wespipeline

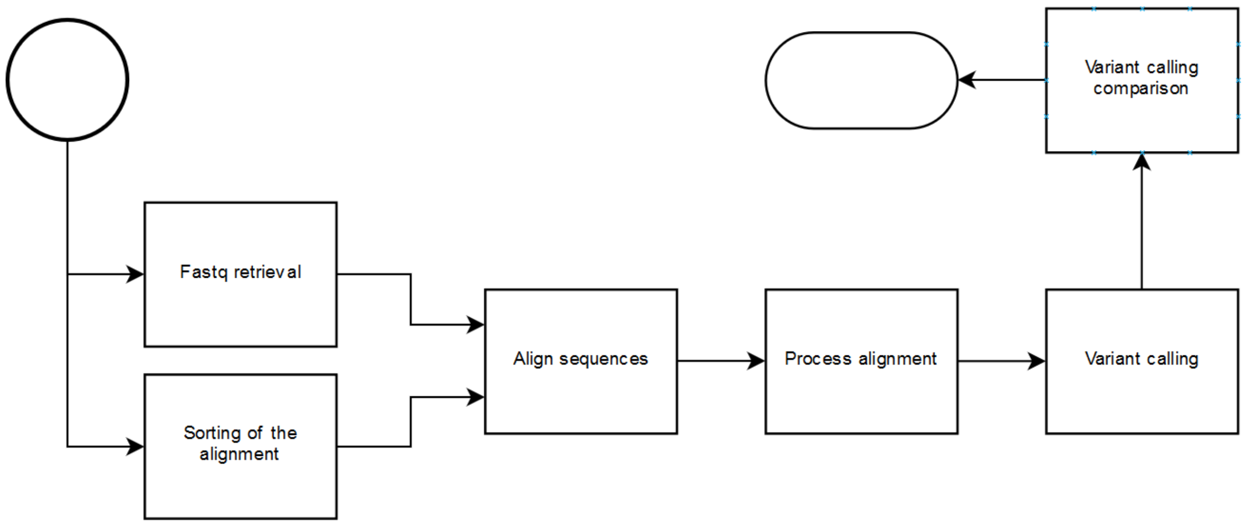
Release 0.0.2

Alejandro Rodríguez Díaz

Jul 04, 2019

USING THE PIPELINE

An implementation of a whole exome analysis pipeline using Luigi for workflow management.



This package provides with the implementation of tasks for executing partial or complete variant calling analysis with the advantages of having a workflow manager: dependency resolution, execution planner, modularity, monitoring and historic.

Documentation for the latest version is being hosted by readthedocs

CHAPTER

ONE

INSTALLATION

Wespipeline is available through pip, conda and manual installation. Install it from the package repositories `pip3 install wespipeline conda install -c jancho wespipeline`, or download the project and build from source: `git clone https://github.com/Janchorizo/wespipeline.git && cd wespipeline && python3 setup.py install`.

Notice that executing the analysis will involve different additional dependencies depending on the steps that executed and the parameters set for these. All possible are cited below and can be downloaded with the Anaconda distribution:

- Secuence retrieval : Sra Toolkit, Fastqc
- Reference genome retrieval : No needed dependency
- Secuence alignment : Bwa
- Alignment processing : Bwa Samtools,
- Variant calling : Freebayes, Varscan, Gatk, Deepvariant
- Variant calling evaluation : Vcf tools

In addition to the dependencies, conda can be used for installing the *wespipeline* package. An example for installing the miniconda distribution, the package and the dependencies is:

```
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O ~/
↪miniconda.sh
bash ~/miniconda.sh -b -p $HOME/miniconda
export PATH="$HOME/miniconda/bin:$PATH"
source $HOME/miniconda/bin/activate && \
    conda config --add channels bioconda && \
    conda config --add channels conda-forge && \
    conda config --add channels jancho && \
    conda install -y samtools && \
    conda install -y bwa && \
    conda install -y picard && \
    conda install -y platypus-variant && \
    conda install -y varscan && \
    conda install -y freebayes && \
    conda install -y fastqc && \
    conda install -y sra-tools && \
    conda install -y wespipeline

rm ~/miniconda.sh
```

CHAPTER

TWO

GETTING STARTED

Installing or downloading the package will provide with a higher level task per step of the analysis, each of which can be executed in a similar fashion to other Luigi tasks.

Each of the six steps have a higher level task that can be scheduled in a similar fashion to other Luigi tasks:

```
python3 -m luigi --module wespipeline.<module> <Taskname> --<Taskname>-param value
```

Download the sequences using the NCBI accession number.

```
python3 -m luigi --module wespipeline.fastq FastqRetrieval \
        --FastqRetrieval-paired-end true \
        --FastqRetrieval-accession-number SRR9209557 \
        --FastqRetrieval-create-report true
```

Or an external url.

```
python3 -m luigi --module wespipeline.fastq FastqRetrieval \
        --FastqRetrieval-paired-end true \
        --FastqRetrieval-compressed false \
        --FastqRetrieval-accession-number SRR9209557 \
        --FastqRetrieval-create-report true
```

Download the reference genome and create a report using FastqC.

```
python3.6 -m luigi --module tasks.reference ReferenceRetrieval
        --workers 3 \
        --ReferenceGenome-ref-url ftp://hgdownload.cse.ucsc.edu/goldenPath/hg19/
↪bigZips/hg19.2bit \
        --ReferenceGenome-from2bit True \
        --GlobalParams-base-dir ./tfm_experiment \
        --GlobalParams-log-dir .logs \
        --GlobalParams-exp-name hg19
```

Or run the whole analysis, specifying the parameters for each of the steps.

```
python3 -m luigi --module tasks.vcf VariantCalling
        --workers 3
        --VariantCalling-use-platypus true
        --VariantCalling-use-freebayes true
        --VariantCalling-use-samtools false
        --VariantCalling-use-gatk false
        --VariantCalling-use-deepcalling false
        --AlignProcessing-cpus 6
        --FastqAlign-cpus 6
```

(continues on next page)

(continued from previous page)

```
        --FastqAlign-create-report True
        --GetFastq-gz-compressed True
        --GetFastq-fastq1-url ftp://ftp-trace.ncbi.nih.gov/giab/ftp/data/NA12878/
↪Garvan_NA12878_HG001_HiSeq_Exome/NIST7035_TAAGGCGA_L001_R1_001.fastq.gz
        --GetFastq-fastq2-url ftp://ftp-trace.ncbi.nih.gov/giab/ftp/data/NA12878/
↪Garvan_NA12878_HG001_HiSeq_Exome/NIST7035_TAAGGCGA_L001_R2_001.fastq.gz
        --GetFastq-from-ebi False
        --GetFastq-paired-end True
        --ReferenceGenomeRetrieval-ref-url ftp://hgdownload.cse.ucsc.edu/goldenPath/
↪hg19/bigZips/hg19.2bit --ReferenceGenomeRetrieval-from2bit True
        --GlobalParams-base-dir ./tfm_experiment
        --GlobalParams-log-dir .logs
        --GlobalParams-exp-name hg19
```

CHAPTER

THREE

TASKS IMPLEMENTED

Module	Task
reference	ReferenceGenomeRetrieval
fastq	FastqRetrieval
align	FastqAlignment
processalign	FastqProcessing
variantcalling	VariantCalling
processalign	VariantProcessing

CHAPTER

FOUR

ACKNOWLEDGEMENTS

Special thanks to professor Luis Antonio Miguel Quintales for all the guidance and help provided during the development of this project.

CHAPTER

FIVE

WESPIPELINE : A WHOLE EXOME SECUENCING VARIANT CALLING PIPELINE

5.1 Requirements

5.1.1 Python dependencies

Wespipeline depends on an existing installation of the library Luigi. If this package was installed following the recomended steps, this dependency should be fulfilled.

5.1.2 External dependencies

Whole exome sequencing variant calling analysis needs for external programs for doing both the processing, and the analysis and summaries. Following is a list of the different dependencies.

Optionally, some type of database is needed for making use of the persistent storage of executions. In the configuration proposed, is used.

Even though pip packages dependencies are resolved upon installation, third party tools are not. These extra dependencies are *not compulsary for all executions of the pipeline*, but depend on the parameters and tasks selected.

Each of the dependencies correspond to a specific need in one or more of the steps, and thus are organized in that manner bellow.

