



pyfraglib Documentation

Release 0.6.1

Daniel Schütte

Jun 02, 2026

CONTENTS:

1	Installation	3
2	Quick Start Guide	4
3	Command Line Interface	8
4	API Reference	20
5	Simulation	108
6	Contributing	112
7	Changelog	114
8	Indices	115
	Python Module Index	116
	Index	117

LIST OF FIGURES

2.1	Overview of the pyfraglib workflow: fragment extraction from BAM files, feature analysis (lengths, motifs, scores), and result visualization.	5
5.1	cfDNA fragment simulation architecture showing the key components and their relationships.	109

`pyfraglib` is a Python library for analyzing high-throughput sequencing data of cell-free DNA (cfDNA), focusing on fragmentomics features. It provides tools to extract, analyze, and visualize fragment characteristics – either starting from BAM files or using a custom, memory-efficient file format.

INSTALLATION

`pyfraglib` depends on several external Python packages. The recommended installation method below is `setup.py` and `pyproject.toml` based so dependencies should be installed for you. For `pysam`, please refer to their documentation to learn about common issues (if you have any, that is!). Please also take a look at our `README.md` for the latest installation instructions. Installing `pyfraglib` from PyPI will be supported in the future.

1.1 Conda Installation (Recommended)

1. Create the conda environment from the provided YAML file:

```
conda env create -f pyfraglib.yml
conda activate pyfraglib
```

2. Install `pyfraglib` into the environment:

```
python -m pip install .
```

1.2 Development Installation

For development work, use the development installation script:

```
./tools/dev_install.sh
```

This script will run type checking with `mypy`, perform linting with `flake8`, install the package, and run all tests.

1.3 Verification

To verify your installation, run the test suite:

```
python -m unittest discover -s tests -v
```

Some integration tests might be reported as skipped if the necessary data files are not available. We do not provide data files for our more complex tests for privacy reasons.

You can have `pyfraglib` print version information as follows:

```
pyfrag.py version
```

QUICK START GUIDE

This guide will help you get started with `pyfraglib`. We'll cover the basic workflow from BAM file processing to generating fragmentomics analyses. Please let us know if you think that this page is missing important information!

2.1 Basic Workflow

The typical `pyfraglib` workflow involves:

1. **Extract fragments** from BAM files
2. **Analyze fragment characteristics** (lengths, motifs, etc.)
3. **Calculate fragmentomics scores** (WPS, motif diversity)
4. **Visualize results** with plots and statistics

2.2 Command Line Interface

`pyfraglib` provides a command-line interface through the `pyfrag.py` script that lets you perform these tasks in a few commands:

2.2.1 Extract Fragments

```
# Extract from single BAM file
pyfrag.py extract -o fragments/ --bam-file sample.bam

# Include mutation annotation (BAM and VCF file names must match!)
pyfrag.py extract -o fragments/ --bam-file sample.bam --with-vcf
```

2.2.2 Generate Overview Statistics

```
pyfrag.py stats -o stats/ --frag-file sample.frag
```

2.2.3 Analyze Fragment Lengths

```
pyfrag.py lengths -o lengths/ --frag-file sample.frag --config-file configs/gmm_3.json
```

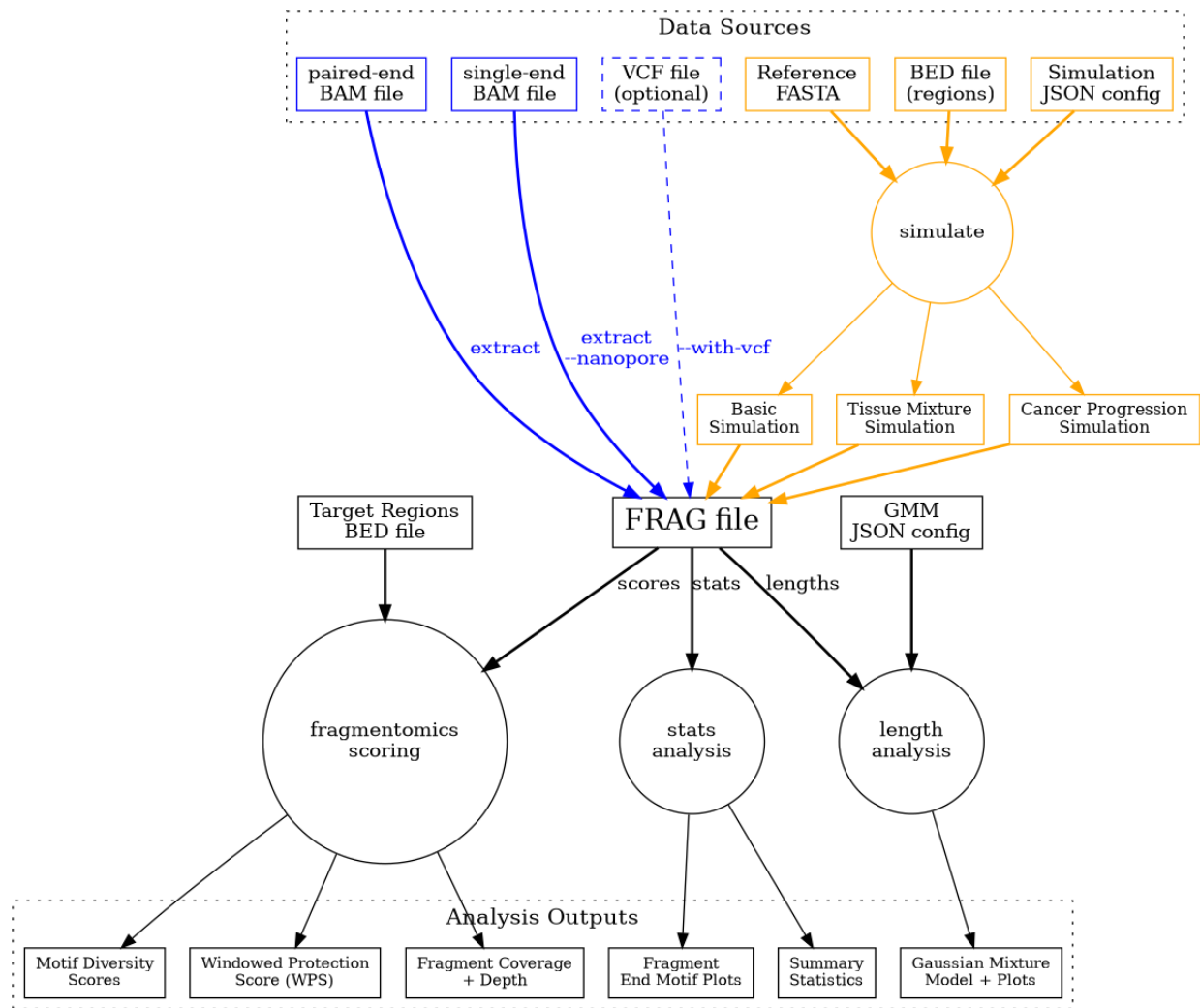


Fig. 2.1: Overview of the pyfraglib workflow: fragment extraction from BAM files, feature analysis (lengths, motifs, scores), and result visualization.

2.2.4 Calculate Fragmentomics Scores

```
pyfrag.py scores -o scores/ --frag-file sample.frag --bed-file regions.bed
```

2.3 Python API

You can also use pyfraglib programmatically and thus implement custom workflows easily:

2.3.1 Fragment Extraction

```
from pyfraglib import Fragment

# Extract fragments from BAM file
fragments = Fragment.from_bam("sample.bam")

# With VCF annotation
fragments = Fragment.from_bam("sample.bam", "variants.vcf")

# Save to fragment file
fragments.to_frag_file("sample", "output_directory")
```

2.3.2 Fragment Analysis

```
from pyfraglib import FragFile

# Load fragment file
frag_file = FragFile("sample.frag")
fragments = frag_file.get_fragment_list()

# Basic statistics
print(f"Total fragments: {fragments.length()}")
print(f"Bogus fragments: {fragments.count_bogus_fragments()}")
print(f"Mutated fragments: {fragments.count_mutated_fragments()}")

# End motif analysis
motifs_5p, motifs_3p, num_frags = fragments.count_endmotifs(4)
print(f"Most common 5' motif: {max(motifs_5p, key=motifs_5p.get)}")
```

2.3.3 Length Distribution Analysis

```
from pyfraglib import fit_gmm, plot_gmm

# Extract fragment lengths
lengths = [frag.length for frag in fragments if not frag.is_bogus]

# Fit Gaussian mixture model
gmm_result = fit_gmm(lengths, n_components=3)

# Plot results
plot_gmm(lengths, gmm_result, "length_distribution.png")
```


2.3.4 Fragmentomics Scores

```
from pyfraglib.scores import windowed_protection_score

# Calculate WPS for genomic regions
bed_regions = [("chr1", 1000, 2000), ("chr1", 5000, 6000)]
wps_scores = windowed_protection_score(fragments, bed_regions)
```

2.4 Simulation

A key features of pyfraglib is its simulation module. You can generate synthetic cfDNA data like so:

```
from pyfraglib import FragmentSimulator
from pyfraglib.simulator.fragment_simulator import SequenceContextGenerator

# Create sequence generator and simulator
seq_gen = SequenceContextGenerator("reference.fasta")
simulator = FragmentSimulator(seq_gen)

# Define regions to simulate
regions = [("chr1", 1000000, 2000000)]

# Generate fragments
fragments = simulator.simulate_fragments(regions, num_fragments=10000)

# Save results
fragments.to_frag_file("synthetic_sample", "output/")
```

2.5 Working with Configuration Files

pyfraglib uses JSON configuration files for various analyses. Examples can be found in the `configs/` directory of the repository as well as in the API documentation.

2.6 Next Steps

Please refer to the [API Reference](#) documentation for detailed API reference. CLI usage is explained here [Command Line Interface](#) with a reference of all available command-line options. Lastly, you can learn about [Simulation](#) for generating synthetic data.

COMMAND LINE INTERFACE

pyfraglib provides a command-line interface through the `pyfrag.py` script. This section covers all available commands and most of their options.

3.1 Extract Command

The `extract` command processes BAM files to extract cfDNA fragments and optionally annotate them with mutation information from VCF files.

3.1.1 Syntax

```
pyfrag.py extract -o <OUT_DIR> [OPTIONS]
```

3.1.2 Options

<code>--bam-file PATH</code>	Single BAM file to process
<code>--bam-dir PATH</code>	Directory containing BAM files to process
<code>--with-vcf</code>	Search for VCF file for mutation annotation (optional)
<code>--nanopore</code>	Process BAM as single-ended Nanopore data

If the `--with-vcf` flag is set, pyfraglib will search for one *VCF* file per *BAM*. The *VCF* must be named identically, except for the file extension which must be `.vcf` instead of `.bam`. *BAM* files must be indexed.

3.1.3 Examples

```
pyfrag.py --out-dir fragments/ extract --bam-file sample.bam --with-vcf
```

3.1.4 Output

The `extract` command creates `.frag` files in the specified output directory. These files contain compressed, serialized fragment data that can be used by other pyfraglib commands. For an input file named `sample.bam`, an output file called `sample.frag` will be created in `<OUT_DIR>`.

3.1.5 Quality Filters

The extraction process applies several quality filters that are currently hard-coded as source code constants:

- **MAPQ $\geq$ 20**: Minimum mapping quality score
- **Insert size $\leq$ 900bp**: Maximum insert size for paired-end reads
- **Valid chromosomes**: Only standard chromosomes (1-22, X, Y, M)

- **No ‘N’ bases:** Filters out fragments with ambiguous nucleotides
- **Proper pairing:** For paired-end data, both reads must be properly paired

3.1.6 General Tips

If you want to process a large set of samples, we recommend against using the `--bam-dir` flag. Even though the latter works on *BAM* files in parallel and writes fragment data to disk immediately, it still requires potentially large amounts of memory. We much rather recommend adopting the Nextflow pipeline in our `tools/` directory for working with many samples.

3.2 Stats Command

The stats command generates a basic set of statistics and visualizations from fragment files.

3.2.1 Syntax

```
pyfrag.py stats -o <OUT_DIR> [OPTIONS]
```

3.2.2 Options

<code>--frag-file PATH</code>	Single fragment file to analyze
<code>--frag-dir PATH</code>	Directory containing fragment files
<code>--kmer-length INT</code>	K-mer length for end motif analyses

3.2.3 Examples

```
pyfrag.py --out-dir statistics/ stats --frag-file sample.frag --kmer-length 3
```

3.2.4 Output

The stats command generates plots and data files:

- `*_frag_stats.json`: summary statistics regarding the extracted fragments like the total and mutated number of fragments
- `*_mut_fragments_per_chrom.png`: bar plot of mutated vs. wildtype fragments per chromosome
- `**_Xk_Pp_frag_end_motifs.png`: count plots of *X*-mer 5' / 3' end motifs (*X* is currently not parameterized but set to a constant 4)
- 2 CSV files with fragment length and end motif distributions, respectively, for use with e.g. external tools

3.3 Lengths Command

The lengths command plots fragment length distributions and fits a configurable Gaussian Mixture Model (GMM) to the distributions. Model parameters and visualizations are saved to disk.

3.3.1 Syntax

```
pyfrag.py lengths -o <OUT_DIR> [OPTIONS]
```

3.3.2 Options

<code>--frag-file PATH</code>	Single fragment file to analyze
<code>--frag-dir PATH</code>	Directory containing fragment files
<code>--config-file PATH</code>	GMM configuration file (JSON format, required)

3.3.3 Examples

```
pyfrag.py --out-dir lengths/ lengths --frag-file sample.frag --config-file configs/gmm_3.  
↪ json
```

3.3.4 Configuration File Format

GMM configuration files use *JSON* format with the following fields:

```
{  
  "number_of_gaussians": 3,  
  "subsample_percentage": 0.10,  
  "means_lower_bounds": [50, 250, 450],  
  "means_upper_bounds": [250, 450, 650],  
  "std_lower_bounds": [1, 1, 1],  
  "std_upper_bounds": [30, 60, 90],  
  "initial_means": [160, 320, 480],  
  "initial_pis": [0.70, 0.20, 0.10]  
}
```

The parameters determine the number of Gaussians fit, the lower and upper bounds on the means and standard deviations, and the initial parameters for means, standard deviations, and mixture fractions (π). The *subsample_percentage* parameter can be used if the *.frag* file contains a large number of fragments which sometimes makes the GMM fitting prohibitively slow. The model indicates such situations with an informative error message.

3.3.5 Output

The output saved to <OUT_DIR> contains a kernel density estimate plot of the fragment length distribution, stratified into mutated vs. wildtype fragments, and a plot of the GMM fit. Model parameters are saved to a *JSON* file along with goodness-of-fit parameters.

3.4 Scores Command

The scores command calculates fragmentomics scores like the Windowed Protection Score (WPS) and motif diversity metrics.

3.4.1 Syntax

```
pyfrag.py scores -o <OUT_DIR> [OPTIONS]
```

3.4.2 Options

<code>--frag-file PATH</code>	Single fragment file to analyze
<code>--frag-dir PATH</code>	Directory containing fragment files
<code>--bed-file PATH</code>	BED file defining genomic regions (required)

3.4.3 Examples

```
pyfrag.py --out-dir scores/ scores --frag-file sample.frag --bed-file regions.bed
```

3.4.4 BED File Format

The BED file should contain genomic regions of interest for which the windowed protection score is then calculated:

chr1	1000000	1001000	region1
chr1	2000000	2001000	region2
chr2	500000	501000	region3

Required columns: - Column 1: Chromosome name - Column 2: Start position (0-based) - Column 3: End position (0-based) - Column 4: Region name (optional)

3.4.5 Output

As output, this subcommand saves the following *CSV* files to disk:

- `wps_<sample>.csv` — per-position Windowed Protection Score for all regions in the input *BED* file. The file also has a `wps_smooth` column containing the coverage-normalised, detrended, Savitzky-Golay-smoothed signal that is suitable for between-sample comparison.
- `wps_metrics_<sample>.csv` — per-region WPS summary metrics (number of peaks, median peak spacing, median peak prominence and width, and the 170 bp spectral power `power_170`). These are also used by the `differential_wps.py` utility script.
- `global_sample_metrics.csv` — one row per sample, providing end-motif diversity (Shannon entropy and Simpson index for 5' and 3' motifs) together with cohort-level WPS summaries (median `power_170`, median peaks per region, median peak spacing, median peak prominence).

A genome-wide WPS line plot is additionally written per sample as a visual summary, together with a diagnostic `<sample>_wps_spectrum.png` that shows the per-region magnitude spectra and the mean curve across regions with the nucleosome-repeat band highlighted.

3.4.6 Calculated Metrics

Windowed Protection Score (WPS)

For every genomic position i , the WPS measures the depletion of fragment ends within a window surrounding that position:

$$\text{WPS}(i) = N_{\text{span}}(i) - N_{\text{end}}(i)$$

Large values indicate protection (fewer fragment ends), whereas smaller values indicate chromatin accessibility (more fragment ends). Because this metric is not independent of local fragment depth, the output *CSV* file contains positional fragment depth, the number of ending, and the number of spanning fragments as additional fields. This way, users can calculate fractions instead of differences if that makes more sense for their datasets.

Smoothed WPS and per-region metrics

As pointed out above, raw WPS amplitudes are dominated by sequencing depth and by long-wavelength coverage drift, which make between-sample comparisons difficult. To robustly isolate nucleosome phasing signals, the `scores` subcommand runs the smoothing pipeline implemented in `pyfraglib.scores.smooth_wps()`:

1. coverage-normalise via `wps / (median_depth + 1)` per region,
2. subtract a 1001 bp rolling median,

3. apply a Savitzky-Golay filter (window 21 bp, polynomial order 3).

The resulting `wps_smooth` column is appended to `wps_<sample>.csv`. [pyfraglib.scores.wps_region_metrics\(\)](#) then derives a set of interpretable per-region metrics from the same pipeline and writes them to `wps_metrics_<sample>.csv`:

- `n_peaks` — number of peaks detected with `scipy.signal.find_peaks` using a prominence threshold of 0.02 x per-region IQR and a minimum peak-to-peak distance of 120 bp.
- `mean_peak_dist / median_peak_dist` — mean and median of consecutive peak-to-peak spacings within the region.
- `median_prom / median_width` — median peak prominence and median peak width.
- `power_170` — relative spectral power in the nucleosome-repeat band (see below).
- `n_positions` — number of input positions for the region.

Parameter choices

The three smoothing parameters are tied directly to the average nucleosome-repeat period $\lambda = 170$ bp, which sets the target frequency $f_\lambda = 1/\lambda \approx 0.0059$ cycles/bp. Each parameter is chosen so that f_λ sits comfortably inside the passband of the combined filter.

1001 bp rolling-median detrend window. Rolling-median subtraction acts as a high-pass filter with effective cutoff near $1/W$ cycles/bp. With $W = 1001$, the cutoff lies roughly a factor of six below f_λ , so the nucleosome oscillation passes essentially unattenuated while long-wavelength coverage drift (kilobase-scale protected / unprotected blocks that otherwise dominate raw WPS amplitude) is removed. The window is adaptively shrunk for short regions via $\min(W, \max(101, (L/3) \mid 1))$.

21 bp Savitzky-Golay window. A Savitzky-Golay filter with window N has its 3 dB point near $1/(N/2)$ cycles/bp; at $N = 21$ that corresponds to a period floor of roughly 10 bp – more than an order of magnitude below λ , so the 170 bp oscillation is preserved while per-base Poisson noise in the spanning-minus-ending count is averaged out.

Spectral analysis of WPS

Let $x[n]$ denote the detrended, coverage-normalised WPS at positions $n = 0, 1, \dots, N - 1$ where N is the size of the region. We first remove the zero-frequency component,

$$\tilde{x}[n] = x[n] - \frac{1}{N} \sum_{k=0}^{N-1} x[k],$$

then compute the one-sided discrete Fourier transform on a zero-padded signal of length $N_{\text{fft}} = 2^{\lceil \log_2 N \rceil}$ using the `numpy.fft.rfft` implementation:

$$X[k] = \sum_{n=0}^{N_{\text{fft}}-1} \tilde{x}[n] e^{-j2\pi kn/N_{\text{fft}}}, \quad k = 0, 1, \dots, N_{\text{fft}}/2.$$

The corresponding frequency grid is $f_k = k/N_{\text{fft}}$, expressed in cycles per base pair because the per-position sampling interval is 1 bp. The magnitude spectrum $S[k] = |X[k]|$ is then integrated over a narrow band around the nucleosome repeat length $\lambda = 170$ bp. With

$$\mathcal{B} = \left\{ k : \frac{1}{\lambda + 10} < f_k < \frac{1}{\lambda - 10} \right\},$$

the 170 bp spectral-power feature is the in-band mean magnitude normalised by the overall mean magnitude,

$$\text{power_170} = \frac{\frac{1}{|\mathcal{B}|} \sum_{k \in \mathcal{B}} S[k]}{\frac{1}{N_{\text{m}}/2 + 1} \sum_{k=0}^{N_{\text{m}}/2} S[k]}.$$

This ratio is dimensionless and independent of depth-driven amplitude. The mean-over-bins normalisation is chosen so that a flat spectrum (i.e. no preferred periodicity) yields numerator and denominator of equal magnitude, giving $\text{power_170} = 1$. Values above 1 indicate that the nucleosome-repeat band carries more energy per bin than the spectrum-wide average (i.e. nucleosomes are regularly phased).

Motif Diversity

Given an end motif distribution $X = (x_1, \dots, x_n)$ where x_i is the observed count for end motif i , we calculate Shannon entropy $H(X)$ as

$$H(X) = - \sum_{i=1}^n p(x_i) \log_2 p(x_i)$$

and Simpson index $D(X)$ as

$$D(X) = \sum_{i=1}^n x_i^2$$

3.5 Simulate Command

The `simulate` command generates synthetic cfDNA fragment data based on biological parameters and real genomic sequences.

3.5.1 Syntax

```
pyfrag.py simulate -o <OUT_DIR> [OPTIONS]
```

3.5.2 Options

```
--config PATH          Configuration file (JSON format) (required)
```

3.5.3 Examples

```
pyfrag.py --out-dir synthetic/ simulate --config configs/simulation_example.json
```

3.5.4 Configuration File Format

Please take a look at the example configurations in `configs/`. Most options should be self-explanatory.

3.5.5 Simulation Modes

Basic Mode

- Single tissue type simulation

- Uniform fragment characteristics
- Customizable length distribution
- Simple nuclease profile

Tissue Mixture Mode

- Multi-tissue cfDNA simulation
- Tissue-specific fragment patterns
- Configurable tissue fractions

Cancer Progression Mode

- Simulates tumor progression
- Multiple timepoints
- Increasing tumor fractions
- Mutation accumulation

3.5.6 Available Tissue Profiles

- **hematopoietic**: Blood cell-derived cfDNA
- **liver**: Hepatic cfDNA patterns
- **placenta**: Placental cfDNA (fetal)
- **tumor**: Cancer-derived cfDNA

3.5.7 Output

Fragment Files

- `{output_name}.frag` - Main synthetic fragment file
- `{output_name}_timepoint_{time}.frag` - Timepoint-specific files (cancer progression)
- `{output_name}_tissue_{tissue}.frag` - Tissue-specific files (tissue mixture)

3.6 Utility Scripts

pyfraglib includes several utility scripts for specialized tasks. While they were created for solving specific problems during our research, they might still be useful to others, so we briefly describe them here.

3.6.1 `extract_mutated_reads.py`

Extracts reads that support specific variants from BAM files. This script can be used to split a *BAM* file into separate files containing only the mutated reads vs. only the unmutated reads based on a *VCF* file. Mutational status assignment in pyfraglib was extensively tested using *BAM* files prepared in this way.

Syntax

```
extract_mutated_reads.py [OPTIONS]
```


Options

```
--bam PATH      Input BAM file
--vcf PATH      Input VCF file with variants
--output PATH   Output BAM file for mutated reads
--keep         Keep unmutated reads in separate BAM file
```

3.6.2 download_tss_annos.py

Downloads transcription start site (TSS) annotations from Ensembl, generating a *BED* file that can be used with the `pyfraglib scores` subcommand. The genes for which to download annotations must be provided via a *TXT* file with one gene per line.

Syntax

```
download_tss_annos.py [OPTIONS]
```

Options

```
--ref VERSION   Genome version (hg19, hg38)
--outfile PATH  Output directory for annotations
--gene-list PATH File with gene symbols to download
--verbose      Enable verbose logging (e.g. for trouble-shooting)
```

3.6.3 txt_to_vcf.py

Converts custom variant format to standard VCF format. This script is probably not useful for anyone but researchers from our group.

Syntax

```
txt_to_vcf.py [OPTIONS]
```

Options

```
--infile PATH   Input file in custom format
--outfile PATH  Output VCF file
--ref-genome PATH hg19 or hg38
```

3.6.4 nmf_fragment_lengths.py

Performs non-negative matrix factorization (NMF) on fragment length distributions from cfDNA samples. This analysis identifies underlying signatures in fragment length patterns that can reveal tissue-specific or pathological fragmentation patterns.

NMF decomposes the fragment length matrix into basis components (signatures) and mixing coefficients (sample compositions). The script runs multiple random initializations to ensure stable results and normalizes mixing coefficients to represent true proportions.

Input Requirements

- Text file listing fragment length CSV file paths (one per line)
- CSV files from `pyfrag stats` with columns: `fragment_length`, `count`

Output Files

- `nmf_mixing_coefficients.csv` - Sample composition weights (sum to 1.0 per sample)
- `nmf_signatures.csv` - Component signatures across fragment lengths
- `nmf_signatures.png` - Component signature line plots
- `nmf_sample_composition.png` - Sample composition heatmap
- `nmf_component_weights.png` - Component weights bar chart

Syntax

```
nmf_fragment_lengths.py [OPTIONS]
```

Options

<code>--file-list PATH</code>	Text file with CSV file paths (one per line)
<code>--n-components INTEGER</code>	Number of NMF components to extract (default: 3). Ignored when <code>--signatures</code> is given.
<code>--out-dir PATH</code>	Output directory (default: <code>nmf_output</code>)
<code>--signatures PATH</code>	Optional path to an existing <code>nmf_signatures.csv</code> . When set, the script skips refitting and instead projects the input samples onto the loaded signatures basis (fixed-H NMF transform). Fragment-length columns are aligned to the basis layout before projection.
<code>--verbose</code>	Enable verbose logging

Signatures mode lets you evaluate learned signatures on unseen samples.

