ngs$_m apper Documentation$

Release 1.0.0

Tyghe Vallard, Melanie Melendrez

March 30, 2016

Contents

Version: 1.0.0

The ngs_mapper is a configurable pipeline next generation sequence pipeline that aims to be easy for the bioinformatician to install as well as use. It focuses on documentation as well as easy configuration and running. The pipeline is also meant to get you from data to completed genome as easy as possible.

The documentation is a work-in-progress as the aim is to keep it as much up-to-date as possible with the pipeline as it changes. If anything seems out of place or wrong please open a bug report on github

CHAPTER 1

Contents:

1.1 Install

1.1.1 Requirements

Hardware

- **CPU**
  - **Quad Core 2.5GHz or better**
    - More cores = faster run time when running multiple samples
    - Faster GHz = faster each sample runs
- **RAM**
  - At least 1GB per CPU core for small genomes(Dengue, Flu,...)

Python Packages

All python packages can be defined in a pip requirements.txt file The pipeline comes with all of the necessary python packages already defined inside of requirements.txt.

System Packages

The pipeline requires some system level packages(software installed via your Linux distribution's package manager) The installer looks for the system_packages.lst and installs the correct packages using that file. This file is a simple json formatted file that defines packages for each package manager

Roche Utilities

If you intend on using the `roche_sync` you will need to ensure that the `sfffile` command is in your PATH. That is, if you execute `$> sfffile` it returns the help message for the command.

This command should automatically be installed and put in your path if you install the Data Analysis CD #3 that was given to you with your Roche instrument.

MidParse.conf

If you inted on using the `roche_sync` you may need to edit the included ngs_mapper/MidParse.conf file before installing. This file is formatted to be used by the Roche utilities and more information about how it is used can be found in the Roche documentation.

1.1.2 Installation

1. Clone

   Assumes you already have git installed. If not you will need to get it installed by your system administrator.

```
git clone https://githubusername@github.com/VDBWRAIR/ngs_mapper.git
cd ngs_mapper
```

2. Install System Packages

   This is the only part of the installation process that you should need to become the super user

   - Red Hat/CentOS(Requires the root password)

```
su -c 'python setup.py install_system_packages && chmod 666 -R setuptools*'
```

   - Ubuntu

```
sudo python setup.py install_system_packages && sudo chmod 666 -R setuptools*
```

3. Configure the defaults

   You need to configure the ngs_mapper/config.yaml file.

   (a) Copy the default config to config.yaml

```
cp ngs_mapper/config.yaml.default ngs_mapper/config.yaml
```

   (b) Then edit the ngs_mapper/config.yaml file which is in yaml format

   The most important thing is that you edit the NGSDATA value so that it contains the path to your NGSDATA directory.

   The path you use for NGSDATA must already exist

```
mkdir -p /path/to/NGSDATA
```

4. Python

   The ngs_mapper requires python 2.7.3+ but < 3.0

   - Ensure python is installed

```
python setup.py install_python
```

   - Quick verify that Python is installed

   The following should return python 2.7.x(where x is somewhere from 3 to 9)

```
$HOME/bin/python --version
```

5. Setup virtualenv

   (a) Where do you want the pipeline to install? Don't forget this path, you will need it every time you want to activate the pipeline

```
venvpath=$HOME/.ngs_mapper
```

   (b) Install the virtualenv to the path you specified

```
wget --no-check-certificate https://pypi.python.org/packages/source/v/virtualenv/virt
$HOME/bin/python virtualenv-1.11.6/virtualenv.py --prompt="(ngs_mapper) " $venvpath
```

   (a) Activate the virtualenv. You need to do this any time you want to start using the pipeline

```
. $HOME/.ngs_mapper/bin/activate
```

3. Install the pipeline into virtualenv

```
python setup.py install
```

It should be safe to run this more than once in case some dependencies do not fully install.

Build and view complete documentation

```
cd doc
make clean && make html
firefox build/html/install.html#build-and-view-complete-documentation
cd ..
```

Verify install

You can pseudo test the installation of the pipeline by running the functional tests

```
nosetests ngs_mapper/tests/test_functional.py
```

1.2 Upgrade

1.2.1 If you activate the pipeline via ngs_mapper/setup

- Completely remove the existing ngs_mapper directory.
- Then follow Install

1.2.2 If you installed using setup.py

1. First fetch any possible updates

```
cd ~/ngs_mapper; git fetch
```

2. Then check if you need to update

```
git status | grep -q 'Your branch is behind' && echo 'You need to update' || echo 'You are u
```

If it returns You are up-to-date you are done

3. Update(pull new code)

```
git pull
```

4. Go into your ngs_mapper directory and rerun the setup script

```
python setup.py install
```

1.3 config.yaml

When you install the pipeline you are instructed to copy ngs_mapper/config.yaml.default to ngs_mapper/config.yaml This file contains all settings that the pipeline will use by default if you do not change them using any of the script options that are available.

