Nanopype

Nanopype is a snakemake based pipeline providing convenient Oxford Nanopore Technology (ONT) data processing and storage solutions.

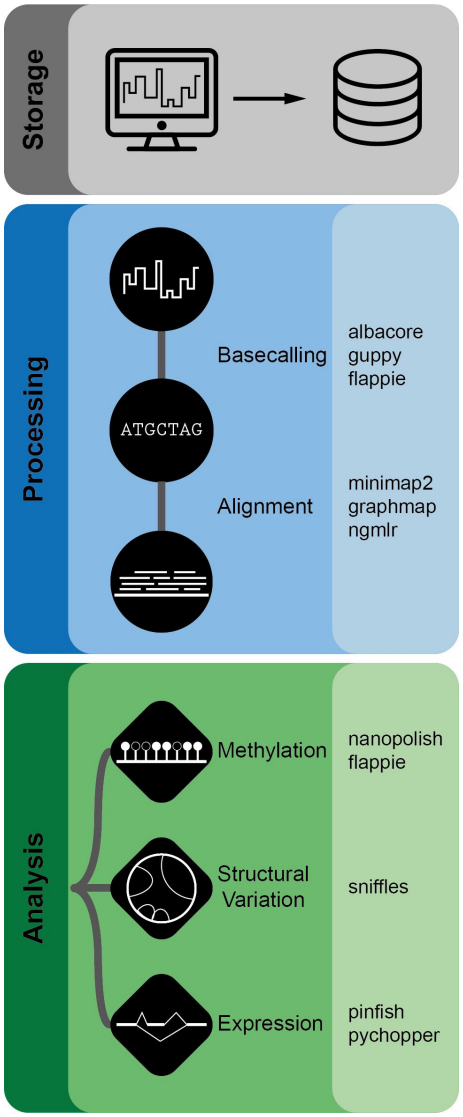
The pipeline is split in a processing part including basecalling and alignment and an analysis part covering further downstream applications. A summary of all included tools is given in the tools section.

To get started the installation chapter describes the available installation options depending on the operation system, available hardware and already existing environments.

Recurring steps of the nanopore data analysis are covered under workflow for both local and cluster usage.

The **modules** part covers an in depth description of all available tools and workflows together with their respective configuration options. This section is the main reference of the pipeline.

Finally for new users the tutorial might be helpful to learn the general concepts and usage of the pipeline. To complete the tutorial the test reads included in the package are sufficient and no separate wet-lab experiemnt is required.



Tools

Nanopype integrates a growing number of different tools merged into processing workflows for common use cases.

Processing

Basecalling

- Albacore (access restricted to ONT community, manual install required)
- Guppy
- Flappie *https://github.com/nanoporetech/flappie*

Alignment

- Minimap2 *https://github.com/lh3/minimap2*
- Graphmap *https://github.com/isovic/graphmap*
- Ngmlr *https://github.com/philres/ngmlr*

Analysis

Methylation detection

- Nanopolish *https://github.com/jts/nanopolish*
- Flappie *https://github.com/nanoporetech/flappie*

Transcriptome

- Pinfish *https://github.com/nanoporetech/pinfish*

Structural variation

- Sniffles *https://github.com/fritzsedlazeck/Sniffles*

Demultiplexing

- Deepbinner *https://github.com/rrwick/Deepbinner*

Miscellaneous

- Samtools *https://github.com/samtools/samtools*
- Bedtools *https://github.com/arq5x/bedtools2*
- UCSCTools *http://hgdownload.cse.ucsc.edu/admin/exe/linux.x86_64/*
    - bedGraphToBigWig

Installation

Important: If you encounter any trouble within the installation, open an issue on github to make others aware of the problem or write an E-Mail to giesselmann[at]molgen.mpg.de. There is nothing more annoying than software that does not work.

In general Snakemake and thus also Nanopype is python based and therefore available on Windows, MacOS and Linux. However the included bioinformatic tools are often only available for Linux like operating systems. We support and test the fully featured pipeline on Linux and MacOS and through containerization enable reduced functionality on Windows.

Docker & Singularity container

Docker and Singularity provide encapsulation of software environments into so-called *containers* or *images*. A docker-image can e.g. based on Ubuntu contain the basecaller *flappie* with all of its dependencies and without further installation be executed on any operating system with just Docker installed. Singularity can handle both Docker and Singularity images and is supported by Snakemake allowing the execution of workflows in isolated environments.

We provide Singularity images per module, wrapper to build and install tools from source and a single stand-alone Docker image of the entire pipeline. Depending on operating system, required functionality and experience one the following methods should be selected:

Method	Singularity	Source	Docker
Linux	yes	yes	yes
MacOS	yes	yes	yes
Windows	no	no	yes
Tools	all	all	public
Cluster	yes	yes	no
Complexity	low	high	low

Independent of the operating system you'll need a working **python3** installation and **git** to clone the pipeline repository. On Linux and MacOS these are likely already present, on Windows we can recommend gitforwindows as a starting point.

Recommendation: If possible use the Singularity workflow of Nanopype. If you have already existing installations or wish to customize or optimize the build process go for the source installation. If you want to quickly test or run the pipeline on a Windows machine use the all-in-one Docker container.

Configuration

Independent of the installation the pipeline needs to be configured once globally and for each sample locally.

Singularity

In order to use the Singularity driven version of Nanopype a working python3 with Snakemake and the pipeline repository itself are sufficient. At least python version 3.4 is required and we recommend to use a virtual environment. Additionally Singularity needs to be installed system wide. On Ubuntu the package is called *singularity-container*. The installation requires root privileges and might require help of your IT department. We currently test workflows with Singularity version 2.4.2.

Start with creating a virtual environment:

```
python3 -m venv /path/to/your/environment
cd /path/to/your/environment
source bin/activate
```

Or using conda:

```
conda create -n nanopype python=3.4 anaconda
source activate nanopype
```

Snakemake

Nanopype is based on the Snakemake pipeline engine. With python3.4 onward pip is already installed and you get Snakemake by executing:

```
pip3 install --upgrade snakemake
```

Alternatively you might use the conda package manager:

```
conda install -c bioconda -c conda-forge snakemake
```

Nanopype relies on latest Snakemake features, please consider updating your Snakemake from the bitbucket repository. From your home or project directory run:

```
mkdir -p src && cd src
git clone https://bitbucket.org/snakemake/snakemake.git
cd snakemake
pip3 install . --upgrade
cd ..
```

Nanopype

Finally install Nanopype from github.com/giesselmann. If you use conda you find pip in the *bin/* folder of the conda installation. If you use a conda virtual environment use the pip from the *envs/nanopype/bin*.

```
git clone --recursive https://github.com/giesselmann/nanopype
cd nanopype
pip3 install -r requirements.txt
cd ..
```

To deactivate a virtual python environment after installation or usage of the pipeline just type:

```
deactivate
```

for plain and

```
source deactivate
```

for conda virtual environments.

Mission accomplished! Everything else is solved at run time by Snakemake and Nanopype.

> **Configuration**
>
> Do not forget to follow the global configuration once and for each sample the local one.

When using the pipeline you'll have to add the **--use-singularity** flag to every Snakemake command. This will pull only the required images in the background.

Please also consider setting the **--singularity-prefix** to a common path, otherwise the images will be downloaded for every sample into *./.snakemake/singularity*. If you want to see all available Nanopype Singularity images or check their download sizes visit our DockerHub.

Source

Nanopype can be installed without root privileges as it's maintaining most of its dependencies by building required tools from source in a user accessible path. Currently the following list of system wide packages is required to build all included software packages (names on MacOS might be different):

- git wget
- gcc g++
- binutils autoconf make cmake
- zlib1g-dev bzip2 libbz2-dev
- liblzma-dev libncurses5-dev libcunit1-dev

These packages are likely present in most production environments. Please also refer to the Dockerfiles in the singularity folder of the pipeline repository. If you need only a subset of the provided tools the number of dependencies might decrease.

The following build process is the most flexible and customizable installation and might despite careful tests lead to errors on your specific system. Please do not hesitate to contact us and ask for help. Also check and open an issue where needed on github.