- Secuence retrieval : Sra Toolkit, Fastqc
- Reference genome retrieval : No needed dependency
- Secuence alignment : Bwa
- Alignment processing : Bwa Samtools,
- Variant calling : Freebayes, Varscan, Gatk, Deepvariant
- Variant calling evaluation : Vcf tools

5.1.3 Installing through Anaconda distributions

Even though most of the programs listed can be installed through various different ways, I encourage the use of the Anaconda Distribution, one of the biggest platforms for installing the tools from well trusted sources. Optionally, Miniconda can be used too for a lighter version of the package manager.

Installing miniconda is a simple task. Following an example installation for a x64 linux machine:

```
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O ~/
↪miniconda.
bash ~/miniconda.sh -b -p $HOME/miniconda && rm ~/miniconda.sh
export PATH="$HOME/miniconda/bin:$PATH"
```

Beware that, in order for the utilities and installed packages to be accessible the environment must be activated:

```
source $HOME/miniconda/bin/activate
```

The package archive is distributed through different channels, two of which are needed for the installation of these packages. Easier than specifying the channel for each command is adding the channels:

```
conda config --add channels bioconda
conda config --add channels conda-forge
```

Installing from the repositories is a simple task doable through one-liner commands. Following is an elaborated list of the installation commands for all of the external depenedencies listed above, and a command for instaling them together:

Installing Samtools

```
conda install -y samtools
```

Installing Bwa

```
conda install -y bwa
```

Installing Picard

```
conda install -y picard
```

Installing Platypus

```
conda install -y platypus-variant
```

Installing Varscan

```
conda install -y varscan
```

Installing Freebayes

```
conda install -y freebayes
```

Installing VCFtools

```
conda install -y vcftools
```

Installing Fastqc

```
conda install -y fastqc
```

Installing Sra Toolkit

```
conda install -y sra-tools
```

Installing all dependencies with a single command:

```
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O ~/
↪miniconda.sh
bash ~/miniconda.sh -b -p $HOME/miniconda
export PATH="$HOME/miniconda/bin:$PATH"
source $HOME/miniconda/bin/activate && \
    conda config --add channels bioconda && \
    conda config --add channels conda-forge && \
    conda install -y samtools && \
    conda install -y bwa && \
    conda install -y picard && \
    conda install -y platypus-variant && \
    conda install -y varscan && \
    conda install -y freebayes && \
    conda install -y vcftools && \
    conda install -y gatk && \
    conda install -y vt

rm ~/miniconda.sh
```

5.2 Installation

Run `` `pip install wespipeline` `` to install the latest stable version from PyPI. Documentation for the latest release is hosted on readthedocs.

To install from source, download the project `` `git clone https://github.com/janchorizo/wespipeline.git` `` and run `` `python3 setup.py install` `` in the root directoy.

5.2.1 Usage

Each of the modules in the package contains tasks for executing a specific setp in the analysis pipline. Use Luigi's typipcal call format for launching the execution of a task:

`` `luigi -m wespipeline.reference GetReference --GlobalParams-exp-name hg19 --workers 2 --local-scheduler` ``

5.3 Configuration

Within the scope of the analysis, there are a set of options that can be set for adjusting the experiment to one's neccesities. There are, however, other configuration options that affect the environment in which the pipeline is executed:

5.3.1 Making Luigi task historic persistent

Luigi offers a web interface for monitoring and analysing the execution of the pipeline. However, it may be the case that a persistent history may be used for later analysis.

For this, Luigi offers a -at the moment beta- option for accessing the excutions through the /history api.

5.4 Running the pipeline

5.4.1 Executing tasks

The execution of the tasks follows the same as other Luigi tasks, using each of the provided modules to execute one of the defined steps of the analysis.

Secuence retrieval

Retrieving the secuencing is a step that takes into account wether the experiment is paired end, or the sources are gz compressed.

Several options can be configured for selecting the source for the exome sequences:

- The origin of the files
- Wether the files are compressed
- Wether the experiment is paired end

Different sources can be set, in which case the one used for retrieving will be the following:

1. Files accessible localy
2. Files specified by the NCBI accession number.
3. External sources set by their url

Warning: This task will fail along with the upstream tasks if the quality report is selected but Fastqc is not installed.

In the case of using the NCBI accession number, the `compressed` is ignored as it is already handled with the `fastq-dump` tool from the *Sra toolkit*.

Warning: This task will fail along with the upstream tasks if the accession number is used but Sra toolkit is not installed.

The execution of this step with all the posible parameters is the following:

```
luigi --module wespipeline.fastq GetFastq \
   --GetFastq-paired-end true \
   --GetFastq-compressed true \
   --GetFastq-fastq1-local-file ./experiment_1.fastq.gz \
   --GetFastq-fastq2-local-file ./experiment_2.fastq.gz \
   --GetFastq-accession_number SRRXXXXXX
   --GetFastq-fastq1-url ftp.archive.x/exp/exome_1.fastq.gz \
   --GetFastq-fastq2-url ftp.archive.x/exp/exome_2.fastq.gz
```

Note: Eventhough parameters for the selected task `GetFastq` can be set directly, it is encourage to prepend the task name in order to keep the call consistent and distinguish them from parameters set to other tasks.

Note: Note that GloablParams is used for setting common pipeline wide parameters.

This task is used required by the `Align` task. If the secuences are available locally, set the `fastqx-local-file` parameter to the correspondent path; for instance, for a set of fastq files relative to the current directory that don't require to be uncompressed the following command would be used:

```
luigi --module wespipeline.fastq GetFastq \
   --GetFastq-paired-end true \
   --GetFastq-fastq1-local-file ./experiment_1.fastq \
   --GetFastq-fastq2-local-file ./experiment_2.fastq
```

Reference genome retrieval

The reference genome, essential for more steps than the alignment, may be obtained in 2bit format (and then converted to fa).

Secuence alignment

Aligning the secuencing against the reference genome is a process that produces a not sorted, nor indexed, sam file.

Alignment processing

Processing the alignment includes sorting, indexing and removing duplicates from the original alignment. In the end, it produces a bam and bai file.

Variant calling

Variants can be obatined with different tools; each of which depends in the reference genome retrieval (`wespipeline.reference.ReferenceGenome`) and the align processing step (`wespipeline.processalign.AlignProcessing`).

The desired tools to be used are specified as boolean parameters, from a total of five eligible:

- Platypus
- Freebayes
- DeepVariant
- Gatk
- Samtools

Warning: DeepVariant requires Docker to be installed. Additionally, it needs the Python Docker package for Luigi to interact with it; this last one is a dependency specified in the package, so it will be automatically installed if *wespipeline* is installed with a package manager.

An example for using DeepVariant on reference genome, and bam files located in the current directory would be the following:

```
python3 -m luigi --module wespipeline.vcf VariantCalling
--VariantCalling-use-deepvariant True  \
--VariantCalling-cpus 2  \
--ReferenceGenome-reference-local-file hg19.fa  \
--AlignProcessing-no-dup-bam-local-file hg19_nodup.bam  \
--AlignProcessing-no-dup-bai-local-file hg19_nodup.bam.bai  \
--GlobalParams-exp-name hg19  \
--GlobalParams-base-dir .  \
--GlobalParams-log-dir . \
```

Variant calling evaluation

Variant calling comparation and estatistical summaries for the variants identifyed.

5.4.2 Global vs task specific parameters

Luigi provides a convenient way to expose a task' parameters both for Python code task instancetiating, and command line usage. The modular approach taken for the design of the pipeline

5.4.3 Whole analysis example

The following command allows to execute the pipeline for. . .