When you install the pipeline the config.yaml file gets installed with the pipeline into the installation directory(probably ~/.ngs_mapper) In order to change the defaults after that you have two options:

- Edit config.yaml inside of the source directory you cloned with git, then go into your ngs_mapper directory and rerun the setup.py command

```
python setup.py install
```

- Use the `make_example_config` to extract the config.yaml into the current directory and use it

1.3.1 Example changing single script defaults

If you want to change the quality threshold to use to trim reads when you run `trim_reads.py` you would probably do something as follows:

1. First what options are available for the command?

```
#> trim_reads.py --help
usage: trim_reads.py [-h] [--config CONFIG] [-q Q] [--head-crop HEADCROP]
                     [-o OUTPUTDIR]
                     readsdir

Trims reads

positional arguments:
  readsdir              Read or directory of read files

optional arguments:
  -h, --help            show this help message and exit
  --config CONFIG, -c CONFIG
                        Path to config.yaml file
  -q Q                  Quality threshold to trim[Default: 20]
  --head-crop HEADCROP  How many bases to crop off the beginning of the reads
                        after quality trimming[Default: 0]
  -o OUTPUTDIR          Where to output the resulting files[Default:
                        trimmed_reads]
```

You can see that there is a -q option you can specify the quality threshold with

2. Now run the command with your specific value

```
#> trim_reads.py -q 5 /path/to/my/input.fastq
```

This process works pretty slick until you notice that there is no way to easily tell `runsample.py` to specify that same value. With the version 1.0 release of the pipeline there is now a config file that you can edit and change the Default value any script will use.

1.3.2 Example running `runsample.py` using config.yaml

1. First we need to get a config file to work with

```
#> make_example_config
/current/working/directory/config.yaml
```

2. We just need to edit that config.yaml file which should be in the current directory and change the trim_reads's q option default value to 5 then save the file
3. Now just run `runsample.py` as follows

```
#> runsample.py /path/to/NGSData /path/to/reference.fasta mysample -od mysample -c config.ya
2014-11-28 14:39:14,906 -- INFO -- runsample       --- Starting mysample ---
2014-11-28 14:39:14,906 -- INFO -- runsample       --- Using custom config from config.yaml
2014-11-28 14:39:35,926 -- INFO -- runsample       --- Finished mysample ---
```

1.3.3 Example running runsamplesheet.sh using a custom config.yaml

You will probably want to be able to run an entire samplesheet with a custom config file as well. If you check out the `scripts/runsamplesheet` page you will notice that you can specify options to pass on to `runsample.py` by using the RUNSAMPLEOPTIONS variable

1. Generate your config.yaml template

```
make_example_config
```

2. Then run `scripts/runsamplesheet` with your custom config.yaml

```
#> RUNSAMPLESHEET="-c config.yaml" runsamplesheet.sh /path/to/NGSData/ReadsBySample samplesh
```

Editing config.yaml

The config.yaml file is just a yaml formatted file that is parsed using the python package pyaml Yaml syntax links for reference:

- Quick start
- More in depth

For the ngs_mapper the most important thing is that the NGSDATA value is filled out and contains a correct path to the root of your Data Structure The rest of the values are pre-filled with defaults that work for most general cases.

1.3.4 Structure of the config.yaml file

The config.yaml basically is divided into sections that represent defaults for each stage/script that the pipeline has. It also contains some global variables such as the NGSDATA variable.

Each script/stage requires at a minimum of the default and help defined.

- default defines the default value that option will use
- **help defines the help message that will be displayed for that option and probably does not need to be modified**
  While yaml does not require you to put text in quotes, it is highly recommended as it will remove some parsing problems if you have special characters in your text such as a : or %

