIIME GUI is system independency. Galaxy fulfills this requirement by allowing the users to download the Galaxy source code from this repositories and, thereby, having local Galaxy installation. Since Galaxy works as a web server, it does not require any specific hardware. Galaxy leaves the door open to local installation because the size of the analyzed datasets is increasingly in such way that often it is not possible to move the data due to the time and money costs. Hence, it makes sense that the users can have a local Galaxy installation placed near to the data, being more efficient. Furthermore, the data used in the analyses are sometimes private or sensitive, so analyze this data outside the institution servers is prohibited.

The second specific requirement for the QIIME GUI is the capacity to leverage cloud computing resources. The Galaxy Team has developed Galaxy CloudMan [30], a software package integrated with Galaxy which allows creating and managing clusters formed by Amazon EC2 instances using a web-based interface, hiding the low-level details. Galaxy CloudMan will be explained in depth on Chapter 5.

Finally, the last specific requirement for the QIIME GUI is the automated generation. As we said before, Galaxy uses an abstraction of the tool to automatically generate the GUI, ensuring the same look and feel and functionality through the integrated tools. Actually, this abstraction is represented by an XML file which contains a detailed description of the tool (see Figure 4.6). Specifically, the XML file must contain the following information:

- **Id**: an internal identifier which allows versioning and getting statistics of the tool (e.g. times the tool is used). It should be unique for each tool integrated on Galaxy.
- **Name**: the name used for display the tool in the tool panel.
- **Description**: a short and precise description of the tool, which will be displayed next to the tool name in the tool panel.
- **Command**: the command needed to invoke the tool with the user-defined parameter values.
- **For each input parameter**:
  - **Name**: the name of the parameter which is used to pass the parameter values to the tool by command line.
- Type: the type of the parameter. It can be integer, float, text, select, data, boolean, among others.

- For each output parameter (output file):
  - Name: the name of the parameter which is used to pass the output file path to the tool by command line.
  - Format: the format of the output file. It can be txt, pdf, fasta, jpg, among others. Furthermore, the developer can extend the list of available formats by defining a new type in a configuration file.

- Help: a long and detailed description of the tool, which is displayed in the detail panel, below the interface needed for defining the parameters.

There is more information that the developer must provide in order to successfully integrate a tool in Galaxy, but it depends on the values of the above fields. For example, in case the type of an input parameter is select, the developer should also provide the list of allowed options. Furthermore, the developer can improve the tool description by adding extra information in the XML file, although it is not needed for generate the interface or call the tool. Some examples of this extra information are the version of the tool, the label for an input parameter (which is showed in the detail panel, giving a short description of the parameter’s meaning) or the format of an input parameter of type data (which applies a filter on the datasets when the user are selecting the parameter value).

```xml
<tool id="example_tool" name="example tool" version="1.0">
  <description>
    Short description of the script
  </description>
  <command>
    example_tool.py -i $input -o $output
  </command>
  #if $float:
  #  $float
  #endif
  <input>
    <param label="Example input option" name="input" optional="False" type="data"/>
    <param label="Example float option" name="float" optional="True" type="float"/>
  </input>
  <output>
    <data format="txt" name="output"/>
  </output>
  <help>
    A more detailed description of the script
  </help>
</tool>
```

Figure 4. 6 – An example of Galaxy’s XML file. This example defines a tool which is a python script and it requires as an input a file and generates a txt file as an output. Optionally, it also accepts a float parameter.

At this point, we have showed how Galaxy generates the interface automatically, but the XML files are still generated manually. In order to completely generate the interface automatically we should also automate this process. Therefore, we will develop a Python
software which, given a QIIME script, generates automatically the XML file. How can we do that?

We can take advantage of the fact that each QIIME script has a dictionary called \texttt{script\_info} (see Figure 4.7) which contains all the information that we need to generate the Galaxy’s XML configuration file. Specifically, the \texttt{script\_info} dictionary has the following keys:

- \texttt{brief\_description}': it is a string with a short and precise description of the script.
- \texttt{script\_description}': it is a string with a long and detailed description of the script.
- \texttt{script\_usage}': it is a list of tuples where each tuple represent a usage example of the script. Each tuple consists in three strings: the title of the usage example, the explanation of the example and an example command line call to execute the example.
- \texttt{output\_description}': it is a string with a short description of the output.
- \texttt{required\_options}': it is a list of option objects. All the options listed here are required options of the script, so the user must specify a value for each of these options. In order to define an option object, the developer must provide the command line string which will be used to pass this option, the type option and a help message.
- \texttt{optional\_options}': it is a list of option objects. All the options listed here are optional options of the script, so the user is not required to specify a value for each of these options.
- \texttt{version}': it is a string with the current version of the script.

```python
script_info = {}  
script_info[\'brief\_description\'] = \"Short description of the script\"  
script_info[\'script\_description\'] = \"A more detailed description of the script\"  
script_info[\'script\_usage\'] = [  
    (\"Example:\", \"How to use the script\", \"prog -i in\dir -o out.tgz\")  
]  
script_info[\'output\_description\'] = \"Short description of the output\"  
script_info[\'required\_options\'] = [  
    make_option(\'-i\', \'--input\', type=\'existing\_filepath\', help=\'Example input option\'),  
    make_option(\'-o\', \'--output\', type=\'new\_filepath\', help=\'Example output option\')  
]  
script_info[\'optional\_options\'] = [  
    make_option(\'-f\', \'--float\', type=\'float\', help=\'Example float option\')  
]  
script_info[\'version\'] = \_version
```

**Figure 4.7** – \texttt{script\_info} dictionary example. This example belongs to a script which it requires an input file, and generates an output file. Optionally, it also accepts a float option.

The QIIME scripts have the \texttt{script\_info} dictionary because they use the PyCogent’s \texttt{option\_parsing} module for parsing the command line options. The \texttt{option\_parsing} module is an extension of the Python’s \texttt{optparse} module. This module allows the users to specify options by using the GNU/POSIX syntax and it generates usage and help messages. Moreover, the developer can define the type of the options and the module will automatically check if the value passed by the user is of the defined type. The \texttt{option\_parsing} module extends \texttt{optparse} by adding new useful types: \texttt{existing\_filepath} (a file path, the module checks if the file exists), \texttt{existing\_filepaths} (comma-separated list of file paths, the module checks if each listed file
exists), `existing_dirpath` (a path to a directory, the module checks if the directory exists),
`existing_path` (a path to a file or directory, the module checks if it exists), `new_filepath` (new path to store an output file, the module does not perform any check), `new_dirpath` (new path where to create the output directory where to store the results in, the module does not perform any check) and `new_path` (a new path where to store the output file or create a directory where to store the results in, the module does not perform any check). Without these extra option types, the automated generation of the XML files could not be possible, since there is no way to distinguish between a string parameter and an input or output file parameter.

Summing up, we will develop a Python software which will extract all the needed information from the `script_info` dictionary and it will automatically generate the XML files. This software is explained in detail on Chapter 4.5 QIIME automated integration.

### 4.3. Galaxy: disadvantages

As we have seen in the previous chapter, Galaxy fulfills the requirements of the QIIME GUI but, on the other hand, it presents few challenges that we will need to overcome in order to successfully integrate QIIME on Galaxy.

The first challenge is to allow Galaxy to keep track of the QIIME scripts’ output. Most of the QIIME scripts generate a directory which contains all the output files. In principle, Galaxy does not accepts directories as a valid output dataset, since it uses a database to manage them, but it provides a mechanism to manage datasets composed by more than one file: the `extra files path` mechanism. This mechanism works as follows. For each output file defined in the XML file, Galaxy creates a directory for this file (`extra files directory`). The path of this directory is stored at the database as metadata of the output file. If the output dataset is composed by more than one file, the `main` file is stored as the output of the analysis step and the other files are stored in the directory created for this file. The important thing is that the `main` file must contain links to the other files. When the `main` file requires any of the extra files generated, Galaxy automatically searches it in the `extra files directory`. We will clarify how it works with the following example (see Figure 4. 8): a given tool generates an html file as an output. This html file contains links to two `jpg` files, called `img1.jpg` and `img2.jpg`. The html file will be stores as the output file and the two `jpg` files will be stores in the `extra files directory`. When the user tries to see the html file through the Galaxy’s interface, Galaxy automatically search the two `jpg` files in the `extra files directory`.

![Figure 4. 8 – Galaxy’s extra files path mechanism example](image)
Unfortunately, we can not take advantage of the Galaxy’s extra files path mechanism because the output files of the QIIME scripts are not necessarily related, so the main output file does not exist. Hence, to allow Galaxy to keep track of the QIIME scripts’ output, we will compress the directories in a tgz file. This way, Galaxy will manage the directory as a single file (see Figure 4.9).

![Figure 4.9 - Directory structure problem](image1)

The next challenge to overcome is also related with allowing Galaxy to keep track of the QIIME scripts output. Galaxy needs to define the name of the output files, so it can give them a unique name and avoid conflicts in the database. Fortunately, the QIIME scripts accept options to define the name of the output file(s). In case the output was a directory, we will use the name defined by Galaxy as the name of the resulted tgz compressed file (see Figure 4.10).

![Figure 4.10 - Predefined output file names problem](image2)

The third challenge presented by Galaxy is to define a priori the type of the output file. Since Galaxy generates the output file name itself, it needs to know the output file type. As we said before, most of the QIIME scripts generate a directory as an output, so we can define the output file type as a tgz file, due to we will compress the directory. The QIIME scripts which do not generate a directory as an output generate files from type biom, fna or txt. Unluckily, we
can not know the output file type from the script_info dictionary. In order to define the file type, we can take advantage of the fact that the biom and fna files are based on plain text files, so we can define them as a txt files (see Figure 4.11).

![Diagram of file format definition problem]

Figure 4.11 - File format definition problem

The last challenge presented by Galaxy is related to pass the options to the QIIME scripts. Galaxy can not provide the option type ‘existing_filepaths’ in the way that QIIME expects. QIIME expects a comma-separated list with the file paths, but Galaxy only provides an object of type list. In order to convert the list object into a comma-separated list, we will use Cheetah [31]. Cheetah is a template engine which will allow us to add Python code in the XML files. So, we will use it to add a Python function to convert the input option in a QIIME-compliant form (see Figure 4.12).

![Diagram of non-QIIME-compliant options problem]

Figure 4.12 - Non-QIIME-compliant options problem
4.4. **Case study: FastUniFrac**

Before we start the development of the software which will automatically integrate QIIME on Galaxy, we have replicated the FastUniFrac website [32] using QIIME scripts fully integrated on Galaxy. This integration was done by manually generating the XML files. This has provided us with the necessary understanding of the logic behind Galaxy XML files, and will allow us next to automatize the generation of all QIIME XML scripts for a complete integration.

The FastUniFrac website was developed before QIIME, and so it was based on custom scripts no longer maintained. Consequently, we had to analyze the FastUniFrac functionalities in order to figure out which QIIME scripts can perform the same analyses. Once we know the QIIME scripts needed, we can manually generate the XML files for the Galaxy integration.

4.4.1. **FastUniFrac functionalities**

The FastUniFrac website implements a suite of tools based on the UniFrac metric [15] and it is designed to handle very large datasets efficiently. These tools allow the user to perform comparisons of microbial communities using phylogenetic information.

The UniFrac metric takes a phylogenetic tree and measures the phylogenetic distance between two samples as the fraction of the branch length that is unique to one sample or the other. In other words, if two samples have completely different taxonomic groups, the UniFrac distance will be 1. On the other side, if they have exactly the same taxonomic groups, the UniFrac distance will be 0. Otherwise, the UniFrac distance will be between 0 and 1 (see Figure 4.13).

![UniFrac metric](image)

*Figure 4.13 - UniFrac metric (image from FastUniFrac website).*

The UniFrac metric described above is the unweighted UniFrac metric. This means that it does not take in account how many identical sequences are in a sample because duplicate sequences does not influence in the branch lengths of the tree. The weighted UniFrac metric takes into account the abundance information during the calculations. This fact allows the user, for example, to detect changes in how many organisms from each lineage are present in each sample. The raw weighted UniFrac distance value of each branch is computed following the equation showed at Figure 4.14.
Comparing samples using the weighted UniFrac metric will emphasize those samples with contains taxa that have evolved more quickly if the phylogenetic tree is not ultrametric (i.e. different sequences in the sample have evolved at different rates). To avoid this situation, the user can utilize the normalized weighted UniFrac metric. In this case, the branch lengths are normalized, so each sample is treated equally. The normalization is done by diving the raw weighted UniFrac distance value by the distance scale factor computed using the equation showed at Figure 4. 15.

\[ u = \sum_{i}^{n} b_i \times \left| \frac{A_i}{A_T} - \frac{B_i}{B_T} \right| \]

Where:
- \( n \) = number of branches
- \( b_i \) = length of the branch \( i \)
- \( A_i \) = descendants of branch \( i \) from community A
- \( B_i \) = descendants of branch \( i \) from community B
- \( A_T \) = number of sequences from community A
- \( B_T \) = number of sequences from community B

Figure 4. 14 - Weighted UniFrac equation

\[ D = \sum_{j}^{n} d_j \times \left( \frac{A_j}{A_T} + \frac{B_j}{B_T} \right) \]

Where:
- \( n \) = number of branches
- \( d_j \) = distance of sequence \( j \) from the root
- \( A_j \) = times sequence \( j \) appears in community A
- \( B_j \) = times sequence \( j \) appears in community B
- \( A_T \) = number of sequences from community A
- \( B_T \) = number of sequences from community B

Figure 4. 15 - Distance scale factor computation

4.4.1.1. Input files

In order to start performing analyses, the user must provide three files: a sample ID mapping file, a category mapping file and a reference tree file.

The sample ID mapping file is a plain text file which shows from what Sample ID a Sequence ID came from and how many times it appears in the sample. This file contains three columns separated by tabs, where the first columns is the Sequence ID, the second columns is the Sample ID and the third columns is the number of times this sequence has been observed in this sample (see Figure 4. 16).

```
1  AFI41999  R4FL  1
2  AFI61411  R4FL  1
3  AFI41400  R4FL  2
4  AFI41403  R4FL  1
5  AFI41406  R4FL  1
6  AFI41409  R4FL  2
7  AFI41981  R4FL  1
8  AFI41984  R4FL  1
9  AFI41987  R4FL  1
10 ...
```

Figure 4. 16 - Sample ID mapping file example
The sample ID mapping file is analogous to the QIIME OTU table file, which is a file in biom (Biological Observation Matrix) format [17]. In order to be able to perform analyses with the QIIME pipeline, it is necessary to transform the sample ID mapping file to an OTU table file in biom format. This process will be explained in detail in Chapter 4.4.2 Developed software.

The category mapping file is a plain text file which contains extra information (a.k.a. metadata) about the samples. This file is useful to explain the variation that the user plans to examine in their studies, so he can view the differences between different groups of samples (e.g. OTUs in samples from mice with two different treatments). This file is a tab delimited file, where the first line starts with a hash character (#) followed by the headers of the different columns, separated by a tab. After the headers line, the file can contain more lines starting with a hash to, which are comments about the study. Finally, the file must contain one line for each sample in the study. There are some restrictions over the category mapping file: the first header must be exactly ‘SampleID’ and the last one ‘Description’. Furthermore, the samples ID’s must be unique, short and meaningful and they can contain only alphanumeric characters plus ‘.’, ‘+’ and ‘#’ (see Figure 4.17).

![Figure 4.17 - Category mapping file example](image)

The category mapping file is analogous to the QIIME mapping file. The QIIME mapping file is also a tab delimited file, with the same structure as the category mapping file, but it requires two more headers: ‘BarcodeSequence’ and ‘LinkerPrimerSequence’, which must be the second and third header, respectively. This two headers are used during the de-multiplexing process (detach the barcode from the sequence and identify to what sample came the sequence). Similarly to the category mapping file, the ‘SampleID’ must be unique for each sample, but in the case of the QIIME mapping file, it can contain only alphanumeric characters plus ‘.’. Due to these differences, it is necessary to transform the category mapping file to the mapping file (see Chapter 4.4.2 Developed software).

The reference tree file is a plain text file which contains a tree in Newick format (see Figure 4.18). This tree is a phylogenetic tree which shows the genetic relationships between the different OTUs present in the study.
The Newick format is a standard representation of a tree, so it will not be necessary to perform any change in this file.

### 4.4.1.2. Cluster samples

The cluster samples analysis is used to determine which samples have similar microbial communities. First, a distance matrix is built by calculating pairwise UniFrac distances between samples. Based on this matrix, hierarchical clustering is performed, getting a tree as a result. This tree is provided to the user in Newick format and in an ASCII representation, making more visible the distances between the samples (see Figure 4. 19).

![Figure 4. 18 - Tree in Newick format (a) and its graphic representation (b)](image)

In order to perform the cluster samples analysis with the QIIME pipeline, we need to concatenate two QIIME scripts: `beta_diversity.py` and `upgma_cluster.py`. The first one generates the distance matrix based on UniFrac distances. The last one computes hierarchical clustering, generating a file with the Newick representation of the resulted tree. QIIME does not provide the functionality of showing a tree in ASCII representation taking into account the distances between samples, so we will develop a script which, given a file with a tree in Newick format, will generate a file with an ASCII representation of the tree.
4.4.1.3. **PCoA (Principal Coordinates Analysis)**

The Principal Coordinates Analysis (PCoA) is a multivariate statistical technique tailored to which axes explains better the samples variation. As the cluster samples analysis, the PCoA analysis starts generating the matrix of pairwise UniFrac distances between samples. Once the distance matrix is generated, the distances are turned into points in a space with n-1 dimensions, where n is the number of samples. This process generates a tab delimited file where rows are samples and the columns are the principal coordinates (PC) axes that explain the variance in this new space. In this output file are also included the % of variation explained by each principal coordinate.

In order to allow the user to see the relationship between samples, the principal coordinates can be paired up and plotted. The FastUniFrac website provides three plots with the first three principal coordinates paired: PC1 vs. PC2, PC1 vs. PC3 and PC2 vs. PC3. Furthermore, it provides a scree plot, which represents the fraction of variance explained by each axis and the cumulative variance of these axes (see Figure 4. 20). Moreover, the FastUniFrac website provides a 3D view of the first three principal coordinates using the KiNG applet to view the generated Kinemage file [33] (see Figure 4. 21). The user can download the Kinemage file and each plot in EPS format.

![Figure 4. 20 – PCoA: 2D plots and scree plot](image-url)
The PCoA analysis can be executed on QIIME by using the workflow script `beta_diversity_through_plots.py`. This script is responsible for executing all the QIIME scripts involved in the whole process of the PCoA analysis, starting with the UniFrac distances matrix to the 2D and 3D plots generation. The output of this script is a directory which contains a file with the distance matrix, a file with the output of the principal coordinates file (in the same format as the FastUniFrac website) and folders containing the 2D and 3D plots, accessible by html files. As we said in Chapter 4.3 Galaxy: disadvantages, Galaxy can not manage directories as an output datasets. In this case, we will take advantage of the Galaxy’s extra files path mechanism to allow the user viewing the results through the Galaxy’s interface. We will generate an html file which will act as a main output and it will have links to download the raw PCoA data, to the html files which gives access to the generated 2D and 3D plots and to download the Kinemage file.
4.4.1.4. P test significance

The P test [34] is a test which tells us how significantly different are two samples based on their phylogenetic information. It computes the number of changes from one sample to another along a branch (a.k.a. parsimony changes) needed to illustrate the distribution of sequences between the different samples in the tree. This result is used as an estimation of the similarity. Then, it performs randomizations of the sample to which each sequence is assigned. For each randomization, it computes the number of parsimony changes in this new tree, building a distribution. The first result obtained is then compared with the distribution. If this value is lower than the average for the distribution, the sequences are significantly clustered. The p-value is defined as the fraction of random permutations where the initial tree has more unique branch length than the randomized trees.

The P test significance can be computed in two ways:

- All samples together: in this case, the P test computes how all the sequences are significantly different from one sample to another. The output is the p-value which shows how significantly the samples are clustered in the tree (see Figure 4.22).