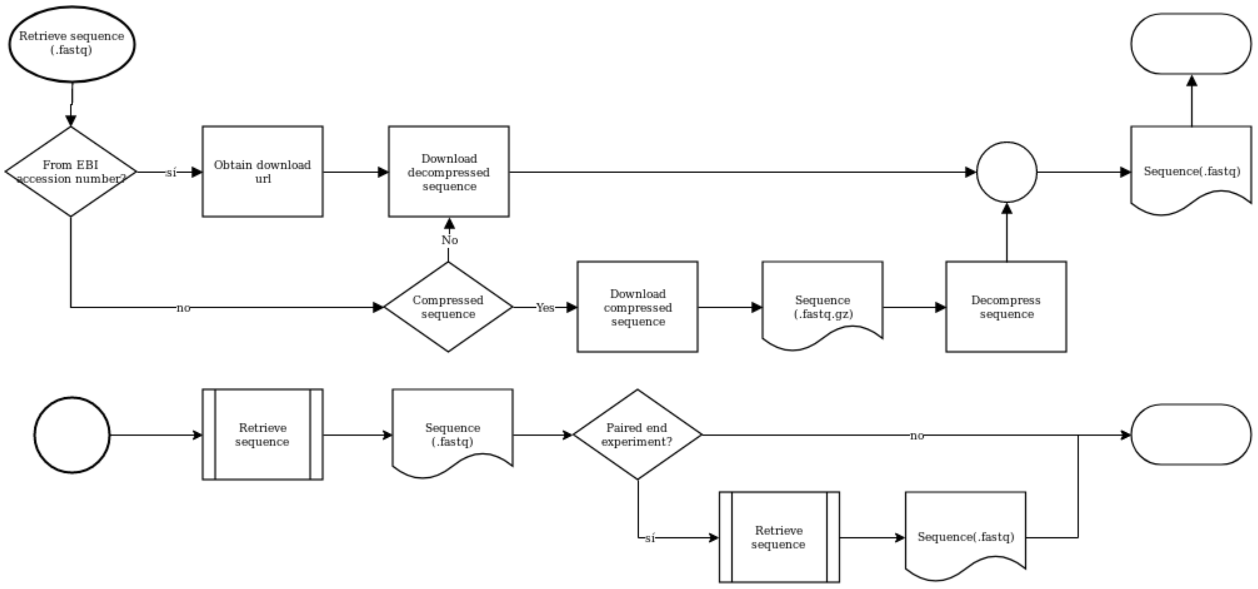
```
nohup python3.6 -m luigi --module wespipeline.vcf_analysis VariantCallingAnalysis \
--workers 3 \
--VariantCalling-use-platypus true \
--VariantCalling-use-freebayes true \
--VariantCalling-use-samtools false \
--VariantCalling-use-gatk false \
--VariantCalling-use-deepcalling false \
--AlignProcessing-cpus 6 \
--FastqAlign-cpus 6 \
--FastqAlign-create-report True \
--GetFastq-gz-compressed True \
--GetFastq-fastq1-url
ftp://ftp-trace.ncbi.nih.gov/giab/ftp/data/NA12878/Garvan_NA12878_HG001
_HiSeq_Exome/NIST7035_TAAGGCGA_L001_R1_001.fastq.gz \
--GetFastq-fastq2-url
ftp://ftp-trace.ncbi.nih.gov/giab/ftp/data/NA12878/Garvan_NA12878_HG001_HiSeq_Exome/
↪NIST7035_TAAGGCGA_L001_R2_001.fastq.gz \
--GetFastq-from-ebi False \
--GetFastq-paired-end True \
--ReferenceGenome-ref-url
ftp://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/hg19.2bit \ --ReferenceGenome-
↪from2bit True \
--GlobalParams-base-dir ./tfm_experiment \
--GlobalParams-log-dir .logs \
--GlobalParams-exp-name hg19 &
```

5.5 The analysis pipeline

Although the basic pipeline for analyzing whole exome sequencing, as propuested in NCBI, consists of three basic phases (secuencing retrieval, aliengment, and variant calling), more steps are distinguished to make the pipeline more flexible:

- Secuence retrieval
- Reference genome retrieval
- Secuence alignment
- Alignment processing
- Variant calling
- Variant calling evaluation

5.5.1 Secuence retrieval



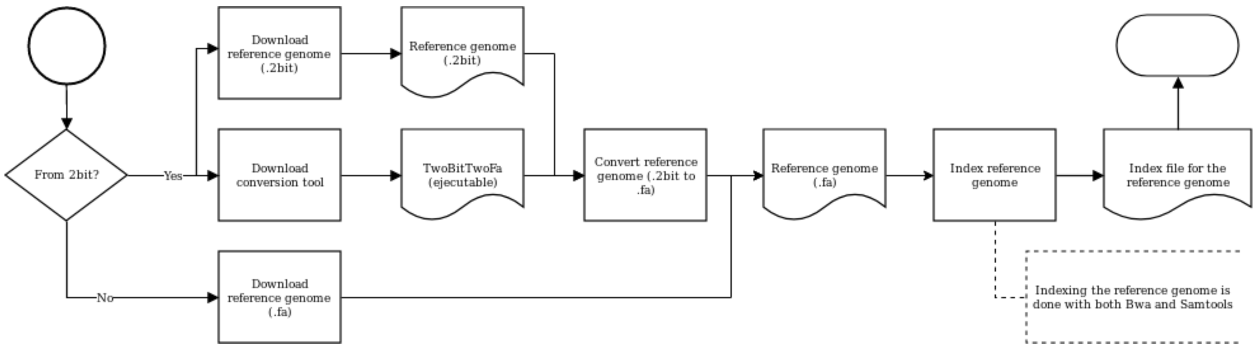
Retrieving the secuencing is a step that takes into account wether the experiment is paired end, or the sources are gz compressed.

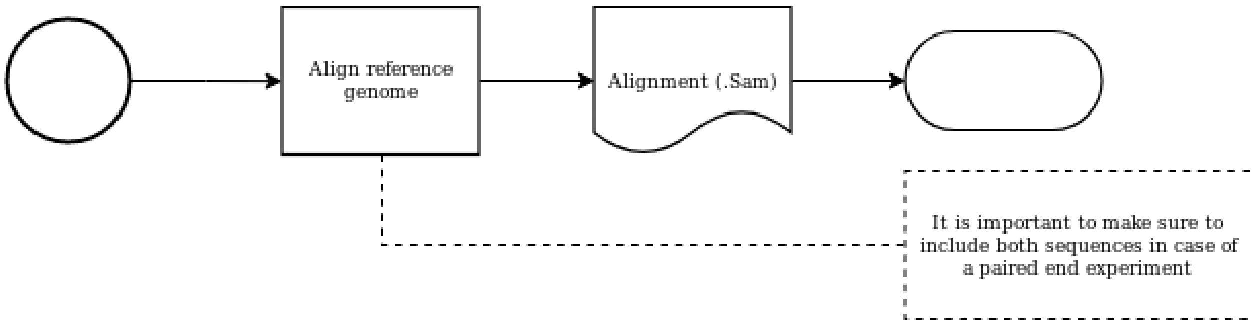
5.5.2 Reference genome retrieval

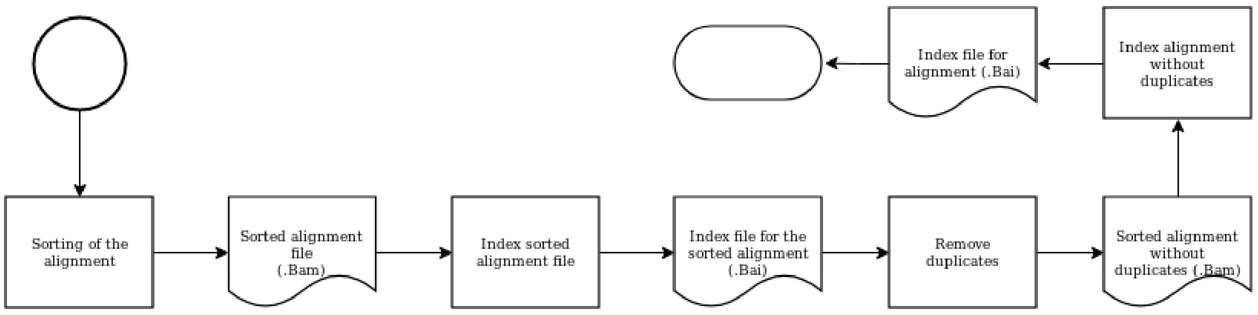
The reference genome, essential for more steps than the alignment, may be obtained in 2bit format (and then converted to fa).

5.5.3 Secuence alignment

Aligning the secuencing against the reference genome is a process that produces a not sorted, nor indexed, sam file.