1.4 Data Structure

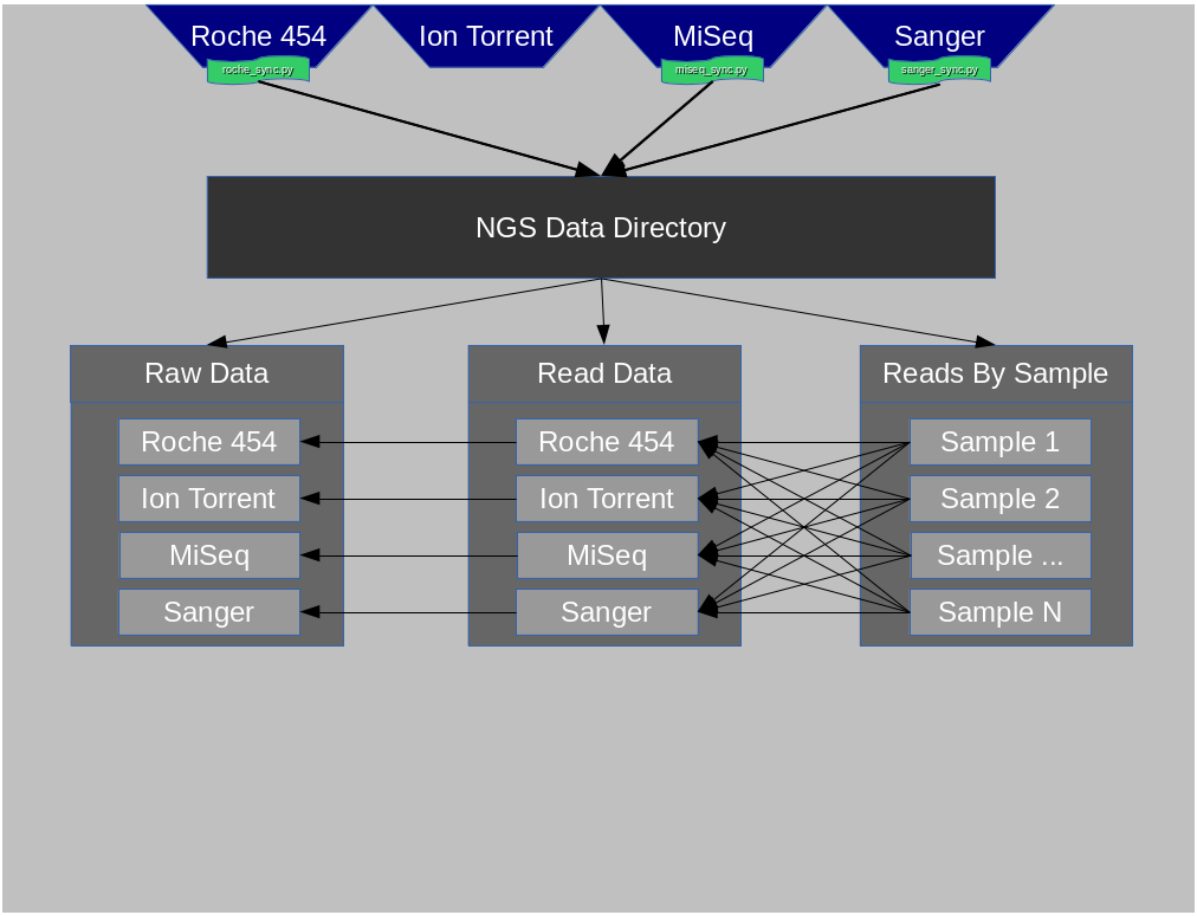
At this time data is organized inside of what is called the NGS Data Structure. This structure is composed of 3 critical directories.

- RawData
- ReadData
- ReadsBySample

1.4.1 Getting Data into the data structure

See `ngsdatasync`

1.4.2 Diagram



1.4.3 RawData

RawData is composed of all files that originate from each of the instruments' Run. Some instruments may create ReadData as well or very close to ReadData, but it is still considered RawData.

Some examples of RawData would be:

- Run_3130xl_ directories containing *.ab1 files(Sanger)
- Directories under the MiSeqOutput directory(MiSeq)
- R_* directories containing signalProcessing or fullProcessing directories(Roche)

1.4.4 ReadData

ReadData is any sequence file format that can be utilized by NGS mapping/assembly applications. At this time these file formats typically end with the following extensions:

.ab1 .sff .fastq

1.4.5 ReadsBySample

This directory contains only directories that are named after each of the samplenames that have been sequenced. The concept of this folder is to make it very easy to look up all data related to a given samplename.

The files that live in these directories must adhere strictly to a specific naming scheme so the pipeline can determine what reads come from which platforms as they need to be handled slightly different.

See the following naming regular expressions defined in `ngs_mapper.data` for more information about how these filenames have to be named:

- `sanger`
- `miseq`
- `roche`
- `iontorrent`

If you have files that do not match these naming standards the pipeline will essentially ignore them and you may get errors when you run `runsample.py`. Inspecting the logs you will see errors about Somehow no reads being compiled which is a good indication that your files are named incorrectly.

To ensure your files are named correctly you can use the various sync scripts listed in `ngsdatasync`

1.5 Analysis

1.5.1 Complete Examples

Here we will show you a complete example of running the pipeline using some test data that is included with the source code.

Note: Any time you see

```
$> command
```

It means you should be able to type that into your terminal

All examples will assume your current working directory is inside of the git cloned ngs_mapper directory, aka the following command ends with ngs_mapper:

```
$> pwd
```

For both examples below, as always when running the pipeline, you need to ensure your installation is activated:

```
$> . ~/.ngs_mapper/bin/activate
```

The location of our data sets are under ngs_mapper/tests/fixtures/functional

```
$> ls ngs_mapper/tests/fixtures/functional
780  780.conf  780.ref.fasta  947  947.conf  947.ref.fasta
```

Here you can see we have 2 data sets to play with.