3.6.5 differential_end_motifs.py

Performs differential analysis of fragment end motif abundances between two groups of cfDNA samples. This statistical analysis identifies motifs that are significantly over-represented in one group versus another, which can reveal tissue-specific cleavage preferences or pathological changes in nuclease activity.

The analysis uses Wilcoxon rank-sum tests for non-parametric comparison, followed by Benjamini-Hochberg FDR correction to control for multiple testing across all motifs.

Input Requirements

- JSON configuration file with `group_a` and `group_b` fields
- Each group contains a list of CSV file paths
- CSV files from `pyfrag stats` with columns: `motif_5p`, `count_5p`, `motif_3p`, `count_3p`

Output Files

- `differential_results.csv` - Complete statistical results for all motifs
- `differential_volcano_plot.png` - Effect size vs significance visualization
- `differential_effect_sizes.png` - Distribution of effect sizes
- `differential_top_motifs.png` - Top significant motifs bar chart

Syntax

```
differential_end_motifs.py [OPTIONS]
```

Options

```
--config PATH      JSON configuration file with group definitions
--out-dir PATH      Output directory (default: differential_output)
--verbose           Enable verbose logging
```

Configuration Format

```
{
  "group_a": [
    "healthy1_k4_end_motifs.csv",
    "healthy2_k4_end_motifs.csv",
    "healthy3_k4_end_motifs.csv"
  ],
  "group_b": [
    "cancer1_k4_end_motifs.csv",
    "cancer2_k4_end_motifs.csv",
    "cancer3_k4_end_motifs.csv"
  ]
}
```

3.6.6 differential_wps.py

Performs differential analysis of per-region Windowed Protection Score (WPS) metrics between two groups of cfDNA samples. The script operates on the per-sample `wps_metrics_<sample>.csv` files produced by `pyfrag.py scores` (i.e. the output of `pyfraglib.scores.wps_region_metrics()`) and identifies genomic regions where a chosen metric (by default `power_170`) differs significantly between groups.

The analysis uses Wilcoxon rank-sum tests for non-parametric comparison, followed by Benjamini-Hochberg FDR correction to control for multiple testing across all regions.

Input Requirements

- JSON configuration file with `group_a` and `group_b` fields
- Each group contains a list of `wps_metrics_<sample>.csv` paths
- CSV files must carry at least the columns `region` and the requested metric (`power_170` by default)

Output Files

- `differential_wps_results_<metric>.csv` — per-region test results (group means, medians, Wilcoxon p-value, FDR-adjusted p-value, rank-biserial effect size, log2 fold change)
- `differential_wps_volcano_<metric>.png` — effect size vs significance visualisation with the top significant regions annotated
- `differential_wps_effect_sizes_<metric>.png` — distribution of rank-biserial effect sizes, split by significance

Syntax

```
differential_wps.py [OPTIONS]
```

Options

<code>--config PATH</code>	JSON configuration file with group definitions
<code>--out-dir PATH</code>	Output directory (default: <code>differential_wps_output</code>)
<code>--metric NAME</code>	WPS metric to <code>test</code> (default: <code>power_170</code>). One of: <code>power_170</code> , <code>n_peaks</code> , <code>median_peak_dist</code> , <code>median_prom</code> , <code>median_width</code>
<code>--min-samples-per-group INT</code>	Minimum non-NaN values per group to <code>test</code> a region (default: <code>3</code>)
<code>--verbose</code>	Enable verbose logging

Configuration Format

```
{
  "group_a": [
    "healthy1/wps_metrics_healthy1.csv",
    "healthy2/wps_metrics_healthy2.csv",
    "healthy3/wps_metrics_healthy3.csv"
  ],
  "group_b": [
    "cancer1/wps_metrics_cancer1.csv",
    "cancer2/wps_metrics_cancer2.csv",
    "cancer3/wps_metrics_cancer3.csv"
  ]
}
```

3.7 Overview

The main CLI script is `pyfrag.py`, which provides several subcommands:

```
pyfrag.py <subcommand> [options]
```

Available subcommands:

- `extract` - Extract fragments from BAM files
- `stats` - Generate fragment statistics
- `lengths` - Analyze fragment length distributions
- `scores` - Calculate fragmentomics scores

- `simulate` - Generate synthetic cfDNA data
- `version` - Show version information

3.8 Global Options

All subcommands support these global options:

```
-h, --help      Show help message and exit  
-v, --verbose   Enable verbose logging  
-o, --out-dir   Set the output directory for the subcommand
```

Importantly, for help with any subcommand:

```
pyfrag.py <subcommand> --help
```

3.9 Additional Scripts

pyfraglib also includes several utility scripts:

- `extract_mutated_reads.py` - Extract reads supporting variants
- `download_tss_annos.py` - Download TSS annotations
- `txt_to_vcf.py` - Convert custom format to VCF

API REFERENCE

This section contains the Python API documentation for `pyfraglib`.

4.1 Core Classes

The following core classes represent cfDNA fragments and collections of them in `pyfraglib`:

<code>pyfraglib.Fragment</code>	Serializable container for cell-free DNA fragment data.
<code>pyfraglib.FragmentList</code>	Collection of <code>Fragment</code> objects from a single sample with analysis methods.
<code>pyfraglib.FragFile</code>	Reader for fragment files (<code>.frag</code>) stored as Apache Parquet.

4.1.1 `pyfraglib.Fragment`

class `pyfraglib.Fragment`(*read*: [AlignedSegment](#), *mate*: [AlignedSegment](#) | *None*, *mutated_reads*: *set*[*str*] | *None*, *max_insert_size*: *int* = 900, *min_mapq*: *int* = 20) → *None*

Serializable container for cell-free DNA fragment data.

This class represents a single cfDNA fragment extracted from BAM file alignment data, containing all relevant information for downstream fragmentomics analysis. Fragments are immutable once created and include quality control flags for reliable analysis.

The `Fragment` class serves as the fundamental data structure throughout `pyfraglib`, supporting both paired-end and single-end sequencing data as well as mutation annotations.

See also

FragmentList

Collection of fragments from single sample

FragmentCollection

Multi-sample fragment collections

FragFile

Reading/writing `.frag` files

[pyfraglib.stats](#)

Fragment statistics and visualization

Parameters

- `read` (`AlignedSegment`)
- `mate` (`Optional[AlignedSegment]`)
- `mutated_reads` (`Optional[set[str]]`)
- `max_insert_size` (`int`, default: 900)
- `min_mapq` (`int`, default: 20)

Methods

<code>__init__(read, mate, mutated_reads[, ...])</code>	Initialize a Fragment from BAM alignment data.
<code>bams_to_fragments(filepaths, vcfpaths, out_dir)</code>	Process multiple BAM files in parallel and save fragments to .frag files.
<code>build_mutated_reads_set(bam_file, vcf_file)</code>	Build a set of read names that contain mutations from VCF analysis.
<code>create_simulated(start_pos, end_pos, chrom, ...)</code>	Create a simulated Fragment without requiring pysam objects.
<code>from_bam(filepath, vcfpath[, is_nanopore, ...])</code>	Extract fragments from a BAM file with optional mutation annotation.
<code>from_bams(filepaths, vcfpaths[, ...])</code>	Process multiple BAM files in parallel and collect fragments in memory.
<code>get_end_motifs(kmer_len)</code>	Extract k-mer sequences from fragment ends.

Attributes

<code>start_pos</code>	First aligned position of this fragment (0-based).
<code>end_pos</code>	One past the last aligned position of this fragment.
<code>chrom</code>	Contig on which this fragment is located.
<code>is_forward</code>	Flag indicating the directionality of this fragment.
<code>length</code>	Length of this fragment in base pairs.
<code>end5p</code>	Bases at the 5' end of this fragment (5' -> 3' orientation).
<code>end3p</code>	Bases at the 3' end of this fragment (5' -> 3' orientation).
<code>is_bogus</code>	Flag indicating whether this is a bogus fragment (see module- and class-level documentation for definition).
<code>is_mutated</code>	Flag indicating whether this fragment carries a mutation, potentially identifying it as a tumor-derived fragment.

start_pos: `int`

First aligned position of this fragment (0-based).

end_pos: `int`

One past the last aligned position of this fragment.

chrom: `str`

Contig on which this fragment is located.

is_forward: `bool`

Flag indicating the directionality of this fragment.

length: `int`

Length of this fragment in base pairs.

end5p: `str`

Bases at the 5' end of this fragment (5' -> 3' orientation).

end3p: `str`

Bases at the 3' end of this fragment (5' -> 3' orientation).

is_bogus: `bool`

Flag indicating whether this is a bogus fragment (see module- and class-level documentation for definition).

is_mutated: `bool` | `None`

Flag indicating whether this fragment carries a mutation, potentially identifying it as a tumor-derived fragment.

__init__(*read*: `AlignedSegment`, *mate*: `AlignedSegment` | `None`, *mutated_reads*: `set[str]` | `None`,
max_insert_size: `int` = 900, *min_mapq*: `int` = 20) → `None`

Initialize a Fragment from BAM alignment data.

Creates a Fragment object from pysam `AlignedSegment` data, handling both paired-end and single-end sequencing reads. Automatically determines fragment coordinates, end motifs, quality flags, and mutation status.

Parameters

- **read** (`AlignedSegment`) – Primary aligned read from BAM file. Must be a valid pysam `AlignedSegment` with proper alignment information.
- **mate** (`Optional[AlignedSegment]`) – Mate read for paired-end data, or `None` for single-end data. When `None`, fragment is processed as single-ended, i.e. Nanopore
- **mutated_reads** (`Optional[set[str]]`) – Set of read names that contain mutations, or `None` if mutation annotation is not available. Used to mark fragments as mutated based on VCF analysis.
- **max_insert_size** (`int`, default: 900) – Upper bound on fragment insert size in bp; used by `_determine_bogus()`. Defaults to `INSERT_SIZE_UPPER_BOUND`.
- **min_mapq** (`int`, default: 20) – Minimum per-read mapping quality; used by `_determine_bogus()`. Defaults to `MIN_MAPQ`.

Note

- Paired-end processing requires both reads to be properly aligned
- Single-end processing uses the full aligned sequence of the read
- Fragment coordinates follow BAM 0-based conventions
- Quality filtering and bogus detection are applied automatically

get_end_motifs(*kmer_len*: `int`) → `tuple[str, str]`

Extract k-mer sequences from fragment ends.

Retrieves nucleotide sequences from the 5' and 3' ends of the fragment for motif analysis. These end motifs reflect nuclease cleavage patterns and can be used for diversity analysis and tissue-specific signatures.

Parameters

kmer_len (**int**) – Length of k-mer sequences to extract. Must be between 1 and MAX_KMER_LEN (4). Invalid lengths are automatically adjusted to DEFAULT_KMER_LEN (3) with a warning.

Returns

A tuple containing (5p_motif, 3p_motif). Both motifs are nucleotide sequences of the specified length.

Return type

tuple[**str**, **str**]

Note

- Motifs are extracted from stored end sequences during fragment creation
- Invalid k-mer lengths trigger automatic correction and logging
- Used primarily for motif diversity and cleavage pattern analysis

Example

Extract 4-mer end motifs:

```
# Get 4-mer sequences from fragment ends
motif_5p, motif_3p = fragment.get_end_motifs(4)
print(f"5' motif: {motif_5p}")
print(f"3' motif: {motif_3p}")
```

static from_bam(filepath: **str**, vcfp_path: **str** | *None*, is_nanopore: **bool** = *False*, max_insert_size: **int** = 900, min_mapq: **int** = 20) → *FragmentList*

Extract fragments from a BAM file with optional mutation annotation.

Processes a BAM file to extract cfDNA fragments, applying quality filtering and optionally annotating fragments that contain mutations from a VCF file. Supports both paired-end and single-end (Nanopore) sequencing data.

Processing includes:

- **Quality filtering:** MAPQ >= 20, proper fragment lengths
- **Duplicate handling:** Automatic detection and filtering
- **Mutation annotation:** Links VCF variants to BAM reads
- **Progress tracking:** Visual progress bar for large files
- **Memory optimization:** Efficient read caching and cleanup

Parameters

- **filepath** (**str**) – Path to indexed BAM file. Must have corresponding .bai index.
- **vcfp_path** (**Optional**[**str**]) – Optional path to VCF file for mutation annotation. If provided, fragments containing variants will be marked as mutated.
- **is_nanopore** (**bool**, default: *False*) – Whether to process as single-end Nanopore data. Default *False* assumes paired-end Illumina data.

Returns

List of extracted fragments with quality metrics and optional mutation annotations.

Return type

FragmentList

Raises

SystemExit – If BAM file lacks required index or processing fails.

Example

Extract fragments with mutation annotation:

```
from pyfraglib.fragment import Fragment

# Basic extraction without mutations
fragments = Fragment.from_bam("sample.bam", None)

# With mutation annotation
fragments = Fragment.from_bam("sample.bam", "variants.vcf")

print(f"Extracted {len(fragments)} fragments")
print(f"Bogus fragments: {fragments.count_bogus_fragments()}")
```

 See also

- `Fragment.from_bams()`: Process multiple files in parallel
- `Fragment.bams_to_frags()`: Batch processing to .frag files

Parameters

- **max_insert_size** (*int*, default: 900)
- **min_mapq** (*int*, default: 20)

static build_mutated_reads_set(*bam_file*: *AlignmentFile*, *vcf_file*: *VariantFile*) → *set[str]*

Build a set of read names that contain mutations from VCF analysis.

Links VCF variant records to BAM reads by checking which reads contain alternative alleles at variant positions. This enables identification of cfDNA fragments carrying specific mutations for downstream analysis.

The function:

- Processes all SNVs in the VCF file
- Finds overlapping reads at each variant position
- Checks read bases against reference and alternative alleles
- Returns read names that contain mutations

Parameters

- **bam_file** (*AlignmentFile*) – Opened BAM file containing aligned reads. Must be indexed and coordinate-sorted for efficient variant lookups.

- **vcf_file** ([VariantFile](#)) – Opened VCF file containing variant calls. Only SNVs (single nucleotide variants) are processed.

Returns

Set of read names that contain alternative alleles at variant positions. Used to mark fragments as mutated.

Return type

`set[str]`

Note

- Only processes single nucleotide variants (SNVs)
- Handles chromosome naming inconsistencies automatically
- Tracks duplex sequencing support for quality assessment (very specific to our workflow, thus not of interest to most other researchers)
- Skips reads with ambiguous bases (neither ref nor alt)

Example

Build mutated reads set for fragment annotation:

```
import pysam

# Open files
bam_file = pysam.AlignmentFile("sample.bam")
vcf_file = pysam.VariantFile("variants.vcf")

# Build mutated reads set
mutated_reads = Fragment.build_mutated_reads_set(
    bam_file, vcf_file
)
print(f"Found {len(mutated_reads)} mutated reads")

# Close files
bam_file.close()
vcf_file.close()
```

static from_bams(*filepaths*: [list\[str\]](#), *vcfpaths*: [list\[str\]](#) | [None](#), *is_nanopore*: [bool](#) = [False](#),
max_insert_size: [int](#) = 900, *min_mapq*: [int](#) = 20) → [FragmentCollection](#)

Process multiple BAM files in parallel and collect fragments in memory.

max_insert_size and *min_mapq* are forwarded to `Fragment.from_bam()` per worker (via `functools.partial`); see that method for their meaning.

Warning

This function has significant memory overhead. Consider using `Fragment.bams_to_frags()` for large-scale processing or the provided Nextflow pipeline.

Parameters

- `filepaths` (`list[str]`)
- `vcfpaths` (`Optional[list[str]]`)
- `is_nanopore` (`bool`, default: `False`)
- `max_insert_size` (`int`, default: `900`)
- `min_mapq` (`int`, default: `20`)

Return type

`FragmentCollection`

`static bams_to_frags(filepaths: list[str], vcfpaths: list[str] | None, out_dir: str, is_nanopore: bool = False, max_insert_size: int = 900, min_mapq: int = 20) → None`

Process multiple BAM files in parallel and save fragments to `.frag` files.

Batch processing that extracts fragments from multiple BAM files in parallel and writes results directly to disk as compressed `.frag` files. This approach avoids the memory overhead of `Fragment.from_bams()` by not storing all fragments in memory simultaneously. It can still be very expensive.

`max_insert_size` and `min_mapq` are forwarded to `Fragment.from_bam()` per worker (via `functools.partial`); see that method for their meaning.

Note

We use the same multiprocessing idioms as in `from_bams`

Parameters

- `filepaths` (`list[str]`)
- `vcfpaths` (`Optional[list[str]]`)
- `out_dir` (`str`)
- `is_nanopore` (`bool`, default: `False`)
- `max_insert_size` (`int`, default: `900`)
- `min_mapq` (`int`, default: `20`)

Return type

`None`

`classmethod create_simulated(start_pos: int, end_pos: int, chrom: str, length: int, end5p: str, end3p: str, is_forward: bool = True, is_mutated: bool | None = None) → Fragment`

Create a simulated `Fragment` without requiring `pysam` objects.

Creates a `Fragment` object with known properties for simulation purposes, bypassing the normal BAM file processing pipeline. This method is primarily used by the simulation module to generate synthetic cfDNA fragments with realistic characteristics.

Parameters

- `start_pos` (`int`) – Fragment start position (0-based genomic coordinate).
- `end_pos` (`int`) – Fragment end position (0-based, exclusive).
- `chrom` (`str`) – Chromosome name (e.g., “chr1”, “1”).
- `length` (`int`) – Fragment length in base pairs.

- **end5p** (*str*) – 5' end motif sequence (nucleotide string).
- **end3p** (*str*) – 3' end motif sequence (nucleotide string).
- **is_forward** (*bool*, default: True) – Strand orientation. Default True for forward strand.
- **is_mutated** (*bool* | *None*, default: None) – Mutation status, or None if unknown.

Returns

A properly initialized Fragment object with simulated properties.

Return type

Fragment

Note

- Automatically applies quality filtering (bogus detection)
- Bypasses BAM file dependency for simulation workflows
- Used primarily by *pyfraglib.simulator* modules

4.1.2 pyfraglib.FragmentList

class pyfraglib.**FragmentList** → *None*

Collection of Fragment objects from a single sample with analysis methods.

A container class for managing collections of Fragment objects extracted from a single BAM file. Provides methods for fragment manipulation, analysis, and serialization to .frag files for efficient storage and reuse.

Key Features:

- **Fragment management:** Add, iterate, and count fragments
- **Quality metrics:** Count bogus and mutated fragments
- **Motif analysis:** Extract and count end motif patterns
- **Serialization:** Save/load to compressed .frag format
- **Interval operations:** Convert to interval trees for overlap queries

Example

Basic fragment collection usage:

```
from pyfraglib.fragment import Fragment, FragmentList

# Create empty collection
fragments = FragmentList()

# Add fragments (typically from Fragment.from_bam)
fragments.append(fragment1)
fragments.append(fragment2)

# Get statistics
total = fragments.length()
bogus = fragments.count_bogus_fragments()
mutated = fragments.count_mutated_fragments()
```

(continues on next page)

(continued from previous page)

```
print(f"Total: {total}, Bogus: {bogus}, Mutated: {mutated}")

# Save to file
fragments.to_frag_file("sample", "output/")
```

➔ See also

- [Fragment](#) - Individual fragment objects
- [FragmentCollection](#) - Multi-sample fragment collections
- [FragFile](#) - Reading .frag files back into memory

Methods

<code>__init__()</code>	
<code>append(fragment)</code>	Append a fragment to this list.
<code>count_bogus_fragments()</code>	Count the number of bogus fragments.
<code>count_endmotifs(kmer_len)</code>	Count the number of unique fragment end motifs.
<code>count_mutated_fragments()</code>	Count the number of mutated fragments.
<code>length()</code>	Return the number of fragments in this list.
<code>to_frag_file(name, out_dir)</code>	Serialize this fragment list to a Parquet-backed .frag file.
<code>to_interval_table()</code>	Convert this fragment list into an interval table.

`__init__()` → [None](#)

`append(fragment: Fragment)` → [None](#)

Append a fragment to this list.

Parameters

fragment ([Fragment](#))

Return type

[None](#)

`length()` → [int](#)

Return the number of fragments in this list.

Return type

[int](#)

`count_bogus_fragments()` → [int](#)

Count the number of bogus fragments.

Return type

[int](#)

`count_mutated_fragments()` → [int](#)

Count the number of mutated fragments.

Return type

[int](#)

to_frag_file(*name*: *str*, *out_dir*: *str*) → *None*

Serialize this fragment list to a Parquet-backed .frag file.

One row per fragment is written with the schema declared in `pyfraglib.fragment.FRAG_SCHEMA`. The file is written with zstd compression (level 3) and default Parquet row-group sizing.

Parameters

- **name** (*str*) – File basename without the .frag extension.
- **out_dir** (*str*) – Directory that will contain the output file. Must already exist.

Return type

None

count_endmotifs(*kmer_len*: *int*) → *tuple*[*defaultdict*[*str*, *int*], *defaultdict*[*str*, *int*], *int*]

Count the number of unique fragment end motifs.

Returns the 5' and 3' motifs as well as the number of analyzed fragments.

Parameters

kmer_len (*int*)

Return type

tuple[*defaultdict*[*str*, *int*], *defaultdict*[*str*, *int*], *int*]

to_interval_table() → *IntervalTable*

Convert this fragment list into an interval table.

Return type

IntervalTable

4.1.3 pyfraglib.FragFile

class `pyfraglib.FragFile`(*path*: *str*) → *None*

Reader for fragment files (.frag) stored as Apache Parquet.

Provides streaming and bulk access to serialized fragment data with context-manager semantics for automatic cleanup. The underlying storage is a `pyarrow.parquet.ParquetFile` and reads go through `pyarrow.parquet.ParquetFile.iter_batches()` for memory-efficient iteration.

Example

Basic usage with context manager:

```
from pyfraglib.fragfile import FragFile

# Streaming iteration (memory efficient)
with FragFile("sample.frag") as fragfile:
    for fragment in fragfile:
        print(f"{fragment.chrom}:{fragment.start_pos}")

# Bulk loading for analysis
with FragFile("sample.frag") as fragfile:
    fragments = fragfile.get_fragment_list()
    print(f"Loaded {len(fragments)} fragments")
```

 See also

- `pyfraglib.fragment.FragmentList.to_frag_file()` - Writing fragment files.
- `pyfraglib.fragment.Fragment` - Individual fragment objects.
- `pyfraglib.fragment.FragmentList` - Fragment collections.
- `FRAG_SCHEMA` - On-disk column schema.

Parameters

path (`str`)

Methods

<code>__init__(path)</code>	Initialize a FragFile reader for the specified fragment file.
<code>close()</code>	Close the fragment file and release system resources.
<code>get_fragment_list()</code>	Load all fragments from the file into a <code>FragmentList</code> .

Attributes

<code>ITER_BATCH_SIZE</code>	Batch size used by <code>__iter__()</code> when pulling rows from Parquet.
<code>closed</code>	Whether <code>close()</code> has been invoked on this file.

ITER_BATCH_SIZE: `int` = 65536Batch size used by `__iter__()` when pulling rows from Parquet.**__init__** (*path*: `str`) → `None`

Initialize a FragFile reader for the specified fragment file.

Opens the Parquet file lazily — metadata is read but fragment rows are not loaded until iteration or bulk loading is invoked.

Parameters

path (`str`) – Path to a `.frag` Parquet file produced by `FragmentList.to_frag_file()`.

Raises

- `FileNotFoundError` – If the specified file does not exist.
- `pyarrow.lib.ArrowInvalid` – If the file is not a valid Parquet file or does not carry the expected fragment columns.
- `PermissionError` – If the file cannot be opened due to permissions.

 Note

- Column presence is validated on open; extra columns beyond the fragment schema are silently ignored.
- Use a context manager for automatic cleanup.

get_fragment_list() → [FragmentList](#)

Load all fragments from the file into a `FragmentList`.

Reads the entire Parquet table in one shot (more efficient than streaming for bulk loads) and loads each row as a `Fragment`. Use `__iter__()` for very large files.

Returns

A `FragmentList` containing every fragment from the file.

Return type

[FragmentList](#)

Raises

- **ValueError** – If called after `close()`.
- **MemoryError** – If the file is too large to fit in memory.

 Warning

This method loads all fragments into memory simultaneously. Prefer streaming iteration for memory-constrained environments.

 See also

- `__iter__()` - Memory-efficient streaming iteration.
- [pyfraglib.fragment.FragmentList](#) - Fragment collection class.

close() → [None](#)

Close the fragment file and release system resources.

Drops the underlying `pyarrow.parquet.ParquetFile` handle. Called automatically by the context manager and destructor, but may be called manually. Subsequent reads raise `ValueError`.

Return type

[None](#)

property closed: [bool](#)

Whether `close()` has been invoked on this file.

4.2 Core Simulation Classes

The simulation code lives in its own module, but the most important classes get re-exported into the `pyfraglib` namespace:

<code>pyfraglib.FragmentSimulator</code>	cfDNA fragment simulator.
<code>pyfraglib.TissueMixtureSimulator</code>	cfDNA simulator for multi-tissue mixtures.
<code>pyfraglib.NucleaseProfile</code>	Nuclease activity and sequence preference parameters.

4.2.1 pyfraglib.FragmentSimulator

class pyfraglib.FragmentSimulator(sequence_generator: SequenceContextGenerator, nucleosome_map: NucleosomeMap | None = None, chromatin_state: ChromatinState | None = None) → None

cfDNA fragment simulator.

This simulator integrates experimental observations to model cfDNA fragmentation:

1. **Nucleosome positioning:** Nucleosome maps with occupancy scores
2. **Chromatin accessibility:** Tissue-specific chromatin states
3. **Nuclease specificity:** Experimentally-determined preferences
4. **Sequence context:** Genomic sequences from reference FASTA
5. **TF binding protection:** Transcription factor binding site shielding

Main references are Lo et al. 2021 Science, Cristiano et al. 2019 Nature.

Parameters

- **sequence_generator** (SequenceContextGenerator)
- **nucleosome_map** (NucleosomeMap | None, default: None)
- **chromatin_state** (ChromatinState | None, default: None)

Methods

<code>__init__(sequence_generator[, ...])</code>	Initialize the fragment simulator.
<code>generate_nucleosome_map(sequence_generator)</code>	Generate a nucleosome map with customizable chromatin state.
<code>simulate_fragments(chrom, start, end, ...[, ...])</code>	Simulate cfDNA fragments from a genomic region with full biological realism.