From 2bit?
Yes
No
Download reference genome (.2bit)
Reference genome (.2bit)
Download conversion tool
TwoBitTwoFa (ejecutable)
Convert reference genome (.2bit to .fa)
Download reference genome (.fa)
Reference genome (.fa)
Index reference genome
Index file for the reference genome
Indexing the reference genome is done with both Bwa and Samtools

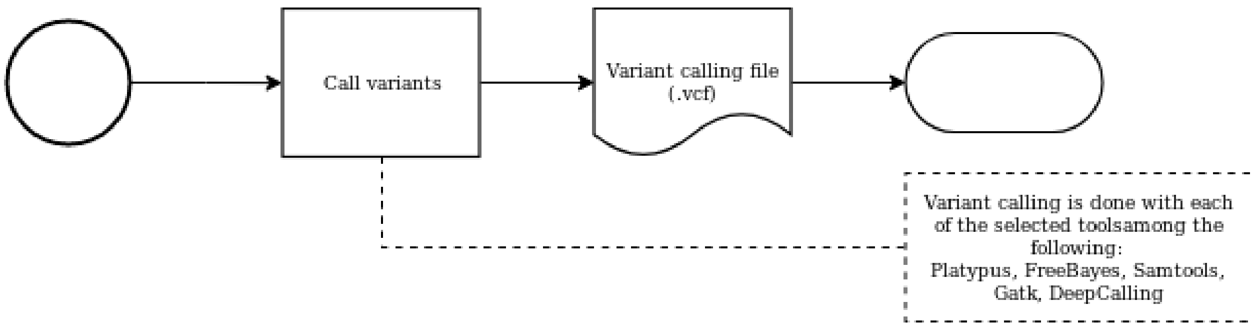

Align reference genome
Alignment (.Sam)
It is important to make sure to include both sequences in case of a paired end experiment


Sorting of the alignment
Sorted alignment file (.Bam)
Index sorted alignment file
Index file for the sorted alignment (.Bai)
Remove duplicates
Sorted alignment without duplicates (.Bam)
Index alignment without duplicates
Index file for alignment (.Bai)

5.5.4 Alignment processing

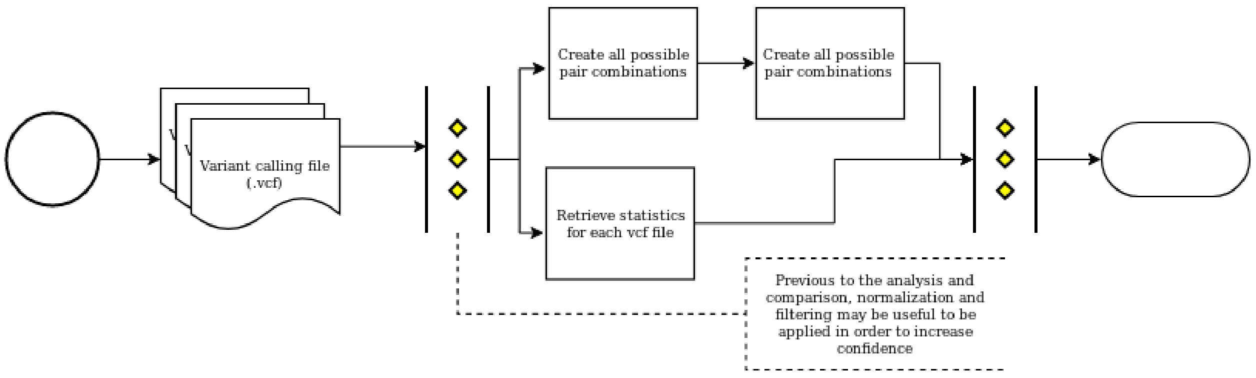
Processing the alignment includes sorting, indexing and removing duplicates from the original alignment. In the end, it produces a bam and bai file.

5.5.5 Variant calling


Call variants
Variant calling file (.vcf)
Variant calling is done with each of the selected toolsamong the following: Platypus, FreeBayes, Samtools, Gatk, DeepCalling

The process of identifying variants within the secuenced exome.

5.5.6 Variant calling evaluation


Variant calling file (.vcf)
Create all possible pair combinations
Create all possible pair combinations
Retrieve statistics for each vcf file
Previous to the analysis and comparison, normalization and filtering may be useful to be applied in order to increase confidence

Variant calling comparation and estatistical summaries for the variants identifyed.

5.6 Case of study

asd

5.7 Steps

5.7.1 Basic Luigi Task class implementation

Luigi's scheduler allows the execution of the task specified and the dependency resolution based on the execution of the *requires* method from the Task implementation.

Any Task in Luigi has the following general structure:

```
class MyTask([luigi.Task, luigi.WrapperTask, luigi.contrib.ExternalProgramTask, ...]):
        param1 = luigi.Parameter(default=[value], description=[string])
        param2 = luigi.Parameter(default=[value], description=[string])
        . . .
        paramN = luigi.Parameter(default=[value], description=[string])

        def requires(self):
                return SomeTask()

                return [
                        SomeTask(param=[value]),
                        OtherTask(),
                ]

                return {
                        'a':SomeTask(param=[value]),
                        'b':OtherTask(),
                }

        def output(self):
                returns luigi.LocalTarget([path])

                returns [luigi.LocalTarget([path]), luigi.contrib.mongodb.MongoTarget]

                returns {
                        'c':luigi.LocalTarget([path]),
                        'd'luigi.contrib.mongodb.MongoTarget
                }
```

- Parameters are exposed in the command line interface, showing the optional argument *description* if provided.
- The requires method can return nothing, a Task instance, or an structure containing Task intances. These tasks will be accessible from within the class through the input method: *self.input()*; which will preserve the structure that was retrieved in the method.
- The output method can return nothing, a luigi.Target instance, or an structure containing Target intances. These targets will be accessible from within the class through the output method: *self.output()*; which will preserve the structure that was retrieved in the method. This is the first method executed,and used to find out if the Task needs to be runned.

Additionally, depending on the specific Task class on which it inherits, the class may have some other specific methods, such as the *run* method for executing Python code, or the *program_args* for returning the arguments for external program executions.

5.7.2 Managing coupling in tasks

The analysis here proposed needs for various tasks within each of the different steps. However, if they are implemented following the previous structure, maintaining and extending the pipeline would we too complicated; let's see this in the following example:

```
import luigi
from somePackage import ExtTask

class FinalTask(luigi.Task):
        param = luigi.Parameter(default=[value])

```

(continues on next page)

(continued from previous page)

```
        def requires(self):
                return IntermediateTask(param=param)

        def output(self):
                returns luigi.LocalTarget(pathA)

        def run(self):
                pass

class IntermediateTask(luigi.Task):
        param = luigi.Parameter()

        def requires(self):
                return InitialTask(param=self.param)

        def output(self):
                returns luigi.LocalTarget(path2)

        def run(self):
                pass

class InitialTask(luigi.Task):
        param = luigi.Parameter()

        def requires(self):
                return ExtTask(param2='1234')

        def output(self):
                returns luigi.LocalTarget(path1)

        def run(self):
                print(self.param1)
```

Here, a couple of problems arise related to the fact that: * Only the first task makes use of the parameter *param*, but all of the previous tasks need to have it in order to pass it.

- If at any point it is needed to change the class required by `` `FinalTask` ``, it would be neccessary to know what input `` `FinalTask` `` expects, what parameters `` `InitialTask` `` needs in order to preserve its interface, and change in `` `FinalTask` `` the name of the class that is being sustituted.
- This last point can be impossible for the case of tasks required in many others, where identitying each place where it is being used is too difficoult.
- This complicates even more when external tasks are imported, such as `` `ExtTask` ``.
- If this pipeline was to be extended, it would be necessary to know in advance what parameters and results would the new task need to forward in order to keep the pipeline working, and the previous upstream dependency would need to be edited to include the new task.

These problems make it difficult to create a modular pipeline. To solve it, the following was done:

A lightweight class (MetaOutputHandler) was implemented to set the output of a task based on the input. This means that inputs are forwarded, and allows for implementing higher level of abstraction tasks that allow to require al neccessary tasks for a step while making the outputs accessible.

Each step of the analysis is implemented in a **separate module**, with a **high level abstraction Task** subclass **as the entrypoint**.

Using this type of tasks allows too for putting parameters together, so that only one task exposes parameters; which makes it easier to use in a decoupled way.

