# ngs$_m apper Documentation$

***Release 1.5.0***

**Tyghe Vallard, Melanie Melendrez**

**Oct 26, 2017**

# Contents

**Version**: 1.5.0

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:

# Install

## 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**
  - This really depends on your data size

    If you are analyzing a 96 sample run then you should be fine with 1GB per CPU core

    If you are analyzing a 24 sample run then you will probably need about 4GB per CPU core since there will be more data

### 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.

## Installation

1. Clone/Download the version you want
   (a) Clone the code

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

   (b) Check which versions are available

```
git tag
```

   (c) Checkout the version you want(current version 1.5.0)

```
git checkout -b vX.Y.Z vX.Y.Z
```

2. 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
```

3. Install

   The project now comes with a much more simplified installer which is based on miniconda.

   The following will install the project into the current directory that you are in.

```
./install.sh miniconda
```

4. PATH Setup

   Once the project is installed you will need to setup your PATH environmental variable to include the

```
export PATH=$PWD/miniconda/bin:$PATH
```

   You can put this into your .bashrc file inside your home directory so that any time you open a new terminal it automatically is run.

   If you don't setup your .bashrc you will have to run the export command from above every time you open a new terminal.

### Verify install

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

```
ngs_mapper/tests/slow_tests.sh
```

# Upgrade

At this point there is no way to upgrade easily so please create a GitHub issue and we can assist you in upgrading.

# 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

## Example changing single script defaults

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

1. First what options are available for the command?

```
$> trim_reads --help
usage: trim_reads [-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 -q 5 /path/to/my/input.fastq
```

This process works pretty slick until you notice that there is no way to easily tell `runsample` 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.

## Example running `runsample` 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` as follows

```
$> runsample /path/to/NGSData /path/to/reference.fasta mysample -od␣
↪mysample -c config.yaml
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 -
↪--
```

## 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` by using the `RUNSAMPLEOPTIONS` variable

1. Generate your config.yaml template

```
make_example_config
```

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

```
$> RUNSAMPLEOPTIONS"-c config.yaml" runsamplesheet.sh /path/to/NGSData/
↪ReadsBySample samplesheet.tsv
```

### 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.

### 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 %

## Data Structure

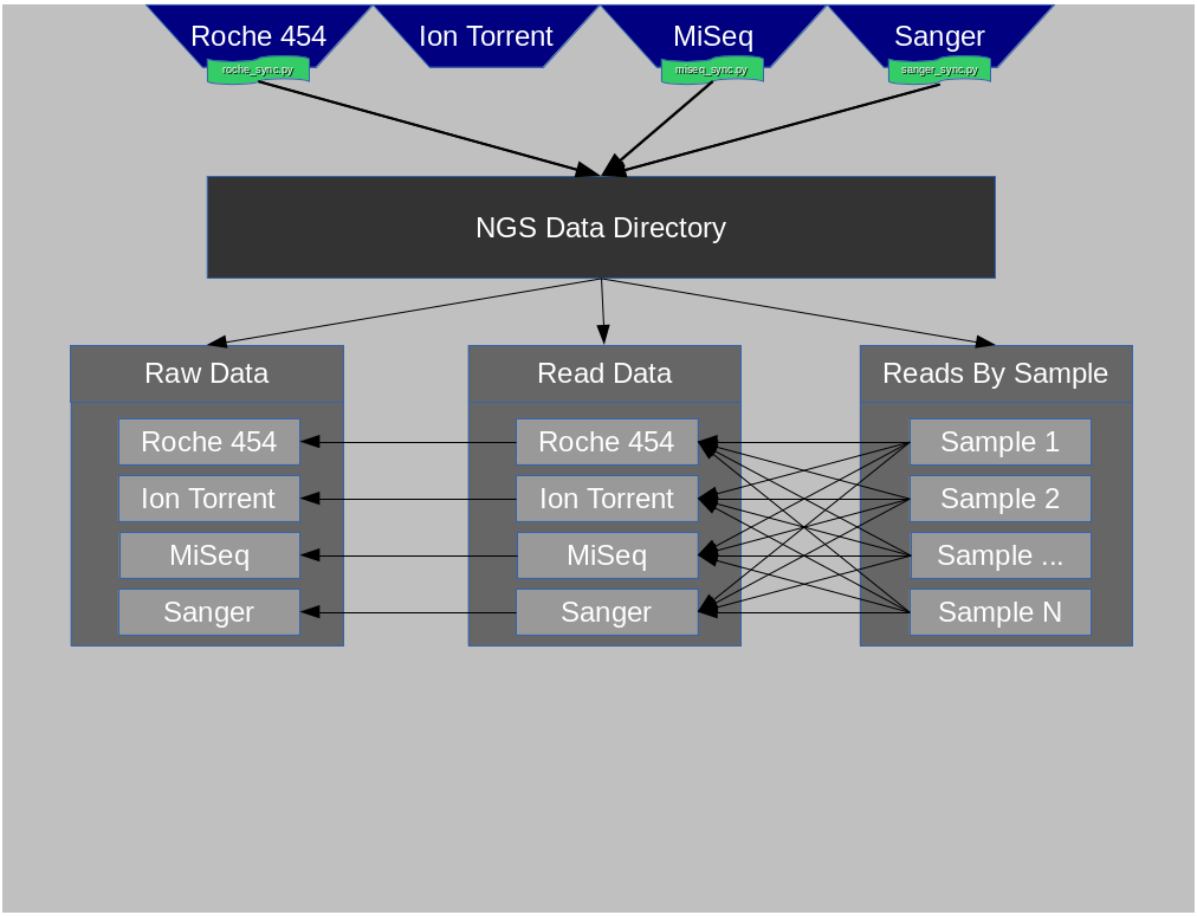
At this time data is organized inside of what is called the NGS Data Structure. This structure is composed of 3 critical directories.

- RawData
- ReadData
- ReadsBySample

### Getting Data into the data structure

See ngsdatasync

## Diagram




## 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)

## 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

### 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.o

### Platform Identification

See the following naming regular expressions defined in `ngs_mapper.data` for more information about how platforms are identified via the read identifiers inside the files

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

If you have files that do not match any platform the pipeline will essentially ignore them and you may get errors when you run `runsample`.

This should only be an issue if you somehow rename the identifiers.

## Primer File

You can set the primer file and the the primer options in the config.yaml file. You'll want to change the fields under trim_reads. To learn more about the options, see the help fields below, and see the documentation at http://www.usadellab.org/cms/?page=trimmomatic