Attributes

DI_FRACTION
DI_NUC_MEAN
DI_NUC_STD
MAX_FRAGMENT_SIZE
MIN_FRAGMENT_SIZE
MONO_FRACTION
MONO_NUC_MEAN
MONO_NUC_STD
MONO_SKEW
PERIODICITY_AMPLITUDE
TRI_NUC_MEAN
TRI_NUC_STD

MONO_NUC_MEAN: Final[int] = 167

MONO_NUC_STD: Final[int] = 10

MONO_FRACTION: Final[float] = 0.8

```

MONO_SKEW: Final[float] = -3.0
DI_NUC_MEAN: Final[int] = 334
DI_NUC_STD: Final[int] = 15
DI_FRACTION: Final[float] = 0.15
TRI_NUC_MEAN: Final[int] = 501
TRI_NUC_STD: Final[int] = 35
PERIODICITY_AMPLITUDE: Final[float] = 10.0
MIN_FRAGMENT_SIZE: Final[int] = 40
MAX_FRAGMENT_SIZE: Final[int] = 900

__init__(sequence_generator: SequenceContextGenerator, nucleosome_map: NucleosomeMap | None =
    None, chromatin_state: ChromatinState | None = None) → None

```

Initialize the fragment simulator.

Parameters

- **sequence_generator** ([SequenceContextGenerator](#)) – Sequence context generator (must be constructed with indexed FASTA file)
- **nucleosome_map** ([NucleosomeMap](#) | `None`, default: `None`) – Custom nucleosome positioning data. If `None`, generates default nucleosome map.
- **chromatin_state** ([ChromatinState](#) | `None`, default: `None`) – Chromatin accessibility information. If `None`, generates default chromatin state.

```

static generate_nucleosome_map(sequence_generator: SequenceContextGenerator, beta_alpha: float =
    4.0, beta_beta: float = 3.0) → NucleosomeMap

```

Generate a nucleosome map with customizable chromatin state.

This static method creates a nucleosome map using beta distribution parameters to control chromatin accessibility. Can be used by both the base simulator and tissue-specific simulators.

Parameters

- **sequence_generator** ([SequenceContextGenerator](#)) – Sequence generator to get chromosome information.
- **beta_alpha** (`float`, default: `4.0`) – Alpha parameter for beta distribution. Higher values increase nucleosome occupancy (more compact chromatin).
- **beta_beta** (`float`, default: `3.0`) – Beta parameter for beta distribution. Higher values decrease nucleosome occupancy (more open chromatin).

Returns

Nucleosome map with specified chromatin characteristics.

Return type

[NucleosomeMap](#)

Notes

Common beta parameter combinations: - Open chromatin: (2, 5) - low occupancy, high accessibility - Normal chromatin: (4, 3) - balanced occupancy - Compact chromatin: (6, 2) - high occupancy, low accessibility

```
simulate_fragments(chrom: str, start: int, end: int, num_fragments: int, tissue_type: str = 'healthy',  
                    nuclease_profile: NucleaseProfile | None = None, fragment_size_params: dict[str,  
                    float] | None = None) → FragmentList
```

Simulate cfDNA fragments from a genomic region with full biological realism.

Uses realistic nucleosome positioning, chromatin accessibility, and nuclease-specific cleavage preferences based on experimental data.

Parameters

- **chrom** (str) – Chromosome name
- **start** (int) – Start position
- **end** (int) – End position
- **num_fragments** (int) – Number of fragments to generate
- **tissue_type** (str, default: 'healthy') – Source tissue type affecting chromatin state
- **nuclease_profile** (NucleaseProfile | None, default: None) – Nuclease activity parameters
- **fragment_size_params** (dict[str, float] | None, default: None) – Custom size distribution parameters

Return type

FragmentList

Returns

FragmentList containing biologically realistic simulated fragments

4.2.2 pyfraglib.TissueMixtureSimulator

```
class pyfraglib.TissueMixtureSimulator(sequence_generator: SequenceContextGenerator, **kwargs:  
                                       dict[str, object]) → None
```

cfDNA simulator for multi-tissue mixtures. This class extends the base FragmentSimulator.

Examples

Basic tissue mixture simulation:

```
>>> seq_gen = SequenceContextGenerator("/path/to/reference.fasta")  
>>> simulator = TissueMixtureSimulator(seq_gen)  
>>> fragments = simulator.simulate_tissue_mixture(  
...     tissue_types=["hematopoietic", "tumor"],  
...     tissue_fractions=[0.90, 0.10],  
...     total_fragments=10000,  
...     genomic_regions=[("chr1", 10000000, 11000000)]  
... )
```

Cancer progression study:

```
>>> progression = simulator.simulate_cancer_progression(  
...     normal_profile="hematopoietic",  
...     tumor_fractions=[0.01, 0.05, 0.15],  
...     time_points=["diagnosis", "3months", "6months"],  
...     fragments_per_timepoint=20000,  
...     genomic_regions=[("chr1", 10000000, 20000000)],  
... )
```

(continues on next page)

(continued from previous page)

```
...     size_shift=-20.0, short_enrichment=0.25
... )
```

Parameters

- **sequence_generator** ([SequenceContextGenerator](#))
- **kwargs** (`dict[str, object]`)

Methods

<code>__init__(sequence_generator, **kwargs)</code>	Initialize the fragment simulator.
<code>add_custom_tissue(tissue_profile)</code>	Add a custom tissue profile to the simulator's available profiles.
<code>generate_nucleosome_map(sequence_generator)</code>	Generate a nucleosome map with customizable chromatin state.
<code>simulate_cancer_progression(normal_profile, ...)</code>	Model longitudinal cancer progression.
<code>simulate_fragments(chrom, start, end, ...[, ...])</code>	Simulate cfDNA fragments from a genomic region with full biological realism.
<code>simulate_tissue_mixture(tissue_types, ...[, ...])</code>	Generate cfDNA mixture from multiple tissue sources.

Attributes

<code>DI_FRACTION</code>
<code>DI_NUC_MEAN</code>
<code>DI_NUC_STD</code>
<code>MAX_FRAGMENT_SIZE</code>
<code>MIN_FRAGMENT_SIZE</code>
<code>MONO_FRACTION</code>
<code>MONO_NUC_MEAN</code>
<code>MONO_NUC_STD</code>
<code>MONO_SKEW</code>
<code>PERIODICITY_AMPLITUDE</code>
<code>TRI_NUC_MEAN</code>
<code>TRI_NUC_STD</code>

`__init__(sequence_generator: SequenceContextGenerator, **kwargs: dict[str, object]) → None`

Initialize the fragment simulator.

Parameters

- **sequence_generator** ([SequenceContextGenerator](#)) – Sequence context generator (must be constructed with indexed FASTA file)
- **nucleosome_map** – Custom nucleosome positioning data. If None, generates default nucleosome map.
- **chromatin_state** – Chromatin accessibility information. If None, generates default chromatin state.
- **kwargs** (`dict[str, object]`)

add_custom_tissue(*tissue_profile*: [TissueProfile](#)) → [None](#)

Add a custom tissue profile to the simulator's available profiles.

This method allows users to define custom tissue types with specific fragmentation characteristics beyond the predefined profiles.

Parameters

tissue_profile ([TissueProfile](#)) – Complete tissue profile definition including name, fragment size distribution, nucleosome spacing, chromatin openness, end motif preferences, and methylation level.

Return type

[None](#)

simulate_tissue_mixture(*tissue_types*: [list\[str\]](#), *tissue_fractions*: [list\[float\]](#), *total_fragments*: [int](#), *genomic_regions*: [list\[tuple\[str, int, int\]\]](#), *add_noise*: [bool](#) = [True](#)) → [FragmentList](#)

Generate cfDNA mixture from multiple tissue sources.

Parameters

- **tissue_types** ([list\[str\]](#)) – List of tissue type names to include in the mixture. Must be present in *tissue_profiles* (either predefined or custom).
- **tissue_fractions** ([list\[float\]](#)) – Fraction contribution from each tissue type. Must sum to 1.0 (within 0.001 tolerance). Order corresponds to *tissue_types*.
- **total_fragments** ([int](#)) – Total number of fragments to generate across all tissues. Individual tissue contributions are calculated proportionally.
- **genomic_regions** ([list\[tuple\[str, int, int\]\]](#)) – List of genomic regions (chromosome, start, end) to sample fragments from. Fragments are distributed across regions.
- **add_noise** ([bool](#), default: [True](#)) – Whether to add realistic biological and technical noise:
 - Random fragment loss (~5%)
 - Size variation (± 2 bp standard deviation)
 - End repair artifacts (~2%)

Returns

Combined fragment list containing fragments from all tissues with tissue-specific characteristics. Fragments are shuffled to simulate realistic mixing.

Return type

[FragmentList](#)

Raises

[SystemExit](#) – If tissue fractions don't sum to 1.0 or if unknown tissue types are requested.

Examples

Cancer detection mixture (5% tumor fraction):

```
>>> fragments = simulator.simulate_tissue_mixture(  
...     tissue_types=["hematopoietic", "tumor"],  
...     tissue_fractions=[0.95, 0.05],  
...     total_fragments=50000,  
...     genomic_regions=[("chr1", 1000000, 1500000)],  
...     add_noise=True  
... )
```

```
simulate_cancer_progression(normal_profile: str, tumor_fractions: list[float], time_points: list[str],
                             fragments_per_timepoint: int, genomic_regions: list[tuple[str, int, int]])
                             → dict[str, FragmentList]
```

Model longitudinal cancer progression.

Parameters

- **normal_profile** (*str*) – Name of normal tissue profile to use as background (typically “hematopoietic” for blood-based cfDNA).
- **tumor_fractions** (*list*[*float*]) – Tumor-derived cfDNA fraction at each time point. Values should be in [0, 1] range and typically increase over time for progression studies.
- **time_points** (*list*[*str*]) – Labels for each time point (e.g., [“baseline”, “3months”]). Must have same length as tumor_fractions.
- **fragments_per_timepoint** (*int*) – Number of fragments to generate at each time point. Consistent across time points for comparative analysis.
- **genomic_regions** (*list*[*tuple*[*str*, *int*, *int*]]) – Genomic regions (chromosome, start, end) to sample fragments from. Same regions used for all time points.

Returns

Dictionary mapping time point labels to *FragmentList* objects. Each *FragmentList* contains the cfDNA profile for that time point with appropriate tumor fraction and cancer signatures.

Return type

dict[*str*, *FragmentList*]

Raises

SystemExit – If tumor_fractions and time_points have different lengths.

Examples

```
>>> progression = simulator.simulate_cancer_progression(
...     normal_profile="hematopoietic",
...     tumor_fractions=[0.01, 0.03, 0.08, 0.15],
...     time_points=["diagnosis", "1month", "3months", "6months"],
...     fragments_per_timepoint=25000,
...     genomic_regions=[("chr1", 1000000, 2000000)]
... )
```

4.2.3 pyfraglib.NucleaseProfile

```
class pyfraglib.NucleaseProfile(dnase1_activity: float = 1.0, dnase1l3_activity: float = 1.0, dffb_activity:
                                float = 1.0, dnase1_motif_preference: dict[str, float] | None = None,
                                dnase1l3_motif_preference: dict[str, float] | None = None,
                                dffb_motif_preference: dict[str, float] | None = None) → None
```

Nuclease activity and sequence preference parameters.

This class defines the activity levels and sequence preferences of 3 different nucleases involved in cfDNA fragmentation. The parameters are based on experimental literature and can be customized for different biological conditions or disease states.

Notes

Activity levels are relative and multiplicative:

- 1.0 represents normal physiological activity
- Values >1.0 increase nuclease activity
- Values <1.0 decrease nuclease activity
- 0.0 completely disables the nuclease

Motif preferences are multiplicative factors applied to base cleavage probability based on local sequence context. Higher values increase cleavage probability for sequences containing the motif.

Examples

```
>>> # Healthy individual profile
>>> healthy_profile = NucleaseProfile(
...     dnase1_activity=1.0,
...     dnase113_activity=1.2,
...     dffb_activity=0.1
... )
>>>
>>> # Cancer patient with increased apoptosis
>>> cancer_profile = NucleaseProfile(
...     dnase1_activity=1.1,
...     dnase113_activity=1.0,
...     dffb_activity=2.0
... )
```

Parameters

- **dnase1_activity** ([float](#), default: 1.0)
- **dnase113_activity** ([float](#), default: 1.0)
- **dffb_activity** ([float](#), default: 1.0)
- **dnase1_motif_preference** ([dict\[str, float\]](#) | [None](#), default: None)
- **dnase113_motif_preference** ([dict\[str, float\]](#) | [None](#), default: None)
- **dffb_motif_preference** ([dict\[str, float\]](#) | [None](#), default: None)

Methods

```
__init__([dnase1_activity, ...])
```

Attributes

dffb_activity	Relative activity level of the apoptotic nuclease DNA Fragmentation Factor B (DFFB).
dffb_motif_preference	Sequence motif preferences for DFFB.
dnase1_activity	Relative activity level of DNase I.
dnase1_motif_preference	Sequence motif preferences for DNase I.

continues on next page

Table 4.13 – continued from previous page

<code>dnase1l3_activity</code>	Relative activity level of DNase I Like 3 (DNase1L3).
<code>dnase1l3_motif_preference</code>	Sequence motif preferences for DNase1L3.

dnase1_activity: `float = 1.0`

Relative activity level of DNase I. DNase I shows preference for AT-rich accessible regions.

dnase1l3_activity: `float = 1.0`

Relative activity level of DNase I Like 3 (DNase1L3). This is the primary nuclease responsible for cfDNA fragmentation in healthy individuals. Shows strong CC dinucleotide preference.

dffb_activity: `float = 1.0`

Relative activity level of the apoptotic nuclease DNA Fragmentation Factor B (DFFB). More active during cell death and shows preference for nucleosome linker regions.

dnase1_motif_preference: `dict[str, float] | None = None`

Sequence motif preferences for DNase I. If None, uses default preferences based on literature (AT-rich sequences preferred).

dnase1l3_motif_preference: `dict[str, float] | None = None`

Sequence motif preferences for DNase1L3. If None, uses default preferences (strong CC dinucleotide preference).

dffb_motif_preference: `dict[str, float] | None = None`

Sequence motif preferences for DFFB. If None, uses default preferences (slight A preference).

__init__(*dnase1_activity*: `float = 1.0`, *dnase1l3_activity*: `float = 1.0`, *dffb_activity*: `float = 1.0`, *dnase1_motif_preference*: `dict[str, float] | None = None`, *dnase1l3_motif_preference*: `dict[str, float] | None = None`, *dffb_motif_preference*: `dict[str, float] | None = None`) → `None`

Parameters

- **dnase1_activity** (`float`, default: 1.0)
- **dnase1l3_activity** (`float`, default: 1.0)
- **dffb_activity** (`float`, default: 1.0)
- **dnase1_motif_preference** (`dict[str, float] | None`, default: None)
- **dnase1l3_motif_preference** (`dict[str, float] | None`, default: None)
- **dffb_motif_preference** (`dict[str, float] | None`, default: None)

4.3 Module List

Alternative, please select a class of interest to you from the following list:

<code>pyfraglib.core</code>	Core Utilities and Infrastructure for pyfraglib
<code>pyfraglib.fragment</code>	Fragment Processing and BAM File Analysis
<code>pyfraglib.fragmentfile</code>	Fragment File I/O and Serialization
<code>pyfraglib.math</code>	Common math functions
<code>pyfraglib.stats</code>	Fragment Statistics and Visualization
<code>pyfraglib.lengths</code>	Fragment length analysis and visualization
<code>pyfraglib.scores</code>	Fragmentomics Scores
<code>pyfraglib.simulator</code>	cfDNA Simulator

continues on next page

Table 4.14 – continued from previous page

<code>pyfraglib.simulator.fragment_simulator</code>	Biologically Realistic cfDNA Fragment Simulation
<code>pyfraglib.simulator.tissue_mixture_simulator</code>	Tissue Mixture cfDNA Simulation

4.3.1 pyfraglib.core

Core Utilities and Infrastructure for pyfraglib

This module provides core utilities and shared functions used throughout the pyfraglib library. It includes chromosome definitions, error handling, logging configuration, multiprocessing support, and statistical diversity indices.

Key Components

- **Chromosome Definitions:** Reference genome coordinates for hg19 and hg38
- **Error Handling:** Graceful failure mechanisms and logging
- **Multiprocessing:** CPU detection and shared memory management
- **Statistics:** Diversity indices (Shannon entropy, Simpson index)
- **Utilities:** Chromosome name standardization and validation

Functions

- `get_logger()`: Configure and retrieve logger instances
- `fail()`: Graceful error handling with proper cleanup
- `detect_cpus()`: Detect available CPU cores for multiprocessing
- `get_chromosome_length()`: Get chromosome length for reference genome
- `homogenize_contig_name()`: Standardize chromosome naming conventions
- `shannon_entropy()`: Calculate Shannon entropy for diversity analysis
- `simpson_index()`: Calculate Simpson diversity index

Example Usage

```
from pyfraglib.core import get_logger, detect_cpus, get_chromosome_length

# Configure logging
logger = get_logger()
logger.info("Starting analysis...")

# Detect CPU cores for multiprocessing
n_cpus = detect_cpus()
print(f"Using {n_cpus} CPU cores")

# Get chromosome length
chr1_length = get_chromosome_length("1", "hg38")
print(f"Chromosome 1 length: {chr1_length} bp")
```

Multiprocessing Support

The module provides multiprocessing infrastructure through the [PyfragManager](#) class, which enables automatic locking while sharing data structures across processes:

```
from pyfraglib.core import PyfragManager
from pyfraglib.fragment import FragmentCollection

# Register shared data structures
PyfragManager.register("FragmentCollection", FragmentCollection)

# Use in multiprocessing context
with PyfragManager() as manager:
    shared_collection = manager.FragmentCollection()
```

Error Handling

The [fail\(\)](#) function provides centralized error handling with proper cleanup:

```
from pyfraglib.core import fail

# Graceful failure with logging
if not os.path.exists(required_file):
    fail(f"Required file not found: {required_file}")
```

Constants

- `LOGGER_NAME`: Standard logger name for pyfraglib
- `hg19_chromosomes`: Chromosome definitions for hg19/GRCh37
- `hg38_chromosomes`: Chromosome definitions for hg38/GRCh38

License

This file is part of pyfraglib, a software suite to calculate fragmentomics features from cfDNA and perform downstream analyses.

Copyright (C) 2025 Daniel Schütte, daniel.schuette@iccb-cologne.org

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program. If not, see <<https://www.gnu.org/licenses/>>.

Functions

detect_cpus()	Return the number of CPUs currently available.
fail(msg)	Terminate the program even if called from a subprocess.
get_chromosome_length(chrom[, genome])	Get the length of a given chromosome in the context of a genome assembly.
get_logger()	Get the standard pyfraglib logger.

continues on next page

Table 4.15 – continued from previous page

<code>homogenize_contig_name(contig)</code>	Transform a <code>contig</code> name into the format used internally by <code>pyfraglib</code> .
<code>homogenize_to_chrom_naming_convention(...)</code>	Take a BAM file header and a <code>contig</code> name and return the <code>contig</code> name according to the headers naming convention.
<code>parse_bed_file(bed_path)</code>	Parse BED file and return list of genomic regions.
<code>shannon_entropy(proportions)</code>	Calculate Shannon entropy for a list of floats.
<code>simpson_index(proportions)</code>	Calculate Simpson index for a list of floats.

Classes

<code>PyfragManager([address, authkey, ...])</code>

Exceptions

<code>CodeUnreachableError(msg)</code>	An exception thrown whenever code should be unreachable.
<code>PyfraglibException(msg)</code>	A base class for custom <code>pyfraglib</code> exceptions.

`pyfraglib.core.get_logger()` → [`Logger`](#)

Get the standard `pyfraglib` logger.

Return type

[`Logger`](#)

`pyfraglib.core.fail(msg: str)` → [`NoReturn`](#)

Terminate the program even if called from a subprocess.

Parameters

`msg` ([`str`](#))

Return type

[`NoReturn`](#)

exception `pyfraglib.core.PyfraglibException(msg: str)` → [`None`](#)

A base class for custom `pyfraglib` exceptions.

Parameters

`msg` ([`str`](#))

`__init__` (`msg: str`) → [`None`](#)

Parameters

`msg` ([`str`](#))

exception `pyfraglib.core.CodeUnreachableError(msg: str)` → [`None`](#)

An exception thrown whenever code should be unreachable.

Parameters

`msg` ([`str`](#))

`__init__` (`msg: str`) → [`None`](#)

Parameters

`msg` ([`str`](#))

```
class pyfraglib.core.PyfragManager(address=None, authkey=None, serializer='pickle', ctx=None, *
                                   (Keyword-only parameters separator (PEP 3102)),
                                   shutdown_timeout=1.0)
```

```
FragmentCollection(*args, **kws)
```

```
pyfraglib.core.get_chromosome_length(chrom: str, genome: str = 'hg19') → int
```

Get the length of a given chromosome in the context of a genome assembly.

Note

- hg38 from <https://www.ncbi.nlm.nih.gov/grc/human/data?asm=GRCh38>
- hg19 from <https://www.ncbi.nlm.nih.gov/grc/human/data?asm=GRCh37>

Parameters

- **chrom** (*str*)
- **genome** (*str*, default: 'hg19')

Return type

int

```
pyfraglib.core.homogenize_contig_name(contig: str) → str
```

Transform a contig name into the format used internally by pyfraglib. All code dealing with chromosome names should use this function to ensure consistency.

Parameters

contig (*str*)

Return type

str

```
pyfraglib.core.homogenize_to_chrom_naming_convention(contig: str, header: dict[str, object]) → str
```

Take a BAM file header and a contig name and return the contig name according to the headers naming convention. If the header is invalid, return contig unaltered.

Parameters

- **contig** (*str*)
- **header** (*dict*[*str*, *object*])

Return type

str

```
pyfraglib.core.shannon_entropy(proportions: list[float]) → float
```

Calculate Shannon entropy for a list of floats.

Note

The input list needs to be normalized to proportions.

Parameters

proportions (*list*[*float*])

Return type`float``pyfraglib.core.simpson_index(proportions: list[float]) → float`

Calculate Simpson index for a list of floats.

Note

The input list needs to be normalized to proportions.

Parameters`proportions (list[float])`**Return type**`float``pyfraglib.core.parse_bed_file(bed_path: str) → list[tuple[str, int, int]]`

Parse BED file and return list of genomic regions.

Parameters`bed_path (str)` – Path to BED file containing genomic regions**Returns**

List of (chromosome, start, end) tuples

Return type`list[tuple[str, int, int]]`**Notes**

Supports standard BED format with at least 3 columns: chromosome, start, end. Additional columns are ignored.

`pyfraglib.core.detect_cpus() → int`

Return the number of CPUs currently available.

Return type`int`

4.3.2 pyfraglib.fragment

Fragment Processing and BAM File Analysis

This module provides the core data structures and functions for processing cell-free DNA (cfDNA) fragments from BAM files. It includes the fundamental `Fragment` class with support for both paired-end and single-end (Nanopore) sequencing data.

Key Classes

- `Fragment`: Individual cfDNA fragment with genomic coordinates and properties
- `FragmentList`: Collection of fragments from a single sample
- `FragmentCollection`: Multi-sample fragment collection for batch processing
- `IntervalTable`: Efficient genomic interval management using interval trees (was mostly used in older versions of certain algorithms, but actively used for test cases)

Core Features

- **BAM File Processing:** Extract fragments from BAM files
- **Mutation Annotation:** Link VCF variants to BAM reads for mutated fragment identification
- **Quality Control:** Filter bogus fragments and apply quality thresholds
- **Batch Processing:** Parallel processing of multiple BAM files
- **Serialization:** Save/load fragment data in compressed .frag format

Fragment Properties

Each Fragment contains:

- **Chromosome:** Standardized name of contig from which the fragment originates
- **Genomic Coordinates:** Start and end positions (0-based)
- **Length:** Fragment length in base pairs
- **End Motifs:** 5' and 3' end sequences (k-mers)
- **Mutation Status:** Whether fragment contains variants
- **Quality Flags:** Bogus fragment detection and quality metrics

Please see the class documentation for additional information.

Example Usage

Basic fragment extraction and analysis:

```
from pyfraglib.fragment import Fragment, FragmentList

# Extract fragments from BAM file
fragments = Fragment.from_bam("sample.bam", "variants.vcf")

# Access fragment properties
for fragment in fragments:
    print(f"Fragment at {fragment.chrom}:{fragment.start_pos}-"
          f"{fragment.end_pos}")
    print(f"Length: {fragment.length} bp")
    print(f"5' motif: {fragment.end5p}")
    print(f"3' motif: {fragment.end3p}")
    print(f"Mutated: {fragment.is_mutated}")

# Filter out bogus fragments
clean_fragments = FragmentList([f for f in fragments if not f.is_bogus])

# Save to file
clean_fragments.to_frag_file("sample_clean", "output/")
```

Batch Processing

Process multiple BAM files efficiently:

```
from pyfraglib.fragment import Fragment

# Process multiple BAM files in parallel
bam_files = ["sample1.bam", "sample2.bam", "sample3.bam"]
vcf_files = ["sample1.vcf", "sample2.vcf", "sample3.vcf"]

# Batch processing to .frag files.
Fragment.bams_to_frags(bam_files, vcf_files, "output/")

# Load all fragments into memory
fragment_collection = Fragment.from_bams(bam_files, vcf_files)
```

Importantly, we discourage this method of processing larger sets of samples. With >20 samples, we recommend using the Nextflow pipeline we are providing.

