GAAF : Genome Assembly Analysis Framework

<u>Manual</u>

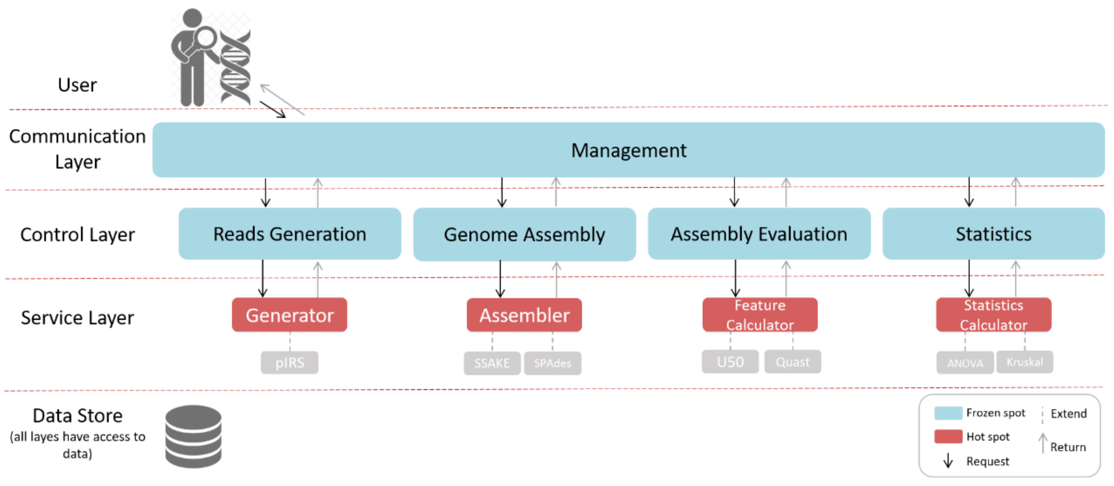
Contents

This document aims to introduce GAAF, its objective, functionality and extensibility.

GAAF is an analysis framework, in the context of Genome Assembly domain, which aims to facilitate the process of post analysis. It emerged from the issue related to the quality interpretation of assembled genome data.

Architecture

GAAF architecture is divided into five main modules: Management, Reads Generation, Genome Assembly, Assembly Evaluation and Statistics.



Management: all configuration, inputs and outputs are managed through a main core module called Management. It works as the main line of communication between all other frozen spots, and between User and GAAF. Such as a Manager in an Enterprise, it acts as a communicator, passing orders to Control Layer.

Reads Generation: as the name mentions, this Module is responsible for generating artificial reads. The way it is done depends on the chosen algorithm, e.g. pIRS (Hu et al.

2012). The Module may receive or not a genome reference entry. When no input is available, the reads are randomly generated, according to each algorithm. In addition, it could also receive raw reads, in order to filter them in a quality trimming hot spot, or by modifying them in any purposes. Reads Generation, as well as all the other modules from Control Layer, may be seen as a Team Leader, who receives orders from Manager and passes those to specific employees.

Genome Assembly: this is the module where the reads are assembled. It may output contigs, or scaffolds, which may be analyzed on Assembly Evaluation Module, or may work as re-input to Genome Assembly, calling hot spots capable of scaffolding, gap filling etc. It works with distinct assemblers, file formats and sequencing technologies.

Assembly Evaluation: the metrics or features qualifying the assemblies are generated in the Assembly Evaluation Module. Not only the evaluation metrics, but also some other post-analysis could be included here, such as genome comparisons. Those results, according to each application, may be still analyzed in external modules, like in a Statistical Module.

Statistics: as a Team Leader from a group of statisticians, Statistics Module organizes feature data and request its service modules to test and create graphs of that data.