```
trim_reads:
    primerfile:
        default:
        help:  'Primer File'
    primerseed:
        default: 2
        help:   'seed mismatches'
    palindromeclip:
        default: 30
        help: 'palindrome clip threshold'
    simpleclip:
        default: 20
        help: 'simple clip threshold'
```

The available options are:

```
primerfile: path to the primer file
primerseed: seed mismatches
palindromeclip: palindrome clip threshold
simpleclip: simple clip threshold
```

## Analysis

**Contents**



## 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 in your environments PATH.

See the *Install* for how to setup your PATH

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 and a/947 directory There is also a 780.ref.fasta and 947.ref.fasta file. The 780 and 947 directory contain all the read files for the 780 and 947 samples while the 780.ref.fasta and 947.ref.fasta is the reference to map to for each project. You can ignore the .conf files, they are used by the automated tests.

### Quick note about platform identification

Reads are identified by the way the first identifier in each file is named.

You can read more about this *here*

### Using runsample to run a single sample

Some times you just need to run a single sample. Here we will use the *runsample* script 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` script to use

This is just an example output and may not match your exact output.

```
$> runsample
usage: runsample [-h] [--config CONFIG] [-trim_qual TRIM_QUAL]
                 ...
                 readsdir reference prefix
runsample: 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

### Simplest form of runsample

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

```
$> runsample ngs_mapper/tests/fixtures/functional/947 ngs_mapper/tests/fixtures/
↪functional/947.ref.fasta -od 947 947
```

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?

### Getting extended help for runsample

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

```
$> runsample --help
runsample --help
usage: runsample [-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 without any arguments, but also contains extended help for all the arguments. The `--help` argument is available for all ngs_mapper scripts. 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.

### Specifying output directory for analysis

Let's tell runsample 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 -od 947 -head_crop 20 ngs_mapper/tests/fixtures/functional/947 ngs_
↪mapper/tests/fixtures/functional/947.ref.fasta 947
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.

### Run with fasta files

Simply run runsample with the –fasta argument:

```
$> runsample --fasta -od 947 ngs_mappers/tests/fixtures/fasta/ ngs_mapper/tests/
↪fixtures/functional/947.ref.fasta 947
```

This will allow the pipeline to read fasta files as well as other input files inside of the readsdir directory. Without –fasta, all fasta files inside readsdir would be skipped. When you use this option, fasta files are converted to fastq files with dummy quality of 40 in the first step of the pipeline.

### Specifying specific platforms to map

Sometimes you may find the need to only run specific platforms. Maybe you only will want to run MiSeq read files through the pipeline.

The 947 example project has Roche454, MiSeq and Sanger read files in it, so we can use it in this example to only map the MiSeq read files

1. Generate your example config which we will edit

   ```
   make_example_config
   ```

2. Now edit the config.yaml file generated in the current directory

   Find the trim_reads section and change the default under platforms to be

   ```
   trim_reads:
       headcrop:
           default: 0
           help: 'How many bases to crop off the beginning of the␣
   ↪reads after quality
                trimming[Default: %(default)s]'
       outputdir:
           default: trimmed_reads
           help: 'Where to output the resulting files[Default:
   ↪%(default)s]'
       q:
           default: 20
           help: 'Quality threshold to trim[Default: %(default)s]'
       platforms:
           choices:
           - MiSeq
           - Sanger
           - Roche454
           - IonTorrent
           default:
           - MiSeq
           #- Sanger
           #- Roche454
           #- IonTorrent
           help: 'List of platforms to include data for[Default:
   ↪%(default)s]'
   ```

   Notice that we have commented out(put # before them) Sanger, Roche454 and IonTorrent. You can either comment them out or completely delete them. It is up to you.

3. Then you can run `runsample` with the `-c config.yaml` argument and it will only use MiSeq reads

   ```
   $> runsample -od 947 -head_crop 20 ngs_mapper/tests/fixtures/functional/
   ↪947 ngs_mapper/tests/fixtures/functional/947.ref.fasta 947 -c config.
   ↪yaml
   ```

### Output from runsample explained

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

### Viewing bam files

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` 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 advanced usage and what each of these output files mean by heading over to the scripts/runsamplesheet

## 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` using the -c option. You can read more about how to create the config and edit it via the *config.yaml* script's page

## 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` was run on them.

        ```
        graphs.sh -norecreate
        ```

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

## Temporary Directories/Files

The pipeline initially creates a temporary analysis directory for each sample that you run with `runsample`.

The name of this temporary directory will be samplenameRANDOMrunsample

This directory will be located inside of each project's specified output directory that was given with `-od`

If the project fails to complete for some reason then you will need to look inside of that directory for relevant log files to inspect what happened.

## Integration with the PBS Schedulers

runsample has the ability to output a PBS job file instead of running. This may be useful if you have access to a PBS Cluster. By default the PBS job that is generated is very simplistic.

- The job will change directory to the same directory that qsub is run from
- runsample is then run with the same arguments that were given to generate the pbs job without the –qsub arguments.

### Example