Interval Operations

Efficient genomic interval queries:

```
from pyfraglib.fragment import IntervalTable

# Create interval table
intervals = IntervalTable()
intervals.add_bed_file("regions.bed")

# Find overlapping fragments
overlapping_fragments = intervals.find_overlapping_fragments(fragments)

# Check if fragment overlaps any interval
for fragment in fragments:
    if intervals.overlaps(
        fragment.chrom, fragment.start_pos, fragment.end_pos
    ):
        print(f"Fragment {fragment} overlaps with intervals")
```

Quality Control

Fragment quality assessment and filtering:

```
from pyfraglib.fragment import Fragment, FragmentList

# Extract with quality filtering
fragments = Fragment.from_bam("sample.bam", "variants.vcf")

# Count quality metrics
total_fragments = len(fragments)
bogus_fragments = sum(1 for f in fragments if f.is_bogus)
mutated_fragments = sum(1 for f in fragments if f.is_mutated)

print(f"Total fragments: {total_fragments}")
print(f"Bogus fragments: {bogus_fragments} "
      f"({bogus_fragments/total_fragments:.1%})")
print(f"Mutated fragments: {mutated_fragments} "
```

(continues on next page)

(continued from previous page)

```

        f"({mutated_fragments/total_fragments:.1%})")

# Apply additional filtering
high_quality_fragments = FragmentList([
    f for f in fragments
    if not f.is_bogus and f.length >= 50 and f.length <= 500
])

```

Nanopore Support

Support for single-end long-read sequencing is implemented generically, but only tested using Nanopore datasets. Thus, the respective options are named “nanopore” instead of “long-read, single-ended sequencing”:

```

from pyfraglib.fragment import Fragment

# Process Nanopore BAM file
nanopore_fragments = Fragment.from_bam("nanopore.bam", is_nanopore=True)

```

Constants

- INSERT_SIZE_UPPER_BOUND: Maximum allowed insert size (900 bp)
- MIN_MAPQ: Minimum mapping quality threshold (20)
- DEFAULT_KMER_LEN: Default k-mer length for end motifs (3)
- MAX_KMER_LEN: Maximum k-mer length for end motifs (4)
- VALID_CHROMOSOME_NAMES: List of valid chromosome names

Thread Safety

The Fragment class is immutable and thus thread-safe. Collection classes use appropriate locking mechanisms for concurrent access. BAM file processing uses multiprocessing for parallel execution across multiple files - which again is not recommended for larger datasets (see above).

Performance Considerations

- Use `bams_to_frags()` for large-scale processing to avoid memory issues; or even better, use the provided Nextflow pipeline
- Fragment collections are memory-intensive
- Interval trees provide $O(\log n)$ query performance for genomic overlaps

License

This file is part of pyfraglib, a software suite to calculate fragmentomics features from cfDNA and perform downstream analyses.

Copyright (C) 2025 Daniel Schütte, daniel.schuette@iccb-cologne.org

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU

General Public License for more details. You should have received a copy of the GNU General Public License along with this program. If not, see <<https://www.gnu.org/licenses/>>.

Functions

<code>is_duplex(read)</code>	Determine whether a read has duplex support, based on an project-specific definition.
<code>task0(filepath, vcfpath, frags_per_bam, ...)</code>	Helper function for parallel BAM file processing.
<code>task1(filepath, vcfpath, out_dir, is_nanopore)</code>	Helper function for parallel BAM file processing.

Classes

<code>Fragment(read, mate, mutated_reads[, ...])</code>	Serializable container for cell-free DNA fragment data.
<code>FragmentCollection()</code>	Multi-sample container for FragmentList collections with batch operations.
<code>FragmentList()</code>	Collection of Fragment objects from a single sample with analysis methods.
<code>IntervalTable()</code>	Efficient genomic interval management using chromosome-indexed interval trees.

```
class pyfraglib.fragment.Fragment(read: AlignedSegment, mate: AlignedSegment | None, mutated_reads:
    set[str] | None, max_insert_size: int = 900, min_mapq: int = 20) →
    None
```

Serializable container for cell-free DNA fragment data.

This class represents a single cfDNA fragment extracted from BAM file alignment data, containing all relevant information for downstream fragmentomics analysis. Fragments are immutable once created and include quality control flags for reliable analysis.

The Fragment class serves as the fundamental data structure throughout pyfraglib, supporting both paired-end and single-end sequencing data as well as mutation annotations.

➡ See also

FragmentList

Collection of fragments from single sample

FragmentCollection

Multi-sample fragment collections

FragFile

Reading/writing .frag files

pyfraglib.stats

Fragment statistics and visualization

Parameters

- **read** (*AlignedSegment*)
- **mate** (*Optional*[*AlignedSegment*])
- **mutated_reads** (*Optional*[set[str]])

- **max_insert_size** ([int](#), default: 900)
- **min_mapq** ([int](#), default: 20)

start_pos: [int](#)

First aligned position of this fragment (0-based).

end_pos: [int](#)

One past the last aligned position of this fragment.

chrom: [str](#)

Contig on which this fragment is located.

is_forward: [bool](#)

Flag indicating the directionality of this fragment.

length: [int](#)

Length of this fragment in base pairs.

end5p: [str](#)

Bases at the 5' end of this fragment (5' -> 3' orientation).

end3p: [str](#)

Bases at the 3' end of this fragment (5' -> 3' orientation).

is_bogus: [bool](#)

Flag indicating whether this is a bogus fragment (see module- and class-level documentation for definition).

is_mutated: [bool](#) | [None](#)

Flag indicating whether this fragment carries a mutation, potentially identifying it as a tumor-derived fragment.

__init__(*read*: [AlignedSegment](#), *mate*: [AlignedSegment](#) | [None](#), *mutated_reads*: [set\[str\]](#) | [None](#), *max_insert_size*: [int](#) = 900, *min_mapq*: [int](#) = 20) → [None](#)

Initialize a Fragment from BAM alignment data.

Creates a Fragment object from pysam [AlignedSegment](#) data, handling both paired-end and single-end sequencing reads. Automatically determines fragment coordinates, end motifs, quality flags, and mutation status.

Parameters

- **read** ([AlignedSegment](#)) – Primary aligned read from BAM file. Must be a valid pysam [AlignedSegment](#) with proper alignment information.
- **mate** ([Optional](#)[[AlignedSegment](#)]) – Mate read for paired-end data, or [None](#) for single-end data. When [None](#), fragment is processed as single-ended, i.e. Nanopore
- **mutated_reads** ([Optional](#)[[set\[str\]](#)]) – Set of read names that contain mutations, or [None](#) if mutation annotation is not available. Used to mark fragments as mutated based on VCF analysis.
- **max_insert_size** ([int](#), default: 900) – Upper bound on fragment insert size in bp; used by [_determine_bogus\(\)](#). Defaults to [INSERT_SIZE_UPPER_BOUND](#).
- **min_mapq** ([int](#), default: 20) – Minimum per-read mapping quality; used by [_determine_bogus\(\)](#). Defaults to [MIN_MAPQ](#).

Note

- Paired-end processing requires both reads to be properly aligned
- Single-end processing uses the full aligned sequence of the read
- Fragment coordinates follow BAM 0-based conventions
- Quality filtering and bogus detection are applied automatically

get_end_motifs(*kmer_len*: *int*) → *tuple*[*str*, *str*]

Extract k-mer sequences from fragment ends.

Retrieves nucleotide sequences from the 5' and 3' ends of the fragment for motif analysis. These end motifs reflect nuclease cleavage patterns and can be used for diversity analysis and tissue-specific signatures.

Parameters

kmer_len (*int*) – Length of k-mer sequences to extract. Must be between 1 and MAX_KMER_LEN (4). Invalid lengths are automatically adjusted to DEFAULT_KMER_LEN (3) with a warning.

Returns

A tuple containing (5p_motif, 3p_motif). Both motifs are nucleotide sequences of the specified length.

Return type

tuple[*str*, *str*]

Note

- Motifs are extracted from stored end sequences during fragment creation
- Invalid k-mer lengths trigger automatic correction and logging
- Used primarily for motif diversity and cleavage pattern analysis

Example

Extract 4-mer end motifs:

```
# Get 4-mer sequences from fragment ends
motif_5p, motif_3p = fragment.get_end_motifs(4)
print(f"5' motif: {motif_5p}")
print(f"3' motif: {motif_3p}")
```

static from_bam(*filepath*: *str*, *vcfpath*: *str* | *None*, *is_nanopore*: *bool* = *False*, *max_insert_size*: *int* = 900, *min_mapq*: *int* = 20) → *FragmentList*

Extract fragments from a BAM file with optional mutation annotation.

Processes a BAM file to extract cfDNA fragments, applying quality filtering and optionally annotating fragments that contain mutations from a VCF file. Supports both paired-end and single-end (Nanopore) sequencing data.

Processing includes:

- **Quality filtering:** MAPQ >= 20, proper fragment lengths

- **Duplicate handling:** Automatic detection and filtering
- **Mutation annotation:** Links VCF variants to BAM reads
- **Progress tracking:** Visual progress bar for large files
- **Memory optimization:** Efficient read caching and cleanup

Parameters

- **filepath** ([str](#)) – Path to indexed BAM file. Must have corresponding .bai index.
- **vcfpath** ([Optional\[str\]](#)) – Optional path to VCF file for mutation annotation. If provided, fragments containing variants will be marked as mutated.
- **is_nanopore** ([bool](#), default: `False`) – Whether to process as single-end Nanopore data. Default `False` assumes paired-end Illumina data.

Returns

List of extracted fragments with quality metrics and optional mutation annotations.

Return type

[FragmentList](#)

Raises

[SystemExit](#) – If BAM file lacks required index or processing fails.

Example

Extract fragments with mutation annotation:

```
from pyfraglib.fragment import Fragment

# Basic extraction without mutations
fragments = Fragment.from_bam("sample.bam", None)

# With mutation annotation
fragments = Fragment.from_bam("sample.bam", "variants.vcf")

print(f"Extracted {len(fragments)} fragments")
print(f"Bogus fragments: {fragments.count_bogus_fragments()}")
```

➡ See also

- [Fragment.from_bams\(\)](#): Process multiple files in parallel
- [Fragment.bams_to_fragments\(\)](#): Batch processing to .frag files

Parameters

- **max_insert_size** ([int](#), default: 900)
- **min_mapq** ([int](#), default: 20)

static [build_mutated_reads_set](#)(*bam_file*: [AlignmentFile](#), *vcf_file*: [VariantFile](#)) → [set\[str\]](#)

Build a set of read names that contain mutations from VCF analysis.

Links VCF variant records to BAM reads by checking which reads contain alternative alleles at variant positions. This enables identification of cfDNA fragments carrying specific mutations for downstream analysis.

The function:

- Processes all SNVs in the VCF file
- Finds overlapping reads at each variant position
- Checks read bases against reference and alternative alleles
- Returns read names that contain mutations

Parameters

- **bam_file** ([AlignmentFile](#)) – Opened BAM file containing aligned reads. Must be indexed and coordinate-sorted for efficient variant lookups.
- **vcf_file** ([VariantFile](#)) – Opened VCF file containing variant calls. Only SNVs (single nucleotide variants) are processed.

Returns

Set of read names that contain alternative alleles at variant positions. Used to mark fragments as mutated.

Return type

[set](#)[[str](#)]

Note

- Only processes single nucleotide variants (SNVs)
- Handles chromosome naming inconsistencies automatically
- Tracks duplex sequencing support for quality assessment (very specific to our workflow, thus not of interest to most other researchers)
- Skips reads with ambiguous bases (neither ref nor alt)

Example

Build mutated reads set for fragment annotation:

```
import pysam

# Open files
bam_file = pysam.AlignmentFile("sample.bam")
vcf_file = pysam.VariantFile("variants.vcf")

# Build mutated reads set
mutated_reads = Fragment.build_mutated_reads_set(
    bam_file, vcf_file
)
print(f"Found {len(mutated_reads)} mutated reads")

# Close files
```

(continues on next page)

(continued from previous page)

```
bam_file.close()
vcf_file.close()
```

static from_bams(*filepaths*: [list\[str\]](#), *vcfpaths*: [list\[str\]](#) | *None*, *is_nanopore*: *bool* = *False*,
max_insert_size: *int* = 900, *min_mapq*: *int* = 20) → [FragmentCollection](#)

Process multiple BAM files in parallel and collect fragments in memory.

max_insert_size and *min_mapq* are forwarded to [Fragment.from_bam\(\)](#) per worker (via `functools.partial`); see that method for their meaning.

Warning

This function has significant memory overhead. Consider using [Fragment.bams_to_frags\(\)](#) for large-scale processing or the provided Nextflow pipeline.

Parameters

- **filepaths** ([list\[str\]](#))
- **vcfpaths** ([Optional\[list\[str\]\]](#))
- **is_nanopore** (*bool*, default: *False*)
- **max_insert_size** (*int*, default: 900)
- **min_mapq** (*int*, default: 20)

Return type

[FragmentCollection](#)

static bams_to_frags(*filepaths*: [list\[str\]](#), *vcfpaths*: [list\[str\]](#) | *None*, *out_dir*: *str*, *is_nanopore*: *bool* =
False, *max_insert_size*: *int* = 900, *min_mapq*: *int* = 20) → *None*

Process multiple BAM files in parallel and save fragments to .frag files.

Batch processing that extracts fragments from multiple BAM files in parallel and writes results directly to disk as compressed .frag files. This approach avoids the memory overhead of [Fragment.from_bams\(\)](#) by not storing all fragments in memory simultaneously. It can still be very expensive.

max_insert_size and *min_mapq* are forwarded to [Fragment.from_bam\(\)](#) per worker (via `functools.partial`); see that method for their meaning.

Note

We use the same multiprocessing idioms as in `from_bams`

Parameters

- **filepaths** ([list\[str\]](#))
- **vcfpaths** ([Optional\[list\[str\]\]](#))
- **out_dir** (*str*)
- **is_nanopore** (*bool*, default: *False*)
- **max_insert_size** (*int*, default: 900)

- **min_mapq** ([int](#), default: 20)

Return type

[None](#)

classmethod **create_simulated**(*start_pos*: [int](#), *end_pos*: [int](#), *chrom*: [str](#), *length*: [int](#), *end5p*: [str](#), *end3p*: [str](#), *is_forward*: [bool](#) = True, *is_mutated*: [bool](#) | [None](#) = None) → [Fragment](#)

Create a simulated [Fragment](#) without requiring [pysam](#) objects.

Creates a [Fragment](#) object with known properties for simulation purposes, bypassing the normal BAM file processing pipeline. This method is primarily used by the simulation module to generate synthetic cfDNA fragments with realistic characteristics.

Parameters

- **start_pos** ([int](#)) – Fragment start position (0-based genomic coordinate).
- **end_pos** ([int](#)) – Fragment end position (0-based, exclusive).
- **chrom** ([str](#)) – Chromosome name (e.g., “chr1”, “1”).
- **length** ([int](#)) – Fragment length in base pairs.
- **end5p** ([str](#)) – 5’ end motif sequence (nucleotide string).
- **end3p** ([str](#)) – 3’ end motif sequence (nucleotide string).
- **is_forward** ([bool](#), default: True) – Strand orientation. Default True for forward strand.
- **is_mutated** ([bool](#) | [None](#), default: None) – Mutation status, or None if unknown.

Returns

A properly initialized [Fragment](#) object with simulated properties.

Return type

[Fragment](#)

Note

- Automatically applies quality filtering (bogus detection)
- Bypasses BAM file dependency for simulation workflows
- Used primarily by [pyfraglib.simulator](#) modules

pyfraglib.fragment.task0(*filepath*: [str](#), *vcfpath*: [str](#) | [None](#), *frags_per_bam*: [FragmentCollection](#), *is_nanopore*: [bool](#), *max_insert_size*: [int](#) = 900, *min_mapq*: [int](#) = 20) → [None](#)

Helper function for parallel BAM file processing.

Note

Constraints of multiprocessing and pickle force us to define this function outside of `from_bams` and `bams_to_frags`. Tasks 0 and 1 only differ in how they treat the resulting [FragmentList](#). Whereas `t0` writes the into a shared buffer, `t1` writes them to disk immediately. That is freeing the used memory and reducing our mem footprint.

Parameters

- `filepath` (`str`)
- `vcfpath` (`str` | `None`)
- `frags_per_bam` (`FragmentCollection`)
- `is_nanopore` (`bool`)
- `max_insert_size` (`int`, default: 900)
- `min_mapq` (`int`, default: 20)

Return type

`None`

`pyfraglib.fragment.task1(filepath: str, vcfpath: str | None, out_dir: str, is_nanopore: bool, max_insert_size: int = 900, min_mapq: int = 20) → None`

Helper function for parallel BAM file processing.

Parameters

- `filepath` (`str`)
- `vcfpath` (`str` | `None`)
- `out_dir` (`str`)
- `is_nanopore` (`bool`)
- `max_insert_size` (`int`, default: 900)
- `min_mapq` (`int`, default: 20)

Return type

`None`

`class pyfraglib.fragment.FragmentList → None`

Collection of Fragment objects from a single sample with analysis methods.

A container class for managing collections of Fragment objects extracted from a single BAM file. Provides methods for fragment manipulation, analysis, and serialization to .frag files for efficient storage and reuse.

Key Features:

- **Fragment management:** Add, iterate, and count fragments
- **Quality metrics:** Count bogus and mutated fragments
- **Motif analysis:** Extract and count end motif patterns
- **Serialization:** Save/load to compressed .frag format
- **Interval operations:** Convert to interval trees for overlap queries

Example

Basic fragment collection usage:

```
from pyfraglib.fragment import Fragment, FragmentList

# Create empty collection
fragments = FragmentList()

# Add fragments (typically from Fragment.from_bam)
```

(continues on next page)

(continued from previous page)

```

fragments.append(fragment1)
fragments.append(fragment2)

# Get statistics
total = fragments.length()
bogus = fragments.count_bogus_fragments()
mutated = fragments.count_mutated_fragments()

print(f"Total: {total}, Bogus: {bogus}, Mutated: {mutated}")

# Save to file
fragments.to_frag_file("sample", "output/")

```

➡ See also

- [Fragment](#) - Individual fragment objects
- [FragmentCollection](#) - Multi-sample fragment collections
- [FragFile](#) - Reading .frag files back into memory

`__init__()` → [None](#)

`append(fragment: Fragment)` → [None](#)

Append a fragment to this list.

Parameters

`fragment` ([Fragment](#))

Return type

[None](#)

`length()` → [int](#)

Return the number of fragments in this list.

Return type

[int](#)

`count_bogus_fragments()` → [int](#)

Count the number of bogus fragments.

Return type

[int](#)

`count_mutated_fragments()` → [int](#)

Count the number of mutated fragments.

Return type

[int](#)

`to_frag_file(name: str, out_dir: str)` → [None](#)

Serialize this fragment list to a Parquet-backed .frag file.

One row per fragment is written with the schema declared in `pyfraglib.fragment.FRAG_SCHEMA`. The file is written with zstd compression (level 3) and default Parquet row-group sizing.

Parameters

- **name** (`str`) – File basename without the `.frag` extension.
- **out_dir** (`str`) – Directory that will contain the output file. Must already exist.

Return type`None`

count_endmotifs(*kmer_len*: `int`) → `tuple[defaultdict[str, int], defaultdict[str, int], int]`

Count the number of unique fragment end motifs.

Returns the 5' and 3' motifs as well as the number of analyzed fragments.

Parameters

kmer_len (`int`)

Return type

`tuple[defaultdict[str, int], defaultdict[str, int], int]`

to_interval_table() → `IntervalTable`

Convert this fragment list into an interval table.

Return type`IntervalTable`

`pyfraglib.fragment.is_duplex`(*read*: `AlignedSegment`) → `bool`

Determine whether a read has duplex support, based on an project-specific definition.

Parameters

read (`AlignedSegment`)

Return type`bool`

class `pyfraglib.fragment.IntervalTable` → `None`

Efficient genomic interval management using chromosome-indexed interval trees.

Provides fast genomic interval operations by maintaining separate interval trees for each chromosome. Enables efficient overlap queries and intersection detection.

Key Features: * **Chromosome indexing**: Separate interval trees per chromosome * **Efficient queries**: $O(\log n)$ overlap detection performance * **Fragment integration**: Direct fragment insertion and querying

Example

Basic interval operations:

```
from pyfraglib.fragment import IntervalTable, Fragment

# Create interval table
intervals = IntervalTable()

# Insert fragments
fragments = Fragment.from_bam("sample.bam")
for fragment in fragments:
    if not fragment.is_bogus:
        intervals.insert(fragment)

# Query overlaps in a genomic window
```

(continues on next page)

(continued from previous page)

```
overlaps = intervals.get_overlaps("chr1", 10000000, 10010000)
print(f"Found {len(overlaps)} overlapping intervals")
```

Note

- Interval trees are created on-demand for each chromosome
- Uses 0-based coordinates following BAM conventions

See also

- [FragmentList.to_interval_table\(\)](#) - Convert fragments to intervals

`__init__()` → `None`

`insert(fragment: Fragment)` → `None`

Parameters

fragment ([Fragment](#))

Return type

`None`

`get_overlaps(chrom: str, win_start: int, win_end: int)` → `list[Interval]`

Parameters

- **chrom** ([str](#))
- **win_start** ([int](#))
- **win_end** ([int](#))

Return type

`list[Interval]`

`class pyfraglib.fragment.FragmentCollection` → `None`

Multi-sample container for `FragmentList` collections with batch operations.

Dictionary-like container that manages `FragmentList` objects from multiple samples, enabling cross-sample analysis and batch processing operations. Typically used for studies involving multiple BAM files or comparative analysis workflows. Obviously very memory-intensive as soon as the number of samples grows larger.

Key Features: * **Multi-sample management**: Store fragments from multiple samples * **Dictionary interface**: Access samples by name with iteration support * **Batch operations**: Apply operations across all samples

Example

Multi-sample fragment analysis:

```
from pyfraglib.fragment import Fragment, FragmentCollection

# Load multiple samples
```

(continues on next page)

(continued from previous page)

```

bam_files = ["sample1.bam", "sample2.bam", "sample3.bam"]
collection = Fragment.from_bams(bam_files, None)

# Iterate through samples
for sample_name, fragments in collection:
    print(f"Sample {sample_name}: {len(fragments)} fragments")
    print(f"Bogus: {fragments.count_bogus_fragments()}")
    print(f"Mutated: {fragments.count_mutated_fragments()}")

# Batch save to .frag files
collection.to_frag_files("output/")

# Access sample names
sample_names = collection.record_names()
print(f"Loaded {len(sample_names)} samples: {sample_names}")

```

⚠ Warning

Large collections can consume significant memory. Consider using [Fragment.bams_to_frags\(\)](#) for memory-efficient batch processing or the Nextflow pipeline.

➡ See also

- [Fragment.from_bams\(\)](#) - Create collections from multiple BAMs
- [Fragment.bams_to_frags\(\)](#) - Memory-efficient batch processing
- [FragmentList](#) - Individual sample fragment collections

`__init__()` → `None`

`append(name: str, fragments: FragmentList)` → `None`

Parameters

- `name` ([str](#))
- `fragments` ([FragmentList](#))

Return type

`None`

`record_names()` → `list[str]`

Return type

`list[str]`

`to_frag_files(out_dir: str)` → `None`

Parameters

`out_dir` ([str](#))

Return type

`None`

4.3.3 pyfraglib.fragfile

Fragment File I/O and Serialization

This module provides efficient I/O operations for fragment data using a columnar on-disk format. The `FragFile` class enables streaming access to serialized fragment data stored in Apache Parquet files.

File Format

Fragment files (`.frag`) use Apache Parquet (via `pyarrow`) with one row per fragment and the following columns:

Column	Arrow dtype	Description
<code>start_pos</code>	<code>int64</code>	First aligned position (0-based).
<code>end_pos</code>	<code>int64</code>	One-past-the-last aligned position.
<code>length</code>	<code>int64</code>	Fragment length in base pairs.
<code>chrom</code>	<code>string</code>	Contig name (dictionary-encoded on disk).
<code>is_forward</code>	<code>bool</code>	Strand / orientation flag.
<code>end5p</code>	<code>string</code>	5' end motif (up to <code>MAX_KMER_LEN</code> nt).
<code>end3p</code>	<code>string</code>	3' end motif (up to <code>MAX_KMER_LEN</code> nt).
<code>is_bogus</code>	<code>bool</code>	Quality flag; True marks excluded fragments.
<code>is_mutated</code>	<code>bool</code>	Mutation-carrying flag; nullable.

Example Usage

Basic fragment file operations:

```
from pyfraglib.fragfile import FragFile
from pyfraglib.fragment import Fragment

# Create fragments and save to file
fragments = Fragment.from_bam("sample.bam", "variants.vcf")
fragments.to_frag_file("sample", "output/")

# Load fragments from file (streaming)
with FragFile("output/sample.frag") as fragfile:
    for fragment in fragfile:
        print(f"{fragment.chrom}:{fragment.start_pos}-{fragment.end_pos}")

# Load all fragments into memory
with FragFile("output/sample.frag") as fragfile:
    all_fragments = fragfile.get_fragment_list()
    print(f"Loaded {len(all_fragments)} fragments")
```

License

This file is part of `pyfraglib`, a software suite to calculate fragmentomics features from cfDNA and perform downstream analyses.

Copyright (C) 2026 Daniel Schütte, daniel.schuette@iccb-cologne.org

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without

even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program. If not, see <<https://www.gnu.org/licenses/>>.

Functions

<code>fragment_from_row(start_pos, end_pos, ...)</code>	Reconstruct a <code>pyfraglib.fragment.Fragment</code> from a row of raw field values read from disk.
---	---

Classes

<code>FragFile(path)</code>	Reader for fragment files (. frag) stored as Apache Parquet.
-----------------------------	--

`pyfraglib.fragfile.fragment_from_row(start_pos: int, end_pos: int, length: int, chrom: str, is_forward: bool, end5p: str, end3p: str, is_bogus: bool, is_mutated: bool | None) → Fragment`

Reconstruct a `pyfraglib.fragment.Fragment` from a row of raw field values read from disk.

Bypasses `Fragment.__init__` (which expects `pysam.AlignedSegment` objects) by using `Fragment.__new__` followed by explicit attribute assignment. `Fragment` uses `@dataclass(slots=True)`, so only the declared fields can be set; unknown fields raise `AttributeError`.

Parameters

- **start_pos** (`int`) – First aligned position (0-based).
- **end_pos** (`int`) – One-past-the-last aligned position.
- **length** (`int`) – Fragment length in base pairs.
- **chrom** (`str`) – Contig name.
- **is_forward** (`bool`) – Strand / orientation flag.
- **end5p** (`str`) – 5' end motif.
- **end3p** (`str`) – 3' end motif.
- **is_bogus** (`bool`) – Quality flag.
- **is_mutated** (`bool` | `None`) – Mutation flag (may be `None`).

Returns

A fully populated `Fragment` instance.

Return type

`Fragment`

`class pyfraglib.fragfile.FragFile(path: str) → None`

Reader for fragment files (. frag) stored as Apache Parquet.

Provides streaming and bulk access to serialized fragment data with context-manager semantics for automatic cleanup. The underlying storage is a `pyarrow.parquet.ParquetFile` and reads go through `pyarrow.parquet.ParquetFile.iter_batches()` for memory-efficient iteration.