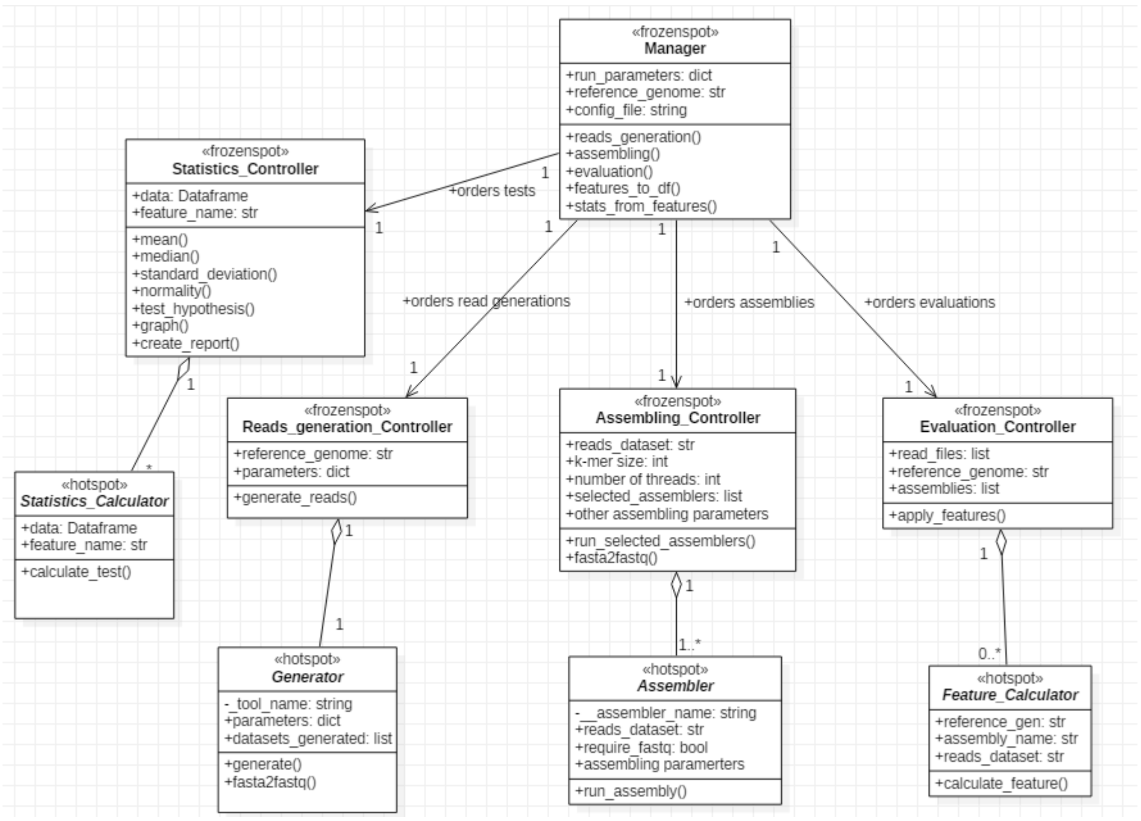
Generator: an abstraction of a service from Reads Generation Module is the Generator Module. Hypothetically, a freelancer, according to Client's request, who is hired to generate reads.

Assembler: another service requested is assembling. Maybe, Assembler Module is the most important service we have. It varies in the way of working, how outputs and how receive input data.

Feature Calculator: a good Calculator Module is needed to calculate the features. Feature Calculator may be a single class capable of calculating a single feature or may be a complex module which calculates many features at once, such as QUAST.

Statistics Calculator: finally we have a module where all statistical tests and graphs are calculated and created, respectively. Statistics Calculator may be extended in a large variety of statistics.

Classes:



The five frozen spots modules are presented in concrete classes. The hot spots are represented through abstract classes, which will be extended into personalized concrete classes. The above presents the Class Diagram, that directly reflects the architecture. More details may be find in code documentation.

Manager: deals with the communication between user and classes, and between the classes. The Class includes the methods to generate reads, assemble reads, evaluate assemblies, load features into dataframes, and analyze those dataframes through statistical tests. It receives a dictionary called run_parameters containing some parameters passed by user and also a config file where details about the experiment are described. It also may receive a reference genome.

Reads_Generation_Controller: calls algorithms to generate artificial reads or to filter raw reads, given or not a reference genome input. Again a *parameters* dictionary is passed.

Generator: is an abstract class to read-generation tools. It reads a dictionary containing all the needed parameters, such as Coverage, read length, reference genome, phred score, mutation rate, sequencing technology and others. During the process of generation, the class store the sample-read names into a list.

Assembling_Controller: is responsible for managing all the assemblies required, given the chosen assemblers. It basically receives the reads, the k-mer size, and number of threads, and runs the software (extended in Assembler).