An example of this type of Task is the ‘ <>‘_.

```
class FastqAlign(utils.MetaOutputHandler, luigi.WrapperTask):
    """Higher level task for the alignment of fastq files.

    It is given preference to local files over processing the alignment
    in order to reduce computational overhead.

    Alignment is done with the Bwa mem utility.

    Parameters:
        fastq1_local_file (str): String indicating the location of a local
            Sam file for the alignment.
        cpus (int): Integer indicating the number of cpus that can be used for
            the alignment.

    Output:
        A dict mapping keys to `luigi.LocalTarget` instances for each of the
        processed files.
        The following keys are available:

        'sam' : Local file with the alignment.

    """

    sam_local_file = luigi.Parameter(default='', description='Optional file path for␣
↪the aligned sam file. If set, the alignment will be skipped.')
    cpus = luigi.Parameter(default='', description="Number of cpus to be used by each␣
↪task thread.")

    def requires(self):
        if self.sam_local_file != '':
            return {'sam': utils.LocalFile(file_path=self.sam_local_file)}
        else:
            return {'sam' : BwaAlignFastq()}
```

This task doesn’t do any actual computation, but requires the Task neccessary for obtaining the fastq alignment. Even though that right now Bwa is being used, changing the task for another one would be easy; as other tasks require wespipeline.align.FastqAlign, and do not care about the tasks required by it. An example below:

```
import luigi
from wespipeline.align import FastqAlign

class MyTask(luigi.Task):

        def requires(self):
                return FastqAlign(cpus=2)

        def output(self):
                returns luigi.LocalTarget("/.../output.txt")

        def run(self):
                print(self.input()['sam'])
```

As shown, `MyTask` requires the higher level task, uses the outputs of the tasks in that step, and accesses the output through an interface which is agnostic from the actual implementation of the task.

Morover, parameters set in this higher Task can be accesses from any other by using an instance (`FastqAlign().sam_local_file`); this means that there is no longer a need for propagating unused parameters.

5.8 How to edit or extend the pipeline

Three main use cases can be distinguished:

- Replacing an existing task for another that serves the same function.
- Adding new tasks to an existing step.
- Adding upstream dependencies within a step, that should be exectued after

the rest of the tasks of the step.

5.8.1 Replacing an existing task for another with the same funtionallity

All tasks implemented in a module serve for one of the two possible objectives: fulfilling other tasks dependencies, or providing with part (or the whole) output of the step.

The first step is identifying which of this two is the case. In both cases it must mantain the parameters and out put format in order for it work as expected; however, if relacing an intermediate task, the upstream tasks should be changed to this new one instead.

In the case of a task that provides part of the output for the higher level task of the module, it only requires to replace the previous task with the new one in the high level task.

```
class HighLevelTask(utils.MetaOutputHandler, luigi.WrapperTask):

        param1 = luigi.Parameter(default="", description="")
        local_file = luigi.Parameter(default='', description="")

        def requires(self):
                dependencies = dict()

                dependencies.update({'a': SomeTask()})
                dependencies.update({'b': OtherTask()})

                return dependencies
```

Beware that high level tasks expose the outputs of all involved tasks of the step, making each output accesible in a dictionary which keys should remain intact in order to preserve the well functioning tasks dependent on this.

Warning: In the example above, the output of `SomeTask` and `OtherTask` maintains the original structure. Be carefull not to change the expected result as this would enter be conflict with what other task would expect to receive from the higher level task.

5.8.2 Adding new tasks to an existing step

When new, independent, tasks are desired to be run for a step, it is only needed to add them to the dictionary containing the dependencies.

It is important to add them in the `requires` method in order for the output of this new task to be checked. This allows Luigi to identify when the step has been completely runned, or if needs to run part -or the whole- of the tasks.

Note: If parameters for the new task are set in the higher level task too, it will allow the users to specify them through the same manner as for other tasks without knowing the specific name of this new task: **–HigherLevelTask-the-parameter value** will be accessible through the command line. Prefer this way rather than making the user know what other tasks in the step need its parameters to be set.

Note: Note too that if parameters for the new task are set in the higher level task, removing the task will not brake dependencies on the step that set this parameter; whilst it will cause an error when a user tries setting a parameter in a task that is not longer available.

5.8.3 Adding upstream dependencies within a step, that should be exectued after the rest of the tasks

A slightly different to the previous use case may occur: the necessity of adding a task that needs to be executed after some step, but which is related to the same one.

When adding functionality that is related to a step, it is best to add it in the same module.

For this, it is possible to replace the base class `luigi.WrapperTask` with `luigi.Task`:

- `wespipeline.MetaOutputHandler` allows to define the output of a class based on the input; which is similar to the

behaviour of `luigi.WrapperTask`, but with the addition of propagating the input. * If any of the dependencies is not fulfilled, the task will run. Thus, requiring the dependencies and then executing the *run* method. * From within the run method, any task can yield dynamic dependencies; we can use this to launch the execution of our step related upstream dependencies.

```
class HighLevelTask(utils.MetaOutputHandler, luigi.Task):

        param1 = luigi.Parameter(default="", description="")
        local_file = luigi.Parameter(default='', description="")

        def requires(self):
                dependencies = dict()

                dependencies.update({'a': SomeTask()})
                dependencies.update({'b': OtherTask()})

                return dependencies

        def run(self):
                yield UpstreamTask(aoutput=self.input()['a'])
```

Thus, this approach allows to extend the step with extra tasks, that can use the outputs without the need of newer dependencies or affecting the interface that the step provides to other task requiring it.

Warning: The default behaviour of a task is running first the output method to check if the task has already been executed correctly; which -in the case of classes inheriting from `wespieline.MetaOutputHandler`- is equivalent to checking the fulfillment of the dependencies. Therefore, if all the inputs are already present, the upstream tasks will be nor executed or even checked.

Note: The default behaviour can be changed by overwritting the ** method, and returning `False` when the task should be runned. Then, if False is always returned, upstream dependencies will always be launched.

Even though, it is best adviced to remove outputs when forced returning False in order to ensure no duplicate or strange behaviour occurs because of the output already existing, this not the case for high level tasks.

This task does not produce the output, but rather forwards its inputs; thus removing the output may cause side effects.

```
class HighLevelTask(utils.MetaOutputHandler, luigi.Task):
        force = luigi.BoolParameter()
        param1 = luigi.Parameter(default="", description="")
        local_file = luigi.Parameter(default='', description="")

        def complete(self):
                outputs = luigi.task.flatten(self.output())

                for output in outputs:
                if self.force and output.exists():
                        output.remove()

                return all(map(lambda output: output.exists(), outputs))

        def requires(self):
                dependencies = dict()

                dependencies.update({'a': SomeTask()})
                dependencies.update({'b': OtherTask()})

                return dependencies

        def run(self):
                yield UpstreamTask(output=self.input()['a'])
```

The example above would work well if `HighLevelTask` did not inherit from `wespieline.MetaOutputHandler`. The following is a better suited implementation, where all dependencies (included `UpstreamTask`) will be check for completition and launched for execution if it is not the case, preserving the desired order:

```
class HighLevelTask(utils.MetaOutputHandler, luigi.Task):
        param1 = luigi.Parameter(default="", description="")
        local_file = luigi.Parameter(default='', description="")

        def complete(self):
                return False

        def requires(self):
                dependencies = dict()

                dependencies.update({'a': SomeTask()})
                dependencies.update({'b': OtherTask()})

                return dependencies

        def run(self):
                yield UpstreamTask(output=self.input()['a'])
```

Upon execution,dependencies will be checked first, ceating if neccessary the outputs for the task. The the run method will be checked and executed if neccessary.

5.9 wespipeline

5.9.1 wespipeline package

Submodules

wespipeline.align module

class `wespipeline.align.`**`BwaAlignFastq`**`(`**args*, ***kwargs*`)`

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task used for aligning fastq files against the reference genome.

It requires the output of both the wespipeline.reference.ReferenceGenome and wespipeline.fastq.GetFastq higher level tasks in order to proceed with the alignment.

If `wespipeline.utils.GlobalParams.exp_name` is set, it will be used for giving name to the Sam file produced.

Parameters **`none`** –

Output: A *luigi.LocalTarget* instance for the aligned sam file.

`output``()`

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

`program_args``()`

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

`requires``()`

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

class `wespipeline.align.`**`FastqAlign`**`(`**args*, ***kwargs*`)`

Bases: *`wespipeline.utils.MetaOutputHandler`*, `luigi.task.Task`

Higher level task for the alignment of fastq files.

It is given preference to local files over processing the alignment in order to reduce computational overhead.

Alignment is done with the Bwa mem utility.

Parameters

- **fastq1_local_file** (*str*) – String indicating the location of a local Sam file for the alignment.
- **cpus** (*int*) – Integer indicating the number of cpus that can be used for the alignment.