Example

Basic usage with context manager:

```
from pyfraglib.fragfile import FragFile

# Streaming iteration (memory efficient)
with FragFile("sample.frag") as fragfile:
    for fragment in fragfile:
        print(f"{fragment.chrom}:{fragment.start_pos}")

# Bulk loading for analysis
with FragFile("sample.frag") as fragfile:
    fragments = fragfile.get_fragment_list()
    print(f"Loaded {len(fragments)} fragments")
```

➡ See also

- [pyfraglib.fragment.FragmentList.to_frag_file\(\)](#) - Writing fragment files.
- [pyfraglib.fragment.Fragment](#) - Individual fragment objects.
- [pyfraglib.fragment.FragmentList](#) - Fragment collections.
- FRAG_SCHEMA - On-disk column schema.

Parameters

path ([str](#))

ITER_BATCH_SIZE: [int](#) = 65536

Batch size used by `__iter__()` when pulling rows from Parquet.

__init__ (*path:* [str](#)) → [None](#)

Initialize a `FragFile` reader for the specified fragment file.

Opens the Parquet file lazily — metadata is read but fragment rows are not loaded until iteration or bulk loading is invoked.

Parameters

path ([str](#)) – Path to a `.frag` Parquet file produced by `FragmentList.to_frag_file()`.

Raises

- [FileNotFoundError](#) – If the specified file does not exist.
- [pyarrow.lib.ArrowInvalid](#) – If the file is not a valid Parquet file or does not carry the expected fragment columns.
- [PermissionError](#) – If the file cannot be opened due to permissions.

Note

- Column presence is validated on open; extra columns beyond the fragment schema are silently ignored.
- Use a context manager for automatic cleanup.

get_fragment_list() → *FragmentList*

Load all fragments from the file into a *FragmentList*.

Reads the entire Parquet table in one shot (more efficient than streaming for bulk loads) and loads each row as a *Fragment*. Use `__iter__()` for very large files.

Returns

A *FragmentList* containing every fragment from the file.

Return type

FragmentList

Raises

- **ValueError** – If called after *close()*.
- **MemoryError** – If the file is too large to fit in memory.

 Warning

This method loads all fragments into memory simultaneously. Prefer streaming iteration for memory-constrained environments.

 See also

- `__iter__()` - Memory-efficient streaming iteration.
- *pyfraglib.fragment.FragmentList* - Fragment collection class.

close() → *None*

Close the fragment file and release system resources.

Drops the underlying `pyarrow.parquet.ParquetFile` handle. Called automatically by the context manager and destructor, but may be called manually. Subsequent reads raise **ValueError**.

Return type

None

property closed: *bool*

Whether *close()* has been invoked on this file.

4.3.4 pyfraglib.math

Common math functions

This module provides mathematical functions used throughout the code base.

License

This file is part of *pyfraglib*, a software suite to calculate fragmentomics features from cfDNA and perform downstream analyses.

Copyright (C) 2025 Daniel Schütte, daniel.schuette@iccb-cologne.org

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without

even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program. If not, see <<https://www.gnu.org/licenses/>>.

Functions

<code>find_skew_normal_mean_from_mode(target_mode, ...)</code>	Find the mean (location parameter) of a skew normal distribution given the desired mode, skewness parameter alpha, and scale (not variance!).
<code>fit_gmm(data, config_filepath)</code>	Fit a GMM of n 1D Gaussians with bounds on the free parameters.
<code>gaussian_mixture(params, n, data)</code>	The ordering of params is as follows (' n ' = # of Gaussians): $\mu_1, \dots, \mu_n, \text{std}_1, \dots$
<code>goodness_of_fit_stats(data, params, n)</code>	
<code>hessian(params, _n, _data)</code>	
<code>jensen_shannon_divergence(p, q)</code>	
<code>mixture_cdf(x, params, n)</code>	
<code>mixture_cdf_wrapper(data, params, n)</code>	
<code>negative_log_likelihood(params, n, data, ...)</code>	params is as described above.
<code>plot_gmm(data, num_gaussians, params, ...)</code>	Given n means and standard deviations, we plot a histogram of data and overlay n normals.
<code>read_gmm_config(config_filepath)</code>	We do some basic validation, but don't exhaust possible error conditions in our sanity checks.

`pyfraglib.math.find_skew_normal_mean_from_mode(target_mode: float, skewness: float, variance: float) → tuple[float, float]`

Find the mean (location parameter) of a skew normal distribution given the desired mode, skewness parameter alpha, and scale (not variance!).

Parameters

- **target_mode** (`float`) – Desired mode of the distribution.
- **skewness** (`float`) – Skewness parameter (shape parameter, sometimes called alpha).
- **variance** (`float`) – Desired variance of the distribution.

Returns

(`location_param`, `scale_param`) where `location_param` is the location parameter found and `scale_param` is the scale parameter used.

Return type

`tuple[float, float]`

`pyfraglib.math.gaussian_mixture(params: list[float], n: int, data: ndarray[tuple[Any, ...], dtype[float64]]) → ndarray[tuple[Any, ...], dtype[float64]]`

The ordering of params is as follows (' n ' = # of Gaussians): $\mu_1, \dots, \mu_n, \text{std}_1, \dots, \text{std}_n, \pi_1, \dots, \pi_n$. The caller is responsible for ensuring that `sum(pi_i) == 1.0`, and that params holds exactly $n*3$ parameters.

Parameters

- **params** (`list[float]`)
- **n** (`int`)
- **data** (`ndarray[tuple[Any, ...], dtype[double]]`)

Return type`ndarray[tuple[Any, ...], dtype[double]]``pyfraglib.math.mixture_cdf(x: float, params: list[float], n: int) → float`**Parameters**

- `x` (`float`)
- `params` (`list[float]`)
- `n` (`int`)

Return type`float``pyfraglib.math.mixture_cdf_wrapper(data: list[float], params: list[float], n: int) → ndarray[tuple[Any, ...], dtype[float64]]`**Parameters**

- `data` (`list[float]`)
- `params` (`list[float]`)
- `n` (`int`)

Return type`ndarray[tuple[Any, ...], dtype[double]]``pyfraglib.math.negative_log_likelihood(params: list[float], n: int, data: ndarray[tuple[Any, ...], dtype[float64]], norm_const: float) → float`

`params` is as described above. It's important to note that we normalize the NLL to avoid numerical instabilities associated with very large values. The normalization factor must be provided upon every iteration, so we must be careful to always use the same float.

Parameters

- `params` (`list[float]`)
- `n` (`int`)
- `data` (`ndarray[tuple[Any, ...], dtype[double]]`)
- `norm_const` (`float`)

Return type`float``pyfraglib.math.read_gmm_config(config_filepath: str) → tuple[int, float, list[float], list[tuple[float, float]]]`

We do some basic validation, but don't exhaust possible error conditions in our sanity checks.

Parameters`config_filepath` (`str`)**Return type**`tuple[int, float, list[float], list[tuple[float, float]]]``pyfraglib.math.hessian(params: list[float], _n: int, _data: list[float]) → ndarray[tuple[Any, ...], dtype[float64]]`**Parameters**

- `params` (`list[float]`)
- `_n` (`int`)

- `_data` (`list[float]`)

Return type`ndarray[tuple[Any, ...], dtype[double]]`

`pyfraglib.math.fit_gmm(data: ndarray[tuple[Any, ...], dtype[float64]], config_filepath: str) → tuple[object, int, list[float], ndarray[tuple[Any, ...], dtype[float64]]]`

Fit a GMM of `n` 1D Gaussians with bounds on the free parameters. The bounds are read from a configuration file. Optimized parameters are returned and ordered as follows: `[m_1, m_2, ..., std_1, std_2, ..., pi_1, pi_2, ...]`.

Parameters

- `data` (`ndarray[tuple[Any, ...], dtype[double]]`)
- `config_filepath` (`str`)

Return type`tuple[object, int, list[float], ndarray[tuple[Any, ...], dtype[double]]]`

`pyfraglib.math.jensen_shannon_divergence(p: ndarray[tuple[Any, ...], dtype[float64]], q: ndarray[tuple[Any, ...], dtype[float64]]) → float`

Parameters

- `p` (`ndarray[tuple[Any, ...], dtype[double]]`)
- `q` (`ndarray[tuple[Any, ...], dtype[double]]`)

Return type`float`

`pyfraglib.math.goodness_of_fit_stats(data: ndarray[tuple[Any, ...], dtype[float64]], params: list[float], n: int) → dict[str, object]`

Parameters

- `data` (`ndarray[tuple[Any, ...], dtype[double]]`)
- `params` (`list[float]`)
- `n` (`int`)

Return type`dict[str, object]`

`pyfraglib.math.plot_gmm(data: ndarray[tuple[Any, ...], dtype[float64]], num_gaussians: int, params: list[float], out_dir: str, name: str) → None`

Given `n` means and standard deviations, we plot a histogram of `data` and overlay `n` normals. The parameters for the normals will most likely come from a GMM, that's why the function is named like this. `name` and `out_dir` are concatenated into a destination filepath.

Parameters

- `data` (`ndarray[tuple[Any, ...], dtype[double]]`)
- `num_gaussians` (`int`)
- `params` (`list[float]`)
- `out_dir` (`str`)
- `name` (`str`)

Return type

None

4.3.5 pyfraglib.stats

Fragment Statistics and Visualization

This module provides overview statistics and visualization for cfDNA fragment lists.

Key Features

- **Fragment Distribution Analysis:** Chromosome-wise fragment counts with mutation status visualization
- **End Motif Analysis:** k-mer frequency analysis for 5' and 3' fragment ends
- **Quality Metrics:** Statistical summaries including bogus and mutated fragment counts
- **CSV Export Functions:** Export fragment length distributions and end motif counts to CSV format for external analysis

License

This file is part of pyfraglib, a software suite to calculate fragmentomics features from cfDNA and perform downstream analyses.

Copyright (C) 2025 Daniel Schütte, daniel.schuette@iccb-cologne.org

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program. If not, see <<https://www.gnu.org/licenses/>>.

Functions

<code>end_motifs_barplot</code> (fragments, out_dir, name, ...)	Generate color-coded bar plots for fragment end motif frequencies.
<code>export_end_motifs_csv</code> (fragments, out_dir, ...)	Export fragment end motif counts to CSV file.
<code>export_length_distribution_csv</code> (fragments, ...)	Export fragment length distribution to CSV file.
<code>fragments_per_chromosome_barplot</code> (fragments, ...)	Generate chromosome-wise fragment distribution plots with mutation status.
<code>log_stats</code> (fragments, logger, out_dir, name)	Generate overview fragment statistics with logging and JSON export.

`pyfraglib.stats.fragments_per_chromosome_barplot`(*fragments*: `FragmentList`, *out_dir*: `str`, *name*: `str`) → None

Generate chromosome-wise fragment distribution plots with mutation status.

Creates a stacked bar chart showing fragment counts per chromosome, with separate visualization of mutated versus wildtype fragments. This provides an overview of fragment distribution across chromosomes and helps identify chromosomal bias in mutation patterns.

Parameters

- **fragments** (`FragmentList`) – Collection of fragments to analyze. Bogus fragments are automatically excluded from the analysis.

- **out_dir** (*str*) – Output directory path where the PNG file will be saved. Directory must exist and be writable.
- **name** (*str*) – Sample identifier used for plot title and output filename. Used to generate informative plot titles and unique output filenames.

Return type*None*

`pyfraglib.stats.end_motifs_barplot`(*fragments*: *FragmentList*, *out_dir*: *str*, *name*: *str*, *kmer_len*: *int*) → *None*

Generate color-coded bar plots for fragment end motif frequencies.

Creates separate bar plots for 5' and 3' end motif distributions, with base-specific color coding to visualize nuclease cleavage preferences. These plots reveal tissue-specific fragmentation patterns and cleavage site preferences.

Parameters

- **fragments** (*FragmentList*) – Fragment collection for motif analysis. Bogus fragments are automatically excluded.
- **out_dir** (*str*) – Output directory for PNG files. Must exist and be writable.
- **name** (*str*) – Sample identifier for plot titles and filenames.
- **kmer_len** (*int*) – Length of k-mer motifs to analyze. Typically 3 or 4 nucleotides. Determines the resolution of motif analysis.

Return type*None***Returns**

- {name}_k{kmer_len}_5p_frag_end_motifs.png - 5' end motifs
- {name}_k{kmer_len}_3p_frag_end_motifs.png - 3' end motifs

`pyfraglib.stats.log_stats`(*fragments*: *FragmentList*, *logger*: *Logger*, *out_dir*: *str*, *name*: *str*) → *None*

Generate overview fragment statistics with logging and JSON export.

Calculates quality metrics for a fragment collection, logs them to the console and exports structured data to JSON format for downstream analysis and record keeping. The JSON file contains:

- **number_of_fragments**: Total fragment count
- **number_of_bogus_fragments**: Low-quality fragment count
- **number_of_mutated_fragments**: Mutation-carrying fragment count

Parameters

- **fragments** (*FragmentList*) – Fragment collection to analyze. All fragments are included in statistics, with separate counts for different quality categories.
- **logger** (*Logger*) – Logger instance for console output. Used to report statistics.
- **out_dir** (*str*) – Output directory for JSON file. Must exist and be writable.
- **name** (*str*) – Sample identifier used for JSON filename and log messages.

Return type*None*

`pyfraglib.stats.export_length_distribution_csv`(*fragments*: [FragmentList](#), *out_dir*: [str](#), *name*: [str](#)) → [None](#)

Export fragment length distribution to CSV file.

Creates a CSV file containing fragment length counts for statistical analysis and visualization in external tools. Each row represents a unique fragment length with its corresponding count. The column names are as follows:

- `fragment_length`: Fragment length in base pairs (sorted ascending)
- `count`: Number of fragments with that specific length

Parameters

- **`fragments`** ([FragmentList](#)) – Fragment collection to analyze. Bogus fragments are automatically excluded.
- **`out_dir`** ([str](#)) – Output directory path where the CSV file will be saved. Directory must exist and be writable.
- **`name`** ([str](#)) – Sample identifier used for the output filename. Used to generate unique CSV filenames for each sample.

Return type

[None](#)

`pyfraglib.stats.export_end_motifs_csv`(*fragments*: [FragmentList](#), *out_dir*: [str](#), *name*: [str](#), *kmer_len*: [int](#)) → [None](#)

Export fragment end motif counts to CSV file.

Creates a CSV file containing 5' and 3' end motif frequencies for downstream analysis and visualization in external tools. Each row represents a unique k-mer motif with counts for both fragment ends. The column names are as follows:

- `motif_5p`: 5' end k-mer sequence (e.g., “AAA”, “AAC”, etc.)
- `count_5p`: Number of fragments with this 5' end motif
- `motif_3p`: 3' end k-mer sequence (e.g., “AAA”, “AAC”, etc.)
- `count_3p`: Number of fragments with this 3' end motif

Parameters

- **`fragments`** ([FragmentList](#)) – Fragment collection for motif analysis. Bogus fragments are automatically excluded.
- **`out_dir`** ([str](#)) – Output directory path where the CSV file will be saved. Directory must exist and be writable.
- **`name`** ([str](#)) – Sample identifier used for the output filename. Used to generate unique CSV filenames for each sample.
- **`kmer_len`** ([int](#)) – Length of k-mer motifs to analyze. Typically 3 or 4 nucleotides.

Return type

[None](#)

4.3.6 pyfraglib.lengths

Fragment length analysis and visualization

This module provides functions for analyzing and visualizing fragment length distributions. It includes kernel density plots to compare wildtype and mutated fragments, and code for fitting Gaussian mixture models (GMMs) to fragment length data. For cohort-wide analysis of fragment length profiles using non-negative matrix factorization (NMF), we are providing a separate script in `scripts/`.

License

This file is part of `pyfraglib`, a software suite to calculate fragmentomics features from cfDNA and perform downstream analyses.

Copyright (C) 2025 Daniel Schütte, daniel.schuette@iccb-cologne.org

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program. If not, see <<https://www.gnu.org/licenses/>>.

Functions

<code>fragment_length_gmm</code> (fragments, ...)	Fit Gaussian mixture models to fragment length distributions.
<code>fragment_length_plot</code> (fragments, out_dir, name)	Create kernel density estimation plots for fragment length distributions.
<code>write_gmm_params</code> (n, params, obj_val, conv, ...)	Export Gaussian mixture model parameters and statistics to JSON.

`pyfraglib.lengths.fragment_length_plot`(fragments: `FragmentList`, out_dir: `str`, name: `str`, kde_bandwidth: `float` = 0.3) → `None`

Create kernel density estimation plots for fragment length distributions.

Generates exploratory plots comparing fragment length distributions between wildtype and mutated fragments using kernel density estimation. The plot helps visualize differences in fragmentation patterns between normal and variant-carrying cfDNA fragments.

Parameters

- **fragments** (`FragmentList`) – List of fragments to analyze. Should contain both wildtype and mutated fragments for meaningful comparison.
- **out_dir** (`str`) – Output directory path where the plot will be saved. Directory will be created if it doesn't exist.
- **name** (`str`) – Base name for the output file. The plot will be saved as `{name}_frags_len_kde.png`.
- **kde_bandwidth** (`float`, default: 0.3) – Bandwidth parameter for the KDE plot.

Return type

`None`

Returns

None. The function saves a PNG plot to the specified output directory.

Note

- Bogus fragments (quality-filtered) are automatically excluded
- Includes sample sizes in the legend

Example

Create length distribution plots for a sample:

```
fragments = Fragment.from_bam("sample.bam", "variants.vcf")
fragment_length_plot(fragments, "plots/", "patient_001")
# Creates: plots/patient_001_frags_len_kde.png
```

`pyfraglib.lengths.fragment_length_gmm(fragments: FragmentList, config_filepath: str, out_dir: str, name: str) → None`

Fit Gaussian mixture models to fragment length distributions.

Performs analysis of fragment length data by fitting Gaussian mixture models (GMMs) with a configurable number of components. The function automatically:

- Fits GMM parameters using maximum likelihood estimation
- Validates model convergence and goodness-of-fit
- Generates diagnostic plots showing observed vs fitted distributions
- Exports results as JSON, including parameters and statistics

Parameters

- **fragments** ([FragmentList](#)) – Fragment list to analyze. All fragments are used except those marked as bogus (quality-filtered).
- **config_filepath** ([str](#)) – Path to JSON configuration file specifying GMM parameters. Must contain number of components and initial values.
- **out_dir** ([str](#)) – Output directory for results. Multiple files will be created including plots and parameter files.
- **name** ([str](#)) – Base name for output files. Used to generate unique filenames for plots and parameter exports.

Return type

[None](#)

Returns

None. Results are saved to multiple output files in the specified directory.

Raises

[SystemExit](#) – If GMM fitting fails due to non-convergence or numerical issues. This indicates problematic data or inappropriate model specification.

Note

- Requires a valid GMM configuration file (see `configs/` directory)
- May not converge for poor data

Example

Fit a 3-component GMM to fragment lengths:

```

fragments = Fragment.from_bam("sample.bam")
fragment_length_gmm(
    fragments,
    "configs/gmm_3.json",
    "analysis/",
    "sample_001"
)
# Creates: analysis/sample_001_gmm_frags_len.json
#          analysis/sample_001_gmm_plot.png

```

➔ See also

- [`pyfraglib.math.fit\_gmm\(\)`](#) - Core GMM fitting algorithm
- [`write\_gmm\_params\(\)`](#) - Parameter export functionality

`pyfraglib.lengths.write_gmm_params(n: int, params: list\[float\], obj_val: float, conv: bool, goodness_of_fit: dict\[str, object\], out_dir: str, name: str) → None`

Export Gaussian mixture model parameters and statistics to JSON.

Serializes comprehensive GMM fitting results to a structured JSON file for downstream analysis. The output includes fitted parameters and goodness-of-fit statistics.

Parameters

- **n** ([int](#)) – Number of Gaussian components in the fitted model.
- **params** ([list\[float\]](#)) – Flattened array of fitted parameters in the order: [means, standard_deviations, mixing_weights]. Each component contributes one value to each parameter type.
- **obj_val** ([float](#)) – Final objective function value from the optimization. Lower values indicate better fits for maximum likelihood estimation.
- **conv** ([bool](#)) – Boolean indicating whether the optimization algorithm converged successfully. False values suggest problematic fits.
- **goodness_of_fit** ([dict\[str, object\]](#)) – Dictionary containing statistical measures of model quality, including Kolmogorov-Smirnov tests and Jensen-Shannon divergence.
- **out_dir** ([str](#)) – Output directory path where the JSON file will be saved.
- **name** ([str](#)) – Base filename. The output will be saved as {name}_gmm_frags_len.json.

Return type

[None](#)

Returns

None. Results are written to a JSON file in the specified directory. The output JSON contains:

```

{
    "number_of_gaussians": 3,
    "objective_value": -12345.67,

```

(continues on next page)

(continued from previous page)

```

    "converged": true,
    "estimated_means": [167.0, 120.0, 200.0],
    "estimated_stds": [25.0, 15.0, 30.0],
    "estimated_pis": [0.6, 0.3, 0.1],
    "ks_statistic": 0.05,
    "ks_p_value": 0.8,
    ...
}

```

4.3.7 pyfraglib.scores

Fragmentomics Scores

This module provides algorithms for calculating fragmentomics scores from cfDNA fragment data and visualizing them along (selected parts of) the genome.

Key Algorithms

- **Windowed Protection Score (WPS):** Measures nucleosome positioning and chromatin accessibility
- **Smoothed WPS:** Coverage-normalised, detrended, Savitzky-Golay-smoothed WPS suitable for between-sample comparisons and peak detection
- **Per-region WPS metrics:** Peak count, spacing, prominence, width, and 170 bp spectral power derived from the smoothed signal
- **Motif Diversity:** Quantifies diversity of fragment end motifs using entropy measures

Windowed Protection Score (WPS)

The WPS algorithm measures the degree to which genomic regions are protected from nuclease digestion, providing insights into nucleosome positioning and chromatin accessibility. Please refer to the library documentation for a detailed explanation of how the WPS is calculated and smoothed.

Motif Diversity

Fragment end motifs reflect nuclease cleavage preferences and can reveal tissue-specific patterns. We provide functions to calculate diversity indices for fragment end motifs (i.e. Shannon entropy and Simpson index).

Example Usage

Calculate WPS for genomic regions:

```

from pyfraglib.scores import windowed_protection_score
from pyfraglib.fragment import Fragment

# Load fragments
fragments = Fragment.from_bam("sample.bam", "variants.vcf")

# Calculate WPS for BED regions
wps_scores = windowed_protection_score(fragments, "regions.bed")

# Access results

```

(continues on next page)

(continued from previous page)

```
for region, score in wps_scores.items():
    print(f"Region {region}: WPS = {score:.3f}")
```

Calculate motif diversity:

```
from pyfraglib.scores import motif_diversity

# Calculate Shannon entropy for 4-mer end motifs
shannon_div = motif_diversity(fragments, kmer_len=4, index="shannon")
print(f"Shannon entropy: {shannon_div:.4f}")

# Calculate Simpson index
simpson_div = motif_diversity(fragments, kmer_len=4, index="simpson")
print(f"Simpson index: {simpson_div:.4f}")
```

Visualize scores:

```
from pyfraglib.scores import score_line_plot

# Create line plot of WPS scores
score_line_plot(wps_scores, "output/", "sample_wps",
                xlabel="Genomic Position", ylabel="WPS Score",
                title="Windowed Protection Score")
```

Performance Considerations

- Consider subsampling fragments for initial exploratory analysis
- WPS calculation is memory-intensive for large genomic regions

Functions

- `windowed_protection_score()`: WPS implementation
- `smooth_wps()`: Coverage-normalise, detrend, and Savitzky-Golay-smooth a per-position WPS frame
- `wps_region_metrics()`: Per-region peak and spectral-power metrics from a smoothed WPS frame
- `wps_power_spectrum_plot()`: Per-sample FFT diagnostic plot (mean magnitude spectrum across regions with the nucleosome-repeat band highlighted)
- `motif_diversity()`: Calculate diversity indices for fragment end motifs
- `score_line_plot()`: Generate line plots for fragmentomics scores

License

This file is part of pyfraglib, a software suite to calculate fragmentomics features from cfDNA and perform downstream analyses.

Copyright (C) 2026 Daniel Schütte, daniel.schuette@iccb-cologne.org

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU

General Public License for more details. You should have received a copy of the GNU General Public License along with this program. If not, see <<https://www.gnu.org/licenses/>>.

Module Attributes

<code>WPS_SMOOTH_WIN</code>	Default Savitzky-Golay window length (bp) for <code>smooth_wps()</code> .
<code>WPS_SMOOTH_POLY</code>	Default Savitzky-Golay polynomial order for <code>smooth_wps()</code> .
<code>WPS_DETREND_WIN</code>	Default rolling-median window (bp) used by <code>smooth_wps()</code> to remove long-wavelength drift before Savitzky-Golay smoothing.
<code>WPS_PEAK_MIN_DIST</code>	Default minimum distance between detected peaks (bp) in <code>wps_region_metrics()</code> .
<code>WPS_PROM_FRAC</code>	Default peak-prominence threshold as a fraction of the per-region IQR of the smoothed signal in <code>wps_region_metrics()</code> .
<code>WPS_NUCL_PERIOD</code>	Nucleosome repeat length (bp) used by <code>wps_region_metrics()</code> to target the spectral-power band $(1 / (\text{nucl_period} + 10))$ to $1 / (\text{nucl_period} - 10)$.

Functions

<code>chromosome_maps_to_df(chrom_map_wps, ..., ...)</code>	Internal utility function.
<code>create_chromosome_map([genome])</code>	Internal utility function.
<code>motif_diversity(fragments, name[, index])</code>	Calculate diversity indices for fragment end motifs.
<code>precalc_size(regions)</code>	Internal utility function.
<code>score_line_plot(df, name, out_dir[, score, ...])</code>	Generate line plots for fragmentomics scores across the genome.
<code>smooth_wps(df[, smooth_win, smooth_poly, ...])</code>	Coverage-normalise, detrend, and Savitzky-Golay-smooth a WPS signal.
<code>windowed_protection_score(fragments, regions)</code>	Calculate Windowed Protection Score (WPS) for genomic regions.
<code>windowed_protection_score_fast(fragments, ...)</code>	This fast WPS implementation replaces the previous pyfraglib WPS implementation.
<code>wps_power_spectrum_plot(df, name, out_dir[, ...])</code>	Per-sample diagnostic plot of the WPS magnitude spectrum.
<code>wps_region_metrics(df[, smooth_win, ...])</code>	Derive per-region peak and spectral-power metrics from a raw per-position WPS frame.

`pyfraglib.scores.WPS_SMOOTH_WIN: int = 21`

Default Savitzky-Golay window length (bp) for `smooth_wps()`.

`pyfraglib.scores.WPS_SMOOTH_POLY: int = 3`

Default Savitzky-Golay polynomial order for `smooth_wps()`.

`pyfraglib.scores.WPS_DETREND_WIN: int = 1001`

Default rolling-median window (bp) used by `smooth_wps()` to remove long-wavelength drift before Savitzky-Golay smoothing.

`pyfraglib.scores.WPS_PEAK_MIN_DIST: int = 120`

Default minimum distance between detected peaks (bp) in `wps_region_metrics()`.

`pyfraglib.scores.WPS_PROM_FRAC: float = 0.02`

Default peak-prominence threshold as a fraction of the per-region IQR of the smoothed signal in `wps_region_metrics()`.

`pyfraglib.scores.WPS_NUCL_PERIOD: int = 170`

Nucleosome repeat length (bp) used by `wps_region_metrics()` to target the spectral-power band $(1 / (\text{nucl_period} + 10))$ to $1 / (\text{nucl_period} - 10)$.

`pyfraglib.scores.motif_diversity(fragments: FragmentList, name: str, index: str = 'shannon') → tuple[float, float]`

Calculate diversity indices for fragment end motifs.

Parameters

- **fragments** (`FragmentList`) – List of fragments to analyze. Bogus fragments are automatically excluded from calculations.
- **name** (`str`) – Sample or analysis name for logging purposes. Used in log messages to identify which sample is being processed.
- **index** (`str`, default: 'shannon') – Diversity index to calculate. Options are:
 - "shannon": Shannon entropy (default)
 - "simpson": Simpson index

Returns

A tuple containing (5p_diversity, 3p_diversity). Returns (0.0, 0.0) if no valid fragments are found

Return type

`tuple[float, float]`

Raises

SystemExit – If an invalid diversity index is specified. Only “shannon” and “simpson” are supported.

Note

- Uses 4-mer sequences from fragment ends for analysis
- Automatically filters out motifs containing ‘N’ bases
- Zero counts are excluded to avoid `log(0)` issues in calculations
- Results are normalized and comparable across samples

See also

- `pyfraglib.core.shannon_entropy()` - Shannon entropy calculation
- `pyfraglib.core.simpson_index()` - Simpson index calculation
- `FragmentList.count_endmotifs` - Low-level motif counting

```
pyfraglib.scores.windowed_protection_score(fragments: FragmentList, regions: TabixFile, win_size: int
                                           = 120, genome: str = 'hg19') → DataFrame
```

Calculate Windowed Protection Score (WPS) for genomic regions.

Computes the Windowed Protection Score, which measures the degree to which genomic regions are protected from nuclease digestion. The WPS algorithm calculates protection as the difference between spanning and ending fragments within genomic windows. Higher WPS values indicate regions with regular nucleosome positioning, while lower values suggest accessible chromatin or irregular positioning.

Parameters

- **fragments** ([FragmentList](#)) – Collection of fragments to analyze. Bogus fragments are automatically excluded from WPS calculations.
- **regions** ([TabixFile](#)) – Indexed BED file containing genomic regions of interest. Must be opened with `pysam.TabixFile` and properly formatted.
- **win_size** ([int](#), default: 120) – Window size in base pairs for WPS calculation. Default is 120bp, which corresponds to nucleosome-sized protection. Must be positive.
- **genome** ([str](#), default: 'hg19') – Reference genome version for chromosome length validation. Supported values are “hg19” and “hg38”.

Returns

DataFrame containing WPS results with columns:

- **chrom**: Chromosome name
- **pos**: Genomic position
- **abs_pos**: Absolute position across the genome
- **wps**: Windowed Protection Score
- **depth**: Fragment coverage depth
- **spanning_frags**: Number of fragments spanning the window
- **ending_frags**: Number of fragments ending in the window
- **info**: Additional information from BED file

Return type

[DataFrame](#)

Note

- Results include both raw WPS scores and supporting coverage metrics

See also

- [windowed_protection_score_fast\(\)](#) - Optimized WPS implementation
- [score_line_plot\(\)](#) - Visualization of WPS results

```
pyfraglib.scores.windowed_protection_score_fast(fragments: FragmentList, regions: TabixFile,
                                                win_size: int = 120, genome: str = 'hg19') →
                                                DataFrame
```

This fast WPS implementation replaces the previous pyfraglib WPS implementation. We removed the `step_size` argument because this algorithm does not need that. It iterates over fragments instead of genomic regions like so:

```
> case 1 (fragment_size < window_size):
    [frag_start - win_size/2, frag_end + win_size/2] <- -1

> case 2 (fragment_size >= window_size):
    [frag_start - win_size/2, frag_start + win_size/2] <- -1
    [frag_start + win_size/2, frag_end - win_size/2] <- +1
    (frag_end - win_size/2, frag_end + win_size/2] <- -1
```

The implementation below tries to be extremely careful about interval definitions to not introduce 1-off errors. The arguments are identical to `windowed_protection_score()`. See there for more information.

Parameters

- **fragments** ([FragmentList](#))
- **regions** ([TabixFile](#))
- **win_size** ([int](#), default: 120)
- **genome** ([str](#), default: 'hg19')

Return type

[DataFrame](#)

`pyfraglib.scores.precalc_size(regions: TabixFile) → int`

Internal utility function.

Parameters

- **regions** ([TabixFile](#))

Return type

[int](#)

`pyfraglib.scores.create_chromosome_map(genome: str = 'hg19') → dict[str, ndarray[tuple[Any, ...], dtype[int64]]]`

Internal utility function.

Parameters

- **genome** ([str](#), default: 'hg19')

Return type

`dict[str, ndarray[tuple[Any, ...], dtype[int_]]]`

`pyfraglib.scores.chromosome_maps_to_df(chrom_map_wps: dict[str, ndarray[tuple[Any, ...], dtype[int64]]], chrom_map_depth: dict[str, ndarray[tuple[Any, ...], dtype[int64]]], chrom_map_span: dict[str, ndarray[tuple[Any, ...], dtype[int64]]], chrom_map_end: dict[str, ndarray[tuple[Any, ...], dtype[int64]]], regions: TabixFile, genome: str = 'hg19') → DataFrame`

Internal utility function.

Note

The input BED file must be sorted with respect to chromosomes. E.g. listing chr1 regions, chr3 regions, and chr2 regions in this order will produce incorrect results with respect to the absolute genome position!

Parameters

- **chrom_map_wps** (`dict[str, ndarray[tuple[Any, ...], dtype[int]]]`)
- **chrom_map_depth** (`dict[str, ndarray[tuple[Any, ...], dtype[int]]]`)
- **chrom_map_span** (`dict[str, ndarray[tuple[Any, ...], dtype[int]]]`)
- **chrom_map_end** (`dict[str, ndarray[tuple[Any, ...], dtype[int]]]`)
- **regions** (`TabixFile`)
- **genome** (`str`, default: 'hg19')

Return type

`DataFrame`

`pyfraglib.scores.smooth_wps(df: DataFrame, smooth_win: int = 21, smooth_poly: int = 3, detrend_win: int = 1001) → DataFrame`

Coverage-normalise, detrend, and Savitzky-Golay-smooth a WPS signal. Refer to the library documentation for technical details.

Note

This pipeline is applied independently per region (as identified by the `info` column of the input BED file), so a single call handles a multi-region WPS frame.

Parameters

- **df** (`DataFrame`) – Per-position WPS frame, as returned by `windowed_protection_score()`. Required columns are `pos`, `wps`, `depth`, and `info`.
- **smooth_win** (`int`, default: 21) – Savitzky-Golay window length in base pairs. Must be odd; automatically shrunk for regions shorter than `smooth_win`.
- **smooth_poly** (`int`, default: 3) – Savitzky-Golay polynomial order.
- **detrend_win** (`int`, default: 1001) – Rolling-median window in base pairs used to remove long-wavelength drift. Adapted per region to `min(detrend_win, max(101, (len(region) // 3) | 1))` so short regions are not swamped.

Returns

A copy of `df` with an additional `wps_smooth` column containing the coverage-normalised, de-trended, and smoothed signal. Rows in regions that are too short to fit the smoother receive `NaN`. The row order and index of `df` are preserved.

Return type

`DataFrame`

Example

Smooth a per-position WPS table produced by `windowed_protection_score()`:

```
from pyfraglib.scores import (
    windowed_protection_score, smooth_wps
)

wps_df = windowed_protection_score(
    fragments, regions, win_size=120
)
wps_df = smooth_wps(wps_df)
wps_df[["pos", "wps", "wps_smooth"]].head()
```

➡ See also

- `wps_region_metrics()` - Per-region summary metrics derived from the smoothed signal.
- `windowed_protection_score()` - Generate the input frame.

```
pyfraglib.scores.wps_region_metrics(df: DataFrame, smooth_win: int = 21, smooth_poly: int = 3,
    detrend_win: int = 1001, peak_min_dist: int = 120, prom_frac: float = 0.02, nucl_period: int = 170) → DataFrame
```

Derive per-region peak and spectral-power metrics from a raw per-position WPS frame.

Parameters

- **df** (`DataFrame`) – Per-position WPS frame, as returned by `windowed_protection_score()`. Required columns are `pos`, `wps`, `depth`, and `info`.
- **smooth_win** (`int`, default: 21) – Savitzky-Golay window length (bp). See `smooth_wps()` for behaviour under short regions.
- **smooth_poly** (`int`, default: 3) – Savitzky-Golay polynomial order.
- **detrend_win** (`int`, default: 1001) – Rolling-median window (bp) used to remove long-wavelength drift before smoothing.
- **peak_min_dist** (`int`, default: 120) – Minimum distance between detected peaks (bp).
- **prom_frac** (`float`, default: 0.02) – Peak prominence threshold expressed as a fraction of the per-region IQR of the smoothed signal.
- **nucl_period** (`int`, default: 170) – Nucleosome repeat length (bp). The spectral-power band spans $1 / (\text{nucl_period} + 10)$ to $1 / (\text{nucl_period} - 10)$.

Returns

One row per unique `info` value with the columns:

- `region`: The `info` label of the region.
- `n_peaks`: Number of detected peaks.
- `mean_peak_dist`: Mean of consecutive peak-to-peak spacings within the region (bp); NaN if fewer than two peaks.
- `median_peak_dist`: Median peak-to-peak spacing (bp).

- `median_prom`: Median peak prominence.
- `median_width`: Median peak width (bp).
- `power_170`: Mean FFT magnitude in the nucleosome-repeat band divided by the overall mean magnitude of the detrended signal.
- `n_positions`: Number of input positions for the region.

Return type`DataFrame`**Note**

- Regions shorter than $3 * \text{smooth_win}$ positions return NaN for every metric except `region` and `n_positions`.
- Cross-region spacings are filtered out by dropping consecutive peak distances greater than 5000 bp.
- The pre-smooth detrended signal drives `power_170` so that values remain comparable to classic depth-normalised WPS spectra.

Example

Compute per-region metrics from a raw WPS frame:

```
from pyfraglib.scores import (
    windowed_protection_score, wps_region_metrics
)

wps_df = windowed_protection_score(fragments, regions)
metrics = wps_region_metrics(wps_df)
metrics[["region", "n_peaks", "median_peak_dist",
        "power_170"]].head()
```

See also

- [`smooth\_wps\(\)`](#) - Produce the smoothed per-position column (for visualisation / export).
- [`windowed\_protection\_score\(\)`](#) - Generate the underlying WPS.

```
pyfraglib.scores.score_line_plot(df: DataFrame, name: str, out_dir: str, score: str = 'wps',
                                exclude_chroms: list[str] = ['Y', 'M'], region: tuple[int, int] = (0, 0),
                                genome: str = 'hg19', log_transform: bool = False,
                                plot_spanning_ending_frags: bool = False) → None
```

Generate line plots for fragmentomics scores across the genome.

Creates exploratory line plots showing fragmentomics scores across chromosomes or specific genomic regions. Supports multiple score types including WPS, fragment depth, and fragment ratios with customizable visualization options.

The function generates genome-wide plots or focused plots for single chromosome regions.

Parameters

- **df** ([DataFrame](#)) – DataFrame containing score data with required columns: `chrom`, `pos`, `abs_pos`, and the specified score column. Additional columns `spanning_frags` and `ending_frags` required if `plot_spanning_ending_frags=True`.
- **name** ([str](#)) – Sample or analysis name used for the plot title and output filename.
- **out_dir** ([str](#)) – Output directory path where the plot PNG file will be saved.
- **score** ([str](#), default: `'wps'`) – Score type to plot. Supported options:
 - `"wps"`: Windowed Protection Score (default)
 - `"depth"`: Fragment coverage depth
 - `"ratio_end_span"`: Ratio of ending to spanning fragments
 - `"ratio_span_total"`: Ratio of spanning to total fragments
- **exclude_chroms** ([list\[str\]](#), default: `['Y', 'M']`) – List of chromosomes to exclude from plotting. Default excludes Y and mitochondrial chromosomes.
- **region** ([tuple\[int, int\]](#), default: `(0, 0)`) – Genomic region to focus on as (start, end) coordinates. Only applied when plotting a single chromosome. Default `(0, 0)` plots the entire chromosome.
- **genome** ([str](#), default: `'hg19'`) – Reference genome version for chromosome definitions. Supported values are `"hg19"` and `"hg38"`.
- **log_transform** ([bool](#), default: `False`) – Whether to apply logarithmic transformation to y-axis. Useful for scores with wide dynamic ranges.
- **plot_spanning_ending_frags** ([bool](#), default: `False`) – Whether to overlay spanning and ending fragment counts on the plot. Only available for single-chromosome plots.

Return type[None](#)**Returns**

None. The function saves a PNG plot to the specified output directory with filename `{name}_{score}.png`.

Raises

[SystemExit](#) – If an unsupported score type is specified.

Note

- Automatically calculates ratio scores if not present in DataFrame

Example

Create WPS line plot across the genome:

```
import pandas as pd
from pyfraglib.scores import score_line_plot

# Assuming wps_results is a df from windowed_protection_score
score_line_plot(
    df=wps_results,
    name="sample_001",
    out_dir="plots/",
```

(continues on next page)

(continued from previous page)

```

score="wps",
exclude_chroms=["Y", "M"]
)
# Creates: plots/sample_001_wps.png

```

➔ See also

- [windowed_protection_score\(\)](#) - Generate WPS data for plotting

`pyfraglib.scores.wps_power_spectrum_plot(df: DataFrame, name: str, out_dir: str, smooth_win: int = 21, smooth_poly: int = 3, detrend_win: int = 1001, nucl_period: int = 170, max_region_overlay: int = 40) → None`

Per-sample diagnostic plot of the WPS magnitude spectrum.

Each region's spectrum is interpolated onto a common period grid so the per-region curves can be averaged. The mean curve is drawn in the foreground with individual region spectra faintly overlaid behind it. The nucleosome-repeat band `[nucl_period - 10, nucl_period + 10]` bp is shaded and the sample-median `power_170` ratio is shown in the title.

Parameters

- **df** (DataFrame) – Per-position WPS frame, as returned by [windowed_protection_score\(\)](#). Required columns are `pos`, `wps`, `depth`, and `info`.
- **name** (str) – Sample or analysis name used for the plot title and output filename.
- **out_dir** (str) – Output directory. The plot is written to `<out_dir>/<name>_wps_spectrum.png`.
- **smooth_win** (int, default: 21) – Unused in this routine today; kept for signature parity with [wps_region_metrics\(\)](#) so that callers can pass the same keyword arguments to both functions.
- **smooth_poly** (int, default: 3) – See `smooth_win`; unused.
- **detrend_win** (int, default: 1001) – Rolling-median window (bp) for the detrender, adapted per region as in [smooth_wps\(\)](#).
- **nucl_period** (int, default: 170) – Nucleosome repeat length (bp) used to position the highlighted band.
- **max_region_overlay** (int, default: 40) – Maximum number of individual region spectra to draw behind the mean curve. Randomly sampled if the region count exceeds this limit so the plot does not become a grey block.

Return type

None

Returns

None. The plot is written to disk.

Note

- Uses the pre-smooth detrended signal for the FFT so the spectrum shown is what `wps_region_metrics()` actually integrates to compute `power_170`.
- Regions shorter than `3 * smooth_win` positions are skipped.
- The y-axis is per-region normalised (each spectrum divided by its own overall mean magnitude) so the null expectation for a white-noise region is 1.0, matching the `power_170` ratio.

➡ See also

- `smooth_wps()` - Smoothing pipeline used upstream.
- `wps_region_metrics()` - Per-region summary metrics derived from the same pipeline.

4.3.8 pyfraglib.simulator

cfDNA Simulator

This module facilitates the simulation of biologically realistic cfDNA fragments.

License

This file is part of pyfraglib, a software suite to calculate fragmentomics features from cfDNA and perform downstream analyses.

Copyright (C) 2025 Daniel Schütte, daniel.schuette@iccb-cologne.org

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

```
class pyfraglib.simulator.FragmentSimulator(sequence_generator: SequenceContextGenerator,
                                           nucleosome_map: NucleosomeMap | None = None,
                                           chromatin_state: ChromatinState | None = None) → None
```

cfDNA fragment simulator.

This simulator integrates experimental observations to model cfDNA fragmentation:

1. **Nucleosome positioning:** Nucleosome maps with occupancy scores
2. **Chromatin accessibility:** Tissue-specific chromatin states
3. **Nuclease specificity:** Experimentally-determined preferences
4. **Sequence context:** Genomic sequences from reference FASTA
5. **TF binding protection:** Transcription factor binding site shielding

Main references are Lo et al. 2021 Science, Cristiano et al. 2019 Nature.

Parameters

- **sequence_generator** ([SequenceContextGenerator](#))
- **nucleosome_map** ([NucleosomeMap](#) | `None`, default: `None`)

- **chromatin_state** (*ChromatinState* | *None*, default: *None*)

MONO_NUC_MEAN: *Final[int]* = 167
MONO_NUC_STD: *Final[int]* = 10
MONO_FRACTION: *Final[float]* = 0.8
MONO_SKEW: *Final[float]* = -3.0
DI_NUC_MEAN: *Final[int]* = 334
DI_NUC_STD: *Final[int]* = 15
DI_FRACTION: *Final[float]* = 0.15
TRI_NUC_MEAN: *Final[int]* = 501
TRI_NUC_STD: *Final[int]* = 35
PERIODICITY_AMPLITUDE: *Final[float]* = 10.0
MIN_FRAGMENT_SIZE: *Final[int]* = 40
MAX_FRAGMENT_SIZE: *Final[int]* = 900

__init__(*sequence_generator*: *SequenceContextGenerator*, *nucleosome_map*: *NucleosomeMap* | *None* = *None*, *chromatin_state*: *ChromatinState* | *None* = *None*) → *None*

Initialize the fragment simulator.

Parameters

- **sequence_generator** (*SequenceContextGenerator*) – Sequence context generator (must be constructed with indexed FASTA file)
- **nucleosome_map** (*NucleosomeMap* | *None*, default: *None*) – Custom nucleosome positioning data. If *None*, generates default nucleosome map.
- **chromatin_state** (*ChromatinState* | *None*, default: *None*) – Chromatin accessibility information. If *None*, generates default chromatin state.

logger: *logging.Logger*

sequence_generator: *SequenceContextGenerator*

nucleosome_map: *NucleosomeMap*

chromatin_state: *ChromatinState*

static generate_nucleosome_map(*sequence_generator*: *SequenceContextGenerator*, *beta_alpha*: *float* = 4.0, *beta_beta*: *float* = 3.0) → *NucleosomeMap*

Generate a nucleosome map with customizable chromatin state.

This static method creates a nucleosome map using beta distribution parameters to control chromatin accessibility. Can be used by both the base simulator and tissue-specific simulators.

Parameters

- **sequence_generator** (*SequenceContextGenerator*) – Sequence generator to get chromosome information.
- **beta_alpha** (*float*, default: 4.0) – Alpha parameter for beta distribution. Higher values increase nucleosome occupancy (more compact chromatin).

- **beta_beta** ([float](#), default: 3.0) – Beta parameter for beta distribution. Higher values decrease nucleosome occupancy (more open chromatin).

Returns

Nucleosome map with specified chromatin characteristics.

Return type

[NucleosomeMap](#)

Notes

Common beta parameter combinations: - Open chromatin: (2, 5) - low occupancy, high accessibility - Normal chromatin: (4, 3) - balanced occupancy - Compact chromatin: (6, 2) - high occupancy, low accessibility

simulate_fragments(*chrom*: [str](#), *start*: [int](#), *end*: [int](#), *num_fragments*: [int](#), *tissue_type*: [str](#) = 'healthy', *nuclease_profile*: [NucleaseProfile](#) | [None](#) = None, *fragment_size_params*: [dict](#)[[str](#), [float](#)] | [None](#) = None) → [FragmentList](#)

Simulate cfDNA fragments from a genomic region with full biological realism.

Uses realistic nucleosome positioning, chromatin accessibility, and nuclease-specific cleavage preferences based on experimental data.

Parameters

- **chrom** ([str](#)) – Chromosome name
- **start** ([int](#)) – Start position
- **end** ([int](#)) – End position
- **num_fragments** ([int](#)) – Number of fragments to generate
- **tissue_type** ([str](#), default: 'healthy') – Source tissue type affecting chromatin state
- **nuclease_profile** ([NucleaseProfile](#) | [None](#), default: None) – Nuclease activity parameters
- **fragment_size_params** ([dict](#)[[str](#), [float](#)] | [None](#), default: None) – Custom size distribution parameters

Return type

[FragmentList](#)

Returns

FragmentList containing biologically realistic simulated fragments

class pyfraglib.simulator.**NucleaseProfile**(*dnase1_activity*: [float](#) = 1.0, *dnase1l3_activity*: [float](#) = 1.0, *dffb_activity*: [float](#) = 1.0, *dnase1_motif_preference*: [dict](#)[[str](#), [float](#)] | [None](#) = None, *dnase1l3_motif_preference*: [dict](#)[[str](#), [float](#)] | [None](#) = None, *dffb_motif_preference*: [dict](#)[[str](#), [float](#)] | [None](#) = None) → [None](#)

Nuclease activity and sequence preference parameters.

This class defines the activity levels and sequence preferences of 3 different nucleases involved in cfDNA fragmentation. The parameters are based on experimental literature and can be customized for different biological conditions or disease states.

Notes

Activity levels are relative and multiplicative:

- 1.0 represents normal physiological activity
- Values >1.0 increase nuclease activity
- Values <1.0 decrease nuclease activity
- 0.0 completely disables the nuclease

Motif preferences are multiplicative factors applied to base cleavage probability based on local sequence context. Higher values increase cleavage probability for sequences containing the motif.

Examples

```
>>> # Healthy individual profile
>>> healthy_profile = NucleaseProfile(
...     dnase1_activity=1.0,
...     dnase1l3_activity=1.2,
...     dffb_activity=0.1
... )
>>>
>>> # Cancer patient with increased apoptosis
>>> cancer_profile = NucleaseProfile(
...     dnase1_activity=1.1,
...     dnase1l3_activity=1.0,
...     dffb_activity=2.0
... )
```

Parameters

- **dnase1_activity** ([float](#), default: 1.0)
- **dnase1l3_activity** ([float](#), default: 1.0)
- **dffb_activity** ([float](#), default: 1.0)
- **dnase1_motif_preference** ([dict\[str, float\]](#) | [None](#), default: [None](#))
- **dnase1l3_motif_preference** ([dict\[str, float\]](#) | [None](#), default: [None](#))
- **dffb_motif_preference** ([dict\[str, float\]](#) | [None](#), default: [None](#))

dnase1_activity: [float](#) = 1.0

Relative activity level of DNase I. DNase I shows preference for AT-rich accessible regions.

dnase1l3_activity: [float](#) = 1.0

Relative activity level of DNase I Like 3 (DNase1L3). This is the primary nuclease responsible for cfDNA fragmentation in healthy individuals. Shows strong CC dinucleotide preference.

dffb_activity: [float](#) = 1.0

Relative activity level of the apoptotic nuclease DNA Fragmentation Factor B (DFFB). More active during cell death and shows preference for nucleosome linker regions.

dnase1_motif_preference: [dict\[str, float\]](#) | [None](#) = [None](#)

Sequence motif preferences for DNase I. If [None](#), uses default preferences based on literature (AT-rich sequences preferred).

dnase113_motif_preference: `dict[str, float] | None = None`

Sequence motif preferences for DNase1L3. If None, uses default preferences (strong CC dinucleotide preference).

dffb_motif_preference: `dict[str, float] | None = None`

Sequence motif preferences for DFFB. If None, uses default preferences (slight A preference).

__init__(*dnase1_activity*: `float = 1.0`, *dnase113_activity*: `float = 1.0`, *dffb_activity*: `float = 1.0`, *dnase1_motif_preference*: `dict[str, float] | None = None`, *dnase113_motif_preference*: `dict[str, float] | None = None`, *dffb_motif_preference*: `dict[str, float] | None = None`) $\rightarrow$ `None`

Parameters

- **dnase1_activity** (`float`, default: 1.0)
- **dnase113_activity** (`float`, default: 1.0)
- **dffb_activity** (`float`, default: 1.0)
- **dnase1_motif_preference** (`dict[str, float] | None`, default: None)
- **dnase113_motif_preference** (`dict[str, float] | None`, default: None)
- **dffb_motif_preference** (`dict[str, float] | None`, default: None)

class `pyfraglib.simulator.SequenceContextGenerator`(*fasta_path*: `str`) $\rightarrow$ `None`

Genomic sequence context provider for cfDNA fragment simulation.

This class interfaces with reference FASTA files to provide authentic genomic sequence context for fragment end motif generation. The class implements sequence caching and provides methods for extracting sequence context around genomic positions, enabling simulation of cfDNA fragments with somewhat realistic end motifs.

Notes

The FASTA file must be indexed with samtools faidx before use.

Examples

Direct usage:

```
seq_gen = SequenceContextGenerator("/path/to/reference.fasta")
context = seq_gen.generate_sequence_context("chr1", 1000000, 20)
print(f"Sequence context: {context}")
seq_gen.close()
```

Using as a context manager:

```
with SequenceContextGenerator("/path/to/reference.fasta") as seq_gen:
    context = seq_gen.generate_sequence_context("chr1", 1000000, 20)
```

Parameters

fasta_path (`str`)

__init__(*fasta_path*: `str`) $\rightarrow$ `None`

Initialize sequence generator with reference FASTA file.

Parameters

fasta_path (`str`) – Path to reference FASTA file (must be indexed)

generate_sequence_context(*chrom*: *str*, *position*: *int*, *length*: *int* = 20) → *str*

Extract genomic sequence context around a specified position.

This method retrieves genomic sequence from the reference FASTA file, centered around the specified position. It handles edge cases at chromosome boundaries by padding with 'N' nucleotides when necessary.

Parameters

- **chrom** (*str*) – Chromosome name (e.g., “chr1”, “1”, “X”). Must match chromosome naming convention in the FASTA file.
- **position** (*int*) – Genomic position (0-based coordinate system) around which to extract sequence context.
- **length** (*int*, default: 20) – Total length of sequence to extract, centered on the position. Must be positive.

Returns

DNA sequence string of specified length, uppercase nucleotides. Contains 'N' characters for undefined or unavailable regions.

Return type

str

Raises

Exception – If chromosome is not found in FASTA file or sequence extraction fails. In such cases, returns a string of 'N' characters as fallback.

Notes

The method extracts sequence symmetrically around the position: - Start position: position - length//2 - End position: start + length

Boundary handling: - If start < 0: sequence is extracted from position 0 - If end > chromosome length: sequence is padded with 'N'

Examples

```
>>> seq_gen = SequenceContextGenerator("reference.fasta")
>>> context = seq_gen.generate_sequence_context("chr1", 1000000, 20)
>>> print(f"Context: {context}") # e.g., "ATCGATCGATCGATCGATCG"
>>> len(context)
```

get_chromosome_length(*chrom*: *str*) → *int*

Retrieve chromosome length from the reference FASTA file.

Parameters

chrom (*str*) – Chromosome name to query. Must match naming convention in FASTA.

Returns

Length of the chromosome in base pairs. Returns 0 if chromosome is not found or an error occurs.

Return type

int

get_available_chromosomes() → *list*[*str*]

List all available chromosome references in the FASTA file.

Returns

List of chromosome names available in the FASTA file, in the order they appear in the file header.

Return type

`list[str]`

`close()` → `None`

Close the FASTA file handle and release resources.

Return type

`None`

class `pyfraglib.simulator.TissueMixtureSimulator`(*sequence_generator*: [SequenceContextGenerator](#),
***kwargs*: `dict[str, object]`) → `None`

cfDNA simulator for multi-tissue mixtures. This class extends the base `FragmentSimulator`.

Examples

Basic tissue mixture simulation:

```
>>> seq_gen = SequenceContextGenerator("/path/to/reference.fasta")
>>> simulator = TissueMixtureSimulator(seq_gen)
>>> fragments = simulator.simulate_tissue_mixture(
...     tissue_types=["hematopoietic", "tumor"],
...     tissue_fractions=[0.90, 0.10],
...     total_fragments=10000,
...     genomic_regions=[("chr1", 1000000, 1100000)]
... )
```

Cancer progression study:

```
>>> progression = simulator.simulate_cancer_progression(
...     normal_profile="hematopoietic",
...     tumor_fractions=[0.01, 0.05, 0.15],
...     time_points=["diagnosis", "3months", "6months"],
...     fragments_per_timepoint=20000,
...     genomic_regions=[("chr1", 1000000, 2000000)],
...     size_shift=-20.0, short_enrichment=0.25
... )
```

Parameters

- **sequence_generator** ([SequenceContextGenerator](#))
- **kwargs** (`dict[str, object]`)

__init__(*sequence_generator*: [SequenceContextGenerator](#), ***kwargs*: `dict[str, object]`) → `None`

Initialize the fragment simulator.

Parameters

- **sequence_generator** ([SequenceContextGenerator](#)) – Sequence context generator (must be constructed with indexed FASTA file)
- **nucleosome_map** – Custom nucleosome positioning data. If `None`, generates default nucleosome map.

- **chromatin_state** – Chromatin accessibility information. If None, generates default chromatin state.
- **kwargs** (`dict[str, object]`)

add_custom_tissue(*tissue_profile*: `TissueProfile`) → `None`

Add a custom tissue profile to the simulator's available profiles.

This method allows users to define custom tissue types with specific fragmentation characteristics beyond the predefined profiles.

Parameters

tissue_profile (`TissueProfile`) – Complete tissue profile definition including name, fragment size distribution, nucleosome spacing, chromatin openness, end motif preferences, and methylation level.

Return type

`None`

simulate_tissue_mixture(*tissue_types*: `list[str]`, *tissue_fractions*: `list[float]`, *total_fragments*: `int`, *genomic_regions*: `list[tuple[str, int, int]]`, *add_noise*: `bool` = `True`) → `FragmentList`

Generate cfDNA mixture from multiple tissue sources.

Parameters

- **tissue_types** (`list[str]`) – List of tissue type names to include in the mixture. Must be present in `tissue_profiles` (either predefined or custom).
- **tissue_fractions** (`list[float]`) – Fraction contribution from each tissue type. Must sum to 1.0 (within 0.001 tolerance). Order corresponds to `tissue_types`.
- **total_fragments** (`int`) – Total number of fragments to generate across all tissues. Individual tissue contributions are calculated proportionally.
- **genomic_regions** (`list[tuple[str, int, int]]`) – List of genomic regions (chromosome, start, end) to sample fragments from. Fragments are distributed across regions.
- **add_noise** (`bool`, default: `True`) – Whether to add realistic biological and technical noise:
 - Random fragment loss (~5%)
 - Size variation (± 2 bp standard deviation)
 - End repair artifacts (~2%)

Returns

Combined fragment list containing fragments from all tissues with tissue-specific characteristics. Fragments are shuffled to simulate realistic mixing.

Return type

`FragmentList`

Raises

SystemExit – If tissue fractions don't sum to 1.0 or if unknown tissue types are requested.

Examples

Cancer detection mixture (5% tumor fraction):

```
>>> fragments = simulator.simulate_tissue_mixture(
...     tissue_types=["hematopoietic", "tumor"],
...     tissue_fractions=[0.95, 0.05],
...     total_fragments=500000,
...     genomic_regions=[("chr1", 1000000, 1500000)],
```

(continues on next page)

(continued from previous page)

```
...     add_noise=True
... )
```

simulate_cancer_progression(*normal_profile*: *str*, *tumor_fractions*: *list[float]*, *time_points*: *list[str]*, *fragments_per_timepoint*: *int*, *genomic_regions*: *list[tuple[str, int, int]]*)
→ *dict[str, FragmentList]*

Model longitudinal cancer progression.

Parameters

- **normal_profile** (*str*) – Name of normal tissue profile to use as background (typically “hematopoietic” for blood-based cfDNA).
- **tumor_fractions** (*list[float]*) – Tumor-derived cfDNA fraction at each time point. Values should be in [0, 1] range and typically increase over time for progression studies.
- **time_points** (*list[str]*) – Labels for each time point (e.g., [“baseline”, “3months”]). Must have same length as *tumor_fractions*.
- **fragments_per_timepoint** (*int*) – Number of fragments to generate at each time point. Consistent across time points for comparative analysis.
- **genomic_regions** (*list[tuple[str, int, int]]*) – Genomic regions (chromosome, start, end) to sample fragments from. Same regions used for all time points.

Returns

Dictionary mapping time point labels to *FragmentList* objects. Each *FragmentList* contains the cfDNA profile for that time point with appropriate tumor fraction and cancer signatures.

Return type

dict[str, FragmentList]

Raises

SystemExit – If *tumor_fractions* and *time_points* have different lengths.

Examples

```
>>> progression = simulator.simulate_cancer_progression(
...     normal_profile="hematopoietic",
...     tumor_fractions=[0.01, 0.03, 0.08, 0.15],
...     time_points=["diagnosis", "1month", "3months", "6months"],
...     fragments_per_timepoint=25000,
...     genomic_regions=[("chr1", 1000000, 2000000)]
... )
```

4.3.9 pyfraglib.simulator.fragment_simulator

Biologically Realistic cfDNA Fragment Simulation

This module provides simulation capabilities for cell-free DNA (cfDNA) fragments based on experimentally observed fragmentation biology. The simulator tries to achieve realism by integrating multiple layers of biological complexity. It might be used e.g. for method validation or training data generation.

Key Features

- **Real genomic sequences:** Uses reference FASTA files to provide sequence context
- **Nucleosome positioning:** Models DNA protection by nucleosome through incorporating nucleosome maps
- **Chromatin accessibility:** Tissue-specific chromatin states and accessibility, DNA shielding by transcription factors
- **Nuclease specificity:** Experimentally-determined nuclease cleavage preferences
- **Fragment size distributions:** Adjustable size patterns with periodicity
- **End motif generation:** Sequence-context aware end motif simulation

Biological Foundation

Core parts of the simulator are based on the following assumptions:

1. Nuclease Activities:

- DNASE1L3: plasma nuclease, CC dinucleotide preference (Serpas et al. 2019 PNAS)
- DNASE1: endonuclease, AT-rich accessibility preference, predominantly produces T ends (Lazarovici et al. 2013 PNAS, Han et al. 2020 Am J Hum Genet.)
- DFFB: Nucleosome linker preference, produces A ends (Han et al. 2020 Am J Hum Genet.)

2. Fragment Size Patterns:

- Left-skewed mono-nucleosomal peak (~167 bp) with 10 bp periodicity
- Symmetric di-nucleosomal components (~334 bp) (reviewed in Lo et al. 2021 Science)

3. Chromatin Context:

- Nucleosome positioning affects cleavage accessibility
- Transcription factor binding site protection (reviewed in Lo et al. 2021 Science)

Example Usage

```
from pyfraglib.simulator.fragment_simulator import (
    FragmentSimulator, NucleaseProfile, SequenceContextGenerator
)

# Initialize sequence generator with reference genome
seq_gen = SequenceContextGenerator("/path/to/reference.fasta")

# Initialize simulator with sequence generator
simulator = FragmentSimulator(seq_gen)

# Define nuclease activity profile
nuclease_profile = NucleaseProfile(
    dnase1_activity=1.0,
    dnase1l3_activity=1.2,
    dffb_activity=0.5
)

# Simulate fragments from a genomic region
fragments = simulator.simulate_fragments(
```

(continues on next page)

(continued from previous page)

```

    chrom="1",
    start=10000000,
    end=11000000,
    num_fragments=10000,
    tissue_type="hematopoietic",
    nuclease_profile=nuclease_profile
)

# Process generated fragments
print(f"Generated {fragments.length()} cfDNA fragments")

# Clean up
seq_gen.close()

```

License

Copyright (C) 2025 Daniel Schütte, daniel.schuette@iccb-cologne.org

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

Classes

<i>ChromatinState</i> (open_regions, ...)	Represents tissue-specific chromatin accessibility and regulatory states.
<i>FragmentSimulator</i> (sequence_generator[, ...])	cfDNA fragment simulator.
<i>NucleaseProfile</i> ([dnase1_activity, ...])	Nuclease activity and sequence preference parameters.
<i>NucleosomeMap</i> (positions, occupancy)	Container for nucleosome positioning data across chromosomes.
<i>SequenceContextGenerator</i> (fasta_path)	Genomic sequence context provider for cfDNA fragment simulation.

class pyfraglib.simulator.fragment_simulator.**NucleosomeMap**(positions: [*dict*](#)[[*str*](#), [*ndarray*](#)[[*tuple*](#)[[*Any*](#), ...], [*dtype*](#)[[*int64*](#)]]], occupancy: [*dict*](#)[[*str*](#), [*ndarray*](#)[[*tuple*](#)[[*Any*](#), ...], [*dtype*](#)[[*float64*](#)]]]) → [*None*](#)

Container for nucleosome positioning data across chromosomes.

This class stores nucleosome positioning information, e.g. derived from experimental data. The positioning is used to model realistic nucleosome protection effects during cfDNA fragmentation.

Notes

Nucleosome occupancy scores affect the probability of DNA cleavage:

- High occupancy (>0.8): Strong protection, low cleavage probability
- Medium occupancy (0.3-0.8): Moderate protection

- Low occupancy (<0.3): Weak protection, higher cleavage probability

The positions array should be sorted in ascending order for efficient binary search operations during simulation.

Examples

```
>>> import numpy as np
>>> nuc_map = NucleosomeMap(
...     positions={"chr1": np.array([1000, 1185, 1370], dtype=np.int64)},
...     occupancy={"chr1": np.array([0.8, 0.6, 0.9], dtype=np.float64)}
... )
>>> print(f"Nucleosome at position {nuc_map.positions['chr1'][0]}")
Nucleosome at position 1000
```

Parameters

- **positions** (`dict[str, ndarray[tuple[Any, ...], dtype[int_]]]`)
- **occupancy** (`dict[str, ndarray[tuple[Any, ...], dtype[double]]]`)

positions: `dict[str, ndarray[tuple[Any, ...], dtype[int64]]]`

occupancy: `dict[str, ndarray[tuple[Any, ...], dtype[float64]]]`

`__init__` (positions: `dict[str, ndarray[tuple[Any, ...], dtype[int64]]]`, occupancy: `dict[str, ndarray[tuple[Any, ...], dtype[float64]]]`) → `None`

Parameters

- **positions** (`dict[str, ndarray[tuple[Any, ...], dtype[int_]]]`)
- **occupancy** (`dict[str, ndarray[tuple[Any, ...], dtype[double]]]`)

`class pyfraglib.simulator.fragment_simulator.ChromatinState`(open_regions: `dict[str, IntervalTree]`,
tf_binding_sites: `dict[str, IntervalTree]`, ctf_sites: `dict[str, IntervalTree]`) → `None`

Represents tissue-specific chromatin accessibility and regulatory states.

This class encapsulates chromatin accessibility information that affects cfDNA fragmentation patterns. Different chromatin states lead to varying nuclease accessibility and cleavage probability.

Notes

Each `IntervalTree` should contain intervals with genomic coordinates:

- Chromosome-specific trees for efficient spatial queries
- 0-based coordinate system following standard conventions
- Additional metadata can be stored as interval data

The chromatin state significantly influences:

- Base cleavage probability (open regions: ~0.6, closed regions: ~0.1)
- Transcription factor protection effects (30% of normal cleavage)
- Tissue-specific fragmentation patterns

Examples

```
>>> from intervaltree import IntervalTree
>>> chr1_open = IntervalTree()
>>> chr1_open.addi(1000, 2000) # Open chromatin region
>>> chr1_tf = IntervalTree()
>>> chr1_tf.addi(1500, 1520) # Protected TF binding site
>>> chromatin = ChromatinState(
...     open_regions={"chr1": chr1_open},
...     tf_binding_sites={"chr1": chr1_tf},
...     ctcf_sites={"chr1": IntervalTree()}
... )
```

Parameters

- **open_regions** (`dict[str, IntervalTree]`)
- **tf_binding_sites** (`dict[str, IntervalTree]`)
- **ctcf_sites** (`dict[str, IntervalTree]`)

open_regions: `dict[str, IntervalTree]`

Genomic intervals representing accessible chromatin regions (fragments originating from these regions have higher cleavage probability).

tf_binding_sites: `dict[str, IntervalTree]`

Transcription factor binding sites that provide protection from nuclease cleavage. These regions typically show reduced fragmentation.

ctcf_sites: `dict[str, IntervalTree]`

CTCF binding sites that create specific chromatin architecture and affect local nuclease accessibility patterns.

__init__ (`open_regions: dict[str, IntervalTree], tf_binding_sites: dict[str, IntervalTree], ctcf_sites: dict[str, IntervalTree]`) → `None`

Parameters

- **open_regions** (`dict[str, IntervalTree]`)
- **tf_binding_sites** (`dict[str, IntervalTree]`)
- **ctcf_sites** (`dict[str, IntervalTree]`)

```
class pyfraglib.simulator.fragment_simulator.NucleaseProfile(dnase1_activity: float = 1.0,
                                                             dnase1l3_activity: float = 1.0,
                                                             dffb_activity: float = 1.0,
                                                             dnase1_motif_preference: dict[str, float] | None = None,
                                                             dnase1l3_motif_preference: dict[str, float] | None = None,
                                                             dffb_motif_preference: dict[str, float] | None = None) → None
```

Nuclease activity and sequence preference parameters.

This class defines the activity levels and sequence preferences of 3 different nucleases involved in cfDNA fragmentation. The parameters are based on experimental literature and can be customized for different biological conditions or disease states.

Notes

Activity levels are relative and multiplicative:

- 1.0 represents normal physiological activity
- Values >1.0 increase nuclease activity
- Values <1.0 decrease nuclease activity
- 0.0 completely disables the nuclease

Motif preferences are multiplicative factors applied to base cleavage probability based on local sequence context. Higher values increase cleavage probability for sequences containing the motif.

Examples

```
>>> # Healthy individual profile
>>> healthy_profile = NucleaseProfile(
...     dnase1_activity=1.0,
...     dnase1l3_activity=1.2,
...     dffb_activity=0.1
... )
>>>
>>> # Cancer patient with increased apoptosis
>>> cancer_profile = NucleaseProfile(
...     dnase1_activity=1.1,
...     dnase1l3_activity=1.0,
...     dffb_activity=2.0
... )
```

Parameters

- **dnase1_activity** ([float](#), default: 1.0)
- **dnase1l3_activity** ([float](#), default: 1.0)
- **dffb_activity** ([float](#), default: 1.0)
- **dnase1_motif_preference** ([dict\[str, float\]](#) | [None](#), default: None)
- **dnase1l3_motif_preference** ([dict\[str, float\]](#) | [None](#), default: None)
- **dffb_motif_preference** ([dict\[str, float\]](#) | [None](#), default: None)

dnase1_activity: [float](#) = 1.0

Relative activity level of DNase I. DNase I shows preference for AT-rich accessible regions.

dnase1l3_activity: [float](#) = 1.0

Relative activity level of DNase I Like 3 (DNase1L3). This is the primary nuclease responsible for cfDNA fragmentation in healthy individuals. Shows strong CC dinucleotide preference.

dffb_activity: [float](#) = 1.0

Relative activity level of the apoptotic nuclease DNA Fragmentation Factor B (DFFB). More active during cell death and shows preference for nucleosome linker regions.

dnase1_motif_preference: [dict\[str, float\]](#) | [None](#) = None

Sequence motif preferences for DNase I. If None, uses default preferences based on literature (AT-rich sequences preferred).

dnase113_motif_preference: `dict[str, float] | None = None`

Sequence motif preferences for DNase1L3. If None, uses default preferences (strong CC dinucleotide preference).

dffb_motif_preference: `dict[str, float] | None = None`

Sequence motif preferences for DFFB. If None, uses default preferences (slight A preference).

__init__(*dnase1_activity*: `float = 1.0`, *dnase113_activity*: `float = 1.0`, *dffb_activity*: `float = 1.0`, *dnase1_motif_preference*: `dict[str, float] | None = None`, *dnase113_motif_preference*: `dict[str, float] | None = None`, *dffb_motif_preference*: `dict[str, float] | None = None`) $\rightarrow$ `None`

Parameters

- **dnase1_activity** (`float`, default: 1.0)
- **dnase113_activity** (`float`, default: 1.0)
- **dffb_activity** (`float`, default: 1.0)
- **dnase1_motif_preference** (`dict[str, float] | None`, default: None)
- **dnase113_motif_preference** (`dict[str, float] | None`, default: None)
- **dffb_motif_preference** (`dict[str, float] | None`, default: None)

class pyfraglib.simulator.fragment_simulator.**SequenceContextGenerator**(*fasta_path*: `str`) $\rightarrow$ `None`

Genomic sequence context provider for cfDNA fragment simulation.

This class interfaces with reference FASTA files to provide authentic genomic sequence context for fragment end motif generation. The class implements sequence caching and provides methods for extracting sequence context around genomic positions, enabling simulation of cfDNA fragments with somewhat realistic end motifs.

Notes

The FASTA file must be indexed with samtools faidx before use.

Examples

Direct usage:

```
seq_gen = SequenceContextGenerator("/path/to/reference.fasta")
context = seq_gen.generate_sequence_context("chr1", 1000000, 20)
print(f"Sequence context: {context}")
seq_gen.close()
```

Using as a context manager:

```
with SequenceContextGenerator("/path/to/reference.fasta") as seq_gen:
    context = seq_gen.generate_sequence_context("chr1", 1000000, 20)
```

Parameters

fasta_path (`str`)

__init__(*fasta_path*: `str`) $\rightarrow$ `None`

Initialize sequence generator with reference FASTA file.

Parameters

fasta_path (`str`) – Path to reference FASTA file (must be indexed)

generate_sequence_context(*chrom*: *str*, *position*: *int*, *length*: *int* = 20) → *str*

Extract genomic sequence context around a specified position.

This method retrieves genomic sequence from the reference FASTA file, centered around the specified position. It handles edge cases at chromosome boundaries by padding with 'N' nucleotides when necessary.

Parameters

- **chrom** (*str*) – Chromosome name (e.g., "chr1", "1", "X"). Must match chromosome naming convention in the FASTA file.
- **position** (*int*) – Genomic position (0-based coordinate system) around which to extract sequence context.
- **length** (*int*, default: 20) – Total length of sequence to extract, centered on the position. Must be positive.

Returns

DNA sequence string of specified length, uppercase nucleotides. Contains 'N' characters for undefined or unavailable regions.

Return type

str

Raises

Exception – If chromosome is not found in FASTA file or sequence extraction fails. In such cases, returns a string of 'N' characters as fallback.

Notes

The method extracts sequence symmetrically around the position: - Start position: position - length//2 - End position: start + length

Boundary handling: - If start < 0: sequence is extracted from position 0 - If end > chromosome length: sequence is padded with 'N'

Examples

```
>>> seq_gen = SequenceContextGenerator("reference.fasta")
>>> context = seq_gen.generate_sequence_context("chr1", 1000000, 20)
>>> print(f"Context: {context}") # e.g., "ATCGATCGATCGATCGATCG"
>>> len(context)
```

get_chromosome_length(*chrom*: *str*) → *int*

Retrieve chromosome length from the reference FASTA file.

Parameters

chrom (*str*) – Chromosome name to query. Must match naming convention in FASTA.

Returns

Length of the chromosome in base pairs. Returns 0 if chromosome is not found or an error occurs.

Return type

int

get_available_chromosomes() → *list*[*str*]

List all available chromosome references in the FASTA file.

Returns

List of chromosome names available in the FASTA file, in the order they appear in the file header.

Return type

`list[str]`

`close()` → `None`

Close the FASTA file handle and release resources.

Return type

`None`

```
class pyfraglib.simulator.fragment_simulator.FragmentSimulator(sequence_generator:
                                                                SequenceContextGenerator,
                                                                nucleosome_map:
                                                                NucleosomeMap | None = None,
                                                                chromatin_state: ChromatinState |
                                                                None = None) → None
```

cfDNA fragment simulator.

This simulator integrates experimental observations to model cfDNA fragmentation:

1. **Nucleosome positioning:** Nucleosome maps with occupancy scores
2. **Chromatin accessibility:** Tissue-specific chromatin states
3. **Nuclease specificity:** Experimentally-determined preferences
4. **Sequence context:** Genomic sequences from reference FASTA
5. **TF binding protection:** Transcription factor binding site shielding

Main references are Lo et al. 2021 Science, Cristiano et al. 2019 Nature.

Parameters

- **sequence_generator** (*SequenceContextGenerator*)
- **nucleosome_map** (*NucleosomeMap* | `None`, default: `None`)
- **chromatin_state** (*ChromatinState* | `None`, default: `None`)

`MONO_NUC_MEAN: Final[int] = 167`

`MONO_NUC_STD: Final[int] = 10`

`MONO_FRACTION: Final[float] = 0.8`

`MONO_SKEW: Final[float] = -3.0`

`DI_NUC_MEAN: Final[int] = 334`

`DI_NUC_STD: Final[int] = 15`

`DI_FRACTION: Final[float] = 0.15`

`TRI_NUC_MEAN: Final[int] = 501`

`TRI_NUC_STD: Final[int] = 35`

`PERIODICITY_AMPLITUDE: Final[float] = 10.0`

MIN_FRAGMENT_SIZE: `Final[int] = 40`

MAX_FRAGMENT_SIZE: `Final[int] = 900`

__init__(*sequence_generator*: `SequenceContextGenerator`, *nucleosome_map*: `NucleosomeMap` | `None` = `None`, *chromatin_state*: `ChromatinState` | `None` = `None`) → `None`

Initialize the fragment simulator.

Parameters

- **sequence_generator** (`SequenceContextGenerator`) – Sequence context generator (must be constructed with indexed FASTA file)
- **nucleosome_map** (`NucleosomeMap` | `None`, default: `None`) – Custom nucleosome positioning data. If `None`, generates default nucleosome map.
- **chromatin_state** (`ChromatinState` | `None`, default: `None`) – Chromatin accessibility information. If `None`, generates default chromatin state.

logger: `logging.Logger`

sequence_generator: `SequenceContextGenerator`

nucleosome_map: `NucleosomeMap`

chromatin_state: `ChromatinState`

static generate_nucleosome_map(*sequence_generator*: `SequenceContextGenerator`, *beta_alpha*: `float` = `4.0`, *beta_beta*: `float` = `3.0`) → `NucleosomeMap`

Generate a nucleosome map with customizable chromatin state.

This static method creates a nucleosome map using beta distribution parameters to control chromatin accessibility. Can be used by both the base simulator and tissue-specific simulators.

Parameters

- **sequence_generator** (`SequenceContextGenerator`) – Sequence generator to get chromosome information.
- **beta_alpha** (`float`, default: `4.0`) – Alpha parameter for beta distribution. Higher values increase nucleosome occupancy (more compact chromatin).
- **beta_beta** (`float`, default: `3.0`) – Beta parameter for beta distribution. Higher values decrease nucleosome occupancy (more open chromatin).

Returns

Nucleosome map with specified chromatin characteristics.

Return type

`NucleosomeMap`

Notes

Common beta parameter combinations: - Open chromatin: (2, 5) - low occupancy, high accessibility - Normal chromatin: (4, 3) - balanced occupancy - Compact chromatin: (6, 2) - high occupancy, low accessibility

simulate_fragments(*chrom*: `str`, *start*: `int`, *end*: `int`, *num_fragments*: `int`, *tissue_type*: `str` = `'healthy'`, *nuclease_profile*: `NucleaseProfile` | `None` = `None`, *fragment_size_params*: `dict[str, float]` | `None` = `None`) → `FragmentList`

Simulate cfDNA fragments from a genomic region with full biological realism.