```
$> runsample ngs_mapper/tests/fixtures/functional/947{,.ref.fasta} 947 --outdir␣
↪947test --qsub_l nodes=1:ppn=1 --qsub_M me@example.com
#!/bin/bash
#PBS -N 947-ngs_mapper
#PBS -j oe
#PBS -l nodes=1:ppn=1
#PBS -m abe
#PBS -M me@example.com
cd $PBS_O_WORKDIR
runsample ngs_mapper/tests/fixtures/functional/947 ngs_mapper/tests/fixtures/
↪functional/947.ref.fasta 947 --outdir 947test
```

You can see that the job that was generated essentialy just stripped off any –qsub_ arguments and will rerun the same runsample command in the job.

# Help

## 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

## 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, in main
    args = parse_args()
  File "/home/username/.ngs_mapper/lib/python2.7/site-packages/ngs_mapper/roche_sync.
↪py", line 236, in parse_args
    defaults = config['roche_sync']
  File "/home/username/.ngs_mapper/lib/python2.7/site-packages/ngs_mapper/config.py",␣
↪line 29, in __getitem__
    '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.

## 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 that has to do with string index out of bounds** This has to do with an outdated version of base_caller generating the vcf file you are trying to run vcf_consensus 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

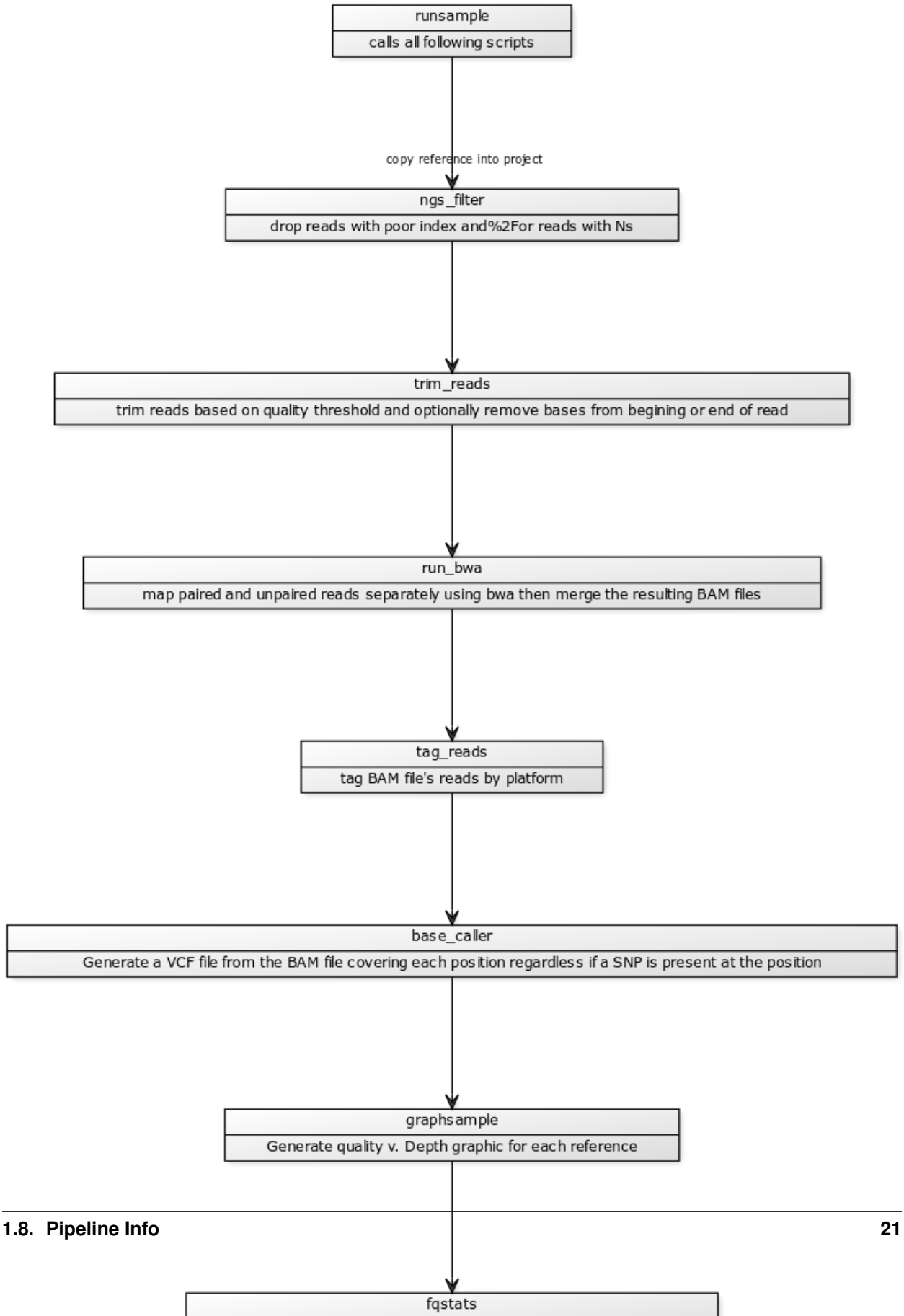
9. **The pipeline keeps failing on all of my samples or the logs say something about No Space Left On Device** Please check your /dev/shm and /tmp to see if either is full(`df -h`). You can clear out all of the left-over junk from the pipeline by issuing `rm -rf /tmp/runsample* /dev/shm/mapbwa*` Also, 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*.

11. **You get errors related to `no display name and no $DISPLAY environemt variable` when createing graphics** See Issue 75

## Pipeline Info



### 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 output
- **Entire Project output**
  - scripts/graphs output

## Scripts

### 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`
- `ion_sync`
- `rename_sample`
- `make_example_config`

### 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
- ngs_filter
- fqstats
- sample_coverage

### 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

### 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

## 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.

### 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 12.04 virtual machine. You can bring either or both up with one of the following commands:

- CentOS 6.5