Start with creating a virtual python environment:

```
python3 -m venv /path/to/your/environment
cd /path/to/your/environment
source bin/activate
```

Or using conda:

```
conda create -n nanopype python=3.4 anaconda
source activate nanopype
```

Snakemake

Nanopype is based on the Snakemake pipeline engine. With python3.4 pip is already installed and you get Snakemake by executing:

```
pip3 install --upgrade snakemake
```

Alternatively you might use the conda package manager:

```
conda install -c bioconda -c conda-forge snakemake
```

Nanopype relies on latest Snakemake features, please consider updating your Snakemake from the bitbucket repository. From your home or project directory run:

```
mkdir -p src && cd src
git clone https://bitbucket.org/snakemake/snakemake.git
cd snakemake
pip3 install . --upgrade
cd ..
```

Nanopype

Finally install Nanopype from github.com/giesselmann. If you use conda you find pip in the *bin/* folder of the conda installation. If you use a conda virtual environment use the pip from the *envs/nanopype/bin*.

```
git clone --recursive https://github.com/giesselmann/nanopype
cd nanopype
pip3 install -r requirements.txt
cd ..
```

Tools

The installation will create the following folder structure relative to a given [INSTALL_PREFIX] directory:

```
|--INSTALL_PREFIX/
   |--src
   |--bin
   |--lib
   |--share
```

Nanopype integrates a variety of different **tools** merged into processing pipelines for common use cases. We provide Snakemake rules to download and build all dependencies. From within the Nanopype repository run

```
snakemake --snakefile rules/install.smk --directory /path/to/
INSTALL_PREFIX all
```

to build and install all tools into **src**, **bin** and **lib** folders of the INSTALL_PREFIX directory. To only build a subset or specific targets e.g. samtools you can use:

```
# core functionality of basecalling and alignment
snakemake --snakefile rules/install.smk --directory
[INSTALL_PREFIX] processing
# extended analysis functionality
snakemake --snakefile rules/install.smk --directory
[INSTALL_PREFIX] analysis
# specific tool only
snakemake --snakefile rules/install.smk --directory
[INSTALL_PREFIX] samtools
```

The --directory argument of Snakemake is used as installation prefix. By running Snakemake with e.g. -j 4 multiple targets are build in parallel at the cost of interleaved output to the shell. To further accelerate the build you may try the following to build 8 tools (-j 8) with 8 threads each (threads_build=8) using the CMake generator Ninja:

```
snakemake --snakefile rules/install.smk --directory
[INSTALL_PREFIX] all -j 8 --config threads_build=8
build_generator=Ninja
```

In case your **bin** directory is not listed in the PATH variable (echo $PATH), execute

```
python3 scripts/setup_path.py /path/to/INSTALL_PREFIX/bin
```

to make Nanopype aware of the installed tools. This will create a .pth file in your python3 installation, modifying the PATH temporarily on python startup. Alternatively you can run the following line once or append it to your ~/.bashrc. Note that in cluster environments this is not necessarily changing the PATH on all nodes!

```
export PATH=/path/to/INSTALL_PREFIX/bin:$PATH
```

To deactivate a virtual python environment after installation or usage of the pipeline just type:

```
deactivate
```

for plain and

```
source deactivate
```

for conda virtual environments.

Mission accomplished! Everything else is solved at run time by Snakemake and Nanopype.

> ⚠ **Configuration**
>
> Do not forget to follow the global configuration once and for each sample the local one.

Troubleshooting

There are some common errors that could arise during the installation process. If you encounter one of the following error messages, please consider the respective solution attempts.

not a supported wheel on this platform

> Nanopype requires at least python3.4 (The Docker image uses python3.5). If you install additional packages (e.g. albacore) from python wheels, make sure the downloaded binary packages matches the local python version.

terminated by signal 4

> Nanopype is mostly compiling integrated tools from source. In heterogeneous cluster environments this can lead to errors if the compilation takes place on a machine supporting modern vector instructions (SSE, AVX, etc.) but execution also uses less recent computers. The error message *terminated by signal 4* indicates an instruction in the software not supported by the underlying hardware. Please re-compile and install the tools from a machine with a common subset of vector instructions in this case.

Docker

The all-in-one Docker image of Nanopype contains all its dependencies and is build automatically on dockerHub. This is primarily meant for local usage and does currently not support Snakemake's cluster engine. The compressed docker image size is ~500 MB. From the Docker shell run:

```
docker pull giesselmann/nanopype
```

The container is tagged with the pipeline version. To obtain a specific version run e.g.:

```
docker pull giesselmann/nanopype:v0.4.0
```

Docker images can get access to the hosts file system with the *bind* argument. Test the container by mounting your current directory:

```
docker run -it --mount type=bind,source=$(pwd),target=/host
giesselmann/nanopype
```

Inside of the container type

```
ls -l /host
```

to see the files of the host system from where the container was started. Leave a running container with *exit*.

Changes made to the file system of the container are not persistent. For configuration purpose we clone the pipeline repository and map it later into the container.

```
mkdir -p src && cd src
git clone --recursive https://github.com/giesselmann/nanopype
cd nanopype
```

To use Docker for processing you will need to mount the pipeline, your data and a processing directory to the container. Any processing results not copied to the host will not persist inside the container after leaving it! From the Nanopype repository run:

```
docker run -it --mount type=bind,source=$(pwd),target=/app \
--mount type=bind,source=/path/to/miniondata,target=/data \
--mount type=bind,source=/path/to/project,target=/processing
giesselmann/nanopype
```

Configuration

Nanopype has two configuration layers: The central **environment** configuration *env.yaml* covers application paths and reference genomes and is set up independent of installation method and operating system once. The environment configuration is stored in the installation directory. If a compute cluster is available the respective Snakemake configuration is only needed once per Nanopype installation and explained here by the example of a custom scheduler called *mxq*.

For a project an additional **workflow** configuration is required providing data sources, tool flags and parameters. The workflow config file *nanopype.yaml* is expected in the root of each project directory. Configuration files are in .yaml format.

Environment

The default environment configuration **env.yaml** in the pipeline repository assumes all required tools to be available through your systems PATH variable. To use Nanopype with existing installations you may need to adjust their paths. Each tools is configured as key:value pair of tool name and the executable.

```
bin:
    # executable is in PATH variable
    albacore: read_fast5_basecaller.py
    # executable is not in PATH variable
    bedtools: /src/bedtools2/bin/bedtools
```

In order to execute workflows containing alignment steps, the respective reference genomes need to be set up. Each reference is provided as a key (e.g. mm9, mm10, hg19, ..), the genome in fasta format and optionally a chromosome size file if using tools such as *bedGraphToBigWig*.

```
references:
    mm9:
```

```
        genome: /path/to/mm9/mm9.fa
        chr_sizes: /path/to/mm9/mm9.chrom.sizes
```

example env.yaml

```
references:
    mm9:
        genome: ~/references/mm9/mm9.fa
        chr_sizes: ~/references/mm9/mm9.chrom.sizes
    mm10:
        genome: ~/references/mm10/mm10.fa
        chr_sizes: ~/references/mm10/mm10.chrom.sizes
    hg19:
        genome: ~/references/hg19/hg19.fa
        chr_sizes: ~/references/hg19/hg19.chrom.sizes
    hg38:
        genome: ~/references/hg38/hg38.fa
        chr_sizes: ~/references/hg38/hg38.chrom.sizes

bin:
    albacore: ~/bin/read_fast5_basecaller.py
    flappie: ~/bin/flappie
    guppy: ~/bin/guppy_basecaller
    bedtools: ~/bin/bedtools
    graphmap: ~/bin/graphmap
    minimap2: ~/bin/minimap2
    nanopolish: ~/bin/nanopolish
    ngmlr: ~/bin/ngmlr
    samtools: ~/bin/samtools
    sniffles: ~/bin/sniffles
    deepbinner: ~/bin/deepbinner-runner.py
    bedGraphToBigWig: ~/bin/bedGraphToBigWig
```

Additional references possibly only needed once in a **workflow** can be configured in the *nanopype.yaml* of the working directory.

Profiles

Snakemake supports the creation of profiles to store flags and options for a particular environment. A profile for a workflow is loaded from the Snakemake command line via:

```
snakemake --profile myprofile ...
```

searching globally (*/etc/xdg/snakemake*) locally (*$HOME/.config/snakemake*) for folders with name *myprofile*. Theese folders are expected to contain a

config.yaml with any of Snakemakes command line parameters. An example profile is given in *nanopype/profiles/mxq* which can be used with

```
snakemake --profile /path/to/nanopype/profiles/mxq ...
```

Additional profiles can be found at https://github.com/snakemake-profiles/doc.

example config.yaml

```
# per node local storage
shadow-prefix: "/scratch/local2/"
# jobscripts
cluster: "mxq-submit.py"
cluster-status: "mxq-status.py"
jobscript: "mxq-jobscript.sh"
# params
latency-wait: 600
jobs: 2048
restart-times: 3
immediate-submit: false
```

Cluster configuration

The cluster configuration of Snakemake is separated from the workflow and can be provided in .json or .yaml format. The configuration is composed of default settings mapping to all rules (e.g. basecalling, alignment, etc.) and rule specific settings enabling a fine grained control of resources such as memory and run time.

example cluster.json for MXQ

```
{
    "__default__" :
    {
        "nCPUs"     : "8",
        "memory"    : "20000",
        "time"      : "60"
    },

    "minimap2" :
    {
        "memory"    : "30000",
        "time"      : "30"
        "nCPUs"     : "16"
```

```
    }
}
```

Rule names of Nanopype can be obtained by starting either a dryrun (-n) or directly from the source. Parameters of the cluster config are accessible inside the cluster submission command:

```
snakemake --cluster-config cluster.json --cluster "mxqsub --
threads={cluster.nCPUs} --memory={cluster.memory} --
runtime={cluster.time}"
```

This cluster configuration is static per target and needs to be conservative enough to handle any type of input data. For a more fine grained control Nanopype specifies per rule a configurable number of threads and calculates the estimated run time and memory consumption accordingly. Integration and customization of these parameters is described in the following section.

Job properties

All Nanopype workflows specify **threads** and **time_min** and **mem_mb** resources. Furthermore runtime and memory are dynamically increased if a clsuter job fails or is killed by the cluster management (Due to runtime or memory violation). To restart jobs Snakemake needs to be executed with e.g. --restart-times 3. If supported by the cluster engine you could then use:

```
snakemake --cluster "mxqsub --threads={threads} --
runtime={resources.time_min} --memory={resources.mem_mb}"
```

Please note that the *threads* resource in this example is now not set per rule as in the previous example, but configured per module in the workflow config file *nanopype.yaml* in the working directory.

If the formats of estimated runtime and memory consumption do not match your cluster system a custom wrapper script is easily set up. To convert from time in minutes to a custom string the following snippet is a starting point:

example cluster_wrapper.py format conversion

```
import os, sys
from snakemake.utils import read_job_properties

jobscript = sys.argv[-1]
job_properties = read_job_properties(jobscript)

# default resources
threads = '1'
runtime = '01:00'
memory = '16000M'
# parse resources
if "threads" in job_properties:
    threads = str(job_properties["threads"])
if "resources" in job_properties:
    resources = job_properties["resources"]
    if "mem_mb" in resources: memory = str(resources["mem_mb"]) + 'M'
    if "time_min" in resources: runtime = "{:02d}:{:
02d}".format(*divmod(resources["time_min"], 60))

# cmd and submission
cmd = 'mxqsub --threads={threads} --memory={memory} --
time="{runtime}" {jobscript}'.format(threads=threads, memory=memory,
runtime=runtime, jobscript=jobscript)
os.system(cmd)
```

The respective Snakemake command line would then be:

```
snakemake --cluster cluster_wrapper.py
```

resulting in a cluster submission of the temporary job script on the shell similar to:

```
mxqsub --threads=1 --memory=16000M --time="00:15" ./snakemake/
tmp123/job_xy.sh
```

Custom cluster integration

A full example on how to enable a not yet supported cluster system is given in *profiles/mxq/* of the git repository. Briefly four components are required:

- config.yaml - cluster specific command line arguments
- submit.py - job submission to cluster management
- status.py - job status request from cluster management

- jobscript.sh - wrapper script for rules to be executed

Important: When working with batches of raw nanopore reads, Nanopype makes use of Snakemakes shadow mechanism. A shadow directory is temporary per rule execution and can be placed on per node local storage to reduce network overhead. The shadow prefix e.g. */scratch/local/* can be set in the profiles config.yaml. The *--profile* argument tells Snakemake to use a specific profile:

```
snakemake --profile /path/nanopype/profiles/mxq [OPTIONS...] [FILE]
```

When running in an environment with **multiple users** the shadow prefix should contain a user specific part to ensure accessibility and to prevent corrupted data. A profile per user is compelling in this case and enforced by Nanopype if not given.

Logging

Writing log files becomes useful to inspect the output of potentially failing jobs. The actual implementation is system and user specific, two possible scenarios are given here:

Log per output file

Some cluster schedulers support the redirection of stdout and stderr. These could be caught using the cluster configuration of Snakemake:

example cluster.json for MXQ logging

```
{
    "__default__" :
    {
        "output"    : "logs/cluster/{rule}.{wildcards}.out",
        "error"     : "logs/cluster/{rule}.{wildcards}.err"
    }
}
```

Parameters of the cluster config are again accessible inside the cluster submission command:

```
snakemake --cluster-config cluster.json --cluster "mxqsub --
stdout={cluster.output} --stderr={cluster.error}"
```

This type of logging will create one log file per output file of Nanopype. Restarting workflows will overwrite previous logging.

Log per job submission

Logging can also be set up independent of Snakemake by modifying the job script itself.

example jobscript.sh for MXQ logging

```
H=${{HOSTNAME}}
J=${{MXQ_JOBID:-${{PID}}}}
DATE=`date +%Y%m%d`

mkdir -p log
({exec_job}) &> log/${{DATE}}_${{H}}_${{J}}.log
```

Snakemake will replace the *{exec_job}* wildcard with the temporary job script (Double curly brackets are needed for generic environment variables to escape the Snakemake substitution). Note that the environment variables *MXQ_JOBID* and *PID* are specific to the mxq scheduler. The above wrapper will create log files with timestamp and hostname in the filename without overwriting previous logs.

The custom job script is included to the Snakemake command line by using:

```
snakemake --jobscript jobscript.sh --cluster "mxqsub ..."
```

Tests

After installation and environment configuration the successful setup can be tested with sample data included in the pipeline repository.

An initial test validates the presence of all required tools. This is only required for source installations and to check if manual e.g. albacore installations are found correctly. The test will check all supported tools, if you do not plan to use parts of the pipeline, you can ignore failing test cases.

```
python3 test/test_install.py
```

Secondly the functionality of the pipeline itself is tested on a small sample data set. The command is slightly different depending on the installation method:

Singularity

```
cd /path/to/nanopype
python3 test/test_rules.py test_unit_singularity.<CASE>
```

Source

```
cd /path/to/nanopype
python3 test/test_rules.py test_unit_src.<CASE>
```

Docker

```
docker run -it giesselmann/nanopype
python3 /app/test/test_rules.py test_unit_src.<CASE>
```

Available test cases are:

Storage

- test_storage

Basecalling

- test_guppy
- test_albacore
- test_flappie

Alignment

- test_minimap2
- test_graphmap
- test_ngmlr

Methylation

- test_nanopolish

Structural Variation

- test_sniffles

To run all tests call the script without any test case e.g.:

```
python3 test/test_rules.py test_unit_src
```

The tests takes ~20 min on 4 cores and downloads ~54 MB reference sequence for the alignment module. Tests cover all modules of the pipeline, if some tools (e.g. albacore in the Docker container) are not installed, the associated tests will fail. Independent parts of the pipeline will however still work. Note that test run times are not representative due to very small batch sizes.

Usage

Nanopype is based on the three key components binaries, raw data handling and processing. After finishing the installation and environment configuration, the pipeline is ready to process nanopore data sets. If you use Nanopype for the first time you might consider following the tutorial first.

The default workflow configuration **nanopype.yaml** in the installation directory of Nanopype serves as a template and needs to be copied to each working directory. The file is required in each processing directory and parsed in every invocation of the pipeline. The meaning of tool specific parameters is documented in the Modules section.

> **Caution**
>
> The parameters in the config file have significant impact on the computed output. In contrast to the incorporated tools the configuration is not controlled or versioned by Nanopype. Changing entries without re-running the pipeline will lead to inconsistency between config and processed data.

Snakemake basics

The basic concept of Snakemake and thus also Nanopype is to request an output file which can be generated from one or more input files. Snakemake will recursively build a graph and determine required intermediate or temporary data to be processed. In general you can run Snakemake from within your processing directory with

```
snakemake --snakefile /path/to/nanopype/Snakefile [OPTIONS...]
[FILE]
```

or from the installation path with specifying the working directory:

```
snakemake --directory /path/to/processing [OPTIONS...] [FILE]
```

Nanopype expects a **nanopype.yaml** in the root of the processing directory. A template with default values is found in the Nanopype repository. You can either edit the parameters within the file or override them from the command line:

```
snakemake [OPTIONS...] [FILE] --config threads_basecalling=16
```

For all available snakemake command line options please also refer to the snakemake documentation. Particularly interesting are:

- -n / --dryrun : Print required steps but do not execute, useful to check before e.g. cluster submissions
- -j / --cores / --jobs : In local mode: number of provided cores; In cluster mode: number of jobs submitted at the same time
- -k / --keep-going : Continue execution of independent jobs, in case of errors

Singularity

Using Nanopype with the Singularity installation method is recommended. Environment and workflow configuration remain the same as above, the Snakemake command line needs to be extended to:

```
snakemake --snakefile /path/to/nanopype/Snakefile --use-
singularity [OPTIONS...] [FILE]
```

Nanopype is detecting its current version from the cloned repository and will fetch the respective Singularity containers. Per default these images are stored in the hidden *.snakemake* folder in the current working directory. For multiple parallel workflows the flag *--singularity-prefix* can be set to fetch and save images only once. Setting the prefix in a profile along with other static flags is our suggestion.

Raw data archives

Raw nanopore read data is organized in directories per flow cell. Their name can be used to encode important experimental information such as flow cell type, kit

and IDs. The name **must** at least contain the flow cell type and the sequencing kit which are used by the basecalling module. Furthermore a *reads* sub folder containing batches of raw nanopore reads in *.tar* or bulk-*.fast5* format is required.

> **raw data archive formats**
>
> In previous versions of MinKNOW each read was stored into a single .fast5 file. A large number of single files leads especially on Linux like systems to directories which are difficult to copy, search and handle in general. To address this issue Nanopype provides an import tool in *scripts/nanopype_import.py* to package batches of e.g. 4000 .fast5 files in tar-archives. Current versions of MinKNOW do fully utilize the capabilities of the underlying hdf5 format and write a configurable number or reads into a bulk-fast5 file. Nanopype can operate on both, packaged single reads and bulk format. Furthermore algorithms requiring fast5 access can be fed with single read fast5 files to maintain backwards compatibility.

A suggested naming pattern for per flow cell directories is:

20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01

with fixed first four fields separated by underscores and an arbitrary number of user defined tags at the end. Parsing of the run names can be configured in the nanopype.yaml configuration file. The settings for the suggested patterns are (indices are zero-based):

```
storage_runname:
    delimiter: '_'
    field_ID: 1
    field_flowcell: 2
    field_kit: 3
```

Copying the **nanopype.yaml** from the pipeline repository into the processing directory is the first step before further local workflow configuration and processing.

Workflow config

The entry *storage_data_raw* is the common base path of the previously described data directories per flow cell. It can be a relative or absolute path and must for

cluster usage be accessible from all computing nodes. With *threads_[module]* the number of threads used per batch and specific module can be controlled. The overall number of available threads is set at run time.

Nanopype supports merging of multiple flow cells into a single track. A **runnames.txt** in the processing directory with one raw data folder name per line is required in that case. Lines starting with # are treated as comments, empty lines are ignored.

Whereas the following command would trigger basecalling of a single flow cell:

```
snakemake --snakefile /path/to/nanopype/Snakefile sequences/guppy/
WA01/batches/20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01.fastq.gz
```

the next will trigger basecalling for all runs listed in *runnames.txt* and merge the results in the end:

```
snakemake --snakefile /path/to/nanopype/Snakefile sequences/guppy/
WA01.fastq.gz
```

The name of the output file is called a **tag** and is in this case is arbitrary. Special tags exist for barcoded runs as described in the demultiplexing module. The extension must be either fasta.gz or fastq.gz.

Processing examples

If available Nanopype tries to implement and provide multiple, possibly competing tools for the same task. The alignment module for instance allows currently to select from three different long read aligners namely minimap2, graphmap and ngmlr. A key design of Nanopype is that the executed pipeline is reflected in the structure of the output directory. Taking the alignment example, exchanging the aligner in a workflow becomes as simple as:

```
snakemake --snakefile /path/to/nanopype/Snakefile alignments/
minimap2/guppy/WA01.hg38.bam -j16
snakemake --snakefile /path/to/nanopype/Snakefile alignments/
graphmap/guppy/WA01.hg38.bam -j16
snakemake --snakefile /path/to/nanopype/Snakefile alignments/ngmlr/
guppy/WA01.hg38.bam -j16
```

All three commands will make use of the same basecalling results but create different .bam files.

Temporary data

In the processing stage Nanopype can create a significant amount of temporary data which is not automatically cleaned up and caused by the batch processing approach. As an example, the merged alignment of multiple runs will create the following output directory structure:

```
|--alignments/
   |--minimap2/                                                #
Minimap2 alignment
      |--guppy/                                                #
Using guppy basecalling
         |--batches
            |--20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01/
               |--0.hg38.bam
               |--1.hg38.bam
               ...
            |--20180101_FAH12345_FLO-MIN106_SQK-
LSK108_WA01.hg38.bam
         |--WA01.hg38.bam
```

Aside from the final output file *WA01.hg38.bam* the individual .bam files per run and the set of batch alignment files in the *batches* sub folder contain the same information. Since other modules of the pipeline might later require the batch alignment output as input files, the deletion of the intermediate results is left to the user.

Nanopype provides a set of cleanup rules to delete the batch processing output. Per module these rules delete everything inside of the **batches** folder. The

cleanup is invoked by the rule name from within the working directory for e.g. the basecalling module:

```
snakemake --snakefile /path/to/nanopype/Snakefile sequences_clean
```

> **Caution**
>
> Deleting the batch output of a module impacts the execution of other workflows as well and should be used with caution at the end of an analysis workflow. Running for instance a structural variation detection requires a merged .bam file. Deleting the batch output of the alignment module afterwards would lead in a later processing of e.g. methylation rates to the reprocessing of the alignment in batches.

The full list of available cleaning rules is given below:

```
* sequences_clean
* alignment_clean
* methylation_clean
* sv_clean
* demux_clean
* transcript_isoforms_clean
```

Cluster usage

Snakemake allows both, cloud and cluster execution of workflows. As of now only cluster execution is implemented and tested in Nanopype. The configuration section covers multiple scenarios of integrating supported and custom cluster management systems into the Nanopype workflow. For supported cluster engines please also refer to the Snakemake documentation.

Following the steps in the general workflow documentation, the usage of a cluster backend requires only a few additional flags and arguments in the Snakemake command. The use of profiles is therefore recommended. The following command line options of Snakmake are of particular interest:

--profile

> Path to profile directory with command line arguments and flags. A *config.yaml* is expected to be found in the profile folder

-n or --dryrun

> Do not execute anything, just display what would be done.

-k or --keep-going

> Continue executing independent jobs, even if an error occured.

--shadow-prefix

> Directory to write per rule temporary data to. Some tools working with raw fast5 files need to unpack these creating a large number of temporary files. These rules are set up to use the shadow mechanism of Snakemake. In order to reduce network and fileserver workload the shadow directory can be placed on per node local storage (e.g. /scratch/tmp, /scratch/local, etc). The path should be the same on all nodes but point to a local harddrive on the respective server

-w or --latency-wait

Time in seconds to wait for output files. On NFS mounted directories in server environments, the output of a rule on a compute node can take a while to appear in the filesystem of the head node.

--use-singularity and --singularity-prefix

Indicate to use the Singularity implementation of Nanopype. The Singularity prefix can be set to a common path for all processed samples, avoiding multiple downloads of the same images. Singularity images are bound to the version of the pipeline, an update of Nanopype will fetch the most recent ones automatically.

-j or --jobs or --core

In cluster mode the maximum number of jobs submitted to the queue.

Storage

The storage module of Nanopype covers the import, indexing and extraction of raw nanopore reads. In general the raw data directory is expected to have one folder per flow cell with a subfolder *reads* containing the packed or bulk *.fast5* files.

The folder structure for packeaged single read fast5 files is:

```
|--/data/raw/
   |--20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01/   # One flow
cell
      |--reads/
         |--0.tar                                     #
Packed .fast5
         |--1.tar
          ...
         |--reads.fofn                                # Index file
```

For bulk-fast5 output from recent MinKNOW versions, the batches can be directly copied to the reads folder.

```
|--/data/raw/
   |--20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01/   # One flow
cell
      |--reads/
         |--batch_0.fast5                             # Bulk-fast5
         |--batch_1.fast5
          ...
         |--reads.fofn                                # Index file
```

Nanopype expects all batches to be found in the *reads* folder of a run. Restarting an experiment in MinKNOW results in a new raw output folder with batch numbers starting from zero. In current versions of MinKNOW a unique run-ID is part of the batch name, therefore bulk-fast5 files from multiple restarts can be copied into the same directory. After updating MinKNOW the output naming should be verified to avoid overwriting batches with equal names.

Import

This section is for backwards compatibility with MinKNOW versions writing each read into a single .fast5 file. Recent versions create bulk-fast5 output which can be directly used with Nanopype

To pack reads e.g. from the MinKNOW output folder we provide an import script *nanopype_import.py* in the scripts folder of the repository. The basic usage is:

```
python3 scripts/nanopype_import.py /data/raw/runname/ /path/to/
import
```

You can specify one or more import directories, also by using wildcards in the path. This is useful after restarting an experiment and importing every folder containing a specific flow cell ID. Consider changing the batch size in case of amplicon or RNA sequencing with significantly more but in general shorter reads. The order of reads in the archives is **not** guaranteed to be the same as in the output folders of MinKNOW. Running the script with the same arguments twice will validate the import process and report any inconsistency between import and raw data directories.

Indexing

An index file *reads.fofn* with one line per read containing the ID and the archive the read is stored in is helpful if later only a subset of the whole dataset needs to be processed. The indexing is triggered from the processing directory by executing e.g.:

```
snakemake --snakefile /path/to/nanopype/Snakefile /data/raw/
20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01/reads.fofn
```

Together with the import, this is the only rule requiring **write access** to the raw data. We highly recommend, running it once after the experiment and making the run folder write protected afterwards with e.g.:

```
run=20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01
chmod 444 /data/raw/$run/reads/*
chmod 444 /data/raw/$run/reads.fofn
chmod 555 /data/raw/$run/reads
chmod 555 /data/raw/$run
```

Tip

Internally we use an isolated unix-user and group *mduser* and *mdgrp* owning the raw data. Setting permissions to e.g. 744 for the batch files and 755 for folders allows any analyst to securely read the raw data without accidentally compromising it.

Extraction

It is possible to extract a **subset** of fast5 files from the packed and indexed run. Extraction requires a previously indexed run, a list of read IDs and works by requesting a directory from Nanopype:

```
snakemake --snakefile /path/to/nanopype/Snakefile subset/roi/
20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01
```

With this command the extraction rule expects an input file *subset/roi.txt* with one read ID per line. Multiple regions of interest are supported by multiple ID files.

```
b4782c09-6edc-4a1e-aad9-ba7d740723bc
1ab15b74-65a1-42d0-9ff6-1d7276c30e46
5d18f246-e96b-43ff-aea7-4b55029929ef
...
```

To extract reads covering a region of interest from multiple runs, provide a *runnames.txt* in the processing directory with one line per sequencing run:

```
20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01
20180202_FAH67890_FLO-MIN106_SQK-LSK108_WA01
```

And execute:

```
snakemake --snakefile /path/to/nanopype/Snakefile subset/roi.done
```

For this rule a flag file indicating completion is required since the exact output is unknown and the output directory might already be existent.

Basecalling

The basecaller translates the raw electrical signal from the sequencer into a nucleotide sequence in fastq or fasta format. As input the packed **fast5** files as provided by the storage module are required.

In order to process the output of one flow cell with the basecaller *albacore* run from within your processing directory:

```
snakemake --snakefile /path/to/nanopype/Snakefile sequences/
albacore/WA01/batches/20180101_FAH12345_FLO-MIN106_SQK-
LSK108_WA01.fastq.gz
```

Valid file extensions are fasta/fa, fastq/fq in gzipped form. Providing a *runnames.txt* with one run name per line it is possible to process multiple flow cells at once and merge the output into a single file e.g.:

```
snakemake --snakefile /path/to/nanopype/Snakefile sequences/
albacore/WA01.fastq.gz
```

The tag *WA01* is arbitrary and may describe a corresponding experiment or cell line. Furthermore a basic quality control of the flow cell can be obtained by running:

```
snakemake --snakefile /path/to/nanopype/Snakefile sequences/
albacore/WA01.fastq.pdf
```

The content of the QC depends on the basecaller and format. The sequence quality for instance is only stored in fastq format.

Folder structure

The basecalling module can create the following file structure relative to the working directory:

```
|--sequences/
   |--albacore/                                                    #
Albacore basecaller
      |--batches/
         |--WA01/
            |--20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01/
               |--0.fastq.gz
# Sequence batches
               |--1.fastq.gz
                ...
            |--20180101_FAH12345_FLO-MIN106_SQK-
LSK108_WA01.fastq.gz
            |--20180101_FAH12345_FLO-MIN106_SQK-
LSK108_WA01.fastq.pdf
      |--WA01.fastq.gz
      |--WA01.fastq.pdf
   |--guppy/                                                       #
Guppy basecaller
      |--...
```

Cleanup

The batch processing output of the basecalling module can be cleaned up by running:

```
snakemake --snakefile /path/to/nanopype/Snakefile sequences_clean
```

This will delete all files and folders inside of any **batches** directory and should only be used at the very end of an analysis workflow. The alignment module for instance relies on the single batch sequence files.

Tools

Depending on the application you can choose from one of the following basecallers, listed with their associated configuration options.

- threads_basecalling: 4

Albacore

The ONT closed source software based on a deep neural network. The installer is accessible after login to the community board.

- basecalling_albacore_barcoding: false
- basecalling_albacore_disable_filtering: true
- basecalling_albacore_flags: ''

Guppy

Pre-Release within ONT-community.

- basecalling_guppy_qscore_filter: 0
- basecalling_guppy_flags: ''

Flappie

The experimental neural network caller from ONT using flip-flop basecalling.

- basecalling_flappie_model: 'r941_5mC'
- basecalling_flappie_flags: ''

Alignment

The alignment step maps basecalled reads against a reference genome. If not already done alignment rules will trigger basecalling of the raw nanopore reads.

To align the reads of one flow cell against reference genome hg38 with *guppy* basecalling and aligner *minimap2* run from within your processing directory:

```
snakemake --snakefile /path/to/nanopype/Snakefile alignments/
minimap2/guppy/WA01/batches/20180101_FAH12345_FLO-MIN106_SQK-
LSK108_WA01.hg38.bam
```

This requires an entry *hg38* in your **env.yaml** in the Nanopype installation directory:

```
references:
    hg38:
        genome: /path/to/references/hg38/hg38.fa
        chr_sizes: /path/to/references/hg38/hg38.chrom.sizes
```

Providing a *runnames.txt* with one runname per line it is possible to process multiple flow cells at once and merge the output into a single track e.g.:

```
snakemake --snakefile /path/to/nanopype/Snakefile alignments/
minimap2/guppy/WA01.hg38.bam
```

Some aligners require an indexed reference genome, Nanopype will automatically build one uppon first use. An alignment job is always connected with a sam2bam conversion and sorting in an additional thread. The above commands with 3 configured alignment threads will therefore fail if Snakemake is invoked with less than 4 threads (at least *-j 4* is required). The default input sequence format for alignment rules is *fastq*, yet if possible, the alignment module will use already basecalled batches in the sequences/[basecaller]/[runname]/ folder of the working directory.

Folder structure

The alignment module can create the following file structure relative to the working directory:

```
|--alignments/
   |--minimap2/                                                  #
Minimap2 alignment
      |--albacore/                                               #
Using albacore basecalling
         |--batches/
            |--WA01/
               |--20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01/
                  |--0.hg38.bam
                  |--1.hg38.bam
                  ...
               |--20180101_FAH12345_FLO-MIN106_SQK-
LSK108_WA01.hg38.bam
         |--WA01.hg38.bam
   |--graphmap/                                                  #
GraphMap alignment
      |--guppy/                                                  #
Using guppy basecalling
         |--batches/
            |--WA01/
               |--20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01/
                  |--0.hg19.bam
                  ...
               |--20180101_FAH12345_FLO-MIN106_SQK-
LSK108_WA01.hg19.bam
         |--WA01.hg19.bam
```

Cleanup

The batch processing output of the alignment module can be cleaned up by running:

```
snakemake --snakefile /path/to/nanopype/Snakefile alignment_clean
```

This will delete all files and folders inside of any **batches** directory and should only be used at the very end of an analysis workflow. The methylation module for instance relies on the single batch alignment files.

Tools

Depending on the application you can choose from one of the listed aligners. All alignment rules share a set of global config variables:

- threads_alignment: 3

Minimap2

Pairwise alignment for nucleotide sequences. Any given command line arguments are directly passed to the aligner:

- alignment_minimap2_flags: '-ax map-ont -L'

GraphMap

Fast and sensitive mapping of nanopore sequencing reads with GraphMap. Any given command line arguments are directly passed to the aligner:

- alignment_graphmap_flags: '-B 100'

NGMLR

Accurate detection of complex structural variations using single-molecule sequencing. Any given command line arguments are directly passed to the aligner:

- alignment_ngmlr_flags: '-x ont --bam-fix'

References

Li, H. Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics. doi:10.1093/bioinformatics/bty191 (2018).

Sovic, I. et al. Fast and sensitive mapping of nanopore sequencing reads with GraphMap. Nat. Commun. 7:11307 doi: 10.1038/ncomms11307 (2016).

Sedlazeck, F. J. et al. Accurate detection of complex structural variations using single-molecule sequencing. Nature Methods 15, 461-468 (2018).

Methylation

Nanopore sequencing enables the simultaneous readout of sequence and base modification status on single molecule level. Nanopype currently integrates nanopolish and flappie to detect 5mC DNA methylation. To obtain the methylation readout of multiple flow cells using nanopolish run:

```
snakemake --snakefile /path/to/nanopype/Snakefile methylation/
nanopolish/ngmlr/albacore/WA01.5x.hg38.bedGraph
```

or

```
snakemake --snakefile /path/to/nanopype/Snakefile methylation/
nanopolish/ngmlr/albacore/WA01.5x.hg38.bw
```

If not already present, this will trigger basecalling and alignment of the raw data. Similar to the basecalling and alignment modules a *runnames.txt* in the working directory is required. Furthermore only CpGs with at least 5x coverage are reported. The coverage wildcard can be any positive integer in combination with arbitrary text (e.g. *5*, *5x* or *min5x* will result in the same output).

Folder structure

```
|--methylation/
   |--nanopolish/
# Nanopolish
      |--ngmlr/
# NGMLR alignment
         |--guppy/
# Guppy sequences
            |--batches/
               |--WA01/
                  |--20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01/
                     |--0.hg38.tsv
# Single read batches
                     |--1.hg38.tsv
                      ...
                  |--20180101_FAH12345_FLO-MIN106_SQK-
```

```
LSK108_WA01.hg38.tsv.gz
            |--WA01.1x.hg38.bedGraph
# Mean methylation level
            |--WA01.1x.hg38.bw
   |--flappie/
# Flappie
      |--ngmlr/
         |--flappie/
            |--batches/
               |--WA01/
                  |--20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01/
                     |--0.hg38.tsv
# Single read batches
                     |--1.hg38.tsv
                  |--20180101_FAH12345_FLO-MIN106_SQK-
LSK108_WA01.hg38.tsv.gz
            |--WA01.1x.hg38.bedGraph
            |--WA01.1x.hg38.bw
```

Cleanup

The batch processing output of the methylation module can be cleaned up by running:

```
snakemake --snakefile /path/to/nanopype/Snakefile methylation_clean
```

This will delete all files and folders inside of any **batches** directory and should only be used at the very end of an analysis workflow.

Tools

The output of nanopore methylation calling tools on a lower level is not consistent. Here we provide the description of the intermediate results, enabling more advanced analysis. All methylation rules share a set of global config variables:

- threads_methylation: 4

Nanopolish

The bed-like intermediate output file contains per read and CpG methylation log-likelihood ratios as well as the raw log-p values from the nanopolish HMM.

```
chr6    31164635    31164635
2975d04a-9890-43cd-880b-28a1d72b0f19    2.31    +   -146.60 -148.91
```

A log-likelihood ratio greater 2.5 is considered a methylated CpG, a ratio less than -2.5 is called unmethylated. To get genome wide mean methylation levels in bedGraph or BigWig format you can use:

To directly get the single read methylation levels of a single flow cell you can run:

```
snakemake --snakefile /path/to/nanopype/Snakefile methylation/
nanopolish/ngmlr/guppy/batches/WA01/20180101_FAH12345_FLO-
MIN106_SQK-LSK108_WA01.hg38.tsv.gz
```

Nanopolish specific configuration parameters are:

- methylation_nanopolish_logp_threshold: 2.5

Flappie

Flappie is a new ONT neural network basecaller with a default output alphabet of *ACGT*. Using a different model, a methylation status prediction in CpG contexts is possible. A methylated Cytosine is then encoded as *Z*.

The Flappie sequence output is aligned against a reference genome. A CG in the reference with matching CG in the read is reported as unmethylated, a CG in the reference with matching ZG is reported as methylated. The single read output is again a bed-like file with the methylation level in the 4^{th} column and the mean q-val over the CG in the 6^{th} column:

```
chr6    31129299    31129301    f56afc41-fc2c-4157-941d-
b8416eb11d2c    0   +   5.0
```

Flappie specific configuration parameters are:

- basecalling_flappie_model: 'r941_5mC'
- methylation_flappie_qval_threshold: 3

References

> Simpson, Jared T. et al. Detecting DNA cytosine methylation using nanopore sequencing. Nature Methods 14.4, 407-410 (2017).

Transcriptome

Another application of the long-read nanopore technology is sequencing of cDNA and RNA molecules directly. Recovery of full-length transcripts enables, for instance, the detection of alternatively spliced isoforms and is implemented in Nanopype using the Pinfish package. The output of polished transcripts is provided in the GFF format.

Isoform detection

The isoform detection in Nanopype is oriented to meet the implementation of the pinfish-pipeline from ONT, extended by the basecalling and alignment back-end of Nanopype. To align the reads of one flow cell against reference genome hg38 with *guppy* basecalling and aligner *minimap2* and detect isoforms using pinfish run from within your processing directory:

```
snakemake --snakefile /path/to/nanopype/Snakefile
transcript_isoforms/pinfish/minimap2/guppy/batches/
WA01/20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01.hg38.gff
```

This requires an entry *hg38* in your **env.yaml** in the Nanopype installation directory:

```
references:
    hg38:
        genome: /path/to/references/hg38/hg38.fa
        chr_sizes: /path/to/references/hg38/hg38.chrom.sizes
```

Providing a *runnames.txt* with one runname per line it is possible to process multiple flow cells at once and merge the output into a single track e.g.:

```
snakemake --snakefile /path/to/nanopype/Snakefile
transcript_isoforms/pinfish/minimap2/guppy/WA01.hg38.gff
```

Folder structure

The transcript module can create the following example file structure relative to the working directory:

```
|--transcript_isoforms
   |--pinfish
     |--minimap2/
# Minimap2 alignment
        |--guppy/
# Using guppy basecalling
           |--batches/
              |--WA01/
                 |--20180101_FAH12345_FLO-MIN106_SQK-
LSK108_WA01.hg38.gff
           |--WA01.hg38.gff
```

Alignment

The transcript module of Nanopype is generic in terms of used basecaller and aligner. However spliced alignments require a different configuration and are not supported by each incorporated aligner. Nanopype's default configuration values are tuned towards processing of long DNA reads. Please observe the following tools section and set the respective configuration flags in the **nanopype.yaml** in the processing directory. Additionally the samtools flags as described below are crucial for the success of the *polish_clusters* past of pinfish.

Tools

All transcript module rules share a common set of configuration options. The thread option is split for rules processing only batches of reads and rules working on the entire dataset. The latter is expected to profit from more provided cores.

- threads_transcript_batch: 4
- threads_transcript: 4

Minimap2

Minimap2 is currently the only incorporated aligner supporting spliced alignments. The program flags depend on the input library. Please also refer to the Minimap2 repository for detailed documentation.

- alignment_samtools_flags: '-F 2304'
- alignment_minimap2_flags: '-ax splice'
- alignment_minimap2_flags: '-ax splice -uf' # for stranded data

Pinfish

Pinfish is an ONT package to generate isoform annotations from nanopore RNA sequencing.

- transcript_spliced_bam2gff_flags: ''
- transcript_cluster_gff_flags: '-c 10 -d 10 -e 30'
- transcript_collapse_partials: '-d 5 -f 5000 -e 30'
- transcript_polish_clusters: '-c 10'

References

> Li, H. Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics. doi:10.1093/bioinformatics/bty191 (2018).
>
> ONT: Pinfish. https://github.com/nanoporetech/pinfish (2019).

Structural Variations

Nanopore sequencing is especially suitable to resolve structural variations including insertions, deletions, duplications, inversions, and translocations. The output of the structural variations module of Nanopype is a vcf file for further downstream processing. To run e.g. Sniffles on the NGMLR alignment output of multiple flow cells specified in a *runnames.txt* execute:

```
snakemake --snakefile /path/to/nanopype/Snakefile sv/sniffles/
ngmlr/guppy/WA01.hg38.vcf
```

Folder structure

The structural variation module can create the following file structure relative to the working directory:

```
|--/sv/
   |--sniffles/
      |--ngmlr/
         |--guppy/
            |--WA01.hg38.vcf/
```

Cleanup

The structural variation module requires a merged alignment file of one or multiple flow cells. Since most of the pipeline is working with batches of reads, the merged files can be deleted to save disk space:

```
|--alignments/
   |--ngmlr/                                                    #
NGMLR alignment
      |--guppy/                                                 #
Using guppy basecalling
         |--batches/
            |--WA01/
               |--20180101_FAH12345_FLO-MIN106_SQK-
```

```
LSK108_WA01/             # keep

|--0.hg38.bam                                                  # keep

|--1.hg38.bam                                                  # keep
                  ...
                |--20180101_FAH12345_FLO-MIN106_SQK-
LSK108_WA01.hg38.bam     # delete
          |--
WA01.hg38.bam                                                      #
delete
```

Tools

The SV module includes the following tools:

Sniffles

Sniffles is reported to work best with the NGMLR aligner also included in this pipeline. It is therefore recommended but not mandatory to use the example command in connection with sniffles. You can further configure the tool by setting any of its command line flags:

- sv_sniffles_flags: '-s 10 -l 30 -r 2000 --genotype'

References

> Sedlazeck, F. J. et al. Accurate detection of complex structural variations using single-molecule sequencing. Nature Methods 15, 461-468 (2018).

Demultiplexing

With barcoded libraries nanopore sequencing allows to pool multiple samples on a single flow cell. Demultiplexing describes the classification of barcodes per read and the assignment of read groups. The output of the demultiplexing module is batches of read IDs belonging to a detected barcode. In order to process a barcoded flow cell with Deepbinner run:

```
snakemake --snakefile /path/to/nanopype/Snakefile demux/deepbinner/
20180101_FAH12345_FLO-MIN106_SQK-RBK004_WA01.tsv
```

Note the different sequencing kit *SQK-RBK004* used in this example. Aside from explicit demultiplexing the module supports mapping of previously introduced tags to barcodes. A **barcodes.yaml** in the working directory is detected by Nanopype and can map raw demux output to tags:

example barcodes.yaml

```
__default__:
    NB01 : '1'
    NB02 : '2'
20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01:
    NB01 : '3'
    NB02 : '4'
```

The file may contain a `__default__` mapping, applied to any not listed run as well as run specific mappings. Each mapping is a key:value pair of tag and barcode. For different runs, the same tag can map to different barcodes allowing the efficient usage of available barcodes in the sequencing kit. Without running the demultiplexing explicitly, an alignment of only one barcode can be obtained by running:

```
snakemake --snakefile /path/to/nanopype/Snakefile alignments/
minimap2/guppy/NB01.hg38.bam
```

Nanopype will detect *NB01* to be a special tag listed in *barcodes.yaml*, start demultiplexing, create batches of read IDs with barcode 1 and run basecalling and alignment only for this subset of reads.

Folder structure

The demultiplexing module can create the following file structure relative to the working directory:

```
|--demux/
   |--deepbinner/                                          #
Deepbinner neural network
      |--batches/
         |--20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01/
            |--0.tsv                                       #
classified batches
            |--1.tsv
           ...
      |--barcodes/
         |--20180101_FAH12345_FLO-MIN106_SQK-LSK108_WA01/
            |--1/                                          # raw
barcode for tag:barcode mapping
               |--0.tsv                                    # 4k
reads of barcode 1
               |--1.tsv
            |--2/
```

Tools

The demultiplexing module includes the following tools and their respective configuration, shared config variables are:

```
threads_demux: 4
demux_batch_size: 4000
demux_default: 'deepbinner'
```

Deepbinner

Deepbinner: Demultiplexing barcoded Oxford Nanopore Technologies reads with deep convolutional neural networks (CNN). The network is trained to classify barcodes based on the raw nanopore signal. The model for the CNN needs to be copied from the Deepbinner repository to the working directory and depends on the used sequencing kit. The kit is parsed from the run name as described in the configuration, alternatively the *default* mapping can be used to override the kit of the runname.

```
deepbinner_models:
    default: SQK-RBK004_read_starts
    EXP-NBD103: EXP-NBD103_read_starts
    SQK-RBK004: SQK-RBK004_read_starts
```

References

> Wick, R. R., Judd, L. M. & Holt, K. E. Deepbinner: Demultiplexing barcoded Oxford Nanopore reads with deep convolutional neural networks. PLOS Computational Biology 14, e1006583 (2018).

Intro

The tutorial is a step by step guide into the usage of the pipeline. Aside from the preparation steps in the introduction, the different parts work independently from each other. The examples will make use of the test data provided for the unit tests of Nanopype. The following steps assume Nanopype is installed in your home directory `~/src/nanopype` and you have activated the virtual environment if used. Also make sure you have run the tests of installation and configuration.

To get started create an empty directory and copy the read archive from the Nanopype repository into it.

Docker

```
docker run -it giesselmann/nanopype
mkdir -p /processing/nanopype_tutorial
cd /processing/nanopype_tutorial
cp /app/test/test_data.tar.gz ./
```

Plain

```
mkdir nanopype_tutorial
cd nanopype_tutorial
cp ~/src/nanopype/test/test_data.tar.gz ./
```

Within the docker, Nanopype is installed into the `/app` folder, subsequent commands need to be adjusted accordingly.

```
mkdir -p data/raw
tar -zxf test_data.tar.gz -C data/raw
ls -l data/raw
```

The last step is to copy the config template to our working directory and adjust the raw data path:

```
cp ~/src/nanopype/nanopype.yaml ./
```

With your favorite text editor change the path to:

```
# data paths
storage_data_raw : data/raw
```

Data Import

Raw data from the sequencer in *fast5* format needs to be imported into Nanopype compatible batches. Each batch contains a configurable number of reads. This tutorial covers the import and indexing of multiple runs.

Since the provided test data is already properly packaged, we need to restore the original MinKNOW output.

```
mkdir -p data/import
for d in data/raw/*; do\
    run=$(basename $d);
    mkdir -p data/import/$run;
    echo $run;
    ls $d/reads/*.tar | xargs -n 1 tar -C data/import/$run -xf;
done
```

Packaging

The next steps use the import script of Nanopype to pack single reads into *tar* archives. First we reproduce the test data archives:

```
mkdir -p data/raw2
for d in data/import/*; do\
    run=$(basename $d);
    python3 ~/src/nanopype/scripts/nanopype_import.py data/
raw2/$run $d \
     --recursive --batch_size 4;
done
```

An import session is generating detailed output similar to the following:

```
19.12.2018 00:23:35 [INFO] Logger created
19.12.2018 00:23:35 [INFO] Writing output to /home/devel/
nanopype_test/data/raw2/20180221_FAH48596_FLO-MIN107_SQK-
LSK108_human_Hues64/reads
19.12.2018 00:23:35 [INFO] Inspect existing files and archives
19.12.2018 00:23:35 [INFO] 0 raw files already archived
19.12.2018 00:23:35 [INFO] 12 raw files to be archived
```

```
19.12.2018 00:23:35 [INFO] Archived 4 reads in /home/devel/
nanopype_test/data/raw2/20180221_FAH48596_FLO-MIN107_SQK-
LSK108_human_Hues64/reads/0.tar
19.12.2018 00:23:35 [INFO] Archived 4 reads in /home/devel/
nanopype_test/data/raw2/20180221_FAH48596_FLO-MIN107_SQK-
LSK108_human_Hues64/reads/1.tar
19.12.2018 00:23:35 [INFO] Archived 4 reads in /home/devel/
nanopype_test/data/raw2/20180221_FAH48596_FLO-MIN107_SQK-
LSK108_human_Hues64/reads/2.tar
19.12.2018 00:23:35 [INFO] Mission accomplished
```

Verification

To compare the Nanopype test archives to the newly created ones you could run:

```
run=20180221_FAH48596_FLO-MIN107_SQK-LSK108_human_Hues64
cmp --silent data/raw2/$run/reads/0.tar data/raw/$run/reads/0.tar
&&\
 echo 'Archives identical!'
```

Re-running the archive script will detect already existing data and create new batches if the source directory contains files not yet present in the output folder.

```
run=20180221_FAH48596_FLO-MIN107_SQK-LSK108_human_Hues64
python3 ~/src/nanopype/scripts/nanopype_import.py data/raw2/$run
data/import/$run\
 --recursive --batch_size 4;
```

It is recommend to always run the verification after an import process to ensure all reads from the sequencer are successfully archived.

Alignment

This tutorial covers the basic steps to align Oxford Nanopore Technologies data using Nanopype. Common alignment algorithms work in sequence space and depend on previously base called reads. Different combinations of base caller and aligner are supported and may be utilized depending on the intended analysis.

Before starting a first alignment, a reference sequence is required. The given test data is extracted from a region on human chr6 which we will download into a reference folder in the current working directory:

```
mkdir -p references
wget -O references/chr6.fa.gz http://hgdownload.cse.ucsc.edu/
goldenpath/hg38/chromosomes/chr6.fa.gz
gzip -d references/chr6.fa.gz
```

The *env.yaml* in the original repository already contains an entry for this tutorial:

```
references:
    test:
        genome: references/chr6.fa
```

Implicit base calling

The easiest way to obtain an alignment is to directly request the output file with:

```
snakemake --snakefile ~/src/nanopype/Snakefile -j 4 alignments/
minimap2/guppy/batches/tutorial/20170725_FAH14126_FLO-MIN107_SQK-
LSK308_human_Hues64.test.bam
```

This will trigger base calling using guppy and an alignment against our test genome with minimap2. In order to process multiple runs in parallel and merge the results into a single output file, a runnames.txt in the working directory is used:

```
for d in data/raw/*; do echo $(basename $d); done > runnames.txt
snakemake --snakefile ~/src/nanopype/Snakefile -j 4 alignments/
minimap2/guppy/tutorial.test.bam
```

Snakemake will automatically detect the already present output from flow cell *FAH14126* and start processing the remaining datasets from the runnames.txt file.

Explicit base calling

Often you may already have base called data and want to run only the alignment. Nanopype will detect existing sequence data in fasta and fastq format and use them as input. For this example we will first run the base calling with a different tool and manually start the alignment afterwards:

```
snakemake --snakefile ~/src/nanopype/Snakefile -j 4 sequences/
albacore/tutorial.fa.gz
snakemake --snakefile ~/src/nanopype/Snakefile -j 4 alignments/
minimap2/albacore/tutorial.test.bam
```

Note that the second command only contains jobs running minimap2.

Batch processing

Browsing the working directory after this tutorial you will notice the batch output in form of e.g. *0.fastq.gz* or *0.test.bam* in a specific batches folder. These files are temporary and can be deleted after the processing. They are however required as input for the processing, thus if you delete the sequence batches a subsequent alignment run will include base calling to get the batch-wise input.

Citation

Nanopype integrates a growing number of nanopore sequencing related tools. If you use nanopype in your project please cite our pipeline. Furthermore please verify the set of tools your analysis made use of and also cite the according publications.

Snakemake

> J. Köster and S. Rahmann; Snakemake - A scalable bioinformatics workflow engine, Bioinformatics (2012).

Processing functions

Minimap2

> H. Li; Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics. doi:10.1093/bioinformatics/bty191 (2018).

GraphMap

> Sovic, I. et al. Fast and sensitive mapping of nanopore sequencing reads with GraphMap. Nat. Commun. 7:11307 doi: 10.1038/ncomms11307 (2016).

NGMLR

> Sedlazeck, F. J. et al. Accurate detection of complex structural variations using single-molecule sequencing. Nature Methods 15, 461-468 (2018).

Analysis functions

Nanopolish

> Simpson, Jared T. et al. Detecting DNA cytosine methylation using nanopore sequencing. Nature Methods 14.4, 407-410 (2017).

Deepbinner

> Wick, R. R., Judd, L. M. & Holt, K. E; Deepbinner: Demultiplexing barcoded Oxford Nanopore reads with deep convolutional neural networks, PLOS Computational Biology 14, e1006583 (2018).

Sniffles

> Sedlazeck, F. J. et al. Accurate detection of complex structural variations using single-molecule sequencing, Nature Methods 15, 461-468 (2018).

Miscellaneous

Bedtools

> Aaron R. Quinlan, Ira M. Hall; BEDTools: a flexible suite of utilities for comparing genomic features, Bioinformatics, doi:10.1093/bioinformatics/btq033 (2010).

Samtools

> Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, 1000 Genome Project Data Processing Subgroup; The Sequence Alignment/Map format and SAMtools, Bioinformatics, doi:10.1093/bioinformatics/btp352 (2009).

UCSCTools

> W. J. Kent, A. S. Zweig, G. Barber, A. S. Hinrichs, D. Karolchik; BigWig and BigBed: enabling browsing of large distributed datasets, Bioinformatics, doi:10.1093/bioinformatics/btq351 (2010).

License

Nanopype

MIT License

Tools

External software downloaded and build for the Nanopype pipeline as described in the installation section is left untouched by the above statement. Please refer to the documented sources to obtain a copy of the respective licenses.