Assembler: is an abstract class to the assembler software. The way each software works is implemented in a concrete class extending the Assembler Class. And it shall be invisible to the Assembling_Controller. In addition, some assembler may require fastq reads.

Evaluation_Controller: calls algorithms to evaluate the assemblies. It may call complete tools, such as QUAST, or calculate individual implemented Features. It receives an assembly and may receive a reference genome and sequencing reads.

Feature_Calculator: is an abstract class to calculate specific metrics over the assembly. It receives an assembly, and can receive reference genome and read files.

Statistics_Controller: is another concrete class responsible for analyzing whole experiments. It has some methods already implemented in its core, such as mean and median, but generally call Statistics_Calculator to calculate certain statistical tests.

Statistics_Calculator: is an abstract class to implement statistical tests or graphs.

Extensibility:

The greatest advantage about using a framework is the easy way of how a new item is added. For example, given the architecture figure, image that any gray box can be removed, and any new gray box can be added. In Genome Assembly, we could freely add new assemblers, and then have all the old assemblies plus the new ones, in a finished experiment.

We now describe how to add new classes to each module.

Generator

We provide pIRS code as an example. Create your class without the__init__() method, receiving Generator as a Parameter of your class. Then, define generate() method, doing whatever it needs to do. You may create extra methods you desire:

```
import os
import logging
import threading as thr
import time
from reads_generation import Generator

class Pirs(Generator):
    """Class responsable for calling pIRS and generating its reads

    Methods
    -------
    command(sample)
        Run the Pirs command to the sample
    """

    def generate(self,sample):
        """
        Run the pIRS command.

        Parameters
        ----------
        sample : str
            The sample name. It names the read files.
        """
        if "." in sample:
            p=sample.find(".")
            sample=sample[0:p]

        if self.parameters['ref'] == '':
            logging.error('There`s no reference file attached')
            exit()
        try:
            if not (os.path.exists(self.out+"reads")):
                os.system("mkdir "+self.out+"reads")

            #In pIRS one can not specify a unique phred value. For that reason,
            #here we convert it.
            if "error_rate" in sample or "Error_rate" in sample:
                command = "pirs simulate -x " + str(self.parameters['coverage']) + " -l
" + str(self.parameters['read_len']) + " -v " +str(self.parameters['var'])+" -m " +
str(self.parameters['read_len']*2) + " --no-indels -e "+self.parameters['Error_rate']+"
--no-gc-bias -o "+self.out+"reads -t "+str(self.t)+" -s "+sample + "
"+self.parameters['ref'] + " | tee -a " +self.out+self.exp+ ".log"
                os.system(command)

            else:
               command = "pirs simulate -x " + str(self.parameters['coverage']) + " -l "
+ str(self.parameters['read_len']) + " -v " +str(self.parameters['var'])+" -m " +
str(self.parameters['read_len']*2) + " --no-indels --no-subst-errors --fasta -e 0 --no-
gc-bias -o "+self.out+"reads -t "+str(self.t)+" -s "+sample + " "+self.parameters['ref']
+ " | tee -a " +self.out+self.exp+ ".log"
                os.system(command)
            self.datasets_generated.append(sample)

        except IOError:
            logging.error(IOError)
```

Beside the methods you created, Pirs already has fasta2fastq() method, and exp, out, datasets_generated, a dictionary of parameters, and logging attributes. Exp gives you the name of the experiment and out the output dir to store the reads. It is recommended to create "/reads" dir inside out. It is important: save the sample names created into a predefined list called datasets_generated. It will be used further by other modules.

Finally, the parameters dictionary gives the class all experiment parameters passed by user. Not all of them may be used. It is up to you. At least, you will probably use the following keys: ref, coverage, read_len, var (meant read normal variation), phred, Error_rate.