  Corrected P Value: 0.006, Raw P Value: 0.006000

  Figure 4. 22 - P test significance: all samples together output

- Each pair of samples: it computes pairwise significance. For each possible pair of samples, it first removes the sequence that does not come from one of the two selected samples from the tree. Then it computes the significance. The output is two all-versus-all tables: the first one contains the corrected p-values and the second one the raw p-values (see Figure 4. 23). The corrected p-values are computed using the Bonferroni correction (each value has been multiplied by the number of comparisons that were made).
Corrected P-Values: p-values have been corrected for multiple comparisons using the Bonferroni correction.

<table>
<thead>
<tr>
<th>EFP</th>
<th>EAPA</th>
<th>DFMN</th>
<th>REPL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Color Description
- (< 0.001) Highly significant
- (0.001-0.01) Significant
- (0.01-0.05) Marginally significant
- (0.05-0.1) Suggestive
- (> 0.1) Not significant

Click to download data

Raw P-Values: p-values have not been corrected for multiple comparisons.

<table>
<thead>
<tr>
<th>EFP</th>
<th>EAPA</th>
<th>DFMN</th>
<th>REPL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Color Description
- (< 0.001) Highly significant
- (0.001-0.01) Significant
- (0.01-0.05) Marginally significant
- (0.05-0.1) Suggestive
- (> 0.1) Not significant

Click to download data.

Figure 4. 23 - P test significance: each pair of samples output

The P test analysis can be executed through the QIIME pipeline using the beta_significance.py script. Currently, it only computes the ‘each pair of samples’ P test analysis. The output of this script is a tab delimited file with each possible pair of samples and the p value (corrected and raw) obtained for this pair. We will modify the beta_significance.py script in order to compute also the ‘all samples together’ P test analysis. Furthermore, we also need to create a Python script which takes the output file of the beta_significance.py and generates an interactive heatmap in order to show the results in the same way as the FastUniFrac web site.
4.4.1.5. UniFrac significance

This test computes the UniFrac values. Can be computed in three ways:

- All samples together: as the p-test analysis, it computes how all the sequences are significantly different from one sample to another. The output is the p-value (the UniFrac value obtained) (see Figure 4.24).

  *Distance P Value(s) - P Value: 0.628000*

  *Figure 4. 24 - UniFrac significance: all sample together output*

- Each pair of sample: it follows the same steps as the P test significance, but in this case use the UniFrac metric. The output is two tables: one with the Bonferroni corrected values and one with the raw values (see Figure 4.25).

  *Corrected P-Values: p-values have been corrected for multiple comparisons using the Bonferroni correction.*

  *Figure 4. 25 - UniFrac significance: each pair of samples*
Each sample individually: for each sample, it assumes that the rest of the samples are a single sample and computes if the selected sample has more unique branch length than expected. The output is a table showing the computed values for each sample (see Figure 4. 26).

<table>
<thead>
<tr>
<th>Sample</th>
<th>Distance</th>
<th>P Value(s)</th>
<th>Sample Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFL</td>
<td>0.9</td>
<td></td>
<td>Free living sample from estuary collected at an intermediate salinity (9 psu) in the North Channel of the estuary, about 1 m above the bed</td>
</tr>
<tr>
<td>EPA</td>
<td>0.064</td>
<td></td>
<td>Particulate attached sample from estuary collected at an intermediate salinity (9 psu) in the North Channel of the estuary, about 1 m above the bed</td>
</tr>
<tr>
<td>O#FL</td>
<td>0.954</td>
<td></td>
<td>Free living sample from coastal ocean collected about 1 m above the bed at the end of a flood tide</td>
</tr>
<tr>
<td>O#UN</td>
<td>0.39</td>
<td></td>
<td>Unfiltered coastal ocean sample collected about 1 m above the bed at the end of a flood tide</td>
</tr>
<tr>
<td>R#FL</td>
<td>0.382</td>
<td></td>
<td>Free living sample from freshwater river sample collected at mid-depth (10 m) at a location above the influence of salinity</td>
</tr>
<tr>
<td>R#PA</td>
<td>0.068</td>
<td></td>
<td>Particulate attached sample from freshwater river sample collected at mid-depth (10 m) at a location above the influence of salinity</td>
</tr>
</tbody>
</table>

The UniFrac significance can be obtained also with the `beta_significance.py` QIIME script. As we said above, this script only provides the ‘each pair of samples’ test. Apart from adding the ‘all samples together’ analysis, we will add the ‘each sample individually’ test. Furthermore, we need to develop a python script that given the ‘each sample individually’ test output generates a table showing the results.

4.4.1.6. Sample counts

The sample counts analysis gives statistics about the samples. This statistics can be the number of sequences in each sample or the number of different OTUs present in the samples. In both cases, the output is a table showing the SampleID, the number of sequences/OTUs and the Description field defined at the category mapping file (see Figure 4. 27).

```
<table>
<thead>
<tr>
<th>Sample ID</th>
<th>Count</th>
<th>Sample Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFL</td>
<td>48</td>
<td>Free living sample from estuary collected at an intermediate salinity (9 psu) in the North Channel of the estuary, about 1 m above the bed</td>
</tr>
<tr>
<td>EPA</td>
<td>44</td>
<td>Particulate attached sample from estuary collected at an intermediate salinity (9 psu) in the North Channel of the estuary, about 1 m above the bed</td>
</tr>
<tr>
<td>O#FL</td>
<td>22</td>
<td>Free living sample from coastal ocean collected about 1 m above the bed at the end of a flood tide</td>
</tr>
<tr>
<td>O#UN</td>
<td>21</td>
<td>Free living sample from coastal ocean collected about 1 m above the bed at the end of a flood tide</td>
</tr>
<tr>
<td>R#FL</td>
<td>16</td>
<td>Particulate attached sample from freshwater river sample collected at mid-depth (10 m) at a location above the influence of salinity</td>
</tr>
<tr>
<td>R#PA</td>
<td>12</td>
<td>Free living sample from freshwater river sample collected at mid-depth (10 m) at a location above the influence of salinity</td>
</tr>
<tr>
<td>Total Count</td>
<td>163</td>
<td>Unfiltered coastal ocean sample collected about 1 m above the bed at the end of a flood tide</td>
</tr>
</tbody>
</table>
```

The QIIME script `per_library_stats.py` provides the number of sequences in each sample, among other statistics. By default, this script prints its output through the standard output. If a mapping file is provided, apart from print the output through the standard output, it modifies
the *mapping* file adding a new header called *NumIndividuals* with for each sample adds the number of sequences for this sample. The *per_library_stats.py* script does not provide the number OTUs in a sample, so we will modify the script to add this functionality.

4.4.1.7. Sample distance matrix

The sample distance matrix analysis computes de UniFrac distance for all possible pairs of samples and generates an all-versus-all matrix with these values. This matrix is the matrix used in the *Cluster samples* and *PCoA* analyses. The output is an interactive table which represents the matrix colored by quartiles (see *Figure 4.28*).

![Sample distance matrix output](image)

*Figure 4.28 - Sample distance matrix output*

The QIIME script which generates this matrix is the *beta_diversity.py* script. The output of this script is a tab delimited file which contains the distance matrix. In order to show the results in the same way as the FastUniFrac website does, we will develop a python script which will take the output of the *beta_diversity.py* script and it will generate an interactive heatmap representing the distance matrix.

4.4.1.8. Jackknife sample clusters

The jackknife sample clusters analysis performs jackknifing the output of the *Cluster samples* analysis. Jackknifing is a statistical resampling technique. It takes a random subset of the sequences and re-runs the *Sample clusters* analysis. Then, it analyzes how often a cluster node of the original tree is recovered. This analysis determines how robust the analysis is. The output of this analysis is similar to the *Sample clusters* analysis: a text file with the tree in Newick format and an ASCII representation of the tree. In this case, the ASCII representation of tree has the nodes colored by the fraction of times the node has been recovered in the randomizations (see *Figure 4.29*).
In order to compute the jackknife sample clusters analysis with the QIIME pipeline we can execute the workflow script `jackknifed_beta_diversity.py`. This script is responsible to execute all the scripts involved during the jackknife process: compute the UniFrac distance matrix of the full OTU table, build the original sample clusters tree, generate the distance matrix and sample clusters from the rarefactions (subsamples) of the input OTU table, compare the rarified trees with the original determining the jackknife support, compute the principal coordinates analysis and build the 2D and 3D plots. Although this script generates a huge amount of files (the rarified distance matrices, rarified OTU tables, rarifies sample clusters tree, the jackknife support file, a Newick tree file with jackknifed values and the 2D and 3D plots) it does not generates an interactive ASCII representation of the tree. We will develop a Python script which will take the Newick tree file with jackknifed values and the jackknife support file and it will generate the interactive ASCII representation of the tree.

4.4.2. Developed software

In this chapter we will describe the Python libraries and scripts developed in order to offer the same functionalities than the FastUniFrac original website. The pseudo-code of the developed libraries and scripts is available on Annex A.

4.4.2.1. Python libraries

The principal motivation for developing Python libraries is to avoid code replication and promote code reusability. Analyzing the code that we need to develop, we can see the advantages of using libraries. For example, for the ‘each pair of samples’ P test and UniFrac significance analysis and for the sample distance matrix we need to generate an interactive heatmap as an output. This way, we can generate a library for the heatmap generation and manage the analysis-dependent aspects in a script. Furthermore, by using libraries, we leave the door open to add these functionalities to the QIIME pipeline in the future.

We decide to develop five libraries:

- make_heatmap.py: this library provides the functionality for generating heatmaps plots from a matrix. In order to generate the plots, it uses the matplotlib library [35], which is a python library to generate 2D plots and figures in a variety of formats.
The main function of this library is `plot_heatmap`. This function takes a matrix with the values to plot, a dictionary defining the horizontal and vertical headers, the base name for the output files, a directory path where the output files will be stored and a dictionary which defines ranges showing how to transform the continue matrix values to discrete values (transform dictionary). In order to define open ranges, the dictionary key can be defined using the `None` value (e.g. `<5 -> (None, 5)`). The `plot_heatmap` function allows that the matrix contains non-defined values (None) but, in this case, the dictionary should contain a key of the form (None, None). The `plot_heatmap` function generates the plot figure and it stores the figure in two different files (one in a png format and the other in eps.gz format) in the provided directory. Furthermore, this function returns the figure width and height and the image object.

- **make_html_heatmap.py**: this library provides the functionality for generating an html file with interactive heatmaps on them. It uses the `make_heatmap.py` library to generate the heatmaps. In order to make the heatmap interactive, it uses the `map` and `area` html tags and the `overlib.js` JavaScript library [36]. This library allows us to show information like tooltips.

  The function `make_html_file` is the function tailored to be called to generate the full html page with heatmaps and to place all the needed files in the correct place to successfully make the heatmap interactive. It takes a file path where to store the html file, a directory path where to place the `overlib.js` library and image files and a list where each position is a dictionary containing all the information needed to generate a heatmap (see the above point, `make_heatmap.py`). This way, the html file can contain as many heatmaps as the list length.

  One important is that the html file generated is tailored to work in Galaxy and to take profit of the extra files path mechanism. This means that the generated output will not work in a browser because the html file does not contains the paths to the images and to the `overlib.js` library, only contains the file names. In order to make the html file works in a browser, it should be moved inside the output folder provided to the `make_html_file` function.

- **newick_to_asciArt.py**: this library encapsulates the functions to generate an ASCII representation of a tree in Newick format which takes into account the branch lengths. This library provides two ways to generate the ASCII representation: a plain string (through `get_tree_by_length_string` function) and an html file with interactive colored nodes (through `make_jackknife_tree_html_file`).

  - `get_tree_by_length_string`: this function takes a PyCogent’s PhyloNode object (which represents a tree node) and it generates a string with the ASCII representation of the tree (see Figure 4. 30).
The first step taken to represent the tree is to scale the tree branches, so the max length of the tree is predefined (currently 100 characters). Once the tree is scaled, the ASCII representation is generated using the recursive function `asciiArt_length`, which takes the tree as an input and returns a list of strings (each string represents a line in the output) and an integer which shows which is the middle line of the returned list. This function generates the ASCII representation of the tree from the leaves to the root, by adding prefixes to the result of the recursive call.

- `make_jackknife_tree_html_file`: this function takes a PyCogent’s PhyloNode object, a dictionary with the contents of the jackknife support file, a dictionary which defines ranges over the support values (same structure as the `transform` dictionary defined at `make_heatmap.py`), a file path where to store the html file and a directory path where to place the `overlib.js` library. This function generates an html file with an interactive ASCII representation of the tree, where the nodes are colored by the jackknife fraction associated to the node (see Figure 4.31).

This function generates the ASCII representation in the same way as the `get_tree_by_length_string` function does: first, it scales the tree branches and then it calls the `asciiArt_length_html` function, which returns the html string with the ASCII representation of the tree. Furthermore, the `make_jackknife_tree_html_file` function moves the `overlib.js` library to the provided directory and stores the html string to the file path provided by parameter.

The `asciiArt_length_html` function implements the same algorithm as the `asciiArt_length` function but, in this case, it includes html tags to make the
nodes interactive and color them. This fact causes that the management of the strings which represents the lines of the output is more complex. For example, during the generation of the tree, the algorithm needs to remove a certain number of chars of the result of the recursive call. If it tries to remove in the same way as the `asciiArt_length` function, it may remove a char from an html tag. To avoid this, we have developed a function that removes chars from the string taking into account the html tags. Since the html tags are used to color one character, this function removes all the html tag as a single character (see Figure 4.32).

\[\text{Figure 4.32 - Html tags removal}\]

- **parse_beta_significance_output.py:** this library provides functions to parse the output files of the `beta_significance.py` script. As the output format of this script change depending on the type of test executed, this library provide two different functions:

  - **parse_beta_significance_output_pairwise:** this is a parser function in case that the type of test executed was ‘each pair of samples’. In this case the output file of `beta_diversity.py` consists of a comment line which contains the test realized, a second line which will contain the headers and one line for each comparison realized, which has the two samples compared, the raw \(p\)-value and the \(p\)-value corrected using Bonferroni separated by tabs (see Figure 4.33). The function return a dictionary where the keys are the samples compared (sample1, sample2) and the values are the \(p\)-values (raw \(p\)-value, corrected \(p\)-value) and the test executed (the comment line).

\[
\begin{array}{l}
1 \text{#unweighted unifrac significance test} \\
2 \text{ sample 1 sample 2 p-value p-value (Bonferroni corrected)} \\
3 \text{ E.FL E.DA 0.12 1.0} \\
4 \text{ E.FL O.FL 0.20 1.0} \\
5 \text{ E.FL T.UN 0.20 1.0} \\
6 \text{ E.FL R.FL 0.72 1.0} \\
7 \text{ E.FL R.PA 0.2 1.0} \\
8 \text{ E.PA O.FL 0.0 <=2.0e-02} \\
9 \text{ E.PA O.UN 0.22 1.0} \\
10 \text{ E.PA R.PA 0.28 1.0} \\
11 \text{ O.FL O.UN 0.06 0.9} \\
12 \text{ O.FL R.FL 0.0 <=2.0e-02} \\
13 \text{ O.FL R.PA 0.2 1.0} \\
14 \text{ O.UN R.FL 0.0 <=2.0e-02} \\
15 \text{ O.UN R.PA 0.2 1.0} \\
16 \text{ R.FL R.PA 0.4 1.0} \\
\end{array}
\]

\[\text{Figure 4.33 - beta_significance.py output file example: each pair of samples}\]

- **Parse_beta_significance_output_each_sample:** this is a parser function in case that the type of test executed was “each sample individually”. In this case, the output file of `beta_diversity.py` consists of a comment line which contains the test realized, a second line which contains the headers and one line for each sample, which has the sample, the raw \(p\)-value and the \(p\)-value corrected using Bonferroni separated by tabs (see Figure 4.34). The function returns a
dictionary where the keys are the samples (sample) and the values are the p-values (raw p-value, corrected p-value) and the test executed (the comment line).

```
# unweighted unifrac significance test
sample  p value p value (Bonferroni corrected)
Σ.PL  0.92  0.92
Σ.PA  0.0 <=2.0e-02
Ο.PL  0.94  0.94
Ο.Un  0.46  0.46
R.PL  0.36  0.36
R.PA  0.96  0.96
```

**Figure 4.34 - beta_significance.py output example: each sample individually**

- **parse_jackknife_support.py**: this library provides a function called `parse_jackknife_support_file` which parses the jackknife support output file of the script `tree_compare.py`, executed during the `jackknifed_beta_diversity.py` workflow script. This file consists of one comment line, a line with the number of trees considered during the process and one line for each internal node in the resulted tree, which has the node name and the jackknife fraction value (see Figure 4.35). The function returns a dictionary with two keys: ‘trees_considered’ (the value will be an integer representing the number of trees considered during the process) and ‘support_dict’ (the value will be a dictionary with the node names as keys and the fraction jackknifed as values).

```
# total support trees considered: 11
# node support is fractional - in range [0,1]
node1 0.818181818182
node0 1.0
node3 0.909090909091
node2 1.0
node5 0.272727272727
node4 0.818181818182
node7 0.363636363636
node6 0.454545454545
```

**Figure 4.35 - jackknife support file example**

### 4.4.2.2. Python scripts

In this section we will describe the Python scripts developed to successfully replicate the FastUniFrac website.

- **category_map_to_id_map_parser.py**: this script converts the category map file into a QIIME-compliant mapping file. As the QIIME mapping file structure can change in the future, this script uses the QIIME library `check_id_map` to fix the potential problems in the category map file. The user must provide two options: the category map file path (-i) and the output mapping file path (-o). This script performs the following checks:
  
  - Checks if the output file path contains spaces. It replaces them by underscores.
  - Checks if there is any blank header (header with any chars or only spaces). In this case, it raises an exception.
o Checks it there is any prohibited char in a header. Headers can only contain alphanumeric characters and the underscore. It replaces the prohibited chars by underscores.

o Check if exists the ‘SampleID’ header. If it does not exist, raise an exception.

o Check if exists the “BarcodeSequence” header. If it does not exist, creates the header and add a barcode sequence to each sample. Since the barcodes sequences must be unique, we have implemented a class called BarcodeGenerator. This class ensures that every barcode sequence generated is unique.

o Check if exists the “LinkerPrimerSequence” header. If it does not exists, creates the header and add a default linker primer sequence to each sample. In this case, all the samples must have the same linker primer sequence, so a default linker primer is defined and added (“AAAAAAAAAAAAAAAAAAAAAA”).

o Check if exists the “Description” header. If it does not exists, creates the header and add a description to each sample. The description added is the sample ID.

o Check if any field contains a prohibited char. The fields can only contain alphanumeric characters, underscore, space and ‘+’, ‘-’, '%', ',', '/', ':' and ';' characters. If replaces the prohibited chars by underscores.

• sample_id_map_to_otu_table_parser.py: this script converts the sample id map file into an OTU table file in biom format. This script uses the SparseOTUTable object from the biom project. In order to create this object, we should provide a list with the OTU IDs, a list with the sample IDs and a dictionary where the keys are tuples of two integers (the first indicates the row and the last the column) and values are the number of time the OTU (row) appears in the sample (column). The SparseOTUTable had methods to get the contents as a string in the biom format. Furthermore, this scripts checks if the sample IDs contains prohibited chars and if the output file name has spaces. It fixes these errors in the same way as category_map_to_id_map_parser.py script. The user must provide two options: the sample id map file path (-i) and the output OTU table biom file path (-o).

• copy_data.py: this is a simple script which copy the file passed through the option ‘-i’ to the file path passed through the option ‘-o’. It uses the function copyfile from the Python shutil module.

• modify_qiime_parameters.py: this script allows changing automatically the QIIME parameters file. This file is a tab delimited file used by the QIIME workflow scripts (such beta_diversity_through_plots.py or jackknifed_beta_diversity.py) to know the parameters to use in the different scripts called during the execution of the workflow. The user must provide four options to execute this script: the QIIME parameters file path to modify (-i), the output file path (-o), a comma separated list of parameters to modify (-p) and a comma separated list of new values (-n). The list of values must be in
the same order as the list of parameters. The script parses the original QIIME parameters file, makes the modifications and stores the result in the provided path.

- **make_cluster_samples_file.py**: this script uses the newick_to_asciiArt library to generate the ASCII representation of the input tree (-i option) and stores it to the provided path through the ‘-o’ option. It uses the QIIME library parse to generate the PhyloNode object required by the get_tree_by_length_string function.