Output: A dict mapping keys to *luigi.LocalTarget* instances for each of the processed files. The following keys are available:

'sam' : Local file with the alignment.

cpus = <luigi.parameter.Parameter object>

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

run()

The task run method, to be overridden in a subclass.

See Task.run

sam_local_file = <luigi.parameter.Parameter object>

wespipeline.fastq module

class `wespipeline.fastq.`**FastqcQualityCheck**(**args*, ***kwargs*)

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task used for creating a quality report on fastq files.

The report is created using the Fastqc utility, reulsting on an html report, an a zip folder containing more detailed information about the quality of the reads.

Parameters **fastq_file** (*str*) – Path for the fastq file to be analyzed.

Output: html (luigi.LocalTarget) : File containing the report for fastqc quality.

fastq_file = <luigi.parameter.Parameter object>

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

program_args()

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

class `wespipeline.fastq.`**`GetFastq`**(**args*, ***kwargs*)

Bases: *`wespipeline.utils.MetaOutputHandler`*, `luigi.task.Task`

Higher level task for the retrieval of the experiment fastq files.

Three diferent sources for the fastq files are accepted: an existing local file, an NCBI accession number for the reads, and an external url indicating the location for the resources. The order in which the sources will be searched is the same as above: it is given preference to local files over external resources in order to reduce computational overhead, and NCBI accession number over external resources for reproducibility reasons.

Parameters

- **`fastq1_local_file`** (*`str`*) – String indicating the location of a local compressed fastq file.
- **`fastq2_local_file`** (*`str`*) – String indicating the location of a local compressed fastq file.
- **`fastq1_url`** (*`str`*) – Url indicating the location of the resource for the compressed fastq file.
- **`fastq2_url`** (*`str`*) – Url indicating the location of the resource for the compressed fastq file.
- **`paired_end`** (*`bool`*) – Non case sensitive boolean indicating wether the reads are paired_end.
- **`compressed`** (*`bool`*) – Non case sensitive boolean indicating wether the reads are compressed.
- **`create_report`** (*`bool`*) – A non case-sensitive boolean indicating wether to create a quality check report.

Output: A dict mapping keys to *luigi.LocalTarget* instances for each of the processed files. The following keys are available:

‘fastq1’ : Local file with the fastq file with the experiment’s reads. ‘fastq2’ : In case of paired end experiments, a local file with the fastq

file with the experiment’s reads.

`accession_number = <luigi.parameter.Parameter object>`

`compressed = <luigi.parameter.BoolParameter object>`

`create_report = <luigi.parameter.BoolParameter object>`

`fastq1_local_file = <luigi.parameter.Parameter object>`

`fastq1_url = <luigi.parameter.Parameter object>`

`fastq2_local_file = <luigi.parameter.Parameter object>`

`fastq2_url = <luigi.parameter.Parameter object>`

`paired_end = <luigi.parameter.BoolParameter object>`

`requires`()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don’t need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

run()
The task run method, to be overridden in a subclass.

See Task.run

class wespipeline.fastq.**SraToolkitFastq**(**args*, ***kwargs*)
Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task used for downloading fastq files from the NVBI archive.

In case of the reads to be paired end, the output will consist of two separate fastq files.

The output file(s) will have for name the accession number and, in the case of paired end reads, a suffix identifying each of the two fastq.

Parameters

- **accession_number** (*str*) – NCBI accession number for the experiment.
- **paired_end** (*bool*) – Non case sensitive boolean indicating wether the reads are paired_end.

Output: A dict mapping keys to *luigi.LocalTarget* instances for each of the processed files. The following keys are available:

'fastq1' : Local file with the fastq file with the experiment's reads. 'fastq2' : In case of paired end experiments, a local file with the fastq

file with the experiment's reads.

accession_number = <luigi.parameter.Parameter object>

output()
The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

paired_end = <luigi.parameter.BoolParameter object>

program_args()
Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

class wespipeline.fastq.**UncompressFastqgz**(**args*, ***kwargs*)
Bases: `luigi.task.Task`

Task for uncompressing fastq files.

The task uses utils.UncompressFile for uncompressing into fastq. If both fastq_local_file and fastq_url are set, the local file will have preference; thus reducing the overhead in the process.

Parameters

- **fastq_local_file** (*str*) – String indicating the location of a local compressed fastq file.

- **fastq_url** (*str*) – Url indicating the location of the resource for the compressed fastq file.
- **output_file** (*str*) – String indicating the desired location and name the output uncompressed fastq file.

Output: A *luigi.LocalTarget* instance for the uncompressed fastq file.

fastq_local_file = <luigi.parameter.Parameter object>

fastq_url = <luigi.parameter.Parameter object>

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

output_file = <luigi.parameter.Parameter object>

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

run()

The task run method, to be overridden in a subclass.

See Task.run

wespipeline.processalign module

class wespipeline.processalign.**AlignProcessing**(**args*, ***kwargs*)

Bases: *wespipeline.utils.MetaOutputHandler*, `luigi.task.Task`

Higher level task for the alignment of fastq files.

It is given preference to local files over processing the alignment in order to reduce computational overhead.

If the bam and bai local files are set, they will be used instead of the

Alignment is done with the Bwa mem utility.

Parameters

- **bam_local_file** (*str*) – String indicating the location of a local bam file with the sorted alignment. If set, this file will not be created.
- **bai_local_file** (*str*) – String indicating the location of a local bai file with the index for the alignment. If set, this file will not be created.
- **no_dup_bam_local_file** (*str*) – String indicating the location of a local sam file without the duplicates. If set, this file will not be created.

- **`no_dup_bai_local_file`** (*`str`*) – String indicating the location of a local file with the index for the bam file without duplicates. If set, this file will not be created.
- **`cpus`** (*`int`*) – Integer indicating the number of cpus that can be used for the alignment.

Output: A dict mapping keys to *luigi.LocalTarget* instances for each of the processed files. The following keys are available:

'bam' : Local file with the sorted alignment. 'bai' : Local file with the alignment index. 'bamNoDup' : Local sorted file with duplicates removed. 'indexNoDup' : Local file with the index for sorted alignment without duplicates.

`bai_local_file = <luigi.parameter.Parameter object>`

`bam_local_file = <luigi.parameter.Parameter object>`

`cpus = <luigi.parameter.IntParameter object>`

`no_dup_bai_local_file = <luigi.parameter.Parameter object>`

`no_dup_bam_local_file = <luigi.parameter.Parameter object>`

`requires``()`

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

`run``()`

The task run method, to be overridden in a subclass.

See Task.run

`class` `wespipeline.processalign.`**`IndexBam`**`(`**args*, ***kwargs*`)`

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task used for indexing the Bam file.

The `wespipeline.utils.GlobalParams.exp_name` will be used for giving name to the Bai file produced.

Parameters **`none`** –

Output: A *luigi.LocalTarget* instance for the index Bai file.

`output``()`

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

`program_args``()`

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

class `wespipeline.processalign.`**IndexNoDup**(**args*, ***kwargs*)

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task used for indexing the Bam file without duplicates.

The `wespipeline.utils.GlobalParams.exp_name` will be used for giving name to the Bai file produced.

Parameters **none** –

Output: A *luigi.LocalTarget* instance for the index Bai file.

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

program_args()

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

class `wespipeline.processalign.`**PicardMarkDuplicates**(**args*, ***kwargs*)

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task used for removing duplicates from the Bam file.

The `wespipeline.utils.GlobalParams.exp_name` will be used for giving name to the Bam file produced.

Parameters **none** –

Output: A *luigi.LocalTarget* instance for the Bam file without the duplicates.

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

`program_args()`

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

`requires()`

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

class `wespipeline.processalign.`**`SortSam`**`(`**args*, ***kwargs*`)`

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task used for sorting the alignment sam file.

It requires the output of the `wespipeline.reference.FastqAlign` step.

The `wespipeline.utils.GlobalParams.exp_name` will be used for giving name to the Bam file produced.

Parameters **`none`** –

Output: A *luigi.LocalTarget* instance for the sorted Sam Bam file.

`output()`

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

`program_args()`

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

`requires()`

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

wespipeline.reference module

class wespipeline.reference.**BwaIndex**(**args*, ***kwargs*)

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task user for indexing the reference genome .fa file with the bwa index utility.

Aligning the reference genome helps reducing access time drastically.

Parameters **None** –

Output: A set of five files are result of indexing the reference genome. The extensions for each of the files are '.amb', '.ann', '.bwt', '.pac', '.sa'.