- 780 is an H3N2 data set
- 947 is a Dengue 4 data set

You will notice that there is a 780/947 directory and a 780/947.ref.fasta file. The 780/947 directory contains all the read files for the 780/947 sample while the 780/947.ref.fasta is the reference to map to. You can ignore the .conf files, they are used by the automated tests.

Using runsample.py to run a single sample

Some times you just need to run a single sample. Here we will use runsample.py to run the 947 example data set and have the analysis be put into a directory called 947 in the current directory

First, let's see what options there are available for the `runsample.py` script to use

```
$> runsample.py
usage: runsample.py [-h] [--config CONFIG] [-trim_qual TRIM_QUAL]
                    [-head_crop HEAD_CROP] [-minth MINTH] [--CN CN]
                    [-od OUTDIR]
                    readsdir reference prefix
runsample.py: error: too few arguments
```

What you can take from this is:

- Anything inside of a [] block means that argument to the script is optional and has a default value that will be used if you do not specify it.
- readsdir, reference and prefix are all required arguments that you **MUST** specify

So to run the project with the fewest amount of arguments would be as follows(don't run this, just an example):

```
$> runsample.py ngs_mapper/tests/fixtures/functional/947 ngs_mapper/tests/fixtures/functional/947.ref
```

This will run the 947 data and use the 947.ref.fasta file to map to. All files will be prefixed with 947. Since we did not specify the -od argument, all the files from the pipeline get dumped into your current directory.

Most likely you will want to specify a separate directory to put all the 947 specific analysis files into. But how?

We can get extended help information which should print the defualts as well from any script by using the `--help` option

```
$> runsample.py --help
runsample.py --help
usage: runsample.py [-h] [--config CONFIG] [-trim_qual TRIM_QUAL]
                    [-head_crop HEAD_CROP] [-minth MINTH] [--CN CN]
                    [-od OUTDIR]
                    readsdir reference prefix

Runs a single sample through the pipeline

positional arguments:
  readsdir              Directory that contains reads to be mapped
  reference             The path to the reference to map to
  prefix                The prefix to put before every output file generated.
                        Probably the samplename

optional arguments:
  -h, --help            show this help message and exit
  --config CONFIG, -c CONFIG
                        Path to config.yaml file
  -trim_qual TRIM_QUAL  Quality threshold to trim[Default: 20]
  -head_crop HEAD_CROP  How many bases to crop off the beginning of the reads
                        after quality trimming[Default: 0]
  -minth MINTH          Minimum fraction of all remaining bases after
                        trimming/N calling that will trigger a base to be
                        called[Default: 0.8]
  --CN CN               Sets the CN tag inside of each read group to the value
                        specified.[Default: None]
  -od OUTDIR, --outdir OUTDIR
```

```
                        The output directory for all files to be put[Default:
                        /home/myusername/ngs_mapper]
```

You can see that `--help` gives us the same initial output as just running runsample.py without any arguments, but also contains extended help for all the arguments. The `--help` argument is available for all ngs_mapper scripts that end in .py(If you find one that doesn't, head over to Creating Issues and file a new Bug Report.

So you can see the -od option's default is our current directory. So if we want our analysis files to go into a specific directory for each sample we run we can specify a different directory. While we are at it, lets try specifying some of the other optional arguments too.

Let's tell runsample.py to put our analysis into a directory called 947 and also tell it to crop off 20 bases from the beginning of each read.

```
$> runsample.py -od 947 -head_crop 20 ngs_mapper/tests/fixtures/functional/947 ngs_mapper/tests/fixtu
2014-12-22 10:17:52,465 -- INFO -- runsample       --- Starting 947 ---
2014-12-22 10:21:28,526 -- INFO -- runsample       --- Finished 947 ---
```

You can see from the output that the sample started and finished. If there were errors, they would show up in between those two lines and you would have to view the Help documentation.

So what analysis files were created? You can see them by listing the output directory:

```
$> ls 947
-rw-r--r--. 1 myusername users 36758279 Dec 22 10:19 947.bam
-rw-r--r--. 1 myusername users       96 Dec 22 10:19 947.bam.bai
-rw-r--r--. 1 myusername users    10869 Dec 22 10:21 947.bam.consensus.fasta
-rw-r--r--. 1 myusername users   269058 Dec 22 10:21 947.bam.qualdepth.json
-rw-r--r--. 1 myusername users   204502 Dec 22 10:21 947.bam.qualdepth.png
-rw-r--r--. 1 myusername users  1291367 Dec 22 10:20 947.bam.vcf
-rw-r--r--. 1 myusername users     2414 Dec 22 10:21 947.log
-rw-r--r--. 1 myusername users   307180 Dec 22 10:21 947.reads.png
-rw-r--r--. 1 myusername users    10840 Dec 22 10:17 947.ref.fasta
-rw-r--r--. 1 myusername users       10 Dec 22 10:18 947.ref.fasta.amb
-rw-r--r--. 1 myusername users       67 Dec 22 10:18 947.ref.fasta.ann
-rw-r--r--. 1 myusername users    10744 Dec 22 10:18 947.ref.fasta.bwt
-rw-r--r--. 1 myusername users     2664 Dec 22 10:18 947.ref.fasta.pac
-rw-r--r--. 1 myusername users     5376 Dec 22 10:18 947.ref.fasta.sa
-rw-r--r--. 1 myusername users     2770 Dec 22 10:21 947.std.log
-rw-r--r--. 1 myusername users    17219 Dec 22 10:18 bwa.log
-rw-r--r--. 1 myusername users      380 Dec 22 10:20 flagstats.txt
-rw-r--r--. 1 myusername users      249 Dec 22 10:21 graphsample.log
-rw-r--r--. 1 myusername users   137212 Dec 22 10:19 pipeline.log
drwxr-xr-x. 2 myusername users     4096 Dec 22 10:21 qualdepth
drwxr-xr-x. 2 myusername users     4096 Dec 22 10:18 trimmed_reads
drwxr-xr-x. 2 myusername users     4096 Dec 22 10:17 trim_stats
```

You can view information about each of the output files via the runsample-output-directory

An easy way to view your bam file quickly from the command line if you have igv installed is like this:

```
igv.sh -g 947/947.ref.fasta 947/947.bam
```

Using runsamplesheet.sh to run multiple samples in parallel

`scripts/runsamplesheet` is just a wrapper script that makes running `runsample.py` on a bunch of samples easier.

You just have to first create a `samplesheet` then you just have to run it as follows:

```
$> runsamplesheet.sh /path/to/NGSData/ReadsBySample samplesheet.tsv
```

So let's run the 947 and 780 samples as our example.

1. Make a directory for all of our analysis to go into

```
$> mkdir -p tutorial
$> cd tutorial
```

2. Create a new file called samplesheet.tsv and put the following in it(you can use `gedit samplesheet.tsv` to edit/save the file):

```
947 ../ngs_mapper/tests/fixtures/functional/947.ref.fasta
780 ../ngs_mapper/tests/fixtures/functional/780.ref.fasta
```

3. Run your samplesheet with runsamplesheet.sh

```
$> runsamplesheet.sh ../ngs_mapper/tests/fixtures/functional samplesheet.tsv
2014-12-22 12:30:25,381 -- INFO -- runsample       --- Starting 780 ---
2014-12-22 12:30:25,381 -- INFO -- runsample       --- Starting 947 ---
2014-12-22 12:30:50,834 -- INFO -- runsample       --- Finished 780 ---
2014-12-22 12:34:08,523 -- INFO -- runsample       --- Finished 947 ---
1.82user 0.05system 0:01.01elapsed 185%CPU (0avgtext+0avgdata 242912maxresident)k
0inputs+728outputs (1major+26371minor)pagefaults 0swaps
5.02user 0.11system 0:04.03elapsed 127%CPU (0avgtext+0avgdata 981104maxresident)k
0inputs+3160outputs (1major+77772minor)pagefaults 0swaps
2014-12-22 12:34:19,843 -- WARNING -- graph_times     Projects/780 ran in only 25 seconds
2014-12-22 12:34:19,843 -- INFO -- graph_times     Plotting all projects inside of Projects
```

You can see that the pipeline ran both of our samples at the same time in parallel. The pipeline tries to determine how many CPU cores your system has and will run that many samples in parallel.

You can then view all of the resulting output files/directories created

```
$> ls -l
total 1184
-rw-r--r--. 1 myusername users   2101 Dec 22 12:34 graphsample.log
-rw-r--r--. 1 myusername users  50794 Dec 22 12:34 MapUnmapReads.png
-rw-r--r--. 1 myusername users 756139 Dec 22 12:34 pipeline.log
-rw-r--r--. 1 myusername users  34857 Dec 22 12:34 PipelineTimes.png
drwxr-xr-x. 4 myusername users   4096 Dec 22 12:34 Projects
-rw-r--r--. 1 myusername users 292764 Dec 22 12:34 QualDepth.pdf
-rw-r--r--. 1 myusername users  52064 Dec 22 12:34 SampleCoverage.png
-rw-r--r--. 1 myusername users    122 Dec 22 12:28 samplesheet.tsv
drwxr-xr-x. 2 myusername users   4096 Dec 22 12:34 vcf_consensus
```

You can view what each of these files means by heading over to the `scripts/runsamplesheet`

1.5.2 Changing defaults for pipeline stages

If you want to change any of the settings of any of the pipeline stages you will need to create a config.yaml and supply it to `runsample.py` using the -c option. You can read more about how to create the config and edit it via the config.yaml script's page

1.5.3 Rerunning Samples

Rerunning samples is very similar to just running samples.

1. Copy and edit the existing `samplesheet` and comment out or delete the samples you do not want to rerun.
2. **Run the `scripts/runsamplesheet` script on the modified samplesheet**
    - **Note**: As of right now, you will have to manually remove the existing project directories that you want to rerun.
3. **Regenerate graphics for all samples**
    - The -norecreate tells it not to recreate the qualdepth.json for each sample which is very time consuming. The reran samples should already have recreated their qualdepth.json files when `runsample.py` was run on them.

```
graphs.sh -norecreate
```

4. You should not have to rerun `scripts/consensuses` as it just symlinks the files

1.5.4 Temporary Directories/Files

The pipeline initially creates a temporary analysis directory for each sample that you run with `runsample.py`. By default this directory will be created in your system's configured temporary directory(most likely /tmp). This is especially useful if your /tmp partition is not very large or if you have a custom temporary partition that is on a very fast hard drive such as a Solid State Drive that you want to use.

It is important that you first create the temporary directory as it will not be created for you(/tmp is already available from when Linux was installed though, FYI).

You can control what directory this is by utilizing the TMPDIR environmental variable as follows:

```
mkdir -p /path/to/custom/tmpdir
export TMPDIR=/path/to/custom/tmpdir
SAMPLE=samplename
runsample.py /path/to/NGSData/ReadsBySample/${SAMPLE} /path/to/reference ${SAMPLE} -od Projects/${SAM
```

1.6 Help

1.6.1 Creating Issues

Since the source code for this project is hosted on GitHub, it also comes with an issue tracker.

All feature requests, bugs and other communications are all kept there. This gives both the developers and users a common place to discuss all aspects of the pipeline as well as give a nice resource to help find answers to questions that might have already been asked.

Submitting a Bug

First, please make sure you read through the *Frequently Asked Questions* and also do a search for existing similar issues

If you can't find anything in either of those sources that address your issue, then go ahead and create a New Issue

Make sure to include the following information:

- Description of the error that you are encountering
- The command you ran that generated the error

- The entire *Traceback Error* if there is one
- Any pertinant information about the issue

You may be asked later to attach files from your project so don't delete any of the files yet.

Eventually you will run across some errors. No application/software is without bugs. Here we will compile all of the most common errors and what to look for to find out what is going on

1.6.2 Traceback Error

You will likely encounter a Traceback error at some point due to either a bug or maybe you are running one of the commands incorrectly.

The traceback errors will look like this:

```
Traceback (most recent call last):
  File "/home/username/.ngs_mapper/bin/roche_sync", line 9, in <module>
    load_entry_point('ngs_mapper==1.0.0', 'console_scripts', 'roche_sync')()
  File "/home/username/.ngs_mapper/lib/python2.7/site-packages/ngs_mapper/roche_sync.py", line 100, i
    args = parse_args()
  File "/home/username/.ngs_mapper/lib/python2.7/site-packages/ngs_mapper/roche_sync.py", line 236, i
    defaults = config['roche_sync']
  File "/home/username/.ngs_mapper/lib/python2.7/site-packages/ngs_mapper/config.py", line 29, in __g
    'Config is missing the key {0}'.format(key)
ngs_mapper.config.InvalidConfigError: Config is missing the key roche_sync
```

The easiest way to get good information from the traceback is by working your way backwards(from the bottom to the top).

From this Traceback you should notice that the last line is telling you that the config.yaml file is missing the key roche_sync. You would then edit your config.yaml file and ensure that key exists and then rerun the `python setup.py install` portion of the Install.

The traceback is simply Python's way of displaying how it got to the error that was encountered. Typically, but not always, the last line of the output contains the most relevant error. If you submit a bug report, make sure to include the entire Traceback though.

1.6.3 Frequently Asked Questions

1. **There is an error. What do I do?** There are a few log files that you can check. The output on your screen should give you the location of the log file to check for errors.

   As well you can look under the directory of any project and look in files that end in .log

   For instance, if a run fails for any reason it will spit many lines to the screen. When you read through the lines you will see one that mentions "Check the log file" followed by a path to a bwa.log. Navigate to the bwa.log to view a detailed log of what happened.

   There are two other log files which are in the same directory as bwa.log [samplename].std.log and [samplename].log. You can check any of these log files to determine what happend during the run.

   Finally, you can also check the pipeline.log file that is generated when the pipeline is done or if it err'd out.