```
vagrant up centos
```

- Ubuntu 14.04

```
vagrant up ubuntu
```

- Both

```
vagrant up
```

## TODO List

CHAPTER 2

# Changelog

## Version 1.5.3

- Fixing readme for docker usage

## Version 1.5.2

- Bugfix for install not being able to find ncurses

## Version 1.5.1

- Bugfix for X Display being require for Matplotlib graphics
- Bugfix for ImageMagick missing fonts issue
- Bugfix for pip/conda setuptools conflict
- Bugfix: qualdepth graph no longer cuts off ends
- Deleted out-of-date INSTALL.md file that used virtualenv instead of conda

## Version 1.5.0

- Continuous Delivery support added for travis
- nfilter will now simply symlink if no options are supplied essentially skipping itself
- nfilter utilizes threads from config file
- Froze versions of all dependencies to remove issues when authors update dependencies that cause unwanted side-effects

- config file now has THREADS default
- fix for bug where some miseq reads were not identified correctly in tagreads
- convert functions now support output directory
- bug fix for nfilter symlinking
- fix for qsub job output from runsample
- no longer name files with filtered. prefix
- Pipeline now works with fasta files using the –fasta flag

## Version 1.4.2

- Pipeline now handles gzip(.gz) input files
- Pipeline now handles ab1 input files
- Added Zenodo badge

## Version 1.4.1

- IGV is installed with pipeline
- samtools version reverted back to same version as pre-1.4.0

## Version 1.4.0

- Installation now utilizes miniconda to handle system dependencies such as bwa, samtools, trimmomatic, imagemagick. This is a substantial difference and will require a complete reinstall of the pipeline to upgrade. Miniconda installation removes a lot of code that needed to be maintained and streamlines the installation and makes it much faster.
- Added install.sh that makes installing/upgrading much easier. The tests also use this so the installation is tested much better now.
- Pipeline utilizes requirements-conda.txt to determine python+system software dependencies. This allows specifying versions and removes need for a system administrator to install.
- runsample now supports –primer-file option and other primer trimming options which will utilize trimmomatic's ILLUMINACLIP option
- runsamplesheet.sh supports an optional additional column in a given samplesheet that represents the primer fasta file to use to find sequences to trim out.
- Pipeline now looks for amount of threads instead of cpu cores. This will mean that on systems with hyperthreading that 2x more samples will run in parallel than before.
- Fixed bug where some parts of pipeline were not logging at all
- Fixed bug where graphs.sh could fail, yet pipeline would continue as if nothing was wrong
- Updated functional tests to include primer test
- Updated functional tests to output more information

## Version 1.3.0

- Added ngs_filter stage/script that can filter based on index fastq files as well as reads that contain an N. This stage is off by default.
- Fixed a bug where some scripts were not logging properly

## Version 1.2.4

- Fixes documentation issue with umask for sync user

## Version 1.2.3

- Added travis-ci support to automatically run tests when code is pushed to github
- Projects now default to running inside of a temporary directory inside of the specified output directory(-od)
- runsample now sets TMPDIR to tmpdir inside of output directory so that all analysis is run within that directory

## Version 1.2.2

- runsample accepts –qsub_m and –qsub_l commands which will direct it to return a PBS qsub job that can be piped into qsub
- Added Python 2.6 support

## Version 1.2.1

- Removed all occurances of bqd.mpileup and replaced with samtools.mpileup
- Changed bqd.parse_pileup such that it utilizes samtools.MPileupColumn to generate the dictionary items
- Remove legacy BamCoverage code that is not used anywhere
- Added support to select reads by specific platforms in runsample.py
- Fixed bug where MiSeq Index reads were being included in the mapping
- Renamed unpaired read file name that is produced by trim_reads from a generic Roche454 read name to simply unpaired_trimmed.fastq

## Version 1.2.0

- Added reflen to qualdepth.json files since length only told you the length of the assembly and not the reference.
- Fixed issue where coverage graphic was not drawing gap lines at the end of references because there was no data.
- sample_coverage colors were hard to distinquish so they were changed

- Bug with sample_coverage where certain combinations of # of references and # of samples would generate a graphic where sub-plots for each reference were overlapping
- Fixed incorrect command in doc/README.rst for how to open documentation with Firefox
- Fixed issue with sample_coverage's usage statement and arguments description
- Fixed issue when no reads mapped and graphsample.py would raise an exception
- Fixed an issue when there were directories inside of the path specified that contains read files
- Replaced all .py scripts with same name but without .py. This is the correct way to have binary scripts for python. Aka, runsample.py is now just runsample

## Version 1.1.0

- Documentation updates
- Platforms now identified via identifiers inside read files instead of filenames
- IonTorrent sync added
- Various bug fixes
- base_caller.py can now utilize multiple processes to speed up analysis
- Documentation now installs with the pipeline
- run_bwa no longer makes temp directory but instead uses output path

CHAPTER 3

# Indices and tables

- genindex
- modindex
- search