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

program_args()

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

class wespipeline.reference.**FaidxIndex**(**args*, ***kwargs*)

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task user for indexing the reference genome .fa file with the samtools faidx utility.

Aligning the reference genome helps reducing access time drastically.

Parameters **None** –

Output: A *luigi.LocalTarget* for the .fai index file for the reference genome .

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

program_args()
Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

requires()
The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

class wespipeline.reference.**GetProgram**(**args*, ***kwargs*)
Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task user for downloading and giving execution permissions to the 2bit program.

The task gives execute permissions to the conversion utility for 2bit files to be converted to fa files which can then be used for aligning the sequences.

The source for the program is ftp://hgdownload.cse.ucsc.edu/admin/exe/linux.x86_64/twoBitToFa.

Parameters **none** –

Output: A *luigi.LocalTarget* for the executable.

output()
The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

program_args()
Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

requires()
The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

class wespipeline.reference.**GetReferenceFa**(**args*, ***kwargs*)
Bases: *`wespipeline.utils.MetaOutputHandler`*, `luigi.task.WrapperTask`

Task user for obtaining the reference genome .fa file.

This task will retrieve an external genome or use a provided local one, and convert it from 2bit format to .fa if neccessary.

Parameters

- **ref_url** (*str*) – Url for the resource with the reference genome.

- **`reference_local_file`** (*`str`*) – Path for the reference genome 2bit file. If given the `ref_url` parameter will be ignored.
- **`from2bit`** (*`bool`*) – Non case sensitive boolean indicating wether the reference genome if in 2bit format. Defaults to false.

Output: A *luigi.LocalTarget* for the reference genome fa file.

`from2bit = <luigi.parameter.BoolParameter object>`

`ref_url = <luigi.parameter.Parameter object>`

`reference_local_file = <luigi.parameter.Parameter object>`

`requires``()`

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

`class` `wespipeline.reference.`**`PicardDict`**`(`**args*, ***kwargs*`)`

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task user for creating a dict file with the reference genome .fa file with the picard utility.

Parameters **`None`** –

Output: A *luigi.LocalTarget* for the .fai index file for the reference genome .

`output``()`

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

`program_args``()`

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

`requires``()`

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

`class` `wespipeline.reference.`**`ReferenceGenome`**`(`**args*, ***kwargs*`)`

Bases: *`wespipeline.utils.MetaOutputHandler`*, `luigi.task.Task`

Higher level task for retrieving the reference genome.

It is given preference to local files over downloading the reference. However the indexing of the reference genome is always done using `GloablParams.exp_name` and `GlobalParams.base_dir` for determining filenames and location for newer files respectively.

The indexing is done using both Samtools and Bwa toolkits.

Parameters

- **reference_local_file** (*str*) – Optional string indicating the location for the reference genome. If set, it will not be downloaded.
- **ref_url** (*str*) – Url for the download of the reference genome.
- **from2bit** (*bool*) – A boolean [True, False] indicating whether the reference genome must be converted from 2bit.

Output: A dict mapping keys to *luigi.LocalTarget* instances for each of the processed files. The following keys are available:

'faidx' : Local file with the index, result of indexing with Samtools. 'bwa' : Set of five files, result of indexing the reference genome with Bwa. 'fa' : Local file with the reference genome.

from2bit = <luigi.parameter.BoolParameter object>

ref_url = <luigi.parameter.Parameter object>

reference_local_file = <luigi.parameter.Parameter object>

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

run()

The task run method, to be overridden in a subclass.

See Task.run

class wespipeline.reference.**TwoBitToFa**(**args*, ***kwargs*)

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task user for Converting 2bit files to the fa format.

The task will use a local executable or require the task for obtaining it, and use with the reference genome.

Parameters

- **ref_url** (*str*) – Url for the resource with the reference genome.
- **reference_local_file** (*str*) – Path for the reference genome 2bit file. If given the `ref_url` parameter will be ignored.

Output: A *luigi.LocalTarget* for the reference genome fa file.

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

`program_args``()`

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

`ref_url = <luigi.parameter.Parameter object>`

`reference_local_file = <luigi.parameter.Parameter object>`

`requires``()`

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

wespipeline.utils module

`class` `wespipeline.utils.`**`GlobalParams`**`(`**args*, ***kwargs*`)`

Bases: `luigi.task.Config`

Task used for specifying globally accessible parameters.

Parameters defined in this class are task independent and should mantain low.

Parameters

- **`exp_name`** (*`str`*) – Name for the experiment. Useful for defining file names.
- **`log_dir`** (*`str`*) – Absolute path for the logs of the application.
- **`base_dir`** (*`str`*) – Absolute path to the directory where files are expected to appear if not specifyed differently.

`base_dir = <luigi.parameter.Parameter object>`

`exp_name = <luigi.parameter.Parameter object>`

`log_dir = <luigi.parameter.Parameter object>`

`class` `wespipeline.utils.`**`GunzipFile`**`(`**args*, ***kwargs*`)`

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task for unzipping compressed files.

Gunzip will allways do the process inplace, deleting the extension.

Parameters **`input_file`** (*`str`*) – Absolute path to the compressed file.

`input_file = <luigi.parameter.Parameter object>`

`output``()`

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

`program_args``()`

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

`requires``()`

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

`class` `wespipeline.utils.`**`LocalFile`**`(`**args*, ***kwargs*`)`

Bases: `luigi.task.Task`

Helper task for making.

No extra processing is done in the task. It allows to make tasks dependent on files using the same strategy as with other tasks.

Parameters **`file`** (*`str`*) – Absolute path to the file to be tested.

`file = <luigi.parameter.Parameter object>`

`output``()`

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

`run``()`

The task run method, to be overridden in a subclass.

See Task.run

`class` `wespipeline.utils.`**`MetaOutputHandler`**

Bases: `object`

Helper class for propagating inputs in WrapperTasks

`output``()`

`class` `wespipeline.utils.`**`UncompressFile`**`(`**args*, ***kwargs*`)`

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task for unzipping compressed files to a desired location.

Gunzip will allways do the process inplace, deleting the extension. This task allows to select the destination.

This operation

Parameters

- **input_file** (*str*) – Absolute path to the compressed file.
- **output_file** (*str*) – Absolute path to the desired final location.
- **copy** (*bool*) – Non case sensitive boolean indicating wether to copy or to move the file. Defaults to false.

copy = <luigi.parameter.BoolParameter object>

input_file = <luigi.parameter.Parameter object>

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

output_file = <luigi.parameter.Parameter object>

program_args()

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

class `wespipeline.utils.`**Wget**(**args*, ***kwargs*)

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task for downloading files using the tool wget.

Parameters

- **url** (*str*) – Url indicating the location of the resource to be retreived.
- **output_file** (*str*) – Absolute path for the destiny location of the retrived resource.

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

output_file = <luigi.parameter.Parameter object>

program_args()

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

url = <luigi.parameter.Parameter object>

wespipeline.vcf module

class `wespipeline.vcf.`**DeepvariantCallVariants**(**args*, ***kwargs*)

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task used for identifying varinats in the bam file provided using DeepVariant.

Parameters **model_type** (*str*) – A string defining the model to use for the variant calling. Valid options are [WGS,WES,PACBIO].

Dependencies: ReferenceGenome AlignProcessing

Output: A *luigi.LocalTarget* instance for the index vcf file.

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

program_args()

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

class `wespipeline.vcf.`**DeepvariantDockerTask**(**args*, ***kwargs*)

Bases: `luigi.contrib.docker_runner.DockerTask`

Task used for identifying varinats in the bam file provided using DeepVariant.

Parameters **model_type** (*str*) – A string defining the model to use for the variant calling. Valid options are [WGS,WES,PACBIO].

Dependencies: ReferenceGenome AlignProcessing

Output: A *luigi.LocalTarget* instance for the index vcf file.

`BIN_VERSION = '0.8.0'`

`property binds`

Override this to mount local volumes, in addition to the /tmp/luigi which gets defined by default. This should return a list of strings. e.g. ['/hostpath1:/containerpath1', '/hostpath2:/containerpath2']

`property command`

`create_gvcf = <luigi.parameter.BoolParameter object>`

`property image`

`model_type = <luigi.parameter.Parameter object>`

`property mount_tmp`

`output``()`

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

`requires``()`

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

`class` `wespipeline.vcf.`**`DockerGatkCallVariants`**`(*args, **kwargs)`

Bases: `luigi.contrib.docker_runner.DockerTask`

Task used for identifying varinats in the bam file provided using DeepVariant.

