Protocols are updated regularly, please check this is the latest version before proceeding. This protocol is for research only.

Overview

Guppy software overview

IMPORTANT

Guppy license

Guppy is available under the Oxford Nanopore TechnologiesTerms and Conditions.

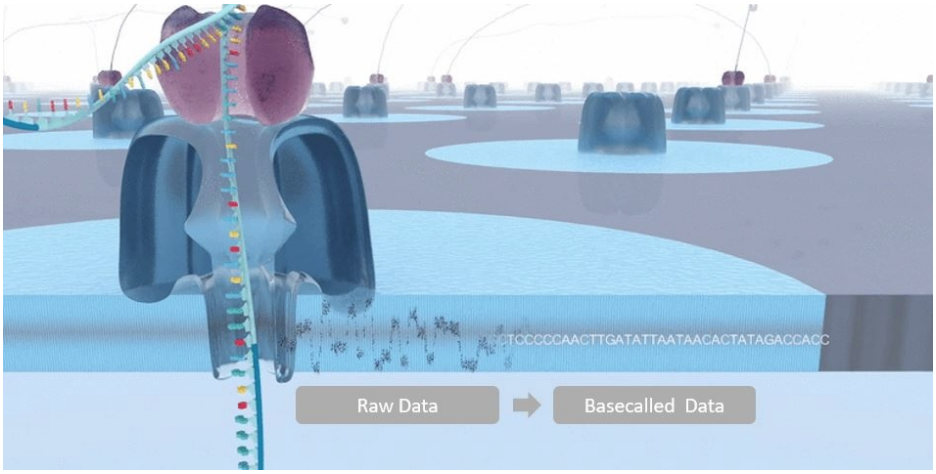
Guppy basecalling software

Guppy is a data processing toolkit that contains the Oxford Nanopore Technologies' production basecalling algorithms and several bioinformatic post-processing features. It is run from the command line in Windows, Mac OS, and on multiple Linux platforms. Guppy is also integrated with our sequencing instrument software, MinKNOW, and a subset of Guppy features are available via the MinKNOW UI. A selection of configuration files allows basecalling of DNA and RNA libraries made with Oxford Nanopore Technologies' current sequencing kits, in a range of flow cells.

The Guppy software contains many configurable parameters that can be used to specify exactly how the data analysis is performed. Adjusting some of these parameters requires a deep knowledge of nanopore data, and as such, Guppy is aimed at more advanced users. For those who are new to sequencing or have limited knowledge of sequencing data analysis, we recommend using the options presented in the MinKNOW software UI for basecalling.

Introduction to basecalling

Basecalling is the process of converting the electrical signals generated by a DNA or RNA strand passing through the nanopore into the corresponding base sequence of the strand. The general data flow in a nanopore sequencing experiment is shown below.



Raw data - a direct measurement of the changes in ionic current as a DNA/RNA strand passes through the pore, which are recorded by the MinKNOW software. MinKNOW also processes the signal into "reads", each read corresponding to a single strand of DNA/RNA. These reads are optionally written out as .fast5 files.

These .fast5 files use the HDF5 format to store data (http://www.hdfgroup.org/HDF5/); libraries exist to read and write these in many popular computer languages (e.g. R, Python, Perl, C, C++, Java).

Basecalling - the raw signal is further processed by the basecalling algorithm to generate the base sequence of the read.

Basecalling is made up of a series of steps that are executed one by one. Ionic current measurements from the sequencing device are

collected by the MinKNOW software and processed into a read. The reads are transformed into basecalls using mathematical models. The results of these analyses are written into FASTQ files, with a default of 4000 reads per FASTQ file. Alternatively, the user can select a .fast5 file output for additional information about the raw signal (similarly, the default is 4000 reads per .fast5 file).

Guppy basecalling models are based on a Recurrent Neural Network (RNN)

The Guppy basecalling models are based on RNNs. For more information about RNNs, as well as other basecalling options and algorithms, please refer to the Data Analysis document in the Nanopore Community.

The Guppy toolkit contains:

- **The basecaller:** The Guppy basecaller implements a neural networks algorithm that allows raw data to be transformed into canonical bases of DNA or RNA, and several types of modified bases.
    - **Calibration strand detection:** The basecaller is also capable of detecting calibration strands by aligning calibration sequences. Reads are aligned against a calibration reference using the basecalled data from an internally present DNA molecule in the flow cell. Calibration strands serve as a quality control for the pore and experimental processing. If the current read is identified as a calibration strand, no barcoding or alignment steps are performed.
    - **Adapter trimming:** This is the processing and removal of the sequencing adapter (e.g. AMX, BAM, AMII, etc.) signal in the basecalled data:
        - For DNA adapters it will exclude the non-sequence adapter region up to a characteristic signal in the adapter that is recognised by the basecaller.
        - For (m)RNA, where the strands are sequenced in the 3' to 5' direction, it will attempt to exclude all data up to the the polyA tail.
- **The 1D^2 basecaller:** The 1D^2 basecaller analyses both forward and reverse strands of double-stranded DNA, to generate a basecall. Adjacent strands are identified as 1D^2 pairs, where the second strand is the reverse complement of the first strand, then the strands are processed together to produce a 1D^2 basecall. The combination of the signals produces a higher quality sequence than the individual 1D reads.
- **Barcoding/demultiplexing:** The beginning and the end of each strand are aligned against the barcodes currently provided by Oxford Nanopore Technologies. Demultiplexing occurs directly from the basecalled results.
- **Alignment:** The user can provide a reference file in FASTA or minimap2 index format. If so, the reads are aligned against this reference via the integrated minimap2 aligner using the standard Oxford Nanopore Technologies preset parameters.
- **Modified basecalling:** It is possible to use Guppy to identify certain types of modified bases: currently 6mA dam/5mC dcm and CpG. This requires the use of a specific basecalling model which is trained to identify both modified and unmodified bases.

Current assumptions and limitations of Guppy

The Guppy basecalling software currently provides basecalling for 1D and 1D^2 chemistry.

Read .fast5 files, used as input to the basecalling software, must contain raw data. Raw data has been included by default in .fast5 files generated by the MinKNOW software for the last several years, so it should not be necessary to update them. Oxford Nanopore Technologies offers two sets of tools for working with .fast5 files that users may find helpful:

- ont_fast5_api: Provides a simple interface to the .fast5 format, including tools for converting between single- and multi-read formats.
- ont_h5_validator: Provides a tool for validating .fast5 file structures against official Oxford Nanopore Technologies file schemas.

Guppy provides configurations for currently-available chemistries and also provides a model compatible with data generated using older PromethION firmware.

Both the alignment and barcoding pipelines accept compressed and uncompressed FASTQ files as input. These can be generated either by the Guppy basecallers, or by the MinKNOW software.

General system requirements for running for Guppy

These system requirements are guidelines - the actual amount of memory and disk space required to run Guppy tools will heavily depend on options and input data.

- 4 GB RAM plus 1 GB per thread for 1D basecalling
- 4 GB RAM plus 2 GB per thread for $1D^2$ basecalling
- Administrator access for .deb or .msi installers
- ~100 Mb of drive space for installation, minimum 512 GB storage space for basecalled read files (1 TB recommended)

When starting with a .fast5 file that only has raw data in it, the output file size will increase to approximately 2x original size. $1D^2$ basecalling only produces FASTQ files, which are significantly smaller than .fast5 files.

CPU and GPU basecalling with Guppy

Oxford Nanopore Technologies provides Guppy executables that can be run on Central Processing Units (CPUs) on Windows, Mac OS and Linux, or on Graphics Processing Units (GPUs) on certain Linux platforms:

- **Windows:** ont-guppy-cpu .msi installer (CPU only)
- **macOS:** ont-guppy-cpu .dmg installer (CPU only)
- **Linux:**
    - **CPU**
        - ont-guppy-cpu .deb for Ubuntu 16
        - ont-guppy-cpu .rpm for Centos 7
        - ont-guppy-cpu .tar.gz – general Linux archives with pre-built binaries (compatible with most Linux versions)
    - **GPU**
        - ont-guppy .deb for Ubuntu 16
        - ont-guppy .rpm for Centos 7
        - ont-guppy .tar.gz – general Linux archives with pre-built binaries (compatible with most Linux versions)

Note that GPU basecalling is only supported on Linux systems. Mac OS systems do not have NVIDIA GPUs or CUDA support, while Windows GPU basecalling is not currently supported by Oxford Nanopore Technologies.

Using external GPUs can dramatically increase basecalling speed. Guppy works with only NVIDIA GPUs, with the following specific models fully supported:

- NVIDIA Tesla V100
- NVIDIA Quadro GV100
- NVIDIA Jetson TX2

If working with a different type of GPU than the models listed above, we recommend CUDA Compute Capability >6.1 (for more information about CUDA-enabled GPUs, see the NVIDIA website)

It is possible to use other NVIDIA GPUs for basecalling, however Oxford Nanopore Technologies develops and tests software on the models stated above, so support for other models is limited.

Fast vs High Accuracy models

The Guppy basecaller, which is also integrated in MinKNOW, offers two different Flip-flop models: a High accuracy (HAC) model and a Fast model. The HAC model provides a higher consensus/raw read accuracy than the Fast model. It contains a more computationally-intense Flip-flop architecture that can deliver higher accuracy using the same data produced by nanopore sequencing. It is currently 5-8 times slower than the Fast model.

The Fast Flip-flop model includes a simplified version of the Flip-flop algorithm and delivers the best level of accuracy that is achievable while keeping up with data generation on all devices. Both models have been trained on the same datasets.

A comparison of the speed and accuracy of the two models is provided in the table and graphs below.

Please note that these numbers represent the theoretical best speeds achievable with the basecallers, and a real sequencing experiment may be basecalled more slowly.

		Data generation in Gbases per hour				
	Output	Flongle	MinION	GridION	P24	P48
Per flow cell	2 Gb Flongle 30 Gb MinION 150 Gb PromethION	0.08	0.42	0.42	2.1	2.1
Per full device		-	-	2.1	50	100

		Basecalling speed in Gbases per hour				
	Raw Read Accuracy	MinION high-spec laptop	MinIT and MinION Mk 1c	GridION Mk 1	P24	P48
R9.4.1 Fast	92.1%	0.13	0.24	40	87	116
R9.4.1 HAC	95%	0.02	0.06	5	18	27
R10.3 Fast	-	-	-	-	-	-
R10.3 HAC	96%	-	0.04	6	11	19

Basecalling speed for Guppy

Aside from the basecalling model, the time taken to basecall a folder of reads depends on the specifications of the computer, the number of threads assigned, the options which Guppy is invoked with, and the number of reads analysed. Guppy is optimised for NVIDIA GPUs using CUDA, and can perform several orders of magnitude faster running on a modern GPU compared to a standard desktop CPU.

Windows

IMPORTANT

Guppy can only be installed from the Administrator account.

CPU basecalling for Guppy in Windows

Only CPU basecalling and post-processing features are available in Guppy for Windows in the `ont-guppy-cpu` toolkit.

Supported platforms for Guppy

- 64-bit Windows 10
- 64-bit Windows 7 SP1 and the Windows 10 Universal C Runtime

IMPORTANT

Additional requirements for Windows 7 to install Guppy:

Guppy is optimised for Windows 10. If you are using earlier versions of Windows, please downloadWindows 10 Universal C Runtime in order for Guppy to work.

If Universal C Runtime in not installed, the following error message may appear:

The procedure entry point ucrtbase.terminate could not be located in the dynamic link library api-ms-win-crt-runtime-l1-1-10.dll

1 **Download the .msi installer for Guppy**

The installer can be found on theSoftware Downloads page of the Community.

2 **Double-click on the installer.**

3 **Follow the prompts from the executable to install Guppy.**

The default install location is:

C:\Program Files\OxfordNanopore\ont-guppy-cpu

Linux

IMPORTANT

Guppy can only be installed from the Administrator account.

Choosing a Guppy package

Depending on whether you are basecalling on a CPU or a GPU, you will need to choose an appropriate Guppy package. These packages are available as separate installers on the Software Downloads page:

- GPU basecalling is available in the ont-guppy package and will require a GPU driver for basecalling to work. It is also possible to perform CPU basecalling using this package. You will need to specify the GPU device(s) you want to use when setting up your experiment, otherwise Guppy will default to CPU calling.
- CPU-only basecalling is available in the ont-guppy-cpu package.

Supported platforms for Guppy

Debian packages:
- 64-bit Ubuntu 16 amd64 (for either GPU-enabled Guppy or cpu-only Guppy)
- 64-bit Ubuntu 16 arm64v8 (for GPU-enabled Guppy only)

RPM packages:
- Centos 7 (for either GPU-enabled Guppy or CPU-only Guppy)

Archive package:

- Most 64-bit amd64 Linux platforms (for either GPU-enabled Guppy or CPU-only Guppy – the package was built on Centos 6)
- 64-bit Ubuntu 16 arm64v8 (for GPU-enabled Guppy only - the package was built on a Jetson TX2)

GPU devices:
- A supported NVIDIA GPU for the ont-guppy packages

From version 2.3.8 onwards, Guppy no longer officially supports Ubuntu 14.04, as that version of Ubuntu has reached its End of Life (EOL) date. It should still be possible to run the Guppy archive releases on that platform, although it is not explicitly supported.

Use this installation process if you are installing from .deb for Guppy:

1. Add Oxford Nanopore's deb repository to your system (this is to install Oxford Nanopore Technologies-specific dependency packages):

```
sudo apt-get update
sudo apt-get install wget lsb-release
export PLATFORM=$(lsb_release -cs)
wget -O- https://mirror.oxfordnanoportal.com/apt/ont-repo.pub | sudo apt-key add -
echo "deb http://mirror.oxfordnanoportal.com/apt ${PLATFORM}-stable non-free" | sudo tee /etc/apt/sources.list.d/nanoporetech.sources.list
sudo apt-get update
```

1. To install the .deb for Guppy, use the following command:

```
sudo apt update
sudo apt install ont-guppy
```

This will install the GPU version of Guppy.

or:

```
sudo apt update
sudo apt install ont-guppy-cpu
```

To install the CPU-only version of Guppy.

Use this installation process if you are installing from .tar.gz:

1. Download the .tar.gz archive file. This can be found on the Software Downloads page of the Nanopore Community.
2. Unpack the archive:

```
tar -xf ont-guppy_xxx_linux<64 or aarch64>.tar.gz
```

or to unpack the CPU-only version of Guppy:

```
tar -xf ont-guppy-cpu_xxx_linux<64 or aarch64>.tar.gz
```

Note: 'xxx' in the command denotes the version number e.g. 3.0.3.

Use this installation process if you are installing from .rpm:

1. Download the .rpm file. This can be found on the Software Downloads page of the Nanopore Community.
2. Install the epel-release repository:

yum install epel-release

1. Install the rpm:

```
yum install <path>/<to>/ont-guppy_xxx.rpm
```

This command installs the GPU version of Guppy.

or

```
yum install <path>/<to>/ont-guppy-cpu_xxx.rpm
```

This command installs the CPU-only version of Guppy.

The GPU-enabled ont-guppy package will not install a GPU driver by default – it will be necessary to handle installing this yourself, e.g. by visiting NVIDIA's website. Guppy requires an NVIDIA driver of at least version 384.

Using unsupported NVIDIA GPUs

Newer GPUs may require more up-to-date versions of NVIDIA drivers and CUDA APIs than the ont-guppy deb recommends. It is possible to install the ont-guppy deb without installing any GPU drivers by not installing the recommended ones, leaving it to the user to ensure that their system has the required minimum drivers:

```
apt-get install ont-guppy --no-install-recommends
```

In general NVIDIA APIs and drivers are backwards compatible, so it will usually be safe to install a later / higher version than Guppy requires. Check for the minimum driver version that Guppy requires using `apt-cache show`:

```
$ apt-cache show ont-guppy
Package: ont-guppy
Version: 0.0.0-1~xenial
Architecture: amd64
Depends: libc6 (>=2.23), libcurl4-openssl-dev, libssl-dev, libhdf5-cpp-11, libzmq5, libboost-atomic1.58.0, libboost-chrono1.58.0, libboost-date-time1.58.0, libboost-filesystem1.58.0, libboost-program-options1.58.0, libboost-regex1.58.0, libboost-system1.58.0, libboost-log1.58.0
Recommends: nvidia-384, libcuda1-384
[...]
```

In this example the driver version (384) can be seen as part of the `nvidia` and `libcuda1` package names in the "Recommends" section. Driver versions higher than 384 will likely be compatible with this version of Guppy.

Note that Guppy only supports GPUs with an NVIDIA compute version of 6.1 or higher.

Setting custom GPU parameters in Guppy

If using an unsupported GPU then the following calculation provides a rough ceiling to the amount of GPU memory that Guppy will use:

```
runners * chunks_per_runner * chunk_size < 100000 * [max GPU memory in GB]
```

For example, a GPU with 8GB of memory would require:

```
runners * chunks_per_runner * chunk_size < 800000
```

macOS

CPU basecalling for Guppy on macOS

Only CPU basecalling and post-processing features are available in Guppy for Mac OS in the ont-guppy-cpu toolkit.

Supported platforms:

- 64-bit OSX 10.11 (El Capitan) or higher

1 **Download the .zip archive.**

This can be found on the Software Downloads page in the Community.

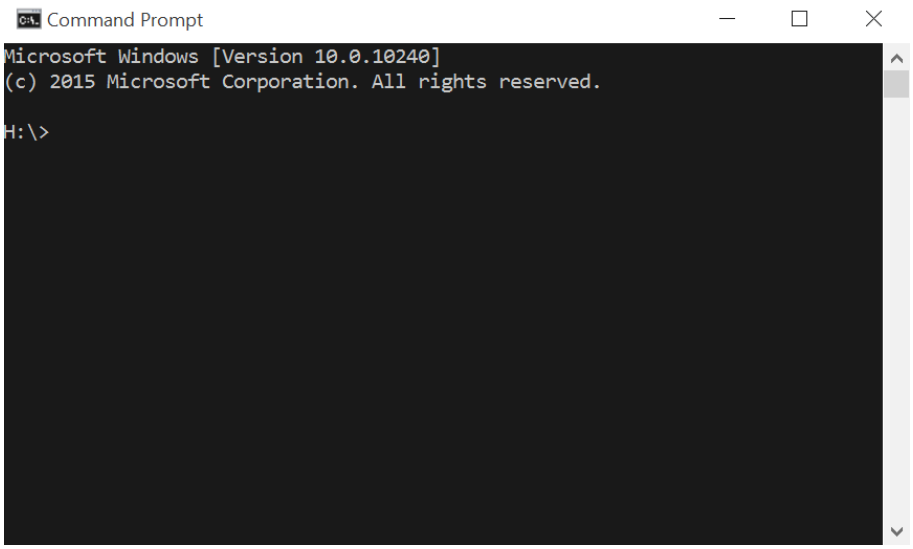
2 **Unzip the archive to a location of your choice.**

Note: You may require Administrator access depending on the location into which you unzip the archive.

Windows

1 **Command prompt**

Launch a command prompt. To do this, click on Start and type "Command prompt" in the search box, then click the link. You will be running all Guppy operations from here:



Required parameters

To start a sequencing run with Guppy, you will need to type a command that specifies, as a minimum, the following parameters:

- Whether you are running 1D or 1D^2 chemistry
- The flow cell and sequencing kit versions that were used*
- The full path to the directory where the raw read files are located
- The full path to the directory where the basecalled files will be saved

* Rather than specifying by flow cell and kit, a specific configuration file can be used.

Default parameters

If you only specify the parameters listed above, Guppy will run with a number of other parameters set at their default values, for example:

- The basecall model will be set as High Accuracy (HAC)
- Q-score filtering will be set to OFF
- File compression will be set to OFF
- Sequence reversal and U to T substitution for RNA will be set to OFF

For a full list of all the optional parameters and their default values, refer to the “Setting up a run: configurations and parameters” section of the protocol.

1D sequencing: command-line entries for basecalling

To basecall 1D reads with Guppy, you will need to use the following commands:

- "C:\Program Files\OxfordNanopore\ont-guppy-cpu\bin\guppy_basecaller.exe"
- --input_path Full or relative path to the directory where the raw read files are located. The folder can be absolute (e.g. C:\data\my_reads) or a relative path to the current working directory (e.g...\my_reads)
- --save_path Full or relative path to the directory where the basecall results will be saved. The folder can be absolute or a relative path to the current working directory. This folder will be created if it does not exist using the path you provide. (e.g. if it is a relative path, it will be relative to the current working directory)

Then either:
- --flowcell flow_cell_version --kit sequencing_kit version

or

- --config configuration file containing Guppy parameters

> **EXAMPLE**
>
> **1D Sequencing: Example command-line entries for basecalling an SQK-LSK109 kit with a FLO-MIN106 flow cell experiment**
>
> "C:\Program Files\OxfordNanopore\ont-guppy-cpu\bin\guppy_basecaller.exe" --input_path C:\my_folder\reads --save_path C:\output_folder\basecall --flowcell FLO-MIN106 --kit SQK-LSK109
>
> or
>
> "C:\Program Files\OxfordNanopore\ont-guppy-cpu\bin\guppy_basecaller.exe" --input_path reads –-save_path output_folder\basecall --config dna_r9.4.1_450bps.cfg

IMPORTANT

Guppy executable in Windows:

By default, the Guppy executables (such as `guppy_basecaller.exe`) will not be on the Windows file path. As a result, you will need to type the full path directory to use an executable.

Below is an example of the full file directory of a Guppy executable from the Guppy toolkit:

```
C:\Program Files\OxfordNanopore\ont-guppy-cpu\bin\guppy_basecaller.exe
```

2 **General help command-line options:**

For general help, the following available command-line options, `-h` or `--help`, are provided within the Guppy toolkit:

```
"C:\Program Files\OxfordNanopore\ont-guppy-cpu\bin\guppy_basecaller_1d2.exe" -h
```

or

```
"C:\Program Files\OxfordNanopore\ont-guppy-cpu\bin\guppy_basecaller.exe" --help
```

To call out a list of available flow cells, kits, and config files, use command:

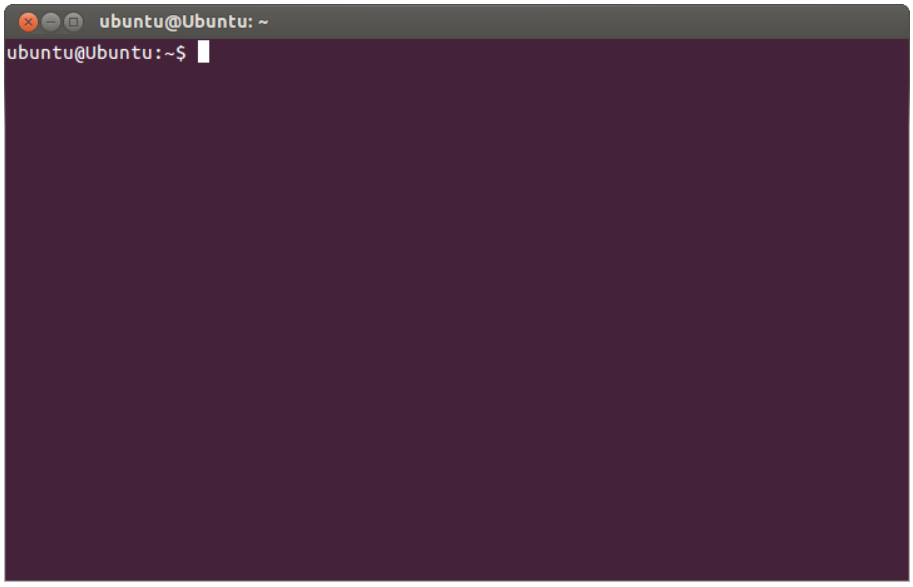
```
"C:\Program Files\OxfordNanopore\ont-guppy-cpu\bin\guppy_basecaller.exe" --print_workflows
```

The command `--version` shows the version of Guppy that is installed.

Linux

1 **Command prompt:**

Launch a terminal on your system. For example, when using Ubuntu, you can type Ctrl + Alt + T. You will run all Guppy operations from here:



Required parameters

To start a sequencing run with Guppy, you will need to type a command that specifies, as a minimum, the following parameters:

- Whether you are running 1D or 1D^2 chemistry
- The flow cell and sequencing kit versions that were used*
- The full path to the directory where the raw read files are located
- The full path to the directory where the basecalled files will be saved

* Rather than specifying by flow cell and kit, a specific configuration file can be used.

Default parameters

If you only specify the parameters listed above, Guppy will run with a number of other parameters set at their default values, for example:

- The basecall model will be set as High Accuracy (HAC)
- Q-score filtering will be set to OFF
- File compression will be set to OFF
- Sequence reversal and U to T substitution for RNA will be set to OFF

For a full list of all the optional parameters and their default values, refer to the “Setting up a run: configurations and parameters” section of the protocol.

1D sequencing: command-line entries for basecalling

To basecall 1D reads with Guppy, you will need to use the following commands:

- guppy_basecaller (or the fully-qualified path if using the archive installer)
- --input_path Full or relative path to the directory where the raw read files are located. The folder can be absolute (e.g.

C:\data\my_reads) or a relative path to the current working directory (e.g...\my_reads)

- --save_path Full or relative path to the directory where the basecall results will be saved. The folder can be absolute or a relative path to the current working directory. This folder will be created if it does not exist using the path you provide. (e.g. if it is a relative path, it will be relative to the current working directory)

Then either:

- --flowcell flow_cell_version --kit sequencing_kit version

or

- --config configuration file containing Guppy parameters

EXAMPLE

1D Sequencing: Example command-line entries for basecalling an SQK-LSK109 kit with a FLO-MIN106 flow cell experiment

```
guppy_basecaller --input_path /data/my_folder/reads --save_path /data/output_folder/basecall --flowcell FLO-MIN106 --kit SQK-LSK109
```

or

```
guppy_basecaller --input_path reads --save_path output_folder/basecall --config dna_r9.4.1_450bps.cfg
```

TIP

Alternative commands:

On Linux-based platforms, it is also possible to enter files into Guppy, as follows:

```
ls input_folder/*.fast5 | guppy_basecaller --save_path output_folder/basecall --config dna_r9.4.1_450bps.cfg
```

2 **Additional options can be specified to enable different basecalling features and output formats and these are discussed further in later sections.**

Note that on Linux-based platforms it is also possible to pipe files into Guppy, as follows:

```
ls input_folder/*.fast5 | guppy_basecaller --save_path output_folder/basecall --config dna_r9.4.1_450bps_fast.cfg
```

3 **General help command-line options:**

For general help, the following available command-line options,`-h` or `--help`, are provided within the Guppy toolkit:

```
guppy_basecaller -h
```

or

```
guppy_basecaller --help
```

To call out a list of available flow cells, kits, and config files, use command::

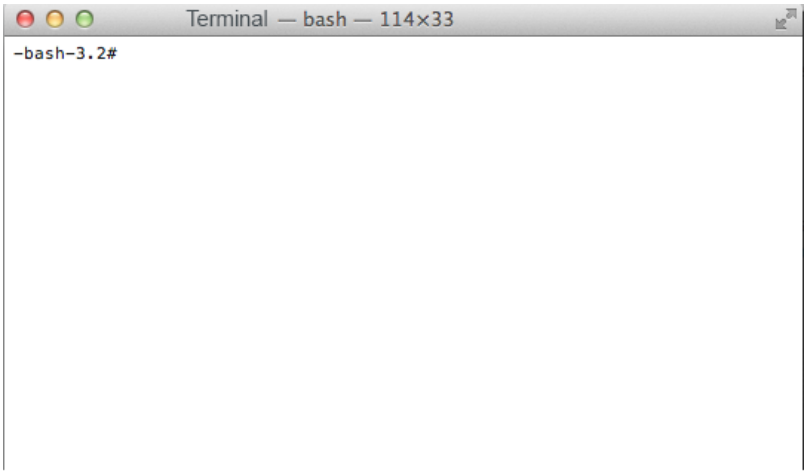
```
guppy_basecaller --print_workflows
```

The command `--version` shows the version of Guppy that is installed.

macOS

Command prompt:

Open a command-line terminal. Open your Applications folder, then open the Utilities folder. Click on the Terminal application to open:



IMPORTANT

Before starting:

You must find where the unzipped 'Guppy archive' is located – this will give you the path you will need to enter in order to run the Guppy executables.

For example, if you extracted the .zip archive to`/Users/myuser/ont-guppy-cpu`, then you can run the 1D basecaller using this:

```
/Users/myuser/ont-guppy-cpu/bin/guppy_basecaller
```

Required parameters

To start a sequencing run with Guppy, you will need to type a command that specifies, as a minimum, the following parameters:

- Whether you are running 1D or $1D^2$ chemistry
- The flow cell and sequencing kit versions that were used*
- The full path to the directory where the raw read files are located
- The full path to the directory where the basecalled files will be saved

* Rather than specifying by flow cell and kit, a specific configuration file can be used.

Default parameters

If you only specify the parameters listed above, Guppy will run with a number of other parameters set at their default values, for example:

- The basecall model will be set as High Accuracy (HAC)
- Q-score filtering will be set to OFF
- File compression will be set to OFF
- Sequence reversal and U to T substitution for RNA will be set to OFF

For a full list of all the optional parameters and their default values, refer to the “Setting up a run: configurations and parameters” section of the protocol.

1D sequencing: command-line entries for basecalling

To basecall 1D reads with Guppy, you will need to use the following commands:

- guppy_basecaller
- --input_path Full or relative path to the directory where the raw read files are located. The folder can be absolute (e.g. C:\data\my_reads) or a relative path to the current working directory (e.g...\my_reads)
- --save_path Full or relative path to the directory where the basecall results will be saved. The folder can be absolute or a relative path to the current working directory. This folder will be created if it does not exist using the path you provide. (e.g. if it is a relative path, it will be relative to the current working directory)

Then either:

- --flowcell flow_cell_version --kit sequencing_kit version

or

- --config configuration file containing Guppy parameters

> **EXAMPLE**
>
> **1D sequencing: example command-line entries for basecalling a SQK-LSK109 kit with a FLO-MIN106 flow cell experiment**
>
> /Users/myuser/ont-guppy-cpu/bin/guppy_basecaller --input_path /data/my_folder/reads --save_path /data/output_folder/basecall --flowcell FLO-MIN106 --kit SQK-LSK109
>
> or
>
> /Users/myuser/ont-guppy-cpu/bin/guppy_basecaller --input_path reads --save_path output_folder/basecall --config dna_r9.4.1_450bps.cfg

General help command-line options:

For general help, the following available command-line options,`-h` or `--help`, are provided within the Guppy toolkit:

```
/Users/myuser/ont-guppy-cpu/bin/guppy_basecaller -h
```

or

```
/Users/myuser/ont-guppy-cpu/bin/guppy_basecaller --help
```

To call out a list of available flow cells, kits, and config files, use command::

```
/Users/myuser/ont-guppy-cpu/bin/guppy_basecaller --print_workflows
```

The command `--version` shows the version of Guppy that is installed.

Setting up a run: configurations and parameters

IMPORTANT

Model selection

Guppy contains several types of basecalling models, some of which are not available by using the flow cell and kit selector. These models will usually have their own config file, and they may then be used with the `--config` argument. The model types are:

- High Accuracy basecalling (set by default when a kit and flow cell are chosen)
- Fast basecalling
- Modified basecalling

For example, basecall using the R9.4.1 Fast Flip-flop model like this:

```
guppy_basecaller -c dna_r9.4.1_450bps_fast.cfg [...]
```

Optional parameters

In addition to the required parameters described in the Quick Start section, Guppy has many optional parameters. You can use them if they are applicable to your experiment. The following optional parameters are commonly used:

Data features:

- Q-score filtering (`--qscore_filtering`): Flag to enable filtering of reads into pass/fail folders inside the output folder, based on their strand q-score. Off by default. See `--min_qscore`
- Minimum q-score (`--min_qscore`): The minimum q-score a read must attain to pass qscore_filtering. The default value for this is 7, roughly corresponding to basecall accuracy of 85%.
- Calibration strand detection (`--calib_detect`): Flag to enable calibration strand detection and filtering. If enabled, any reads which align to the calibration strand reference will be filtered into a separate output folder to simplify downstream processing. Off by default.
- Reverse RNA sequence (`--reverse_sequence`): Reverse the called sequence (used for RNA sequencing, as RNA strands translocate through the pore in the 3’ to 5’ direction). The default value is "false" for DNA sequencing and "true" for RNA sequencing.

- Perform U to T substitution (--u_substitution): Substitute 'U' for 'T' in the called sequence (for RNA sequencing). The default value is "false" for DNA sequencing and "true" for RNA sequencing.

Input/output:

- Quiet mode (-z or --quiet): This option prevents the Guppy basecaller from outputting anything to stdout. Stdout is short for “standard output” and is the default location to which a running program sends its output. For a command line executable, stdout will typically be sent to the terminal window from which the program was run.
- Verbose logging (--verbose_logs): Flag to enable verbose logging (outputting a verbose log file, in addition to the standard log files, which contains detailed information about the application). Off by default.
- Reads per FASTQ file (-q or --records_per_fastq): The number of reads to put in a single FASTQ file (see output format below). Set this to zero to output all reads into one file (per run id, per caller). The default value is 4000.
- Perform FASTQ compression (--compress_fastq): Flag to enable gzip compression of output FASTQ files; this reduces file size to about 50% of the original.
- Recursive (-r or --recursive): Flag to require searching through all subfolders contained in the --input_path value, and basecall any .fast5 files found in them.
- .fast5 file output (--fast5_out): Flag to enable outupt of .fast5 files containing original raw reads, event data from basecall and basecall result sequence. Off by default.
- Override default data path (-d or --data_path): Option to explicitly specify the path to use for loading any data files the application requires (for example, if you have created your own model files or config files).
- Input File List (--input_file_list): Optional file containing list of input .fast5 files to process from the input_path.
- Nested output folder structure (--nested_output_folder): Optional flag, which if set will cause FASTQ files to be output to a nested folder structure similar to that used by MinKNOW.
- Progress stats reporting frequency (--progress_stats_frequency): Frequency in seconds in which to report progress statistics, if supplied will replace the default progress display.

Optimisation:

- Chunks per caller (--chunks_per_caller): A soft limit on the number of chunks in each basecaller's chunk queue. When a read is sent to the basecaller, it is broken up into “chunks” of signal, and each chunk is basecalled in isolation. Once all the chunks for a read have been basecalled, they are combined to produce a full basecall. --chunks_per_caller sets a limit on how many chunks will be collected before they are dispatched for basecalling. On GPU platforms this is an important parameter to obtain good performance, as it directly influences how much computation can be done in parallel by a single basecaller.
- Number of parallel callers (--num_callers): Number of parallel basecallers to create. A thread will be spawned for each basecaller to use. Increasing this number will allow Guppy to make better use of multi-core CPU systems, but may impact overall system performance.
- GPU device (-x or --device): Specify a GPU device to use in order to accelerate basecalling. If this option is not selected, Guppy will default to CPU usage. You can specify one or more devices as well as optionally limiting the amount of GPU memory used (to leave space for other tasks to run on GPUs). GPUs are counted from zero, and the memory limit can be specified as percentage of total GPU memory or as size in bytes. Examples:

device	result
cuda:0	Use the first GPU in the system, no memory limit
cuda:0,1	Use the first two GPUs in the system, no memory limit
"cuda:0 cuda:1"	Same as cuda:0,1
cuda:all:100%	Use all GPUs in the system, no memory limit
cuda:1,2:50%	Use the second and third GPU in the system, and use only up to half of the GPU memory of each GPU
"cuda:0 cuda:1,2:8G"	Use the first three GPUs in the system. Use a maximum of 8 GiB on each of GPUs 1 and 2.
auto	Same as cuda:0

Note: Spaces are only allowed between multiple cuda: specifications. In this case it is necessary to put the entire device specification in quotes. It is strongly recommended to use a supported GPU if one is available, as basecalling will typically

perform orders of magnitude faster.

- Server connection hostname and port (`-p` or `--port`): Specify a hostname and port for connecting to basecall service (ie 'myserver:5555'), or port only (ie '5555'), in which case localhost is assumed. This is the port used to communicate between the basecall client and server. The client and server both need to use the same port number or they will not be able to connect to each other.
- Resume previous run (`--resume`): Flag to enable resuming a previous basecalling run. This option can be used to resume a partially completed basecall if it was interrupted for some reason, or to re-basecall an input directory if more reads were added.
- Client ID (`--client_id`): An identifier for the Guppy Client instance. If supplied, this identifier will be included in any files output by the Guppy Client. This may be used to guarantee unique filenames in the case that multiple Guppy Client processes are writing to the same output folder. This can used when there are multiple Guppy clients processing reads at the same time. To avoid the clients overwriting each other's files, giving each one a unique client ID will allow it to label its output files with the ID and make them unique per client.

Config files - variable parameters

In addition, Guppy must know which basecalling configuration to use. This can be provided in one of two ways:

- By selecting a flow cell and a kit:
  - Flow cell (`-f` or `--flowcell`): the name of the flow cell used for sequencing (e.g. FLO-MIN106).
  - Kit (`-k` or `--kit`): the name of the kit used for sequencing (e.g. SQK-LSK109).
- Or by selecting a config file:
  - Config (`-c` or `--config`): either the name of the config file to use, or a full path to a config file (see the section below). If the argument is only the name of a config file then it must correspond to one of the standard configuration files provided by the package.

Note: If you use the `--config` argument, the `--flowcell` and `--kit` arguments will be ignored. When in doubt, select a flow cell and kit and do not use `--config`.

Config files - selecting kit and flow cell

These should be clearly labelled on the corresponding boxes. Flow cells almost always start with "FLO" and kits almost always start with "SQK" or "VSK".

To see the supported flow cells and kits, run Guppy with the `--print_workflows` option:

```
guppy_basecaller --print_workflows
```

...which will produce output like this:

```
Available flowcell + kit combinations are:
flowcell kit barcoding config_name
FLO-MIN111 SQK-CAS109 dna_r10.3_450bps_hac
FLO-MIN111 SQK-DCS108 dna_r10.3_450bps_hac
FLO-MIN111 SQK-DCS109 dna_r10.3_450bps_hac
FLO-MIN111 SQK-LRK001 dna_r10.3_450bps_hac
FLO-MIN111 SQK-LSK108 dna_r10.3_450bps_hac
FLO-MIN111 SQK-LSK109 dna_r10.3_450bps_hac
FLO-MIN111 SQK-LSK109-XL dna_r10.3_450bps_hac
FLO-MIN111 SQK-LWP001 dna_r10.3_450bps_hac
FLO-MIN111 SQK-PCS108 dna_r10.3_450bps_hac
```

FLO-MIN111 SQK-PCS109 dna_r10.3_450bps_hac
FLO-MIN111 SQK-PRC109 dna_r10.3_450bps_hac
FLO-MIN111 SQK-PSK004 dna_r10.3_450bps_hac
FLO-MIN111 SQK-RAD002 dna_r10.3_450bps_hac
FLO-MIN111 SQK-RAD003 dna_r10.3_450bps_hac
FLO-MIN111 SQK-RAD004 dna_r10.3_450bps_hac
FLO-MIN111 SQK-RAS201 dna_r10.3_450bps_hac
FLO-MIN111 SQK-RLI001 dna_r10.3_450bps_hac
FLO-MIN111 VSK-VBK001 dna_r10.3_450bps_hac
FLO-MIN111 VSK-VSK001 dna_r10.3_450bps_hac
FLO-MIN111 VSK-VSK002 dna_r10.3_450bps_hac
FLO-MIN111 SQK-16S024 included dna_r10.3_450bps_hac
FLO-MIN111 SQK-PCB109 included dna_r10.3_450bps_hac
FLO-MIN111 SQK-RBK001 included dna_r10.3_450bps_hac
FLO-MIN111 SQK-RBK004 included dna_r10.3_450bps_hac
FLO-MIN111 SQK-RLB001 included dna_r10.3_450bps_hac
FLO-MIN111 SQK-LWB001 included dna_r10.3_450bps_hac
FLO-MIN111 SQK-PBK004 included dna_r10.3_450bps_hac
FLO-MIN111 SQK-RAB201 included dna_r10.3_450bps_hac
FLO-MIN111 SQK-RAB204 included dna_r10.3_450bps_hac
FLO-MIN111 SQK-RPB004 included dna_r10.3_450bps_hac
FLO-MIN111 VSK-VMK001 included dna_r10.3_450bps_hac
FLO-MIN111 VSK-VMK002 included dna_r10.3_450bps_hac
[...]

In the case of kits which come with their own barcodes included, the barcoding column will specify "included". Reads which have been prepared with these kits will be able to be demultiplexed using guppy_barcoder (see below).

Choosing a config file for Guppy

Config files are named as follows:

<strand type>_<pore version>_<speed>_<custom tags>.cfg

For example, a config file for high accuracy DNA basecalling on an R9.4.1 pore at 450 bases per second would be called dna_r9.4.1_450bps_hac.cfg. It may not be clear what pore and speed you are using, which is why directly choosing a config file is generally left to expert users. All other users should select flow cell and kit instead.

$1D^2$ basecalling configs have a custom tag with the basecalling method, so a $1D^2$ DNA basecall on an R9.5 pore at 450 bases per second would use the config file dna_r9.5_450bps.cfg.

CPU/GPU basecalling usage

There are two parameters that govern how many CPU threads Guppy uses: callers and CPU threads per caller.

When performing GPU basecalling, there is always one CPU support thread per GPU caller, so the number of callers (--num_callers) dictates the maximum number of CPU threads used. Modifying the number of CPU threads per caller (--num_cpu_threads_per_caller) will have no effect.

When performing CPU basecalling both callers and threads per caller may be set, making the maximum number of CPU threads used equal to num_callers * cpu_threads_per_caller.

The number of CPU threads used should generally not exceed either of these two values:

- The number of logical CPU cores your machine has (as there will probably not be sufficient computational power available for Guppy to run any faster than this).
- When performing CPU basecalling, more than the number of CPU threads your machine's RAM can support:
  - 4GB + 1GB per CPU thread for 1D basecalling
  - 4GB + 2GB per CPU thread for $1D^2$ basecalling

So if your machine has 8 GB of RAM then you can support:

- A maximum of 4 CPU threads for 1D basecalling
- A maximum 2 CPU threads for $1D^2$ basecalling

This assumes your machine is not performing any other computationally-intensive tasks except for using Guppy (e.g. it assumes you are not running MinKNOW).

Resuming runs

If a run of the Guppy basecaller is interrupted for some reason, it is possible to use the `--resume` option to attempt to re-start the basecall from where it was halted. This is useful if basecalling fails during processing particularly large batches of files. Resume should be used with exactly the same parameters as the previous run, or undefined behaviour may occur. If the `--resume` option is specified, the following steps occur:

- The basecaller checks the output directory to find log files from any previous runs
- The log files are interrogated to discover any successfully completed reads (and their source files) from previous runs
- Any files in the output directory, which do not belong to successfully completed reads, are removed (i.e. reads which were partially completed)
- The data for previously completed reads is extracted from the summary file for the previous run

The basecaller then proceeds as normal, filtering out any input reads which were previously processed.

After resumption of a basecall run, a single summary file will have been produced with all reads from the input folder in it, as if the run was completed normally.

Note: It is permissible to chain resume operations together, and it is permissible to resume from a successfully completed operation. This allows the resume functionality to be used to re-basecall an input folder in order to basecall just the read files which have appeared in that folder since the last basecall operation was invoked on it.

The resume system works by batching reads internally, and recording to the logfile when those batches have been completed and written to disk. The `--read_batch_size` argument can be set to control the size of these batches, and controls the granularity at which resume operations can occur. Increasing the batch size will reduce the fragmentation of output FASTQ files but can increase the amount of time a resume operation takes, as more previously basecalled reads may be re-called, because their batch was not completed.

Input and output files

Input files

Read .fast5 files, used as input to the basecalling software, must contain raw data. Raw data is included by default in .fast5 files generated by the MinKNOW software. Make sure you are using recent .fast5 files from the latest version of MinKNOW, as older files may not basecall properly with the set-out models and parameters provided in stand-alone Guppy.

Both the alignment and barcoding software accept FASTQ files as input. These can be generated either by the Guppy basecallers or by the MinKNOW software.

Output file size

If you start with a .fast5 file that **only** has raw data in it, then rough benchmarking suggests file size will increase in the following fashion:

- If you perform 1D basecalling, file size increases to roughly 2X original size.
- $1D^2$ basecalling only produces FASTQ files, which are significantly smaller than .fast5 files.

Folder structure

If using a version of MinKNOW which outputs reads in separate subfolders, it is necessary to use the `--recursive` option listed above to search through them to find .fast5 files.

For example, if MinKNOW's output folder structure looks like this:

```
minknow_output_folder/
--- 0/
| --- file1.fast5
| --- file2.fast5
| [...]
--- 1/
| --- file10.fast5
| --- file11.fast5
| [...]
```

Then calling Guppy as follows will search through the numbered subfolders for .fast5 files:

```
guppy_basecaller --input_path minknow_output_folder --recursive [...]
```

Output formats

Guppy supports outputting FASTQ files, and optionally .fast5, via the `--fast5_out` argument. FASTQ files may contain multiple reads per file, according to the `--records_per_fastq` argument. Each FASTQ file contains only reads from a specified run.

In the case where `--records_per_fastq` is set to:

- **1** - only a single read will be output per file
- **0** - all reads will be written into a single file (per caller)

The default FASTQ header is:

```
{read_id} runid={run_id} read={read_number} ch={channel_id} start_time={start_time_utc}
```

Where the unique id is the `read_id`.

Contents of the output folder

The save path will have the following structure once Guppy has finished running:

- `guppy_basecaller_<time_and_date>.log` A log file of what Guppy did during this basecall session.
- `sequencing_summary.txt` A tab-delimited text file containing useful information for each read analysed during this Guppy basecall.
- `workspace` If .fast5 output was enabled, a folder named `workspace` will be created, containing read .fast5 files basecalled by

Guppy. .fast5 files are copies of the files found in the folder provided as the --input_path argument which have been modified to contain the results of the Guppy basecall, in addition to whatever they had before. For example, if the read files already contained MinKNOW local basecalling results, those results will be preserved and additional Guppy basecalls will be added.

- FASTQ files and folders. The structure and naming of these is dependent on whether the optional argument --nested_output_folder was used (see below).

Default output of FASTQ files

By default (where the --nested_output_folder parameter is missing), the FASTQ filename is:

```
fastq_runid_{run_id}_{batch_counter}.fastq
```

Where batch_counter increments from zero.

A collection of FASTQ files will be emitted to the save folder containing the basecall results. Each FASTQ file may contain many reads. A set of FASTQ files will be generated for each run ID in the input file set. Additionally, depending on the records_per_fastq setting, a single run ID may generate multiple FASTQ files.

Note that the FASTQ files in the output folder and the .fast5 files in the workspace sub-folder may be separated into 'pass', 'fail', and 'calibration_strands' folders, depending on whether they pass or fail the filtering conditions or whether they have been identified as a calibration strand. This behaviour may be controlled with the --qscore_filtering and --calib_detect options. For example, if both options are enabled, the output folder structure would look like this:

```
guppy_output_folder/
--- pass/
| fastq_runid_777_0.fastq
| fastq_runid_abc_0.fastq
| fastq_runid_abc_1.fastq
--- fail/
| fastq_runid_777_0.fastq
--- calibration_strands/
| fastq_runid_777_0.fastq
```

Whereas turning both options off would produce a folder layout like this:

```
guppy_output_folder/
| fastq_runid_777_0.fastq
| fastq_runid_abc_0.fastq
| fastq_runid_abc_1.fastq
```

Guppy will not empty the save path before writing the output, but it will overwrite existing FASTQ files.

Nested folder output of FASTQ files

If the --nested_output_folder flag was supplied on the command line, then the FASTQ files will be named and emitted to a folder structure in the same manner as that used by MinKNOW, i.e.

```
{guppy_output_folder}/
|
{experiment_id}/
|
{sample_id}/
|
```

{date_time}_{device_id}_{flowcell_id}_{short_protocol_run_id}/
|
fastq{basecall_status}/
|
{barcode_arrangement}/
|
{flow_cell_id}{basecall_status}_{barcode_arrangement}_{short_run_id}_{batch_counter}.fastq

Where {basecall_status} is one of "_pass", "_fail" or "_calibration_strand". If --qscore_filtering or --calib_detect are disabled, then this field may be empty.

.fast5 output data

The specifications for output .fast5 files are listed in the ont_h5_validator GitHub repository. The contents of this output format are generally intended for expert users and is not extensively documented at the moment.

Note: If the software cannot produce a sensible basecall for a read, it will not add any additional information for that read into the read's .fast5 file.

Ping information

Guppy collects high-level summary information when it is used, and by default this information is sent over your internet connection to Oxford Nanopore Technologies. This is important information that allows us to analyse the performance of Guppy and identify areas where we need to improve. Nothing specific about the genomic content of individual reads is included - only generic information is logged, such as sequence length and q-score, aggregated over all the reads processed by Guppy. The sending of this summary information can be turned off if desired by providing the --disable_pings option to Guppy.

Guppy collects this high-level summary information as follows:

- Individual reads are added to an aggregator as they are basecalled
- The summary ping(s) are written out to a file (.js)
- If not disabled, the summary ping(s) are sent to Oxford Nanopore

This type of information is collected:

- General information about the configuration of Guppy and the run(s) that the data came from:
  - the options provided to Guppy
  - the total number of reads seen, and those seen per channel
- 1D basecalling information:
  - the numbers of reads which passed or failed basecalling
  - the average sequence length
  - the distribution of mean q-scores
  - the distribution of basecalling speeds
- $1D^2$ basecalling information:
  - the numbers of reads which passed or failed basecalling
  - the number of potential $1D^2$ candidate reads which were seen
  - the distribution of mean q-scores

Users are encouraged to browse the summary_telemetry.js file if they wish to see exactly what information Guppy is aggregating for telemetry.

Summary file contents

Guppy produces a summary file named `sequencing_summary.txt` during basecalling, which contains high-level information on every read analysed by the basecaller. This file is a tab-delimited text file which can be imported into common spreadsheet applications such as Excel or LibreOffice Calc, or read by software libraries such as NumPy or Pandas. Every read that is sent to the basecaller will have an entry in the summary file, regardless of whether or not that read was successfully basecalled.

When enabling extra functionality such as barcoding or alignment, additional columns will be added to the summary file. For this reason, and because the columns may occasionally be re-ordered, it is recommended that specific columns are accessed by their name (e.g. the `read_id` column) instead of the order in which they occur in the file.

Below is a list of summary file columns with a description of their contents. Very occasionally new columns may be added to the file without being described here; these columns should be considered unreliable and subject to change or removal.

- **filename** The name of the .fast5 file the read came from.
- **read_id** The uuid that uniquely identifies this read.
- **run_id** The uuid that uniquely identifies the sequencing run that this read came from.
- **batch_id** Integer identifier of the batch that Guppy put this read in. See the `--read_batch_size` parameter and the `--resume` option.
- **channel** The channel on the flow cell that the read came from.
- **mux** The mux in the channel that the read came from.
- **start_time** Start time of the read, in seconds since the beginning of the run.
- **duration** Duration of the read, in seconds.
- **num_events** Legacy field -- **duration** should be used instead.
- **passes_filtering** Whether or not the read passed the qscore filter. See the `--qscore_filtering` flag and the `--min_qscore` parameter.
- **template_start** Start time of the portion of the read that was sent to the basecaller after adapter trimming, in seconds since the beginning of the run. See the `--trim_threshold`, `--trim_min_events`, `--max_search_len`, `--trim_strategy`, and `--dmean_win_size` parameters.
- **num_events_template** Legacy field -- **template_duration** should be used instead.
- **template_duration** Duration of the portion of the read that was sent to the basecaller after adapter trimming, in seconds.
- **sequence_length_template** Number of bases in the output sequence, taking into account any sequence trimming performed by options such as `--trim_barcodes`.
- **mean_qscore_template** The qscore corresponding to the mean error rate of the sequence.
- **strand_score_template** Legacy field - no longer populated reliably.
- **median_template** The median current of the read, in pA.
- **mad_template** The median absolute deviation of the current of the read, in pA.
- **scaling_median_template** The "median_template" value used by the basecaller to scale incoming data. May be different than **median_template** if adapter scaling or scaling overrides are used. See the `--scaling_med` parameter.
- **scaling_mad_template** The "mad_template" value used by the basecaller to scale incoming data. May be different than **mad_template** if adapter scaling or scaling overrides are used. See the `--scaling_mad` parameter.

If barcoding/demultiplexing is enabled via the `--barcode_kits` argument, then the following columns are added to the sequencing summary file:

- **barcode_arrangement** The normalized name of the barcode classification, without a kit (e.g. "barcode01"), or "unclassified" if no classification could be made.
- **barcode_full_arrangement** The full name for the highest-scoring barcode match, including kit, variation, and direction (e.g. "RAB19_var2").
- **barcode_kit** The kit name belonging to the highest-scoring barcode match (e.g. "RAB").
- **barcode_variant** Which of the forward / reverse variants the highest-scoring barcode matched (e.g. "var1"), or "n/a" if no variants are available.
- **barcode_score** The score for either the front or rear barcode, whichever is higher. The maximum score is 100, with no minimum.

- **barcode_front_id** The full name for the barcode at the front of the strand, including direction (forward/reverse) and variant (1st/2nd) (e.g. "RAB19_2nd_FWD").
- **barcode_front_score** The score for the barcode at the front of the strand.
- **barcode_front_refseq** The reference sequence the barcode at the front of the strand was matched against.
- **barcode_front_foundseq** The sequence of the barcode at the front of the strand that matched barcode_front_refseq.
- **barcode_front_foundseq_length** The length of barcode_front_foundseq.
- **barcode_front_begin_index** The position in the called sequence, counting from the beginning, that barcode_front_foundseq begins at.
- **barcode_rear_score** The score for the barcode at the rear of the strand.
- **barcode_rear_refseq** The reference sequence the barcode at the rear of the strand was matched against.
- **barcode_rear_foundseq** The sequence of the barcode at the rear of the strand that matched barcode_rear_refseq.
- **barcode_rear_foundseq_length** The length of barcode_rear_foundseq.
- **barcode_rear_end_index** The position in the called sequence, counting backwards from the end, that barcode_rear_foundseq ends at.

If barcode trimming is enabled via the --trim_barcodes argument, then the following additional columns will also be present:

- **barcode_front_total_trimmed** The number of bases removed from the front of the sequence as part of barcode trimming.
- **barcode_rear_total_trimmed** The number of bases removed from the rear of the sequence as part of barcode trimming.

If dual barcoding is used the following additional columns will be present:

- **barcode_front_id_inner**
- **barcode_front_score_inner**
- **barcode_rear_id_inner**
- **barcode_rear_score_inner**

These columns have the same meaning as the standard "id" and "score" columns above, but apply only to the inner front and rear barcodes. The standard "id" and "score" columns now apply to the outer barcodes.

For further details on how barcoding works see the "How barcode demultiplexing works" section.

If alignment is enabled via the --align_ref argument, then the following colums are added to the sequencing summary file:

- **alignment_genome** The name of the reference which the read aligned to, or "*" if no alignment was found.
- **alignment_genome_start** The position in the reference where the alignment started, or -1 if no alignment was found.
- **alignment_genome_end** The position in the reference where the alignment ended, or -1 if no alignment was found.
- **alignment_strand_start** The position in the called sequence where the alignment started, or -1 if no alignment was found.
- **alignment_strand_end** The position in the called sequence where the alignment ended, or -1 if no alignment was found.
- **alignment_num_insertions** The number of insertions in the alignment, or -1 if no alignment was found.
- **alignment_num_deletions** The number of deletions in the alignment, or -1 if no alignment was found.
- **alignment_num_aligned** The number of bases in the called sequence which aligned to bases in the reference, or -1 if no alignment was found.
- **alignment_num_correct** The number of aligned bases in the called sequence which match their corresponding reference base, or -1 if no alignment was found.
- **alignment_identity** The percentage of aligned bases which correctly match their corresponding reference base (**alignment_num_correct**/**alignment_num_aligned**), or -1 if no alignment was found.
- **alignment_accuracy** The percentage of all bases in the alignment which are correct (**alignment_num_correct**/(**alignment_num_aligned** + **alignment_num_insertions** + **alignment_num_deletions**)), or -1 if no alignment was found.
- **alignment_score** The score returned by minimap2, or -1 if no alignment was found.
- **alignment_coverage** The percentage of either the called sequence or the reference (whichever is shorter) that aligns (e.g. (**alignment_strand_end** - **alignment_strand_start** + 1)/(**sequence_length_template**), or -1 if no alignment was found.
- **alignment_direction** The direction of the alignment, either forwards (+) or reverse (-), or "' *if no alignment was found. Note*

*that genome positions (e.g. **alignment_genome_start**) are always given in the forwards direction.*

Guppy basecall server

Guppy basecall server

Guppy includes an additional executable called `guppy_basecall_server` which provides basecalling as a network-enabled service. The basecall server may be useful in situations where a set of compute resources such as GPUs need to be shared between several concurrently-running basecalling clients. It is enables client applications to perform basecalling by communicating with the server via the ZMQ socket interface. ONT products which support multiple flow cells typically use Guppy in a server configuration in order to share the embedded GPUs between all flow cells.

The server is launched as follows:

```
guppy_basecall_server --config <config file> --log_path <log file folder> --port 5555
```

The basecall server requires a basecalling config file, just the same as the standalone basecaller. It also requires a `--log_path` to be specified, which will be used to output the server execution log. The final required parameter is `--port` which specifies which socket port the server will listen for connections on.

On startup the server will output something similar to the following:

```
ONT Guppy basecall server software version 3.0.3+7e7b7d0
config file: /opt/ont/guppy/data/dna_r9.4.1_450bps_fast.cfg
model file: /opt/ont/guppy/data/template_r9.4.1_450bps_fast.jsn
log path: /tmp
chunk size: 1000
chunks per runner: 48
max queued reads: 2000
num basecallers: 1
num socket threads: 1
gpu device: cuda:0
kernel path:
runners per device: 2

Starting server on port: 5555
```

The server may take a few seconds to fully launch, but once the "Starting server" line is output, the server is ready for connections. Once the server is running, it can be used to basecall by running the Guppy basecaller in client mode. This is exactly the same as launching the Guppy basecaller locally, except a connection port is specified:

```
guppy_basecaller --input_path reads --save_path output_folder/basecall --config dna_r9.4.1_450bps_fast.cfg --port 5555
```

If only a port is specified to the Guppy basecaller as above, Guppy will assume the server is running on the local host. However, it is also possible to specify an address or hostname:

```
guppy_basecaller --input_path reads --save_path output_folder/basecall --config dna_r9.4.1_450bps_fast.cfg --port 192.168.0.64:5555
```

or

```
guppy_basecaller --input_path reads --save_path output_folder/basecall --config dna_r9.4.1_450bps_fast5.cfg --port my_basecall_server:5555
```

In this case, the connection can be made to a remote server.

Note: Basecalling performance may be compromised by network bandwidth when using a remote server. It is possible for multiple clients to connect to a basecall server simultaneously and the server will distribute processing resource between them using a fair queuing system.

Note: Read trimming and file output will be performed on the client, so any parameters to control those steps must be specified when launching the client, not the server.

Basecall server-specific parameters

To start the basecall server you will need to specify the path for logging.

- Logging path (`--log_path`): The path to the folder to save a basecall log. The logs contain all the messages that are output to the terminal, plus additional informational messages. For example, the log will contain a record for each input file which is loaded and each file which is written out. Any error or warning messages generated during the run will also go in the log, which can be used for diagnosing problems. If the user specifies the `--verbose` flag, an additional verbose log file is written out. The `log_path` is only set for the server (as it has no other output files), but the `guppy_basecaller` app also emits logs, which go into the `save_path`
- Maximum queue size (`--max_queued_reads`): Maximum number of reads to queue per client. When running in client/server mode, the client will load files from disk and send them immediately to the server for basecalling. If the client can load and send reads faster than the basecaller can process them, queued reads will pile up on the basecall server, increasing memory consumption. To avoid this problem, `--max_queued_reads` specifies a maximum number of reads that an individual client can have in flight on the server at once. This has a default value of 2000, which is sufficient for MinION Mk1B and GridION setups with a single client attached. When running multiple clients, the number should be reduced to prevent excessive memory usage.

Guppy basecall supervisor

Guppy includes an executable called `guppy_basecaller_supervisor`.

A single `guppy_basecaller` running as a client will struggle to read files fast enough to supply a`guppy_basecall_server` especially in a multiple GPU system. In order to improve the GPU utilization it is necessary to have multiple clients connecting to the basecall server. The `guppy_basecall_supervisor` application is provided in order simplify the process of connecting multiple clients to a server while all reading from the same input location and writing to the same save path.

This supervisor application ensures that:

1. All files from the input location are distributed amongst the child basecaller clients, and
2. Each client is launched with a unique client_id guaranteeing all files written to the save folder will be uniquely named.

Once all basecaller clients have completed the supervisor exits, a return code of zero indicating success.

Usage

The basecaller supervisor is launched with exactly the same parameters as the Guppy basecaller running in client mode, but with the addition of a `--num_clients` parameter.

For example, to launch three Guppy basecallers running in client mode all processing the same input location and writing to the same save location:

(The following assumes that the basecall server has already been launched and is listening on port 5555)

```
guppy_basecaller_supervisor --num_clients 5 --input_path reads --save_path ./save_folder/ --config dna_r9.4.1_450bps_fast.cfg --port 5555
```

Note: the output will be written by each client individually and is not merged. In particular, it is worth noting that there

will be one sequencing summary per client.

Depending on your requirements some further processing may be necessary in order to merge the sequencing summary files.

Example output files:

```
--- /save_folder/
| fastq_runid_6dce0a5_client0_0_0.fastq
| fastq_runid_6dce0a5_client1_0_0.fastq
| fastq_runid_6dce0a5_client2_0_0.fastq
| guppy_basecaller_0_log-2019-11-25_15-11-53.log
| guppy_basecaller_1_log-2019-11-25_15-11-53.log
| guppy_basecaller_2_log-2019-11-25_15-11-53.log
| guppy_basecaller_supervisor_log-2019-11-25_15-11-53.log
| sequencing_summary_0.txt
| sequencing_summary_1.txt
| sequencing_summary_2.txt
| sequencing_telemetry_0.js
| sequencing_telemetry_1.js
| sequencing_telemetry_2.js
```

Command-line configuration arguments

Any configuration parameters currently passed to the guppy_basecaller, e.g. --num_callers, --ipc_threads, --gpu_runners_per_device, --chunks_per_runner, etc., should also be suitable for the guppy_basecall_supervisor as these will be directly forwarded to the clients.

To choose an optimum value for the --num_clients parameter, some trial and error is necessary, for example start with num_clients 1 and increase until no further benefit is noticed. The output from the supervisor may well be useful in determining this as it reports the samples/second, i.e.

```
Caller time: 5405 ms, Samples called: 186589921, samples/s: 3.45217e+07
```

For more detailed metrics, the --progress_stats_frequency argument can be used, although this reports bases called/second as opposed to samples. Below is some sample output with progress_stats_frequency 5

```
Found 38 fast5 files to process.
Processing ...
[PROG_STAT_HDR] time elapsed(secs), time remaining (estimate), total reads processed, total reads (estimate), interval(secs), interval reads processed, interval bases processed, bases/sec
[PROG_STAT] 5.00439, 10.8428, 12, 38, 5.00439, 12, 66073, 13203.0
[PROG_STAT] 10.0091, 8.10263, 21, 38, 5.00466, 9, 61161, 12220.8
[PROG_STAT] 15.0133, 1.76627, 34, 38, 5.0041, 13, 71410, 14270.3
[PROG_STAT] 17.1152, 0, 38, 38, 2.10173, 4, 35785, 17026.5
Caller time: 17530 ms, Samples called: 2157249, samples/s: 123060
All instances of guppy_basecaller completed successfully.
```

Notes

- When launching the Guppy basecaller supervisor, do not use the parameters --client_id or --input_file_list, as these will be used by the supervisor itself to manage the Guppy basecaller clients.
- The intended usage is that the child Guppy basecallers are run as as clients, therefore it is necessary to supply the --port argument.
- Since the child Guppy basecallers are run as as clients, the --device argument should NOT be supplied.

Expert settings

Parameters for expert users

There are additional advanced options for expert users. Experimenting with these parameters may significantly impact the performance or accuracy of the basecaller:

Data features

- Calibration strand reference file (--calib_reference): Provide a FASTA file to override the reference calibration strand.
- Calibration strand candidate minimum sequence length (--calib_min_sequence_length): Minimum sequence length for reads to be considered candidate calibration strands.
- Calibration strand candidate maximum sequence length (--calib_max_sequence_length): Maximum sequence length for reads to be considered candidate calibration strands.
- Calibration strand minimum coverage (--calib_min_coverage): Minimum reference coverage of candidate strand required for a read to pass calibration strand detection.
- DNA Adapter trimming threshold (--trim_threshold): Threshold above which data will be trimmed (in standard deviations of current level distribution).
- DNA Adapter trimming minimum events (--trim_min_events): Adapter trimmer minimum stride intervals after stall that must be seen.
- DNA Adapter trimming maximum search length (--max_search_len): Maximum number of samples from the beginning of the read to search through for the stall.
- Override automatic read scaling (--override_scaling): Flag to manually provide scaling parameters rather than estimating them from each read. See the --scaling_med and --scaling_mad options below.
- Manual read scaling median (--scaling_med): Median current value to use for manual scaling.
- Manual read scaling median absolute deviation (--scaling_mad): Median absolute deviation to use for manual scaling.
- Adapter Trimming strategy (--trim_strategy): Trimming strategy to apply to the raw signal before basecalling (must be one of dna, rna or none). The adapter looks different in the signal depending on whether DNA or RNA is being basecalled, so the two cases require a different adapter trimming algorithm. This should be set automatically by the config file, and usually it is not required to set this at the command line.
- RNA Adapter Trimming Window size (--dmean_win_size): Window size for coarse stall event detection. This parameter,--dmean_threshold and –jump_threshold are used to override how the RNA adapter trimming code operates. Generally, users should not need to change these unless they are familiar with how RNA adapter trimming works.
- RNA Adapter Trimming threshold (--dmean_threshold): Threshold for coarse stall event detection.
- RNA Adapter Trimming jump threshold (--jump_threshold): Threshold level for RNA stall detection.
- Disable event table transmission (--disable_events): Flag to disable the transmission of event tables when receiving reads back from the basecall server. If the event tables are not required for downstream processing (e.g. for 1D^2) then it is more efficient to disable them.
- Enable poly-T/non-sequence adapter-based read scaling (--pt_scaling): Flag to enable polyT/adapter max detection for read scaling. This will be used in preference to read median/median absolute deviation to perform read scaling if the poly-T to non-sequence adapter current level change can be detected.
- Poly-T scaling median offset (--pt_median_offset): Set polyT median offset for setting read scaling median (default 2.5)
- Poly-T scaling range scale (--adapter_pt_range_scale): Set polyT/adapter range scale for setting read scaling median absolute deviation (default 5.2)
- Poly-T scaling minimum adapter drop (--pt_required_adapter_drop): Set minimum required current drop from adapter max to polyT detection. (default 30.0)
- Poly-T scaling minimum read start index (--pt_minimum_read_start_index): Set minimum index for read start sample required to attempt polyT scaling. (default 30)
- Read ID whitelist (--read_id_list): A filename for a text file containing a whitelist of read IDs (one per line, no whitespace). If this option is specified, Guppy will only basecall reads from the input which have read IDs that are in the read whitelist.
- Barcoding configuration file (--barcoding_config_file): A filename from which to load the barcoding configuration, allowing users to override all barcoding parameters without specifying them at the command line. Defaults to 'configuration.cfg'.

Optimisation

- Model file (`-m` or `--model_file`): A path to a JSON RNN model file to use instead of the model specified in the configuration file.
- Chunk size (`--chunk_size`): Set the size of the chunks of data which are sent to the basecaller for analysis. Chunk size is specified in signal blocks, so the total chunk size in samples will be `chunk_size * event_stride`.
- Chunk overlap (`--overlap`): The overlap between adjacent chunks, specified in signal blocks. An overlap is required for chunks to be stitched back into a continuous read.
- Max chunks per runner (`--chunks_per_runner`): The maximum number of chunks which can be submitted to a single neural network runner before it starts computation. Increasing this figure will increase GPU basecalling performance when it is enabled.
- Number of GPU runners per device (`--gpu_runners_per_device`): The number of neural network runners to create per CUDA device. Increasing this number may improve performance on GPUs with a large number of compute cores, but will increase GPU memory use. This option only affects GPU calling.
- CPU threads per caller (`--cpu_threads_per_caller`): The number of CPU threads to create for each caller to use. Increasing this number may improve performance on CPUs with a large number of cores, but will increase system load. This option only affects CPU calling.
- Stay penalty (`--stay_penalty`): Scaling factor to apply to stay probability calculation during transducer decode.
- Q-score offset (`--qscore_offset`): Override the q-score offset to apply when calibrating output q-scores for the read. There is an offset and scale (see `--qscore_scale` below) that are applied to the output base probabilities in the FASTQ for a basecall, to make the q-scores as close as possible to the Phred quality scores. Once a basecall model has been trained, these scores are calculated and added to the config files.
- Q-score scale (`--qscore_scale`): Override the q-score scale to apply when calibrating output q-scores for the read.
- Use built-in GPU kernels (`--builtin_scripts`): Set this flag to false to disable built-in GPU kernels, allowing custom kernels to be used (see `--kernel_path`).
- GPU Kernel source path (`--kernel_path`): Path to GPU kernel files, which will be used if `--builtin_scripts` is set to false.
- Trace logging (`--trace_categories_logs`): Enable logging of the specified trace categories. Specify a comma-separated list.
- Disable pings (`--disable_pings`): Flag to disable sending any telemetry information to Oxford Nanopore Technologies. See the "Ping information" section for a summary of what is included in the Guppy telemetry.
- Telemetry URL (`--ping_url`): Override the default URL for sending telemetry pings.
- Ping segment duration (`--ping_segment_duration`): Duration in minutes of each ping segment.
- Read batch size (`--read_batch_size`): The maximum batch size, in reads, for grouping input files. This controls the granularity at which resume can operate.

Using unsupported NVIDIA GPUs

Newer GPUs may require more up-to-date versions of NVIDIA drivers and CUDA APIs than the ont-guppy deb recommends. It is possible to install the ont-guppy deb without installing any GPU drivers by not installing the recommended ones, leaving it to the user to ensure that their system has the required minimum drivers:

```
apt-get install ont-guppy --no-install-recommends
```

In general NVIDIA APIs and drivers are backwards compatible, so it will usually be safe to install a later / higher version than Guppy requires. Check for the minimum driver version that Guppy requires using `apt-cache show`:

```
$ apt-cache show ont-guppy
Package: ont-guppy
Version: 0.0.0-1~xenial
Architecture: amd64
Depends: libc6 (>=2.23), libcurl4-openssl-dev, libssl-dev, libhdf5-cpp-11, libzmq5, libboost-atomic1.58.0, libboost-chrono1.58.0, libboost-date-time1.58.0, libboost-filesystem1.58.0, libboost-program-options1.58.0, libboost-regex1.58.0, libboost-system1.58.0, libboost-log1.58.0
Recommends: nvidia-384, libcuda1-384
[...]
```

In this example the driver version (384) can be seen as part of the `nvidia` and `libcuda1` package names in the "Recommends" section. Driver versions higher than 384 will likely be compatible with this version of Guppy.

Note that Guppy only supports GPUs with an NVIDIA compute version of 6.1 or higher.

Setting custom GPU parameters in Guppy

If using an unsupported GPU then the following calculation provides a rough ceiling to the amount of GPU memory that Guppy will use:

```
runners * chunks_per_runner * chunk_size < 100000 * [max GPU memory in GB]
```

For example, a GPU with 8GB of memory would require:

```
runners * chunks_per_runner * chunk_size < 800000
```

Overriding configuration parameters from the command-line

Guppy configuration files specify many of the optional parameters discussed previously. For example, the basecalling section of a configuration file could look like this:

```
Basic configuration file for ONT Guppy basecaller software.
Basecalling.
model_file = template_r9.5_450bps_5mer_raw.jsn
chunk_size = 1000
runners = 20
chunks_per_runner = 20
overlap = 50
qscore_offset = -0.06
qscore_scale = 1.16
builtin_scripts = 1
```

The parameters specified in the configuration file can be overwritten from the command-line by arguments of the form `--parameter value`, e. g.

```
guppy_basecaller --config dna_r9.5_450bps.cfg --runners 40 [other options]
```

Command-line parameters always take priority over config file parameters, so running Guppy with these arguments would override the `runners` setting from the config file, forcing it to 40. This facilitates small changes to parameters. Please note that no spaces are allowed in arguments, but the argument can be wrapped in quotes. For example, to run Guppy with two GPU devices, you would set the devices like so:

```
guppy_basecaller --device "cuda:0 cuda:1" [other options]
```

Barcoding/demultiplexing

Barcoding/demultiplexing overview

In the Guppy suite, barcoding can be performed by a separate executable. This allows barcoding to be performed as an offline analysis step without having to re-basecall the source reads. To perform barcoding in this way, invoke the barcoder with the minimum required parameters:

```
guppy_barcoder --input_path <folder containing FASTQ and/or FASTA files> --save_path <output folder> --config configuration.cfg
```

When performing barcode detection, Guppy will create a `barcoding_summary.txt` file in the output folder, which contains information about the best-matching barcodes for each read in the FASTQ/FASTA files in the input folder (see "Summary file contents" in the "Input and output files" section for details). The output FASTQ/FASTA files will be written into barcode-specific subdirectories for the barcode detected. A log file is also emitted with information about the execution run.

The Guppy barcoder supports the following optional parameters:

- Version (`-v` or `--version`): Prints the version of Guppy barcoder.
- Help (`-h` or `--help`): Print a help message describing usage and all the available parameters.

Data features

- Require a barcode on both ends of the read (`--require_barcode_both_ends`): Option to only classify reads where a barcode has been detected at both the front and rear of the read. This can significantly reduce the number of reads that are classified, and is also not a valid argument for the Rapid kits (which do not have a rear barcode).
- Allow inferior barcodes to be used in arrangements (`--allow_inferior_barcodes`): Option to still classify reads when the barcode selected at each end of the read was not the highest-scoring barcode detected (assuming one was detected above the minimum score). This can slightly increase the number of reads that are classified but can increase the false-positive rate in classifications.
- Front window size (`--front_window_size`): Specify the maximum window of the start of the read (in bases) to search for the front barcode in. The default is 150 bases.
- Rear window size (`--rear_window_size`): Specify the maximum window of the end of the read (in bases) to search for the rear barcode in. The default is 150 bases.
- Detect mid-strand barcodes (`--detect_mid_strand_barcodes`): Flag option to enable detection of barcodes within the strand. This option can be used to detect abnormal reads such as chimeras. If a mid-strand barcode is detected, the read will be classified as "unclassified".
- Minimum score for detection of mid-strand barcodes (`--min_score_mid_barcodes`): Minimum score to consider a barcode detected mid strand to be considered a valid alignment. Mid-strand barcodes below this threshold will be ignored. The default is 60.0.

Input/output

- Quiet mode (`-z` or `--quiet`): This option prevents the Guppy basecaller from outputting anything to stdout. Stdout is short for “standard output” and is the default location to which a running program sends its output. For a command line executable, stdout will typically be sent to the terminal window from which the program was run.
- Verbose logging (`--verbose_logs`): Flag to enable verbose logging (outputting a verbose log file, in addition to the standard log files, which contains detailed information about the application). Off by default.
- Recursive (`-r` or `--recursive`): search through all subfolders contained in the `--input_path` value, and perform barcode detection on any FASTQ or FASTA files found in them.
- Configuration file (`-c` or `--config`): This option allows you to specify a configuration file, which contains details of the barcoding arrangements to attempt to detect. The default cfg file supplied with Guppy should be sufficient for most users as it contains all the barcodes supplied in ONT barcoding kits. There is an additional configuration_dual.cfg containing settings for using dual-barcode preparations.
- Override default data path (`-d` or `--data_path`): Option to explicitly specify the path to use for loading any data files the application requires (for example, if you have created your own model files or config files).
- Records per FASTQ (`-q` or `--records_per_fastq`): The number of reads to put in a single FASTQ or FASTA file. Set this to zero to output all reads into one file (per run id, per caller). The default value is 4000.
- Perform FASTQ compression (`--compress_fastq`): Flag to enable gzip compression of output FASTQ/FASTA files; this reduces file size to about 50% of the original.

Optimisation

- Worker thread count (-t or --worker_threads): The number of worker threads to spawn for the barcoder to use. Increasing this number will allow Guppy barcoder to make better use of multi-core CPU systems, but may impact overall system performance.
- GPU device (-x or --device): Specify the CUDA-enabled GPU to use to perform barcode alignment. Parameters are specified the same way as in the basecaller application.
- Limit the kits to detect against (--barcode_kits): List of barcoding kit(s) or expansion kit(s) used to limit the number of barcodes to be detected against. This speeds up barcoding. Multiple kits must be a space-separated list in double quotes.
- Number of parallel GPU barcoding buffers (--num_barcoding_buffers): Number of parallel memory buffers to supply to the GPU for barcode strand detection. Greater numbers will increase parallelism on the GPU at an increased memory cost. The default is 16.

To see the supported barcoding kits, run the --print_kits argument with the barcoder:

```
guppy_barcoder --print_kits
```

To limit the kits to detect against:

```
guppy_barcoder --input_path <folder containing FASTQ and/or FASTA files> --save_path <output folder> --config configuration.cfg --barcode_kits SQK-RPB004
```

Or for multiple kits add a space-separated list in double quotes:

```
guppy_barcoder --input_path <folder containing FASTQ and/or FASTA files> --save_path <output folder> --config configuration.cfg --barcode_kits "EXP-NBD104 EXP-NBD114"
```

Barcoding of dual-barcode arrangements is also supported. To use dual-barcode arrangements, the correct configuration file must be specified:

```
guppy_barcoder --input_path <folder containing FASTQ and/or FASTA files> --save_path <output folder> --config configuration_dual.cfg --barcode_kits "EXP-DUAL00"
```

Note that running barcode detection on dual- and single- barcode kits at the same time is not currently supported. New columns will be emitted into the barcoding_summary.txt or sequencing_summary.txt when performing demultiplexing of dual barcode kits: barcode_front_id_inner, barcode_front_score_inner, barcode_rear_id_inner and barcode_rear_score_inner.

Barcoding during basecalling

It is also possible to perform barcode detection during the basecalling process. When invoking the guppy_basecaller executable, simply provide a valid set of kits to the barcode_kits argument to enable barcoding, for example:

```
guppy_basecaller --input_path <folder containing .fast5 files> --save_path <output folder> --config dna_r9.4.1_450bps_fast.cfg --barcode_kits SQK-RBK001
```

Note that options such as barcode trimming and demultiplexing output FASTQ/FASTA files are all supported by the guppy_basecaller executable as well as guppy_barcoder. Guppy also supports barcoding demultiplexing during basecalling when using the guppy_basecall_server. If a barcoding configuration file other than the default configuration.cfg is required, the basecaller executable supports selecting a barcode config using --barcoding_config_file command-line option.

Barcode FASTQ output

The barcoding executable will output FASTQ/FASTA files into barcode-specific subdirectories in the output folder depending on the barcode that was detected. The FASTQ naming follows the same rules as for basecalling (see "Guppy features, settings and analysis"). A barcode directory will only exist if the barcode was detected. The output structure will look like this:

```
guppy_output_folder/
| barcoding_summary.txt
--- barcode01/
| fastq_runid_777_0.fastq
| fastq_runid_abc_0.fastq
| fastq_runid_abc_1.fastq
--- barcode03/
| fastq_runid_777_0.fastq
| fasta_runid_xyz_0.fasta
--- unclassified/
| fastq_runid_777_0.fastq
```

Barcode trimming

The barcoding executable offers the user an option to trim the detected barcodes from the sequence before being output to the FASTQ/FASTA file. This is off by default. To enable barcode trimming add the `--trim_barcodes` argument:

```
guppy_barcoder --input_path <folder containing FASTQ and/or FASTA files> --save_path <output folder> --config configuration.cfg --trim_barcodes
```

Two extra columns will then be written into the `barcoding_summary.txt` output: `barcode_front_total_trimmed` and `barcode_rear_total_trimmed`. A barcode will only be trimmed if it is above the `min_score` threshold (default 60), and the aligned sequence that matches to the barcode will be removed from the front and/or rear of the sequence that is then written to the FASTQ/FASTA.

If the user wants to be more severe with trimming, there is a `--num_extra_bases_trim` argument, which defaults to 0. Setting this to, for example, 2 would trim the detected barcode sequence plus an extra 2 bases. If the user wants to be more cautious then give this argument a negative number; for example, -3 would trim 3 fewer bases than was detected as the barcode sequence.

```
guppy_barcoder --input_path <folder containing FASTQ and/or FASTA files> --save_path <output folder> --config configuration.cfg --trim_barcodes --num_extra_bases_trim 2
```

Expert users - adjusting barcode classification thresholds

The classification threshold has been chosen to produce a low number of incorrect classifications while retaining an acceptable classification rate. The user may override this, but note that small changes can have a significant effect on the false-positive rate, so it is important to always test any changes before using them.

To change the threshold used for both the front and rear barcode modify the `--min_score` argument. The following would increase the threshold for barcodes to be classified to 70, so that if either the front or rear barcode has a score of 70 or more the read will be classified:

```
guppy_barcoder --input_path <folder containing FASTQ and/or FASTA files> --save_path <output folder> --config configuration.cfg --min_score 70
```

The user may also have different front and rear thresholds by also supplying the `--min_score_rear_override` argument. If this is specified then `--min_score` will be used for the front barcode and `--min_score_rear_override` will be used for the rear barcode. For example, in the following a read will be classified if either the front barcode is above the default (which is currently 60), or the rear barcode 55 or more:

guppy_barcoder --input_path <folder containing FASTQ and/or FASTA files> --save_path <output folder> --config configuration.cfg --min_score_rear_override 55

How barcode demultiplexing works in Guppy

This is a general outline of how the Guppy barcoder works and how you can adjust its classification thresholds.

The regions of a barcode

A complete barcode arrangement comprises three sections:

- The **upstream flanking region**, which comes between the barcode and the sequencing adapter
- The **barcode sequence**
- The **downstream flanking region**, which comes between the barcode and the sample sequence

A complete dual-barcode arrangement comprises five sections:

- The **upstream flanking region**, which comes between the outer barcode and the sequencing adapter
- The **outer barcode sequence**
- The **mid flanking region**, which comes between the outer barcode and the inner barcode
- The **inner barcode sequence**
- The **downstream flanking region**, which comes between the inner barcode and the sample sequence

The barcode sequences remain constant across almost all of Oxford Nanopore Technologies' kits. For example, the flanking regions for barcode 10 in the Rapid Barcoding Kit (SQK-RBK004) are different from the flanking regions for barcode 10 in the native barcoding expansion kit (EXP-NBD114), but the barcode sequence itself is the same.

While native kits use the same barcode sequences as other kits, barcodes 1-12 in the native kit are the reverse complement of the standard barcodes 1-12.

There is one other exception to this: barcode 12a in the Rapid Barcoding Kit SQK-RBK004 has a different barcode sequence to barcode 12 in other kits. For this reason, the oligonucleotide of this sequence is referred to as "barcode 12a".

Different barcoding chemistries

While each barcoding chemistry type (e.g. native, rapid, or PCR) will produce barcodes with the pattern described in "The regions of a barcode", there can be variations in the flanking regions within a particular kit. These are referred to as either "forward" and "reverse" variations or "variation 1" and "variation 2" depending on the configuration. When these variations are present the full double-stranded sequence can look like this:

<barcodeXX_var1---><sample sequence top strand---><barcodeXX_var2_rc>

<barcodeXX_var1_rc><sample sequence bottom strand><barcodeXX_var2--->

The PCR Barcoding Expansion kit (EXP-PBC001) produces barcodes like the example directly above.

Or like this:

<barcodeXX_var1><sample sequence top strand---><barcodeXX_var2>

<barcodeXX_var2><sample sequence bottom strand><barcodeXX_var1>

The Native Barcoding Expansion kit (EXP-NBD114) produces barcodes like the second example, directly above.

The barcoding algorithm

The barcoding algorithm uses a modified Needleman-Wunsch method. We modify the Needleman-Wunsh algorithm by adding "gap open" and "gap extension" penalties, as well as separate "start gap" and "end gap" penalties. These penalties and the match / mismatch scores for aligning a barcode to a sequence are detailed in two places:

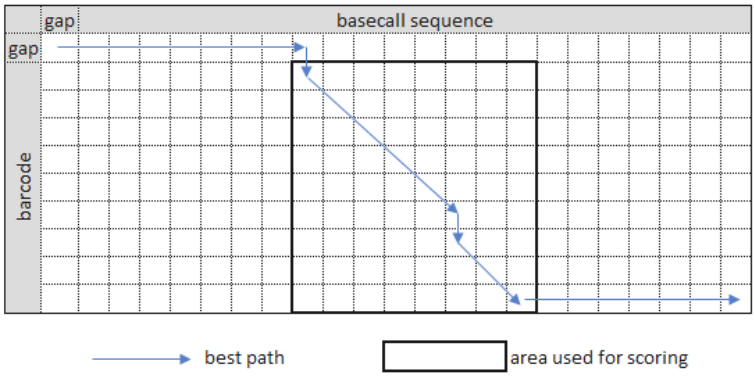
Generic gap penalties are in the barcoding configuration file configuration.cfg, or configuration_dual.cfg for dual-barcode arrangements.

DNA-specific match/mismatch scores are stored in the file 4x4_mismatch_matrix.txt. Note that these scores are shifted such that the highest score is 100 – this means that the final barcode score will share the same maximum. There is also a 5x5_mismatch_matrix.txt file which includes the ability to match any cardinal base to a mask base 'N'.

Each barcode is aligned to a section of the basecall, usually the first and / or last 150 bases. This generates a grid of size 150 * < barcode_length >.

The barcoding score for a particular grid is calculated in a two-step process:

The score for only the section of the grid that corresponds to the barcode itself is considered. This corresponds to removing the initial gap row and discarding all scores past the alignment of the last base of the barcode, or removing those sections where the "start gap" and "end gap" penalties are applied.



The score is normalized by the total length of the barcode sequence. This ensures the final score is no more than the highest score in the mismatch table (which should be 100). Note that this potentially allows for negative scores when there are a relatively high number of gaps and/or mismatches.

Measuring classification

The classification for a particular barcode is determined by comparing the barcoding score to a fixed classification threshold – scores that exceed the threshold are considered (successful) classifications. The current threshold is set to 60 for single barcode arrangements and 50 for dual barcode arrangements.

Classification for a read is determined by taking the single highest-scoring (successful) barcode classification. This includes both classifications made at the beginning of the sequence and (where applicable) the end. If no classification exists then the read is considered "unclassified".

Classification threshold criteria

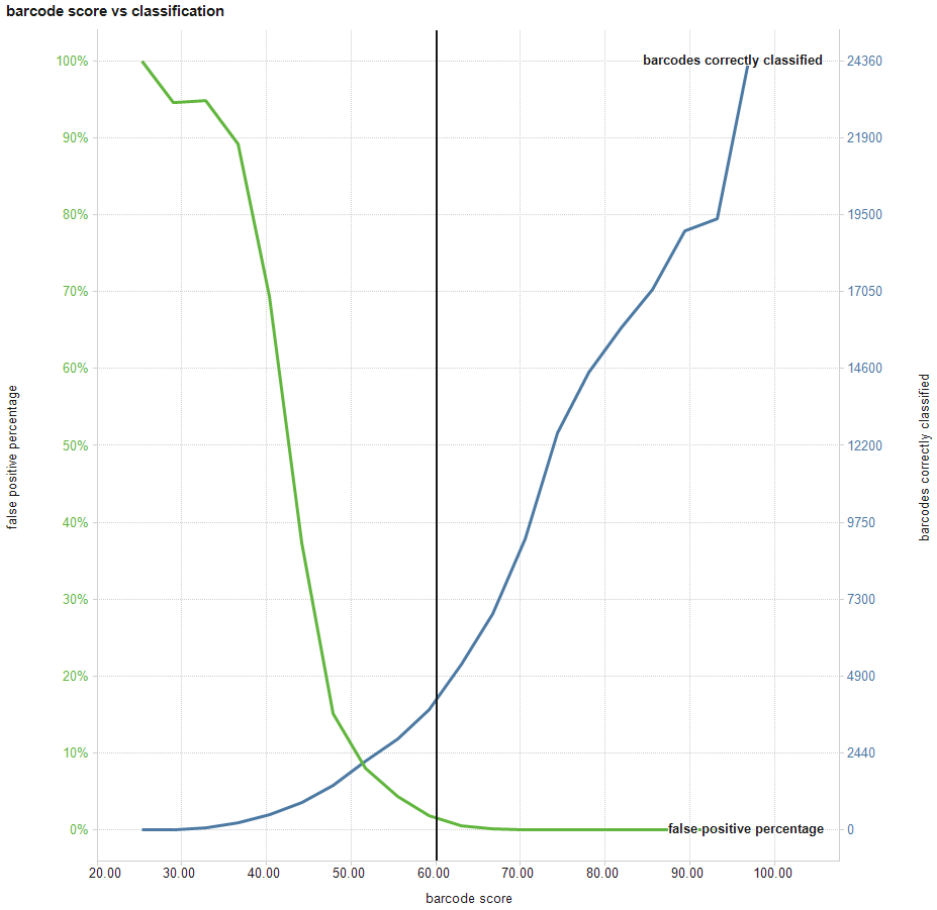
The classification threshold has been chosen to produce a low number of incorrect classifications while retaining an acceptable classification rate. This means that when a read has been classified as having a particular barcode, that classification will be incorrect a low number of times. Ideally this false-positive rate is around 1 in 1000, though this can be dependent on how well individually-barcoded samples are purified before they are pooled together. Classification rates should be 90% or above for samples with barcodes on both ends.

It is important to note that the above evaluation criteria assume that only reads which pass Guppy's quality filters are used. This corresponds to reads which are placed in the "pass" folder after basecalling; generally these will be reads with a mean q-score value greater than 7.

Modifying classification thresholds

It is possible to increase the number of classifications at the cost of the false-positive rate. Small changes to this can have a significant effect on the false-positive rate, so it is important to test any changes to the thresholds before using them.

For example, here is a graph of the number of reads classified for particular binned values of the (best) barcoding score. The data set is a collection of around 200,000 reads barcoded with the Native Expansion kit (EXP-NBD114):



This graph shows that, for example, reads where the best barcode score is around 30 will have about ~95% incorrect classifications. In contrast, for those reads where the highest barcode score is around 95 there will be near 0% incorrect classifications, and we correctly classify around 22,000 reads.

By reducing the threshold by a few points additional correct classifications may be obtained, but the cost in false positive percentage can go up significantly.

The threshold may be changed by modifying the `--min_score` argument, which applies the threshold to both the front and rear barcode. To have different thresholds for the front and rear barcode modify the `--min_score_rear_override` argument to change the rear barcode threshold. In that case the `--min_score` argument will apply to only the front barcode.

How classifications are reported

When barcodes are loaded into Guppy for classification, they are loaded in arrangements. An arrangement consists of either:

- One barcode, when searching for barcodes only at the front of a read.
- A front barcode and a rear barcode, when searching for barcodes at both ends of a read.

Once the classification for a particular read has been determined (by choosing the single highest-scoring barcode alignment), there may be another barcode in the arrangement corresponding to the other end of the read. The score for this barcode is also retrieved and reported, regardless of its classification – this means the entire arrangement is always reported.

For example, if a barcode arrangement is loaded containing `barcode01_FWD + barcode01_REV` with `barcode01_FWD` matching the front of the read with a score of 90 and `barcode01_REV` matching the rear of the read with a score of 10, then the final reported result will be:

```
front_barcode: barcode01_FWD
front_score: 90
rear_barcode: barcode01_REV
rear_score: 10
```

Alignment

Alignment overview

The Guppy toolchain provides the guppy_aligner executable to allow users to perform reference genome alignment on basecalled reads. Alignment is performed against the supplied reference via an integrated minimap2 aligner, full details of which can be found: https://github.com/lh3/minimap2. To perform alignment, invoke the Guppy aligner with the minimum required parameters:

```
guppy_aligner --input_path <folder containing FASTQ and/or FASTA files> --save_path <output folder> --align_ref <reference FASTA>
```

The input path will be searched for input FASTQ and FASTA files to perform alignment on. The `align_ref` is used to specify the reference genome. When performing alignment, Guppy aligner creates the following files in the output folder:

- `alignment_summary.txt`: Contains information about the best-quality alignment result for each read, such as alignment start, end, accuracy, etc. See "Summary file contents" in the "Input and output files" section for details.
- `read_processor_log-\<date and time\>.log`: A log file with information about the execution run.
- `.sam` or `.bam`: A SAM or BAM file is produced for each corresponding FASTQ/FASTA file located in the input folder. If a successful alignment is found which passes the coverage filter, the SAM/BAM file will contain a CIGAR string representing the alignment. The default alignment coverage required to consider a result successful is 60%. If BAM file output is enabled, BAM files will be sorted by reference ID and then the leftmost coordinate.

Guppy aligner supports the following optional parameters:

- Version (`--version`): Prints the version of Guppy aligner.
- Help (`-h` or `--help`): Print a help message describing usage and all the available parameters.
- Quiet mode (`-z` or `--quiet`): This option prevents the Guppy basecaller from outputting anything to stdout. Stdout is short for “standard output” and is the default location to which a running program sends its output. For a command line executable, stdout will typically be sent to the terminal window from which the program was run.
- Verbose logging (`--verbose_logs`): Flag to enable verbose logging (outputting a verbose log file, in addition to the standard log files, which contains detailed information about the application). Off by default.
- Worker thread count (`-t` or `--worker_threads`): The number of worker threads to spawn for the aligner to use. Increasing this number will allow Guppy aligner to make better use of multi-core CPU systems, but may impact overall system performance.
- Recursive (`-r` or `--recursive`): search through all subfolders contained in the `--input_path` value, and perform alignment on any .fastq, .fq, .fasta or .fa files found in them.

- BAM file output (--bam): This flag enables BAM file output. If the flag is not present, guppy_aligner defaults to SAM output.

If the aligner reports more than one possible alignment, only the best one is output. An alignment that covers less than 60% of the read or of the reference will be rejected.

Index files produced by the bwa aligner should also work as an align_ref but are not explicitly supported.

The integrated minimap2 aligner is run with no additional arguments supplied to it - the default values are used for all alignments. It is not possible to modify the arguments at this time.

The minimap library integration Oxford Nanopore uses is available on our GitHub page here: http://github.com/nanoporetech/ont_minimap2

Alignment index files

When aligning to large references (≥100 Mb) it is recommended to prepare an index file in advance for performance (to avoid generating the index during each run).

To create a minimap2 index file:

1. Download and install the minimap2 tool from: https://github.com/lh3/minimap2
2. Run the command:

```
minimap2 <input.fasta> <output.idx> -I 32G
```

-I 32G indicates the size of reference in bases before sharding occurs - this should be set to be larger than your reference length.

Sharding: In minimap2, by default, infers references greater than 4 Gb are split into 'shards' within the index in order to reduce RAM usage. The strand is then aligned separately against each reference shard which can lead to Guppy returning an incorrect alignment, if the strand aligns to a reference that is not within the first shard.

Calibration Strand detection

Calibration Strand detection

The DNA calibration strand (DCS) is a 3.6 kb amplicon of the Lambda phage genome. The calibration strand is added to DNA samples during the DNA repair/end-prep stage of library preparation, and is processed and included in the sample library.

Detection of calibration strands by the basecaller can be used to assess how well basecalling has worked, and to confirm that the sample preparation was successful.

If --calib_detect is enabled, Guppy will attempt to identify and analyse any calibration strands which have been basecalled. It does this by first checking to see if a basecalled strand is approximately the correct length (controlled by --calib_min_sequence_length and --calib_max_sequence_length), and it then aligns the basecalled strand to the calibration strand reference. Successfully aligned reads are placed in the calibration_strands folder, and alignment accuracy and identity metrics are added to the sequencing_summary.txt file. This can be a useful way of evaluating the quality of a run.

1D^2 basecalling

Requirements for 1D^2 basecalling

The 1D^2 basecall uses intermediate data generated during 1D basecalling, and the 1D basecall should be completed as above **ensuring the optional output flag --fast5_out is set**.

Alternatively, it is possible to use basecalling results produced by MinKNOW basecalling. The following are required:

- .fast5 files with results produced by MinKNOW basecalling, where the "event table" has been enabled. By default this table will be present in .fast5 files produced when selecting a 1D^2 kit (SQK-LSK309) in the MinKNOW GUI.
- The .txt file produced by MinKNOW basecalling.

1D^2 basecalling - Windows

The 1D reads can be processed in the 1D^2 basecaller with the following parameters:

- guppy_basecaller_1d2.exe
- --input_path: Full or relative path to the directory where the basecalled 1D .fast5 files were written in the previous step
- --index_file: Full or relative path to the sequencing_summary.txt file from the previous 1D basecall step
- --save_path: Full path to the directory where the basecalled 1D^2 results will be saved
- --config: configuration file containing Guppy parameters

For example, the following call will run 1D^2 analysis on a folder of pre-processed 1D reads:

```
"C:\Program Files\OxfordNanopore\ont-guppy-cpu\bin\guppy_basecaller_1d2.exe" --input_path output_folder\basecall\workspace --index_file output_folder\basecall\sequencing_summary.txt --save_path output_folder\basecall_1d2 --config dna_r9.5_450bps_1d2_raw.cfg
```

1D^2 basecalling - Linux

The 1D reads can be processed in the 1D^2 basecaller with the following parameters:

- guppy_basecaller_1d2.exe
- --input_path: Full or relative path to the directory where the basecalled 1D .fast5 files were written in the previous step
- --index_file: Full or relative path to the sequencing_summary.txt file from the previous 1D basecall step
- --save_path: Full path to the directory where the basecalled 1D^2 results will be saved
- --config: configuration file containing Guppy parameters

For example, the following call will run 1D^2 analysis on a folder of pre-processed 1D reads:

```
guppy_basecaller_1d2 --input_path output_folder/basecall/workspace --index_file output_folder/basecall/sequencing_summary.txt --save_path output_folder/basecall_1d2 --config dna_r9.5_450bps_1d2_raw.cfg
```

1D^2 basecalling - macOS

The 1D reads can be processed in the 1D^2 basecaller with the following parameters:

- guppy_basecaller_1d2.exe
- --input_path: Full or relative path to the directory where the basecalled 1D .fast5 files were written in the previous step
- --index_file: Full or relative path to the sequencing_summary.txt file from the previous 1D basecall step
- --save_path: Full path to the directory where the basecalled 1D^2 results will be saved
- --config: configuration file containing Guppy parameters

For example, the following call will run $1D^2$ analysis on a folder of pre-processed 1D reads:

```
/Users/myuser/ont-guppy-cpu/bin/guppy_basecaller_1d2 --input_path output_folder/basecall/workspace --index_file
output_folder/basecall/sequencing_summary.txt --save_path output_folder/basecall_1d2 --config dna_r9.5_450bps_1d2_raw.cfg
```

Modified base calling

Modified base calling

It is now possible to use Guppy to identify certain types of modified bases. This requires the use of a specific basecalling model which is trained to identify one or more types of modification. Configuration files for these new models can generally be identified by the inclusion of "modbases" in their name (e.g. dna_r9.4.1_450bps_modbases_dam-dcm-cpg_hac.cfg)

Modified base output consists of two parts:

1. A normal FASTQ record, exactly as you would get from normal basecalling, and available either as part of FASTQ files or as FASTQ entries embedded in .fast5 files.
2. A supplementary table provided as part of .fast5 output, which contains estimated probabilities that a particular base in the FASTQ entry is a modified one.

As is normal, it is necessary to specify the `--fast5_out` flag in order for Guppy to write additional tables into .fast5 files.

The supplementary table is named ModBaseProbs and is stored in the BaseCalled_template section of the latest basecall analysis. If using Python and the Oxford Nanopore Technologies' ont-fast5-api package, you can extract the table like this:

```
from ont_fast5_api.fast5_interface import get_fast5_file
with get_fast5_file(fast5_filepath, mode="r") as f5:
for read_id in f5.get_read_ids():
read = f5.get_read(read_id)
latest_basecall = read.get_latest_analysis('Basecall_1D')
mod_base_table = read.get_analysis_dataset(
latest_basecall, 'BaseCalled_template/ModBaseProbs')
print(read_id, mod_base_table)
```

The table is a two-dimensional array, where each row of the table corresponds to the corresponding base in the FASTQ entry. For example, the first row of the table (row 0) will correspond to the first base in the FASTQ entry.

Each row contains a number of columns equal to the number of canonical bases (four) plus the number of modifications present in the model. The columns list the bases in alphabetical order (ACGT for DNA, ACGU for RNA), and each base is immediately followed by columns corresponding to the modifications that apply to that particular base. For example, with a model that identified modifications for 6mA and 5mC, the column ordering would be A 6mA C 5mC G T.

Each table row describes the likelihood that, given that a particular base was called at that position, that that base is either a canonical one (i.e. a base that the model considers to be "unmodified"), or one of the modifications that is contained within the model. The contents of the table are integers in the range of 0-255, which represent likelihoods in the range of 0-100% (storing these values as integers allows us to reduce .fast5 file size). For example, a likelihood of 100% corresponds to a table entry of 255. Within a given row the table entries for a particular base will sum to 100%.

Following from our previous example with 6mA and 5mC, you might see a table row with entries like this:

```
[63, 192, 255, 0, 255, 255]
```

This would mean that:

- Given that an A was called, the likelihood that it is a canonical A is ~25% (63 / 255), and the likelihood that it is 6mA is ~75% (192 / 255).
- Given that a C was called, the likelihood that it is a canonical C is 100% (255 / 255), with no chance (0 / 255) of it being a 5mC.
- Given that a G or T was called, the likelihood that they are canonical bases is 100% (255 / 255 -- this should always be the case, as the model does not include any modification states for G or T).

The names of the modifications are contained in the metadata for the `ModBaseProbs` table. Extract them using `ont-fast5-api` like this (following from the previous code example):

```
table_path = '{}/BaseCalled_template/ModBaseProbs'.format(latest_basecall)
metadata = read.get_analysis_attributes(table_path)
print(read_id, metadata['modified_base_long_names'])
```

FAQ

FAQ and common issues

My basecalling speed is a lot slower than it was the last time I ran Guppy.

> This is most likely because the model you are using is different. The default configuration for Guppy is to use High Accuracy models, and these will be quite a lot slower than the "fast" models that Guppy used to have as a default. See the above section Setting up a run: configurations and parameters.

When I run Guppy with my GPU I get a `CUDA_ERROR_OUT_OF_MEMORY` error.

> This is because your GPU does not have enough memory to run basecalling. See the section CPU/GPU basecalling usage above about how to estimate the amount of memory to use. When in doubt, setting
>
> ```
> --num_callers 1
> --gpu_runners_per_device 1
> --chunks_per_runner 1
> ```
>
> Should result in very low GPU memory use (and correspondingly low basecalling speed), and you can experiment with increasing those numbers until running out of GPU memory again.

When I run the GPU version of Guppy I get the message `error while loading shared libraries: libcuda.so.1: cannot open shared object file.`

> This will happen when NVIDIA GPU drivers for Guppy have not been installed, or have been installed incorrectly. (Re)installing GPU driver packages should solve this.

I get slightly different results from Guppy when I run on different platforms or with different GPUs.

> Guppy should provide completely deterministic and repeatable results for a given platform, operating system, GPU, and GPU driver, but its output may be slightly different if any of those things change. The overall basecall results should be very similar, and in most cases completely identical.

Can I call short reads with Guppy?

Yes, it is possible to call reads as short as 50 bases with Guppy. Most configurations in Guppy use scaling algorithms based on the signal of the adapter sequence, and are robust to basecalling reads of any length. For configurations that do not yet support adapter scaling, a prototype short-read scaling method is available for DNA via the --pt_scaling option.

Troubleshooting

Memory usage for Guppy

If the memory requirements for Guppy exceed available memory on the host machine, or if other computationally-intensive work is performed while Guppy is running, then Guppy may run out of memory and crash. If Guppy fails for this reason the cause may not be directly apparent - the application may either simply crash or it may return a "segmentation fault" or "killed" error. In such cases, either the number of threads should be reduced or other computationally-intensive tasks should be stopped.

Note: If GPU basecalling is enabled, and available memory on the GPU device is exceeded, an out-of-memory error from CUDA may be emitted on startup. In this case, reduce the number of runners or the number of chunks per runner, to allow Guppy to run.

Advanced troubleshooting for Guppy crashes

Despite our best efforts, Guppy will occasionally crash due to bugs, and being able to diagnose and fix these is very important. This generally involves a two-step process:

1 Provide general configuration information from Guppy

1. Provide general configuration information This is just information about how you ran Guppy. It should include:
    1. Your Guppy version (find this by running guppy_basecaller with --version).
    2. The full command used to call Guppy.
    3. Any relevant files, such as your configuration file (if a custom one was used), or read files if available.
    4. Any other information you think is relevant, such as the hardware Guppy was running on (GPU model, etc).

2 Attempt to isolate the read which caused the crash in Guppy

Frequently, Guppy crashes will be due to a single read, and figuring out which read caused the crash will make it much easier to reproduce it. Do that like this:

1. If you're outputting reads to subfolders, check your guppy_basecaller_\<timestamp\>.log file to see which subfolder Guppy was writing to when it crashed. Any problem reads are likely in this folder.
2. Run the same Guppy command again (on just the affected subfolder if possible), enabling verbose logging.
3. Wait for Guppy to crash again.
4. Look in your log file for the last read which was sent to Guppy (it will probably not have a "finished processing" message present in the log).
5. Run Guppy again with **only** that one read, and see if it crashes. If it did, you've found the problem read and can send it to customer services.
6. If Guppy did not crash, try again with the last few reads listed in the log file, as Guppy may be processing several reads in parallel (especially while running on GPU).