- **make_pcoa_html.py**: this script is tailored to create an html file with a summary of the results generated by the beta_diversity_through_plots.py script. This workflow script generates an output directory with a predefined structure (see Figure 4. 36), so the make_pcoa_html.py script can walk through the directory structure and generate the html links for the summary html file. In order to execute this script, the user must provide the beta_diversity_through_plots.py output directory (-d option) and the file path for the generated html file (-o option).

![Figure 4. 36 - beta_diversity_through_plots.py output directory structure](image)

This way, we can take profit from the Galaxy’s extra files path mechanism to allow the user viewing the results through the Galaxy’s interface. In this case, the generated html file will be the main output dataset and the output directory of the beta_diversity_through_plots.py will be placed in the output created by the Galaxy’s extra files path mechanism. In order to allow the user navigate through the results without leaving from the Galaxy’s interface, the links of the html file are showed in an inline HTML frame embedded in the same file (<iframe> tag).

- **make_beta_significance_heatmap.py**: this script is used to generate an html file with an interactive heatmap from the output of the beta_significance.py script, in case that
the executed test was ‘each pair of samples’. This script uses the make_html_heatmap.py library to generate the html file. Thus, this script is responsible for generating the data needed by the make_html_file function.

First, it parses the beta_significance.py output file using the parse_beta_significance_output library. Then, it generates two all-versus-all matrices from the dictionary returned by the parser (a dictionary where the keys are tuples with the samples IDs compared and values are tuples with the raw p-values and Bonferroni corrected p-values): one with the raw p-values and one with the Bonferroni corrected p-values. Since these matrices are symmetric, we can plot only the upper triangle. Thus, this script modifies these matrices and puts ‘None’ values in the lower triangle and in the principal diagonal. Next, it generates the dictionary with all the information to generate the plot. This dictionary contains the transform dictionary, which tells to make_html_file function how to color the data from the matrix. In this case, the data is colored by UniFrac or p-test significance and the transform dictionary is always the same: it defines five ranges (<0.001, 0.001-0.01, 0.01-0.05, 0.05-0.1, >0.1) plus the (None, None) key, because the matrices contain ‘None’ values. Once the data are generated, it creates a list with the two dictionaries with the data needed to generate the plot and it calls the make_html_file function with this list, the file path of the html file and the path of the directory, provided by the user (options ‘-o’ and ‘--output_dir’, respectively).

- **make_unifrac_significance_each_sample.py**: this script is used to generate an html file showing the results of the UniFrac significance analyses in case that the executed test was ‘each sample individually’. The user must provide the beta_significance.py output file (-i option) and a file path for the output html file (-o option). This script uses the parse_beta_significance.py library to parse the output file provided by the user and it generates a simple html file with two tables: one showing the raw p-values and the other showing the Bonferroni corrected p-values. It also contains the legend to understand the coloring.

- **make_distance_matrix_heatmap.py**: this script is used to generate an html file with an interactive heatmap showing the output of the beta_diversity.py script. This script uses the make_html_heatmap.py library to generate the html file. Thus, this script is responsible for generating the data needed by the make_html_file function.

The first step taken is parsing the distance matrix file using the parse_distmat function from the QIIME library parse. The parse_distmat function returns a list with the headers (sample IDs) and the matrix (represented as a list of lists). Then, the make_distance_matrix_heatmap.py modifies the matrix by putting ‘None’ values in the lower triangle and in the principal diagonal. Then, it generates the dictionary with the information needed to create the plot, which includes the transform dictionary. As we said above, the distance matrix is colored by quartiles, so then this script needs to compute the quartiles of the data. This script uses the algorithm defined by Moore and McCabe [37]. Once it has computed the quartiles, generates the transform dictionary
defining the ranges as the quartiles plus the \((None, None)\) key, because the matrix has ‘None’ values. Finally, the script calls the \textit{make\_html\_file} function with a list of one element (the dictionary with the data), the file path for the html file, the file path of the html file and the path of the directory, provided by the user (options ‘-o’ and ‘--output\_dir’, respectively).

- \textit{make\_jackknife\_html.py}: this script is used to generate an html file with an interactive representation of the jackknifed tree. This script parses the jackknife support file using the \textit{parse\_jackknife\_support\_file} library, parses the Newick tree using the \textit{parse\_newick} function from the QIIME’s library \textit{parse} and generates the html file using the \textit{newick\_to\_asciiArt} library. The only thing that has to do this script is define the \textit{transform} dictionary in order to tell the \textit{make\_jackknife\_tree\_html\_file} how has to color the tree nodes. Since the nodes are colored by jackknife fraction, the \textit{transform} dictionary is always the same: the keys define five ranges \((<0.5, 0.5-0.7, 0.7-0.9, 0.9-0.999, >0.999)\) and the values define which color has to be used to color the node. In order to use this script, the user must provide the jackknife support file (-s option), the jackknifed tree file path (-t option), the file path of the output html (-o option) and the directory path where to place the \textit{overlib.js} library (--output\_dir option)

### 4.4.3. FastUniFrac integration

In this chapter we will explain the generation of the XML files for the FastUniFrac website replication using QIIME scripts. Furthermore, we will also show which configuration files from Galaxy we need to modify in order to make that Galaxy recognizes the new tools.

First, we will explain the modifications done in the Galaxy’s configuration file. Then, we will explain in detail an XML file using the \textit{Cluster samples} analysis as an example. Finally, we will go through the other analysis explaining only those aspects that are different from the \textit{Cluster samples} XML file. The full XML files are available on \textit{Annex B}.

#### 4.4.3.1. Galaxy configuration files

Under the Galaxy’s installation directories there are a lot of configuration files and folders. We are interested in two of these files and in one folder: the \textit{tool\_conf.xml} and \textit{universe\_wsgi.ini} files and the \textit{tools} folder.

The \textit{tool\_conf.xml} is the file which defines which tools are available in Galaxy and where Galaxy can find their XML files. Also, it allows defining sections under the Galaxy’s tool panel, keeping it clean and understandable. As we can see in Figure 4. 37, it has a very simple structure. First, the parent tag \texttt{<toolbox>} is defined, without any attribute. Then, for each section that we want to define, we add a \texttt{<section>} tag, defining two parameters: \texttt{name}, which defines the name of the section as it will be showed in the tool panel, and \texttt{id}, which is a unique identifier that Galaxy requires. Finally, under the \texttt{<section>} tag, we define a \texttt{<tool>} tag for each tool that we want to add, defining the parameter \texttt{file}, which contains the path to the XML file with the definition of the tool.
The universe_wsgi.ini file contains a huge amount of Galaxy's configuration variables, but we are interested in a two of them. One is the port variable, which defines which port is listened by Galaxy. In our case, in order to avoid conflicts with Apache, we define the 1234 port as the port where Galaxy has to listen. The other one is the sanitize_all_html variable. If this variable is set to True, Galaxy does not allow that the html datasets execute JavaScript code when they are showed through the Galaxy’s interface. We, therefore, set this variable to False, allowing the JavaScript execution.

The tools folder contains the XML files defining the tools. Each section defined in the tool_conf.xml file has one folder here containing the XML files of the tools. Galaxy automatically searches the XML files under the tools directory, that’s why in the tool_conf.xml we define the relative path of the XML file to this folder (e.g. fastunifrac/sample_counts.xml).

Finally, we have notice that Galaxy does not allow folders under the directory created by the extra files path mechanism. When Galaxy launches a tool, it is executed in a temporary directory and, when it finishes, Galaxy copies the results to a folder in the Galaxy’s installation folder. The problem raises during this last step, due to Galaxy expects files under the folder of the extra files path mechanism. In order to successfully show the PCoA analyses result, we need to modify the Galaxy’s source code in order to copy the folders too. Since Galaxy is open source, we can modify the source code without any problem. We have modified the function responsible of copying the result files, which is located in the /lib/galaxy/objectstore/__init__.py Galaxy's library. Originally, it uses the shutil.copy function to copy the files. We have added an if-else statement to check if the path that we want to move is a folder or a file. In case that there is a folder, we copy it using the shutil.copytree function. Otherwise, we use the default code.

```xml
<?xml version="1.0"?>
<toolbox>
  <section name="FastUnifrac" id="fastunifrac">
    <tool file="fastunifrac/cluster_samples.xml" />
    <tool file="fastunifrac/PCoA.xml" />
    <tool file="fastunifrac/p_test_significance.xml" />
    <tool file="fastunifrac/unifrac_significance.xml" />
    <tool file="fastunifrac/sample_distance_matrix.xml" />
    <tool file="fastunifrac/sample_counts.xml" />
    <tool file="fastunifrac/jackknife_sample_clusters.xml" />
  </section>
  <section name="Get Data" id="getext">
    <tool file="data_source/upload.xml" />
  </section>
</toolbox>
```

Figure 4.37 - tool_conf.xml file example
Cluster samples

We have generated the *cluster_samples.xml* file in order to provide the *Cluster samples* analysis through Galaxy using QIIME scripts. The XML file must begin with the tag `<tool>`, with two parameters defined: id and name (see Figure 4. 38). The id parameter defines the Galaxy’s internal identifier and the name parameter defines the name which the analysis will be displayed in the Galaxy’s tool panel. Moreover, we also define the version parameter, which will allow us to do versioning over the tool.

```
<tool id="cluster_samples" name="Cluster samples" version="1.0">
```

*Figure 4. 38 - cluster_samples.xml: tool tag*

Once the `<tool>` tag is defined, we describe the function by nesting the `<description>`, `<command>`, `<inputs>`, `<outputs>` and `<help>` tags. The `<description>` tag (see Figure 4. 39) contains a short description of the tool, which will be displayed next to the name on the Galaxy’s tool panel.

```
<description>
Uses the UniFrac metric to cluster the samples based on phylogenetic lineages they contain.
</description>
```

*Figure 4. 39 - cluster_samples.xml: description tag*

The `<command>` tag defines the terminal commands that should be called in order to successfully execute the tool (see Figure 4. 40). First, we should convert the *category mapping* file and the *sample id map* file to the *mapping* file and *OTU table biom* file, respectively. Here, we are using Cheetah to define the input files: the ‘$’ symbol indicates that the following string (e.g. *category_map_file*) is a Cheetah variable. This variable is replaced by the value defined by the user (e.g. the path to the *category mapping file*) when Galaxy is going to execute the tool, i.e. before calling the commands. Once the input files are in a QIIME-compliant format, we can call the *beta_diversity.py* command. Since the option passed to this command depends on the user selection through the interface, we use Cheetah to select the option: the ‘#’ symbol indicates that the following string (e.g. if *$use_abundance_weights.weights:* ) is a Cheetah statement (e.g. an if-else statement). As the Cheetah variables, these statements are replaced by python statements when Galaxy executes the commands, allowing us to select which options are passed to the *beta_diversity.py* script (in this case, we are selecting the UniFrac metric to compute). Due to the *beta_diversity.py* output file name depends on the selected UniFrac metric, we define a new variable (using the Cheetah statement #set) with the path to the output file. Next, we execute *upgma_cluster.py* to generate the Newick tree file (stored at the $cluster_sample_output path, provided by Galaxy). Finally, we execute *make_cluster_samples_file.py* in order to generate a text file with the ASCII representation of the generated tree (stored at the $ascii_art_output path, provided by Galaxy).
The <inputs> tag defines the interface showed to the user. Here, we should define a <param> tag for each option that we allow being defined by the user (see Figure 4. 41). For each <param> tag, we should define three parameters: name (which is the name of the Cheetah variable used in the <command> tag), type (which defines what kind of interface is provided to the user) and label (which contains a little description of the option, giving information to the user). In this case, we need to define three options of type data (the reference tree, the sample ID mapping file and the category mapping file). The type data tells Galaxy that this option should be a datasets. Thus, Galaxy will show a combo box with a list of the available datasets. The <conditional> tag allows us to define the interface depending on the value of some <param>. In our case, the <conditional> tag causes that the parameter with name normalized is showed or not depending on the value of the parameter with name weights. The parameter weights is a boolean parameter. Galaxy will show a check box in order to select the value of this parameter. In case that the user selects the check box (i.e. change the value to True), Galaxy will automatically change the interface to show the normalized parameter. The <when> tag defines which is the value that will activate the new interface (in this case, when the value is True).
The `<outputs>` tag defines the datasets that the tool will generate. For each dataset generated, we should define a `<data>` tag with two parameters: `format` (which is the format of the output file, e.g. `txt`) and `name` (which is the name of the Cheetah variable used in the `<command>` tag). In our case, the output of `upgma_cluster.py` is a `tre` file but, as we mentioned in Chapter 4.3 Galaxy: disadvantages, this file is based on a plain text file, so we can define it as a `txt` format. The other generated dataset is the output of `make_cluster_samples_file.py`, which is a `txt` file. When the tool is executed, Galaxy replaces the Cheetah variable with the name defined in the `<data>` tag with a new file path. When the execution finishes, Galaxy shows the new datasets in the history panel.

Finally, the `<help>` tag includes a long and detailed description of the tool (see Figure 4.43). In our case, we left this tag empty. If we include text in the `<help>` tag, Galaxy will show it below the interface generated from the `<inputs>` tag.

In the Figure 4.44 we can see the interface generated for the Cluster samples analysis.
Figure 4.44 - Cluster samples generated interface: (a) is the default interface and (b) is the interface when the weights parameter is selected.

4.4.3.3. PCoA

As we said before, starting from this chapter we are going to describe the functional differences of the XML files, since all the XML files have many aspects in common.

The main functional difference of the PCoA.xml file is the command tag (see Figure 4.45). In this command tag we are using two interesting characteristics of Galaxy: reserved variable names and the extra files path mechanism. When we are defining the call to the modify_qiime_parameters.py script, we are using the Galaxy’s reserved variable name
__root_dir__. This variable contains the path of the Galaxy’s installation directory. As we can see in the XML file, we are accessing to the qiime_parameters.txt file that is located at the /tools/fastunifrac folder, which is the directory created for the fastunifrac section, as we said above. Hence, the qiime_parameters.txt file is located with the XML files that define the tools. We are using the extra files path mechanism when we are defining the call to the beta_diversity_through_plots.py: the $output_html variable contains the path to the html file generated by the make_pcoa_html.py script which, as we explained above, is the main output dataset. Then, we add the .files_path suffix to the variable name and we get access to the extra files path folder. Therefore, the output of beta_diversity_through_plots.py is stored in that folder, since we are passing the –o option as $output_html.files_path.

```python
<command interpreter="python">
    bin/category_map_to_id_map_parser.py -i $category_map_file -o fasting_map.txt;
    sample_id_map_to_otu_table_parser.py -i $sample_id_file -o otu_table.biom;
    modify_qiime_parameters.py -i __root_dir__/tools/fastunifrac/qiime_parameters.txt -o param_file.txt -p beta_diversity:metrics,make_2d_plots:scree
    #if $use_abundance_weights.weights:
    #if $use_abundance_weights.normalized:
        -n weighted_normalized_unifrac,True;
    #else:
        -n weighted_unifrac,True;
    #end if
    #else:
    -n unweighted_unifrac,True;
    #end if
    beta_diversity_through_plots.py -i otu_table.biom -m fasting_map.txt -t $tree_file -o $output_html.files_path -p param_file.txt;
    make_pcoa_html.py -o $output_html -d $output_html.files_path;
</command>
```

Figure 4. 45 - PCoA.xml: command tag

In the Figure 4. 46 we can see the generated interface for the PCoA analysis.

Figure 4. 46 - PCoA generated interface
4.4.3.4. P test significance

The p_test_significance.xml introduce a new type of parameter: select (see Figure 4.47). This type of parameter is a parameter where the user has different predefined options to select. These options are defined using the <option> tag. The value parameter of the <option> tag tells Galaxy which value has to use in the command when it replace the value of the Cheetah variable. For example, although the user selects the “All samples together” value, when Galaxy has to replace the value of the variable Type_of_test, it will use the “all_together” value, because it is the value defined in the parameter value of the <option> tag.

Figure 4.47 - p_test_significance.xml: select parameter type

As we said in Chapter 4.4.1.4 P test significance, the user can execute the test in two ways, getting different outputs: in case of ‘all samples together’ is getting a single p-value and in case of ‘each pair of samples’ is getting a heatmap. This means that depending on the input parameters values, the output will be different. First, we have to know when we should call the make_beta_significance_heatmap.py. Galaxy allows us to check the value of the select parameter values, enabling us to decide if we should call a script or not. As we can see in Figure 4.48, in case of select parameters we need to use the str function to check their value.

Figure 4.48 - p_test_significance.xml: check select value
From the last piece of XML file we can deduce that the $output\_html$ file is not always generated. It means that Galaxy will show an empty dataset when this file is not generated. How we can avoid this situation? We can use the $<$filter$>$ tag (see Figure 4. 49). The $<$filter$>$ tag allows us to define a boolean expression and Galaxy will generate the output dataset only if the expression evaluates true. It is important to note that in the boolean expression we do not use the ‘$’ symbol to access the variables, because this piece of code is not Cheetah. Galaxy deals with it as a pure Python code.

```xml
<outputs>
  <data format="txt" name="p_test_output"/>
  <data format="html" name="output_html">
    <filter>(type_of_test == "each_pair")</filter>
  </data>
</outputs>
```

*Figure 4. 49 - p_test_significance.xml: filter tag*

In the *Figure 4. 50* we can see the generated interface for the *p-test significance* analysis.
4.4.3.5. UniFrac significance

In the unifrac_significance.xml file we have an example of an if-else-if Cheetah statement (see Figure 4.51). We can notice the power of using Cheetah, since it allow us to decide which scripts call depending on the user’s input values.

```python
<command interpreter="python">
    ...
    #if str($type_of_test) == "each_pair":
        make_beta_significance_heatmap.py -i $unifrac_output -o $output_html --output_dir=$output_html.files_path
    #else if str($type_of_test) == "each_sample":
        make_unifrac_significance_each_sample_html.py -i $unifrac_output -o $output_html_indiv
    #end if
</command>
```

**Figure 4.51 - unifrac_significance.xml: if-else-if Cheetah statement**

Here we get the same problem as in the p_test_significance.xml file: we can get empty datasets. Since there is no limit with the number of output datasets, we can define a `<data>` tag for each possible output dataset and define a `<filter>` tag to whose datasets that are not always generated (see Figure 4.52)

```xml
<outputs>
    <data format="txt" name="unifrac_output"/>
    <data format="html" name="output_html">
        <filter>(type_of_test == "each_pair")</filter>
    </data>
    <data format="html" name="output_html_indiv">
        <filter>(type_of_test == "each_sample")</filter>
    </data>
</outputs>
```

**Figure 4.52 - unifrac_significance.xml: multiple filter tags**

In the Figure 4.53 we can see the generated interface for the unifrac significance analysis.
4.4.3.6. Sample counts

The sample_counts.xml file has an example of a different use of the parameter type boolean (see Figure 4. 54). In the other XML files we were using the boolean parameters to know which value we need to use to call the tool. In this case, we use the boolean parameter to know if we need to pass a parameter or not. As we can notice, we can use boolean operators in the if Cheetah statement. Here, we are using the not operator, but the and and or operators are also available.

```python
<command interpreter="python">
  bin/category_map_to_id_map_parser.py -i $category_map_file -o fasting_map.txt;
  sample_id_map_to_otu_table_parser.py -i $sample_id_file -o otu_table.biom;
  per_library_stats.py -i otu_table.biom -o $sample_counts_output -m fasting_map.txt
  #if not $weights:
  --otu_counts
  #end if
</command>
```

Figure 4. 54 - sample_counts.xml: boolean parameter usage

In the Figure 4. 55 we can see the generated interface for the sample counts analysis.
4.4.3.7. Sample distance matrix

We can see a usage example of the copy_data.py script in the sample_distance_matrix.xml file (see Figure 4. 56). In order to generate the sample distance matrix, we need to execute the beta_diversity.py script, which generates a directory as an output with only one file. In the same as we have done in the Cluster samples analysis, we keep track of the output file path using the set Cheetah statement and, then, we can pass this path to the copy_file.py script. This way, we can allow Galaxy to provide the distance matrix as a dataset.

```python
<command interpreter="python">
  bin/category_map_to_id_map_parser.py -i $category_map_file -o fasting_map.txt;
  sample_id_map_to_otu_table_parser.py -i $sample_id_file -o otu_table.biom;
  beta_diversity.py -i otu_table.biom -t $tree_file -o ./output
  #if $use_abundance_weights.weights:
  #if $use_abundance_weights.normalized:
    -m weighted_normalized_unifrac;
    #set filename = 'output/weighted_normalized_unifrac_otu_table.txt'
  #else:
    -m weighted_unifrac;
    #set filename = 'output/weighted_unifrac_otu_table.txt'
  #end if
  #else:
    -m unweighted_unifrac;
    #set filename = 'output/unweighted_unifrac_otu_table.txt'
  #end if
  copy_data.py -i $filename -o $output_distance_matrix;
  make_distance_matrix_heatmap.py -i $output_distance_matrix -o $output_html --
  output_dir=$output_html.files_path
</command>
```

Figure 4. 56 - sample_distance_matrix.xml: copy_data.py example
In the Figure 4. 57 we can see the generated interface for the sample distance matrix analysis.