Parameters **`model_type`** (*`str`*) – A string defining the model to use for the variant calling. Valid options are [WGS,WES,PACBIO].

Dependencies: ReferenceGenome AlignProcessing

Output: A *luigi.LocalTarget* instance for the index vcf file.

`BIN_VERSION = '0.8.0'`

`property binds`

Override this to mount local volumes, in addition to the /tmp/luigi which gets defined by default. This should return a list of strings. e.g. ['/hostpath1:/containerpath1', '/hostpath2:/containerpath2']

`property command`

`property image`

`model_type = <luigi.parameter.Parameter object>`

`property mount_tmp`

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

class wespipeline.vcf.**FreebayesCallVariants**(**args*, ***kwargs*)

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task used for identifying varinats in the bam file provided using Freebayes.

The `wespipeline.utils.GlobalParams.exp_name` will be used for giving name to the vcf produced.

Parameters **none** –

Dependencies: ReferenceGenome AlignProcessing

Output: A *luigi.LocalTarget* instance for the index vcf file.

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

program_args()

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

class wespipeline.vcf.**GatkCallVariants**(**args*, ***kwargs*)

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task used for identifying varinats in the bam file provided using GatkCallVariants.

The `wespipeline.utils.GlobalParams.exp_name` will be used for giving name to the vcf produced.

Parameters **`none`** –

Dependencies: ReferenceGenome AlignProcessing

Output: A *luigi.LocalTarget* instance for the index vcf file.

`output``()`

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

`program_args``()`

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

`requires``()`

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

class `wespipeline.vcf.`**`PlatypusCallVariants`**`(`**args*, ***kwargs*`)`

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task used for identifying varinats in the bam file provided using Platypus.

The `wespipeline.utils.GlobalParams.exp_name` will be used for giving name to the vcf produced.

Parameters **`none`** –

Dependencies: ReferenceGenome AlignProcessing

Output: A *luigi.LocalTarget* instance for the index vcf file.

`output``()`

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

program_args()

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

class `wespipeline.vcf.`**VariantCalling**(**args*, ***kwargs*)

Bases: *`wespipeline.utils.MetaOutputHandler`*, `luigi.task.Task`

Higher level task for the alignment of fastq files.

It is given preference to local files over processing the alignment in order to reduce computational overhead.

If the bam and bai local files are set, they will be used instead of the

Alignment is done with the Bwa mem utility.

Parameters

- **use_platypus** (*bool*) – A non-case sensitive boolean indicating wether to use Platypus for variant callign.
- **use_freebayes** (*bool*) – A non-case sensitive boolean indicating wether to use Free-bayesfor variant callign.
- **use_samtools** (*bool*) – A non-case sensitive boolean indicating wether to use Samtools for variant callign.
- **use_gatk** (*bool*) – A non-case sensitive boolean indicating wether to use Gatk for variant callign.
- **use_deepvariant** (*bool*) – A non-case sensitive boolean indicating wether to use DeepVariant for variant callign.
- **vcf_local_files** (*string*) – A comma delimited list of vfc files to be used instead of using the variant calling tools.
- **cpus** (*int*) – Number of cpus that are available for each of the methods selected.

Output: A dict mapping keys to *luigi.LocalTarget* instances for each of the processed files. The following keys are available:

'platypus' : Local file with the variant calls obtained using Platypus. 'freebayes' : Local file with the variant calls obtained using Freevayes. 'Varscan' : Local sorted file with variant calls obtained using Varscan. 'gatk' : Local file with the variant calls obtained using GATK. 'deepvariant' : Local file with the variant calls obtained using DeepVariant.

cpus = <luigi.parameter.IntParameter object>

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

run()

The task run method, to be overridden in a subclass.

See Task.run

use_deepvariant = <luigi.parameter.BoolParameter object>

use_freebayes = <luigi.parameter.BoolParameter object>

use_gatk = <luigi.parameter.BoolParameter object>

use_platypus = <luigi.parameter.BoolParameter object>

use_varscan = <luigi.parameter.BoolParameter object>

vcf_local_files = <luigi.parameter.Parameter object>

class `wespipeline.vcf.`**VarscanCallVariants**(**args*, ***kwargs*)

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task used for identifying varinats in the bam file provided using Varscan..

The `wespipeline.utils.GlobalParams.exp_name` will be used for giving name to the vcf produced.

Parameters **none** –

Dependencies: ReferenceGenome AlignProcessing

Output: A *luigi.LocalTarget* instance for the index vcf file.

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

program_args()

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

wespipeline.vcfanalysis module

class `wespipeline.vcfanalysis.`**DockerVTnormalizeVCF**(**args*, ***kwargs*)

Bases: `luigi.contrib.docker_runner.DockerTask`

VERSION = '0.57721--hdf88d34_2'

biallelic_block_substitutions = <luigi.parameter.BoolParameter object>

biallelic_clumped_variant = <luigi.parameter.BoolParameter object>

property binds

Override this to mount local volumes, in addition to the /tmp/luigi which gets defined by default. This should return a list of strings. e.g. ['/hostpath1:/containerpath1', '/hostpath2:/containerpath2']

property command

decomposes_multiallelic_variants = <luigi.parameter.BoolParameter object>

property image

property mount_tmp

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

vcf = <luigi.parameter.Parameter object>

class wespipeline.vcfanalysis.**NormalizeVcfFiles**(**args*, ***kwargs*)

Bases: *wespipeline.utils.MetaOutputHandler*, luigi.task.Task

docstring for NormalizeVcfFiles

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

run()
The task run method, to be overridden in a subclass.

See Task.run

class `wespipeline.vcfanalysis.`**VTnormalizeVCF**(**args*, ***kwargs*)
Bases: `luigi.contrib.external_program.ExternalProgramTask`

out = <luigi.parameter.Parameter object>

output()
The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

program_args()
Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

requires()
The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

vcf = <luigi.parameter.Parameter object>

class `wespipeline.vcfanalysis.`**VariantCallingAnalysis**(**args*, ***kwargs*)
Bases: `luigi.task.Task`

Higher level task for comparing variant calls.

Comparing variant calls is a delicate task that increments in complexity when dealing in diploid sequences (such us the human genome), where different variants can appear in the same position in each of the pair chromomes.

The normalization is done with vt, and the comparison with VcfTools

Parameters **None** –

Output: None. The resulting files are not provided as task output. Each of the n vcf files is analyzed and comparied by pairs. It is a total of 2n-1 files.

normalize = <luigi.parameter.BoolParameter object>

output()
The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

run()

The task run method, to be overridden in a subclass.

See Task.run

class `wespipeline.vcfanalysis.`**VcftoolsCompare**(**args*, ***kwargs*)

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task used for comparing a pair of vcf files using VcfTools.

Parameters

- **vcf1** (*str*) – Absolute path to the first file to be compared.
- **vcf2** (*str*) – Absolute path to the second file to be compared.

Dependencies: None

Output: A *luigi.LocalTarget* instance for the result of comparing the files.

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

program_args()

Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

vcf1 = <luigi.parameter.Parameter object>

vcf2 = <luigi.parameter.Parameter object>

class `wespipeline.vcfanalysis.`**VcftoolsDepthAnalysis**(**args*, ***kwargs*)

Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task used for extracting basic statistics for the variant calls using VcfTools.

Parameters **vcf** (*str*) – Absolute path to the file with the variant annotations.

Dependencies: None

Output: A *luigi.LocalTarget* instance for the file with the vcf statistics.

output()
The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

program_args()
Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

requires()
The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

vcf = <luigi.parameter.Parameter object>

class wespipeline.vcfanalysis.**VcftoolsFreqAnalysis**(**args*, ***kwargs*)
Bases: `luigi.contrib.external_program.ExternalProgramTask`

Task used for extracting basic statistics for the variant calls using VcfTools.

Parameters **vcf** (*str*) – Absolute path to the file with the variant annotations.

Dependencies: None

Output: A *luigi.LocalTarget* instance for the file with the vcf statistics.

output()
The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

program_args()
Override this method to map your task parameters to the program arguments

Returns list to pass as `args` to `subprocess.Popen`

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don’t need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

vcf = <luigi.parameter.Parameter object>

Module contents

wespipeline.**name = 'wespipeline_pkg'**

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX