CNVkit Documentation

Release 0.9.6.dev0

Eric Talevich

Dec 21, 2018

Contents

1 Quick start **3**
- 1.1 Install CNVkit . . . 3
- 1.2 Download the reference genome . . . 3
- 1.3 Map sequencing reads to the reference genome . . . 4
- 1.4 Build a reference from normal samples and infer tumor copy ratios . . . 4
- 1.5 Next steps . . . 5

2 Command line usage **7**
- 2.1 Copy number calling pipeline . . . 8
- 2.2 Plots and graphics . . . 19
- 2.3 Text and tabular reports . . . 28
- 2.4 Compatibility and other I/O . . . 31
- 2.5 RNA expression . . . 35
- 2.6 Additional scripts . . . 36

3 FAQ & How To **39**
- 3.1 File formats . . . 39
- 3.2 Chromosomal sex . . . 42
- 3.3 Calling copy number gains and losses . . . 43
- 3.4 Allele frequencies and copy number . . . 43
- 3.5 Bias corrections . . . 44
- 3.6 Tumor analysis . . . 46
- 3.7 Tumor heterogeneity . . . 46
- 3.8 Germline analysis . . . 48
- 3.9 Whole-genome sequencing and targeted amplicon capture . . . 49

4 Python API **51**
- 4.1 cnvlib package . . . 51
- 4.2 scikit-genome package . . . 81

5 Citation **91**
- 5.1 Who else is using CNVkit? . . . 91

6 Indices and tables **93**

Python Module Index **95**

Source code GitHub

License Apache License 2.0

Packages PyPI | Docker | Galaxy | DNAnexus

Article PLOS Computational Biology

Q&A Biostars

CNVkit is a Python library and command-line software toolkit to infer and visualize copy number from high-throughput DNA sequencing data. It is designed for use with hybrid capture, including both whole-exome and custom target panels, and short-read sequencing platforms such as Illumina and Ion Torrent.

CHAPTER 1

Quick start

If you would like to quickly try CNVkit without installing it, try our app on DNAnexus.

To run CNVkit on your own machine, keep reading.

1.1 Install CNVkit

Download the source code from GitHub:

https://github.com/etal/cnvkit

And read the README file.

1.2 Download the reference genome

Go to the UCSC Genome Bioinformatics website and download:

1. Your species' reference genome sequence, in FASTA format [required]
2. Gene annotation database, via RefSeq or Ensembl, in BED or "RefFlat" format (e.g. refFlat.txt) [optional]

You probably already have the reference genome sequence. If your species' genome is not available from UCSC, use whatever reference sequence you have. CNVkit only requires that your reference genome sequence be in FASTA format. Both the reference genome sequence and the annotation database must be single, uncompressed files.

Gene annotations: The gene annotations file (refFlat.txt) is useful to apply gene names to your baits BED file, if the BED file does not already have short, informative names for each bait interval. This file can be used in the next step.

If your targets look like:

```
chr1        1508981 1509154
chr1        2407978 2408183
chr1        2409866 2410095
```

Then you want refFlat.txt.

Otherwise, if they look like:

```
chr1        1508981 1509154 SSU72
chr1        2407978 2408183 PLCH2
chr1        2409866 2410095 PLCH2
```

Then you don't need refFlat.txt.

If your sequencing protocol is *WGS*, then you don't need a "target" BED file, but you probably do still want refFlat.txt.

1.3 Map sequencing reads to the reference genome

If you haven't done so already, use a sequence mapping/alignment program such as BWA to map your sequencing reads to the reference genome sequence.

You should now have one or BAM files corresponding to individual samples.

1.4 Build a reference from normal samples and infer tumor copy ratios

Here we'll assume the BAM files are a collection of "tumor" and "normal" samples, although germline disease samples can be used equally well in place of tumor samples.

CNVkit uses the bait BED file (provided by the vendor of your capture kit), reference genome sequence, and (optionally) *sequencing-accessible regions* along with your BAM files to:

1. Create a pooled *reference* of per-bin copy number estimates from several normal samples; then
2. Use this reference in processing all tumor samples that were sequenced with the same platform and library prep.

All of these steps are automated with the *batch* command. Assuming normal samples share the suffix "Normal.bam" and tumor samples "Tumor.bam", a complete `batch` command could be:

```
cnvkit.py batch *Tumor.bam --normal *Normal.bam \
    --targets my_baits.bed --fasta hg19.fasta \
    --access data/access-5kb-mappable.hg19.bed \
    --output-reference my_reference.cnn --output-dir example/
```

See the built-in help message to see what these options do, and for additional options:

```
cnvkit.py batch -h
```

If you have no normal samples to use for the *reference*, you can create a "flat" reference which assumes equal coverage in all bins by using the `--normal/-n` flag without specifying any additional BAM files:

```
cnvkit.py batch *Tumor.bam -n -t my_baits.bed -f hg19.fasta \
    --access data/access-5kb-mappable.hg19.bed \
    --output-reference my_flat_reference.cnn -d example2/
```

In either case, you should run this command with the reference genome sequence FASTA file to extract GC and RepeatMasker information for bias corrections, which enables CNVkit to improve the copy ratio estimates even without a paired normal sample.

If your targets are missing gene names, you can add them here with the `--annotate` argument:

```
cnvkit.py batch *Tumor.bam -n *Normal.bam -t my_baits.bed -f hg19.fasta \
    --annotate refFlat.txt --access data/access-5kb-mappable.hg19.bed \
    --output-reference my_flat_reference.cnn -d example3/
```

Note: Which BED file should I use?

- *target* vs. *bait* BED files: For hybrid capture, the targeted regions (or "primary targets") are the genomic regions your capture kit attempts to ensure are well covered, e.g. exons of genes of interest. The baited regions (or "capture targets") are the genomic regions your kit actually captures, usually including about 50bp flanking either side of each target. Give CNVkit the bait/capture BED file, not the primary targets.
- For *Whole-Genome Sequencing (WGS)*, use the `batch --method wgs` option and optionally give the genome's "access" file – if not given, it will be calculated from the genome sequence FASTA file.
- For *Targeted Amplicon Sequencing (TAS)*, use the `batch --method amplicon` option and give the target BED file.

See also: *Whole-genome sequencing and targeted amplicon capture*

1.5 Next steps

You can reuse the reference file you've previously constructed to extract copy number information from additional tumor sample BAM files, without repeating the steps above. Assuming the new tumor samples share the suffix "Tumor.bam" (and let's also spread the workload across all available CPUs with the `-p` option, and generate some figures):

```
cnvkit.py batch *Tumor.bam -r my_reference.cnn -p 0 --scatter --diagram -d example4/
```

The coordinates of the target and antitarget bins, the gene names for the targets, and the GC and RepeatMasker information for bias corrections are automatically extracted from the reference .cnn file you've built.

Now, starting a project from scratch, you could follow any of these approaches:

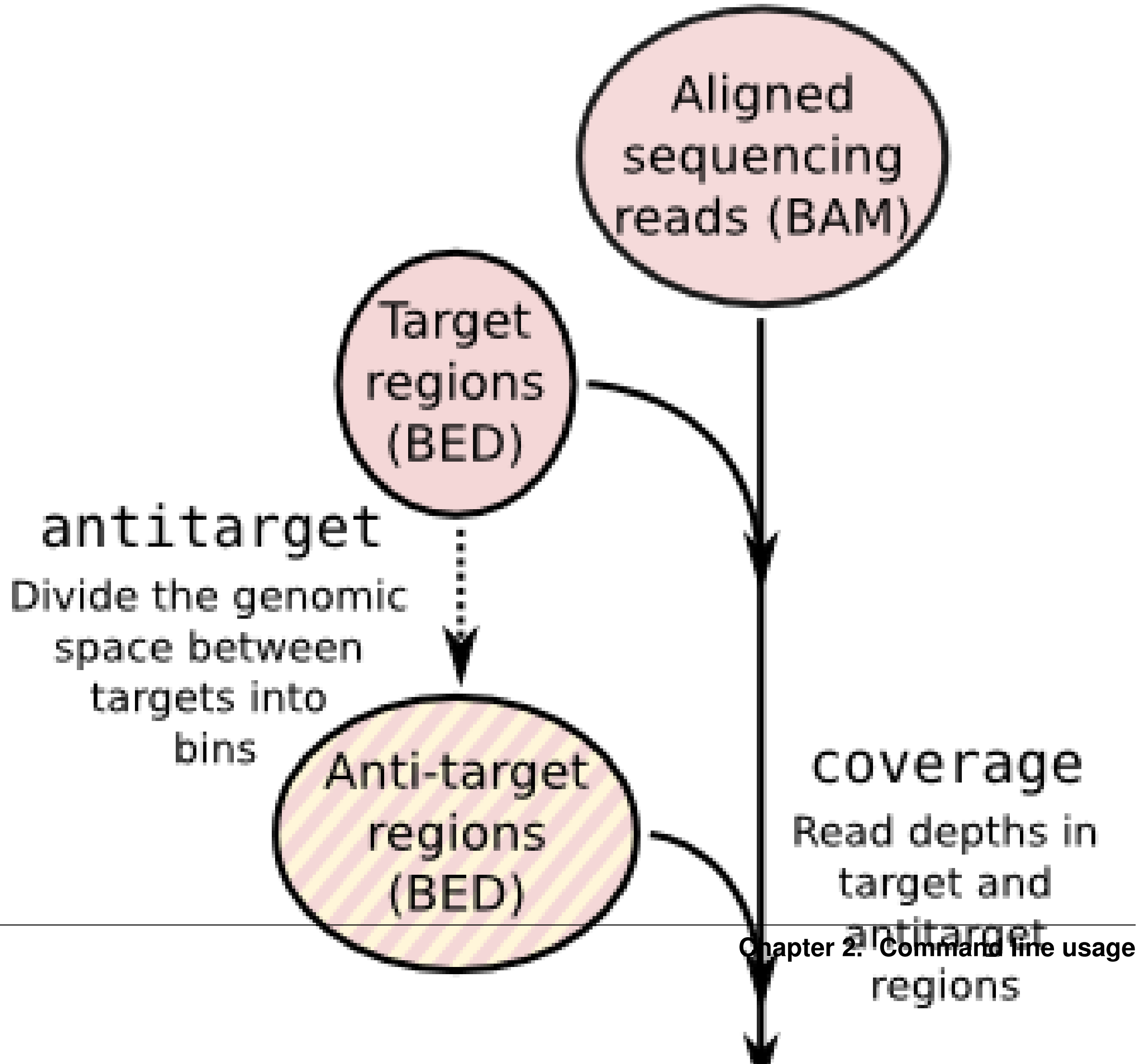
- Run `batch` as above with all tumor/test and normal/control samples specified as they are, and hope for the best. (This should usually work fine.)
- *For the careful:* Run `batch` with just the normal samples specified as normal, yielding coverage .cnn files and a **pooled reference**. Inspect the coverages of all samples with the *metrics* command, eliminating any poor-quality samples and choosing a larger or smaller antitarget bin size if necessary. Build an updated pooled reference using *batch* or *coverage* and *reference* (see *Copy number calling pipeline*), coordinating your work in a Makefile, Rakefile, or similar build tool.
    - See also: Ten Simple Rules for Reproducible Computational Research
- *For the power user:* Run `batch` with all samples specified as tumor samples, using `-n` by itself to build a **flat reference**, yielding coverages, copy ratios, segments and optionally plots for all samples, both tumor and normal. Inspect the "rough draft" outputs and determine an appropriate strategy to build and use a **pooled reference** to re-analyze the samples – ideally coordinated with a build tool as above.
- Use a framework like bcbio-nextgen to coordinate the complete sequencing data analysis pipeline.

See the command-line usage pages for additional *visualization*, *reporting* and *import/export* commands in CNVkit.

CHAPTER 2

Command line usage

2.1 Copy number calling pipeline


Aligned sequencing reads (BAM)
Target regions (BED)
antitarget
Divide the genomic space between targets into bins
Anti-target regions (BED)
coverage
Read depths in target and antitarget regions

Each operation is invoked as a sub-command of the main script, `cnvkit.py`. A listing of all sub-commands can be obtained with `cnvkit --help` or `-h`, and the usage information for each sub-command can be shown with the `--help` or `-h` option after each sub-command name:

```
cnvkit.py -h
cnvkit.py target -h
```

A sensible output file name is normally chosen if it isn't specified, except in the case of the text reporting commands, which print to standard output by default, and the matplotlib-based plotting commands (not `diagram`), which will display the plots interactively on the screen by default.

2.1.1 `batch`

Run the CNVkit pipeline on one or more BAM files:

```
# From baits and tumor/normal BAMs
cnvkit.py batch *Tumor.bam --normal *Normal.bam \
    --targets my_baits.bed --annotate refFlat.txt \
    --fasta hg19.fasta --access data/access-5kb-mappable.hg19.bed \
    --output-reference my_reference.cnn --output-dir results/ \
    --diagram --scatter

# Reusing a reference for additional samples
cnvkit.py batch *Tumor.bam -r Reference.cnn -d results/

# Reusing targets and antitargets to build a new reference, but no analysis
cnvkit.py batch -n *Normal.bam --output-reference new_reference.cnn \
    -t my_targets.bed -a my_antitargets.bed \
    -f hg19.fasta -g data/access-5kb-mappable.hg19.bed
```

With the `-p` option, process each of the BAM files in parallel, as separate subprocesses. The status messages logged to the console will be somewhat disorderly, but the pipeline will take advantage of multiple CPU cores to complete sooner.

```
cnvkit.py batch *.bam -r my_reference.cnn -p 8
```

The pipeline executed by the `batch` command is equivalent to:

```
cnvkit.py access hg19.fa -o access.hg19.bed
cnvkit.py autobin *.bam -t baits.bed -g access.hg19.bed [--annotate refFlat.txt --
↪short-names]

# For each sample...
cnvkit.py coverage Sample.bam baits.target.bed -o Sample.targetcoverage.cnn
cnvkit.py coverage Sample.bam baits.antitarget.bed -o Sample.antitargetcoverage.cnn

# With all normal samples...
cnvkit.py reference *Normal.{,anti}targetcoverage.cnn --fasta hg19.fa -o my_reference.
↪cnn

# For each tumor sample...
cnvkit.py fix Sample.targetcoverage.cnn Sample.antitargetcoverage.cnn my_reference.
↪cnn -o Sample.cnr
cnvkit.py segment Sample.cnr -o Sample.cns

# Optionally, with --scatter and --diagram
```

(continues on next page)

(continued from previous page)

```
cnvkit.py scatter Sample.cnr -s Sample.cns -o Sample-scatter.pdf
cnvkit.py diagram Sample.cnr -s Sample.cns -o Sample-diagram.pdf
```

This is for hybrid capture protocols in which both on- and off-target reads can be used for copy number detection. To run alternative pipelines for targeted amplicon sequencing or whole genome sequencing, use the `--method` option with value `amplicon` or `wgs`, respectively. The default is `hybrid`.

See the rest of the commands below to learn about each of these steps and other functionality in CNVkit.

2.1.2 `target`

Prepare a BED file of baited regions for use with CNVkit.

```
cnvkit.py target my_baits.bed --annotate refFlat.txt --split -o my_targets.bed
```

The BED file should be the baited genomic regions for your target capture kit, as provided by your vendor. Since these regions (usually exons) may be of unequal size, the `--split` option divides the larger regions so that the average bin size after dividing is close to the size specified by `--average-size`. If any of these three (`--split`, `--annotate`, or `--short-names`) flags are used, a new target BED file will be created; otherwise, the provided target BED file will be used as-is.

If you don't have the capture regions BED file, but you do know which commercial exome capture kit was used to prepare your samples, you might find the file you need in Astra-Zeneca's reference data repository. Otherwise, try searching the vendor's website or contacting their customer support for the right file. Failing that, you can try the *script guess_baits.py* included with the CNVkit distribution, along with the known exonic regions for your reference genome. But CNVkit will work much better if you have the real capture region coordinates – contact your sequencing lab for help with this step.

Bin size and resolution

If you need higher resolution, you can select a smaller average size for your target and *antitarget* bins.

Exons in the human genome have an average size of about 200bp. The target bin size default of 267 is chosen so that splitting larger exons will produce bins with a minimum size of 200. Since bins that contain fewer reads result in a noisier copy number signal, this approach ensures the "noisiness" of the bins produced by splitting larger exons will be no worse than average.

Setting the average size of target bins to 100bp, for example, will yield about twice as many target bins, which might result in higher-resolution segmentation. However, the number of reads counted in each bin will be reduced by about half, increasing the variance or "noise" in bin-level coverages. An excess of noisy bins can make visualization difficult, and since the noise may not be normally distributed, especially in the presence of many bins with zero reads, the segmentation algorithm could produce less accurate results on low-coverage samples. In practice we see good results with an average of 200–300 reads per bin; we therefore recommend an overall on-target sequencing coverage depth of at least 200x to 300x with a read length of 100bp to justify reducing the average target bin size to 100bp.

For hybrid capture, if your targets are not **tiled with uniform density** – for example, if your target panel is designed with a subset of targets having twice or half the usual number of tiles for a fixed number of genomic bases – you do not need to do anything in particular to compensate for this as long as you are using a pooled *reference*. When a test sample's read depths are normalized to the pooled reference, the log2 ratios will even out. However, the "spread" of those bins in your pooled reference, and the "weight" of the corresponding bins in the test sample's .cnr file, will be correspondingly higher or lower.

If some targets are enriched separately for each sample via **spike-in**, rather than as part of the original capture panel (which is assumed to have a fairly consistent capture efficiency across targets for all test and control samples), then the spike-in capture efficiency will typically vary too much to be useful as a copy number signal. In that case, the spike-in region should **not** be included in the target BED file, and **excluded** from the *access* regions (which determine *antitarget* regions) by using the `-x` option.

Labeling target regions

In case the vendor BED file does not label each region with a corresponding gene name, the `--annotate` option can add or replace these labels. Gene annotation databases, e.g. RefSeq or Ensembl, are available in "flat" format from UCSC (e.g. refFlat.txt for hg19).

In other cases the region labels are a combination of human-readable gene names and database accession codes, separated by commas (e.g. "ref|BRAF,mRNA|AB529216,ens|ENST00000496384"). The `--short-names` option splits these accessions on commas, then chooses the single accession that covers in the maximum number of consecutive regions that share that accession, and applies it as the new label for those regions. (You may find it simpler to just apply the refFlat annotations.)

The targets do not need to be genes, but for convenience CNVkit's documentation and source code generally refer to consecutive targeted regions with the same label as "genes".

2.1.3 `access`

Calculate the sequence-accessible coordinates in chromosomes from the given reference genome, output as a BED file.

```
cnvkit.py access hg19.fa -x excludes.bed -o access-excludes.hg19.bed
cnvkit.py access mm10.fasta -s 10000 -o access-10kb.mm10.bed
```

Many fully sequenced genomes, including the human genome, contain large regions of DNA that are inaccessable to sequencing. (These are mainly the centromeres, telomeres, and highly repetitive regions.) In the reference genome sequence these regions are filled in with large stretches of "N" characters. These regions cannot be mapped by resequencing, so CNVkit avoids them when calculating the *antitarget* bin locations.

The `access` command computes the locations of the accessible sequence regions for a given reference genome based on these masked-out sequences, treating long spans of 'N' characters as the inaccessible regions and outputting the coordinates of the regions between them.

Other known unmappable, variable, or poorly sequenced regions can be excluded with the `-x`/`--exclude` option. This option can be used more than once to exclude several BED files listing different sets of regions. For example, regions of poor mappability have been precalculated by others and are available from the UCSC FTP Server (see here for hg19).

If there are many small excluded/inaccessible regions in the genome, then small, less-reliable antitarget bins would be squeezed into the remaining accessible regions. The `-s` option ignores short regions that would otherwise be excluded, allowing larger antitarget bins to overlap them.

An "access" file precomputed for the UCSC reference human genome build hg19, with some know low-mappability regions excluded, is included in the CNVkit source distribution under the `data/` directory (`data/access-5kb-mappable.hg19.bed`).

2.1.4 `antitarget`

Given a "target" BED file that lists the chromosomal coordinates of the tiled regions used for targeted resequencing, derive a BED file off-target/"antitarget" regions.

```
cnvkit.py antitarget my_targets.bed -g data/access-5kb-mappable.hg19.bed -o my_
↪antitargets.bed
```

Certain genomic regions cannot be mapped by short-read resequencing (see *access*); we can avoid them when calculating the antitarget locations by passing the locations of the accessible sequence regions with the `-g` or `--access` option. CNVkit will then compute "antitarget" bins only within the accessible genomic regions specified in the "access" file.

CNVkit uses a cautious default off-target bin size that, in our experience, will typically include more reads than the average on-target bin. However, we encourage the user to examine the coverage statistics reported by CNVkit and specify a properly calculated off-target bin size for their samples in order to maximize copy number information.

Off-target bin size

An appropriate off-target bin size can be computed as the product of the average target region size and the fold-enrichment of sequencing reads in targeted regions, such that roughly the same number of reads are mapped to on– and off-target bins on average — roughly proportional to the level of on-target enrichment. The *autobin* command (below) can quickly estimate these values, but you are free to specify your own.

Average off-target coverage depths can also be obtained with the script CalculateHsMetrics in the Picard suite (http://picard.sourceforge.net/), or from the console output of the CNVkit *coverage* command when run on the target regions.

Note: The generated off-target bins are given the label "Antitarget" in CNVkit versions 0.9.0 and later. In earlier versions, the label "Background" was used – CNVkit will still accept this label for compatibility.

2.1.5 `autobin`

Quickly estimate read counts or depths in a BAM file to estimate reasonable on- and (if relevant) off-target bin sizes. If multiple BAMs are given, use the BAM with median file size.

Generates target and (if relevant) antitarget BED files, and prints a table of estimated average read depths and recommended bin sizes on standard output.

```
cnvkit.py autobin *.bam -t my_targets.bed -g access.hg19.bed
cnvkit.py autobin *.bam -m amplicon -t my_targets.bed
cnvkit.py autobin *.bam -m wgs -b 50000 -g access.hg19.bed --annotate refFlat.txt
```

The BAM index (.bai) is used to quickly determine the total number of reads present in a file, and random sampling of targeted regions (`-t`) is used to estimate average on-target read depth much faster than the *coverage* command.

2.1.6 `coverage`

Calculate coverage in the given regions from BAM read depths.

By default, coverage is calculated via mean read depth from a pileup. Alternatively, using the *–count* option counts the number of read start positions in the interval and normalizes to the interval size.

```
cnvkit.py coverage Sample.bam targets.bed -o Sample.targetcoverage.cnn
cnvkit.py coverage Sample.bam antitargets.bed -o Sample.antitargetcoverage.cnn
```

Summary statistics of read counts and their binning are printed to standard error when CNVkit finishes calculating the coverage of each sample (through either the *batch* or *coverage* commands).

BAM file preparation

For best results, use an aligner such as BWA-MEM, with the option to mark secondary mappings of reads, and flag PCR duplicates with a program such as SAMBLASTER, SAMBAMBA, or the MarkDuplicates script in Picard tools, so that CNVkit will skip these reads when calculating read depth.

You will probably want to index the finished BAM file using samtools or SAMBAMBA. But if you haven't done this beforehand, CNVkit will automatically do it for you.

Note: The BAM file must be sorted. CNVkit will check that the first few reads are sorted in positional order, and raise an error if they are not. However, CNVkit might not notice if reads later in the file are unsorted; it will just silently ignore the out-of-order reads and the coverages will be zero after that point. So be safe, and sort your BAM file properly.

Note: If you've prebuilt the BAM index file (.bai), make sure its timestamp is later than the BAM file's. CNVkit will automatically index the BAM file if needed – that is, if the .bai file is missing, *or* if the timestamp of the .bai file is older than that of the corresponding .bam file. This is done in case the BAM file has changed after the index was initially created. (If the index is wrong, CNVkit will not catch this, and coverages will be mysteriously truncated to zero after a certain point.) *However,* if you copy a set of BAM files and their index files (.bai) together over a network, the smaller .bai files will typically finish downloading first, and so their timestamp will be earlier than the corresponding BAM or FASTA file. CNVkit will then consider the index files to be out of date and will attempt to rebuild them. To prevent this, use the Unix command `touch` to update the timestamp on the index files after all files have been downloaded.

2.1.7 `reference`

Compile a copy-number reference from the given files or directory (containing normal samples). If given a reference genome (-f option), also calculate the GC content and repeat-masked proportion of each region.

The reference can be constructed from zero, one or multiple control samples (see below).

A reference should be constructed specifically for each target capture panel, using a BED file listing the genomic coordinates of the baited regions. Ideally, the control or "normal" samples used to build the reference should match the type of sample (e.g. FFPE-extracted or fresh DNA) and library preparation protocol or kit used for the test (e.g. tumor) samples.

For *target amplicon or whole-genome sequencing* protocols, the "antitarget" BED and .cnn files can be omitted. Otherwise, ensure the filename prefixes are the same for each pair of ".targetcoverage.cnn" and ".antitargetcoverage.cnn" files (as it's done by default).

Paired or pooled normals

Provide the `*.targetcoverage.cnn` and `*.antitargetcoverage.cnn` files created by the *coverage* command:

```
cnvkit.py reference *coverage.cnn -f ucsc.hg19.fa -o Reference.cnn
```

To analyze a cohort sequenced on a single platform, we recommend combining all normal samples into a pooled reference, even if matched tumor-normal pairs were sequenced – our benchmarking showed that a pooled reference performed slightly better than constructing a separate reference for each matched tumor-normal pair. Furthermore, even matched normals from a cohort sequenced together can exhibit distinctly different copy number biases (see Plagnol et al. 2012 and Backenroth et al. 2014); reusing a pooled reference across the cohort provides some consistency to help diagnose such issues.

Notes on sample selection:

- You can use `cnvkit.py metrics *.cnr -s *.cns` to see if any samples are especially noisy. See the *metrics* command.
- CNVkit will usually call larger CNAs reliably down to about 10x on-target coverage, but there will tend to be more spurious segments, and smaller-scale or subclonal CNAs can be hard to infer below that point. This is well below the minimum coverage thresholds typically used for SNV calling, especially for targeted sequencing of tumor samples that may have significant normal-cell contamination and subclonal tumor-cell populations. So, a normal sample that passes your other QC checks will probably be OK to use in building a CNVkit reference – assuming it was sequenced on the same platform as the other samples you're calling.

If normal samples are not available, it will sometimes be acceptable to build the reference from a collection of tumor samples. You can use the `scatter` command on the raw `.cnn` coverage files to help choose samples with relatively minimal and non-recurrent CNVs for use in the reference.

With no control samples

Alternatively, you can create a "flat" reference of neutral copy number (i.e. log2 0.0) for each probe from the target and antitarget interval files. This still computes the GC content of each region if the reference genome is given.

```
cnvkit.py reference -o FlatReference.cnn -f ucsc.hg19.fa -t targets.bed -a␣
↪antitargets.bed
```

Possible uses for a flat reference include:

1. Extract copy number information from one or a small number of tumor samples when no suitable reference or set of normal samples is available. The copy number calls will not be quite as accurate, but large-scale CNVs should still be visible.
2. Create a "dummy" reference to use as input to the `batch` command to process a set of normal samples. Then, create a "real" reference from the resulting `*.targetcoverage.cnn` and `*.antitargetcoverage.cnn` files, and re-run `batch` on a set of tumor samples using this updated reference.
3. Evaluate whether a given paired or pooled reference is suitable for an analysis by repeating the CNVkit analysis with a flat reference and comparing the CNAs found with both the original and flat reference for the same samples.

How it works

CNVkit uses robust methods to extract a usable signal from the reference samples.

Each input sample is first median-centered, then read-depth *bias corrections* (the same used in the *fix* command) are performed on each of the normal samples separately.

The samples' median-centered, bias-corrected log2 read depth values are then combined to take the weighted average (Tukey's biweight location) and spread (Tukey's biweight midvariance) of the values at each on– and off-target genomic bin among all samples. (For background on these statistical methods see Lax (1985) and Randal (2008).) To adjust for the lower statistical reliability of a smaller number of samples for estimating parameters, a "pseudocount" equivalent to one sample of neutral copy number is included in the dataset when calculating these values.

These values are saved in the output "reference.cnn" file as the "log2" and "spread" columns, indicating the expected read depth and the reliability of this estimate.

If a FASTA file of the reference genome is given, for each genomic bin the fraction of GC (proportion of "G" and "C" characters among all "A", "T", "G" and "C" characters in the subsequence, ignoring "N" and any other ambiguous characters) and repeat-masked values (proportion of lowercased non-"N" characters in the sequence) are calculated and stored in the output "reference.cnn" file as columns "gc" and "rmask". For efficiency, the samtools FASTA index file (.fai) is used to locate the binned sequence regions in the FASTA file. If the GC or RepeatMasker bias corrections are skipped using the `--no-gc` or `--no-rmask` options, then those columns are omitted from the output file; if both are skipped, then the genome FASTA file (if provided) is not examined at all.

The result is a reference copy-number profile that can then be used to correct other individual samples.

Note: As with BAM files, CNVkit will automatically index the FASTA file if the corresponding .fai file is missing or out of date. If you have copied the FASTA file and its index together over a network, you may need to use the `touch` command to update the .fai file's timestamp so that CNVkit will recognize it as up-to-date.

2.1.8 `fix`

Combine the uncorrected target and antitarget coverage tables (.cnn) and *correct for biases* in regional coverage and GC content, according to the given reference. Output a table of copy number ratios (.cnr).

```
cnvkit.py fix Sample.targetcoverage.cnn Sample.antitargetcoverage.cnn Reference.cnn -
↪o Sample.cnr
```

How it works

The "observed" on- and off-target read depths are each median-centered and *bias-corrected*, as when constructing the *reference*. The corresponding "expected" normalized log2 read-depth values from the reference are then subtracted for each set of bins.

Bias corrections use the GC and RepeatMasker information from the "gc" and "rmask" columns of the reference .cnn file; if those are missing (i.e. the reference was built without those corrections), `fix` will skip them too (with a warning). If you constructed the reference but then called *fix* with a different set of bias correction flags, the biases could be over- or under-corrected in the test sample – so use the options `--no-gc`, `--no-rmask` and `--no-edge` consistently or not at all.

CNVkit filters out bins failing certain predefined criteria: those where the reference log2 read depth is below a threshold (default -5), or the spread of read depths among all normal samples in the reference is above a threshold (default 1.0).

A weight is assigned to each remaining bin depending on:

1. The size of the bin;
2. The deviation of the bin's log2 value in the reference from 0;
3. The "spread" of the bin in the reference.

(The latter two only apply if at least one normal/control sample was used to build the reference.)

Finally, the corrected on- and off-target bin-level copy ratios with associated weights are concatenated, sorted, and written to a .cnr file.

2.1.9 `segment`

Infer discrete copy number segments from the given coverage table:

```
cnvkit.py segment Sample.cnr -o Sample.cns
```

Segmentation runs independently on each chromosome arm, and can be parallelized with the `-p` option (except for `flasso` and the HMM methods), similar to `batch`.

The significance threshold to accept a segment or breakpoint is passed to the underlying method with the option `--threshold/-t`. This is typically the p-value or q-value cutoff, or whatever parameter the underlying method uses to adjust its sensitivity.

Segmentation methods

The following segmentation algorithms can be specified with the `-m` option:

- `cbs` – the default, circular binary segmentation (CBS). This method performed best in our benchmarking on mid-size target panels and exomes. Requires the R package DNAcopy.
- `flasso` – Fused Lasso, reported by some users to perform best on exomes, whole genomes, and some target panels. Sometimes faster than CBS, but the current implementation cannot be parallelized over multiple CPUs. Beyond identifying breakpoints, additionally performs significance testing to distinguish CNAs from regions of neutral copy number, so large swathes of the output may have log2 values of exactly 0. Requires the R package cghFLasso.
- `haar` – a pure-Python implementation of HaarSeg, a wavelet-based method. Very fast and performs reasonably well on small panels, but tends to over-segment large datasets.
- `hmm` *(experimental)* – a 3-state Hidden Markov Model suitable for most samples. Faster than CBS, and slower but more accurate than Haar. Requires the Python package hmmlearn, as do the next two methods.
- `hmm-tumor` *(experimental)* – a 5-state HMM suitable for finer-grained segmentation of good-quality tumor samples. In particular, this method can detect focal amplifications within a larger-scale, smaller-amplitude copy number gain, or focal deep deletions within a larger-scale hemizygous loss. Training this model takes a bit more CPU time than the simpler `hmm` method.
- `hmm-germline` *(experimental)* – a 3-state HMM with fixed amplitude for the loss, neutral, and gain states corresponding to absolute copy numbers of 1, 2, and 3. Suitable for germline samples and single-cell sequencing of samples with mostly-diploid genomes that are not overly aneuploid.
- `none` – simply calculate the weighted mean log2 value of each chromosome arm. Useful for testing or debugging, or as a baseline for benchmarking other methods.

The first two methods use R internally, and to use them you will need to have R and the R package dependencies installed (i.e. DNAcopy, cghFLasso). If you installed CNVkit with `conda` as recommended, these should have been installed for you automatically. If you installed the R packages in a nonstandard or non-default location, you can specify the location of the right `Rscript` executable you want to use with `--rscript-path`.

The HMM methods `hmm`, `hmm-tumor` and `hmm-germline` were introduced provisionally in CNVkit v.0.9.2, and may change in future releases. They depend on the Python package `hmmlearn`, which is not installed by default. You can install the latest `hmmlearn` (ideally within a conda or virtualenv environment) after installing the rest of CNVkit with the command:

```
pip install hmmlearn
```

The methods `haar` and `none` do not have any additional dependencies beyond the basic CNVkit installation.

Bin filtering

Bins with a weight of 0 are dropped before segmentation. Additional filters:

- `--drop-low-coverage` – drop bins with a read depth of 0 or very close to 0, i.e. a log2 value suggesting the same. Use with tumor samples.
- `--drop-outliers` – drop bins with log2 value too far from a rolling average, taking local variability into account. Applied by default.

SNP allele frequencies

If a *VCF* file is given with the `--vcf` option (and accompanying options `-i`, `-n`, `-z`, and `--min-variant-depth`, which work as in other commands), then after segmenting log2 ratios, a second pass of segmentation will run within each log2-ratio-based segment on the SNP allele frequencies loaded from the VCF.

See also *Calling copy number gains and losses* for suggestions on how to interpret and post-process the resulting segments.

2.1.10 `call`

Given segmented log2 ratio estimates (.cns), derive each segment's absolute integer copy number using either:

- A list of threshold log2 values for each copy number state (`-m threshold`), or rescaling - for a given known tumor cell fraction and normal ploidy, then simple rounding to the nearest integer copy number (`-m clonal`).

```
cnvkit.py call Sample.cns -o Sample.call.cns
cnvkit.py call Sample.cns -y -m threshold -t=-1.1,-0.4,0.3,0.7 -o Sample.call.cns
cnvkit.py call Sample.cns -y -m clonal --purity 0.65 -o Sample.call.cns
cnvkit.py call Sample.cns -y -v Sample.vcf -m clonal --purity 0.7 -o Sample.call.cns
```

The output is another .cns file, with an additional "cn" column listing each segment's absolute integer copy number. This .cns file is still compatible with the other CNVkit commands that accept .cns files, and can be plotted the same way with the *scatter*, *heatmap* and *diagram* commands. To get these copy number values in another format, e.g. BED or VCF, see the *export* command.

With a VCF file of SNVs (`-v`/`--vcf`), the b-allele frequencies of SNPs in the tumor sample are extracted and averaged for each segment:

```
cnvkit.py call Sample.cns -y -v Sample.vcf -o Sample.call.cns
```

The segment b-allele frequencies are also used to calculate major and minor allele-specific integer copy numbers (see below).

Alternatively, the `-m none` option performs rescaling, re-centering, and extracting b-allele frequencies from a VCF (if requested), but does not add a "cn" column or allele copy numbers:

```
cnvkit.py call Sample.cns -v Sample.vcf --purity 0.8 -m none -o Sample.call.cns
```

Transformations

If there is a known level of normal-cell DNA contamination in the analyzed tumor sample (see the page on *tumor heterogeneity*), you can opt to rescale the log2 copy ratio estimates in your .cnr or .cns file to remove the impact of this contamination, so the resulting log2 ratio values in the file match what would be observed in a completely pure tumor sample.

With the `--purity` option, log2 ratios are rescaled to the value that would be seen a completely pure, uncontaminated sample. The observed log2 ratios in the input .cns file are treated as a mix of some fraction of tumor cells (specified by `--purity`), possibly with altered copy number, and a remainder of normal cells with neutral copy number (specified by `--ploidy` for autosomes; by default, diploid autosomes, haploid Y or X/Y depending on reference sex). This equation is rearranged to find the absolute copy number of the tumor cells alone, rounded to the nearest integer.

The expected and observed ploidy of the *sex chromosomes* (X and Y) is different, so if the option `-y` / `--male-reference` / `--haploid-x-reference` was used when constructing the *reference*, it's important to specify use the same option here. The sample sex can be specified if known, otherwise it will be guessed from the average log2 ratio of chromosome X. (See also: *Chromosomal sex*)

When a VCF file containing SNV calls for the same tumor sample (and optionally a matched normal) is given using the `-v`/`--vcf` option, the b-allele frequencies (BAFs) of the heterozygous, non-somatic SNVs falling within each segment are mirrored, averaged, and listed in the output .cns file as an additional "baf" column (using the same logic as `export nexus-ogt`). If `--purity` was specified, then the BAF values are also rescaled.

The `call` command can also optionally re-center the log2 values, though this will typically not be needed since the .cnr files are automatically median-centered by the *fix* command when normalizing to a reference and correcting biases. However, if the analyzed genome is highly aneuploid and contains widespread copy number losses or gains unequally, the default median centering may place copy-number-neutral regions slightly above or below the expected log2 value of 0.0. To address such cases, alternative centering approaches can be specified with the `--center` option:

```
cnvkit.py call -m none Sample.cns --center mode
```

Calling methods

After the above adjustments, the "threshold" and "clonal" methods calculate the absolute integer copy number of each segment.

The "clonal" method converts the log2 values to absolute scale using the given `--ploidy`, then simply rounds the absolute copy number values to the nearest integer. This method is reasonable for germline samples, highly pure tumor samples (e.g. cell lines), or when the tumor fraction is accurately known and specified with `--purity`.

The "threshold" method applies fixed log2 ratio cutoff values for each integer copy number state. This approach can be an alternative to specifying and adjusting for the tumor cell fraction or purity directly. However, if `--purity` is specified, then the log2 values will still be rescaled before applying the copy number thresholds.

The default threshold values are reasonably "safe" for a tumor sample with purity of at least 30%. The inner cutoffs of +0.2 and -0.25 are sensitive enough to detect a single-copy gain or loss in a diploid tumor with purity (or subclone cellularity) as low as 30%. But the outer cutoffs of -1.1 and +0.7 assume 100% purity, so a more extreme copy number, i.e. homozygous deletion (0 copies) or multi-copy amplification (4+ copies), is only assigned to a CNV if there is strong evidence for it. For germline samples, the `-t` values shown below (or `-m clonal`) may yield more precise calls.

The thresholds map to integer copy numbers in order, starting from zero: log2 ratios up to the first threshold value are assigned a copy number 0, log2 ratios between the first and second threshold values get copy number 1, and so on.

If log2 value is up to	Copy number
-1.1	0
-0.4	1
0.3	2
0.7	3
…	…

For homogeneous samples of known ploidy, you can calculate cutoffs from scatch by log-transforming the integer copy number values of interest, plus .5 (for rounding), divided by the ploidy. For a diploid genome:

```
>>> import numpy as np
>>> copy_nums = np.arange(5)
>>> print(np.log2((copy_nums+.5) / 2)
[-2.         -0.4150375   0.32192809  0.80735492  1.169925  ]
```

Or, in R:

```
> log2( (0:4 + .5) / 2)
[1] -2.0000000 -0.4150375  0.3219281  0.8073549  1.1699250
```

For arbitrary purity and ploidy:

```
> purity = 0.6
> ploidy = 4
> log2( (1 - purity) + purity * (0:6 + .5) / ploidy )
[1] -1.0740006 -0.6780719 -0.3677318 -0.1124747  0.1043367  0.2927817  0.4594316
```

Allele frequencies and counts

If a VCF file is given using the `-v`/`--vcf` option, then for each segment containing SNVs in the VCF, an average b-allele frequency (BAF) within that segment is calculated, and output in the "baf" column. Allele-specific integer copy number values are then inferred from the total copy number and BAF, and output in columns "cn1" and "cn2". This calculation uses the same method as PSCBS: total copy number is multiplied by the BAF, and rounded to the nearest integer.

Allelic imbalance, including copy-number-neutral loss of heterozygosity (LOH), is then apparent when a segment's "cn1" and "cn2" fields have different values.

Filtering segments

New in version 0.8.0.

Finally, segments can be filtered according to several criteria, which may be combined:

- Integer copy number (`cn`), merging adjacent with the same called value.
- Keeping only high-level amplifications (5 copies or more) and homozygous deletions (0 copies) (`ampdel`).
- Confidence interval overlapping zero (`ci`).
- Standard error of the mean (`sem`), a parametric estimate of confidence intervals which behaves similarly.

In each case, adjacent segments with the same value according to the given criteria are merged together and the column values are recalculated appropriately. Segments on different chromosomes or with different allele-specific copy number values will not be merged, even if the total copy number is the same.

2.2 Plots and graphics

The *scatter* and *heatmap* plots can be used in two ways:

1. Open the plot in an interactive window with zoom and other features. This is also compatible with Jupyter/IPython notebooks to render the plots inline.
2. Generate a static image plot with the `--output`/`-o` option.

- While PDF is a good choice to generate publication-quality figures that can be easily edited in Inkscape or Adobe Illustrator, other formats will work, indicated by the output filename extension – e.g. "-o myplot.png" to create PNG, or "-o myplot.svg" to create SVG.

(The *diagram* command can only generate a PDF file.)

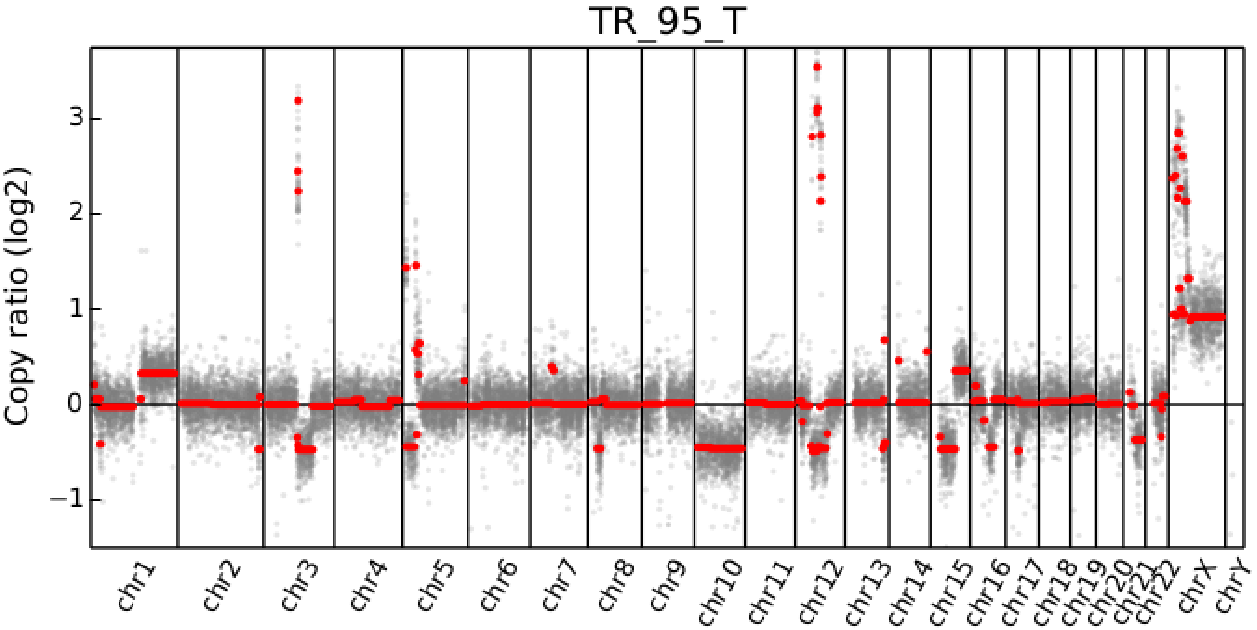
As with any CNVkit command, the `-h` option will show the complete list of options available:

```
cnvkit.py scatter -h
cnvkit.py diagram -h
cnvkit.py heatmap -h
```

2.2.1 `scatter`

Plot bin-level log2 coverages and segmentation calls together. Without any further arguments, this plots the genome-wide copy number in a form familiar to those who have used array CGH.

```
cnvkit.py scatter Sample.cnr -s Sample.cns
# Shell shorthand
cnvkit.py scatter -s TR_95_T.cn{s,r}
```



The options `--chromosome` and `--gene` (or their single-letter equivalents) focus the plot on the specified region:

```
cnvkit.py scatter -s Sample.cn{s,r} -c chr7
cnvkit.py scatter -s Sample.cn{s,r} -c chr7:140434347-140624540
cnvkit.py scatter -s Sample.cn{s,r} -g BRAF
```

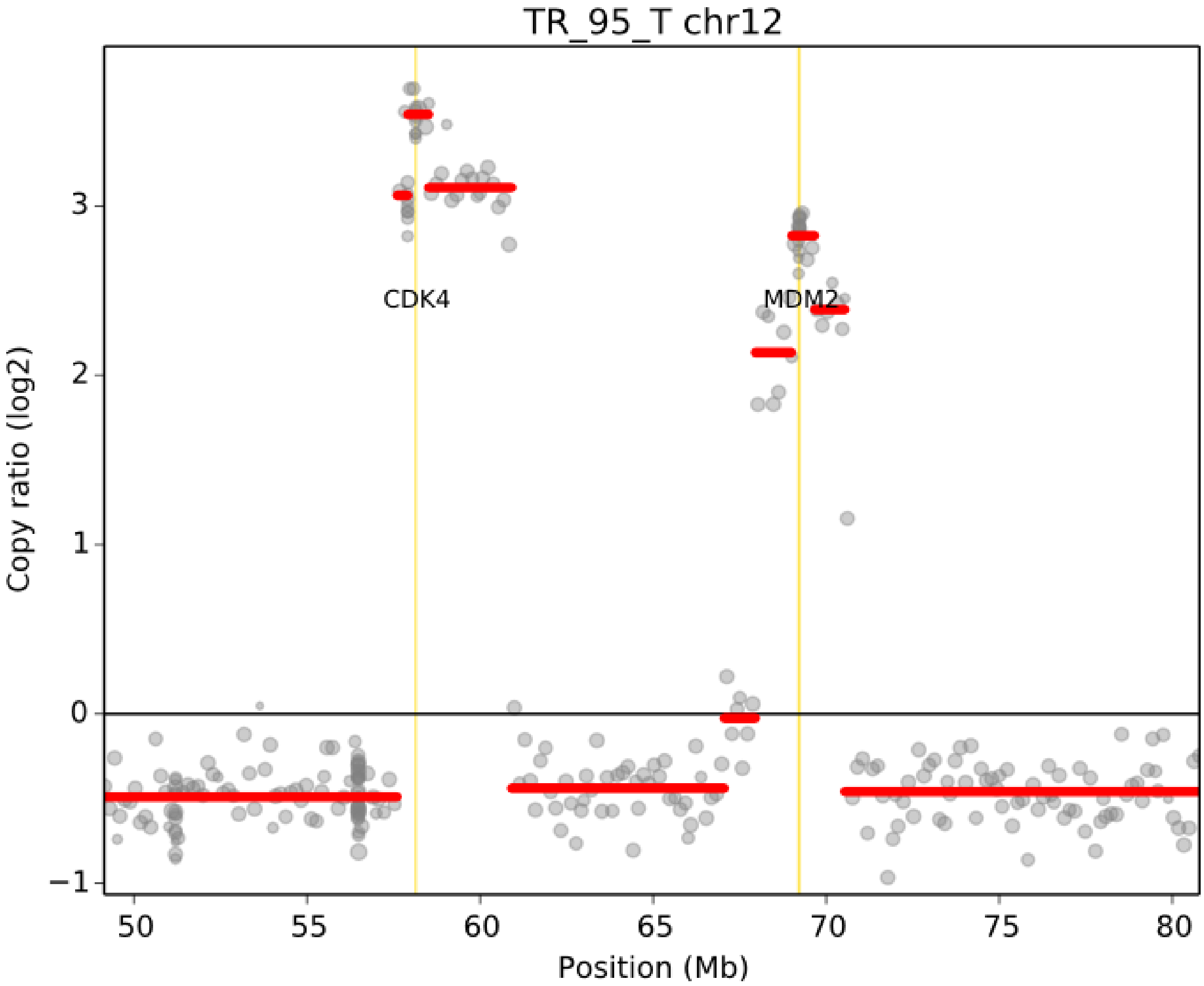
In the latter two cases, the genes in the specified region or with the specified names will be highlighted and labeled in the plot. The arguments `-c` and `-g` can be combined to e.g. highlight specific genes in a wider context:

```
# Show a chromosome arm, highlight one gene
cnvkit.py scatter -s Sample.cn{s,r} -c chr5:100-50000000 -g TERT
# Show the whole chromosome, highlight two genes
cnvkit.py scatter -s Sample.cn{s,r} -c chr7 -g BRAF,MET
```

(continues on next page)

(continued from previous page)

```
# Highlight two genes in a specified range
cnvkit.py scatter -s TR_95_T.cn{s,r} -c chr12:50000000-80000000 -g CDK4,MDM2
```



When a chromosomal region is plotted with CNVkit's "scatter" command , the size of the plotted datapoints is proportional to the weight of each point used in segmentation – a relatively small point indicates a less reliable bin. Therefore, if you see a cluster of smaller points in a short segment (or where you think there ought to be a segment, but there isn't one), then you can cast some doubt on the copy number call in that region. The dispersion of points around the segmentation line also visually indicates the level of noise or uncertainty.

The bin-level log2 ratios or coverages can also be plotted without segmentation calls:

```
cnvkit.py scatter Sample.cnr
```

This can be useful for viewing the raw, un-corrected coverage depths when deciding which samples to use to build a profile, or simply to see the coverages without being helped/biased by the called segments.

The `--trend` option (`-t`) adds a smoothed trendline to the plot. This can be helpful if the segmentation is not available, or if you're skeptical of the segmentation in a region.

Selection and highlighting

Chromosome-level views are controlled with the `--chromosome`/`-c` and `--gene`/`-g` options:

- A gene name (e.g. `-g TERT`) or multiple gene names separated by commas (e.g. `-g CDK4,MDM2`) will plot the genomic around that gene, or genes, and highlight the gene or genes with a vertical gold stripe.
    - If multiple genes, they must all be on the same chromosome.
    - The `--width`/`-w` argument determines the size of the plotted genomic region, in terms of basepairs flanking the selected region.
    - Any other genes in the plotted region will not be shown unless also specified with `-g`.
- A chromosome name alone (e.g. `-c chr5`) plots the whole chromosome. (No genes are highlighted.)
- A region label with chromosome name and 1-based start and end coordinates (e.g. `-c chr5:1000000-4000000`) plots the specified region, with the start and end coordinates as the x-axis limits. All genes in this region (that are labeled in the input .cnr file) are highlighted and labeled.
    - If the start or end coordinate is left off (e.g. `-c chr5:-4000000` or `-c chr7:140000000-`), the region is extended to the end of the chromosome in the direction of the open coordinate, i.e. it does what you'd think. If both are left off but `-` remains (e.g. `-c chrY:-`), the whole chromosome is shown, with all genes highlighted.
    - If `-c` is used, `-w` is ignored – only the specified genomic region will be shown, with no padding.
    - The `-g` option overrides the default behavior of showing all genes in the selection – only the genes specified with `-g` will be highlighted and labeled. To not show any genes, specify an empty string: `-g ''`
    - Special behavior occurs if there are no genes in the selected region: Instead, the selection itself is treated as a "gene", highlighted and labeled with the string "Selection", with padding controlled by `-w`. This behavior can be blocked by specifying an empty list of genes: `-g ''` – then the specified region will be plotted as usual, with nothing highlighted and no padding.

To create multiple region-specific plots at once, the regions of interest can be listed in a separate file and passed to the `scatter` command with the `-l`/`--range-list` option. This is equivalent to creating the plots separately with the `-c` option and then combining the plots into a single multi-page PDF.

Note: Only targeted genes can be highlighted and labeled; genes that are not included in the list of targets are not labeled in the .cnn or .cnr files and are therefore invisible to CNVkit.

SNV b-allele frequencies

The allelic frequencies of heterozygous SNPs can be viewed alongside copy number by passing variants as a *VCF* file with the `-v` option. These allele frequences are rendered in a subplot below the CNV scatter plot.

```
cnvkit.py scatter Sample.cnr -s Sample.cns -v Sample.vcf
```

If only the VCF file is given by itself, just the allelic frequencies are plotted:

```
cnvkit.py scatter -v Sample.vcf
```

When given segments, the plot will show the mean b-allele frequency values above and below 0.5 of SNVs falling within each segment. Divergence from 0.5 indicates loss of heterozygosity (LOH) or allelic imbalance in the tumor sample.

```
cnvkit.py scatter -s Sample.cns -v Sample.vcf -i TumorID -n NormalID
```

Given a VCF with only the tumor sample called, it is difficult to focus on just the informative SNPs because it's not known which SNVs are present and heterozygous in normal, germline cells. Better results can be had by giving CNVkit more information:

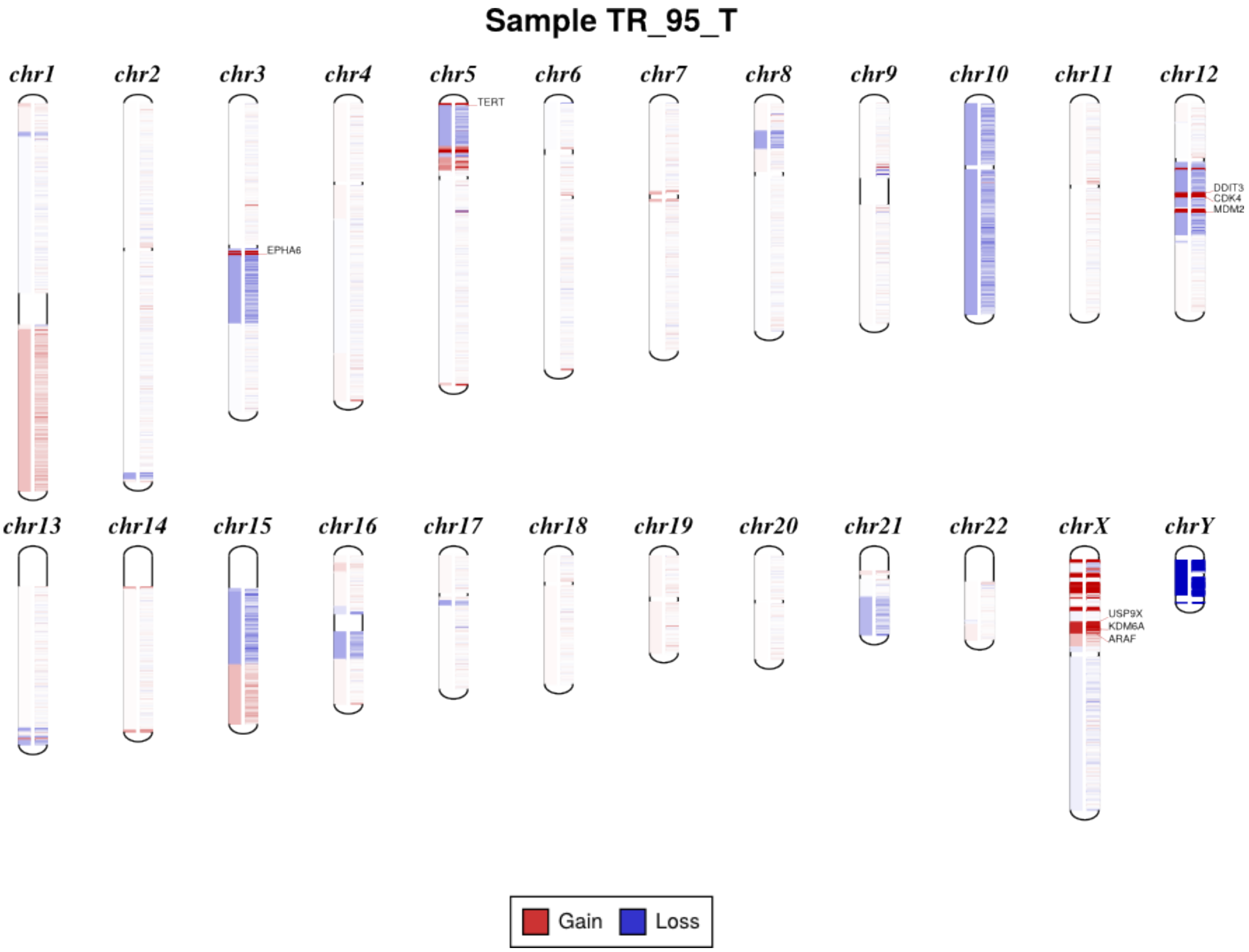
- Call somatic mutations using paired tumor and normal samples. In the VCF, the somatic variants should be flagged in the INFO column with the string "SOMATIC". (MuTect does this automatically.) Then CNVkit will skip these for plotting.
- Add a "PEDIGREE" tag to the VCF header, listing the tumor sample as "Derived" and the normal as "Original". (MuTect doesn't do this, but it does add a nonstandard GATK header that CNVkit can extract the same information from.)
- In lieu of a PEDIGREE tag, tell CNVkit which sample IDs are the tumor and normal using the `-i` and `-n` options, respectively.
- If no paired normal sample is available, you can still filter for likely informative SNPs by intersecting your tumor VCF with a set of known SNPs such as 1000 Genomes, ESP6500, or ExAC. Drop the private SNVs that don't appear in these databases to create a VCF more amenable to LOH detection.

2.2.2 `diagram`

Draw copy number (either individual bins (.cnn, .cnr) or segments (.cns)) on chromosomes as an ideogram. If both the bin-level log2 ratios and segmentation calls are given, show them side-by-side on each chromosome (segments on the left side, bins on the right side).

```
cnvkit.py diagram Sample.cnr
cnvkit.py diagram -s Sample.cns
cnvkit.py diagram -s Sample.cns Sample.cnr
```

If bin-level log2 ratios are provided (.cnr), genes with log2 ratio values beyond a fixed threshold will be labeled on the plot. This plot style works best with target panels of a few hundred genes at most; with whole-exome sequencing there are often so many genes affected by CNAs that the individual gene labels become difficult to read.



By default, the sex chromosomes X and Y are colorized relative to the expected ploidy, i.e. for male samples analyzed with default options, where the X chromosome in the input .cnr and .cns files has a log2 copy ratio near -1.0, in the output diagram it will be shown as neutral copy number (white or faint colors) rather than a loss (blue), because the sample's X chromosome (and Y) is recognized and expected to be haploid. (See *Chromosomal sex*.) The sample sex can be specified with the `-x`/`--sample-sex` option, or will otherwise be guessed automatically. This visual correction is done by default, but can be disabled with the option `--no-shift-xy`.

To get the same results in text form, i.e. a table of the amplified and deleted genes in a sample, use the *genemetrics* command.

Reducing cluttered gene labels

With tumor WGS or exome samples, the `diagram` output often appears extremely cluttered with hundreds or thousands of genes labeled.

You can reduce the number of labels by using a higher threshold (`diagram -t`) to limit labeling to deep deletions and high-level amplifications. The *genemetrics* command can help you determine the log2 value of genes of interest, and then a `-t` value slightly below that will disply only alterations at least that exteme.

To reduce the number of false-positive calls in the .cns file (see *Calling copy number gains and losses*), consider:

- Making the initial segmentation more stringent with *segment -t* or a larger bin size
- Filtering segments by confidence interval via *segmetrics –ci* and *call –filter ci*

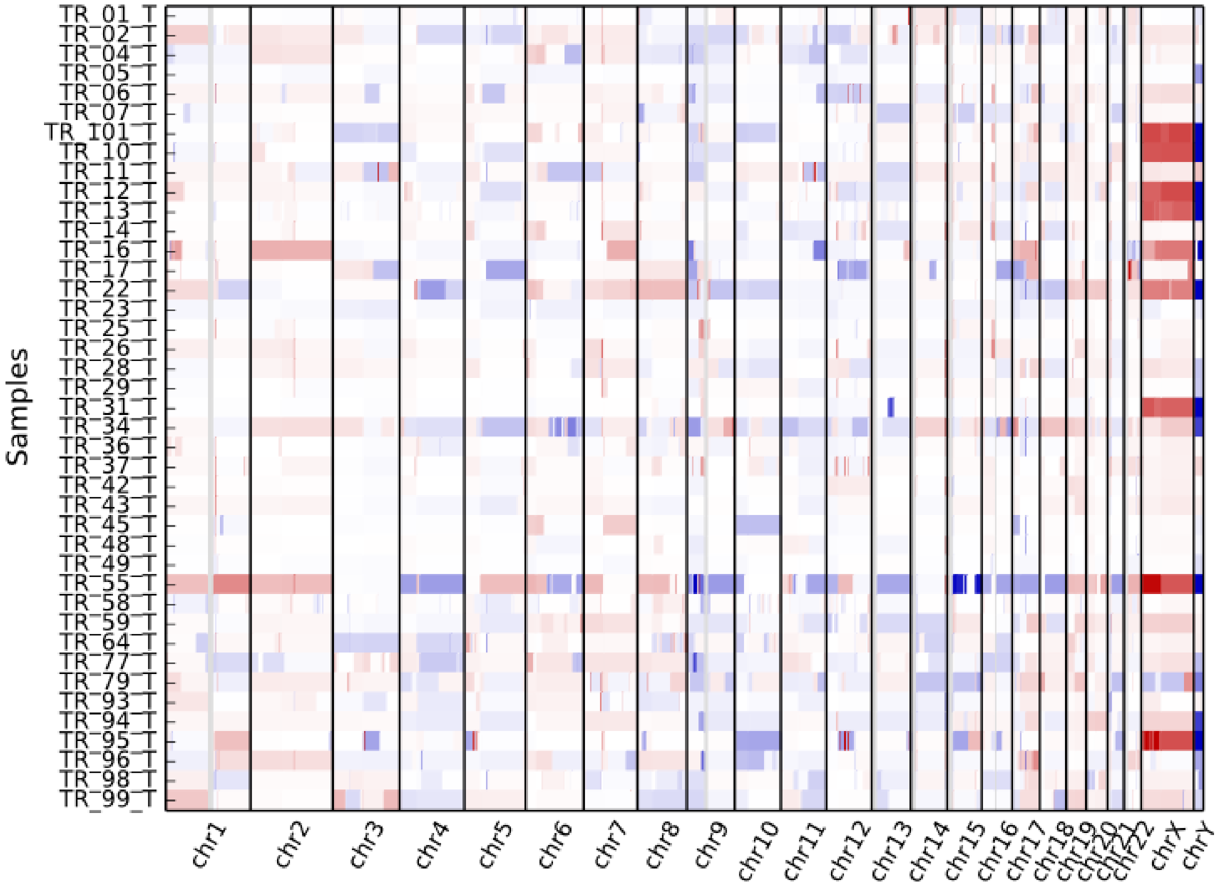
Alternatively, simply stick to the *scatter* and *heatmap* plots for visualizing these samples.

2.2.3 `heatmap`

Draw copy number (either bins (.cnn, .cnr) or segments (.cns)) for multiple samples as a heatmap.

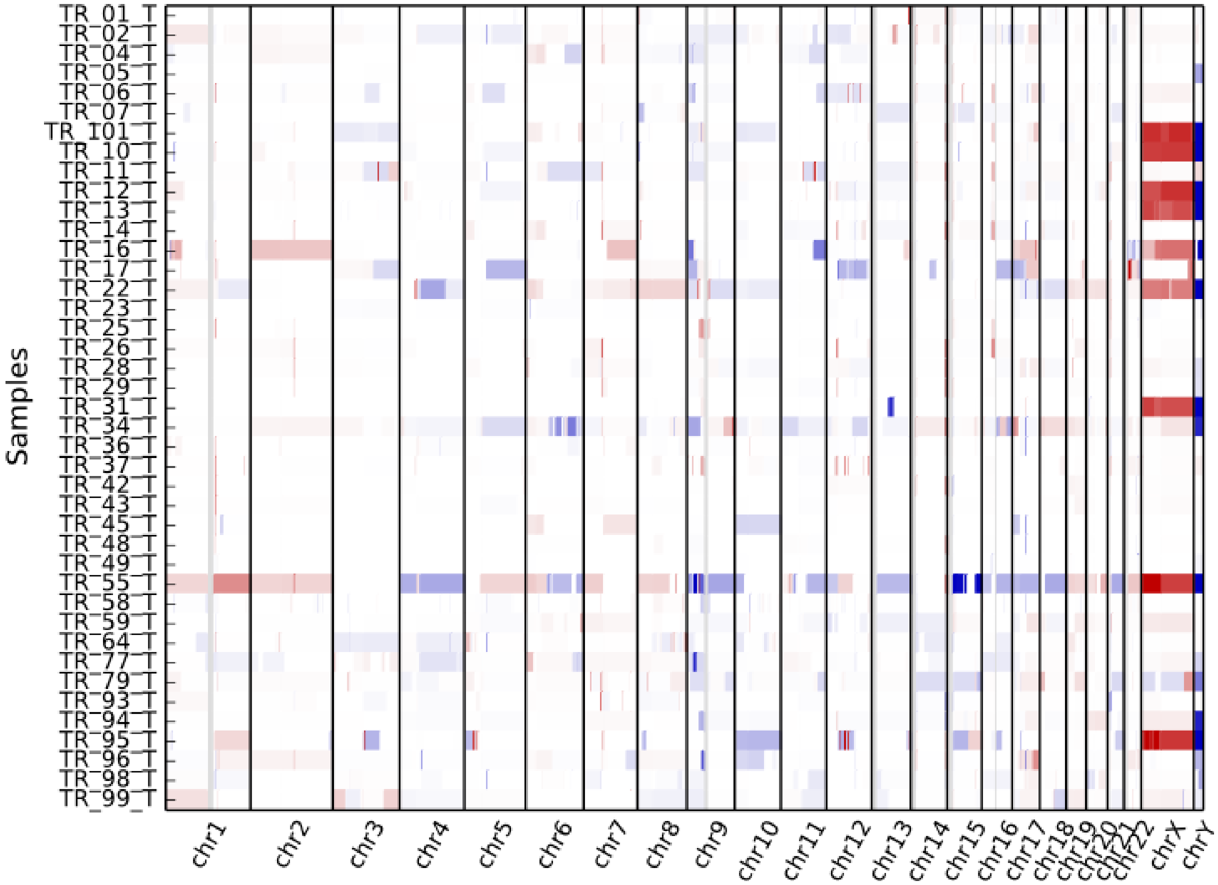
To get an overview of the larger-scale CNVs in a cohort, use the “heatmap” command on all .cns files:

```
cnvkit.py heatmap *.cns
```



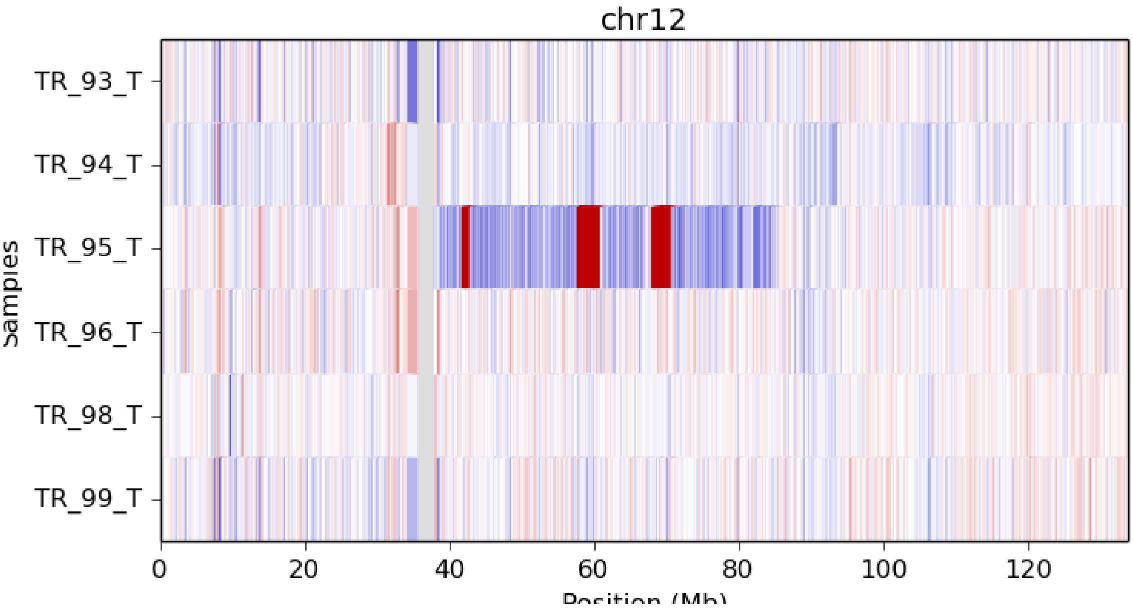
The color range can be subtly rescaled with the `-d` option to de-emphasize low-amplitude segments, which are likely spurious CNAs:

```
cnvkit.py heatmap *.cns -d
```



A heatmap can also be drawn from bin-level log2 coverages or copy ratios (.cnn, .cnr), but this can be slow to render at the genome-wide level. Consider doing this with a smaller number of samples and only for one chromosome or chromosomal region at a time, using the `-c` option:

```
cnvkit.py heatmap TR_9*T.cnr -c chr12
cnvkit.py heatmap TR_9*T.cnr -c chr7:125000000-145000000
```



If an output file name is not specified with the `-o` option, an interactive matplotlib window will open, allowing you to select smaller regions, zoom in, and save the image as a PDF or PNG file.

The samples are shown in the order there's given on the command line. If you use "*.cns" then the filenames might always be fetched alphabetically (depending on your operating system), but if you type them out in the order you like, it should keep that order. You can use the Unix shell to pull the names out of a file on the fly, e.g.:

```
cnvkit.py heatmap `cat filenames.txt`
```

As with *diagram*, the sex chromosomes X and Y are colorized relative to the expected ploidy, based on the sample and reference sex (see *Chromosomal sex*). This correction can be disabled with the option `--no-shift-xy`.

2.2.4 Customizing plots

The plots generated with the *scatter* and *heatmap* commands use the Python plotting library matplotlib.

To quickly adjust the displayed area of the genome in a plot, run either plotting command without the *-o* option to generate an interactive plot in a new window. You can then resize that plot up to the full size of your screen, use the plot window's selection mode to select a smaller area of the genome, and use the plot window's save button to save the plot in your preferred format.

You can customize font sizes and other aspects of the plots by configuring matplotlib. If you're running CNVkit on the command line and not using it as a Python library, then you can just create a file in your home directory (or the same directory as `cnvkit.py`) called `.matplotlibrc`. For example, to shrink the font size of the x- and y-axis labels, put this line in the configuration file:

```
axes.labelsize       : small
```

For more control, in the Python intepreter (or a script, or a Jupyter notebook), import the *cnvlib package* module and call the *do_scatter* or *do_heatmap* function to create a plot. Then you can use matplotlib.pyplot to get the current axis and modify the plot elements, change font sizes, or anything else you like:

```
from glob import glob
from matplotlib import pyplot as plt
```

(continues on next page)

(continued from previous page)

```
import cnvlib

segments = [cnvlib.read(f) for f in glob("*.cns")]
ax = cnvlib.do_heatmap(segments)
ax.set_title("All my samples")
plt.rcParams["font.size"] = 9.0
plt.show()
```

2.3 Text and tabular reports

2.3.1 `breaks`

List the targeted genes in which a segmentation breakpoint occurs.

```
cnvkit.py breaks Sample.cnr Sample.cns
```

This helps to identify genes in which (a) an unbalanced fusion or other structural rearrangement breakpoint occured, or (b) CNV calling is simply difficult due to an inconsistent copy number signal.

The output is a text table of tab-separated values, which is amenable to further processing by scripts and standard Unix tools such as `grep`, `sort`, `cut` and `awk`.

Columns:

- *gene*, *chromosome* – as in .cns (see *Target and antitarget bin-level coverages (.cnn)*), the gene where the breakpoint occurs and the chromosome it lies on.
- *location* – the `end` of the segment to the left of the breakpoint, and `start` of the segment to the right.
- *change* – the difference in `log2` values between the adjacent segments.
- *probes_left*, *probes_right* – the number of probes on each side of the breakpoint within the gene. (Not the same as the number of probes supporting each segment; just the portion within the gene.)

For example, to get a list of the names of genes that contain a possible copy number breakpoint (e.g. unbalanced translocation):

```
cnvkit.py breaks Sample.cnr Sample.cns | cut -f1 | sort -u > gene-breaks.txt
```

2.3.2 `genemetrics`

Identify targeted genes with copy number gain or loss above or below a threshold. (Formerly called `gainloss`.)

```
cnvkit.py genemetrics Sample.cnr
cnvkit.py genemetrics Sample.cnr -s Sample.cns -t 0.4 -m 5 -y
```

The first four columns of output table show each targeted gene's name and its genomic coordinates (based on the first and last bins with that label in the original target BED file, and thus the .cnr file).

The remaining output columns have slightly different meaning depending on whether or not segments were provided. Without segments (.cnr alone):

- *log2*: Weighted mean of log2 ratios of all the gene's bins, including any off-target intronic bins.
- *depth*: Weighted mean of un-normalized read depths across all this gene's bins.

- *weight*: Sum of this gene's bins' weights.
- *nbins*: The number of bins assigned to this gene.

With segments (`-s`):

- *log2*: The log2 ratio value of the segment covering the gene, i.e. weighted mean of all bins covered by the whole segment, not just this gene.
- *depth*, *weight*, *probes*: As above.
- *seg_weight*: The sum of the weights of the bins supporting the segment.
- *seg_probes*: The number of probes supporting the segment.

The `-t`/`--threshold` and `-m`/`--min-probes` options are used to control which genes are reported:

- A threshold of .2 (the default) will report single-copy gains and losses in a completely pure tumor sample (or germline CNVs), but a lower threshold would be necessary to call somatic CNAs if significant normal-cell contamination is present.
- Some likely false positives can be eliminated by dropping CNVs that cover a small number of bins, at the risk of missing some true positives. With `-m 3`, the default, genes where only 1 or 2 bins show copy number change will not be reported. This applies to the segment's bin count (*seg_probes*) if segments are provided with `-s`, otherwise it's the gene's bin count (*nbins*).

Specify the reference X-chromosome ploidy (`-y` if the same option was used when constructing the reference) to ensure CNVs on the X chromosome are reported correctly; otherwise, a large number of spurious gains or losses may be reported.

Note: Where more than one segment overlaps the gene, i.e. if the gene contains a breakpoint, each segment's value will be reported as a separate row for the same gene. If a large-scale CNA covers multiple genes, each of those genes will be listed individually.

The output is a text table of tab-separated values, like that of *breaks*. Continuing the Unix example, we can try `genemetrics` both with and without the segment files, take the intersection of those as a list of "trusted" genes, and visualize each of them with *scatter*:

```
cnvkit.py genemetrics -y Sample.cnr -s Sample.cns | tail -n+2 | cut -f1 | sort >␣
↪segment-genes.txt
cnvkit.py genemetrics -y Sample.cnr | tail -n+2 | cut -f1 | sort > ratio-genes.txt
comm -12 ratio-genes.txt segment-genes.txt > trusted-genes.txt
for gene in `cat trusted-genes.txt`
do
    cnvkit.py scatter -s Sample.cn{s,r} -g $gene -o Sample-$gene-scatter.pdf
done
```

(The point is that it's possible.)

2.3.3 `sex`

Guess samples' chromosomal sex from the relative coverage of chromosomes X and Y. A table of the sample name (derived from the filename), inferred sex (string "Female" or "Male"), and log2 ratio value of chromosomes X and Y is printed.

```
cnvkit.py sex *.cnn *.cnr *.cns
cnvkit.py sex -y *.cnn *.cnr *.cns
```

If there is any confusion in specifying either the sex of the sample or the construction of the reference copy number profile, you can check what happened using the `sex` command. If the reference and intermediate .cnn files are available (.targetcoverage.cnn and .antitargetcoverage.cnn, which are created before most of CNVkit's corrections), CNVkit can report the reference sex and the sample's relative coverage of the X and Y chromosomes:

```
cnvkit.py sex reference.cnn Sample.targetcoverage.cnn Sample.antitargetcoverage.cnn
```

The output looks like this, where columns are filename, inferred sex, and ratio of chromosome X and Y log2 coverages relative to autosomes:

```
cnv_reference.cnn   Female  -0.176  -1.061
Sample.targetcoverage.cnn   Female  -0.0818 -12.471
Sample.antitargetcoverage.cnn       Female  -0.265  -15.139
```

If the `-y` option was not specified when constructing the reference (e.g. `cnvkit.py batch ...`), then you have a female reference, and in the final plots you will see chrX with neutral copy number in female samples and around -1 log2 ratio in male samples.

2.3.4 `metrics`

Calculate the spread of bin-level copy ratios from the corresponding final segments using several statistics. These statistics help quantify how "noisy" a sample is and help to decide which samples to exclude from an analysis, or to select normal samples for a reference copy number profile.

For a single sample:

```
cnvkit.py metrics Sample.cnr -s Sample.cns
```

(Note that the order of arguments and options matters here, unlike the other commands: Everything after the `-s` flag is treated as a segment dataset.)

Multiple samples can be processed together to produce a table:

```
cnvkit.py metrics S1.cnr S2.cnr -s S1.cns S2.cns
cnvkit.py metrics *.cnr -s *.cns
```

Several bin-level log2 ratio estimates for a single sample, such as the uncorrected on- and off-target coverages and the final bin-level log2 ratios, can be compared to the same final segmentation (reusing the given segments for each coverage dataset):

```
cnvkit.py metrics Sample.targetcoverage.cnn Sample.antitargetcoverage.cnn Sample.cnr -
↪s Sample.cns
```

In each case, given the bin-level copy ratios (.cnr) and segments (.cns) for a sample, the log2 ratio value of each segment is subtracted from each of the bins it covers, and several estimators of spread are calculated from the residual values. The output table shows for each sample:

- Total number of segments (in the .cns file) – a large number of segments can indicate that the sample has either many real CNAs, or noisy coverage and therefore many spurious segments.
- Uncorrected sample standard deviation – this measure is prone to being inflated by a few outliers, such as may occur in regions of poor coverage or if the targets used with CNVkit analysis did not exactly match the capture. (Also note that the log2 ratio data are not quite normally distributed.) However, if a sample's standard deviation is drastically higher than the other estimates shown by the `metrics` command, that helpfully indicates the sample has some outlier bins.
- Median absolute deviation (MAD) – very robust against outliers, but less statistically efficient.

- Interquartile range (IQR) – another robust measure that is easy to understand.
- Tukey's biweight midvariance – a robust and efficient measure of spread.

Note that many small segments will fit noisy data better, shrinking the residuals used to calculate the other estimates of spread, even if many of the segments are spurious. One possible heuristic for judging the overall noisiness of each sample in a table is to multiply the number of segments by the biweight midvariance – the value will tend to be higher for unreliable samples. Check questionable samples for poor coverage (using e.g. bedtools, chanjo, IGV or Picard CalculateHsMetrics).

Finally, visualizing a sample with CNVkit's *scatter* command will often make it apparent whether a sample or the copy ratios within a genomic region can be trusted.

2.3.5 `segmetrics`

Calculate summary statistics of the residual bin-level log2 ratio estimates from the segment means, similar to the existing *metrics* command, but for each segment individually.

Results are output in the same format as the CNVkit segmentation file (.cns), with the stat names and calculated values printed in additional columns.

```
cnvkit.py segmetrics Sample.cnr -s Sample.cns --iqr
cnvkit.py segmetrics -s Sample.cn{s,r} --ci --pi
```

Supported stats:

- Alternative estimators of segment mean, which ignore bin weights: `--mean`, `-median`, `--mode`.
- As in *metrics*: standard deviation (`--std`), median absolute deviation (`--mad`), inter-quartile range (`--iqr`), Tukey's biweight midvariance (`--bivar`)
- Additionally: mean squared error (`--mse`), standard error of the mean (`-sem`).
- Confidence interval of the segment mean (`--ci`), estimated by bootstrap (100 resamplings) of the bin-level log2 ratio values within the segment. The upper and lower bounds are output as separate columns `ci_lo` and `ci_hi`.
- Prediction interval (`--pi`), estimated by the range between the 2.5-97.5 percentiles of the segment's bin-level log2 ratios. The upper and lower bounds are output as columns `pi_lo` and `pi_hi`.

The `--ci` and `--sem` values obtained here can also be used in the *call* command for filtering segments.

2.4 Compatibility and other I/O

2.4.1 `version`

Print CNVkit's version as a string on standard output:

```
cnvkit.py version
```

If you submit a bug report or feature request for CNVkit, please include the CNVkit version in your message so we can help you more efficiently.

2.4.2 `import-picard`

Convert Picard CalculateHsMetrics per-target coverage files (.csv) to the CNVkit .cnn format:

```
cnvkit.py import-picard *.hsmetrics.targetcoverages.csv *.hsmetrics.
↪antitargetcoverages.csv
cnvkit.py import-picard picard-hsmetrics/ -d cnvkit-from-picard/
```

You can use Picard tools to perform the bin read depth and GC calculations that CNVkit normally performs with the *coverage* and *reference* commands, if need be.

Procedure:

1. Use the *target* and *antitarget* commands to generate the "targets.bed" and "antitargets.bed" files.
2. Convert those BED files to Picard's "interval list" format by adding the BAM header to the top of the BED file and rearranging the columns – see the Picard command BedToIntervalList.
3. Run Picard CalculateHsMetrics on each of your normal/control BAM files with the "targets" and "antitargets" interval lists (separately), your reference genome, and the "PER_TARGET_COVERAGE" option.
4. Use *import-picard* to convert all of the PER_TARGET_COVERAGE files to CNVkit's .cnn format.
5. Use *reference* to build a CNVkit reference from those .cnn files. It will retain the GC values Picard calculated; you don't need to provide the reference genome sequence again to get GC (but you if you do, it will also calculate the RepeatMaster fraction values)
6. Use *batch* with the `-r`/`--reference` option to process the rest of your test samples.

2.4.3 `import-seg`

Convert a file in the *SEG* format (e.g. the output of standard CBS or the GenePattern server) into one or more CNVkit .cns files.

The chromosomes in a SEG file may have been converted from chromosome names to integer IDs. Options in `import-seg` can help recover the original names.

- To add a "chr" prefix, use "-p chr".
- To convert chromosome indices 23, 24 and 25 to the names "X", "Y" and "M" (a common convention), use "-c human".
- To use an arbitrary mapping of indices to chromosome names, use a comma-separated "key:value" string. For example, the human convention would be: "-c 23:X,24:Y,25:M".

2.4.4 `import-theta`

Convert the ".results" output of THetA2 to one or more CNVkit .cns files representing subclones with integer absolute copy number in each segment.

```
cnvkit.py import-theta Sample.cns Sample.BEST.results
```

See the page on tumor *Tumor heterogeneity* for more guidance on performing this analysis.

2.4.5 `export`

Convert copy number ratio tables (.cnr files) or segments (.cns) to another format.

bed

Segments can be exported to BED format to support a variety of other uses, such as viewing in a genome browser. By default only regions with copy number different from the given ploidy (default 2) are output. (Notice what this means for allosomes.) To output all segments, use the `--show all` option.

The BED format represents integer copy numbers in absolute scale, not log2 ratios. If the input .cns file contains a "cn" column with integer copy number values, as generated by the *call* command, *export bed* will use those values. Otherwise the log2 ratio value of each input segment is converted and rounded to an integer value, similar to the *call -m clonal* method.

```
# Estimate integer copy number of each segment
cnvkit.py call Sample.cns -y -o Sample.call.cns
# Show estimated integer copy number of all regions
cnvkit.py export bed Sample.call.cns --show all -y -o Sample.bed
```

The same BED format can also specify CNV regions to the FreeBayes variant caller with FreeBayes's `--cnv-map` option:

```
# Show only CNV regions
cnvkit.py export bed Sample.call.cns -o all-samples.cnv-map.bed
```

vcf

Convert segments, ideally already adjusted by the *call* command, to a *VCF* file. Copy ratios are converted to absolute integers, as with BED export, and VCF records are created for the segments where the copy number is different from the expected ploidy (e.g. 2 on autosomes, 1 on haploid sex chromosomes, depending on sample sex).

A sample's *chromosomal sex* can be specified with the `-x`/`--sample-sex` option, or will be guessed automatically. If the option `-y` / `--male-reference` / `--haploid-x-reference` was used to construct the *reference*, use it here, too.

```
cnvkit.py export vcf Sample.cns -y -g female -i "SampleID" -o Sample.cnv.vcf
```

cdt, jtv

A collection of probe-level copy ratio files (`*.cnr`) can be exported to Java TreeView via the standard CDT format or a plain text table:

```
cnvkit.py export jtv *.cnr -o Samples-JTV.txt
cnvkit.py export cdt *.cnr -o Samples.cdt
```

seg

Similarly, the segmentation files for multiple samples (`*.cns`) can be exported to the standard SEG format to be loaded in the Integrative Genomic Viewer (IGV):

```
cnvkit.py export seg *.cns -o Samples.seg
```

nexus-basic

The format `nexus-basic` can be loaded directly by the commercial program Biodiscovery Nexus Copy Number, specifying the "basic" input format in that program. This allows viewing CNVkit data as if it were from array CGH.

This is a tabular format very similar to .cnr files, with the columns:

1. chromosome
2. start
3. end
4. log2

nexus-ogt

The format `nexus-ogt` can be loaded directly by the commercial program Biodiscovery Nexus Copy Number, specifying the "Custom-OGT" input format in that program. This allows viewing CNVkit data as if it were from a SNP array.

This is a tabular format similar to .cnr files, but with B-allele frequencies (BAFs) extracted from a corresponding VCF file. The format's columns are (with .cnr equivalents):

1. "Chromosome" (chromosome)
2. "Position" (start)
3. "Position" (end)
4. "Log R Ratio" (log2)
5. "B-Allele Frequency" (from VCF)

The positions of each heterozygous variant record in the given VCF are matched to bins in the given .cnr file, and the variant allele frequencies are extracted and assigned to the matching bins.

- If a bin contains no variants, the BAF field is left blank
- If a bin contains multiple variants, the BAFs of those variants are "mirrored" to be all above .5 (e.g. BAF of .3 becomes .7), then the median is taken as the bin-wide BAF.

theta

THetA2 is a program for estimating normal-cell contamination and tumor subclone population fractions based on a tumor sample's copy number profile and, optionally, SNP allele frequencies. (See the page on tumor *Tumor heterogeneity* for more guidance.)

THetA2's input file is a BED-like file, typically with the extension `.interval_count`, listing the read counts within each copy-number segment in a pair of tumor and normal samples. CNVkit can generate this file given the CNVkit-inferred tumor segmentation (.cns), bypassing the initial step of THetA2, CreateExomeInput, which counts the reads in each sample's BAM file.

The normal-sample read counts in this file are used for weighting each segment in THetA2's calculations. We recommend providing these to `export theta` via the CNVkit pooled or paired reference file (.cnn) you created for your panel:

```
# From an existing CNVkit reference
cnvkit.py export theta Sample_Tumor.cns reference.cnn -o Sample.theta2.interval_count
```

The THetA2 normal read counts can also be derived from the normal sample's bin log2 ratios, if for some reason this is all you have:

```
# From a paired normal sample
cnvkit.py export theta Sample_Tumor.cns Sample_Normal.cnr -o Sample.theta2.interval_
↪count
```

If neither file is given, the THetA2 normal read counts will be calculated from the segment weight values in the given .cns file, or the number of probes if the "weight" column is missing, or as a last resort, the segment sizes if the "probes" column is also missing:

```
# From segment weights and/or probe counts
cnvkit.py export theta Sample_Tumor.cns -o Sample.theta2.interval_count
```

THetA2 also can take the tumor and normal samples' SNP allele frequencies as input to improve its estimates. THetA2 uses another custom format for these values, and provides another script for creating these files from VCF that we'd again prefer to bypass. CNVkit's `export theta` command produces these two additional files when given a VCF file of paired tumor-normal SNV calls with the `-v`/`--vcf` option:

```
cnvkit.py export theta Sample_Tumor.cns reference.cnn -v Sample_Paired.vcf
```

This produces three output files; `-o` will be used for the read count file, while the SNV allele count files will be named according to the .cns file, e.g. `Sample_Tumor.tumor.snp_formatted.txt` and `Sample_Tumor.normal.snp_formatted.txt`.

2.5 RNA expression

The relationship between genes' copy number and mRNA expression varies across the genome. For a subset of genes, mostly housekeeping genes, the mRNA expression levels measured by transcriptome sequencing are mostly explained by underlying the genic regions' genomic copy number. CNVkit can use this information to estimate coarse-grained copy number from RNA sequencing of a process-matched cohort of samples.

Samples are processed simultaneously as a cohort, and two additional input files are needed to complete the processing pipeline:

- **Gene info** – a table of per-transcript and per-gene metadata exported from Ensembl BioMart. A snapshot of this file for the human transcriptome is bundled with CNVkit under `data/ensembl-gene-info.hg38.tsv`.
- **CNV-expression correlation coefficients** – calculated per gene via the script `cnv_expression_correlate.py` included with CNVkit. A pre-calculated set of coefficients calculated on TCGA melanoma datasets, obtained from cBioPortal, is bundled with CNVkit under `data/tcga-skcm.cnv-expr-corr.tsv`.

With the above files, the RNA analysis can be run with either of the following commands:

2.5.1 `import-rna`

Use like this:

```
cnvkit.py import-rna [ *_rsem.genes.results | *.txt ] \
    --gene-resource data/ensembl-gene-info.hg38.tsv \
    --correlations data/tcga-skcm.cnv-expr-corr.tsv \
    --output out-summary.tsv --output-dir out/
```

Each gene's read counts and average transcript length are taken from the input file for each sample. Normalized, bias-corrected `*.cnr` files are written to `--output-dir`, and an optional summary table with all samples' data is written to `-output`.

Input file sources:

- **RSEM:** The third-party RNA quantification program RSEM produces `*_rsem.genes.results` output files that can be used as input to the `import-rna` command.
- **Gene counts:** Alternatively, the gene Ensembl IDs and per-gene read counts can be read from a simple 2-column, tab-delimited file. This format is used by TCGA level 2 RNA expression data. You can also create the equivalent on your own from the output of another RNA quantification tool like Salmon or Kallisto.

2.5.2 Segmentation

The output `*.cnr` files can be used with CNVkit's *segment*, *scatter*, and other commands similarly to the `*.cnr` files generated from DNA sequencing data. Each gene is represented by a single bin, and bins may be overlapping or even nested.

The `cbs` segmentation method performs reasonably well to produce a coarse-grained segmentation of the file. The `hmm` segmentation method also does well.

When using the *scatter* command to plot these files, note that the bin (gene-level) weights are particularly important, and are visually represented by circle size. A smoothed trendline (`-t`/`--trend`) can be helpful to supplement the coarse-grained segmentation.

2.5.3 Considerations

- Input samples should be process-matched and ideally of the same source tissue type. The DNA source for each sample can be single cells or bulk tissue.
- The cohort size should be at least 5, preferably at least 10, samples.
- The `--correlations` input is not required but is strongly recommended. The TCGA melanoma cohort correlations can be used for analysis of any tissue type, not just neoplastic melanocytes. However, best results will usually be achieved with a correlations table specific to the test cohort. The script `cnv_expression_correlate.py` generates this table from input tables of per-gene and per-sample copy number and expression levels, typically retrieved from cBioPortal for TCGA cancer-specific cohorts.

2.6 Additional scripts

`cnv_annotate.py` Update gene names (the 'gene' column) in CNVkit .cnn/.cnr files, using gene annotations from another UCSC RefFlat, BED, or GFF file (e.g. refFlat.txt). This may be useful if you notice at the end of an analysis that vendor-annotated targets are not the desired gene names, and want to change the labeling without repeating the analysis with an updated target BED file.

`cnv_expression_correlate.py` Create a table of correlation coefficients between gene copy number and mRNA expression. See: *RNA expression*

`cnv_updater.py` Update .cnn, .cnr and .cns files previously generated by earlier versions of CNVkit to add a "depth" column used in CNVkit version 0.8.0 and later. The script reads each input file, calculates absolute-scale depth from the file's existing "log2" column value in each row, and creates a corresponding output file with a modified name – the input files are not modified in-place.

Running this script is not necessary for new analyses, but may help ease the transition for analyses that have already begun.

`cnv_ztest.py` Test each bin in a .cnr file individually for non-neutral copy number. Specifically, calculate the probability of a bin's log2 value versus a normal distribution with a mean of of 0 and standard deviation back-calculated from bin weight. Output another .cnr file with z-test probabilities in the additional column "ztest"; drop rows where the probability is above the threshold (`--alpha`/`-a`).

`guess_baits.py` Use the read depths in one or more given BAM files to infer which regions were targeted in a hybrid capture or targeted amplicon capture sequencing protocol. This script can be used in case the original BED file of targeted intervals is unavailable. (However, CNVkit will give much better results if the true targeted intervals can be provided.) It works in 2 modes, guided and unguided:

- **Guided:** Given candidate targets, such as all known exons in the reference genome, test the mean coverage depth in each candidate target and drop those that did not receive sufficient coverage, presumed to be those exons or genes that were not targeted by the sequencing library.

```
guess_baits.py Sample1.bam Sample2.bam -t ucsc-exons.bed -o baits.bed
```

- **Unguided:** Scan every base in the sample BAM(s), inferring likely boundaries for enriched regions. (This is usually much slower then the guided approach.)

```
guess_baits.py -g access.hg19.bed Sample1.bam Sample2.bam -o baits.bed
```

In either mode, the input region coordinates can be provided in any of the formats handled by skgenome.tabio, but it's best to first run them through either the command *target* or script `skg_convert.py --flatten` (see below) to ensure the input regions do not overlap.

`reference2targets.py` Extract target and antitarget BED files from a CNVkit reference file. While the *batch* command does this step automatically when an existing reference is provided, you may find this standalone script useful to recover the target and antitarget BED files that match the reference if those BED files are missing or you're not sure which ones are correct.

Alternatively, once you have a stable CNVkit reference for your platform, you can use this script to drop the "bad" bins from your target and antitarget BED files (and subsequently built references) to avoid unnecessarily calculating coverage in those bins during future runs.

`skg_convert.py` Convert between any of the tabular data *formats* supported by skgenome.tabio, including BED and UCSC RefFlat (e.g. refFlat.txt from UCSC Genome Bioinformatics).

CHAPTER 3

FAQ & How To

3.1 File formats

We've tried to use standard file formats where possible in CNVkit. However, in a few cases we have needed to extend the standard BED format to accommodate additional information.

All of the non-standard file formats used by CNVkit are tab-separated plain text and can be loaded in a spreadsheet program, R or other statistical analysis software for manual analysis, if desired.

3.1.1 BED and GATK/Picard Interval List

- UCSC Genome Browser's BED definition and FAQ
- GATK's Interval List description and FAQ

Note that BED genomic coordinates are 0-indexed, like C or Python code – for example, the first nucleotide of a 1000-basepair sequence has position 0, the last nucleotide has position 999, and the entire region is indicated by the range 0-1000.

GATK and Picard interval list coordinates are 1-indexed, like R or Matlab code. In the same example, the first nucleotide of a 1000-basepair sequence has position 1, the last nucleotide has position 1000, and the entire region is indicated by the range 1-1000. These files usually have the extension *.interval_list*.

In GATK4, the term "interval list" also refers to samtools-style genomic coordinate specifications of the form *chromosome:start-end*, e.g. *chr1:1-1000*. As with Picard and older GATK style interval lists, the coordinates are 1-indexed. When used with GATK4, these files usually have the extension *.list* or *.interval*.

CNVkit will load these files by automatically determining the specific format based on the file contents, not the filename extension.

3.1.2 SEG

The SEG format is the tabular output of DNAcopy, the reference implementation of Circular Binary Segmentation (CBS). It is a tab-separated table with the following 5 or 6 columns:

- *ID* – sample name
- *chrom* – chromosome name or ID
- *loc.start* – segment's genomic start position, 1-indexed
- *loc.end* – segment end position
- *num.mark* – (optional) number of probes or bins covered by the segment
- *seg.mean* – segment mean value, usually in log2 scale

The column names in the first line are not enforced, and can vary across implementations.

SEG files can be used with a number of other programs that operate on segmented log2 copy ratios – including GISTIC 2.0, IGV, the GenePattern server, and many R packages.

To convert CNVkit's .cns files to SEG, use the command *export* `seg`, and to convert SEG files produced outside of CNVkit into CNVkit's own segmented format (.cns), use *import-seg*.

3.1.3 VCF

See the VCF specifications.

CNVkit currently uses VCF files in two ways:

- To extract single-nucleotide variant (SNV) allele frequencies, which can be plotted in the *scatter* command, used to assign allele-specific copy number in the *call* command, or exported along with bin-level copy ratios to the "nexus-ogt" format.
- To *export* CNVs, describing/encoding each CNV segment as a structural variant (SV).

For the former – investigating allelic imbalance and loss of heterozygosity (LOH) – it's most useful to perform paired calling on matched tumor/normal samples. You can use a separate SNV caller such as FreeBayes, VarDict, or MuTect to do this. For best results, ensure that:

- Both the tumor and normal samples are present in the same VCF file.
- Include both germline and somatic variants (if any) in the VCF file. (For MuTect, this means keeping the "REJECT" records.) Mark somatic variants with the "SOMATIC" flag in the INFO column.
- Add a PEDIGREE tag to the VCF header declaring the tumor sample(s) as "Derived" and the normal as "Original". Without this tag, you'll need to tell CNVkit which sample is which using the *-i* and *-n* options in each command.

An example VCF constructed from the 1000 Genomes samples NA12878 and NA12882 is included in CNVkit's test suite.

3.1.4 Target and antitarget bin-level coverages (.cnn)

CNVkit saves its information in a tabular format similar to BED, but with additional columns. Each row in the file indicates an on-target or off-target (a.k.a. "antitarget") bin. Genomic coordinates are 0-indexed, like BED. Column names are shown as the first line of the file.

In the output of the *coverage* command, the columns are:

- Chromosome or reference sequence name (`chromosome`)
- Start position (`start`)
- End position (`end`)
- Gene name (`gene`)

- Log2 mean coverage depth (`log2`)
- Absolute-scale mean coverage depth (`depth`)

Essentially the same tabular file format is used for coverages (.cnn), ratios (.cnr) and segments (.cns) emitted by CNVkit.

3.1.5 Copy number reference profile (.cnn)

In addition to the columns present in the "target" and "antitarget" .cnn files, the reference .cnn file has the columns:

- GC content of the sequence region (`gc`)
- RepeatMasker-masked proportion of the sequence region (`rmask`)
- Statistical spread or dispersion (`spread`)

The **log2** coverage depth is the robust average of coverage depths, excluding extreme outliers, observed at the corresponding bin in each the sample .cnn files used to construct the *reference*. The **spread** is a similarly robust estimate of the standard deviation of normalized log2 coverages in the bin. The **depth** column is the robust average of absolute-scale coverage depths from the input .cnn files, but without any bias corrections.

To manually review potentially problematic targets in the built reference, you can sort the file by the **spread** column; bins with higher values are the noisy ones.

It is important to keep the copy number reference file consistent for the duration of a project, reusing the same reference for bias correction of all tumor samples in a cohort. If your library preparation protocol changes, it's usually best to build a new reference file and use the new file to analyze the samples prepared under the new protocol.

3.1.6 Bin-level log2 ratios (.cnr)

In addition to the `chromosome`, `start`, `end`, `gene`, `log2` and `depth` columns present in .cnn files, the .cnr file includes each bin's proportional weight or reliability (`weight`).

The **weight** value is derived from several sources:

- The size of the bin relative to the average bin size (for targets or antitargets, separately)
- For a paired or pooled reference, the deviation of the reference log2 value from neutral coverage (i.e. distance from 0.0)
- For a pooled reference, the inverse of the variance (i.e. square of `spread` in the reference) of normalized log2 coverage values seen among all normal samples at that bin.

This calculated value is used to weight the bin log2 ratio values during segmentation. Also, when a genomic region is plotted with CNVkit's "scatter" command, the size of the plotted datapoints is proportional to each bin's weight – a relatively small point indicates a less reliable bin.

3.1.7 Segmented log2 ratios (.cns)

In addition to the `chromosome`, `start`, `end`, `gene`, `log2`, `depth` and `weight` columns present in .cnr files, the .cns file format has the additional column `probes`, indicating the number of bins covered by the segment.

The **gene** column concatenates the gene names of all the bins that the segment covers. The **weight** column sums the bin-level weights, and the **depth** and **log2** is the weighted mean of the input bin-level values corresponding to the segment.

3.2 Chromosomal sex

CNVkit attempts to handle chromosomal sex correctly at every step. Several commands automatically infer a given sample's chromosomal sex from the relative copy number of the autosomes and chromosomes X and Y; the status log messages will indicate when this is happening. In most cases the inference can be skipped or overridden by using the `-x`/`--sample-sex` option.

The *sex* command runs and report's CNVkit's inference for one or more given samples, and can be used on .cnn, .cnr or .cns files at any stage of processing.

3.2.1 Reference sex-chromosome ploidy

By default, copy number calls and log2 ratios will be relative to a diploid X chromosome and haploid Y. This happens regardless of the control samples used in the *reference* command; if any input samples are male (haploid X), their X chromosome coverage depths are doubled in order to be equivalent to diploid X.

However, some users prefer calls relative to a haploid X chromosome – this is how array CGH data are usually presented, for example. In that context it's often referred to as a "male reference". This convention can be enabled in CNVkit by using the `-y` / `--male-reference` / `--haploid-x-reference` option. Note that this does not require any of the control samples to be male; female control samples' X coverage depth is automatically halved so that it appears as haploid in the CNVkit pipeline. Chromosome Y is always treated as haploid in either case.

3.2.2 Chromosomal sex in calling absolute copy number

If `-y` is used to construct a reference, then the same option should also be used with the commands *call*, *export*, and *genemetrics* for samples processed with that reference.

Note that the options `-x`/`--sample-sex` and `-y` / `--male-reference` / `--haploid-x-reference` are different: If a female sample is run with a haploid-X reference, segments on chromosome X with log2-ratio +1 will be treated as copy-number-neutral, because that's the expected copy number, while an X-chromosome segment with log2-ratio 0 will be treated as a hemizygous loss.

3.2.3 Plots and sex chromosomes

diagram adjusts the sex chromosomes for sample and reference sex so that gains and losses on chromosomes X and Y are highlighted and labeled appropriately.

scatter and *heatmap* do not adjust the sex chromosomes for sample or reference sex.

3.2.4 FAQ

Why does chromosome X/Y show a gain/loss?

The copy number status of sex chromosomes may be surprising if you are unsure about the sex of the samples and reference used:

- Female samples normalized to a male reference will show a doubling of chromosome X (log2 value about +1.0) and complete loss of chromosome Y (log2 value below -3.0, usually far below).
- Male samples normalized to a female reference will show a single-copy loss of chromosome X (log2 value about -1.0). The chromosome Y value should be near 0.0, but the log2 values may be noisier and less reliable than on autosomes.

In the output of the *diagram*, *call*, and *export* commands, the X-chromosome copy number may be wrong if the X-ploidy of the reference (`-y`) or sample (`-x`) was not specified correctly. If sample sex was not specified on the command line, check the command's logged status messages to see if the sample's sex was guessed incorrectly.

After you've verified the above, the CNV might be real.

CNVkit is not detecting my sample's sex correctly. What can I do?

In lower-quality samples, particularly tumor samples analyzed without a robust reference (see *Tumor analysis*), there may be many bins with no coverage which bias the segment means. Try repeating the *segment* command with the `--drop-low-coverage` option if you did not do so originally.

See also: https://www.biostars.org/p/210080/

3.3 Calling copy number gains and losses

The relationship between the observed copy ratio and the true underlying copy number depends on tumor cell fraction (purity), genome ploidy (which may be heterogeneous in a tissue sample), and the size of the subclonal population containing the CNA. Because of these ambiguities, CNVkit initially reports only the estimated log2 copy ratio, and supports several approaches to deriving absolute integer copy number or evaluating copy number gains and losses.

In a diploid genome, a single-copy gain in a perfectly pure, homogeneous sample has a copy ratio of 3/2. In log2 scale, this is log2(3/2) = 0.585, and a single-copy loss is log2(1/2) = -1.0.

In the *diagram* plot, for the sake of providing a clean visualization of confidently called CNAs, the default threshold to label genes is 0.5. This threshold will tend to display gene amplifications, fully clonal single-copy gains in fairly pure samples, most single-copy losses, and total (homozygous) deletions.

When using the *genemetrics* command, choose a threshold to suit your needs depending on your knowledge of the sample's purity, heterogeneity, and likely features of interest. As a starting point, try 0.1 or 0.2 if you are going to do your own filtering downstream, or 0.3 if not.

To evaluate the level of support for each segment, use the *segmetrics* command. In particular, the *–ci* option estimates the confidence interval of the segment mean: the difference between the upper and lower limits suggests the reliability of the estimate, and the value of the upper or lower limit suggests the minimum loss or gain value, respectively, supported by the bin-level log2 ratios in that segment.

The *call* command implements two simple methods to convert the log2 ratios in a segmented .cns file to absolute integer copy number values. Given known or estimated tumor purity and ploidy values, this command can also adjust for *tumor heterogeneity*.

A .cns file can be converted to BED or VCF format using the *export* command. These output styles display the inferred absolute integer copy number value of each segment. To first adjust for known purity and ploidy of the sample, or apply log2 thresholds for integer calls, use *call* before exporting.

3.4 Allele frequencies and copy number

3.4.1 What is BAF?

In this context, the "B" allele is the non-reference allele observed in a germline heterozygous SNP, i.e. in the normal/control sample. Since the tumor cells' DNA originally derived from normal cells' DNA, most of these SNPs will also be present in the tumor sample. But due to allele-specific copy number alterations, loss of heterozygosity or allelic imbalance, the allelic frequency of these SNPs may be different in the tumor, and that's evidence that one (or both) of the germline copies was gained or lost during tumor evolution.

The shift in b-allele frequency is calculated relative to the expected heterozygous frequency 0.5, and minor allele frequencies are "mirrored" above and below 0.5 so that it does not matter which allele is considered the reference – the relative shift from 0.5 will be the same either way. (Multiple alternate alleles are not considered here.)

3.4.2 How does it work?

Estimation from a SNP b-allele frequencies works by comparing the shift in allele frequency of heterozygous, germline SNPs in the tumor sample from the expected ~50% – e.g. a 3-copy segment in a diploid genome would have log2 ratio of +0.58 and heterozygous SNPs would have an average BAF of 67% or 33% if the tumor sample is fully clonal, and closer to log2 0.0 and BAF 0.5 if there is normal-cell contamination and/or *tumor heterogeneity*.

Typically you would use a properly formatted *VCF* from joint tumor-normal SNV calling, e.g. the output of MuTect, VarDict, or FreeBayes, having already flagged somatic mutations so they can be skipped in this analysis. If you have no matched normal sample for a given tumor, you can use 1000 Genomes common SNP sites to extract the likely germline SNVs from a tumor-only VCF, and use just those sites with THetA2 (or another tool like PyClone or BubbleTree).

Use SNP b-allele frequencies from a VCF in these commands:

- *call*
- *scatter*
- *export* `nexus-ogt` and `theta`

3.5 Bias corrections

The sequencing coverage depth obtained at different genomic regions is variable, particularly for targeted capture. Much of this variability is due to known biochemical effects related to library prep, target capture and sequencing. Normalizing the read depth in each on– and off-target bin to the expected read depths derived from a reference of normal samples removes much of the biases attributable to GC content, target density. However, these biases also vary between samples, and must still be corrected even when a normal reference is available.

To correct each of these known effects, CNVkit calculates the relationship between observed bin-level read depths and the values of some known biasing factor, such as GC content. This relationship is fitted using a simple rolling median, then subtracted from the original read depths in a sample to yield corrected estimates.

In the case of many similarly sized target regions, there is the potential for the bias value to be identical for many targets, including some spatially near each other. To ensure that the calculated biases are independent of genomic position, the probes are randomly shuffled before being sorted by bias value.

The GC content and repeat-masked fraction of each bin are calculated during generation of the *reference* from the user-supplied genome. The bias corrections are then performed in the *reference* and *fix* commands.

3.5.1 GC content

Genomic regions with extreme GC content, the fraction of sequence composed of guanine or cytosine bases, are less amenable to hybridization, amplification and sequencing, and will generally appear to have lower coverage than regions of average GC content.

To correct this bias in each sample, CNVkit calculates the association between each bin's GC content (stored in the reference) and observed read depth, fits a trendline through the bin read depths ordered by GC value, and subtracts this trend from the original read depths.

3.5.2 Sequence repeats

Repetitive elements in the genome can be masked out with RepeatMasker – and the genome sequences provided by the UCSC Genome Bioinformatics Site have this masking applied already. The fraction of each genomic bin masked out for repetitiveness indicates both low mappability and the susceptibility to Cot-1 blocking, both of which can reduce the bin's observed coverage.

CNVkit removes the association between repeat-masked fraction and bin read depths for each sample similarly to the GC correction.

3.5.3 Targeting density

In hybridization capture, two biases occur near the edge of each baited region:

- Within the baited region, read depth is lower at the "shoulders" where sequence fragments are not completely captured.
- Just outside the baited region, in the "flanks", read depth is elevated to nearly that of the adjacent baited sites due to the same effect. If two targets are very close together, the sequence fragments captured for one target can increase the read depth in the adjacent target.

CNVkit roughly estimates the potential for these two biases based on the size and position of each baited region and its immediate neighbors. The biases are modeled together as a linear decrease in read depth from inside the target region to the same distance outside. These biases occur within a distance of the interval edges equal to the sequence fragment size (also called the insert size for paired-end sequencing reads). Density biases are calculated from the start and end positions of a bin and its neighbors within a fixed window around the target's genomic coordinates equal to the sequence fragment size.

Shoulder effect: Letting i be the average insert size and t be the target interval size, the negative bias at interval shoulders is calculated as $i/4t$ at each side of the interval, or $i/2t$ for the whole interval. When the interval is smaller than the sequence fragment size, the portion of the fragment extending beyond the opposite edge of the interval should not be counted in this calculation. Thus, if $t < i$, the negative bias value must be increased (absolute value reduced) by $\frac{(i-t)^2}{2it}$.

Flank effect: Additionally letting g be the size of the gap between consecutive intervals, the positive bias that occurs when the gap is smaller than the insert size ($g < i$) is $\frac{(i-g)^2}{4it}$. If the target interval and gap together are smaller than the insert size, the reads flanking the neighboring interval may extend beyond the target, and this flanking portion beyond the target should not be counted. Thus, if $t + g < i$, the positive value must be reduced by $\frac{(i-g-t)^2}{4it}$. If a target has no close neighbors ($g > i$, the common case), the "flank" bias value is 0.

These values are combined into a single value by subtracting the estimated shoulder biases from the flank biases. The result is a negative number between -1 and 0, or 0 for a target with immediately adjacent targets on both sides. Thus, subdividing a large targeted interval into a consecutive series of smaller targets does not change the net "density" calculation value.

The association between targeting density and bin read depths is then fitted and subtracted, as with GC and RepeatMasker.

CNVkit applies the density bias correction to only the on-target bins; the negative "shoulder" bias is not expected to occur in off-target regions because those regions are not specifically captured by baits, and the positive "flank" bias from neighboring targets is avoided by allocating off-target bins around existing targets with a margin of twice the expected insert size.

3.6 Tumor analysis

CNVkit has been used most extensively on solid tumor samples sequenced with a target panel or whole-exome sequencing protocol. Several options and approaches are available to support this use case:

- If you have unpaired tumor samples, or no normal samples sequenced on the same platform, see the *reference* command for strategies.
- Use `--drop-low-coverage` to ignore bins with log2 normalized coverage values below -15. Virtually all tumor samples, even cancer cell lines, are not completely homogeneous. Even in regions of homozygous deletion in the largest tumor-cell clonal population, some sequencing reads will be obtained from contaminating normal cells without the deletion. Therefore, extremely low log2 copy ratio values do not indicate homozygous deletions but failed sequencing or mapping in all cells regardless of copy number status at that site, which are not informative for copy number. This option in the *batch* command applies to segmentation; the option is also available in the *segment*, *metrics*, *segmetrics*, *genemetrics* and *Tumor heterogeneity* commands.
  - Why -15? The null log2 value substituted for bins with zero coverage is -20 (about 1 millionth the average bin's coverage), and the maximum positive shift that can be introduced by normalizing to the reference is 5 (for bins with 1/32 the average coverage; bins below this are masked out by the reference). In a .cnr file, any bins with log2 value below -15 are probably based on dummy values corresponding to zero-coverage (perhaps unmappable) bins, and not real observations.
- The *batch* command does not directly output integer copy number calls (see *Tumor heterogeneity*). Instead, use the `--ploidy` and `--purity` options in *call* to calculate copy number for each sample individually using known or estimated tumor-cell fractions. Also consider using `--center median` in highly aneuploid samples to shift the log2 value of true neutral regions closer to zero, as it may be slightly off initially.
- If SNV calls are available in VCF format, use the `-v`/`--vcf` option in the *call* and *scatter* commands to calculate or plot b-allele frequencies alongside each segment's total copy number or log2 ratio. These values reveal allelic imbalance and loss of heterozygosity (LOH), supporting and extending the inferred CNVs.

3.7 Tumor heterogeneity

DNA samples extracted from solid tumors are rarely completely pure. Stromal or other normal cells and distinct subclonal tumor-cell populations are typically present in a sample, and can confound attempts to fit segmented log2 ratio values to absolute integer copy numbers.

CNVkit provides several points of integration with existing tools and methods for dealing with tumor heterogeneity and normal-cell contamination.

3.7.1 Estimating tumor purity and normal contamination

A rough estimate of tumor purity can usually be obtained using one or more of these approaches:

1. A pathologist can visually estimate the purity of an sample taken from a solid tumor by examination under a microscope, counting stromal and neoplastic cells.
2. If the tumor is belived to be driven by a somatic point mutation, e.g. BRAF V600E in melanoma, then that mutation is assumed to be fully clonal and its allele frequency indicates the tumor purity. This can be complicated by copy number alterations at the same site and whether the point mutation is homozygous or heterozygous, but the frequencies of other somatic mutations in the same sample may resolve this satisfactorily.
3. Larger-scale, hemizygous losses that cover germline heterozygous SNPs shift the allele frequencies of the same SNPs as they are present in the tumor sample. In a 50% pure tumor sample, for example, these SNP b-allele frequencies would shift from 50% to 67% or 33%, assuming a diploid sample (i.e. 1 of 2 copies from the normal

sample and 0 or 1 of 1 copy from the tumor, depending on whether the variant allele was lost or retained). The general calculation is a bit more complicated than in #1 or #2, and can be done similarly for copy number gains and homozygous deletions.

4. The log2 ratio values of CNAs in a tumor sample correspond to integer copy numbers in tumor cells, and in aggregate these log2 values will cluster around values that indicate subclone populations, each with a given ploidy and clonality. For example, a single-copy loss in a 50% pure tumor sample will have 3/4 the coverage of a neutral site (2/2 normal copies, 1/2 tumor copies), for a log2 value of log2(.75) = -0.415. This calculation can also be generalized to other copy number states.

Software implementations of the latter three approaches can be used directly on DNA sequencing data.

3.7.2 Inferring tumor purity and subclonal population fractions from sequencing

While inferring the tumor population structure is currently out of the scope of CNVkit, this work can be done using other third-party programs such as PureCN, THetA2, PyClone, or BubbleTree. Each of these programs can be used to estimate tumor cell content and infer integer copy number of tumor subclones in a sample.

PureCN accepts CNVKit's .cnn and .cnr files directly as input, and is recommended.

Using CNVkit with THetA2

CNVkit provides wrappers for exporting .cns segments to THetA2's input format and importing THetA2's result file as CNVkit's segmented .cns files. See the commands *theta* and *import-theta* for usage instructions.

After running the CNVkit *Copy number calling pipeline* on a sample, and calling SNVs jointly on the tumor and normal samples, generate the THetA2 input files from the .cns and .vcf files:

```
cnvkit.py export theta Sample_T.cns reference.cnn -v Sample_Paired.vcf
```

This produces three output files: `Sample_T.interval_count`, `Sample_T.tumor.snp_formatted.txt`, and `Sample_T.normal.snp_formatted.txt`.

Then, run THetA2 (assuming the program was unpacked at `/path/to/theta2/`):

```
# Generates Sample_T.BEST.results:
/path/to/theta2/bin/RunTHetA Sample_T.interval_count \
    --TUMOR_FILE Sample_T.tumor.snp_formatted.txt \
    --NORMAL_FILE Sample_T.normal.snp_formatted.txt \
    --BAF --NUM_PROCESSES `nproc` --FORCE
```

Finally, import THetA2's results back into CNVkit's .cns format, matching the original segmentation (.cns) to the THetA2-inferred absolute copy number values.:

```
cnvkit.py import-theta Sample_T.cns Sample_T.BEST.results
```

THetA2 adjusts the segment log2 values to the inferred cellularity of each detected subclone; this can result in one or two .cns files representing subclones if more than one clonal tumor cell population was detected. THetA2 also performs some significance testing of each segment representing a CNA, so there may be fewer segments derived from THetA2 than were originally found by CNVkit.

The segment values are still log2-transformed in the resulting .cns files, for convenience in plotting etc. with CNVkit. These files are also easily converted to other formats using the *export* command.

3.7.3 Adjusting copy ratios and segments for normal cell contamination

CNVkit's *call* command uses an estimate of tumor fraction (from any source) to directly rescale segment log2 ratio values, and SNV b-allele frequencies if present, to the value that would be seen a completely pure, uncontaminated sample. Example with tumor purity of 60% and a male reference:

```
cnvkit.py call -m none Sample.cns --purity 0.6 -y -o Sample.call.cns
```

The *call* command can also convert the segmented log2 ratio estimates to absolute integer copy numbers. If the tumor cell fraction is known confidently, use the `-m clonal` method to round the log2 ratios to the nearest integer copy number. Alternatively, the `-m threshold` method to applies hard thresholds. Note that rescaling for purity is optional; either way, integer copy numbers are emitted unless the `-m none` option is used to skip it.

```
cnvkit.py call -m clonal Sample.cns -y --purity 0.65 -o Sample.call.cns
# Or, if already rescaled
cnvkit.py call -m clonal Sample.call.cns -y -o Sample.call.cns
# With CNVkit's default cutoffs
cnvkit.py call -m threshold Sample.cns -y -o Sample.call.cns
# Or, using a custom set of cutoffs
cnvkit.py call -t=-1.1,-0.4,0.3,0.7 Sample.cns -y -o Sample.call.cns
```

3.7.4 Export integer copy numbers as BED or VCF

The *export* `bed` and `vcf` commands emit integer copy number calls in the standard BED or VCF formats:

```
cnvkit.py export bed Sample.call.cns -y -o Sample.bed
cnvkit.py export vcf Sample.call.cns -y -o Sample.vcf
```

If the *.call.cns* files were generated by the *call* command, the integer copy numbers calculated in that step will be exported as well.

3.8 Germline analysis

CNVkit can be used with exome sequencing of constitutional (non-tumor) samples, for example to detect germline copy number alterations associated with heritable conditions. However, note that CNVkit is less accurate in detecting CNVs smaller than 1 Mbp, typically only detecting variants that span multiple exons or captured regions. When used on exome or target panel datasets, CNVkit will not detect the small CNVs that are more common in populations.

To use CNVkit to detect medium-to-large CNVs or unbalanced SVs in constitutional samples:

- The *call* command can be used directly without specifying the `--purity` and `--ploidy` values, as the defaults will be correct for mammalian cells. (For non-diploid species, use the correct `--ploidy`, of course.) The default `--method threshold` assigns integer copy number similarly to `--method clonal`, but with smaller thresholds for calling single-copy changes. The default thresholds allow for mosaicism in CNVs, which have smaller log2 value than a single-copy CNV would indicate. (They're more common than often thought.)
- The `--filter` option in *call* can be used to reduce the number of false-positive segments returned. To use the `ci` (recommended) or `sem` filters, first run each sample's segmented .cns file through *segmetrics* with the `--ci` option, which adds upper and lower confidence limits to the .cns output that `call --filter ci` can then use.
- The `--drop-low-coverage` option (see *Tumor analysis*) should not be used; it will typically remove germline deep deletions altogether, which is not desirable.

- For using CNVkit with whole-genome sequencing datasets, see *Whole-genome sequencing and targeted amplicon capture*.

3.9 Whole-genome sequencing and targeted amplicon capture

CNVkit is primarily designed for use on **hybrid capture** sequencing data, where off-target reads are present and can be used improve copy number estimates. However, CNVkit can also be used on **whole-genome sequencing** (WGS) and **targeted amplicon sequencing** (TAS) datasets by using alternative command-line options.

The *batch* command supports these workflows through the `-m/--method` option.

3.9.1 Whole-Genome Sequencing (WGS)

CNVkit treats WGS data as a capture of all of the genome's sequencing-accessible regions, with no off-target regions.

The `batch --method wgs` option uses the given reference genome's sequencing-accessible regions ("access" BED) as the "targets" – these will be calculated on the fly if not provided. No "antitarget" regions are used. Since the input does not contain useful per-target gene labels, a gene annotation database is required and used to label genes in the outputs:

```
cnvkit.py batch Sample1.bam Sample2.bam -n Control1.bam Control2.bam \
        -m wgs -f hg19.fasta --annotate refFlat.txt
```

To speed up and/or improve the accuracy of WGS analyses, try any or all of the following:

- Instead of analyzing the whole genome, use the "target" BED file to limit the analysis to just the genic regions. You can get such a BED file from the [UCSC Genome Browser](https://genome.ucsc.edu/cgi-bin/hgTables), for example.
- Increase the "target" average bin size (`--target-avg-size`), e.g. to at least 1000 bases for 30x coverage, or proportionally more for lower-coverage sequencing.
- Specify a smaller p-value threshold (`segment -t`). For the CBS method, `1e-6` may work well. Or, try the `flasso` segmentation method.
- Use the `-p/--processes` option in the *batch*, *coverage* and *segment* commands to ensure all available CPUs are used.
- Ensure you are using the most recent version of CNVkit. Each release includes some performance improvements.
- Turn off the "edge" bias correction in the *reference* and *fix* commands (*–no-edge*).

The `batch -m wgs` option does all of these except the first automatically.

3.9.2 Targeted Amplicon Sequencing (TAS)

When amplicon sequencing is used as a targeted capture method, no off-target reads are sequenced. While this limits the copy number information available in the sequencing data versus hybrid capture, CNVkit can analyze TAS data using only on-target coverages and excluding all off-target regions from the analysis.

The `batch -m amplicon` option uses the given targets to infer coverage, ignoring off-target regions:

```
cnvkit.py batch -m amplicon -t targets.bed *.bam
```

Equivalently:

```
cnvkit.py target targets.bed --split -o targets.split.bed
# Create a blank file to substitute for antitargets
touch MT
# For each sample
cnvkit.py coverage Sample.bam targets.split.bed -p 0 -o Sample.targetcoverage.cnn
cnvkit.py reference *.targetcoverage.cnn --no-edge -o ref-tas.cnn
cnvkit.py fix Sample.targetcoverage.cnn MT ref-tas.cnn --no-edge
```

This approach does not collect any copy number information between targeted regions, so it should only be used if you have in fact prepared your samples with a targeted amplicon sequencing protocol. It also does not attempt to further normalize each amplicon at the gene level, though this may be addressed in a future version of CNVkit.

Note: Do not mark duplicates in the BAM files for samples sequenced by this method.

Picard MarkDuplicates, samtools rmdup, *et al.* are designed to flag possible PCR duplicates (originally for WGS datasets, but also useful for hybrid capture). Variant callers like GATK and CNVkit will ignore those reads in their internal calculations, considering these reads to be non-independent measurements. (This SeqAnswers thread has details and background).

In targeted amplicon sequencing, all of the amplified reads are in fact PCR duplicates by design. By marking and thus omitting these reads, the remaining coverage will be low, as if no amplification were performed.

CHAPTER 4

Python API

4.1 cnvlib package

4.1.1 Module `cnvlib` contents

The one function exposed at the top level, *read*, loads a file in CNVkit's BED-like tabular format and returns a CopyNumArray instance. For your own scripting, you can usually accomplish what you need using just the CopyNumArray and GenomicArray methods available on this returned object (see *Core classes*).

To load other file formats, see *Tabular file I/O (tabio)*. To run functions equivalent to CNVkit commands within Python, see *Interface to CNVkit sub-commands*.

4.1.2 Core classes

The core objects used throughout CNVkit. The base class is GenomicArray from *scikit-genome package*. All of these classes wrap a pandas DataFrame instance, which is accessible through the `.data` attribute and can be used for any manipulations that aren't already provided by methods in the wrapper class.

`cnary`

CNVkit's core data structure, a copy number array.

class `cnvlib.cnary.`**CopyNumArray**(*data_table*, *meta_dict=None*)

Bases: *`skgenome.gary.GenomicArray`*

An array of genomic intervals, treated like aCGH probes.

Required columns: chromosome, start, end, gene, log2

Optional columns: gc, rmask, spread, weight, probes

by_gene(*ignore=('-', '.', 'CGH')*)

Iterate over probes grouped by gene name.

Group each series of intergenic bins as an "Antitarget" gene; any "Antitarget" bins within a gene are grouped with that gene.

Bins' gene names are split on commas to accommodate overlapping genes and bins that cover multiple genes.

Parameters **`ignore`** (*`list or tuple of str`*) – Gene names to treat as "Antitarget" bins instead of real genes, grouping these bins with the surrounding gene or intergenic region. These bins will still retain their name in the output.

Yields *tuple* – Pairs of: (gene name, CNA of rows with same name)

`center_all`(*estimator=<unbound method Series.median>*, *by_chrom=True*, *skip_low=False*, *verbose=False*)

Re-center log2 values to the autosomes' average (in-place).

Parameters

- **`estimator`** (*`str or callable`*) – Function to estimate central tendency. If a string, must be one of 'mean', 'median', 'mode', 'biweight' (for biweight location). Median by default.
- **`skip_low`** (*`bool`*) – Whether to drop very-low-coverage bins (via *drop_low_coverage*) before estimating the center value.
- **`by_chrom`** (*`bool`*) – If True, first apply *estimator* to each chromosome separately, then apply *estimator* to the per-chromosome values, to reduce the impact of uneven targeting or extreme aneuploidy. Otherwise, apply *estimator* to all log2 values directly.

`compare_sex_chromosomes`(*male_reference=False*, *skip_low=False*)

Compare coverage ratios of sex chromosomes versus autosomes.

Perform 4 Mood's median tests of the log2 coverages on chromosomes X and Y, separately shifting for assumed male and female chromosomal sex. Compare the chi-squared values obtained to infer whether the male or female assumption fits the data better.

Parameters

- **`male_reference`** (*`bool`*) – Whether a male reference copy number profile was used to normalize the data. If so, a male sample should have log2 values of 0 on X and Y, and female +1 on X, deep negative (below -3) on Y. Otherwise, a male sample should have log2 values of -1 on X and 0 on Y, and female 0 on X, deep negative (below -3) on Y.
- **`skip_low`** (*`bool`*) – If True, drop very-low-coverage bins (via *drop_low_coverage*) before comparing log2 coverage ratios. Included for completeness, but shouldn't affect the result much since the M-W test is nonparametric and p-values are not used here.

Returns

- *bool* – True if the sample appears male.
- *dict* – Calculated values used for the inference: relative log2 ratios of chromosomes X and Y versus the autosomes; the Mann-Whitney U values from each test; and ratios of U values for male vs. female assumption on chromosomes X and Y.

`drop_low_coverage`(*verbose=False*)

Drop bins with extremely low log2 coverage or copy ratio values.

These are generally bins that had no reads mapped due to sample-specific issues. A very small log2 ratio or coverage value may have been substituted to avoid domain or divide-by-zero errors.

`expect_flat_log2`(*is_male_reference=None*)

Get the uninformed expected copy ratios of each bin.

Create an array of log2 coverages like a "flat" reference.

This is a neutral copy ratio at each autosome (log2 = 0.0) and sex chromosomes based on whether the reference is male (XX or XY).

guess_xx (*male_reference=False*, *verbose=True*)

Detect chromosomal sex; return True if a sample is probably female.

Uses *compare_sex_chromosomes* to calculate coverage ratios of the X and Y chromosomes versus autosomes.

Parameters

- **male_reference** (`bool`) – Was this sample normalized to a male reference copy number profile?
- **verbose** (`bool`) – If True, print (i.e. log to console) the ratios of the log2 coverages of the X and Y chromosomes versus autosomes, the "maleness" ratio of male vs. female expectations for each sex chromosome, and the inferred chromosomal sex.

Returns True if the coverage ratios indicate the sample is female.

Return type bool

log2

residuals (*segments=None*)

Difference in log2 value of each bin from its segment mean.

Parameters segments (`GenomicArray`, `CopyNumArray`, `or None`) – Determines the "mean" value to which *self* log2 values are relative:

- If CopyNumArray, use the log2 values as the segment means to subtract.
- If GenomicArray with no log2 values, group *self* by these ranges and subtract each group's median log2 value.
- If None, subtract each chromosome's median.

Returns Residual log2 values from *self* relative to *segments*; same length as *self*.

Return type array

shift_xx (*male_reference=False*, *is_xx=None*)

Adjust chrX log2 ratios to match the ploidy of the reference sex.

I.e. add 1 to chrX log2 ratios for a male sample vs. female reference, or subtract 1 for a female sample vs. male reference, so that chrX log2 values are comparable across samples with different chromosomal sex.

smoothed (*window=None*, *by_arm=True*)

Smooth log2 values with a sliding window.

Account for chromosome and (optionally) centromere boundaries. Use bin weights if present.

Returns Smoothed log2 values from *self*, the same length as *self*.

Return type array

squash_genes (*summary_func=<function biweight_location>*, *squash_antitarget=False*, *ignore=('-', '.', 'CGH')*)

Combine consecutive bins with the same targeted gene name.

Parameters

- **summary_func** (`callable`) – Function to summarize an array of log2 values to produce a new log2 value for a "squashed" (i.e. reduced) region. By default this is the biweight location, but you might want median, mean, max, min or something else in some cases.

- **`squash_antitarget`** (*`bool`*) – If True, also reduce consecutive "Antitarget" bins into a single bin. Otherwise, keep "Antitarget" and ignored bins as they are in the output.
- **`ignore`** (*`list or tuple of str`*) – Bin names to be treated as "Antitarget" instead of as unique genes.

Returns Another, usually smaller, copy of *self* with each gene's bins reduced to a single bin with appropriate values.

Return type *CopyNumArray*

`vary`

An array of genomic intervals, treated as variant loci.

class `cnvlib.vary.`**`VariantArray`**(*data_table*, *meta_dict=None*)

Bases: `skgenome.gary.GenomicArray`

An array of genomic intervals, treated as variant loci.

Required columns: chromosome, start, end, ref, alt

`baf_by_ranges`(*ranges*, *summary_func=<function nanmedian>*, *above_half=None*, *tumor_boost=False*)

Aggregate variant (b-allele) frequencies in each given bin.

Get the average BAF in each of the bins of another genomic array: BAFs are mirrored above/below 0.5 (per *above_half*), grouped in each bin of *ranges*, and summarized into one value per bin with *summary_func* (default median).

Parameters

- **`ranges`** (`GenomicArray or subclass`) – Bins for grouping the variants in *self*.
- **`above_half`** (*`bool`*) – The same as in *mirrored_baf*.
- **`tumor_boost`** (*`bool`*) – The same as in *mirrored_baf*.

Returns Average b-allele frequency in each range; same length as *ranges*. May contain NaN values where no variants overlap a range.

Return type float array

`heterozygous`()

Subset to only heterozygous variants.

Use 'zygosity' or 'n_zygosity' genotype values (if present) to exclude variants with value 0.0 or 1.0. If these columns are missing, or there are no heterozygous variants, then return the full (input) set of variants.

Returns The subset of *self* with heterozygous genotype, or allele frequency between the specified thresholds.

Return type *VariantArray*

`mirrored_baf`(*above_half=None*, *tumor_boost=False*)

Mirrored B-allele frequencies (BAFs).

Parameters

- **above_half** (*bool or None*) – If specified, flip BAFs to be all above 0.5 (True) or below 0.5 (False), respectively, for consistency. Otherwise, if None, mirror in the direction of the majority of BAFs.
- **tumor_boost** (*bool*) – Normalize tumor-sample allele frequencies to the matched normal sample's allele frequencies.

Returns Mirrored b-allele frequencies, the same length as *self*. May contain NaN values.

Return type float array

tumor_boost()

TumorBoost normalization of tumor-sample allele frequencies.

De-noises the signal for detecting LOH.

See: TumorBoost, Bengtsson et al. 2010

zygosity_from_freq(*het_freq=0.0*, *hom_freq=1.0*)

Set zygosity (genotype) according to allele frequencies.

Creates or replaces 'zygosity' column if 'alt_freq' column is present, and 'n_zygosity' if 'n_alt_freq' is present.

Parameters

- **het_freq** (*float*) – Assign zygosity 0.5 (heterozygous), otherwise 0.0 (i.e. reference genotype), to variants with alt allele frequency of at least this value.
- **hom_freq** (*float*) – Assign zygosity 1.0 (homozygous) to variants with alt allele frequency of at least this value.

4.1.3 Interface to CNVkit sub-commands

commands

The public API for each of the commands defined in the CNVkit workflow.

Command-line interface and corresponding API for CNVkit.

cnvlib.commands.**do_target**(*bait_arr*, *annotate=None*, *do_short_names=False*, *do_split=False*, *avg_size=266.6666666666667*)

Transform bait intervals into targets more suitable for CNVkit.

cnvlib.commands.**do_access**(*fa_fname*, *exclude_fnames=()*, *min_gap_size=5000*, *skip_noncanonical=True*)

List the locations of accessible sequence regions in a FASTA file.

cnvlib.commands.**do_antitarget**(*targets*, *access=None*, *avg_bin_size=150000*, *min_bin_size=None*)

Derive off-targt ("antitarget") bins from target regions.

cnvlib.commands.**do_autobin**(*bam_fname*, *method*, *targets=None*, *access=None*, *bp_per_bin=100000.0*, *target_min_size=20*, *target_max_size=50000*, *antitarget_min_size=500*, *antitarget_max_size=1000000*)

Quickly calculate reasonable bin sizes from BAM read counts.

Parameters

- **bam_fname** (*string*) – BAM filename.
- **method** (*string*) – One of: 'wgs' (whole-genome sequencing), 'amplicon' (targeted amplicon capture), 'hybrid' (hybridization capture).

- **`targets`** (`GenomicArray`) – Targeted genomic regions (for 'hybrid' and 'amplicon').
- **`access`** (`GenomicArray`) – Sequencing-accessible regions of the reference genome (for 'hybrid' and 'wgs').
- **`bp_per_bin`** (*int*) – Desired number of sequencing read nucleotide bases mapped to each bin.

Returns

((target depth, target avg. bin size), (antitarget depth, antitarget avg. bin size))

Return type 2-tuple of 2-tuples

`cnvlib.commands.`**`do_coverage`**(*bed_fname*, *bam_fname*, *by_count=False*, *min_mapq=0*, *processes=1*)

Calculate coverage in the given regions from BAM read depths.

`cnvlib.commands.`**`do_reference`**(*target_fnames*, *antitarget_fnames=None*, *fa_fname=None*, *male_reference=False*, *female_samples=None*, *do_gc=True*, *do_edge=True*, *do_rmask=True*)

Compile a coverage reference from the given files (normal samples).

`cnvlib.commands.`**`do_reference_flat`**(*targets*, *antitargets=None*, *fa_fname=None*, *male_reference=False*)

Compile a neutral-coverage reference from the given intervals.

Combines the intervals, shifts chrX values if requested, and calculates GC and RepeatMasker content from the genome FASTA sequence.

`cnvlib.commands.`**`do_fix`**(*target_raw*, *antitarget_raw*, *reference*, *do_gc=True*, *do_edge=True*, *do_rmask=True*)

Combine target and antitarget coverages and correct for biases.

`cnvlib.commands.`**`do_segmentation`**(*cnarr*, *method*, *threshold=None*, *variants=None*, *skip_low=False*, *skip_outliers=10*, *min_weight=0*, *save_dataframe=False*, *rlibpath=None*, *rscript_path='Rscript'*, *processes=1*)

Infer copy number segments from the given coverage table.

`cnvlib.commands.`**`do_call`**(*cnarr*, *variants=None*, *method='threshold'*, *ploidy=2*, *purity=None*, *is_reference_male=False*, *is_sample_female=False*, *filters=None*, *thresholds=(-1.1*, *-0.25*, *0.2*, *0.7)*)

`cnvlib.commands.`**`do_scatter`**(*cnarr*, *segments=None*, *variants=None*, *show_range=None*, *show_gene=None*, *do_trend=False*, *by_bin=False*, *window_width=1000000.0*, *y_min=None*, *y_max=None*, *antitarget_marker=None*, *segment_color='darkorange'*, *title=None*)

Plot probe log2 coverages and segmentation calls together.

`cnvlib.commands.`**`do_heatmap`**(*cnarrs*, *show_range=None*, *do_desaturate=False*, *by_bin=False*)

Plot copy number for multiple samples as a heatmap.

`cnvlib.commands.`**`do_breaks`**(*probes*, *segments*, *min_probes=1*)

List the targeted genes in which a copy number breakpoint occurs.

`cnvlib.commands.`**`do_genemetrics`**(*cnarr*, *segments=None*, *threshold=0.2*, *min_probes=3*, *skip_low=False*, *male_reference=False*, *is_sample_female=None*)

Identify targeted genes with copy number gain or loss.

`cnvlib.commands.`**`do_genemetrics`**(*cnarr*, *segments=None*, *threshold=0.2*, *min_probes=3*, *skip_low=False*, *male_reference=False*, *is_sample_female=None*)

Identify targeted genes with copy number gain or loss.

`cnvlib.commands.`**`do_sex`**(*cnarrs*, *is_male_reference*)

Guess samples' sex from the relative coverage of chromosomes X and Y.

`cnvlib.commands.`**`do_sex`**(*cnarrs*, *is_male_reference*)

Guess samples' sex from the relative coverage of chromosomes X and Y.

`cnvlib.commands.`**`do_metrics`**(*cnarrs*, *segments=None*, *skip_low=False*)

Compute coverage deviations and other metrics for self-evaluation.

`cnvlib.commands.`**`do_segmetrics`**(*cnarr*, *segarr*, *location_stats=()*, *spread_stats=()*, *interval_stats=()*, *alpha=0.05*, *bootstraps=100*)

Compute segment-level metrics from bin-level log2 ratios.

`cnvlib.commands.`**`do_import_theta`**(*segarr*, *theta_results_fname*, *ploidy=2*)

`cnvlib.commands.`**`do_import_rna`**(*gene_count_fnames*, *in_format*, *gene_resource_fname*, *correlations_fname=None*, *normal_fnames=()*, *do_gc=True*, *do_txlen=True*, *max_log2=3*)

Convert a cohort of per-gene read counts to CNVkit .cnr format.

The expected data source is TCGA gene-level expression counts for individual samples, but other sources should be fine, too.

The following modules implement lower-level functionality specific to each of the CNVkit sub-commands.

`access`

List the locations of accessible sequence regions in a FASTA file.

Inaccessible regions, e.g. telomeres and centromeres, are masked out with N in the reference genome sequence; this script scans those to identify the coordinates of the accessible regions (those between the long spans of N's).

`cnvlib.access.`**`do_access`**(*fa_fname*, *exclude_fnames=()*, *min_gap_size=5000*, *skip_noncanonical=True*)

List the locations of accessible sequence regions in a FASTA file.

`cnvlib.access.`**`drop_noncanonical_contigs`**(*region_tups*)

Drop contigs with noncanonical names.

region_tups is an iterable of (chrom, start, end) tuples.

Yield the same, but dropping noncanonical chrom.

`cnvlib.access.`**`get_regions`**(*fasta_fname*)

Find accessible sequence regions (those not masked out with 'N').

`cnvlib.access.`**`join_regions`**(*regions*, *min_gap_size*)

Filter regions, joining those separated by small gaps.

`cnvlib.access.`**`log_this`**(*chrom*, *run_start*, *run_end*)

Log a coordinate range, then return it as a tuple.

`antitarget`

Supporting functions for the 'antitarget' command.

`cnvlib.antitarget.`**`compare_chrom_names`**(*a_regions*, *b_regions*) `cnvlib.antitarget.`**`do_antitarget`**(*targets*, *access=None*, *avg_bin_size=150000*, *min_bin_size=None*)

Derive off-targt ("antitarget") bins from target regions.

`cnvlib.antitarget.`**`drop_noncanonical_contigs`**(*accessible*, *targets*, *verbose=True*)

Drop contigs that are not targeted or canonical chromosomes.

Antitargets will be binned over chromosomes that:

- Appear in the sequencing-accessible regions of the reference genome sequence, and
- Contain at least one targeted region, or
- Are named like a canonical chromosome (1-22,X,Y for human)

This allows antitarget binning to pick up canonical chromosomes that do not contain any targets, as well as non-canonical or oddly named chromosomes that were targeted.

`cnvlib.antitarget.`**`get_antitargets`**(*targets*, *accessible*, *avg_bin_size*, *min_bin_size*)

Generate antitarget intervals between/around target intervals.

Procedure:

- Invert target intervals
- Subtract the inverted targets from accessible regions
- For each of the resulting regions:
  - Shrink by a fixed margin on each end
  - If it's smaller than min_bin_size, skip
  - Divide into equal-size (region_size/avg_bin_size) portions
  - Emit the (chrom, start, end) coords of each portion

`cnvlib.antitarget.`**`guess_chromosome_regions`**(*targets*, *telomere_size*)

Determine (minimum) chromosome lengths from target coordinates.

`cnvlib.antitarget.`**`is_canonical_contig_name`**(*name*)

autobin

Estimate reasonable bin sizes from BAM read counts or depths.

`cnvlib.autobin.`**`average_depth`**(*rc_table*, *read_length*)

Estimate the average read depth across the genome.

Returns Median of the per-chromosome mean read depths, weighted by chromosome size.

Return type float

`cnvlib.autobin.`**`do_autobin`**(*bam_fname*, *method*, *targets=None*, *access=None*, *bp_per_bin=100000.0*, *target_min_size=20*, *target_max_size=50000*, *antitarget_min_size=500*, *antitarget_max_size=1000000*)

Quickly calculate reasonable bin sizes from BAM read counts.

Parameters

- **`bam_fname`** (`string`) – BAM filename.
- **`method`** (`string`) – One of: 'wgs' (whole-genome sequencing), 'amplicon' (targeted amplicon capture), 'hybrid' (hybridization capture).
- **`targets`** (`GenomicArray`) – Targeted genomic regions (for 'hybrid' and 'amplicon').

- **access** (`GenomicArray`) – Sequencing-accessible regions of the reference genome (for ‘hybrid’ and ‘wgs’).
- **bp_per_bin** (*int*) – Desired number of sequencing read nucleotide bases mapped to each bin.

Returns

((target depth, target avg. bin size), (antitarget depth, antitarget avg. bin size))

Return type 2-tuple of 2-tuples

`cnvlib.autobin.`**`hybrid`**(*rc_table*, *read_len*, *bam_fname*, *targets*, *access=None*)
Hybrid capture sequencing.

`cnvlib.autobin.`**`idxstats2ga`**(*table*, *bam_fname*)

`cnvlib.autobin.`**`midsize_file`**(*fnames*)
Select the median-size file from several given filenames.

If an even number of files is given, selects the file just below the median.

`cnvlib.autobin.`**`region_size_by_chrom`**(*regions*)

`cnvlib.autobin.`**`sample_midsize_regions`**(*regions*, *max_num*)
Randomly select a subset of up to *max_num* regions.

`cnvlib.autobin.`**`sample_region_cov`**(*bam_fname*, *regions*, *max_num=100*)
Calculate read depth in a randomly sampled subset of regions.

`cnvlib.autobin.`**`shared_chroms`**(**tables*)
Intersection of DataFrame .chromosome values.

`cnvlib.autobin.`**`total_region_size`**(*regions*)
Aggregate area of all genomic ranges in *regions*.

`cnvlib.autobin.`**`update_chrom_length`**(*rc_table*, *regions*)

batch

The ‘batch’ command.

`cnvlib.batch.`**`batch_make_reference`**(*normal_bams*, *target_bed*, *antitarget_bed*, *male_reference*, *fasta*, *annotate*, *short_names*, *target_avg_size*, *access_bed*, *antitarget_avg_size*, *antitarget_min_size*, *output_reference*, *output_dir*, *processes*, *by_count*, *method*)
Build the CN reference from normal samples, targets and antitargets.

`cnvlib.batch.`**`batch_run_sample`**(*bam_fname*, *target_bed*, *antitarget_bed*, *ref_fname*, *output_dir*, *male_reference*, *plot_scatter*, *plot_diagram*, *rlibpath*, *by_count*, *skip_low*, *method*, *processes*)
Run the pipeline on one BAM file.

`cnvlib.batch.`**`batch_write_coverage`**(*bed_fname*, *bam_fname*, *out_fname*, *by_count*, *processes*)
Run coverage on one sample, write to file.

call

Call copy number variants from segmented log2 ratios.

`cnvlib.call.`**`absolute_clonal`**(*cnarr*, *ploidy*, *purity*, *is_reference_male*, *is_sample_female*)
Calculate absolute copy number values from segment or bin log2 ratios.

`cnvlib.call.`**`absolute_dataframe`**(*cnarr*, *ploidy*, *purity*, *is_reference_male*, *is_sample_female*)

Absolute, expected and reference copy number in a DataFrame.

`cnvlib.call.`**`absolute_expect`**(*cnarr*, *ploidy*, *is_sample_female*)

Absolute integer number of expected copies in each bin.

I.e. the given ploidy for autosomes, and XY or XX sex chromosome counts according to the sample's specified chromosomal sex.

`cnvlib.call.`**`absolute_pure`**(*cnarr*, *ploidy*, *is_reference_male*)

Calculate absolute copy number values from segment or bin log2 ratios.

`cnvlib.call.`**`absolute_reference`**(*cnarr*, *ploidy*, *is_reference_male*)

Absolute integer number of reference copies in each bin.

I.e. the given ploidy for autosomes, 1 or 2 X according to the reference sex, and always 1 copy of Y.

`cnvlib.call.`**`absolute_threshold`**(*cnarr*, *ploidy*, *thresholds*, *is_reference_male*)

Call integer copy number using hard thresholds for each level.

Integer values are assigned for log2 ratio values less than each given threshold value in sequence, counting up from zero. Above the last threshold value, integer copy numbers are called assuming full purity, diploidy, and rounding up.

Default thresholds follow this heuristic for calling CNAs in a tumor sample: For single-copy gains and losses, assume 50% tumor cell clonality (including normal cell contamination). Then:

```
R> log2(2:6 / 4)
-1.0  -0.4150375  0.0  0.3219281  0.5849625
```

Allowing for random noise of +/- 0.1, the cutoffs are:

```
DEL(0)  <  -1.1
LOSS(1) <  -0.25
GAIN(3) >=  +0.2
AMP(4)  >=  +0.7
```

For germline samples, better precision could be achieved with:

```
LOSS(1) <  -0.4
GAIN(3) >=  +0.3
```

`cnvlib.call.`**`do_call`**(*cnarr*, *variants=None*, *method='threshold'*, *ploidy=2*, *purity=None*, *is_reference_male=False*, *is_sample_female=False*, *filters=None*, *thresholds=(-1.1, -0.25, 0.2, 0.7)*)

`cnvlib.call.`**`log2_ratios`**(*cnarr*, *absolutes*, *ploidy*, *is_reference_male*, *min_abs_val=0.001*, *round_to_int=False*)

Convert absolute copy numbers to log2 ratios.

Optionally round copy numbers to integers.

Account for reference sex & ploidy of sex chromosomes.

`cnvlib.call.`**`rescale_baf`**(*purity*, *observed_baf*, *normal_baf=0.5*)

Adjust B-allele frequencies for sample purity.

Math:

```
t_baf*purity + n_baf*(1-purity) = obs_baf
obs_baf - n_baf * (1-purity) = t_baf * purity
t_baf = (obs_baf - n_baf * (1-purity))/purity
```

coverage

Supporting functions for the 'antitarget' command.

`cnvlib.coverage.`**`bedcov`**(*bed_fname*, *bam_fname*, *min_mapq*)

Calculate depth of all regions in a BED file via samtools (pysam) bedcov.

i.e. mean pileup depth across each region.

`cnvlib.coverage.`**`detect_bedcov_columns`**(*text*)

Determine which 'bedcov' output columns to keep.

Format is the input BED plus a final appended column with the count of basepairs mapped within each row's region. The input BED might have 3 columns (regions without names), 4 (named regions), or more (arbitrary columns after 'gene').

`cnvlib.coverage.`**`do_coverage`**(*bed_fname*, *bam_fname*, *by_count=False*, *min_mapq=0*, *processes=1*)

Calculate coverage in the given regions from BAM read depths.

`cnvlib.coverage.`**`interval_coverages`**(*bed_fname*, *bam_fname*, *by_count*, *min_mapq*, *processes*)

Calculate log2 coverages in the BAM file at each interval.

`cnvlib.coverage.`**`interval_coverages_count`**(*bed_fname*, *bam_fname*, *min_mapq*, *procs=1*)

Calculate log2 coverages in the BAM file at each interval.

`cnvlib.coverage.`**`interval_coverages_pileup`**(*bed_fname*, *bam_fname*, *min_mapq*, *procs=1*)

Calculate log2 coverages in the BAM file at each interval.

`cnvlib.coverage.`**`region_depth_count`**(*bamfile*, *chrom*, *start*, *end*, *gene*, *min_mapq*)

Calculate depth of a region via pysam count.

i.e. counting the number of read starts in a region, then scaling for read length and region width to estimate depth.

Coordinates are 0-based, per pysam.

diagram

Chromosome diagram drawing functions.

This uses and abuses Biopython's BasicChromosome module. It depends on ReportLab, too, so we isolate this functionality here so that the rest of CNVkit will run without it. (And also to keep the codebase tidy.)

`cnvlib.diagram.`**`bc_chromosome_draw_label`**(*self*, *cur_drawing*, *label_name*)

Monkeypatch to Bio.Graphics.BasicChromosome.Chromosome._draw_label.

Draw a label for the chromosome. Mod: above the chromosome, not below.

`cnvlib.diagram.`**`bc_organism_draw`**(*org*, *title*, *wrap=12*)

Modified copy of Bio.Graphics.BasicChromosome.Organism.draw.

Instead of stacking chromosomes horizontally (along the x-axis), stack rows vertically, then proceed with the chromosomes within each row.

Parameters

- **`org`** – The chromosome diagram object being modified.
- **`title`** (*`str`*) – The output title of the produced document.
- **`wrap`** (*`int`*) – Maximum number of chromosomes per row; the remainder will be wrapped to the next row(s).

`cnvlib.diagram.`**`build_chrom_diagram`**(*features*, *chr_sizes*, *sample_id*, *title=None*)

Create a PDF of color-coded features on chromosomes.

`cnvlib.diagram.`**`create_diagram`**(*cnarr*, *segarr*, *threshold*, *min_probes*, *outfname*, *title=None*)

Create the diagram.

`export`

Export CNVkit objects and files to other formats.

`cnvlib.export.`**`assign_ci_start_end`**(*segarr*, *cnarr*)

Assign ci_start and ci_end fields to segments.

Values for each segment indicate the CI boundary points within that segment, i.e. the right CI boundary for the left-side breakpoint (segment start), and left CI boundary for the right-side breakpoint (segment end).

This is a little unintuitive because the CI refers to the breakpoint, not the segment, but we're storing the info in an array of segments.

Calculation: Just use the boundaries of the bins left- and right-adjacent to each segment breakpoint.

`cnvlib.export.`**`export_bed`**(*segments*, *ploidy*, *is_reference_male*, *is_sample_female*, *label*, *show*)

Convert a copy number array to a BED-like DataFrame.

For each region in each sample (possibly filtered according to *show*), the columns are:

- reference sequence name
- start (0-indexed)
- end
- sample name or given label
- integer copy number

By default (show="ploidy"), skip regions where copy number is the default ploidy, i.e. equal to 2 or the value set by –ploidy. If show="variant", skip regions where copy number is neutral, i.e. equal to the reference ploidy on autosomes, or half that on sex chromosomes.

`cnvlib.export.`**`export_gistic_markers`**(*cnr_fnames*)

Generate a GISTIC 2.0 "markers" file from a set of .cnr files.

GISTIC documentation:

ftp://ftp.broadinstitute.org/pub/GISTIC2.0/GISTICDocumentation_standalone.htm http://genepattern.broadinstitute.org/ftp/distribution/genepattern/modules_public_server_doc/GISTIC2.pdf http://gdac.broadinstitute.org/runs/analyses__2013_05_23/reports/cancer/KICH-TP/CopyNumber_Gistic2/nozzle.html

The markers file identifies the marker names and positions of the markers in the original dataset (before segmentation). It is a three column, tab-delimited file with an optional header. The column headers are:

1. Marker Name
2. Chromosome
3. Marker Position (in bases)

GISTIC also needs an accompanying SEG file generated from corresponding .cns files.

`cnvlib.export.`**`export_nexus_basic`**(*cnarr*)

Biodiscovery Nexus Copy Number "basic" format.

Only represents one sample per file.

`cnvlib.export.`**`export_nexus_ogt`**(*cnarr*, *varr*, *min_weight=0.0*)

Biodiscovery Nexus Copy Number "Custom-OGT" format.

To create the b-allele frequencies column, alterate allele frequencies from the VCF are aligned to the .cnr file bins. Bins that contain no variants are left blank; if a bin contains multiple variants, then the frequencies are all "mirrored" to be above or below .5 (majority rules), then the median of those values is taken.

`cnvlib.export.`**`export_seg`**(*sample_fnames*, *chrom_ids=False*)

SEG format for copy number segments.

Segment breakpoints are not the same across samples, so samples are listed in serial with the sample ID as the left column.

`cnvlib.export.`**`export_theta`**(*tumor_segs*, *normal_cn*)

Convert tumor segments and normal .cnr or reference .cnn to THetA input.

Follows the THetA segmentation import script but avoid repeating the pileups, since we already have the mean depth of coverage in each target bin.

The options for average depth of coverage and read length do not matter crucially for proper operation of THetA; increased read counts per bin simply increase the confidence of THetA's results.

THetA2 input format is tabular, with columns: ID, chrm, start, end, tumorCount, normalCount

where chromosome IDs ("chrm") are integers 1 through 24.

`cnvlib.export.`**`export_theta_snps`**(*varr*)

Generate THetA's SNP per-allele read count "formatted.txt" files.

`cnvlib.export.`**`export_vcf`**(*segments*, *ploidy*, *is_reference_male*, *is_sample_female*, *sample_id=None*, *cnarr=None*)

Convert segments to Variant Call Format.

For now, only 1 sample per VCF. (Overlapping CNVs seem tricky.)

Spec: https://samtools.github.io/hts-specs/VCFv4.2.pdf

`cnvlib.export.`**`fmt_cdt`**(*sample_ids*, *table*)

Format as CDT.

See:

- http://jtreeview.sourceforge.net/docs/JTVUserManual/ch02s11.html
- http://www.eisenlab.org/FuzzyK/cdt.html

`cnvlib.export.`**`fmt_gct`**(*sample_ids*, *table*)

`cnvlib.export.`**`fmt_jtv`**(*sample_ids*, *table*)

Format for Java TreeView.

`cnvlib.export.`**`merge_samples`**(*filenames*)

Merge probe values from multiple samples into a 2D table (of sorts).

Input: dict of {sample ID: (probes, values)}

Output: list-of-tuples: (probe, log2 coverages…)

`cnvlib.export.`**`ref_means_nbins`**(*tumor_segs*, *normal_cn*)

Extract segments' reference mean log2 values and probe counts.

Code paths:

```
wt_mdn  wt_old  probes  norm -> norm, nbins
+       *       *       -       0,  wt_mdn
-       +       +       -       0,  wt_old * probes
-       +       -       -       0,  wt_old * size?
-       -       +       -       0,  probes
-       -       -       -       0,  size?

+       -       +       +       norm, probes
+       -       -       +       norm, bin counts
-       +       +       +       norm, probes
-       +       -       +       norm, bin counts
-       -       +       +       norm, probes
-       -       -       +       norm, bin counts
```

cnvlib.export.**segments2vcf**(*segments*, *ploidy*, *is_reference_male*, *is_sample_female*)

Convert copy number segments to VCF records.

cnvlib.export.**theta_read_counts**(*log2_ratio*, *nbins*, *avg_depth=500*, *avg_bin_width=200*, *read_len=100*)

Calculate segments' read counts from log2-ratios.

Math: nbases = read_length * read_count

and nbases = bin_width * read_depth

where read_depth = read_depth_ratio * avg_depth

So: read_length * read_count = bin_width * read_depth read_count = bin_width * read_depth / read_length

fix

Supporting functions for the 'fix' command.

cnvlib.fix.**apply_weights**(*cnarr*, *ref_matched*, *epsilon=0.0001*)

Calculate weights for each bin.

Weights are derived from:

- bin sizes
- average bin coverage depths in the reference
- the "spread" column of the reference.

cnvlib.fix.**center_by_window**(*cnarr*, *fraction*, *sort_key*)

Smooth out biases according to the trait specified by sort_key.

E.g. correct GC-biased bins by windowed averaging across similar-GC bins; or for similar interval sizes.

cnvlib.fix.**do_fix**(*target_raw*, *antitarget_raw*, *reference*, *do_gc=True*, *do_edge=True*, *do_rmask=True*)

Combine target and antitarget coverages and correct for biases.

cnvlib.fix.**edge_gains**(*target_sizes*, *gap_sizes*, *insert_size*)

Calculate coverage gain from neighboring baits' flanking reads.

Letting i = insert size, t = target size, g = gap to neighboring bait, the gain of coverage due to a nearby bait, if g < i, is:

```
.. math :: (i-g)^2 / 4it
```

If the neighbor flank extends beyond the target (t+g < i), reduce by:

```
.. math :: (i-t-g)^2 / 4it
```

If a neighbor overlaps the target, treat it as adjacent (gap size 0).

`cnvlib.fix.`**`edge_losses`**(*target_sizes*, *insert_size*)

Calculate coverage losses at the edges of baited regions.

Letting i = insert size and t = target size, the proportional loss of coverage near the two edges of the baited region (combined) is:

$$i/2t$$

If the "shoulders" extend outside the bait $(t < i), reduce by:

$$(i-t)^2/4it$$

on each side, or (i-t)^2 / 2it total.

`cnvlib.fix.`**`get_edge_bias`**(*cnarr*, *margin*)

Quantify the "edge effect" of the target tile and its neighbors.

The result is proportional to the change in the target's coverage due to these edge effects, i.e. the expected loss of coverage near the target edges and, if there are close neighboring tiles, gain of coverage due to "spill over" reads from the neighbor tiles.

(This is not the actual change in coverage. This is just a tribute.)

`cnvlib.fix.`**`load_adjust_coverages`**(*cnarr*, *ref_cnarr*, *skip_low*, *fix_gc*, *fix_edge*, *fix_rmask*)

Load and filter probe coverages; correct using reference and GC.

`cnvlib.fix.`**`mask_bad_bins`**(*cnarr*)

Flag the bins with excessively low or inconsistent coverage.

Returns A boolean array where True indicates bins that failed the checks.

Return type np.array

`cnvlib.fix.`**`match_ref_to_sample`**(*ref_cnarr*, *samp_cnarr*)

Filter the reference bins to match the sample (target or antitarget).

heatmap

The 'heatmap' command.

`cnvlib.heatmap.`**`do_heatmap`**(*cnarrs*, *show_range=None*, *do_desaturate=False*, *by_bin=False*)

Plot copy number for multiple samples as a heatmap.

`cnvlib.heatmap.`**`set_colorbar`**(*axis*)

importers

Import from other formats to the CNVkit format.

`cnvlib.importers.`**`do_import_picard`**(*fname*, *too_many_no_coverage=100*)

`cnvlib.importers.`**`do_import_theta`**(*segarr*, *theta_results_fname*, *ploidy=2*)

`cnvlib.importers.`**`parse_theta_results`**(*fname*)

Parse THetA results into a data structure.

Columns: NLL, mu, C, p*

`cnvlib.importers.`**`unpipe_name`**`(`*name*`)`

Fix the duplicated gene names Picard spits out.

Return a string containing the single gene name, sans duplications and pipe characters.

Picard CalculateHsMetrics combines the labels of overlapping intervals by joining all labels with '|', e.g. 'BRAF|BRAF' – no two distinct targeted genes actually overlap, though, so these dupes are redundant. Meaningless target names are dropped, e.g. 'CGH|FOO|-' resolves as 'FOO'. In case of ambiguity, the longest name is taken, e.g. "TERT|TERT Promoter" resolves as "TERT Promoter".

import_rna

`cnvlib.import_rna.`**`aggregate_gene_counts`**`(`*filenames*`)`

`cnvlib.import_rna.`**`aggregate_rsem`**`(`*fnames*`)`

Pull out the expected read counts from each RSEM file.

The format of RSEM's `*_rsem.genes.results` output files is tab-delimited with a header row. We extract the Ensembl gene ID, expected read counts, and transcript lengths from each file.

Returns

- **sample_counts** (*DataFrame*) – Row index is Ensembl gene ID, column index is filename.
- **tx_lengths** (*Series*) – Gene lengths.

`cnvlib.import_rna.`**`do_import_rna`**`(`*gene_count_fnames*, *in_format*, *gene_resource_fname*, *correlations_fname=None*, *normal_fnames=()*, *do_gc=True*, *do_txlen=True*, *max_log2=3*`)`

Convert a cohort of per-gene read counts to CNVkit .cnr format.

The expected data source is TCGA gene-level expression counts for individual samples, but other sources should be fine, too.

metrics

Robust metrics to evaluate performance of copy number estimates.

`cnvlib.metrics.`**`do_metrics`**`(`*cnarrs*, *segments=None*, *skip_low=False*`)`

Compute coverage deviations and other metrics for self-evaluation.

`cnvlib.metrics.`**`ests_of_scale`**`(`*deviations*`)`

Estimators of scale: standard deviation, MAD, biweight midvariance.

Calculates all of these values for an array of deviations and returns them as a tuple.

`cnvlib.metrics.`**`zip_repeater`**`(`*iterable*, *repeatable*`)`

Repeat a single segmentation to match the number of copy ratio inputs

reference

Supporting functions for the 'reference' command.

`cnvlib.reference.`**`bed2probes`**`(`*bed_fname*`)`

Create a neutral-coverage CopyNumArray from a file of regions.

`cnvlib.reference.`**`calculate_gc_lo`**`(`*subseq*`)`

Calculate the GC and lowercase (RepeatMasked) content of a string.

`cnvlib.reference.`**`combine_probes`**(*filenames*, *fa_fname*, *is_male_reference*, *sexes*, *skip_low*, *fix_gc*, *fix_edge*, *fix_rmask*)

Calculate the median coverage of each bin across multiple samples.

Parameters

- **`filenames`** (*`list`*) – List of string filenames, corresponding to targetcoverage.cnn and antitargetcoverage.cnn files, as generated by 'coverage' or 'import-picard'.
- **`fa_fname`** (*`str`*) – Reference genome sequence in FASTA format, used to extract GC and RepeatMasker content of each genomic bin.
- **`is_male_reference`** (*`bool`*) –
- **`skip_low`** (*`bool`*) –
- **`fix_gc`** (*`bool`*) –
- **`fix_edge`** (*`bool`*) –
- **`fix_rmask`** (*`bool`*) –

Returns One object summarizing the coverages of the input samples, including each bin's "average" coverage, "spread" of coverages, and GC content.

Return type *CopyNumArray*

`cnvlib.reference.`**`do_reference`**(*target_fnames*, *antitarget_fnames=None*, *fa_fname=None*, *male_reference=False*, *female_samples=None*, *do_gc=True*, *do_edge=True*, *do_rmask=True*)

Compile a coverage reference from the given files (normal samples).

`cnvlib.reference.`**`do_reference_flat`**(*targets*, *antitargets=None*, *fa_fname=None*, *male_reference=False*)

Compile a neutral-coverage reference from the given intervals.

Combines the intervals, shifts chrX values if requested, and calculates GC and RepeatMasker content from the genome FASTA sequence.

`cnvlib.reference.`**`fasta_extract_regions`**(*fa_fname*, *intervals*)

Extract an iterable of regions from an indexed FASTA file.

Input: FASTA file name; iterable of (seq_id, start, end) (1-based) Output: iterable of string sequences.

`cnvlib.reference.`**`get_fasta_stats`**(*cnarr*, *fa_fname*)

Calculate GC and RepeatMasker content of each bin in the FASTA genome.

`cnvlib.reference.`**`infer_sexes`**(*cnn_fnames*, *is_male_reference*)

Map sample IDs to inferred chromosomal sex, where possible.

For samples where the source file is empty or does not include either sex chromosome, that sample ID will not be in the returned dictionary.

`cnvlib.reference.`**`reference2regions`**(*refarr*)

Split reference into target and antitarget regions.

`cnvlib.reference.`**`warn_bad_bins`**(*cnarr*, *max_name_width=50*)

Warn about target bins where coverage is poor.

Prints a formatted table to stderr.

`reports`

Supports the sub-commands *breaks* and *genemetrics*.

Supporting functions for the text/tabular-reporting commands.

Namely: breaks, genemetrics.

`cnvlib.reports.`**`do_breaks`**(*probes*, *segments*, *min_probes=1*)
List the targeted genes in which a copy number breakpoint occurs.

`cnvlib.reports.`**`do_genemetrics`**(*cnarr*, *segments=None*, *threshold=0.2*, *min_probes=3*, *skip_low=False*, *male_reference=False*, *is_sample_female=None*)
Identify targeted genes with copy number gain or loss.

`cnvlib.reports.`**`gene_metrics_by_gene`**(*cnarr*, *threshold*, *skip_low=False*)
Identify genes where average bin copy ratio value exceeds *threshold*.

NB: Adjust the sample's sex-chromosome log2 values beforehand with shift_xx, otherwise all chrX/chrY genes may be reported gained/lost.

`cnvlib.reports.`**`gene_metrics_by_segment`**(*cnarr*, *segments*, *threshold*, *skip_low=False*)
Identify genes where segmented copy ratio exceeds *threshold*.

In the output table, show each segment's weight and probes as segment_weight and segment_probes, alongside the gene-level weight and probes.

NB: Adjust the sample's sex-chromosome log2 values beforehand with shift_xx, otherwise all chrX/chrY genes may be reported gained/lost.

`cnvlib.reports.`**`get_breakpoints`**(*intervals*, *segments*, *min_probes*)
Identify segment breaks within the targeted intervals.

`cnvlib.reports.`**`get_gene_intervals`**(*all_probes*, *ignore=('-', '.', 'CGH')*)
Tally genomic locations of each targeted gene.

Return a dict of chromosomes to a list of tuples: (gene name, starts, end), where gene name is a string, starts is a sorted list of probe start positions, and end is the last probe's end position as an integer. (The endpoints are redundant since probes are adjacent.)

`cnvlib.reports.`**`group_by_genes`**(*cnarr*, *skip_low*)
Group probe and coverage data by gene.

Return an iterable of genes, in chromosomal order, associated with their location and coverages:

> [(gene, chrom, start, end, [coverages]), . . .]

`scatter`

Command-line interface and corresponding API for CNVkit.

`cnvlib.scatter.`**`choose_segment_color`**(*segment*, *highlight_color*, *default_bright=True*)
Choose a display color based on a segment's CNA status.

Uses the fields added by the 'call' command. If these aren't present, use *highlight_color* for everything.

For sex chromosomes, some single-copy deletions or gains might not be highlighted, since sample sex isn't used to infer the neutral ploidies.

`cnvlib.scatter.`**`chromosome_scatter`**(*cnarr*, *segments*, *variants*, *show_range*, *show_gene*, *antitarget_marker*, *do_trend*, *by_bin*, *window_width*, *y_min*, *y_max*, *title*, *segment_color*)
Plot a specified region on one chromosome.

Possibilities:

```
     Options | Shown
------------ | --------
-c      | -g | Genes | Region
------- | -- | ----- | ------
-       | +  | given | auto: gene(s) + margin
chr     | -  | none  | whole chrom
chr     | +  | given | whole chrom
chr:s-e | -  | all   | given
chr:s-e | +  | given | given
```

`cnvlib.scatter.`**`cnv_on_chromosome`**(*axis*, *probes*, *segments*, *genes*, *antitarget_marker=None*, *do_trend=False*, *x_limits=None*, *y_min=None*, *y_max=None*, *segment_color='darkorange'*)

Draw a scatter plot of probe values with optional segments overlaid.

Parameters **`genes`** (`list`) – Of tuples: (start, end, gene name)

`cnvlib.scatter.`**`cnv_on_genome`**(*axis*, *probes*, *segments*, *do_trend=False*, *y_min=None*, *y_max=None*, *segment_color='darkorange'*)

Plot bin ratios and/or segments for all chromosomes on one plot.

`cnvlib.scatter.`**`do_scatter`**(*cnarr*, *segments=None*, *variants=None*, *show_range=None*, *show_gene=None*, *do_trend=False*, *by_bin=False*, *window_width=1000000.0*, *y_min=None*, *y_max=None*, *antitarget_marker=None*, *segment_color='darkorange'*, *title=None*)

Plot probe log2 coverages and segmentation calls together.

`cnvlib.scatter.`**`genome_scatter`**(*cnarr*, *segments=None*, *variants=None*, *do_trend=False*, *y_min=None*, *y_max=None*, *title=None*, *segment_color='darkorange'*)

Plot all chromosomes, concatenated on one plot.

`cnvlib.scatter.`**`get_segment_vafs`**(*variants*, *segments*)

Group SNP allele frequencies by segment.

Assume variants and segments were already subset to one chromosome.

Yields *tuple* – (segment, value)

`cnvlib.scatter.`**`highlight_genes`**(*axis*, *genes*, *y_posn*)

Show gene regions with background color and a text label.

`cnvlib.scatter.`**`select_range_genes`**(*cnarr*, *segments*, *variants*, *show_range*, *show_gene*, *window_width*)

Determine which datapoints to show based on the given options.

Behaviors:

```
start/end   show_gene
   +           +       given region + genes; err if any gene outside it
   -           +       window +/- around genes
   +           -       given region, highlighting any genes within it
   -           -       whole chromosome, no genes
```

If *show_range* is a chromosome name only, no start/end positions, then the whole chromosome will be shown.

If region start/end coordinates are given and *show_gene* is '' or ',' (or all commas, etc.), then instead of highlighting all genes in the selection, no genes will be highlighted.

`cnvlib.scatter.`**`set_xlim_from`**(*axis*, *probes=None*, *segments=None*, *variants=None*)

Configure axes for plotting a single chromosome's data.

Parameters

- **probes** (CopyNumArray) –
- **segments** (CopyNumArray) –
- **variants** (VariantArray) – All should already be subsetted to the region that will be plotted.

`cnvlib.scatter.`**`setup_chromosome`**(*axis*, *y_min=None*, *y_max=None*, *y_label=None*)

Configure axes for plotting a single chromosome's data.

`cnvlib.scatter.`**`snv_on_chromosome`**(*axis*, *variants*, *segments*, *genes*, *do_trend*, *by_bin*, *segment_color*)

`cnvlib.scatter.`**`snv_on_genome`**(*axis*, *variants*, *chrom_sizes*, *segments*, *do_trend*, *segment_color*)

Plot a scatter-plot of SNP chromosomal positions and shifts.

`segmentation`

Segmentation of copy number values.

`cnvlib.segmentation.`**`do_segmentation`**(*cnarr*, *method*, *threshold=None*, *variants=None*, *skip_low=False*, *skip_outliers=10*, *min_weight=0*, *save_dataframe=False*, *rlibpath=None*, *rscript_path='Rscript'*, *processes=1*)

Infer copy number segments from the given coverage table.

`cnvlib.segmentation.`**`drop_outliers`**(*cnarr*, *width*, *factor*)

Drop outlier bins with log2 ratios too far from the trend line.

Outliers are the log2 values *factor* times the 90th quantile of absolute deviations from the rolling average, within a window of given *width*. The 90th quantile is about 1.97 standard deviations if the log2 values are Gaussian, so this is similar to calling outliers *factor* * 1.97 standard deviations from the rolling mean. For a window size of 50, the breakdown point is 2.5 outliers within a window, which is plenty robust for our needs.

`cnvlib.segmentation.`**`transfer_fields`**(*segments*, *cnarr*, *ignore=('-', '.', 'CGH')*)

Map gene names, weights, depths from *cnarr* bins to *segarr* segments.

Segment gene name is the comma-separated list of bin gene names. Segment weight is the sum of bin weights, and depth is the (weighted) mean of bin depths.

Also: Post-process segmentation output.

1. Ensure every chromosome has at least one segment.
2. Ensure first and last segment ends match 1st/last bin ends (but keep log2 as-is).

`segmetrics`

Robust metrics to evaluate performance of copy number estimates.

`cnvlib.segmetrics.`**`calc_intervals`**(*bins_log2s*, *weights*, *func*)

Compute a stat that yields intervals (low & high values)

`cnvlib.segmetrics.`**`confidence_interval_bootstrap`**(*values*, *weights*, *alpha*, *bootstraps=100*, *smoothed=True*)

Confidence interval for segment mean log2 value, estimated by bootstrap.

`cnvlib.segmetrics.`**`do_segmetrics`**(*cnarr*, *segarr*, *location_stats=()*, *spread_stats=()*, *interval_stats=()*, *alpha=0.05*, *bootstraps=100*)

Compute segment-level metrics from bin-level log2 ratios.

cnvlib.segmetrics.**make_ci_func**(*alpha*, *bootstraps*)

cnvlib.segmetrics.**make_pi_func**(*alpha*)
Prediction interval, estimated by percentiles.

cnvlib.segmetrics.**segment_mean**(*cnarr*, *skip_low=False*)
Weighted average of bin log2 values.

target

Transform bait intervals into targets more suitable for CNVkit.

cnvlib.target.**do_target**(*bait_arr*, *annotate=None*, *do_short_names=False*, *do_split=False*, *avg_size=266.6666666666667*)
Transform bait intervals into targets more suitable for CNVkit.

cnvlib.target.**filter_names**(*names*, *exclude=('mRNA'*, *)*)
Remove less-meaningful accessions from the given set.

cnvlib.target.**shorten_labels**(*gene_labels*)
Reduce multi-accession interval labels to the minimum consistent.

So: BED or interval_list files have a label for every region. We want this to be a short, unique string, like the gene name. But if an interval list is instead a series of accessions, including additional accessions for sub-regions of the gene, we can extract a single accession that covers the maximum number of consecutive regions that share this accession.

e.g.:

```
...
mRNA|JX093079,ens|ENST00000342066,mRNA|JX093077,ref|SAMD11,mRNA|AF161376,
↪mRNA|JX093104
ens|ENST00000483767,mRNA|AF161376,ccds|CCDS3.1,ref|NOC2L
...
```

becomes:

```
...
mRNA|AF161376
mRNA|AF161376
...
```

cnvlib.target.**shortest_name**(*names*)
Return the shortest trimmed name from the given set.

4.1.4 Helper modules

cmdutil

Functions reused within command-line implementations.

cnvlib.cmdutil.**load_het_snps**(*vcf_fname*, *sample_id=None*, *normal_id=None*, *min_variant_depth=20*, *zygosity_freq=None*, *tumor_boost=False*)

cnvlib.cmdutil.**read_cna**(*infile*, *sample_id=None*, *meta=None*)
Read a CNVkit file (.cnn, .cnr, .cns) to create a CopyNumArray object.

cnvlib.cmdutil.**verify_sample_sex**(*cnarr*, *sex_arg*, *is_male_reference*) `cnvlib.cmdutil.`**`write_dataframe`**(*outfname*, *dframe*, *header=True*)

Write a pandas.DataFrame to a tabular file.

`cnvlib.cmdutil.`**`write_text`**(*outfname*, *text*, **more_texts*)

Write one or more strings (blocks of text) to a file.

`cnvlib.cmdutil.`**`write_tsv`**(*outfname*, *rows*, *colnames=None*)

Write rows, with optional column header, to tabular file.

core

CNV utilities.

`cnvlib.core.`**`assert_equal`**(*msg*, ***values*)

Evaluate and compare two or more values for equality.

Sugar for a common assertion pattern. Saves re-evaluating (and retyping) the same values for comparison and error reporting.

Example:

```
>>> assert_equal("Mismatch", expected=1, saw=len(['xx', 'yy']))
...
ValueError: Mismatch: expected = 1, saw = 2
```

`cnvlib.core.`**`call_quiet`**(**args*)

Safely run a command and get stdout; print stderr if there's an error.

Like subprocess.check_output, but silent in the normal case where the command logs unimportant stuff to stderr. If there is an error, then the full error message(s) is shown in the exception message.

`cnvlib.core.`**`check_unique`**(*items*, *title*)

Ensure all items in an iterable are identical; return that one item.

`cnvlib.core.`**`ensure_path`**(*fname*)

Create dirs and move an existing file to avoid overwriting, if necessary.

If a file already exists at the given path, it is renamed with an integer suffix to clear the way.

`cnvlib.core.`**`fbase`**(*fname*)

Strip directory and all extensions from a filename.

`cnvlib.core.`**`temp_write_text`**(**args*, ***kwds*)

Save text to a temporary file.

NB: This won't work on Windows b/c the file stays open.

descriptives

Robust estimators of central tendency and scale.

See: https://en.wikipedia.org/wiki/Robust_measures_of_scale https://astropy.readthedocs.io/en/latest/_modules/astropy/stats/funcs.html

`cnvlib.descriptives.`**`biweight_location`**(*a*, ***kwargs*)

Compute the biweight location for an array.

The biweight is a robust statistic for estimating the central location of a distribution.

`cnvlib.descriptives.`**`biweight_midvariance`**`(`*a*, ***kwargs*`)`

Compute the biweight midvariance for an array.

The biweight midvariance is a robust statistic for determining the midvariance (i.e. the standard deviation) of a distribution.

See:

- https://en.wikipedia.org/wiki/Robust_measures_of_scale#The_biweight_midvariance
- https://astropy.readthedocs.io/en/latest/_modules/astropy/stats/funcs.html

`cnvlib.descriptives.`**`gapper_scale`**`(`*a*, ***kwargs*`)`

Scale estimator based on gaps between order statistics.

See:

- Wainer & Thissen (1976)
- Beers, Flynn, and Gebhardt (1990)

`cnvlib.descriptives.`**`interquartile_range`**`(`*a*, ***kwargs*`)`

Compute the difference between the array's first and third quartiles.

`cnvlib.descriptives.`**`mean_squared_error`**`(`*a*, ***kwargs*`)`

Mean squared error (MSE).

By default, assume the input array *a* is the residuals/deviations/error, so MSE is calculated from zero. Another reference point for calculating the error can be specified with *initial*.

`cnvlib.descriptives.`**`median_absolute_deviation`**`(`*a*, ***kwargs*`)`

Compute the median absolute deviation (MAD) of array elements.

The MAD is defined as: `median(abs(a - median(a)))`.

See: https://en.wikipedia.org/wiki/Median_absolute_deviation

`cnvlib.descriptives.`**`modal_location`**`(`*a*, ***kwargs*`)`

Return the modal value of an array's values.

The "mode" is the location of peak density among the values, estimated using a Gaussian kernel density estimator.

Parameters **a** (*`np.array`*) – A 1-D array of floating-point values, e.g. bin log2 ratio values.

`cnvlib.descriptives.`**`on_array`**`(`*default=None*`)`

Ensure *a* is a numpy array with no missing/NaN values.

`cnvlib.descriptives.`**`on_weighted_array`**`(`*default=None*`)`

Ensure *a* and *w* are equal-length numpy arrays with no NaN values.

For weighted descriptives – *a* is the array of values, *w* is weights.

1. Drop any cells in *a* that are NaN from both *a* and *w*
2. Replace any remaining NaN cells in *w* with 0.

`cnvlib.descriptives.`**`q_n`**`(`*a*, ***kwargs*`)`

Rousseeuw & Croux's (1993) Q_n, an alternative to MAD.

`Qn := Cn first quartile of (|x_i - x_j|:  i < j)`

where Cn is a constant depending on n.

Finite-sample correction factors must be used to calibrate the scale of Qn for small-to-medium-sized samples.

n E[Qn] – —– 10 1.392 20 1.193 40 1.093 60 1.064 80 1.048 100 1.038 200 1.019

`cnvlib.descriptives.`**`weighted_mad`**(*a*, *w*, ***kwargs*)
Median absolute deviation (MAD) with weights.

`cnvlib.descriptives.`**`weighted_median`**(*a*, *w*, ***kwargs*)
Weighted median of a 1-D numeric array.

`cnvlib.descriptives.`**`weighted_std`**(*a*, *w*, ***kwargs*)
Standard deviation with weights.

parallel

Utilities for multi-core parallel processing.

class `cnvlib.parallel.`**`SerialFuture`**(*result*)
Bases: `future.types.newobject.newobject`

Mimic the concurrent.futures.Future interface.

`result`()

class `cnvlib.parallel.`**`SerialPool`**
Bases: `future.types.newobject.newobject`

Mimic the concurrent.futures.Executor interface, but run in serial.

`map`(*func*, *iterable*)
Just apply the function to *iterable*.

`shutdown`(*wait=True*)
Do nothing.

`submit`(*func*, **args*)
Just call the function on the arguments.

`cnvlib.parallel.`**`pick_pool`**(**args*, ***kwds*)

`cnvlib.parallel.`**`rm`**(*path*)
Safely remove a file.

`cnvlib.parallel.`**`to_chunks`**(*bed_fname*, *chunk_size=5000*)
Split a BED file into *chunk_size*-line parts for parallelization.

params

Defines several constants used in the pipeline.

Hard-coded parameters for CNVkit. These should not change between runs.

plots

Plotting utilities.

`cnvlib.plots.`**`chromosome_sizes`**(*probes*, *to_mb=False*)
Create an ordered mapping of chromosome names to sizes.

`cnvlib.plots.`**`cvg2rgb`**(*cvg*, *desaturate*)
Choose a shade of red or blue representing log2-coverage value.

`cnvlib.plots.`**`gene_coords_by_name`**(*probes*, *names*)
Find the chromosomal position of each named gene in probes.

Returns Of: {chromosome: [(start, end, gene name), . . .]}

Return type dict

`cnvlib.plots.`**`gene_coords_by_range`**(*probes*, *chrom*, *start*, *end*, *ignore=('-', '.', 'CGH')*)

Find the chromosomal position of all genes in a range.

Returns Of: {chromosome: [(start, end, gene), . . .]}

Return type dict

`cnvlib.plots.`**`get_repeat_slices`**(*values*)

Find the location and size of each repeat in *values*.

`cnvlib.plots.`**`plot_x_dividers`**(*axis*, *chrom_sizes*, *pad=None*)

Plot vertical dividers and x-axis labels given the chromosome sizes.

Draws vertical black lines between each chromosome, with padding. Labels each chromosome range with the chromosome name, centered in the region, under a tick. Sets the x-axis limits to the covered range.

Returns A table of the x-position offsets of each chromosome.

Return type OrderedDict

`cnvlib.plots.`**`translate_region_to_bins`**(*region*, *bins*)

Map genomic coordinates to bin indices.

Return a tuple of (chrom, start, end), just like unpack_range.

`cnvlib.plots.`**`translate_segments_to_bins`**(*segments*, *bins*)

`cnvlib.plots.`**`update_binwise_positions`**(*cnarr*, *segments=None*, *variants=None*)

Convert start/end positions from genomic to bin-wise coordinates.

Instead of chromosomal basepairs, the positions indicate enumerated bins.

Revise the start and end values for all GenomicArray instances at once, where the *cnarr* bins are mapped to corresponding *segments*, and *variants* are grouped into *cnarr* bins as well – if multiple *variants* rows fall within a single bin, equally-spaced fractional positions are used.

Returns copies of the 3 input objects with revised *start* and *end* arrays.

`cnvlib.plots.`**`update_binwise_positions_simple`**(*cnarr*)

rna

RNA expression levels quantified per gene.

Process per-gene expression levels, or the equivalent, by cohort.

`cnvlib.rna.`**`align_gene_info_to_samples`**(*gene_info*, *sample_counts*, *tx_lengths*, *normal_ids*)

Align columns and sort.

Also calculate weights and add to gene_info as 'weight', along with transcript lengths as 'tx_length'.

`cnvlib.rna.`**`attach_gene_info_to_cnr`**(*sample_counts*, *sample_data_log2*, *gene_info*, *read_len=100*)

Join gene info to each sample's log2 expression ratios.

Add the Ensembl gene info to the aggregated gene expected read counts, dropping genes that are not in the Ensembl table I.e., filter probes down to those genes that have names/IDs in the gene resource table.

Split out samples to individual .cnr files, keeping (most) gene info.

`cnvlib.rna.`**`before`**(*char*)

`cnvlib.rna.`**`correct_cnr`**(*cnr*, *do_gc*, *do_txlen*, *max_log2*)

Apply bias corrections & smoothing.

- Biases: 'gc', 'length'
- Smoothing: rolling triangle window using weights.

`cnvlib.rna.`**`dedupe_ens_hugo`**(*dframe*)

Emit the "best" index from a group of the same Ensembl ID.

The RSEM gene rows are the data of interest, and they're associated with Ensembl IDs to indicate the transcribed gene being measured in each row. The BioMart table of gene info can have duplicate rows for Ensembl ID, which would cause duplicate rows in RSEM import if joined naively. So, we need to select a single row for each group of "gene info" rows with the same Ensembl ID (column 'gene_id').

The keys we can use for this are:

- Entrez ID ('entrez_id')
- Ensembl gene name ('gene')
- Entrez gene name ('hugo_gene')

Entrez vs. Ensembl IDs and gene names are potentially many-to-many, e.g. CALM1/2/3. However, if we also require that the Ensembl and HUGO gene names match within a group, that (usually? always?) results in a unique row remaining.

(Example: CALM1/2/3 IDs are many-to-many, but of the 3 Entrez IDs associated with Ensembl's CALM1, only 1 is called CALM1 in the Entrez/corr. table.)

Failing that (no matches or multiple matches), prefer a lower Entrez ID, because well-characterized, protein-coding genes tend to have been discovered and accessioned first.

`cnvlib.rna.`**`dedupe_ens_no_hugo`**(*dframe*)

Deduplicate Ensembl ID using Entrez ID but not HUGO gene name.

`cnvlib.rna.`**`dedupe_tx`**(*dframe*)

Deduplicate table rows to select one transcript length per gene.

Choose the lowest-number Entrez ID and the transcript with the greatest support (primarily) and length (secondarily).

This is done at the end of Ensembl ID deduplication, after filtering on gene names and for single-row tables.

Returns an integer row index corresponding to the original table.

`cnvlib.rna.`**`filter_probes`**(*sample_counts*)

Filter probes to only include high-quality, transcribed genes.

The human genome has ~25,000 protein coding genes, yet the RSEM output includes ~58,000 rows. Some of these rows correspond to read counts over control probes (e.g. spike-in sequences). Some rows correspond to poorly mapped genes in contigs that have not been linked to the 24 chromosomes (e.g. HLA region). Others correspond to pseudo-genes and non-coding genes. For the purposes of copy number inference, these rows are best removed.

`cnvlib.rna.`**`load_cnv_expression_corr`**(*fname*)

`cnvlib.rna.`**`load_gene_info`**(*gene_resource*, *corr_fname*, *default_r=0.1*)

Read gene info from BioMart, and optionally TCGA, into a dataframe.

RSEM only outputs the Ensembl ID. We have used the BioMART tool in Ensembl to export a list of all Ensembl genes with a RefSeq mRNA (meaning it is high quality, curated, and bona fide gene) and resides on chromosomes 1-22, X, or Y. The tool also outputs the GC content of the gene, chromosomal coordinates of the gene, and HUGO gene symbol.

The gene resource input can be obtained from a resource bundle we provide (for reference genome hg19) or generated from BioMart.

`cnvlib.rna.`**`locate_entrez_dupes`**`(`*dframe*`)`

In case the same Entrez ID was assigned to multiple Ensembl IDs.

Use HUGO vs. HGNC name again, similar to *dedupe_hugo*, but instead of emiting the indices of the rows to keep, emit the indices of the extra rows – their correlation values will then be filled in with a default value (np.nan or 0.1).

It will then be as if those genes hadn't appeared in the TCGA tables at all, i.e. CNV-expression correlation is unknown, but all entries are still retained in the BioMart table (gene_info).

`cnvlib.rna.`**`normalize_read_depths`**`(`*sample_depths*, *normal_ids*`)`

Normalize read depths within each sample.

Some samples have higher sequencing depth and therefore read depths need to be normalized within each sample. TCGA recommends an upper quartile normalization.

After normalizing read depths within each sample, normalize (median-center) within each gene, across samples.

Finally, convert to log2 ratios.

`cnvlib.rna.`**`safe_log2`**`(`*values*, *min_log2*`)`

Transform values to log2 scale, safely handling zeros.

Parameters

- **`values`** (*`np.array`*) – Absolute-scale values to transform. Should be non-negative.
- **`min_log2`** (*`float`*) – Assign input zeros this log2-scaled value instead of -inf. Rather than hard-clipping, input values near 0 (especially below 2^min_log2) will be squeezed a bit above *min_log2* in the log2-scale output.

`cnvlib.rna.`**`tsl2int`**`(`*tsl*`)`

Convert an Ensembl Transcript Support Level (TSL) code to an integer.

The code has the format "tsl([1-5]|NA)".

See: https://www.ensembl.org/Help/Glossary?id=492

samutil

BAM utilities.

`cnvlib.samutil.`**`bam_total_reads`**`(`*bam_fname*`)`

Count the total number of mapped reads in a BAM file.

Uses the BAM index to do this quickly.

`cnvlib.samutil.`**`ensure_bam_index`**`(`*bam_fname*`)`

Ensure a BAM file is indexed, to enable fast traversal & lookup.

For MySample.bam, samtools will look for an index in these files, in order:

- MySample.bam.bai
- MySample.bai

`cnvlib.samutil.`**`ensure_bam_sorted`**`(`*bam_fname*, *by_name=False*, *span=50*`)`

Test if the reads in a BAM file are sorted as expected.

by_name=True: reads are expected to be sorted by query name. Consecutive read IDs are in alphabetical order, and read pairs appear together.

by_name=False: reads are sorted by position. Consecutive reads have increasing position.

`cnvlib.samutil.`**`get_read_length`**(*bam*, *span=1000*)

Get (median) read length from first few reads in a BAM file.

Illumina reads all have the same length; other sequencers might not.

Parameters

- **bam** (*str or pysam.Samfile*) – Filename or pysam-opened BAM file.
- **n** (*int*) – Number of reads used to calculate median read length.

`cnvlib.samutil.`**`idxstats`**(*bam_fname*, *drop_unmapped=False*)

Get chromosome names, lengths, and number of mapped/unmapped reads.

Use the BAM index (.bai) to get the number of reads and size of each chromosome. Contigs with no mapped reads are skipped.

`cnvlib.samutil.`**`is_newer_than`**(*target_fname*, *orig_fname*)

Compare file modification times.

segfilters

Filter copy number segments.

`cnvlib.segfilters.`**`ampdel`**(*segarr*)

Merge segments by amplified/deleted/neutral copy number status.

Follow the clinical reporting convention:

- 5+ copies (2.5-fold gain) is amplification
- 0 copies is homozygous/deep deletion
- CNAs of lesser degree are not reported

This is recommended only for selecting segments corresponding to actionable, usually focal, CNAs. Any real and potentially informative but lower-level CNAs will be dropped.

`cnvlib.segfilters.`**`bic`**(*segarr*)

Merge segments by Bayesian Information Criterion.

See: BIC-seq (Xi 2011), doi:10.1073/pnas.1110574108

`cnvlib.segfilters.`**`ci`**(*segarr*)

Merge segments by confidence interval (overlapping 0).

Segments with lower CI above 0 are kept as gains, upper CI below 0 as losses, and the rest with CI overlapping zero are collapsed as neutral.

`cnvlib.segfilters.`**`cn`**(*segarr*)

Merge segments by integer copy number.

`cnvlib.segfilters.`**`enumerate_changes`**(*levels*)

Assign a unique integer to each run of identical values.

Repeated but non-consecutive values will be assigned different integers.

`cnvlib.segfilters.`**`require_column`**(**colnames*)

Wrapper to coordinate the segment-filtering functions.

Verify that the given columns are in the CopyNumArray the wrapped function takes. Also log the number of rows in the array before and after filtration.

`cnvlib.segfilters.`**`sem`**(*segarr*)

Merge segments by Standard Error of the Mean (SEM).

Use each segment's SEM value to estimate a 95% confidence interval (via *zscore*). Segments with lower CI above 0 are kept as gains, upper CI below 0 as losses, and the rest with CI overlapping zero are collapsed as neutral.

`cnvlib.segfilters.`**`squash_by_groups`**(*cnarr*, *levels*, *by_arm=False*)

Reduce CopyNumArray rows to a single row within each given level.

`cnvlib.segfilters.`**`squash_region`**(*cnarr*)

Reduce a CopyNumArray to 1 row, keeping fields sensible.

Most fields added by the *segmetrics* command will be dropped.

smoothing

Signal smoothing functions.

`cnvlib.smoothing.`**`check_inputs`**(*x*, *width*, *as_series=True*)

Transform width into a half-window size.

width is either a fraction of the length of *x* or an integer size of the whole window. The output half-window size is truncated to the length of *x* if needed.

`cnvlib.smoothing.`**`convolve_unweighted`**(*window*, *signal*, *wing*)

Convolve a weighted window over array *signal*.

Input array is assumed padded by *_pad_array*; output has padding removed.

`cnvlib.smoothing.`**`convolve_weighted`**(*window*, *signal*, *weights*)

Convolve a weighted window over a weighted signal array.

Source: https://stackoverflow.com/a/46232913/10049

`cnvlib.smoothing.`**`guess_window_size`**(*x*, *weights=None*)

Choose a reasonable window size given the signal.

Inspired by Silverman's rule: bandwidth is proportional to signal's standard deviation and the length of the signal ^ 4/5.

`cnvlib.smoothing.`**`kaiser`**(*x*, *width=None*, *weights=None*, *do_fit_edges=False*)

Smooth the values in *x* with the Kaiser windowed filter.

See: https://en.wikipedia.org/wiki/Kaiser_window

Parameters

- **x** (*array-like*) – 1-dimensional numeric data set.
- **width** (*float*) – Fraction of x's total length to include in the rolling window (i.e. the proportional window width), or the integer size of the window.

`cnvlib.smoothing.`**`outlier_iqr`**(*a*, *c=3.0*)

Detect outliers as a multiple of the IQR from the median.

By convention, "outliers" are points more than 1.5 * IQR from the median, and "extremes" or extreme outliers are those more than 3.0 * IQR.

`cnvlib.smoothing.`**`outlier_mad_median`**(*a*)

MAD-Median rule for detecting outliers.

X_i is an outlier if:

```
 | X_i - M |
_____________   > K ~= 2.24

 MAD / 0.6745
```

where $K = sqrt(X^2_{0.975,1})$, the square root of the 0.975 quantile of a chi-squared distribution with 1 degree of freedom.

This is a very robust rule with the highest possible breakdown point of 0.5.

Returns A boolean array of the same size as *a*, where outlier indices are True.

Return type np.array

References

- Davies & Gather (1993) The Identification of Multiple Outliers.
- Rand R. Wilcox (2012) Introduction to robust estimation and hypothesis testing. Ch.3: Estimating measures of location and scale.

`cnvlib.smoothing.`**`rolling_median`**(*x*, *width*)

Rolling median with mirrored edges.

`cnvlib.smoothing.`**`rolling_outlier_iqr`**(*x*, *width*, *c=3.0*)

Detect outliers as a multiple of the IQR from the median.

By convention, "outliers" are points more than 1.5 * IQR from the median (~2 SD if values are normally distributed), and "extremes" or extreme outliers are those more than 3.0 * IQR (~4 SD).

`cnvlib.smoothing.`**`rolling_outlier_quantile`**(*x*, *width*, *q*, *m*)

Detect outliers by multiples of a quantile in a window.

Outliers are the array elements outside *m* times the *q*'th quantile of deviations from the smoothed trend line, as calculated from the trend line residuals. (For example, take the magnitude of the 95th quantile times 5, and mark any elements greater than that value as outliers.)

This is the smoothing method used in BIC-seq (doi:10.1073/pnas.1110574108) with the parameters width=200, q=.95, m=5 for WGS.

Returns A boolean array of the same size as *x*, where outlier indices are True.

Return type np.array

`cnvlib.smoothing.`**`rolling_outlier_std`**(*x*, *width*, *stdevs*)

Detect outliers by stdev within a rolling window.

Outliers are the array elements outside *stdevs* standard deviations from the smoothed trend line, as calculated from the trend line residuals.

Returns A boolean array of the same size as *x*, where outlier indices are True.

Return type np.array

`cnvlib.smoothing.`**`rolling_quantile`**(*x*, *width*, *quantile*)

Rolling quantile (0–1) with mirrored edges.

`cnvlib.smoothing.`**`rolling_std`**(*x*, *width*)

Rolling quantile (0–1) with mirrored edges.

`cnvlib.smoothing.`**`savgol`**(*x*, *width=None*, *weights=None*, *order=None*)

Savitzky-Golay smoothing.

4.2 scikit-genome package

4.2.1 Module `skgenome` contents

4.2.2 Tabular file I/O (tabio)

`tabio`

I/O for tabular formats of genomic data (regions or features).

`skgenome.tabio.`**`get_filename`**`(`*infile*`)`

`skgenome.tabio.`**`read`**`(`*infile*, *fmt='tab'*, *into=None*, *sample_id=None*, *meta=None*, ***kwargs*`)`

Read tabular data from a file or stream into a genome object.

Supported formats: see *READERS*

If a format supports multiple samples, return the sample specified by *sample_id*, or if unspecified, return the first sample and warn if there were other samples present in the file.

Parameters

- **`infile`** (*`handle or string`*) – Filename or opened file-like object to read.
- **`fmt`** (*`string`*) – File format.
- **`into`** (*`class`*) – GenomicArray class or subclass to instantiate, overriding the default for the target file format.
- **`sample_id`** (*`string`*) – Sample identifier.
- **`meta`** (*`dict`*) – Metadata, as arbitrary key-value pairs.
- **`**kwargs`** – Additional keyword arguments to the format-specific reader function.

Returns The data from the given file instantiated as *into*, if specified, or the default base class for the given file format (usually GenomicArray).

Return type *GenomicArray* or subclass

`skgenome.tabio.`**`read_auto`**`(`*infile*`)`

Auto-detect a file's format and use an appropriate parser to read it.

`skgenome.tabio.`**`safe_write`**`(`**args*, ***kwds*`)`

Write to a filename or file-like object with error handling.

If given a file name, open it. If the path includes directories that don't exist yet, create them. If given a file-like object, just pass it through.

`skgenome.tabio.`**`sniff_region_format`**`(`*infile*`)`

Guess the format of the given file by reading the first line.

Returns The detected format name, or None if the file is empty.

Return type str or None

`skgenome.tabio.`**`write`**`(`*garr*, *outfile=None*, *fmt='tab'*, *verbose=True*, ***kwargs*`)`

Write a genome object to a file or stream.

4.2.3 Base class: GenomicArray

The base class of the core objects used throughout CNVkit and scikit-genome is GenomicArray. It wraps a pandas DataFrame instance, which is accessible through the `.data` attribute and can be used for any manipulations that aren't already provided by methods in the wrapper class.

`gary`

Base class for an array of annotated genomic regions.

class `skgenome.gary.`**`GenomicArray`**(*data_table*, *meta_dict=None*)

Bases: `future.types.newobject.newobject`

An array of genomic intervals. Base class for genomic data structures.

Can represent most BED-like tabular formats with arbitrary additional columns.

`add`(*other*)

Combine this array's data with another GenomicArray (in-place).

Any optional columns must match between both arrays.

`add_columns`(***columns*)

Add the given columns to a copy of this GenomicArray.

Parameters **`**columns`** (*`array`*) – Keyword arguments where the key is the new column's name and the value is an array of the same length as *self* which will be the new column's values.

Returns A new instance of *self* with the given columns included in the underlying dataframe.

Return type *GenomicArray* or subclass

`as_columns`(***columns*)

Wrap the named columns in this instance's metadata.

`as_dataframe`(*dframe*)

Wrap the given pandas dataframe in this instance's metadata.

`as_rows`(*rows*)

Wrap the given rows in this instance's metadata.

`autosomes`(*also=()*)

Select chromosomes w/ integer names, ignoring any 'chr' prefixes.

`by_arm`(*min_gap_size=100000.0*, *min_arm_bins=50*)

Iterate over bins grouped by chromosome arm (inferred).

`by_chromosome`()

Iterate over bins grouped by chromosome name.

`by_ranges`(*other*, *mode='outer'*, *keep_empty=True*)

Group rows by another GenomicArray's bin coordinate ranges.

For example, this can be used to group SNVs by CNV segments.

Bins in this array that fall outside the other array's bins are skipped.

Parameters

- **`other`** (`GenomicArray`) – Another GA instance.
- **`mode`** (*`string`*) – Determines what to do with bins that overlap a boundary of the selection. Possible values are:

  - `inner`: Drop the bins on the selection boundary, don't emit them.
  - `outer`: Keep/emit those bins as they are.
  - `trim`: Emit those bins but alter their boundaries to match the selection; the bin start or end position is replaced with the selection boundary position.
- **keep_empty** (*bool*) – Whether to also yield *other* bins with no overlapping bins in *self*, or to skip them when iterating.

Yields *tuple* – (other bin, GenomicArray of overlapping rows in self)

chromosome

concat (*others*)

Concatenate several GenomicArrays, keeping this array's metadata.

This array's data table is not implicitly included in the result.

coords (*also=()*)

Iterate over plain coordinates of each bin: chromosome, start, end.

Parameters

- **also** (*str, or iterable of strings*) – Also include these columns from *self*, in addition to chromosome, start, and end.
- **yielding rows in BED format** (*Example,*) –
- **probes.coords(also=["gene", "strand"])** (*>>>*) –

copy ()

Create an independent copy of this object.

cut (*other*, *combine=None*)

Split this array's regions at the boundaries in *other*.

drop_extra_columns ()

Remove any optional columns from this GenomicArray.

Returns A new copy with only the minimal set of columns required by the class (e.g. chromosome, start, end for GenomicArray; may be more for subclasses).

Return type *GenomicArray* or subclass

end

filter (*func=None*, ***kwargs*)

Take a subset of rows where the given condition is true.

Parameters

- **func** (*callable*) – A boolean function which will be applied to each row to keep rows where the result is True.
- ****kwargs** (*string*) – Keyword arguments like `chromosome="chr7"` or `gene="Antitarget"`, which will keep rows where the keyed field equals the specified value.

Returns Subset of *self* where the specified condition is True.

Return type *GenomicArray*

flatten (*combine=None*, *split_columns=None*)

Split this array's regions where they overlap.

classmethod from_columns(*columns*, *meta_dict=None*)

Create a new instance from column arrays, given as a dict.

classmethod from_rows(*rows*, *columns=None*, *meta_dict=None*)

Create a new instance from a list of rows, as tuples or arrays.

in_range(*chrom=None*, *start=None*, *end=None*, *mode='outer'*)

Get the GenomicArray portion within the given genomic range.

Parameters

- **chrom** (`str or None`) – Chromosome name to select. Use None if *self* has only one chromosome.
- **start** (`int or None`) – Start coordinate of range to select, in 0-based coordinates. If None, start from 0.
- **end** (`int or None`) – End coordinate of range to select. If None, select to the end of the chromosome.
- **mode** (`str`) – As in *by_ranges*: `outer` includes bins straddling the range boundaries, `trim` additionally alters the straddling bins' endpoints to match the range boundaries, and `inner` excludes those bins.

Returns The subset of *self* enclosed by the specified range.

Return type *GenomicArray*

in_ranges(*chrom=None*, *starts=None*, *ends=None*, *mode='outer'*)

Get the GenomicArray portion within the specified ranges.

Similar to *in_ranges*, but concatenating the selections of all the regions specified by the *starts* and *ends* arrays.

Parameters

- **chrom** (`str or None`) – Chromosome name to select. Use None if *self* has only one chromosome.
- **starts** (`int array, or None`) – Start coordinates of ranges to select, in 0-based coordinates. If None, start from 0.
- **ends** (`int array, or None`) – End coordinates of ranges to select. If None, select to the end of the chromosome. If *starts* and *ends* are both specified, they must be arrays of equal length.
- **mode** (`str`) – As in *by_ranges*: `outer` includes bins straddling the range boundaries, `trim` additionally alters the straddling bins' endpoints to match the range boundaries, and `inner` excludes those bins.

Returns Concatenation of all the subsets of *self* enclosed by the specified ranges.

Return type *GenomicArray*

intersection(*other*, *mode='outer'*)

Select the bins in *self* that overlap the regions in *other*.

The extra fields of *self*, but not *other*, are retained in the output.

into_ranges(*other*, *column*, *default*, *summary_func=None*)

Re-bin values from *column* into the corresponding ranges in *other*.

Match overlapping/intersecting rows from *other* to each row in *self*. Then, within each range in *other*, extract the value(s) from *column* in *self*, using the function *summary_func* to produce a single value if multiple bins in *self* map to a single range in *other*.

For example, group SNVs (self) by CNV segments (other) and calculate the median (summary_func) of each SNV group's allele frequencies.

Parameters

- **`other`** (GenomicArray) – Ranges into which the overlapping values of *self* will be summarized.
- **`column`** (*`string`*) – Column name in *self* to extract values from.
- **`default`** – Value to assign to indices in *other* that do not overlap any bins in *self*. Type should be the same as or compatible with the output field specified by *column*, or the output of *summary_func*.
- **`summary_func`** (*`callable, dict of string-to-callable, or None`*) – Specify how to reduce 1 or more *other* rows into a single value for the corresponding row in *self*.
  - If callable, apply to the *column* field each group of rows in *other* column.
  - If a single-element dict of column name to callable, apply to that field in *other* instead of *column*.
  - If None, use an appropriate summarizing function for the datatype of the *column* column in *other* (e.g. median of numbers, concatenation of strings).
  - If some other value, assign that value to *self* wherever there is an overlap.

Returns The extracted and summarized values from *self* corresponding to other's genomic ranges, the same length as *other*.

Return type pd.Series

`iter_ranges_of`(*other*, *column*, *mode='outer'*, *keep_empty=True*)

Group rows by another GenomicArray's bin coordinate ranges.

For example, this can be used to group SNVs by CNV segments.

Bins in this array that fall outside the other array's bins are skipped.

Parameters

- **`other`** (GenomicArray) – Another GA instance.
- **`column`** (*`string`*) – Column name in *self* to extract values from.
- **`mode`** (*`string`*) – Determines what to do with bins that overlap a boundary of the selection. Possible values are:
  - `inner`: Drop the bins on the selection boundary, don't emit them.
  - `outer`: Keep/emit those bins as they are.
  - `trim`: Emit those bins but alter their boundaries to match the selection; the bin start or end position is replaced with the selection boundary position.
- **`keep_empty`** (*`bool`*) – Whether to also yield *other* bins with no overlapping bins in *self*, or to skip them when iterating.

Yields *tuple* – (other bin, GenomicArray of overlapping rows in self)

`keep_columns`(*colnames*)

Extract a subset of columns, reusing this instance's metadata.

`labels`()

merge (*bp=0*, *stranded=False*, *combine=None*)
Merge adjacent or overlapping regions into single rows.

Similar to 'bedtools merge'.

resize_ranges (*bp*, *chrom_sizes=None*)
Resize each genomic bin by a fixed number of bases at each end.

Bin 'start' values have a minimum of 0, and *chrom_sizes* can specify each chromosome's maximum 'end' value.

Similar to 'bedtools slop'.

Parameters

- **bp** (`int`) – Number of bases in each direction to expand or shrink each bin. Applies to 'start' and 'end' values symmetrically, and may be positive (expand) or negative (shrink).
- **chrom_sizes** (`dict of string-to-int`) – Chromosome name to length in base pairs. If given, all chromosomes in *self* must be included.

sample_id

shuffle ()
Randomize the order of bins in this array (in-place).

sort ()
Sort this array's bins in-place, with smart chromosome ordering.

sort_columns ()
Sort this array's columns in-place, per class definition.

squash (*combine=None*)
Combine some groups of rows, by some criteria, into single rows.

start

subdivide (*avg_size*, *min_size=0*, *verbose=False*)
Split this array's regions into roughly equal-sized sub-regions.

subtract (*other*)
Remove the overlapping regions in *other* from this array.

total_range_size ()
Total number of bases covered by all (merged) regions.

4.2.4 Genomic interval arithmetic

intersect

DataFrame-level intersection operations.

Calculate overlapping regions, similar to bedtools intersect.

The functions here operate on pandas DataFrame and Series instances, not GenomicArray types.

`skgenome.intersect.`**by_ranges** (*table*, *other*, *mode*, *keep_empty*)
Group rows by another GenomicArray's bin coordinate ranges.

`skgenome.intersect.`**by_shared_chroms** (*table*, *other*, *keep_empty=True*)

`skgenome.intersect.`**idx_ranges** (*table*, *chrom*, *starts*, *ends*, *mode*)
Iterate through sub-ranges.

`skgenome.intersect.`**`into_ranges`**(*source*, *dest*, *src_col*, *default*, *summary_func*)
Group a column in *source* by regions in *dest* and summarize.

`skgenome.intersect.`**`iter_ranges`**(*table*, *chrom*, *starts*, *ends*, *mode*)
Iterate through sub-ranges.

`skgenome.intersect.`**`iter_slices`**(*table*, *other*, *mode*, *keep_empty*)
Yields indices to extract ranges from *table*.

Returns an iterable of integer arrays that can apply to Series objects, i.e. columns of *table*. These indices are of the DataFrame/Series' Index, not array coordinates – so be sure to use DataFrame.loc, Series.loc, or Series getitem, as opposed to .iloc or indexing directly into Numpy arrays.

`skgenome.intersect.`**`venn`**(*table*, *other*, *mode*)

`merge`

DataFrame-level merging operations.

Merge overlapping regions into single rows, similar to bedtools merge.

The functions here operate on pandas DataFrame and Series instances, not GenomicArray types.

`skgenome.merge.`**`flatten`**(*table*, *combine=None*, *split_columns=None*)

`skgenome.merge.`**`merge`**(*table*, *bp=0*, *stranded=False*, *combine=None*)
Merge overlapping rows in a DataFrame.

`subdivide`

DataFrame-level subdivide operation.

Split each region into similar-sized sub-regions.

The functions here operate on pandas DataFrame and Series instances, not GenomicArray types.

`skgenome.subdivide.`**`subdivide`**(*table*, *avg_size*, *min_size=0*, *verbose=False*)

`subtract`

DataFrame-level subtraction operations.

Subtract one set of regions from another, returning the one-way difference.

The functions here operate on pandas DataFrame and Series instances, not GenomicArray types.

`skgenome.subtract.`**`subtract`**(*table*, *other*)

4.2.5 Helper modules

`chromsort`

Operations on chromosome/contig/sequence names.

`skgenome.chromsort.`**`detect_big_chroms`**(*sizes*)
Determine the number of "big" chromosomes from their lengths.

In the human genome, this returns 24, where the canonical chromosomes 1-22, X, and Y are considered "big", while mitochrondria and the alternative contigs are not. This allows us to exclude the non-canonical chromosomes from an analysis where they're not relevant.

Returns

- **n_big** (*int*) – Number of "big" chromosomes in the genome.
- **thresh** (*int*) – Length of the smallest "big" chromosomes.

`skgenome.chromsort.`**`sorter_chrom`**(*label*)

Create a sorting key from chromosome label.

Sort by integers first, then letters or strings. The prefix "chr" (case-insensitive), if present, is stripped automatically for sorting.

E.g. chr1 < chr2 < chr10 < chrX < chrY < chrM

combiners

Combiner functions for Python list-like input.

`skgenome.combiners.`**`first_of`**(*elems*)

Return the first element of the input.

`skgenome.combiners.`**`get_combiners`**(*table*, *stranded=False*, *combine=None*)

Get a *combine* lookup suitable for *table*.

Parameters

- **`table`** (`DataFrame`) –
- **`stranded`** (`bool`) –
- **`combine`** (`dict or None`) – Column names to their value-combining functions, replacing or in addition to the defaults.

Returns Column names to their value-combining functions.

Return type dict

`skgenome.combiners.`**`join_strings`**(*elems*, *sep=', '*)

Join a Series of strings by commas.

`skgenome.combiners.`**`last_of`**(*elems*)

Return the last element of the input.

`skgenome.combiners.`**`make_const`**(*val*)

`skgenome.combiners.`**`merge_strands`**(*elems*)

rangelabel

Handle text genomic ranges as named tuples.

A range specification should look like `chromosome:start-end`, e.g. `chr1:1234-5678`, with 1-indexed integer coordinates. We also allow `chr1:1234-` or `chr1:-5678`, where missing start becomes 0 and missing end becomes None.

`class` `skgenome.rangelabel.`**`NamedRegion`**(*chromosome*, *start*, *end*, *gene*)

Bases: `tuple`

`chromosome`
Alias for field number 0

`end`
Alias for field number 2

`gene`
Alias for field number 3

`start`
Alias for field number 1

`class` `skgenome.rangelabel.`**`Region`**(*chromosome*, *start*, *end*)
Bases: `tuple`

`chromosome`
Alias for field number 0

`end`
Alias for field number 2

`start`
Alias for field number 1

`skgenome.rangelabel.`**`from_label`**(*text*, *keep_gene=True*)
Parse a chromosomal range specification.

Parameters **`text`** (*`string`*) – Range specification, which should look like `chr1:1234-5678` or `chr1:1234-` or `chr1:-5678`, where missing start becomes 0 and missing end becomes None.

`skgenome.rangelabel.`**`to_label`**(*row*)
Convert a Region or (chrom, start, end) tuple to a region label.

`skgenome.rangelabel.`**`unpack_range`**(*a_range*)
Extract chromosome, start, end from a string or tuple.

Examples:

```
"chr1" -> ("chr1", None, None)
"chr1:100-123" -> ("chr1", 99, 123)
("chr1", 100, 123) -> ("chr1", 100, 123)
```

CHAPTER 5

Citation

If you use this software in a publication, please cite our paper describing CNVkit:

> Talevich, E., Shain, A.H., Botton, T., & Bastian, B.C. (2014). CNVkit: Genome-wide copy number detection and visualization from targeted sequencing. *PLOS Computational Biology* 12(4):e1004873

Also please cite the supporting paper for the segmentation method you use:

PSCBS and DNAcopy (`cbs`, the default):

- Olshen, A.B., Bengtsson, H., Neuvial, P., Spellman, P.T., Olshen, R.A., & Seshan, V.E. (2011). Parent-specific copy number in paired tumor-normal studies using circular binary segmentation. *Bioinformatics* 27(15):2038–46.
- Venkatraman, E.S., & Olshen, A.B. (2007). A faster circular binary segmentation algorithm for the analysis of array CGH data. *Bioinformatics* 23(6):657–63

HaarSeg (`haar`): Ben-Yaacov, E., & Eldar, Y.C. (2008). A fast and flexible method for the segmentation of aCGH data. *Bioinformatics* 24(16):i139-45.

CGH Fused Lasso (`flasso`): Tibshirani, R., & Wang, P. (2008). Spatial smoothing and hot spot detection for CGH data using the fused lasso. *Biostatistics* 9(1):18–29

5.1 Who else is using CNVkit?

Google Scholar lists some of the studies where CNVkit has been used by other researchers. We'd like to highlight:

- TRACERx Renal Consortium studies:
  - Turajlic, S. *et al.* (2018). Tracking Cancer Evolution Reveals Constrained Routes to Metastases: TRACERx Renal. *Cell*, 173(3), 581-594.e12
  - Turajlic, S. *et al.* (2018). Deterministic Evolutionary Trajectories Influence Primary Tumor Growth: TRACERx Renal. *Cell*, 173(3), 595-610.e11
- Seed, G. *et al.* (2017). Gene copy number estimation from targeted next generation sequencing of prostate cancer biopsies: Analytic validation and clinical qualification. *Clinical Cancer Research*, 23(20), 6070–6077

- McCreery, M.Q. *et al.* (2015). Evolution of metastasis revealed by mutational landscapes of chemically induced skin cancers. *Nature Medicine*, 21, 1514–1520
- Shain, A.H. *et al.* (2015). Exome sequencing of desmoplastic melanoma identifies recurrent NFKBIE promoter mutations and diverse activating mutations in the MAPK pathway. *Nature Genetics*, 47(10), 1194-1199
- Shain, A.H. *et al.* (2015). The Genetic Evolution of Melanoma from Precursor Lesions. *New England Journal of Medicine*, 373(20), 1926-1936

Specific support for CNVkit is included in bcbio-nextgen, PureCN, THetA2, and MetaSV. CNVkit is also available on the commercial platforms DNAnexus, Bina RAVE, and Diploid InHelix.

Finally, CNVkit can *export* files to several standard formats that can be used with many other software packages, including BioDiscovery Nexus Copy Number and Integrative Genomics Viewer (IGV).

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

c

cnvlib, 51
cnvlib.access, 57
cnvlib.antitarget, 57
cnvlib.autobin, 58
cnvlib.batch, 59
cnvlib.call, 59
cnvlib.cmdutil, 71
cnvlib.cnary, 51
cnvlib.commands, 55
cnvlib.core, 72
cnvlib.coverage, 61
cnvlib.descriptives, 72
cnvlib.diagram, 61
cnvlib.export, 62
cnvlib.fix, 64
cnvlib.heatmap, 65
cnvlib.import_rna, 66
cnvlib.importers, 65
cnvlib.metrics, 66
cnvlib.parallel, 74
cnvlib.params, 74
cnvlib.plots, 74
cnvlib.reference, 66
cnvlib.reports, 67
cnvlib.rna, 75
cnvlib.samutil, 77
cnvlib.scatter, 68
cnvlib.segfilters, 78
cnvlib.segmentation, 70
cnvlib.segmetrics, 70
cnvlib.smoothing, 79
cnvlib.target, 71
cnvlib.vary, 54

s

skgenome, 81
skgenome.chromsort, 87
skgenome.combiners, 88
skgenome.gary, 82
skgenome.intersect, 86
skgenome.merge, 87
skgenome.rangelabel, 88
skgenome.subdivide, 87
skgenome.subtract, 87
skgenome.tabio, 81

Index

A

absolute_clonal() (in module cnvlib.call), 59
absolute_dataframe() (in module cnvlib.call), 59
absolute_expect() (in module cnvlib.call), 60
absolute_pure() (in module cnvlib.call), 60
absolute_reference() (in module cnvlib.call), 60
absolute_threshold() (in module cnvlib.call), 60
add() (skgenome.gary.GenomicArray method), 82
add_columns() (skgenome.gary.GenomicArray method), 82
aggregate_gene_counts() (in module cnvlib.import_rna), 66
aggregate_rsem() (in module cnvlib.import_rna), 66
align_gene_info_to_samples() (in module cnvlib.rna), 75
ampdel() (in module cnvlib.segfilters), 78
apply_weights() (in module cnvlib.fix), 64
as_columns() (skgenome.gary.GenomicArray method), 82
as_dataframe() (skgenome.gary.GenomicArray method), 82
as_rows() (skgenome.gary.GenomicArray method), 82
assert_equal() (in module cnvlib.core), 72
assign_ci_start_end() (in module cnvlib.export), 62
attach_gene_info_to_cnr() (in module cnvlib.rna), 75
autosomes() (skgenome.gary.GenomicArray method), 82
average_depth() (in module cnvlib.autobin), 58

B

baf_by_ranges() (cnvlib.vary.VariantArray method), 54
bam_total_reads() (in module cnvlib.samutil), 77
batch_make_reference() (in module cnvlib.batch), 59
batch_run_sample() (in module cnvlib.batch), 59
batch_write_coverage() (in module cnvlib.batch), 59
bc_chromosome_draw_label() (in module cnvlib.diagram), 61
bc_organism_draw() (in module cnvlib.diagram), 61
bed2probes() (in module cnvlib.reference), 66
bedcov() (in module cnvlib.coverage), 61
before() (in module cnvlib.rna), 75
bic() (in module cnvlib.segfilters), 78
biweight_location() (in module cnvlib.descriptives), 72
biweight_midvariance() (in module cnvlib.descriptives), 72
build_chrom_diagram() (in module cnvlib.diagram), 61
by_arm() (skgenome.gary.GenomicArray method), 82
by_chromosome() (skgenome.gary.GenomicArray method), 82
by_gene() (cnvlib.cnary.CopyNumArray method), 51
by_ranges() (in module skgenome.intersect), 86
by_ranges() (skgenome.gary.GenomicArray method), 82
by_shared_chroms() (in module skgenome.intersect), 86

C

calc_intervals() (in module cnvlib.segmetrics), 70
calculate_gc_lo() (in module cnvlib.reference), 66
call_quiet() (in module cnvlib.core), 72
center_all() (cnvlib.cnary.CopyNumArray method), 52
center_by_window() (in module cnvlib.fix), 64
check_inputs() (in module cnvlib.smoothing), 79
check_unique() (in module cnvlib.core), 72
choose_segment_color() (in module cnvlib.scatter), 68
chromosome (skgenome.gary.GenomicArray attribute), 83
chromosome (skgenome.rangelabel.NamedRegion attribute), 88
chromosome (skgenome.rangelabel.Region attribute), 89
chromosome_scatter() (in module cnvlib.scatter), 68
chromosome_sizes() (in module cnvlib.plots), 74
ci() (in module cnvlib.segfilters), 78
cn() (in module cnvlib.segfilters), 78
cnv_on_chromosome() (in module cnvlib.scatter), 69
cnv_on_genome() (in module cnvlib.scatter), 69
cnvlib (module), 51
cnvlib.access (module), 57
cnvlib.antitarget (module), 57
cnvlib.autobin (module), 58
cnvlib.batch (module), 59
cnvlib.call (module), 59
cnvlib.cmdutil (module), 71

cnvlib.cnary (module), 51
cnvlib.commands (module), 55
cnvlib.core (module), 72
cnvlib.coverage (module), 61
cnvlib.descriptives (module), 72
cnvlib.diagram (module), 61
cnvlib.export (module), 62
cnvlib.fix (module), 64
cnvlib.heatmap (module), 65
cnvlib.import_rna (module), 66
cnvlib.importers (module), 65
cnvlib.metrics (module), 66
cnvlib.parallel (module), 74
cnvlib.params (module), 74
cnvlib.plots (module), 74
cnvlib.reference (module), 66
cnvlib.reports (module), 67
cnvlib.rna (module), 75
cnvlib.samutil (module), 77
cnvlib.scatter (module), 68
cnvlib.segfilters (module), 78
cnvlib.segmentation (module), 70
cnvlib.segmetrics (module), 70
cnvlib.smoothing (module), 79
cnvlib.target (module), 71
cnvlib.vary (module), 54
combine_probes() (in module cnvlib.reference), 66
compare_chrom_names() (in module cnvlib.antitarget), 57
compare_sex_chromosomes() (cnvlib.cnary.CopyNumArray method), 52
concat() (skgenome.gary.GenomicArray method), 83
confidence_interval_bootstrap() (in module cnvlib.segmetrics), 70
convolve_unweighted() (in module cnvlib.smoothing), 79
convolve_weighted() (in module cnvlib.smoothing), 79
coords() (skgenome.gary.GenomicArray method), 83
copy() (skgenome.gary.GenomicArray method), 83
CopyNumArray (class in cnvlib.cnary), 51
correct_cnr() (in module cnvlib.rna), 75
create_diagram() (in module cnvlib.diagram), 62
cut() (skgenome.gary.GenomicArray method), 83
cvg2rgb() (in module cnvlib.plots), 74

D

dedupe_ens_hugo() (in module cnvlib.rna), 76
dedupe_ens_no_hugo() (in module cnvlib.rna), 76
dedupe_tx() (in module cnvlib.rna), 76
detect_bedcov_columns() (in module cnvlib.coverage), 61
detect_big_chroms() (in module skgenome.chromsort), 87
do_access() (in module cnvlib.access), 57
do_access() (in module cnvlib.commands), 55
do_antitarget() (in module cnvlib.antitarget), 57
do_antitarget() (in module cnvlib.commands), 55
do_autobin() (in module cnvlib.autobin), 58
do_autobin() (in module cnvlib.commands), 55
do_breaks() (in module cnvlib.commands), 56
do_breaks() (in module cnvlib.reports), 68
do_call() (in module cnvlib.call), 60
do_call() (in module cnvlib.commands), 56
do_coverage() (in module cnvlib.commands), 56
do_coverage() (in module cnvlib.coverage), 61
do_fix() (in module cnvlib.commands), 56
do_fix() (in module cnvlib.fix), 64
do_genemetrics() (in module cnvlib.commands), 56
do_genemetrics() (in module cnvlib.reports), 68
do_heatmap() (in module cnvlib.commands), 56
do_heatmap() (in module cnvlib.heatmap), 65
do_import_picard() (in module cnvlib.importers), 65
do_import_rna() (in module cnvlib.commands), 57
do_import_rna() (in module cnvlib.import_rna), 66
do_import_theta() (in module cnvlib.commands), 57
do_import_theta() (in module cnvlib.importers), 65
do_metrics() (in module cnvlib.commands), 57
do_metrics() (in module cnvlib.metrics), 66
do_reference() (in module cnvlib.commands), 56
do_reference() (in module cnvlib.reference), 67
do_reference_flat() (in module cnvlib.commands), 56
do_reference_flat() (in module cnvlib.reference), 67
do_scatter() (in module cnvlib.commands), 56
do_scatter() (in module cnvlib.scatter), 69
do_segmentation() (in module cnvlib.commands), 56
do_segmentation() (in module cnvlib.segmentation), 70
do_segmetrics() (in module cnvlib.commands), 57
do_segmetrics() (in module cnvlib.segmetrics), 70
do_sex() (in module cnvlib.commands), 57
do_target() (in module cnvlib.commands), 55
do_target() (in module cnvlib.target), 71
drop_extra_columns() (skgenome.gary.GenomicArray method), 83
drop_low_coverage() (cnvlib.cnary.CopyNumArray method), 52
drop_noncanonical_contigs() (in module cnvlib.access), 57
drop_noncanonical_contigs() (in module cnvlib.antitarget), 58
drop_outliers() (in module cnvlib.segmentation), 70

E

edge_gains() (in module cnvlib.fix), 64
edge_losses() (in module cnvlib.fix), 65
end (skgenome.gary.GenomicArray attribute), 83
end (skgenome.rangelabel.NamedRegion attribute), 89
end (skgenome.rangelabel.Region attribute), 89
ensure_bam_index() (in module cnvlib.samutil), 77
ensure_bam_sorted() (in module cnvlib.samutil), 77

ensure_path() (in module cnvlib.core), 72
enumerate_changes() (in module cnvlib.segfilters), 78
ests_of_scale() (in module cnvlib.metrics), 66
expect_flat_log2() (cnvlib.cnary.CopyNumArray method), 52
export_bed() (in module cnvlib.export), 62
export_gistic_markers() (in module cnvlib.export), 62
export_nexus_basic() (in module cnvlib.export), 62
export_nexus_ogt() (in module cnvlib.export), 62
export_seg() (in module cnvlib.export), 63
export_theta() (in module cnvlib.export), 63
export_theta_snps() (in module cnvlib.export), 63
export_vcf() (in module cnvlib.export), 63

F

fasta_extract_regions() (in module cnvlib.reference), 67
fbase() (in module cnvlib.core), 72
filter() (skgenome.gary.GenomicArray method), 83
filter_names() (in module cnvlib.target), 71
filter_probes() (in module cnvlib.rna), 76
first_of() (in module skgenome.combiners), 88
flatten() (in module skgenome.merge), 87
flatten() (skgenome.gary.GenomicArray method), 83
fmt_cdt() (in module cnvlib.export), 63
fmt_gct() (in module cnvlib.export), 63
fmt_jtv() (in module cnvlib.export), 63
from_columns() (skgenome.gary.GenomicArray class method), 83
from_label() (in module skgenome.rangelabel), 89
from_rows() (skgenome.gary.GenomicArray class method), 84

G

gapper_scale() (in module cnvlib.descriptives), 73
gene (skgenome.rangelabel.NamedRegion attribute), 89
gene_coords_by_name() (in module cnvlib.plots), 74
gene_coords_by_range() (in module cnvlib.plots), 75
gene_metrics_by_gene() (in module cnvlib.reports), 68
gene_metrics_by_segment() (in module cnvlib.reports), 68
genome_scatter() (in module cnvlib.scatter), 69
GenomicArray (class in skgenome.gary), 82
get_antitargets() (in module cnvlib.antitarget), 58
get_breakpoints() (in module cnvlib.reports), 68
get_combiners() (in module skgenome.combiners), 88
get_edge_bias() (in module cnvlib.fix), 65
get_fasta_stats() (in module cnvlib.reference), 67
get_filename() (in module skgenome.tabio), 81
get_gene_intervals() (in module cnvlib.reports), 68
get_read_length() (in module cnvlib.samutil), 78
get_regions() (in module cnvlib.access), 57
get_repeat_slices() (in module cnvlib.plots), 75
get_segment_vafs() (in module cnvlib.scatter), 69
group_by_genes() (in module cnvlib.reports), 68
guess_chromosome_regions() (in module cnvlib.antitarget), 58
guess_window_size() (in module cnvlib.smoothing), 79
guess_xx() (cnvlib.cnary.CopyNumArray method), 53

H

heterozygous() (cnvlib.vary.VariantArray method), 54
highlight_genes() (in module cnvlib.scatter), 69
hybrid() (in module cnvlib.autobin), 59

I

idx_ranges() (in module skgenome.intersect), 86
idxstats() (in module cnvlib.samutil), 78
idxstats2ga() (in module cnvlib.autobin), 59
in_range() (skgenome.gary.GenomicArray method), 84
in_ranges() (skgenome.gary.GenomicArray method), 84
infer_sexes() (in module cnvlib.reference), 67
interquartile_range() (in module cnvlib.descriptives), 73
intersection() (skgenome.gary.GenomicArray method), 84
interval_coverages() (in module cnvlib.coverage), 61
interval_coverages_count() (in module cnvlib.coverage), 61
interval_coverages_pileup() (in module cnvlib.coverage), 61
into_ranges() (in module skgenome.intersect), 86
into_ranges() (skgenome.gary.GenomicArray method), 84
is_canonical_contig_name() (in module cnvlib.antitarget), 58
is_newer_than() (in module cnvlib.samutil), 78
iter_ranges() (in module skgenome.intersect), 87
iter_ranges_of() (skgenome.gary.GenomicArray method), 85
iter_slices() (in module skgenome.intersect), 87

J

join_regions() (in module cnvlib.access), 57
join_strings() (in module skgenome.combiners), 88

K

kaiser() (in module cnvlib.smoothing), 79
keep_columns() (skgenome.gary.GenomicArray method), 85

L

labels() (skgenome.gary.GenomicArray method), 85
last_of() (in module skgenome.combiners), 88
load_adjust_coverages() (in module cnvlib.fix), 65
load_cnv_expression_corr() (in module cnvlib.rna), 76
load_gene_info() (in module cnvlib.rna), 76
load_het_snps() (in module cnvlib.cmdutil), 71
locate_entrez_dupes() (in module cnvlib.rna), 77

log2 (cnvlib.cnary.CopyNumArray attribute), 53
log2_ratios() (in module cnvlib.call), 60
log_this() (in module cnvlib.access), 57

M

make_ci_func() (in module cnvlib.segmetrics), 70
make_const() (in module skgenome.combiners), 88
make_pi_func() (in module cnvlib.segmetrics), 71
map() (cnvlib.parallel.SerialPool method), 74
mask_bad_bins() (in module cnvlib.fix), 65
match_ref_to_sample() (in module cnvlib.fix), 65
mean_squared_error() (in module cnvlib.descriptives), 73
median_absolute_deviation() (in module cnvlib.descriptives), 73
merge() (in module skgenome.merge), 87
merge() (skgenome.gary.GenomicArray method), 85
merge_samples() (in module cnvlib.export), 63
merge_strands() (in module skgenome.combiners), 88
midsize_file() (in module cnvlib.autobin), 59
mirrored_baf() (cnvlib.vary.VariantArray method), 54
modal_location() (in module cnvlib.descriptives), 73

N

NamedRegion (class in skgenome.rangelabel), 88
normalize_read_depths() (in module cnvlib.rna), 77

O

on_array() (in module cnvlib.descriptives), 73
on_weighted_array() (in module cnvlib.descriptives), 73
outlier_iqr() (in module cnvlib.smoothing), 79
outlier_mad_median() (in module cnvlib.smoothing), 79

P

parse_theta_results() (in module cnvlib.importers), 65
pick_pool() (in module cnvlib.parallel), 74
plot_x_dividers() (in module cnvlib.plots), 75

Q

q_n() (in module cnvlib.descriptives), 73

R

read() (in module skgenome.tabio), 81
read_auto() (in module skgenome.tabio), 81
read_cna() (in module cnvlib.cmdutil), 71
ref_means_nbins() (in module cnvlib.export), 63
reference2regions() (in module cnvlib.reference), 67
Region (class in skgenome.rangelabel), 89
region_depth_count() (in module cnvlib.coverage), 61
region_size_by_chrom() (in module cnvlib.autobin), 59
require_column() (in module cnvlib.segfilters), 78
rescale_baf() (in module cnvlib.call), 60
residuals() (cnvlib.cnary.CopyNumArray method), 53
resize_ranges() (skgenome.gary.GenomicArray method), 86
result() (cnvlib.parallel.SerialFuture method), 74
rm() (in module cnvlib.parallel), 74
rolling_median() (in module cnvlib.smoothing), 80
rolling_outlier_iqr() (in module cnvlib.smoothing), 80
rolling_outlier_quantile() (in module cnvlib.smoothing), 80
rolling_outlier_std() (in module cnvlib.smoothing), 80
rolling_quantile() (in module cnvlib.smoothing), 80
rolling_std() (in module cnvlib.smoothing), 80

S

safe_log2() (in module cnvlib.rna), 77
safe_write() (in module skgenome.tabio), 81
sample_id (skgenome.gary.GenomicArray attribute), 86
sample_midsize_regions() (in module cnvlib.autobin), 59
sample_region_cov() (in module cnvlib.autobin), 59
savgol() (in module cnvlib.smoothing), 80
segment_mean() (in module cnvlib.segmetrics), 71
segments2vcf() (in module cnvlib.export), 64
select_range_genes() (in module cnvlib.scatter), 69
sem() (in module cnvlib.segfilters), 78
SerialFuture (class in cnvlib.parallel), 74
SerialPool (class in cnvlib.parallel), 74
set_colorbar() (in module cnvlib.heatmap), 65
set_xlim_from() (in module cnvlib.scatter), 69
setup_chromosome() (in module cnvlib.scatter), 70
shared_chroms() (in module cnvlib.autobin), 59
shift_xx() (cnvlib.cnary.CopyNumArray method), 53
shorten_labels() (in module cnvlib.target), 71
shortest_name() (in module cnvlib.target), 71
shuffle() (skgenome.gary.GenomicArray method), 86
shutdown() (cnvlib.parallel.SerialPool method), 74
skgenome (module), 81
skgenome.chromsort (module), 87
skgenome.combiners (module), 88
skgenome.gary (module), 82
skgenome.intersect (module), 86
skgenome.merge (module), 87
skgenome.rangelabel (module), 88
skgenome.subdivide (module), 87
skgenome.subtract (module), 87
skgenome.tabio (module), 81
smoothed() (cnvlib.cnary.CopyNumArray method), 53
sniff_region_format() (in module skgenome.tabio), 81
snv_on_chromosome() (in module cnvlib.scatter), 70
snv_on_genome() (in module cnvlib.scatter), 70
sort() (skgenome.gary.GenomicArray method), 86
sort_columns() (skgenome.gary.GenomicArray method), 86
sorter_chrom() (in module skgenome.chromsort), 88
squash() (skgenome.gary.GenomicArray method), 86
squash_by_groups() (in module cnvlib.segfilters), 79

squash_genes() (cnvlib.cnary.CopyNumArray method), 53
squash_region() (in module cnvlib.segfilters), 79
start (skgenome.gary.GenomicArray attribute), 86
start (skgenome.rangelabel.NamedRegion attribute), 89
start (skgenome.rangelabel.Region attribute), 89
subdivide() (in module skgenome.subdivide), 87
subdivide() (skgenome.gary.GenomicArray method), 86
submit() (cnvlib.parallel.SerialPool method), 74
subtract() (in module skgenome.subtract), 87
subtract() (skgenome.gary.GenomicArray method), 86

T

temp_write_text() (in module cnvlib.core), 72
theta_read_counts() (in module cnvlib.export), 64
to_chunks() (in module cnvlib.parallel), 74
to_label() (in module skgenome.rangelabel), 89
total_range_size() (skgenome.gary.GenomicArray method), 86
total_region_size() (in module cnvlib.autobin), 59
transfer_fields() (in module cnvlib.segmentation), 70
translate_region_to_bins() (in module cnvlib.plots), 75
translate_segments_to_bins() (in module cnvlib.plots), 75
tsl2int() (in module cnvlib.rna), 77
tumor_boost() (cnvlib.vary.VariantArray method), 55

U

unpack_range() (in module skgenome.rangelabel), 89
unpipe_name() (in module cnvlib.importers), 66
update_binwise_positions() (in module cnvlib.plots), 75
update_binwise_positions_simple() (in module cnvlib.plots), 75
update_chrom_length() (in module cnvlib.autobin), 59

V

VariantArray (class in cnvlib.vary), 54
venn() (in module skgenome.intersect), 87
verify_sample_sex() (in module cnvlib.cmdutil), 71

W

warn_bad_bins() (in module cnvlib.reference), 67
weighted_mad() (in module cnvlib.descriptives), 73
weighted_median() (in module cnvlib.descriptives), 74
weighted_std() (in module cnvlib.descriptives), 74
write() (in module skgenome.tabio), 81
write_dataframe() (in module cnvlib.cmdutil), 71
write_text() (in module cnvlib.cmdutil), 72
write_tsv() (in module cnvlib.cmdutil), 72

Z

zip_repeater() (in module cnvlib.metrics), 66
zygosity_from_freq() (cnvlib.vary.VariantArray method), 55