Assembler

We provide Abyss code as an example. Create your class without the__init__() method, receiving Assembler as a Parameter of your class. Then, define run_assembly() method, doing whatever it needs to do. You may create extra methods you desire:

```
import os
import logging
from assembly import Assembler

class Abyss(Assembler):
    """
    Abyss assembler

    Attributes
    ----------
    __assembler_name : str
```

```
        The name of the assembler tool
    require_fastq : bool
        if the assembler only work with fastq files, please set as True
        (default False)
    python_threads : bool
        in case the assembler do not use multithread, you can at least
        activate it to run with python threads. Those threads are used
        by Assembly Module (default False).

    Methods
    -------
    run_assembly()
        Run the assembly
    """

    __assembler_name='abyss'
    require_fastq=False
    python_threads=False


    def run_assembly(self):
        """
        Run the assembly. By the moment, it only works with Illumina.
        """
        if not(os.path.exists(self.out+"assemblies/"+self.__assembler_name)):
            os.system("mkdir "+self.out+"assemblies/"+self.__assembler_name)
        os.system("mkdir "+self.out+"assemblies/"+self.__assembler_name+"/"+self.sample)
        try:
            os.system("cp "+self.out+"reads/"+self.sample+"_1."+self.file_format+" "
+self.out+"assemblies/abyss/"+self.sample)
            os.system("cp "+self.out+"reads/"+self.sample+"_2."+self.file_format+" "
+self.out+"assemblies/abyss/"+self.sample)
            command='abyss-pe k='+ str(self.k) +' name='+self.sample+'
in=\''+self.sample+'_1.'+self.file_format+' '+self.sample+'_2.'+self.file_format+'\'  --
directory='+self.out +'assemblies/abyss/'+self.sample+' j='+str(self.t)+' | tee -a '
+self.out + self.exp+ '.log'
            os.system(command)
        except IOError:
            logging.error(IOError)
            exit()
```

Feature Calculator

We provide QUAST code as an example. Create your class without the__init__() method, receiving Feature_Calculator as a Parameter of your class. Then, define calculate() method, doing whatever it needs to do. You may create extra methods you desire:

```
import os
import logging
from evaluation import Feature_Calculator

class Quast(Feature_Calculator):
    """Quast tool for Genome Assembly Evaluation

    Attributes
    ----------
    reads_format : str
            the format of the reads, generally fa or fq
    reference : str
            Genome Reference
    assembly : str
            Path to the assembly

    Methods
    -------
    run(sample,t, assembler)
        Runs quast command
    """
    reads_format="fa"
    reference=''
    assembly=''


    def calculate(self,sample,t, assembler):
        """
        It runs quast, and outputs report.tex.

        Parameters
        ----------
        sample : str
            Sample name
        t : int
            Number of threads
        assembler : str
            Assembler name
        """

        try:
            if not (os.path.exists(self.out+"features/quast/"+sample+"--"+assembler)):
                os.system("mkdir "+self.out+"features/quast/"+sample+"--"+assembler)
            logging.info(" Calling Quast tool")
            command = "python3 quast/quast.py -r "+self.ref+" -t "+str(t)+" -o " +
self.out+ "features/quast/"+sample+"--"+assembler+" --glimmer --rna-finding -b -1
"+self.out+"reads/"+sample+"_1."+self.reads_format+" -2
"+self.out+"reads/"+sample+"_2."+self.reads_format+" "+self.assembly+" | tee -a " +
self.out+self.exp+ ".log"
            os.system(command)

        except IOError:
            logging.error(IOError)
```

Statistics Calculator

We provide Kruskal code as an example. Create your class without the__init__() method, receiving Statistics_Calculator as a Parameter of your class. Then, define calculate_test() method, doing whatever it needs to do. You may create extra methods you desire:

```
from scipy import stats
from statistics import Statistics_Calculator
class Kruskal(Statistics_Calculator):
    def calculate_test(self):
        df=self.data.copy()
        k=stats.kruskal(*df.T.values)
        self.results.write("\nKruskal to columns "+str(df.columns)+str(k)+"\n")
        print("\nKruskal to columns "+str(df.columns)+str(k)+"\n")

        k2=stats.kruskal(*df.values)
        self.results.write("\nKruskal to rows "+str(df.index)+str(k2)+"\n")
        print("\nKruskal to rows "+str(df.index)+str(k2)+"\n")

        return (k,k2)
```

Installation

You do not need to install GAAF in order to use it. However, some dependencies are required:

```
python >=2.7
biopython  (pip install biopython)
```

```
scipy       (pip install scipy)
pandas      (pip install pandas)
seaborn     (pip install seaborn)
scikit      (pip install scikit-posthocs)
```

Nonetheless, whether you want to use our extended classes, you may install them through:

```
sudo install_extentions.sh
```