   If you are still not sure, you can search through previous issues on the GitHub Issue Tracker and/or submit a new bug/feature

2. **Where should I run the analysis?** This is for the most part up to you but eventually you will want the entire analysis folder to end up under /path/to/Analysis somewhere You will want to minimize how much the traffic has to travel across the network though. So if you simply create a folder under

/path/to/Analysis/PipelineRuns and then you run the pipeline from there, you will essentially be doing the following:

- Reading the reads across the network for each sample
- Writing the bam files across the network for each sample
- Reading the bam files across the network for each sample
- Writing stats across the network
- Reading the stats file across the network
- Writing graphics files across the network

Suggestion Create the analysis folder somewhere on your computer and run the pipeline there and then transfer the entire folder to the storage server afterwards

3. **How many CPUs does my computer have?** Try running the following command to get how many physical CPU's and how many cores/threads they have

```
for pid in $(awk '/physical id/ {print $4}' /proc/cpuinfo |sort|uniq)
do
    echo "--- Processor $pid ---"
    egrep -xA 12 "processor[[:space:]]: $pid" /proc/cpuinfo
done
```

4. **How many CPUs should I use?** Check out the command above for more info on how to get how many CPU/Core/Threads you have. Probably best to use (cpu cores * number of processors)

   If your output was the following then you would probably want to use (2 * 6)

```
--- Processor 0 ---
processor         : 0
...
physical id:      : 0
siblings          : 12
core id           : 0
cpu cores         : 6
...
--- Processor 1 ---
...
processor         : 0
physical id:      : 1
siblings          : 12
core id           : 0
cpu cores         : 6
...
```

   That all being said, you could also try using (number of processors * siblings) or 24 in the above example, but that may actually slow down your analysis

5. **How much RAM do I have?** The following command will tell you how much memory you have in MB

```
free -m | awk '/Mem:/ {print $2}'
```

6. **The pipeline fails on samples and the bwa.log says something about failing on the reference index** Make sure to check that you have permissions to read the reference file. The last thing to check is that the reference is formatted correctly in fasta format.

7. **There is an error running vcf_consensus.py that has to do with string index out of bounds** This has to do with an outdated version of base_caller.py generating the vcf file you are trying to run vcf_consensus.py on. See Issue #143 for more information on how to fix that.

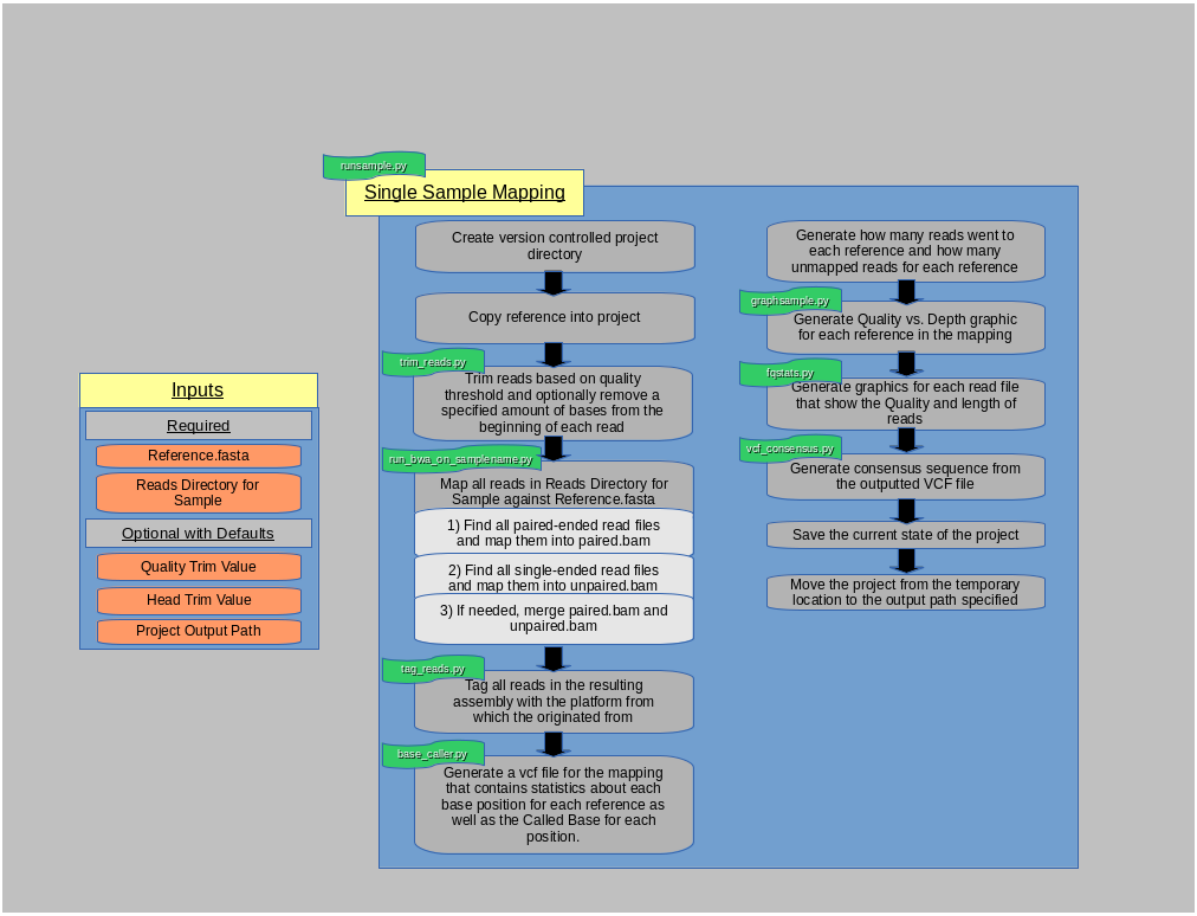
8. **The pipeline fails on a sample and the log says Somehow no reads were compiled** This usually indicates that it could not find any reads inside of the location you specified that should contain sample reads. Make sure that the directory you specified when you ran `scripts/runsamplesheet` or `ngs_mapper.runsample` actually contains a directory with reads for every sample you are running. Also check for errors near the top of the log file that say anything about why any reads might have been skipped