![Sample distance matrix generated interface](image)

**Figure 4. 57 - Sample distance matrix generated interface**

### 4.4.3.8. Jackknife cluster samples

The `jackknife_sample_clusters.xml` file is the most complete XML file because it uses almost all the functionalities described above. It uses the `__$root_dir__$` variable to access to the `qiime_parameters.txt` file, the `set` Cheetah statement to keep track of the output directory path, the `copy_data.py` script to allow Galaxy keeping track of the output, the `select` parameter type, among others. Furthermore, it uses a new parameter type that we did not use before: `integer` (see Figure 4. 58). The `integer` parameter has the difference from the other types that we are required to define a default value through the `value` parameter from the `<param>` tag. The interface generated for the `integer` parameter is like a text box, but it only allows writing numbers.

```xml
<inputs>
  ...
  <param name="num_sec" type="integer" label="Number of sequences to keep" value="12"/>
  ...
</inputs>
```

*Figure 4. 58 - jackknife_sample_clusters.xml: integer parameter type*

In the Figure 4. 59 we can see the generated interface for the sample jackknife sample clusters analysis.
4.4.4. Deploy package

At this point we have described how we replicated the FastUniFrac website using QIIME scripts fully integrated on Galaxy. Once we have developed the needed libraries, scripts and XML files, we have developed a deploy package that given a machine with a clean QIIME and Galaxy installations, it configures Galaxy to provide the FastUniFrac website.

The deploy package consists in a Python script (called integrate.py) and a folder which contains the tool_conf.xml and universe_wsgi.ini configuration files, the /lib/galaxy/objectstore/__init__.py modified Galaxy’s library and a .tar.gz file with all the developed Python libraries and scripts and the tool’s XML configuration files.

The pseudo-code of the integrate.py script is available on Annex C. In order to execute this script, the user must provide the Galaxy installation folder through the –i option and the bash configuration file path through the –b option. Optionally, the user can pass the –tcshe option if he uses the tcshe shell. Summing up, this script replaces the original Galaxy’s configuration files and objectstore library by the files included with the package and places the content of the .tar.gz file under the tools folder of the Galaxy’s installation directory. Finally, it modifies the bash configuration file by adding new folders to the PATH and PYTHONPATH environment variables in order to be able to execute the scripts and libraries developed.

4.5. QIIME automated integration

Once we have replicated the FastUniFrac website, we already learned the logic behind the XML files and which configuration files we need to modify in order to integrate new tools on Galaxy. This information is essential to ensure the complete automation of the integration process, since our main goal is to develop a package which, given a clean installation of Galaxy and QIIME, it automatically generates the XML files and changes the Galaxy’ configuration files in order to allow Galaxy recognize the QIIME tools.
4.5.1. QIIME modifications

As we stated on Chapter 4.2.2 QIIME-specific GUI requirements, we will extract the needed information from the script_info dictionary present in all QIIME scripts. Unfortunately, not all the QIIME scripts have the script_info updated or with the correct values. For example, there are some scripts that the option type is not defined. In this case, the PyCogent’s module option_parsing defines its type as string by default. Thus, the script is responsible to perform the needed input checking. Therefore, the first step to carry out in order to automate the process is to complete and correct the script_info dictionary of each QIIME script.

During this process, we have noticed that we need a new option type which will make possible a better integration of QIIME on Galaxy: ‘multiple_choice’. The option type ‘choice’ present in the option_parsing module does not permit multiple choices. This fact causes that some options are defined as strings because they need to allow multiple values. One example is the --metrics option of the beta_diversity.py script. The user can provide a comma separated list of the available metrics. Creating this new type, the Galaxy interface can create a combo box with multiple selection (a <param> tag of type select setting the parameter multiple to True). This way, the user does not need to remember the available metrics on QIIME, he just get a list and can select the desired options.

In order to add this new type, we need to modify the option_parsing module. Concretely, we will modify the CogentOption class, which defines and checks the option types. For add a new type in the CogentOption class we need to modify two class attributes: TYPES and TYPE_CHECKER. The attribute TYPES is a tuple of strings which contains the available option types, so then we add the ‘multiple_choice’ string to allow this being a new type. The attribute TYPE_CHECKER is a dictionary where the keys are the different types and the values are functions that checks if the value provided by the user is of the defined type. Hence, we have implemented a function that checks if the values provided by the user are in the list of predefined values. In the case of an option of type ‘choice’, the list of predefined values are defined in an attribute called choices. We are not allowed to use this attribute to store the list of predefined values because the optparse module checks that this attribute is set if and only if the type is ‘choice’. Therefore, we are required to create our own attribute to store the list of predefined values. That can be done by adding the new attribute to an existing attribute: ATTRS. The attribute ATTRS is a list which defines all the attributes needed by the different types of options. We decide to call the new attribute ‘mchoices’. Creating a new attribute requires to add a new method to the list CHEK_METHODS. This list contains methods that check that the attributes defined in ATTRS are set when the type requires it (e.g. there is a method that checks the attribute choices is set if and only if the type is choices). Accordingly, we have defined a method which checks that the ‘mchoices’ attribute is only set if the type is ‘multiple_choices’; and we added it to the CHECK_METHODS list.
4.5.2. Design

At this point, we have the QIIME scripts updated, so we can design the automation of the integration process.

The current QIIME pipeline consists of 131 python scripts but not all scripts should be integrated on Galaxy. The reason is because these scripts are not executed by the user: are executed from other QIIME scripts. For example, the `denoiser_worker.py` script performs part of the `denoise` process and it is called by the `denoise_wrapper.py` script. Hence, we need some mechanism to tell the automation process which script to integrate and which not. This mechanism will be a simple configuration file showing which script should be integrated on Galaxy.

Furthermore, due to the changes applied to the QIIME scripts and some Galaxy’s characteristics, some options of the scripts are not needed to be defined in the GUI. For example, the `beta_diversity.py` script has the `--show_metrics` option. When this option is provided, the script does not execute any analysis, it only prints the available metrics through the standard output. Since we added a new option type that allows showing all the metrics in the GUI and the user only needs to select the desired options, the `--show_metrics` option is not necessary. Hence, we need to tell the automation process those options that are not needed in the GUI. We will use the configuration file in order to provide this information, too.

Once we have defined the configuration file, we can design the automation process as follows (see Figure 4. 60). First, the user should provide the path of configuration file, the path of the QIIME’s scripts directory and the path of the Galaxy’s installation directory. Then, the process parses the configuration file to know the scripts needed to be integrated and the options which are not needed. The next step in the process is to traverse the QIIME scripts directory and, for each script, check if it is in the configuration file. If it is in the configuration file, then the XML file for this script is generated. Once all the QIIME scripts are processed, the next task is to modify the Galaxy’s `tool_conf.xml` configuration file. Finally, the process updates the environment variables in order to ensure the proper functioning of the GUI.

The XML generation process is a complex process which is defined as four sub-processes (see Figure 4. 60). First, it extracts all the information needed for the XML file from the `script_info` dictionary of the QIIME script. Second, it generates the command line string needed to successfully call the script. Third, it generates the XML tag for the input/output options. Finally, it completes the XML file and writes it in the correct path.
4.5.3. Implementation

As we can see in the design described in the previous chapter, there are two main processes: generate an XML file and manage the scripts to be processed and Galaxy’s configuration files. Hence, we decided to develop two Python libraries, one for each main process: `galaxy_integration.py` and `xml_generator.py` (see Figure 4.61). Furthermore, since the XML generation process is more complex, we have implemented a group of classes which encapsulate the different processes. The extraction of the data is done by two classes: the `ScriptInfo` class, which represents all the information needed to generate the XML file, and the `OptionInfo` class, which represents all the information about the options needed to generate the command line string and the parameter attributes. The generation of the command line string is done by a class called `CommandGenerator`, which encapsulates all the functions needed to generate the command line string. Finally, the generation of the input/output XML tags is encapsulated in the `XmlOptionsAttributesGenerator` class.

As we stated on Chapter 4.3 Galaxy: disadvantages, we will compress the output directories in a `.tgz` file. In order to do that, we have developed another library: `tgz_manager.py`. This library provides functions to compress and decompress `.tgz` files.

Next, we are going to describe in detailed the configuration file structure and the developed libraries. The pseudo-code is available in Annex D.
4.5.3.1. **Configuration file**

The configuration file is a plain text file which indicates the scripts to integrate and the options of those scripts that are not necessary in the GUI.

The user must define a line in the configuration file for each QIIME script that should be integrated. Each line is composed by the name of the script (e.g. `beta_diversity.py`) followed by a tab character. The tab character can be followed for a comma separated list of those options that should not be included in the GUI (e.g. `force,print_only`).

As we said above, the tools can be showed grouped by a section in the Galaxy’s tool panel. In order to define these sections in the configuration file, the user has to define a line which starts with a tab character followed by the name of the section. Hence, all the scripts defined in the following lines will be grouped by this section.

Finally, the user can add comments to the configuration file by starting the line with a hash character (`#`). In Figure 4. 62 we have an example of configuration file.
4.5.3.2. galaxy_integration.py

The `galaxy_integration.py` library manages the whole integration process. The `integrate` function is the function which should be called to start the integration process. This function takes five parameters: a string with the path of the scripts directory, a string with the path of the Galaxy’s installation directory, a string with the path of the configuration file, a boolean to indicate if the `tool_conf.xml` file should be updated (`True`) or replaced (`False`) and a string with the path where to store the log file. If the string with the path where to store the log file is not provided, it is stored by default in the scripts directory.

The `integrate` function starts by parsing the configuration file. Then, for each section defined in the configuration file, this script creates a new directory named as the section under the `tools` folder of the Galaxy’s installation directory. Depending on the value of the input boolean, it parses the `tool_conf.xml` file or creates a new one. Next, it adds the scripts directory to the python path in order to allow the `xml_generator.py` library importing the scripts. After, it traverses the scripts directory and for each file checks it is a python script and, if it is, it check is there is in the configuration file. If it is in the configuration file, it calls the `make_xml` function of the `xml_generator.py` library to create the XML file. Once all the XML files are generated, it updates the `tool_conf.xml` file. Finally, it updates the environment by adding the `GALAXY_HOME` variable to the `.bashrc` file.

4.5.3.3. xml_generator.py

The `xml_generator.py` library is tailored to encapsulate all the classes and methods needed to generate an XML file from a QIIME script. It contains four classes (`ScriptInfo`, `OptionInfo`, `CommandGenerator` and `XmlOptionsAttributesGenerator`) and two functions (`make_xml`, `generate_xml_string`).

The `make_xml` function manages all the steps during the XML generation process. It takes three parameters: a string with the script path, a string with the path to the directory where to place the XML file and a string with a comma-separated list of the options to not to include in the XML file. First, it gets the base name of the script and realizes an import from the script. This way, we have access to the `script_info` dictionary as a member of the module. Then, it creates an instance of `ScriptInfo` class. At this point, we have all the information of the script in

```python
# Configuration file for QIIME automated integration on Galaxy

AutoQime
add_qiime_labels.py
add_taxa.py
adjust_seq_orientation.py
align_seqs.py blast_db
alpha_diversity.py show_metrics
alpha_rarefaction.py force,print_only
ampliconnoise.py force,print_only
assign_taxonomy.py blast_db
beta_diversity.py show_metrics
beta_diversity_through_plots.py force,print_only
beta_significance.py
blast_wrapper.py
```

Figure 4. 62 - Configuration file example
this instance. Next, it removes the options from the ScriptInfo class that are not to be included in the XML file. After, it calls the generate_xml_string function to generate a string which contains all the XML string. Finally, it stores the XML string in a file called as the script name plus .xml placed at the output directory provided by parameter.

The generate_xml_string function is tailored to generate a string which contains all the XML file contents. In order to do that, it uses the Python module xml.dom.minidom which is a DOM (Document Object Model) light-weight implementation. This function takes as a parameter an instance of ScriptInfo. The first step that it takes is the generation of an instance of xml.dom.minidom.Document class. Then, it defines the tool and description tags with the needed attributes. Next, it creates an instance of CommandGenerator and adds the tag command with the string resulted from the CommandGenerator instance. Once the command line string is generated, it creates the inputs and output tags. Then, it instantiates the XmlOptionsAttributesGenerator and fulfills the input/output parameters. Finally, it adds the help tag with the proper value and returns the full XML string.

The OptionInfo class and the ScriptInfo class contain all the information needed for a given script. The OptionInfo class represents an option of the script. It is formed by the name and type of the option, the command line string representation, the default value, the list of available options in case of being choice or multiple_choice type and the file format in case of being new_path, new_filepath or new_dirpath type. The ScriptInfo class models a script. It contains the id for the tool, the name, the version, the description, a list of required options (represented by a list of OptionInfo), a list of optional options (represented by list of OptionInfo), the help string of the tool and the command line string to invoke the tool without parameters.

The CommandGenerator class encapsulates all the methods related with the command line string generation. To instantiate this class, the user must provide a ScriptInfo object. Then, by calling the update method, the command line string is generated. Finally, the user can get it by accessing at the command_text attribute. Since the command line string depends on the option types, this class has a function for defining the command line for each option type. One special case is the existing_filepaths option type. As we stated before, Galaxy provides an object of type list, but QIIME expects a comma-separated list in a string. Thus, this class adds a Cheetah code in the command line string in order to convert the list object in a comma separated list in a string (see Figure 4.63).

```python
#def list_dict_to_string(list_dict):
    #set $file_list = list_dict[0]['additional_input'].__getattr__('file_name')
    #for d in list_dict[1:]:
        #set $file_list = $file_list + ',' + d['additional_input'].__getattr__('file_name')
    #end for
    #return $file_list
#end def
```

Figure 4.63 - Cheetah code to convert a list object in a comma separated list string
Furthermore, if the script has a directory as an input, this class adds a call to `uncompress_tgz.py` command in order to decompress the input tgz file provided by Galaxy. Also, if the output of the script is a directory, it adds the call to the `compress_tgz.py` file to generate a tgz file from the output directory, allowing Galaxy to keep track of it.

The `XmlOptionsAttributesGenerator` class encapsulates all the methods needed to define the attributes of the input and output tags. To create a new instance of this class, the user must provide a `ScriptInfo` object, a `xml.dom.minidom.Document` object, and the `xml.dom.minidom.Node` objects that represents the `inputs` and `outputs` tags from the provided `Document` object. Next, by calling the `update` method, this class updates the `inputs` and `outputs` tags of the XML document. The input and output attributes also is dependent of the option type, so this class has one method defined for each type of option, in order to generate the right XML file.

### 4.5.3.4. `tgz_manager.py`

The `tgz_manager.py` is a simple library that provides two methods: `compress_tgz` and `extract_from_tgz`. The `compress_tgz` function takes a path and a string and generates a tgz file containing the given path (could be a file or a directory) named as the given string. The `extract_from_tgz` takes a path of a tgz file and a path and extracts the contents of the tgz file in the given path. In case of the contents of the tgz file was a directory, it checks if it has more than one file. If the directory only contains one file, the function renames it as the given path. This way, we can reduce the amount of tgz files managed by Galaxy, since some scripts generates a directory as an output if the input is a directory and generates a file if the input is a file.

### 4.5.3.5. Scripts

In the previous sections we have defined the developed libraries, but in order to use these libraries we have developed some scripts:

- **`integrate_on_galaxy.py`:** this script is the script to be called to automatically integrate QIIME on Galaxy. The user has to provide the scripts directory (-i option), the Galaxy’s installation directory (-g option) and the path of the configuration file (-c option). Optionally, the user can provide the `--update_tool_conf` option, which is used to indicate that the user does not want to override the `tool_conf.xml` file; and the `-l` option, which is a file path where to store the log file. This script parses the options and calls the `integrate` function of the `galaxy_integration.py` library.

- **`make_xml.py`:** this script is used to generate a XML file for a given script. The user should provide the script path (-i option) and the directory path where to store the xml file (-o option). Optionally, the user can provide the `-r` option with a comma separated list with the names of those options of the given script that does not to be included in the XML file. This script parses the options and calls the `make_xml` function of the `xml_generator.py` library. In order to successfully generate the XML file, this script
should be called from the directory where the script is placed or the directory should be present in the PYTHONPATH variable.

- **compress_path.py**: this script is used to compress a path in a .tgz file. The user should provide the path (of a directory or file) to compress (-i option) and the file path of the resulted .tgz file (-o option). This script parses the outputs and calls the compress_to_tgz function of the tgz_manager.py library.

- **uncompress_path.py**: this script is used to extract the contents of a .tgz file. The user should provide the file path of the .tgz file (-i option) and the path to store the contents. This script parses the options and, then, checks if the input file is a .tgz file. If the input path is a .tgz file, it calls the extract_from_tgz function of the tgz_manager.py library. Otherwise, it copies the input file to the output path. This check is necessary for those QIIME scripts that have an option of type existing_path, since Galaxy can provide either a directory in a .tgz file or a single file.

### 4.5.4. Testing

In order to test all the developed software, we have used unit tests. There is one test for each developed library: test_galaxy_integration.py, test_xml_generator.py and test_tgz_manager.py. To perform the tests, for each function we have defined a group of inputs which generates a known output. Hence, we compare the output of the function with the expected output. If the test fails, we know which function is that are not well implemented, since the tests are done at function level.

The test_tgz_manager.py provides a .tgz file to the extract_from_tgz function and checks if the output file exists in the proper location. To test the compress_to_tgz function, it provides a path of a directory and checks that the .tgz file is created in the proper location.

The test_galaxy_integration.py checks each function of the galaxy_integration.py library as follows:

- **parse_config_file**: we execute two tests on this function. First, we provide a well formatted configuration file and we check that the output is as expected. For the second test, we provide a bad formatted configuration file and we check that the function raises an exception.

- **create_dirs**: we create a directory structure similar to the Galaxy’s installation folder and we execute the function. Then, we check if it creates the directories in the right place. We execute another check by creating a directory structure different from the Galaxy’s installation folder and we check if the function raises an exception.

- **get_galaxy_tool_conf_file**: we provide a tool_conf.xml and we check if the read XML string is the same as expected. Also, we check that if we want to overwrite the tool_conf.xml file, the XML string returned is the default one.
• **get_section_node**: we provide an `xml.dom.minidom.Document` object which represents a `tool_conf.xml` file and we try to get some section nodes from this document. Then, we check if it returns the correct ones or `None` if the section does not exist.

• **exist_script_in_section**: we provide an `xml.dom.minidom.Document` object which represents a `tool_conf.xml` file and we ask for some scripts. Then, we check if the output is `True` or `False` as expected.

• **add_section_to_xml**: we provide an `xml.dom.minidom.Document` object which represents a `tool_conf.xml` file and we use this function to add some new sections. Then, we check that the string resulted is the same as expected.

• **update_tool_conf_xml**: we provide an `xml.dom.minidom.Document` object which represents a `tool_conf.xml` and a dictionary where the keys are sections and the values are lists of scripts and we call this function to add the `Document` object. Then, we check that the resulted string is the same as expected.

• **integrate**: we create a directory structure similar to the Galaxy’s installation folder, a directory with some python scripts and a configuration file. Then, we call this function and we check that log file is created in the proper location.

The `test_xml_generator.py` checks each class and functions of `xml_generator.py` library as follows:

• **OptionInfo**: for each type of option we create an instance of this class and we check if the attributes of the class have the right values. Also, we execute the methods and we check if the output is as expected.

• **ScriptInfo**: we create an instance of this class and we check if the attributes of the class have the right values. Furthermore, we execute the methods of this class and we check if the output is as expected.

• **CommandGenerator**: for each method defined for each different option type, we create a `ScriptInfo` object with a required and optional option of this type and we check that the command line generated is correct. Also, we create an example `ScriptInfo` object with few options of different types and we check that the `update` method generates the full correct command line.

• **XmlOptionsAttributesGenerator**: for each method defined for each different option type, we create a `ScriptInfo` object with a required and optional option of this type and an `xml.dom.minidom.Document` object with the needed tags and we check that the `Document` object is updated in the correct way. Also, we create an example `ScriptInfo` with few options of different types and we check that the `update` method updates correctly the `Document` object.
- **generate_xml_string**: we create an example `ScriptInfo` object and we check that the output XML string is the same as the expected string.

- **make_xml**: we create an example Python script and we check that the XML file is stored in the correct path.
5. Galaxy on the Cloud: Galaxy CloudMan

Galaxy can leverage cloud computing resources by using Galaxy CloudMan [30]. Galaxy CloudMan is a project that allows executing Galaxy on the cloud by creating a cluster using Amazon EC2 instances [38]. The main advantage of using Galaxy CloudMan is that it hides the low-level details, giving the opportunity of using it to researches without informatics expertise. The cloud cluster is managed through a web interface (see Figure 5.1).

![Figure 5.1 - Galaxy CloudMan web interface](image)

The user only needs to perform two steps to get a cloud cluster running. First, the user should start the cluster master node through the AWS (Amazon Web Services) Management console. Second, the user accesses to the master instance through a web browser and the Galaxy CloudMan management console page will be showed. Through the management console page, the user can exploit the elasticity of the cloud. The user can enlarge the cloud cluster and the capacity of the persistent data volume (Elastic Block Storage volumes [39]) at running time.

To illustrate how Galaxy CloudMan simplifies the cloud resources management, we will describe the process that it executes when the user wants to enlarge the capacity of the EBS volumes of the cluster. While the user selects the new size of the EBS volume through a simple web interface, Galaxy CloudMan performs the following steps:

1. Ensures that the EBS volume is not used by stopping all the services that are using the volume.
2. Detaches the EBS volume from the master node.
3. Creates a snapshot from the EBS volume. A snapshot is a copy from the data of the EBS volumes.
4. Creates a new EBS volume by instantiating the snapshot created in the previous step with the size defined by the user.
5. Attaches the new EBS volume to the master node.
6. Creates the file system in the new EBS volume.
7. Re-starts the previous stopped services.
Once we have seen that Galaxy CloudMan is the perfect solution for those researches that needs cloud computing resources but do not have computer expertise, we can apply this solution to the QIIME pipeline. How can we do that?

Galaxy CloudMan is provided through an AMI (Amazon Machine Instance [40]). In order to provide QIIME with Galaxy CloudMan, we will create a new AMI with QIIME and Galaxy CloudMan installed.

5.1. QIIME + Galaxy CloudMan AMI

The new AMI will be created from the one provided with Galaxy CloudMan. This means that we will take the Galaxy CloudMan AMI and we will install QIIME on it. Then, we will use the developed software to integrate QIIME on Galaxy. Finally, we will generate a new AMI from the current configuration through the Galaxy CloudMan management console.

But, what is an AMI? An AMI (Amazon Machine Image) is a clone of an instance. It contains the hardware definition and the software environment. This provides us the capacity of lunch as many instances as we want ensuring that all the instances will have the same setup. The advantage of using AMIs is that they can be published, so any user can lunch an instance with the published AMI. That’s why we will create a new AMI with QIIME and Galaxy CloudMan. This way, we will free the user to perform the needed steps to configure an instance with QIIME and Galaxy.

Before creating the new AMI, we should log into the EC2 AWS management console [41] and perform the following two steps:

- First, we need to create a key pair. To create a new key pair we select Key pairs in the Navigation menu on the left and then we click on the Create key pair button on the top of the window (see Figure 5. 2). A pop-up window will appear asking us for the key pair name. We introduce the key pair name and the download of the private key file (.pem file) will start. The reason why we need to create a key pair is because EC2 instances do not have password to log in by default. Therefore, key pairs will allow us to log into our EC2 instances through the command line.
Second, we need to create a Security Group. By default, EC2 instances do not have any port open, so there is no way to connect to them. When we launch an instance, we should apply to it a Security Group. The Security Group defines TCP rules to open the instance’s ports. To create a new Security Group we select the Security Groups option on the Navigation menu and then click the Create Security Group button on the top of the window. A pop-up window will appear asking us for the group name and a description of the group. The description field is very useful to know the characteristics of the group when we are launching an instance. Once we provide these fields, the Security Group is created. Then, we have to define which ports are open when we apply this security group to an instance. To add new rules, we select the Inbound tab. After, for each rule that we want to add, we select the rule type from the drop-down menu, define the different fields of the rule and then click on the Add rule button. Finally, to store the changes, we click on the Apply Rule Changes button. In Figure 5.3 we can see the Security Group creation process.
In our case we should apply the following rules:

- An SSH rule, which will open the TCP port 22, allowing us to log into the instance through ssh.
- An HTTP rule, which will open the TCP port 80, allowing us connect to the instance using a web browser.
- A custom TCP rule opening the TCP port 42284. This port is used by the cloud controller web interface.
- A custom TCP rule opening the TCP ports 20 and 21, enabling FTP file transfers.
- A custom TCP rule opening the TCP ports range 30000-30100, enabling passive FTP file transfers.
- An All TCP rule which will open all the TCP ports. In this case, we define the filed Source with the security group ID. This means that all the instances from this security group can communicate with each other.

Once we have performed these two steps, we can create our AMI. First, we need to launch an instance with the Galaxy CloudMan AMI. To create a new instance from an existing AMI, we select the AMI’s option in the Navigation menu. Then, we select the Public Images option from the drop-down menu and we type galaxy on the search area. Next, we select the galaxy-cloudman-2011-03-22 AMI and we click on the Launch button (see Figure 5.4).
A pop-up wizard will appear. This wizard will guide us throw the instance launching process, allowing us to configure the instance. In the first wizard window we can select the number of instances to be launched, the instance type and the availability zone to launch the instance (see Figure 5. 5). In our case, we only need one instance: the master node of the cluster. In case that we need more instances, we will upgrade the cluster through the Galaxy CloudMan management console page. The instance type depends on the hardware requirements. Since we will install QIIME, we will use the medium extra-large instance, which give us 8 cores and 15GB of memory. This way, we can ensure that we can deploy QIIME in a reasonable amount of time and we can execute all the QIIME tests, to check that the installation is successful. We can select any availability zone, but we had to remember the selection in order to be able to launch the cluster again once it is stopped. Once we have defined all the fields, we click on the continue button.
The second wizard window allows us to define advance instance option, such the Kernel ID or the RAM disk ID (see Figure 5. 6). The important field of this window is the User Data field. In this field, we should define the cluster name, a password to access to the Galaxy CloudMan management console, our AWS access key and our AWS secret key. This data is important because this data will allow Galaxy CloudMan creating and stopping EC2 instances and, therefore, modify the cluster size.

The third wizard window allows us to adding tags to our instance. We will ignore this step and click the Continue button. The next wizard window asks us for a key pair (see Figure 5. 7). Here, we will select the key pair created before. Then, we click on the Continue button.
The fifth wizard window allows us to select which security group we want to apply on the instance (see Figure 5.8). We will select the security group created before and click on the Continue button.

Finally, the last wizard window shows us the defined instance configuration and allows us to perform modification on it (see Figure 5.9). If everything is correct, we can click on the Launch button and the instance will be launched.
Once we click the launch button, we will return to the Instances page. Here, we should wait until the launched instance is running (see Figure 5. 10).

When the instance is ready, we login into it through ssh using the command showed in Figure 5. 11. Once we are logged in, we will install QIIME. The AMI with Galaxy CloudMan comes with an Ubuntu 10.04 image. The QIIME team provides a package called `app-deploy` which automatically installs QIIME and its dependencies in a Ubuntu system [42]. Unfortunately, the `app-deploy` package is configured to run in an Ubuntu 11.10 and higher. As a consequence, we need to perform some modifications.
First, before using the app-deploy package, we need to install some libraries and applications from the repositories, as stated in the QIIME Ubuntu installation page. Since the Ubuntu version of the image is a bit old, we need to add a few repositories manually (chris-lea/zeromq, chris-lea/libpgm and internetbroadcasting/mysql55) and install extra libraries (libpgm-5.1-0, libzmq1, libmysqlclient-dev). The used commands are shown in Figure 5. 12. Once these libraries have been installed, we can continue installing the libraries indicated in the QIIME Ubuntu installation page.

```bash
sudo add-apt-repository ppa:chris-lea/zeromq
sudo add-apt-repository ppa:chris-lea/libpgm
sudo add-apt-repository ppa:internetbroadcasting/mysql55
sudo apt-get update
sudo apt-get install libpgm-5.1-0 libzmq1 libmysqlclient-dev
```

Figure 5. 12 - Commands for add repositories and install extra libraries

Previously to run the app-deploy package we need to modify the qiime_1.5.0_repository.conf configuration file of the app-deploy package. This file contains the information of how to install the dependencies. Specifically, we need to modify the python installation configuration by adding the definition of the LD_LIBRARY_PATH through the set-environment-variables-deploypath option (see Figure 5. 13). This definition will allow to the app-deploy to find the python libraries during the deploy process. Without this definition, most of the other dependencies will fail during the deployment.

```python
[python]
version: 2.7.1
build-type: autoconf
release-file-name: Python-2.7.1.tgz
release-location: ftp://thebeast.colorado.edu/pub/QIIME-v1.5.0-
dependencies/Python-2.7.1.tgz
relative-directory-add-to-path: bin
autoconf-configure-options: --enable-shared --with-zlib=/usr/include --enable-
unicode=ucs2 --enable-unicode=ucs4
set-environment-variables-deploypath: LD_LIBRARY_PATH=lib/
```

Figure 5. 13 - qiime_1.5.0_repository.conf file modification

At this point, we can launch the app-deploy.py script to start the deployment of QIIME and its dependencies. When the app-deploy.py finished we already have QIIME installed on the instance. The next step will be to integrate QIIME on Galaxy. In order to integrate QIIME on Galaxy we will use the developed software. As a first challenge, we are not able to modify the environment variables to tell Galaxy where QIIME is installed. In this case, we should add to each XML file the <requirements> tag. This tag tells Galaxy that, before executing the tool, it
should execute the package defined. We have modified the XML generation to include this tag (see Figure 5.14). In addition to this code modification, we also have to add two symbolic links in the QIIME folder. The first one is called default and points to the QIIME folder itself, and the second one is called env.sh and points to the activate.sh file, where are all the environment definitions. Then, when Galaxy is going to execute a tool, it first will setup the environment through this script and then it will execute the tool.

```
# Setting requirements attributes
requirements = doc.createElement("requirements")
req = doc.createElement("requirement")
req.setAttribute("type", "package")
req_text = doc.createTextNode("qiime")
req.appendChild(req_text)
requirements.appendChild(req)
tool.appendChild(requirements)
```

*Figure 5.14 - XML generator added code*

Once we have performed these changes, we can execute the integrate_on_galaxy.py script to integrate QIIME on Galaxy. At this point, we have QIIME successfully installed and integrated on Galaxy. The final step is to make the changes permanent and create the AMI.

To make the changes permanent we will use the Galaxy CloudMan Admin Console page. First, we access to this page by clicking on the Admin option on the top-right of the Galaxy CloudMan management console page. Then, we click on the galaxyTools under the Persist changes to file system option (see Figure 5.15). This option will create a snapshot of the galaxyTools volume, making the changes persistent.

*Figure 5.15 - Persist changes on the file system*
Before creating the AMI, we have to remove our data from the instance. On the second page of the wizard to launch an instance we introduced our AWS data. This data is stored in the instance and, if we create an AMI from this instance, this data is also copied. In this case, any user that instantiates our AMI will be using our account. To remove this data from the cluster, we modify the /mnt/cm/userData.yaml and we remove the access key and secret key values. Then, we can go to the Instances page of the Amazon EC2 console. Here, we select our instance and we will click on the Instance Action button, which will show us a drop-down menu. In this menu we will select the Create Image (EBS AMI) option (see Figure 5. 16). A pop-up window will appear asking us for an Image name and Image description; apart from giving us information from the instance (see Figure 5. 17). Finally, we click on the Yes, create button and the AMI will be created.

![Figure 5. 16 - Creating an AMI](image)

![Figure 5. 17 - Popup window for the AMI creation](image)

The instance you are using as a template for a new image has the following volumes:
- will delete on termination, /dev/sda1, vol-t3bf62e4d (15 GB)
- will not delete on termination, /dev/sdg1, vol-c9a8fa8a3 (700 GB)
- will not delete on termination, /dev/sdg3, vol-0f9f6fe61 (100 GB)
- will not delete on termination, /dev/sdg4, vol-11a78cf (10 GB)

Total size of EBS volumes: 825 GB.
When you create an EBS image an EBS snapshot will also be created for each of the above volumes.
6. Conclusions

QIIME is a powerful software pipeline used by a large number of biologist and bioinformaticians to analyze high-throughput microbial community data. QIIME partially follows an agile development methodology, with continuous development to incorporate new functionality in each release, and priming flexibility over planned development in order to quickly address changes in the underlying data formats and third-party tools. However, the lack of a GUI makes it difficult for users to fully utilize all the functionality included in QIIME, which prevents further progress in the field of microbial ecology and microbiome studies in general.

In this project we have developed a Python software tool that provides a GUI through Galaxy, which makes QIIME utilization more accessible to researchers. As Galaxy integrates several other tools for the analysis of genomic data, a Galaxy-based GUI for QIIME has the added advantage of providing an interface that some users might already be experienced with. The innovation is that the developed software in this project generates automatically the XML files needed to integrate QIIME on Galaxy. This way, even if the underlying QIIME scripts are modified, the GUI would still be accessible by simply re-generating the XML files that Galaxy requires. This feature is fundamental to be able to keep pace with the ever-changing QIIME code base.

In addition, we have replicated the FastUniFrac website through a Galaxy interface, which will be the first step to integrate all web services currently hosted in the Knight lab through a common interface. Since these web services used ad hoc scripts for some processing steps, the integration will also reduce the maintenance effort as the Galaxy-mediated interface is fully based on QIIME scripts. We also provide an Amazon EC2 AMI incorporating these tools, which further facilitates the analysis of large datasets for researchers without access to high-capacity computational resources.

6.1. Future work

In the short term, we will modify the CogentOption class to include the file type in the scripts option definition. By doing so we will be able to include this information in the generated XML files and Galaxy will be up to apply a filter on the existing datasets when the user defines the option values.

In the long term, we will analyze how the software developed for this project could be applied to provide a GUI for other software pipelines. Currently, our software does not rely on any QIIME-specific library, although it uses the PyCogent toolkit. Our first step will therefore be to analyze how other pipelines based on PyCogent could be integrated into Galaxy using a similar framework to the one presented here.
7. Financial report

In this chapter we will explain the costs of this project and its planning.

7.1. Planning

In Figure 7.1 we can see the Gantt diagram with the planning of the project.

Figure 7.1 - Project planning
7.2. Budget

Here, we will describe the approximate cost of the project. This approximation has been done based on the amount of time expended during the development of the project. In

*Figure 7.2* we have a table showing the amount of hours dedicated on each task.

<table>
<thead>
<tr>
<th>Task</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading documentation</td>
<td>120h</td>
</tr>
<tr>
<td>Analysis</td>
<td>150h</td>
</tr>
<tr>
<td>Development</td>
<td>512h</td>
</tr>
<tr>
<td>Testing</td>
<td>160h</td>
</tr>
<tr>
<td>Documentation</td>
<td>180h</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1122h</strong></td>
</tr>
</tbody>
</table>

*Figure 7.2 - Task - hour table*

In order to perform the computation of the costs, we suppose that the project has been developed by a Computer Scientist which works per hours, and his salary is 20€ per hour. Thus, the costs of the workers in this project is 1122 hours × 20€/hour = **22,440€**.

Moreover, we also have to add the costs of the machines where the project has been developed. In our case, the project has been developed in a laptop with an Intel Core i5 2.3 GHz with 4GB of RAM. The cost of this laptop is **599€**.

Finally, we have to take into account the costs of using the cloud. This project has been developed using Amazon EC2. The type of instance used during the development was a *medium extra large* which, at the moment of the development, costs 0.160€ per hour, in the US East (Virginia) availability zone. We have been used the cloud instance during 180 hours. Then, the total cost of using the cloud is 180 hours × 0.160€/hour = **28,80€**.

Hence, the total cost of the project is 22,440€ + 599€ + 28,80€ = **23,067,80€**.
Annex A – FastUniFrac pseudo-code

Libraries

make_heatmap.py

```
make_heatmap.py


    num_values, boundaries, ticks, ticklabels = get_info_from_dict(trans_values)
    width = length(headers['horizontal'])
    height = length(headers['vertical'])
    figure = matplotlib.pyplot.figure(width, height)
    cmap = matplotlib.pyplot.get_cmap('spectral', num_values)
    plot_data = make_plot_data(matrix, trans_values)
    plot = matplotlib.pyplot.imshow(plot_data, cmap)
    axes = figure.axes[0]
    set_xticks_position(axes, 'none')
    set_yticks_position(axes, 'none')
    add_xticks_labels(headers['horizontal'])
    add_yticks_labels(headers['vertical'])
    rotate_xlabels(90)
    save_img(output_dir, 'png')
    save_img(output_dir, 'eps.gz')

    end_function

function get_info_from_dict(trans_values: dictionary) returns num_values: integer, boundaries: list of integers, ticks: list of floats, ticklabels: list of strings

    sorted_values = get_values_sorted(trans_values)
    num_values = length(sorted_values)
    ticks = []
    for i in [1:num_values]:
        ticks.append(float(i))
    boundaries = []
    for i in ticks:
        boundaries.append(i-0.5)
    boundaries.append(float(num_values) - 0.5)
    ticklabels = []
    for values in sorted_values[1:]:
        ticklabels.append(get_label(value))

    end_function

function make_plot_data(matrix: list of lists of floats, trans_values: dictionary) returns results: list of lists of integers

    result = []
    for each row in matrix:
        new_row = []
        for each value in row:
            new_row.append(get_matrix_value(value, trans_values))
        result.append(new_row)

    end_function
```
function get_matrix_value(value: float, trans_values: dictionary) returns val: integer
if value is None then:
    val = trans_values[(None, None)][0]
else:
    for each key in trans_values.keys():
        lower, upper = get_values(key)
        if lower is not None or upper is not None then:
            if lower is None and value <= upper then:
                val = trans_values(key)[0]
            else if lower <= value and value is None then:
                val = trans_values(key)[0]
            else if lower <= value and value <= upper then:
                val = trans_values(key)[0]
        end_if
    end_for
end_if
end_function

make_html_heatmap.py

global
PAGE_HTML: string // contains a full page html code
IMG_MAP_SRC: string // contains the html code of an img tag
AREA_SRC: string // contains the html code of an area tag
MAP_SRC: string // contains the html code of a map tag
DOWNLOAD_LINK: string // contains the html code of an a tag
TABLE_HTML: string // contains the html code of an html table
end_global

action make_html_file(list_data: list of dicts, html_fp: string, output_dir: string)
    page_html_string = get_html_page_string(list_data, output_dir)
    copy_overlib_file(output_dir)
    save_file(html_fp, page_html_string)
end_action

function get_html_page_string(list_data: list of dicts, output_dir: string) returns html: string
    html_table = ''
    for each item in list_data:
        html_table = html_table + get_html_table_string(item, output_dir)
    end_for
    html = add_table(PAGE_HTML, html_table)
end_function

function get_html_table_string(data: dictionary, output_dir: string) returns html: string
    width, height, plot = make_heatmap.plot_heatmap(data['name'], data['headers'], data['matrix'],
    data['transform_values'], output_dir)
    xmap, img_height, img_width = generate_xmap(height, width, data['headers'], data['matrix'],
    plot)
    img_src = add_values(IMG_MAP_SRC, data['name'] + '.png', data['name'], img_width, img_height)
    img_map = add_values(MAP_SRC, data['name'], xmap)
    eps_link = add_values(DOWNLOAD_LINK, data['name'] + '.eps.gz', 'Download figure')
    html = add_values(TABLE_HTML, data['table_title'], img_src, img_map, eps_link)
end_function
    all_cids, all_xcoords, all_ycoords = get_coords(headers, matrix, plot)
    img_height = x_len * 80
    img_width = y_len * 80
    xmap = []
    for i in [0..length(all_cids)]:
        xmap.append(add_values(AREA_SRC, all_xcoords[i], img_height – all_ycoords[i], all_cids[i]))
    end_for
end_function

function get_coords(headers: dictionary, matrix: list of lists of integers, plot: matplotlib.image.AxesImage) returns all_dics: list of strings, all_xcoords: list of integers, all_ycoords: list of integers
    all_cids = []
    all_xcoords = []
    all_ycoords = []
    transform_function = plot.get_transform()
    for j in [0:length(headers[‘vertical’])]
        for i in [0:length(headers[‘horizontal’])]
            if matrix[j][i] is not None then:
                xcoord, ycoord = transform_function(i,j)
                all_cids.append(headers[‘vertical’][j] + ‘ vs ’ + headers[‘horizontal’][i] + ‘ ‘ + string(matrix[j][i]))
                all_xcoords.append(xcoord)
                all_ycoords.append(ycoord)
            end_if
        end_for
    end_for
end_function

newick_to_asciiArt.py

global
PAGE_HTML: string //contains a full page html code
FORMATED_HTML: string // contains the html code of an a tag
ROW_TABLE_LEGEND_HTML: string // contains the html code of a tr tag
TABLE_LEGEND_HTML: string // contains the html code of an html table
end_global

function get_tree_by_length_string(tree: cogen.core.tree.PhyloNode) returns str_tree: string
    tips = tree.tips()
    unscaled_max_length = tips[0].Length
    for each node in tips[1:]:
        if node.Length > unscaled_max_length then:
            unscaled_max_length = node.Length
        end_if
    end_for
    tree.scaleBranchLengths(100)
    tips = tree.tips()
    scaled_max_length = tips[0].Length
    for each node in tips[1:]:
        if node.Length > unscaled_max_length then:
            scaled_max_length = node.Length
        end_if
    end_for
    branch_scale = float(unscaled_max_length) / float(scaled_max_length)
    lines, mid = asciiArt_length(tree)
    str_tree = ‘Scale: 1 dash, slash, backslash ~’ + branch_scale + ‘branch length units\n’
    str_tree = str_tree + join_list(lines)
end_function
function asciiArt_length(tree: cogent.core.tree.PhylNode, char1: string default=' - ') returns
result: list of strings, mid: integer
LEN = tree.Length
PAD = ' ' * LEN
PA = ' ' * (LEN - 1)
namestr = tree.Name
if tree.Children is not None then:
mids = []
result = []
for each child in tree.Children:
  if child == tree.Children[0] then:
    char2 = '/'
  else if child == tree.Children[length(tree.Children) - 1] then:
    char2 = '\'
  else:
    char2 = '-'
  end_for
clines, mid = asciiArt_length(c, char2)
mids.append(mid + length(result)
result.extend(clines)
prefixes = []
for i in [0:mids[0]+1]:
  prefixes.append(PAD)
end_for
for i in [0:mids[length(mids)-1] - mids[0] - 1]:
  prefixes.append(PA + '|')
end_for
for i in [0:length(mids) - mids[length(mids)-1]]:
  prefixes.append(PAD)
end_for
mid = (mids[0] + mids[length(mids) - 1]) / 2
if LEN != 0 then:
prefixes[mid] = char1 + ' - ' * (LEN - 2) + prefixes[mid][length(prefixes[mid])-1]
end_if
for i in [0:length(result)]:
  result[i] = prefixes[i] + result[i]
end_for
if LEN == 0 then:
n = mids[length(mids)-1] - mids[0] - 1
for i in [0:n]:
  result[i + n] = '|' + result[i + n][2:]
end_for
else:
  result = [char1 + ' - ' * LEN + '>' + namestr + '\n']
  mid = 0
end_if
end_function

tree_html_text = get_jackknife_tree_html_string(tree, support[‘trees_considered’], trans_values)
copy_overlib_file(output_dir)
save_file(html_fp, tree_html_text)
end_action

function get_jackknife_tree_html_string(tree: cogent.core.tree.PhoNode, num_trees: integer, trans_values: dictionary) returns html: string
html_string = draw_jackknife_tree_html(tree, num_trees, trans_values)
html_string = html_string + get_legend_table_html(trans_values)
html = add_values(PAGE_HTML, html_string)
end_function
function draw_jackknife_tree_html(tree: cogent.core.tree.PhyloNode, num_trees: integer, trans_values: dictionary) returns html: string

tips = tree.tips()
unscaled_max_length = tips[0].Length
for each node in tips[1:]:
    if node.Length > unscaled_max_length then:
        unscaled_max_length = node.Length
    end_if
end_for

tree.scaleBranchLengths(100)
tips = tree.tips()
scaled_max_length = tips[0].Length
for each node in tips[1:]:
    if node.Length > unscaled_max_length then:
        scaled_max_length = node.Length
    end_if
end_for

branch_scale = float(unscaled_max_length) / float(scaled_max_length)
lines, mid = asciiArt_length_html(tree, num_trees, trans_values)
new_lines = []
for each line in lines:
    new_lines.append(replace(line, '+', '&#62;'))
end_for

html = '<pre> Scale: 1 dash, slash, backslash ~' + branch_scale + ' branch length units\n' html = html + join_list(lines) + '</pre>
end_function
function asciiArt_length_html(tree: cogent.core.tree.PhyloNode, num_trees: integer, trans_values: dictionary, char1: string default='\'-') returns result: list of strings, mid: integer
LEN = tree.Length
PAD = ' ' * LEN
PA = ' ' * (LEN - 1)
namestr = tree.Name
if tree.Children is not None then:
  mids = []
  result = []
  for each child in tree.Children:
    if child == tree.Children[0] then:
      char2 = get_formated_char_html('/', num_trees, namestr, trans_values)
    else if child == tree.Children[length(tree.Children) - 1] then:
      char2 = get_formated_char_html('\', num_trees, namestr, trans_values)
    else:
      char2 = get_formated_char_html('\-', num_trees, namestr, trans_values)
  end_for
  clines, mid = asciiArt_length_html(c, num_trees, trans_values, char2)
  mids.append(mid + length(result))
  result.extend(clines)
  prefixes = []
  for i in [0:mids[0]+1]:
    prefixes.append(PAD)
  end_for
  for i in [0:mids[length(mids)-1] - mids[0] - 1]:
    prefixes.append(PA + get_formated_char_html('|', num_trees, namestr, trans_values))
  end_for
  for i in [0:length(mids) - mids[length(mids)-1]]:
    prefixes.append(PAD)
  end_for
  mid = (mids[0] + mids[length(mids)-1]) / 2
  if LEN == 0 then:
    result[i] = prefixes[i] + result[i]
  end_for
  end_if
else:
  result = [char1 + '\-' * LEN + '>\n' + namestr + '\n']
  mid = 0
end_if
end_function

function get_formated_char_html(char: string, num_trees: integer, fraction: string, trans_values: dictionary) returns html: string
fraction = float(fraction)
count = num_trees * fraction
color = make_heatmap.get_matrix_value(fraction, trans_values)
html = add_values(FORMATED_HTML, count, fraction, color, char)
end_function
function get_last_char_of_html_string(html_string: string) returns html: string
    if length(html_string) == 0 then:
        html = ''
    else if html_string[length(html_string) - 1] == '>' then:
        left, sep, right = divide_string_from_right(html_string, '<a')
        html = sep + right
    else:
        html = html_string[length(html_string) - 1]
    end_if
end_function

function remove_first_chars_of_html_string(html_string: string, num_chars: integer) returns html: string
    if length(html_string) == 0 then:
        html = ''
    else if num_chars == 0 then:
        html = html_string
    else if html_string[0] == '<' then:
        left, sep, right = divide_string_from_left(html_string, '</a')
        html = remove_first_chars_of_html_string(right, num_chars - 1)
    else:
        html = remove_first_chars_of_html_string(html_string[1:], num_chars-1)
    end_if
end_function

function get_legend_table_html(trans_values: dictionary) returns html: string
    rows = ''
    sorted_keys = get_keys_sorted(trans_values)
    for each key in sorted_keys:
        color = trans_values[key][0]
        description = replace(trans_values[key][1], '>', '&gt;')
        description = replace(description, '<', '&lt;')
        rows = rows + add_values(ROW_TABLE_LEGEND_HTML, color, description)
    end_for
    html = add_values(TABLE_LEGEND_HTML, rows)
end_function

parse_beta_significance_output.py
function parse_beta_significance_output_pairwise(file: file object) returns result: dictionary, test_name: string
    comment = file.next()
    file.next()
    result = empty_dictionary()
    for each line in file:
        s1, s2, pval, pcorr = split(line, '\t')
        if pval[0] == '<':
            sym, pval = split(pval, '=')
        end_if
        if pcorr[0] == '<':
            sym, pcorr = split(pcorr, '=')
        end_if
        result[(s1, s2)] = (float(pval), float(pcorr))
    end_for
    test_name = string(comment[1:])
    test_name = replace(test_name, '\n', '')
end_function
function parse_beta_significance_output_each_sample(file: file object) returns result:
dictionary, test_name: string
    comment = file.next()
    file.next()
    result = empty_dictionary()
    for each line in file:
        sample, pval, pcorr = split(line, '\t')
        if pval[0] == '<':
            sym, pval = split(pval, '=')
        end_if
        if pcorr[0] == '<':
            sym, pcorr = split(pcorr, '=')
        end_if
        result[sample] = (float(pval), float(pcorr))
    end_for
    test_name = string(comment[1:])
    test_name = replace(test_name, '\n', '')
end_function

parse_jackknife_support.py
function parse_jackknife_support_file(file: file object) returns result: dictionary
result = empty_dictionary()
    dict_support = empty_dictionary()
    for each line in file:
        if line[0] == '#' then:
            if ':' in line then:
                comment, num_trees = split(line, ':')
                result['trees_considered'] = int(num_trees)
            end_if
        else:
            node_name, value = split(line, '\t')
            dict_support[node_name] = float(value)
        end_if
    end_for
    result['support_dict'] = dict_support
end_function

Scripts

category_map_to_id_map_parser.py
class
    attributes
        Alphabet: list of strings
        Current_indexes: list of integers
    end_attributes
    methods
        def Constructor():
            for each key in keys(cogent.seqsim.sequence_generators.IUPAC_DNA):
                if key == cogent.seqsim.sequence_generators.IUPAC_DNA[key] then:
                    Alphabet.append(key)
            end_if
        end_for
        for i in [0:12]:
            Current_indexes.append(0)
        end_for
    end_def
function __call__() returns barcode: string
    barcode = ''
    for i in [0:12]:
        barcode = barcode + Alphabet[Current_indexes[i]]
    end_for
    updateIndexes(11)
end_function

action updateIndexes(index: integer):
    if index == -1 then:
        throw_exception('Unable to create more uniques barcodes with 12 nucleotides')
    else:
        Current_indexes[index] = (Current_indexes[index] + 1) % length(Alphabet)
        if Current_indexes[index] == 0 then:
            updateIndexes(index-1)
        end_if
    end_if
end_action
end_methods
end_class

action main():
    input_fp, output_fp = parse_command_line_parameters()
    mapping_data, comments = convert_category_map_to_id_map(input_fp)
    write_corrected_file(mapping_data, comments, output_fp)
end_action

function convert_category_map_to_id_map(file: file object) returns mapping_data: list of lists of strings, comments: list of strings
    mapping_data, header, comments = qiime.parse.parse_mapping_file(file)
    header, add_barcode, add_linker, add_descr = header_checks(header)
    mapping_data, type, errs = qiime.check_id_map.check_bad_chars_sampleids(mapping_data)
    mapping_data, type, errs = qiime.check_id_map.check_bad_chars(mapping_data)
    if add_barcode then:
        mapping_data = add_barcode(mapping_data)
    end_if
    if add_linker then:
        mapping_data = add_linker_primer(mapping_data)
    end_if
    if add_descr then:
        mapping_data = add_description(mapping_data)
    end_if
end_function

function header_checks(header: list of strings) returns header: list of strings, add_barcode: boolean, add_linker: boolean, add_descr: boolean
    header, err = qiime.check_id_map.blank_header(header)
    if err is not None then:
        throw_exception(string(err))
    end_if
    header, err = qiime.check_id_map.bad_char_in_header(header)
    header[0] = '#' + header[0]
    header, err = qiime.check_id_map.sampleid_missing(header)
    if err is not None then:
        throw_exception(string(err))
    end_if
    header, err = qiime.check_id_map.barcode_missing(header)
    add_barcode = err is None
    header, err = qiime.check_id_map.linker_primer_missing(header)
    add_linker = err is None
    header, err = qiime.check_id_map.description_missing(header)
end_function
add_descr = err is None

end_function

function addBarcode(data: list of lists of strings) return data: list of lists of strings
data[0].insert(1, 'BarcodeSequence')
barcode_gen = BarcodeGenerator()
for each row in data[1:]:
    row.insert(1, barcode_gen())
end_for
end_function

function addLinkerPrimer(data: list of lists of strings) return data: list of lists of strings
data[0].insert(2, 'LinkerPrimerSequence')
for each row in data[1:]:
    row.insert(1, 'AAAAAAAAAAAAAAAAAAAAA')
end_for
end_function

function addDescription(data: list of lists of strings) return data: list of lists of strings
for each row in data[1:]:
    row.append(row[0])
end_for
end_function

action write_corrected_file(data: list of lists of strings, comments: list of strings, output_fp: string)
    file = open(output_fp)
    writes_tab_delimited_row(file, data[0])
    for each comment in comments:
        file.write('#' + comment + '\n')
    end_for
    for each row in data[1:]
        writes_tab_delimited_row(file, row)
    end_for
end_action

action writes_tab_delimited_row(file: file object, row: list of strings)
    file.write(row[0])
    for each field in row[1:]
        file.write('\t' + field)
    end_for
end_action
sample_id_map_to_otu_table_parser.py

```python
action main()
    input_fp, output_fp = parse_command_line_parameters()
    biom_otu_table = convert_sample_id_mapping_file(input_fp)
    write_file(output_fp, biom_otu_table)
end_action

function convert_sample_id_mapping_file(file: file object) returns otu_table_str: string
    otu_ids = []
    sample_ids = []
    two_d_dict = empty_dictionary()
    for each line in file:
        if line[0] != '#':
            sec_id, sample_id, times = split(line, '\t')
            if not sec_id in otu_ids then:
                otu_ids.append(sec_id)
            end_if
            sample_id, e = qiime.check_id_map.sample_id_filter.resultAndError(sample_id)
            if not sample_id in sample_ids then:
                sample_ids.append(sample_id)
            end_if
            two_d_dict[(otu_ids.index(sec_id), sample_ids.index(sec_id))] = float(times)
        end_if
    end_for
    data = biom.table.dict_to_sparsedict(two_d_dict)
    table_obj = biom.table.SparseOTUTable(data, sample_ids, otu_ids)
    otu_table_str = table_obj.getBiomFormatJsonString()
end_function
```

copy_data.py

```python
action main()
    input_fp, output_fp = parse_command_line_parameters()
    shutil.copyfile(input_fp, output_fp)
end_action
```

modify_qiime_parameters.py

```python
action main()
    input_fp, output_fp, parameters, new_values = parse_command_line_parameters()
    modify_qiime_parameters(input_fp, output_fp, parameters, new_values)
end_action

action modify_qiime_parameters(param_file: file object, out_file: file object, parameters: string, new_values: string)
    changes = create_dict_of_changes(parameters, new_values)
    new_file = list(param_file)
    for each key in keys(changes):
        new_file = replace_value(new_file, key, changes[key])
    end_for
    write_file(output_fp, new_file)
end_action

function create_dict_of_changes(params: string, values: string) returns changes: dictionary
    list_params = split(params, ',')
    list_values = split(values, ',')
    changes = empty_dictionary()
    for i in [0:length(list_params)]
        changes[list_param[i]] = list_values[i]
    end_for
end_function
```
function replace_value(lines: list of strings, param: string, value: string) returns new_lines:
list of strings
new_lines = []
finded = False
for each line in lines:
    if length(line) != 0 and line[0] != '#' and param in line then:
        line = param + '\t' + replace(value, '-', ',') + '\n'
        finded = True
    end_if
    new_lines.append(line)
end_for
if not finded then:
    new_lines.append(param + '\t' + replace(value, '-', ',') + '\n')
end_if
end_function

make_cluster_samples_file.py

action main():
    input_fp, output_fp = parse_command_line_parameters()
    tree = qiime.parse.parse_newick(input_fp)
    tree_text = newick_to_asciiArt.get_tree_by_length_string(tree)
    write_file(output_fp, tree_text)
end_action

make_pcoa_html.py

global
    PAGE_HTML: string // contains a full page html code
    PCOA_STRING: string // string with the text of the link to a PCoA html file
    RAW_DATA_STRING: string // string with the text of the link to the raw data file
    KINEMAGE_CONTINUOUS_STRING: string // string with the text of the link to the kinemage file
    KINEMAGE_DISCRETE_STRING: string // string with the text of the link to the kinemage file
    LINK_HTML: string // contains the html code of an a tag
end_global

action main():
    pcoa_dir, output_fp = parse_command_line_parameters()
    make_html_file(pcoa_dir, output_fp)
end_action

action make_html_file(pcoa_dir: string, html_fp: string):
    html_string = get_html_string(pcoa_dir)
    write_file(html_fp, html_string)
end_action

function get_html_string(pcoa_dir: string) returns html: string
links = get_dict_links(pcoa_dir)
html_links = get_full_html_links(links)
html = add_values(PAGE_HTML, html_links)
end_function
function get_dict_links(pcoa_dir: string) returns links: dictionary

dir_content = list_directory_contents(pcoa_dir)

links = empty_dictionary()

for i in [0:length(dir_content)]:
    if is_directory(dir_content[i][0]) then:
        view_idx, view_link, download_idx, download_link = get_html_links(dir_content[i][0], dir_content[i][1])
        links[view_idx] = view_link
        if download_idx is not None then:
            links[download_idx] = download_link
    else:
        spl_name = split(dir_content[i][1], '_')
        if spl_name[length(spl_name)–1] == 'pc.txt' then:
            links[4] = get_raw_pcoa_download_link(dir_content[i][0], dir_content[i][1])
    end_if
end_for
end_function

function get_html_links(path: string, name: string) returns view_idx: integer, view_link: string, download_idx: integer, download_link: string

view_idx, download_idx = get_link_indexes(name)

dir_content = list_directory_contents(path)

for i in [0:length(dir_content)]:
    if is_file(dir_content[i][0]) then:
        str_link = add_values(PCOA_STRING, name)
        view_link = add_values(LINK_HTML, name, str_link)
    end_if
end_for

download_link = None if download_idx is not None then:
download_link = get_kinemage_link(path, name, download_idx)
end_if
end_function

function get_link_indexes(name: string) returns idx1: integer, idx2: integer

spl_name = split(name, '_')

if spl_name[length(spl_name)–2] == '2d' then:
    if spl_name[length(spl_name)–1] == 'continuous' then:
        idx1 = 0
        idx2 = None
    else if spl_name[length(spl_name)–1] == 'discrete' then:
        idx1 = 1
        idx2 = None
    else:
        throw_exception('Wrong PCoA directory structure')
    end_if
else if spl_name[length(spl_name)–2] == '3d' then:
    if spl_name[length(spl_name)–1] == 'continuous' then:
        idx1 = 2
        idx2 = 5
    else if spl_name[length(spl_name)–1] == 'discrete' then:
        idx1 = 3
        idx2 = 6
    else:
        throw_exception('Wrong PCoA directory structure')
    end_if
else:
    throw_exception('Wrong PCoA directory structure')
end_if
end_function
function get_kinemage_link(path: string, name: string, index: integer) returns html: string

dir_content = list_directory_contents(path)
kinemage_fp = None
for i in range(length(dir_content)):
    if is_directory(dir_content[i][0]) then:
        d_con = list_directory_contents(dir_content[i][0])
        if length(d_con) != 0 then:
            f_name, ext = split(d_con[0][1], '.')
            if ext == 'kin' then:
                kinemage_fp = join_path(name, dir_content[i][1], d_con[0][1])
        end_if
    end_if
end_for
if kinemage_fp is None then:
    throw_exception('Wrong PCoA directory structure')
end_if
if index == 6 then:
    str_link = KINEMAGE_DISCRETE_STRING
end_if
else if index == 5 then:
    str_link = KINEMAGE_CONTINUOUS_STRING
else:
    throw_exception('Wrong kinemage download link index')
end_if
html = add_values(LINK_HTML, kinemage_fp, str_link)
end_function

function get_raw_pcoa_download_link(path: string, name: string) returns html: string
html = add_values(LINK_HTML, name, RAW_DATA_STRING)
end_function

function get_full_html_links(links_dict: dictionary) returns html: string
sorted_keys = get_keys_sorted(link_dict)
html = '
for each key in sorted_keys:
    html = html + links_dict[key]
end_for
end_function

make_beta_significance_heatmap.py

action main()
    input_fp, output_fp, output_dir = parse_command_line_parameters()
    list_data = generate_data_make_html(input_fp)
    make_html_heatmap.make_html_file(list_data, output_fp, output_dir)
end_action

function generate_data_make_html(file: file object) returns result: list of dictionaries
result = []
dict_data, test_name = parse_beta_significance_output.parse_beta_significance_output_pairwise(file)
raw_headers, raw_matrix = generate_headers_and_matrix(dict_data, 0)
corr_headers, corr_matrix = generate_headers_and_matrix(dict_data, 1)
result.append(generate_dict_data('Raw values', raw_headers, raw_matrix, test_name))
result.append(generate_dict_data('Corrected values', corr_headers, corr_matrix, test_name))
end_function
function generate_headers_and_matrix(d_data: dictionary, index: string) returns headers: dictionary, result: list of lists of floats

if index != 0 and index != 1 then:
    throw_exception('Index must be 0 or 1!')
end_if

sorted_keys = get_keys_sorted(d_data)
headers = {'vertical': [sorted_keys[0][0]], 'horizontal': [sorted_keys[0][0]]}
result = []
row = []
row.append(None)
none_ct = 1
for each key in sorted_keys:
    value = d_data[key][index]
    if headers['vertical'][none_ct - 1] != key[0] then:
        result.append(row)
        none_ct = none_ct + 1
        row = []
        for i in [0: none_ct]:
            row.append(None)
        end_for
    end_if
if key[1] not in headers['horizontal'] then:
    headers['horizontal'].append(key[1])
end_if
row.append(value)
result.append(row)
end_for
result.append(row)
end_function

function generate_dict_data(name: string, headers: list of strings, matrix: list of lists of floats, test_name: string) returns result: dictionary

result = empty_dictionary()
result['name'] = name
result['headers'] = headers
result['matrix'] = matrix
result['transform_values'] = generate_trans_values_dict()
result['table_title'] = test_name + ':' + name
end_function

function generate_trans_values() return trans_values: dictionary

trans_values = empty_dictionary()
trans_values[{(None, None)] = (0, '')
trans_values[{(None, 0.001)] = (1, '(<0.001)\nHighly\nsignificant')
trans_values[{(0.001, 0.01)] = (2, '(<0.001-0.01)\nSignificant')
trans_values[{(0.01, 0.05)] = (3, '(0.01-0.05)\nMarginally\nsignificant')
trans_values[{(0.05, 0.1)] = (4, '(0.05-0.1)\nSuggestive')
trans_values[{(0.1, None)] = (5, '(>0.1)\nNot\nsignificant')
end_function

make_unifrac_significance_each_sample.py

make_unifrac_significance_each_sample.py

PAGE_HTML: string // contains a full page html code
ROW_TABLE_LEGEND_HTML: string // contains the html code of a tr tag
TABLE_LEGEND_HTML: string // contains the html code of an html table
ROW_TABLE_HTML: string // contains the html code of a tr tag
TABLE_HTML: string // contains the html code of an html table
DICT_TRANS_VALUES: dictionary // contains the transform dictionary
end_global
action main()
    input_fp, output_fp = parse_command_line_parameters()
    dict_data, test_name = parse_beta_significance_output.parse_beta_significance_output_each_sample(input_fp)
    make_html_file(dict_data, test_name, output_fp)
end_action

action make_html_file(d_data: dictionary, test_name: string, html_fp: string)
    page_html_string = get_html_page_string(d_data, test_name)
    write_file(html_fp, page_html_string)
end_action

function get_html_page_string(d_data, test_name)
    returns html: string
    out_table = get_html_table(d_data, test_name + ': Raw values', 0)
    out_table = out_table + get_html_table(d_data, test_name + ': Corrected values', 1)
    out_table = out_table + get_html_legend_table()
    html = add_values(PAGE_HTML, out_table)
end_function

function get_html_table(d_data, title, index)
    returns html: string
    if index != 0 and index != 1 then:
        throw_exception('Index must be 0 or 1!')
    end_if
    rows = ''
    sorted_keys = get_keys_sorted(d_data)
    for key in sorted_keys:
        value = d_data[key][index]
        color = make_heatmap.get_matrix_value(value, DICT_TRANS_VALUE)
        rows = rows + add_values(ROW_TABLE_HTML, key, color, value)
    end_for
    html = add_values(TABLE_HTML, title, rows)
end_function

function get_html_legend_table()
    returns html: string
    rows = ''
    sorted_keys = get_keys_sorted(DICT_TRANS_VALUES)
    for key in sorted_keys:
        color = DICT_TRANS_VALUES[key][0]
        description = DICT_TRANS_VALUES[key][1]
        rows = rows + add_values(ROW_TABLE_LEGEND_HTML, color, description)
    end_for
    html = add_values(TABLE_LEGEND_HTML, rows)
end_function

make_distance_matrix_heatmap.py

action main()
    input_fp, output_fp, output_dir = parse_command_line_parameters()
    data = generate_data_make_html(input_fp)
    make_html_heatmap.make_html_file([data], output_fp, output_dir)
end_action

function generate_data_make_html(file)
    returns result: dictionary
    header, dist_mat = qiime.parse.parse_distmat(file)
    dist_mat = get_upper_triangle(dist_mat)
    result = empty_dictionary()
    result['name'] = 'Distance matrix'
    result['headers'] = {'vertical': header, 'horizontal': header}
    result['matrix'] = dist_mat
    result['transform_values'] = generate_trans_values_dict(dist_mat)
    result['table_title'] = 'Distance matrix'
end_function
function get_upper_triangle(matrix: list of lists of floats) returns result: list of lists of floats
result = []
for i in [0:length(matrix)]:
    row = []
    row.append(None)
    for j in [0:i]:
        row.append(matrix[i][j])
    result.append(row)
end_for
end_function

function generate_trans_values_dict(dist_mat: list of lists of floats) returns trans_values: dictionary
trans_values = make_quartiles(dist_mat)
trans_values[(None, None)] = (0, '')
end_function

function make_quartiles(dist_mat: list of lists of floats) returns result: dictionary
data = []
for each row in dist_mat:
    for each item in row:
        data.append(item)
    end_for
end_for
min_value = min(data)
max_value = max(data)
lq, mq, uq = compute_quartiles(data)
result = empty_dictionary()
result[(min_value - 0.01, lq)]: (1, "(0-25%) Lower Quartile")
result[(lq, mq)]: (2, "(25-50%)")
result[(mq, uq)]: (3, "(50-75%)")
result[(uq, max_value + 0.01): (4, "(75-100%) Upper Quartile")
end_function

function compute_quartiles(data: list of floats) returns lq: float, mq: float, uq: float
if length(data) < 4 then:
    throw_exception('Not enough values to compute quartiles')
end_if
data = sort(data)
mq = median(data)
len_data = length(data)
m_ix = len_data / 2
if len_data % 2 == 0 then:
    lq = median(data[0:m_ix])
    uq = median(data[m_ix:]):
else:
    lq = median(data[0:m_ix])
    uq = median(data[m_ix + 1:]):
end_if
end_function
make_jackknife_html.py

global
   DICT_TRANS_VALUES: dictionary // contains the transform dictionary
end_global

action main()
   tree_fp, support_fp, output_fp, output_dir = parse_command_line_parameters()
   support = parse_jackknife_support.parse_jackknife_support_file(support_fp)
   tree = qiime.parse.parse_newick(tree_fp)
   newick_to_asciiArt.make_jackknife_support_file(tree, support, DICT_TRANS_VALUES, output_fp, output_dir)
end_action
Annex B – FastUniFrac XML files

cluster_samples.xml

<tool id="cluster_samples" name="Cluster samples" version="1.0">
  <description>
    Uses the UniFrac metric to cluster the samples based on phylogenetic lineages they contain.
  </description>
  <command interpreter="python">
    bin/category_map_to_id_map_parser.py -i $category_map_file -o fasting_map.txt;
    sample_id_map_to_otu_table_parser.py -i $sample_id_file -o otu_table.biom;
    beta_diversity.py -i otu_table.biom -t $tree_file -o ./output
    #if $use_abundance_weights.weights:
    #if $use_abundance_weights.normalized:
      #set filename = 'output/weighted_normalized_unifrac_otu_table.txt'
    #else:
      #set filename = 'output/weighted_unifrac_otu_table.txt'
    #end if
    #else:
      #set filename = 'output/unweighted_unifrac_otu_table.txt'
    #end if
    upgma_cluster.py -i $filename -o $cluster_samples_output;
    make_cluster_samples_file.py -i $cluster_samples_output -o $ascii_art_output
  </command>
  <inputs>
    <param name="tree_file" type="data" label="Select reference tree"/>
    <param name="sample_id_file" type="data" label="Select sample ID mapping file"/>
    <param name="category_map_file" type="data" label="Select category mapping file"/>
    <conditional name="use_abundance_weights">
      <param name="weights" type="boolean" default="False" label="Use abundance weights"/>
    </conditional>
  </inputs>
  <outputs>
    <data format="txt" name="cluster_samples_output"/>
    <data format="txt" name="ascii_art_output"/>
  </outputs>
  <help/>
</tool>
PCoA.xml

<tool id="pcoa" name="PCoA" version="1.0">
  <description>
  Uses the UniFrac metric to perform principal coordinates analysis on your samples, allowing you to see whether different types of samples are separated in different dimensions.
  </description>
  <command interpreter="python">
    bin/category_map_to_id_map_parser.py -i $category_map_file -o fasting_map.txt;
    sample_id_map_to_otu_table_parser.py -i $sample_id_file -o otu_table.biom;
    modify_qiime_parameters.py -i $__root_dir__/tools/fastunifrac/qiime_parameters.txt -o param_file.txt -p beta_diversity:metrics,make_2d_plots:scree
    #if $use_abundance_weights.weights:
      -n weighted_normalized_unifrac,True;
    #else:
      -n unweighted_unifrac,True;
    #end if
    beta_diversity_through_plots.py -i otu_table.biom -m fasting_map.txt -t $tree_file -o $output_html.files_path -p param_file.txt;
    make_pcoa_html.py -o $output_html -d $output_html.files_path;
  </command>
  <inputs>
    <param name="tree_file" type="data" label="Select reference tree"/>
    <param name="sample_id_file" type="data" label="Select sample ID mapping file"/>
    <param name="category_map_file" type="data" label="Select category mapping file"/>
    <conditional name="use_abundance_weights">
      <param name="weights" type="boolean" default="False" label="Use abundance weights"/>
      <when value="true">
        <param name="normalized" type="boolean" default="false" label="Normalized"/>
      </when>
    </conditional>
  </inputs>
  <outputs>
    <data format="html" name="output_html"/>
  </outputs>
  <help> </help>
</tool>
<tool id="p_test_significance" name="P Test Significance" version="1.0">
  <description>
    Tells you which pairs of samples are significantly different using the P Test.
  </description>
  <command interpreter="python">
    bin/category_map_to_id_map_parser.py -i $category_map_file -o fasting_map.txt;
    sample_id_map_to_otu_table_parser.py -i $sample_id_file -o otu_table.biom;
    beta_significance.py -i otu_table.biom -o $p_test_output -s p-test -t $tree_file -n
    $permutations -k $type_of_test;
    #if str($type_of_test) == "each_pair":
      make_beta_significance_heatmap.py -i $p_test_output -o $output_html --
      output_dir=$output_html.files_path
    #end if
  </command>
  <inputs>
    <param name="tree_file" type="data" label="Select reference tree"/>
    <param name="sample_id_file" type="data" label="Select sample ID mapping file"/>
    <param name="category_map_file" type="data" label="Select category mapping file"/>
    <param name="permutations" type="select" label="Number of permutations">
      <option value="50">50</option>
      <option value="100">100</option>
      <option value="500">500</option>
      <option value="750">750</option>
      <option value="1000">1000</option>
    </param>
    <param name="type_of_test" type="select" label="Type of test">
      <option value="all_together">All samples together</option>
      <option value="each_pair">Each pair of samples</option>
    </param>
  </inputs>
  <outputs>
    <data format="txt" name="p_test_output"/>
    <data format="html" name="output_html">
      <filter>(type_of_test == "each_pair")</filter>
    </data>
  </outputs>
  <help> </help>
</tool>
**unifrac_significance.xml**

<tool id="unifrac_significance" name="Unifrac Significance" version="1.0">
    <description>
        Tells you which pairs of samples are significantly different using the UniFrac significance test.
    </description>
    <command interpreter="python">
        bin/category_map_to_id_map_parser.py -i $category_map_file -o fasting_map.txt;
        sample_id_map_to_otu_table_parser.py -i $sample_id_file -o otu_table.biom;
        beta_significance.py -i otu_table.biom -o $unifrac_output -t $tree_file -n $u_permutations -k $type_of_test
        #if $use_abundance_weights.weights:
            #if $use_abundance_weights.normalized:
                -s weighted_normalized_unifrac;
            #else:
                -s weighted_unifrac;
            #end if
        #else:
            -s unweighted_unifrac;
        #end if
        #if str($type_of_test) == "each_pair":
            make_beta_significance_heatmap.py -i $unifrac_output -o $output_html --output_dir=$output_html.files_path
        #else if str($type_of_test) == "each_sample":
            make_unifrac_significance_each_sample_html.py -i $unifrac_output -o $output_html_indiv
        #end if
    </command>
    <inputs>
        <param name="tree_file" type="data" label="Select reference tree"/>
        <param name="sample_id_file" type="data" label="Select sample ID mapping file"/>
        <param name="category_map_file" type="data" label="Select category mapping file"/>
        <param name="u_permutations" type="select" label="Number of permutations">
            <option value="50">50</option>
            <option value="100">100</option>
            <option value="500">500</option>
            <option value="750">750</option>
            <option value="1000">1000</option>
            <option value="2500">2500</option>
            <option value="5000">5000</option>
            <option value="7500">7500</option>
        </param>
        <conditional name="use_abundance_weights">
            <param name="weights" type="boolean" default="False" label="Use abundance weights"/>
            <when values="true">
                <param name="normalized" type="boolean" default="false" label="Normalized"/>
            </when>
        </conditional>
        <param name="type_of_test" type="select" label="Type of test">
            <option value="all_together">All samples together</option>
            <option value="each_pair">Each pair of samples</option>
            <option value="each_sample">Each sample individually</option>
        </param>
    </inputs>
    <outputs>
        <data format="txt" name="unifrac_output"/>
        <data format="html" name="output_html">
            <filter>(type_of_test == "each_pair")</filter>
        </data>
        <data format="html" name="output_html_indiv">
            <filter>(type_of_test == "each_sample")</filter>
        </data>
    </outputs>
</tool>
sample_counts.xml

<tool id="sample_counts" name="Sample counts" version="1.0">
  <description>
  Tells you how many sequences are in each sample.
  </description>
  <command interpreter="python">
    bin/category_map_to_id_map_parser.py -i $category_map_file -o fasting_map.txt;
    sample_id_map_to_otu_table_parser.py -i $sample_id_file -o otu_table.biom;
    per_library_stats.py -i otu_table.biom -o $sample_counts_output -m fasting_map.txt
    #if not $weights:
    --otu_counts
    #endif
  </command>
  <inputs>
    <param name="tree_file" type="data" label="Select reference tree"/>
    <param name="sample_id_file" type="data" label="Select sample ID mapping file"/>
    <param name="category_map_file" type="data" label="Select category mapping file"/>
    <param name="weights" type="boolean" default="false" label="Use abundance weights"/>
  </inputs>
  <outputs>
    <data format="txt" name="sample_counts_output"/>
  </outputs>
</tool>
sample_distance_matrix.xml

<tool id="sample_distance_matrix" name="Sample Distance Matrix" version="1.0">
  <description>
  Shows you the UniFrac distances between each pair of samples and is used as input for sample clustering and PCoA.
  </description>
  <command interpreter="python">
    bin/category_map_to_id_map_parser.py -i $category_map_file -o fasting_map.txt;
    sample_id_map_to_otu_table_parser.py -i $sample_id_file -o otu_table.biom;
    beta_diversity.py -i otu_table.biom -t $tree_file -o ./output
    #if $use_abundance_weights.weights:
      #if $use_abundance_weights.normalized:
        -m weighted_normalized_unifrac;
        #set filename = 'output/weighted_normalized_unifrac_otu_table.txt'
      #else:
        -m weighted_unifrac;
        #set filename = 'output/weighted_unifrac_otu_table.txt'
      #end if
    #else:
      -m unweighted_unifrac;
      #set filename = 'output/unweighted_unifrac_otu_table.txt'
    #end if
    copy_data.py -i $filename -o $output_distance_matrix;
    make_distance_matrix_heatmap.py -i $output_distance_matrix -o $output_html --output_dir=$output_html.files_path
  </command>
  <inputs>
    <param name="tree_file" type="data" label="Select reference tree"/>
    <param name="sample_id_file" type="data" label="Select sample ID mapping file"/>
    <param name="category_map_file" type="data" label="Select category mapping file"/>
    <conditional name="use_abundance_weights">
      <param name="weights" type="boolean" default="False" label="Use abundance weights"/>
      <when value="true">
        <param name="normalized" type="boolean" default="false" label="Normalized"/>
      </when>
    </conditional>
  </inputs>
  <outputs>
    <data format="txt" name="output_distance_matrix" />
    <data format="html" name="output_html" />
  </outputs>
  <help> </help>
</tool>
jackknife_sample_clusters.xml

<tool id="jackknife_sample_clusters" name="Jackknife Sample Clusters" version="1.0">
  <description>
  Performs statistical resampling and will allow you to see how confident you should be in the sample clustering results.
  </description>
  <command interpreter="python">
    bin/category_map_to_id_map_parser.py -i $category_map_file -o fasting_map.txt;
    sample_id_map_to_otu_table_parser.py -i $sample_id_file -o otu_table.biom;
    modify_qiime_parameters.py -i $__root_dir__/tools/fastunifrac/qiime_parameters.txt -o param_file.txt -p beta_diversity:metrics,multiple_rarefactions_even_depth:num-reps
    #if $use_abundance_weights.weights:
    #if $use_abundance_weights.normalized:
    #set jack_dir='./output/weighted_normalized_unifrac/upgma_cmp/
    -n weighted_normalized_unifrac,$permutations;
    #else:
    #set jack_dir='./output/weighted_unifrac/upgma_cmp/
    -n weighted_unifrac,$permutations;
    #end if
    #else:
    #set jack_dir='./output/unweighted_unifrac/upgma_cmp/
    -n unweighted_unifrac,$permutations;
    #end if
    jackknifed_beta_diversity.py -i otu_table.biom -o ./output -e $num_sec -m fasting_map.txt -t
    $tree_file -p param_file.txt;
    make_jackknife_html.py -s $jack_dir/jackknife_support.txt -t
    $jack_dir/jackknife_named_nodes.tre -o $output_html --output_dir=$output_html.files_path;
    copy_data.py -i $jack_dir/jackknife_named_nodes.tre -o $output_jack_tree
  </command>
  <inputs>
    <param name="tree_file" type="data" label="Select reference tree"/>
    <param name="sample_id_file" type="data" label="Select sample ID mapping file"/>
    <param name="category_map_file" type="data" label="Select category mapping file"/>
    <param name="permutations" type="select" label="Number of permutations">
      <option value="10">10</option>
      <option value="50">50</option>
      <option value="100">100</option>
      <option value="1000">1000</option>
    </param>
    <conditional name="use_abundance_weights">
      <param name="weights" type="boolean" default="False" label="Use abundance weights"/>
      <when value="true">
        <param name="normalized" type="boolean" default="false" label="Normalized"/>
      </when>
    </conditional>
  </inputs>
  <outputs>
    <data format="txt" name="output_jack_tree" />
    <data format="html" name="output_html" />
  </outputs>
  <help> </help>
</tool>
Annex C - FastUniFrac deploy package pseudo-code

```
integrate.py

global

    STR_BASH = 'export PATH=$PATH:%s
    export PYTHONPATH=$PYTHONPATH:%s
    export GALAXY_HOME=%s
',

    STR_TCSH = 'setenv PATH $PATH:%s
    setenv PYTHONPATH $PYTHONPATH:%s
    setenv GALAXY_HOME %s
',

    TGZ_FILE = './files/fastunifrac.tar.gz'
    TOOL_CONF_FILE = './files/tool_conf.xml.sample'
    UNIVERSE_FILE = './files/universe_wsgi.ini.sample'
    INIT_FILE = './files/__init__.py'

end_global

action main()

    galaxy_home, bash_fp, is_tcsh = parse_command_line_parameters()
    tgz_dest = join_path(galaxy_home, 'tools')
    tool_conf_dest = join_path(galaxy_home, 'tool_conf.xml')
    universe_wsgi_dest = join_path(galaxy_home, 'universe_wsgi.ini')
    init_file_dest = join_path(galaxy_home, 'lib/galaxy/objectstore/__init__.py')
    bin_path = join_path(tgz_dest, 'fastunifrac/bin/')
    lib_path = join_path(tgz_dest, 'fastunifrac/bin/')
    file = tarfile.open(TGZ_FILE)
    file.extract(tgz_dest)
    file.close()
    append_string = ''
    if is_tcsh:
        append_string = add_values(STR_TCSH, bin_path, bin_path + ':' + lib_path, galaxy_home)
    else:
        append_string = add_values(STR_BASH, bin_path, bin_path + ':' + lib_path, galaxy_home)
    command_string = 'echo "" > append_string + "" >> ' + bash_file
    execute_command(command_string)

end_action
```
Annex D – QIIME automated integration pseudo-code

galaxy_integration.py

**action** integrate(script_dir: string, galaxy_dir: string, conf_file: string, update: bool, log_file: string)

- d_scripts, sections = parse_config_file(open_file(conf_file))
- create_dirs(galaxy_dir, sections)
- tool_conf = get_galaxy_tool_conf_file(galaxy_dir, update)
- d_sections = empty_dictionary()
- for each sect in sections:
  - d_sections[sect] = []
- end_for
- add_sitedir(script_dir)
- log_file = open_file(log_file)
- for each file in script_dir:
  - if is_python_script(file) then:
  - write_log_file("Generating XML file for " + file + " script...")
  - if file in d_scripts then:
    - section, opts_discarded = d_scripts[file]
    - xml_generator.make_xml(file, join_path(galaxy_dir, section), opts_discarded)
    - d_sections[section].append(file)
    - write_log_file("Ok")
  - else:
    - write_log_file("skipped, not in configuration file")
  - end_if
  - end_if
- end_for
- update_tool_conf_xml(tool_conf, d_sections)
- write_file(join_path(galaxy_dir, "tool_conf.xml"), tool_conf)
- update_environment()
- end_action

**function** parse_config_file(file: file object) returns d_scripts: dictionary, sections: list

- d_scripts = empty_dictionary()
- sections = []
- current_section = None
  - for each line in file:
    - if starts_with_tab(line) then:
      - current_section = get_section_name(line)
      - sections.append(current_section)
    - else:
      - if current_section == None then throw_exception('Bad configuration file')
      - script, opts_discarded = get_info(line)
      - d_scripts[script] = new_tuple(current_section, opts_discarded)
    - end_if
  - end_for
- end_function

**action** create_dirs(galaxy_dir: string, sections: list)

- tools_dir = join_path(galaxy_dir, 'tools')
- for each sect in sections:
  - sect = lowercase(sect)
  - make_dir(join_path(tools_dir, sect))
- end_for
- end_action
function get_galaxy_tool_conf_file(galaxy_dir: string, update: boolean) returns tool_conf:
  tool_conf_path = join_path(galaxy_dir, 'tool_conf.xml')
  if not exists_path(tool_conf_path) then throw_exception('Bad galaxy installation folder')
  if update:
    tool_conf = parse_xml(tool_conf_path)
  else:
    tool_conf = new_xml_document()
    toolbox = add_tag(tool_conf, 'toolbox')
    section = add_tag(toolbox, 'section')
    add_attribute(section, 'name', 'Get data')
    add_attribute(section, 'id', 'getext')
    tool = add_tag(section, 'tool')
    add_attribute(tool, 'file', 'data_source/upload.xml')
  end_if
end_function

  for each sect in keys(d_section):
    sect_node = get_node(sect, tool_conf)
    if sect_node is not None:
      for each script in d_section[sect]:
        if not exists_script_in_section(script, sect_node) then:
          tool = add_tag(sect_node, 'tool')
          add_attribute(tool, 'file', path_join(sect, base_name(script), '.xml'))
        end_if
      end_for
    else:
      section = add_tag(get_first_tag(tool_conf), 'section')
      add_attribute(section, 'name', sect)
      add_attribute(section, 'id', lowercase(sect))
      for each script in d_section[sect]:
        tool = add_tag(section, 'tool')
        add_attribute(tool, 'file', path_join(sect, base_name(script), '.xml'))
      end_for
    end_if
  end_for
end_action
xml_generator.py

class OptionInfo:
    attributes:
    name: string
    type: string
    short_opt: string
    long_opt: string
    label: string
    default: string
    choices: list of strings
    format: string

methods:
    def Constructor(opt: cogent.util.option_parsing.CogentOption)
    name = opt.get_opt_string()
    if opt.action in ['store_true', 'store_false'] then:
        type = 'boolean'
    else:
        type = convert_to_galaxy_type(opt.type)
    end_if
    short_opt = None
    long_opt = None
    if length(opt._short_opts) > 0 then:
        short_opt = _short_opts[0]
    end_if
    if length(opt._long_opts) > 0 then:
        long_opt = _long_opts[0]
    end_if
    default = None
    if type == 'boolean' then:
        default = 'False'
    else if has_default(opt) then:
        default = opt.default
    end_if
    choices = None
    if type == 'select' then:
        choices = opt.choices
    else if type == 'multiple_select' then:
        choices = opt.mchoices
    end_if
    format = None
    if type == 'output' then:
        format = 'txt'
    else if type == 'output_dir' then:
        format = 'tgz'
    end_if
end_def

function get_command_line_string() returns opt_str: string
    if short_opt is not None then:
        opt_str = short_opt
    else:
        opt_str = long_opt
end_if
end_function
function is_short_command_line() returns is_short: boolean
    is_short = short_opt is not None
end_function

function has_default() returns has_def: boolean
    has_def = default is not None
end_function

end_methods
end_class

class ScriptInfo:
    attributes:
        id: string
        name: string
        version: string
        description: string
        required_opts: list of OptionInfo
        optional_opts: list of OptionInfo
        help: string
        command: string
    end_attributes

methods:
    def Constructor(script_info: dictionary, script_name: string, cmd: string)
        id = script_name
        name = replace(script_name, '_', ' ')
        version = script_info['version']
        description = script_info['brief_description']
        required_opts = []
        for each opt in script_info['required_options']:
            required_opts.append(OptionInfo(opt))
        end_for
        optional_opts = []
        for each opt in script_info['optional_options']:
            optional_opts.append(OptionInfo(opt))
        end_for
        help = script_info['script_description']
        command = cmd
    end_def

function _get_optional_opt(name) returns opt: OptionInfo
    opt = None
    for each option in optional_opts:
        if option.name == name then:
            opt = option
        end_if
    end_for
end_function

action remove_options(opt_names: string)
    names = split(opt_names, ',')
    for each name in names:
        opt = _get_optional_opt(name)
        if opt is not None then:
            optional_opts.remove(opt)
        else:
            throw_exception('Option does not exists or it is required')
        end_if
    end_for
end_action
LIST_DICT_TO_STRING_FUNCTION = '   #def list_dict_to_string(list_dict):
#set $file_list = list_dict[0]['additional_input'].__getattr__('file_name')
#for d in list_dict[1:]:
#set $file_list = $file_list + ',' + d['additional_input'].__getattr__('file_name')
#end for
#return $file_list
#end def
   end_def
   end_global

class CommandGenerator
   attributes:
       info: ScriptInfo
       command_text: string
       list_dict_to_string_defined: boolean
       is_optional: boolean
       uncompress_command: string
       compress_command: string
       type_dependant_functions: dictionary
   end_attributes

   methods:
       def Constructor(info_scr: ScriptInfo)
           info = info_scr
           command_text = info_scr.command
           list_dict_to_string_defined = False
           is_optional = False
           uncompress_command = ''
           compress_command = ''
           type_dependant_functions = empty_dictionary()
           type_dependant_functions['text'] = generate_text_command_text
           type_dependant_functions['integer'] = generate_integer_float_command_text
           type_dependant_functions['float'] = generate_integer_float_command_text
           type_dependant_functions['select'] = generate_data_select_command_text
           type_dependant_functions['multiple_select'] = generate_data_select_command_text
           type_dependant_functions['data'] = generate_data_select_command_text
           type_dependant_functions['input_dir'] = generate_input_dir_command_text
           type_dependant_functions['repeat'] = generate_repeat_command_text
           type_dependant_functions['output'] = generate_output_command_text
           type_dependant_functions['output_dir'] = generate_output_dir_command_text
           type_dependant_functions['boolean'] = generate_boolean_command_text
       end_def

       action update()
           is_optional = False
           for each option in info.required_opts:
               type_dependant_functions[option.type](option)
           end_for
           is_optional = True
           for each option in info.optional_opts:
               type_dependant_functions[option.type](option)
           end_for
           command_text = uncompress_command + command_text + compress_command
       end_action
action generate_text_command_text(option: OptionInfo)
str = option.get_command_line_string()
if option.is_short_command_line() then:
    str = str + ' '
else:
    str = str + '='
end_if
str = str + '$' + option.name
if is_optional then:
    str = '\n#if str($' + option.name + '):\n' + str + '\n#end if\n'
end_if
command_text = command_text + str
end_action

action generate_data_select_command_text(option: OptionInfo)
str = option.get_command_line_string()
if option.is_short_command_line() then:
    str = str + ' '
else:
    str = str + '='
end_if
str = str + '$' + option.name
if is_optional then:
    str = '\n#if str($' + option.name + ') != "None":\n' + str + '\n#end if\n'
end_if
command_text = command_text + str
end_action

action generate_integer_float_command_text(option: OptionInfo)
str = option.get_command_line_string()
if option.is_short_command_line() then:
    str = str + ' '
else:
    str = str + '='
end_if
str = str + '$' + option.name
if is_optional then:
    str = '\n#if $' + option.name + ':
' + str + '\n#end if\n'
end_if
command_text = command_text + str
end_action

action generate_boolean_command_text(option: OptionInfo)
command_text = command_text + '\n#if $' + option.name + ':
' + option.get_command_line_string() + '\n#end if\n'
end_action

action generate_repeat_command_text(option: OptionInfo)
str = ''
if not list_dict_to_string_defined then:
    str = LIST_DICT_TO_STRING_FUNCTION
    list_dict_to_string_defined = True
end_if
str = option.get_command_line_string()
if option.is_short_command_line() then:
    str = str + ' '
else:
    str = str + '='
end_if
str = str + '($list_dics_to_string($input_files_$' + option.name + ')')
if is_optional then:
    str = '\n#if $input_files_' + option.name + ':
' + str + '\n#end if\n'
end_if
command_text = command_text + str
end_action

action generate_output_command_text(option: OptionInfo)
    str = option.get_command_line_string()
    if option.is_short_command_line() then:
        str = str + ' ' 
    else:
        str = str + '='
    end_if
    str = str + '$' + option.name
    command_text = command_text + str
end_action

action generate_output_dir_command_text(option: OptionInfo)
    if compress_command != '' then:
        throw_exception('Two options which generate a directory as output is not allowed!')
    end_if
    output_dir_name = info.id + '_output'
    str = option.get_command_line_string()
    if option.is_short_command_line() then:
        str = str + ' ' 
    else:
        str = str + '='
    end_if
    str = str + output_dir_name
    compress_command = '\ncompress_path.py -i ' + output_dir_name + ' -o ' + option.name + ' \n'
    command_text = command_text + str
end_action

action generate_input_dir_command_text(option: OptionInfo)
    if uncompress_command != '' then:
        throw_exception('Two options which generate a directory as input is not allowed!')
    end_if
    input_dir_name = info.id + '_input'
    str = option.get_command_line_string()
    if option.is_short_command_line() then:
        str = str + ' ' 
    else:
        str = str + '='
    end_if
    str = str + input_dir_name
    compress_command = '\nnuncompress_path.py -i ' + option_name + ' -o ' + input_dir_name + ' \n'
    command_text = command_text + str
end_action

end_methods
end_class
class attributes
    info: ScriptInfo
    doc: xml.dom.minidom.Document
    inputs: xml.dom.minidom.Node
    outputs: xml.dom.minidom.Node
    is Optional: boolean
    type_dependant_functions: dictionary
end_attributes

methods
        info = info_scr
        doc = doc_xml
        inputs = inputs_node
        outputs = outputs_node

        type_dependant_functions = empty_dictionary()
        type_dependant_functions[‘text’] = generate_text_data_attributes
        type_dependant_functions[‘integer’] = generate_integer_float_attributes
        type_dependant_functions[‘float’] = generate_integer_float_attributes
        type_dependant_functions[‘select’] = generate_select_attributes
        type_dependant_functions[‘multiple_select’] = generate_multiple_select_attributes
        type_dependant_functions[‘data’] = generate_text_data_attributes
        type_dependant_functions[‘input_dir’] = generate_input_dir_attributes
        type_dependant_functions[‘repeat’] = generate_repeat_attributes
        type_dependant_functions[‘output’] = generate_output_attributes
        type_dependant_functions[‘output_dir’] = generate_output_attributes
        type_dependant_functions[‘boolean’] = generate_boolean_attributes
    end_def

    action update()
        is Optional = False
        for each option in info.required_opts:
            type_dependant_functions[option.type](option)
        end_for
        is Optional = True
        for each option in info.optional_opts:
            type_dependant_functions[option.type](option)
        end_for
    end_action

    action generate_integer_float_attributes(option: OptionInfo)
        param = add_tag(inputs, ‘param’)
        add_attribute(param, ‘name’, option.name)
        add_attribute(param, ‘type’, option.type)
        add_attribute(param, ‘label’, replace(option.label, ‘%default’, string(option.default))
        add_attribute(param, ‘optional’, string(is Optional))
        if option.has_default() then:
            add_attribute(param, ‘default’, option.default)
        end if
        if not is Optional then:
            add_attribute(param, ‘value’, ‘0’)
        end if
    end_action
action generate_text_data_attributes(option: OptionInfo)
    param = add_tag(inputs, 'param')
    add_attribute(param, 'name', option.name)
    add_attribute(param, 'type', option.type)
    add_attribute(param, 'label', replace(option.label, '%default', string(option.default)))
    add_attribute(param, 'optional', string(is_optional))
    if option.has_default():
        add_attribute(param, 'default', option.default)
    end_if
end_action

action generate_input_dir_attributes(option: OptionInfo)
    param = add_tag(inputs, 'param')
    add_attribute(param, 'name', option.name)
    add_attribute(param, 'type', 'data')
    add_attribute(param, 'label', replace(option.label, '%default', string(option.default)))
end_action

action generate_select_attributes(option: OptionInfo)
    param = add_tag(inputs, 'param')
    add_attribute(param, 'name', option.name)
    add_attribute(param, 'type', option.type)
    add_attribute(param, 'label', replace(option.label, '%default', string(option.default)))
    add_attribute(param, 'optional', string(is_optional))
    if option.has_default():
        opt = add_tag(param, 'option')
        add_attribute(opt, 'value', 'None')
        add_attribute(opt, 'selected', 'True')
        add_text(opt, 'Selection is optional')
    end_if
    for each choice in option.choices:
        opt = add_tag(param, 'option')
        add_attribute(opt, 'value', choice)
        add_text(opt, choice)
    end_for
end_action

action generate_multiple_select_attributes(option: OptionInfo)
    param = add_tag(inputs, 'param')
    add_attribute(param, 'name', option.name)
    add_attribute(param, 'type', option.type)
    add_attribute(param, 'label', replace(option.label, '%default', string(option.default)))
    add_attribute(param, 'optional', string(is_optional))
    add_attribute(param, 'multiple', 'True')
    if option.has_default():
        opt = add_tag(param, 'option')
        add_attribute(opt, 'value', 'None')
        add_attribute(opt, 'selected', 'True')
        add_text(opt, 'Selection is optional')
    end_if
    for each choice in option.choices:
        opt = add_tag(param, 'option')
        add_attribute(opt, 'value', choice)
        add_text(opt, choice)
    end_for
end_action
action generate_repeat_attributes(option: OptionInfo)
    repeat = add_tag(inputs, 'repeat')
    add_attribute(repeat, 'name', 'input_files_' + option.name)
    add_attribute(repeat, 'title', option.name)
    add_attribute(repeat, 'optional', string(is_optional))
    param = add_tag(repeat, 'param')
    add_attribute(param, 'name', 'additional_input')
    add_attribute(param, 'type', 'data')
    add_attribute(param, 'label', replace(option.label, '%default', string(option.default)))
end_action

action generate_output_attributes(option: OptionInfo)
    data = add_tag(outputs, 'data')
    add_attribute(data, 'name', option.name)
    add_attribute(data, 'format', option.format)
end_action

action generate_boolean_attributes(option: OptionInfo)
    param = add_tag(inputs, 'param')
    add_attribute(param, 'type', option.type)
    add_attribute(param, 'name', option.name)
    add_attribute(param, 'label', replace(option.label, '%default', string(option.default)))
    add_attribute(param, 'selected', option.default)
end_action

end_methods
end_class

function generate_xml_string(info: ScriptInfo) returns str: string
    doc = xml.dom.minidom.Document()
    tool = add_tag(doc, 'tool')
    add_attribute(tool, 'id', info.id)
    add_attribute(tool, 'name', info.name)
    add_attribute(tool, 'version', info.version)
    description = add_tag(tool, 'description')
    add_text(description, info.description)
    command = add_tag(tool, 'command')
    com_gen = CommandGenerator(info)
    com_gen.update()
    add_text(command, com_gen.command_text)
    inputs = add_tag(tool, 'inputs')
    outputs = add_tag(tool, 'outputs')
    xml_opt_gen = XmlOptionsAttributesGenerator(info, doc, inputs, outputs)
    xml_opt_gen.update()
    help = add_tag(tool, 'help')
    add_text(help, info.help)
    str = get_xml_string(doc)
end_function

action make_xml(script_fp: string, output_dir: string, remove_opts: string)
    script_name = get_base_name(script_fp)
    script_command = get_base_name_plus_extension(script_fp)
    script = import(script_name)
    info = InfoScript(script.script_info, script_name, script_command)
    info.remove_opts(remove_opts)
    xml_string = generate_xml_string(info)
    write_file(output_dir, xml_string)
end_action
**tgz_manager.py**

```python
action compress_to_tgz(in_path: string, tgz_name: string)
    file = tarfile.open(tgz_name)
    file.add(in_path)
    file.close()
end_action

action extract_from_tgz(tgz_file: string, output_path: string)
    if not is_tgz_file(tgz_file) then:
        throw_exception('Not a tgz file')
    end_if
    file = tarfile.open(tgz_file)
    if number_of_files(tgz_file) > 1 then:
        file.extract(output_path)
    else:
        file.extract()
        move(file.getNames()[0], output_path)
    end_if
    file.close()
end_action
```
Bibliography

8. Stephenson, N., In the beginning... was the command line. 1999: Harper Perennial.