9. **The pipeline keeps failing on all of my samples or the logs say something about No Space Left On Device** Please check your /dev/shm and /tmp to see if either is full(`df -h`). You can clear out all of the left-over junk from the pipeline by issuing `rm -rf /tmp/runsample* /dev/shm/mapbwa*` Also, you may need to tell the pipeline to use a different temporary directory. See *Temporary Directories/Files* for more information.

10. **You get a Traceback error that contains ngs_mapper.config.InvalidConfigError: Config is missing the key missingkey** This indicates that the initial config.yaml file that you created during the Install is missing a required key: value pair called missingkey. This most likely happened because you updated the pipeline which introduced new keys in config.yaml.base that you need to add to your config.yaml.

    Once you add those new keys, you will need to rerun the `python setup.py install` portion of the Install.

1.7 Pipeline Info



1.7.1 Pipeline Output

The pipeline can be run as a batch job or it can be run individually. That is, you can run it on many samples by supplying a `samplesheet` to `runsamplesheet.sh` or a single sample can be run via `runsample`. As such, you need to understand that [[runsamplesheet.sh]] essentially just runs `runsample` for every sample in your [[samplesheet]] then runs a few graphics scripts afterwards on all the completed projects.

- **Individual sample project directories under Projects/**
  - runsample.py output
- **Entire Project output**
  - `scripts/graphs` output

1.8 Scripts

1.8.1 User Scripts

These are scripts that you run directly that will run the Supplemental scripts

- stats_at_refpos
- runsamplesheet
- runsample
- graphs
- consensuses
- miseq_sync
- roche_sync
- sanger_sync
- rename_sample
- make_example_config

1.8.2 Supplemental

These are scripts that you can run manually, however, they are run automatically by the User Scripts above

- run_bwa_on_samplename
- vcf_consensus
- gen_flagstats
- graphsample
- graph_mapunmap
- tagreads
- base_caller
- graph_times
- trim_reads
- fqstats
- sample_coverage

1.8.3 Libraries

Python Scripts/Modules that you can import to do other analysis

- ngs_mapper.run_bwa
- ngs_mapper.reads
- ngs_mapper.data
- ngs_mapper.bam
- ngs_mapper.alphabet
- ngs_mapper.stats_at_refpos
- ngs_mapper.samtools
- ngs_mapper.log

1.8.4 Deprecated

Scripts that are no longer used, but kept for reference in the deprecated directory

- varcaller.py
- variants.sh
- perms.sh
- gen_consensus.sh
- setup
- install.sh

1.9 Development

Contributing to the pipeline is fairly straight forward. The process is as follows:

1. Fork the VDBWRAIR ngs_mapper project on GitHub
2. git clone your forked version to your local computer
3. **Make changes to the code and ensure everything is tested**
    - [[Make Tests]]
4. Once you have tested all your changes you should commit them and push them up to your github fork of the project
5. After you have pushed your changes to your fork on github you can create a pull request which essentially notifies the ngs_mapper maintainers that you have changes that you would like to apply and they can try them out.

1.9.1 Test Environment

The easiest way to ensure that the installer and everything works is to bring up a blank virtual machine to test inside of

The project is configured with a Vagrant file to make this easier

The Vagrant file that comes with the pipeline is configured to automatically `provision` either a CentOS 6.5 or Ubuntu 14.04 virtual machine. You can bring either or both up with one of the following commands:

- CentOS 6.5

```
vagrant up centos65
```

- Ubuntu 14.04

```
vagrant up ubuntu1404
```

- Both

```
vagrant up
```

1.10 TODO List

CHAPTER 2

Indices and tables

- genindex
- modindex
- search