Uses realistic nucleosome positioning, chromatin accessibility, and nuclease-specific cleavage preferences based on experimental data.

Parameters

- **chrom** ([str](#)) – Chromosome name
- **start** ([int](#)) – Start position
- **end** ([int](#)) – End position
- **num_fragments** ([int](#)) – Number of fragments to generate
- **tissue_type** ([str](#), default: 'healthy') – Source tissue type affecting chromatin state
- **nuclease_profile** ([NucleaseProfile](#) | [None](#), default: [None](#)) – Nuclease activity parameters
- **fragment_size_params** ([dict](#)[[str](#), [float](#)] | [None](#), default: [None](#)) – Custom size distribution parameters

Return type

[FragmentList](#)

Returns

FragmentList containing biologically realistic simulated fragments

4.3.10 pyfraglib.simulator.tissue_mixture_simulator

Tissue Mixture cfDNA Simulation

Simulation module for modeling more complex biological scenarios involving multiple tissue sources and cancer progression. This module extends the base `FragmentSimulator` to handle multi-tissue mixtures.

Key Applications

- **Cancer Detection:** Tumor-derived cfDNA mixed with normal tissue background
- **Disease Progression:** Longitudinal cfDNA changes over time

Example Usage

```
from pyfraglib.simulator.tissue_mixture_simulator import (
    TissueMixtureSimulator, TissueProfile
)
from pyfraglib.simulator.fragment_simulator import SequenceContextGenerator

# Initialize sequence generator and simulator
seq_gen = SequenceContextGenerator("/path/to/reference.fasta")
simulator = TissueMixtureSimulator(seq_gen)

# Simulate cancer cfDNA mixture
cancer_fragments = simulator.simulate_tissue_mixture(
    tissue_types=["hematopoietic", "tumor"],
    tissue_fractions=[0.95, 0.05], # 5% tumor fraction
    total_fragments=10000,
```

(continues on next page)

(continued from previous page)

```

    genomic_regions=[("chr1", 1000000, 1100000)]
)

# Cancer progression study
progression_data = simulator.simulate_cancer_progression(
    normal_profile="hematopoietic",
    tumor_fractions=[0.01, 0.05, 0.15, 0.30],
    time_points=["baseline", "3months", "6months", "12months"],
    fragments_per_timepoint=20000,
    genomic_regions=[("chr1", 1000000, 2000000)]
)

# Clean up
seq_gen.close()

```

Tissue Profiles

Predefined tissue profiles:

- **hematopoietic**: Blood cell-derived cfDNA (normal background)
- **liver**: Hepatocyte-derived cfDNA (organ damage monitoring)
- **tumor**: Generic tumor-derived cfDNA (cancer detection)

Custom tissue profiles can be defined using the `TissueProfile` dataclass.

References

- Wan et al. (2017) Nature Reviews: cfDNA fragmentation patterns
- Cristiano et al. (2019) Nature: Fragmentomics for cancer detection
- Sun et al. (2019) PNAS: Tissue-specific cfDNA fragmentation

License

Copyright (C) 2025 Daniel Schütte, daniel.schuette@iccb-cologne.org

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

Classes

<code>TissueMixtureSimulator</code> (sequence_generator, ...)	cfDNA simulator for multi-tissue mixtures.
<code>TissueProfile</code> (name, ...)	Tissue-specific fragmentation parameter profile.

```
class pyfraglib.simulator.tissue_mixture_simulator.TissueProfile(name: str,  
                                                                  fragment_size_distribution:  
                                                                    dict[str, float],  
                                                                  nuclease_activities: dict[str,  
                                                                    float], chromatin_beta_params:  
                                                                    tuple[float, float]) → None
```

Tissue-specific fragmentation parameter profile.

This class encapsulates tissue-specific parameters that influence cfDNA fragmentation patterns like fragment size distributions, chromatin accessibility, and nuclease activity patterns.

Note

Currently, chromatin accessibility can only be set through the alpha and beta parameters of beta distribution that generates the default nucleosome map. In future releases, data-based nucleosome maps will be properly handled

Examples

```
>>> custom_tissue = TissueProfile(  
...     name="custom_cancer",  
...     fragment_size_distribution={  
...         "mean": 155, "std": 15, "mono_fraction": 0.8  
...     },  
...     nuclease_activities={  
...         "dnase1_activity": 1.2,  
...         "dnase1l3_activity": 1.0,  
...         "dfffb_activity": 1.5  
...     },  
...     chromatin_beta_params=(2.5, 4.5) # Open, disrupted chromatin  
... )
```

Parameters

- **name** (`str`)
- **fragment_size_distribution** (`dict[str, float]`)
- **nuclease_activities** (`dict[str, float]`)
- **chromatin_beta_params** (`tuple[float, float]`)

name: `str`

Tissue identifier (e.g., “hematopoietic”, “liver”, “tumor”).

fragment_size_distribution: `dict[str, float]`

Parameters defining fragment size distribution. See `_generate_fragment_sizes` for possible parameters.

nuclease_activities: `dict[str, float]`

Tissue-specific nuclease activity levels. Available nucleases:

- `dnase1_activity`: DNase I activity level (typical range: 0.5-1.5)
- `dnase1l3_activity`: DNase1L3 activity level (typical range: 0.8-1.5)
- `dfffb_activity`: DFFB activity level (typical range: 0.1-2.0)

chromatin_beta_params: `tuple[float, float]`

Beta distribution parameters for nucleosome occupancy:

- alpha: Shape parameter α (higher = more occupied nucleosomes)
- beta: Shape parameter β (higher = more open chromatin)
- Typical combinations: open (2,5), normal (4,3), compact (6,2)

__init__ (name: `str`, fragment_size_distribution: `dict[str, float]`, nuclease_activities: `dict[str, float]`, chromatin_beta_params: `tuple[float, float]`) $\rightarrow$ `None`

Parameters

- name (`str`)
- fragment_size_distribution (`dict[str, float]`)
- nuclease_activities (`dict[str, float]`)
- chromatin_beta_params (`tuple[float, float]`)

class pyfraglib.simulator.tissue_mixture_simulator.TissueMixtureSimulator(sequence_generator: `SequenceContextGenerator`, **kwargs: `dict[str, object]`) $\rightarrow$ `None`

cfDNA simulator for multi-tissue mixtures. This class extends the base FragmentSimulator.

Examples

Basic tissue mixture simulation:

```
>>> seq_gen = SequenceContextGenerator("/path/to/reference.fasta")
>>> simulator = TissueMixtureSimulator(seq_gen)
>>> fragments = simulator.simulate_tissue_mixture(
...     tissue_types=["hematopoietic", "tumor"],
...     tissue_fractions=[0.90, 0.10],
...     total_fragments=10000,
...     genomic_regions=[("chr1", 1000000, 11000000)]
... )
```

Cancer progression study:

```
>>> progression = simulator.simulate_cancer_progression(
...     normal_profile="hematopoietic",
...     tumor_fractions=[0.01, 0.05, 0.15],
...     time_points=["diagnosis", "3months", "6months"],
...     fragments_per_timepoint=20000,
...     genomic_regions=[("chr1", 1000000, 20000000)],
...     size_shift=-20.0, short_enrichment=0.25
... )
```

Parameters

- sequence_generator (`SequenceContextGenerator`)
- kwargs (`dict[str, object]`)

__init__(*sequence_generator*: [SequenceContextGenerator](#), ***kwargs*: [dict\[str, object\]](#)) → [None](#)

Initialize the fragment simulator.

Parameters

- **sequence_generator** ([SequenceContextGenerator](#)) – Sequence context generator (must be constructed with indexed FASTA file)
- **nucleosome_map** – Custom nucleosome positioning data. If None, generates default nucleosome map.
- **chromatin_state** – Chromatin accessibility information. If None, generates default chromatin state.
- **kwargs** ([dict\[str, object\]](#))

add_custom_tissue(*tissue_profile*: [TissueProfile](#)) → [None](#)

Add a custom tissue profile to the simulator's available profiles.

This method allows users to define custom tissue types with specific fragmentation characteristics beyond the predefined profiles.

Parameters

tissue_profile ([TissueProfile](#)) – Complete tissue profile definition including name, fragment size distribution, nucleosome spacing, chromatin openness, end motif preferences, and methylation level.

Return type

[None](#)

simulate_tissue_mixture(*tissue_types*: [list\[str\]](#), *tissue_fractions*: [list\[float\]](#), *total_fragments*: [int](#), *genomic_regions*: [list\[tuple\[str, int, int\]\]](#), *add_noise*: [bool](#) = [True](#)) → [FragmentList](#)

Generate cfDNA mixture from multiple tissue sources.

Parameters

- **tissue_types** ([list\[str\]](#)) – List of tissue type names to include in the mixture. Must be present in *tissue_profiles* (either predefined or custom).
- **tissue_fractions** ([list\[float\]](#)) – Fraction contribution from each tissue type. Must sum to 1.0 (within 0.001 tolerance). Order corresponds to *tissue_types*.
- **total_fragments** ([int](#)) – Total number of fragments to generate across all tissues. Individual tissue contributions are calculated proportionally.
- **genomic_regions** ([list\[tuple\[str, int, int\]\]](#)) – List of genomic regions (chromosome, start, end) to sample fragments from. Fragments are distributed across regions.
- **add_noise** ([bool](#), default: [True](#)) – Whether to add realistic biological and technical noise:
 - Random fragment loss (~5%)
 - Size variation (± 2 bp standard deviation)
 - End repair artifacts (~2%)

Returns

Combined fragment list containing fragments from all tissues with tissue-specific characteristics. Fragments are shuffled to simulate realistic mixing.

Return type

[FragmentList](#)

Raises

[SystemExit](#) – If tissue fractions don't sum to 1.0 or if unknown tissue types are requested.

Examples

Cancer detection mixture (5% tumor fraction):

```
>>> fragments = simulator.simulate_tissue_mixture(
...     tissue_types=["hematopoietic", "tumor"],
...     tissue_fractions=[0.95, 0.05],
...     total_fragments=50000,
...     genomic_regions=[("chr1", 1000000, 1500000)],
...     add_noise=True
... )
```

simulate_cancer_progression(normal_profile: *str*, tumor_fractions: *list[float]*, time_points: *list[str]*, fragments_per_timepoint: *int*, genomic_regions: *list[tuple[str, int, int]]*)
→ *dict[str, FragmentList]*

Model longitudinal cancer progression.

Parameters

- **normal_profile** (*str*) – Name of normal tissue profile to use as background (typically “hematopoietic” for blood-based cfDNA).
- **tumor_fractions** (*list[float]*) – Tumor-derived cfDNA fraction at each time point. Values should be in [0, 1] range and typically increase over time for progression studies.
- **time_points** (*list[str]*) – Labels for each time point (e.g., [“baseline”, “3months”]). Must have same length as tumor_fractions.
- **fragments_per_timepoint** (*int*) – Number of fragments to generate at each time point. Consistent across time points for comparative analysis.
- **genomic_regions** (*list[tuple[str, int, int]]*) – Genomic regions (chromosome, start, end) to sample fragments from. Same regions used for all time points.

Returns

Dictionary mapping time point labels to *FragmentList* objects. Each *FragmentList* contains the cfDNA profile for that time point with appropriate tumor fraction and cancer signatures.

Return type

dict[str, FragmentList]

Raises

SystemExit – If tumor_fractions and time_points have different lengths.

Examples

```
>>> progression = simulator.simulate_cancer_progression(
...     normal_profile="hematopoietic",
...     tumor_fractions=[0.01, 0.03, 0.08, 0.15],
...     time_points=["diagnosis", "1month", "3months", "6months"],
...     fragments_per_timepoint=25000,
...     genomic_regions=[("chr1", 1000000, 2000000)]
... )
```

SIMULATION

`pyfraglib` includes a simulation module for generating synthetic cfDNA data. This is useful for testing algorithms, validating methods, and creating benchmarking datasets.

5.1 Overview

Simulations are based on the biological processes that underlie DNA fragmentation and shedding into the blood stream. The API reference explains the assumptions (and the scientific references) in greater detail. The simulation module provides different simulation modes (see below). All simulations use genomic sequences from FASTA files to generate realistic fragment end motifs and maintain biological accuracy to a certain degree.

The API integrates with the rest of the `pyfraglib` ecosystem:

```
from pyfraglib import FragmentSimulator
from pyfraglib.simulator.fragment_simulator import SequenceContextGenerator

seq_gen = SequenceContextGenerator("reference.fasta")
simulator = FragmentSimulator(seq_gen)

fragments = simulator.simulate_fragments(
    chrom="chr1", start=1_000_000, end=1_100_000,
    num_fragments=10000
)
fragments.to_frag_file("synthetic_sample", "output/")
```

For use in pipelines, we provide a CLI, too:

```
pyfrag.py simulate --config simulation_config.json --out-dir synthetic/
```

5.2 Simulation Modes

1. Basic Simulation: Generate fragments from a single tissue type with customizable parameters
2. Tissue Mixture Simulation: Simulate mixtures of multiple tissue types with specified fractions
3. Cancer Progression: Simulate tumor fraction changes over time for longitudinal studies

Please refer to the API documentation to learn more about the specific parameters available.

5.3 Architecture

The simulation module is implemented as a multi-layered architecture. The following diagram shows the key components and their relationships:

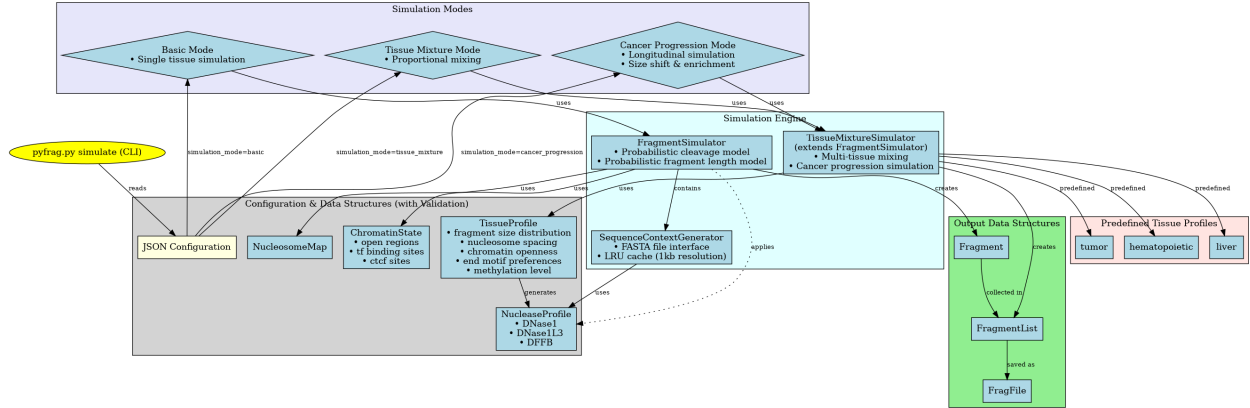


Fig. 5.1: cfDNA fragment simulation architecture showing the key components and their relationships.

5.4 Configuration

Simulations are configured through JSON files with the following structure:

```
{
  "fasta_path": "/path/to/reference.fasta",
  "output_name": "simulation_output",
  "regions": "/path/to/regions.bed",
  "num_fragments": 50000,
  "simulation_mode": "tissue_mixture",
  "mean": 167,
  "std": 10,
  "tissue_fractions": {
    "hematopoietic": 0.7,
    "liver": 0.2,
    "tumor": 0.1
  }
}
```

Since the API is still under development, this specification might slightly change in the future. Please have a look at the example configuration files to learn more about what is available.

5.5 Probabilistic Fragment Size Model

The simulator generates fragment size distributions by using a simplified, but biology-oriented probabilistic model. That fragment size model combines mono-, di-, and tri-nucleosomal components with characteristic periodicity patterns observed in cfDNA.

The fragment size distribution is modeled as a mixture of skewed and normal components:

$$L \sim f_{\text{mono}} \cdot \text{SkewNormal}(\text{loc}_{\text{adj}}, \sigma_{\text{mono}}^2, \alpha_{\text{mono}}) + f_{\text{di}} \cdot \mathcal{N}(\mu_{\text{di}} + \Delta, \sigma_{\text{di}}^2) + f_{\text{tri}} \cdot \mathcal{N}(\mu_{\text{tri}} + \Delta, \sigma_{\text{tri}}^2)$$

where L is the fragment length, f_{mono} , f_{di} , and f_{tri} are the mixing fractions with $f_{\text{mono}} + f_{\text{di}} + f_{\text{tri}} = 1.0$, and α_{mono} is the skewness parameter for the mono-nucleosomal peak (negative values create left-skewed distributions).

Location Parameter Adjustment: To ensure the peak (mode) of the skew-normal distribution occurs at the user-configured mean μ_{mono} , the location parameter is automatically adjusted:

$$\begin{aligned}\delta &= \frac{\alpha_{\text{mono}}}{\sqrt{1 + \alpha_{\text{mono}}^2}} \\ \text{mode}_{\text{offset}} &= \sigma_{\text{mono}} \times \delta \times \sqrt{\frac{2}{\pi}} \\ \text{loc}_{\text{adj}} &= \mu_{\text{mono}} + \Delta - \text{mode}_{\text{offset}}\end{aligned}$$

This adjustment ensures that when users specify a mono-nucleosomal mean of 167 bp, the distribution peak appears at approximately 167 bp rather than having the peak heavily shifted due to skewness.

Default Parameters:

$$\begin{aligned}\mu_{\text{mono}} &= 167 \text{ bp (mono-nucleosomal peak)} \\ \sigma_{\text{mono}} &= 10 \text{ bp} \\ \alpha_{\text{mono}} &= -3.0 \text{ (left-skewed)} \\ \mu_{\text{di}} &= 334 \text{ bp (di-nucleosomal peak)} \\ \sigma_{\text{di}} &= 15 \text{ bp} \\ \mu_{\text{tri}} &= 501 \text{ bp (di-nucleosomal peak)} \\ \sigma_{\text{tri}} &= 35 \text{ bp} \\ f_{\text{mono}} &= 0.80 \text{ (mono-nucleosomal fraction)} \\ f_{\text{di}} &= 0.15 \text{ (di-nucleosomal fraction)} \\ f_{\text{tri}} &= 0.15 \text{ (di-nucleosomal fraction)}\end{aligned}$$

The size shift parameter Δ allows modeling changes in fragment size distributions observed e.g. in cancer, where an increase in short fragments (with $\Delta < 0$) can be observed.

10 bp Periodicity: cfDNA fragments exhibit characteristic 10 bp periodicity due to nucleosome positioning and DNA helical structure. The periodicity is applied only to fragments shorter than the mono-nucleosomal peak, with amplitude decreasing toward the peak:

$$L_{\text{final}} = L + P_{\text{push}} \quad \text{for } L \leq \mu_{\text{mono}}$$

The periodicity works by attracting fragments toward the nearest 10 bp multiple using Gaussian attraction:

$$P_{\text{push}} = -\Delta L \times \exp\left(-\frac{\Delta L^2}{2\sigma_{\text{gaussian}}^2}\right) \times A_{\text{gradient}}$$

where $\Delta L = L - \text{round}(L/10) \times 10$ is the offset from the nearest 10 bp multiple, and the amplitude gradient decreases toward the mono-nucleosomal peak:

$$A_{\text{gradient}} = \frac{\mu_{\text{mono}} - L}{\mu_{\text{mono}} - L_{\text{min}}} \times A_{\text{period}}$$

The Gaussian width $\sigma_{\text{gaussian}} = 5.0$ controls the smoothness of the periodicity, and A_{period} is the configurable periodicity amplitude with default value 10.0.

Size Constraints: Lastly, size constraints are applied as follows:

$$L_{\text{constrained}} = \text{clip}(L_{\text{final}}, L_{\text{min}}, L_{\text{max}})$$

with default bounds $L_{\text{min}} = 40$ bp and $L_{\text{max}} = 900$ bp.

5.6 Probabilistic Cleavage Model

The simulator uses a biologically-grounded probabilistic model to determine where cfDNA fragments are cleaved. The per-base cleavage probability integrates multiple biological factors through multiplicative interactions.

The cleavage probability at genomic position i is given by:

$$P'_{\text{cleavage}}(i) = P_{\text{base}}(i) \times F_{\text{nucleosome}}(i) \times F_{\text{tissue}} \times F_{\text{nuclease}}(i) \times F_{\text{tf}}(i)$$

where each factor represents a different biological process influencing DNA accessibility and nuclease activity. The base accessibility depends on chromatin state:

$$P_{\text{base}}(i) = \begin{cases} 0.6 & \text{if position } i \in \text{open chromatin regions} \\ 0.1 & \text{if position } i \notin \text{open chromatin regions} \end{cases}$$

Nucleosome positioning provides distance-dependent protection (multiplicative factors where values < 1.0 reduce cleavage probability):

$$F_{\text{nucleosome}}(i) = \begin{cases} 0.05 + (1 - O_j) \times 0.15 & \text{if } d_{ij} \leq 73 \text{ bp (core)} \\ 0.20 + (1 - O_j) \times 0.30 & \text{if } 73 < d_{ij} \leq 120 \text{ bp (edge)} \\ 0.70 + (1 - O_j) \times 0.30 & \text{if } d_{ij} > 120 \text{ bp (linker)} \end{cases}$$

where d_{ij} is the distance from position i to the nearest nucleosome center j , O_j is the occupancy score of nucleosome j (range: 0-1). Different tissues exhibit characteristic chromatin accessibility patterns:

$$F_{\text{tissue}} = \begin{cases} 1.0 & \text{healthy baseline} \\ 1.2 & \text{hematopoietic (open chromatin)} \\ 0.9 & \text{liver (compact chromatin)} \\ 1.4 & \text{tumor (disrupted chromatin)} \end{cases}$$

The nuclease factor integrates activity levels and sequence preferences across all active nucleases:

$$F_{\text{nuclease}}(i) = \frac{\sum_k A_k \times P_k(i)}{\sum_k A_k}$$

where $k \in \{\text{DNase1, DNase1L3, DFFB}\}$, A_k is the activity level of nuclease k , $P_k(i)$ is the sequence preference factor for nuclease k at position i .

DNase I and DNase1L3 sequence preferences:

$$P_k(i) = \prod_m [1 + (\text{pref}_m - 1) \times \text{freq}_m(s_i)]$$

DFFB combines sequence preferences with nucleosome linker preference:

$$P_{\text{DFFB}}(i) = \left[\prod_m (1 + (\text{pref}_m - 1) \times \text{freq}_m(s_i)) \right] \times (0.5 + 1.5 \times F_{\text{nucleosome}}(i))$$

where m represents different motifs (e.g., “CC”, “AT”, “A”, “T”), pref_m is the preference value for motif m (> 1.0 favored, < 1.0 disfavored), $\text{freq}_m(s_i)$ is the frequency/occurrence of motif m in the local sequence context s_i , s_i is the 20bp sequence window centered at position i . Transcription factor binding sites provide protection from nuclease cleavage:

$$F_{\text{tf}}(i) = \begin{cases} 0.3 & \text{if position } i \in \text{TF binding sites} \\ 1.0 & \text{if position } i \notin \text{TF binding sites} \end{cases}$$

The final cleavage probability is bounded:

$$P_{\text{cleavage}}(i) = \min(1.0, \max(0.001, P'_{\text{cleavage}}(i)))$$

CONTRIBUTING

We welcome contributions to `pyfraglib`! This document outlines how to contribute to the project.

6.1 Getting Started

See [Installation](#) for installation instructions. After the initial setup, please create a feature branch and follow our coding standards outlined below. Please make sure that your commit messages roughly follow our style. After every commit, use

```
./tools/dev_install.sh
```

to run our unit test suite, linting, and type checking. If all tests pass, you can create a pull request and we will merge your code after a quick review.

6.2 Coding Standards

6.2.1 Code Style

- Follow PEP 8 style guidelines
- Use double quotes for strings (" not ')
- Maximum line length: < 80 characters (this is also enforced by the linter)
- Use descriptive variable and function names (`snake_case` for functions, methods, and variables; `CamelCase` for class names)

6.2.2 Type Annotations

All functions must have complete type annotations! Please use `typing.Final` for constants and `typing.NoReturn` for functions that don't return to the caller. We are trying to follow very strict mypy settings as defined in `pyproject.toml`, but because some Python libraries are not typed and others rely on dynamic typing a lot, we sometimes need to disable mypy. Nonetheless, please use `# type: ignore` very, very sparingly

6.2.3 Documentation and Testing

- All public functions and classes must have docstrings
- Include parameter descriptions and return value information
- Add examples for complex functions
- Update relevant `.rst` files when implementing new features
- Write as many unit tests for new functionality as possible

- Use the existing test fixtures in `tests/test_fixtures.py` and Follow the naming convention `test_<functionality>`

Thank you for contributing to pyfraglib!

CHANGELOG

7.1 Version 0.5.1 (Current)

New Features:

- Added comprehensive simulation module
- Added unit tests
- Added Sphinx documentation

7.2 Future Releases

Roadmap of planned features:

- Add additional simulation features

7.3 Contributing

See [Contributing](#) for information on how to contribute to future releases.

INDICES

The following pages provide indices for navigating Python modules and API symbols.

PYTHON MODULE INDEX

p

- `pyfraglib.core`, [40](#)
- `pyfraglib.fragfile`, [60](#)
- `pyfraglib.fragment`, [44](#)
- `pyfraglib.lengths`, [70](#)
- `pyfraglib.math`, [63](#)
- `pyfraglib.scores`, [73](#)
- `pyfraglib.simulator`, [84](#)
- `pyfraglib.simulator.fragment_simulator`, [92](#)
- `pyfraglib.simulator.tissue_mixture_simulator`, [102](#)
- `pyfraglib.stats`, [67](#)

Symbols

__init__() (pyfraglib.core.CodeUnreachableError method), 42

__init__() (pyfraglib.core.PyfraglibException method), 42

__init__() (pyfraglib.fragfile.FragFile method), 62

__init__() (pyfraglib.fragment.Fragment method), 49

__init__() (pyfraglib.fragment.FragmentCollection method), 59

__init__() (pyfraglib.fragment.FragmentList method), 56

__init__() (pyfraglib.fragment.IntervalTable method), 58

__init__() (pyfraglib.simulator.FragmentSimulator method), 85

__init__() (pyfraglib.simulator.NucleaseProfile method), 88

__init__() (pyfraglib.simulator.SequenceContextGenerator method), 88

__init__() (pyfraglib.simulator.TissueMixtureSimulator method), 90

__init__() (pyfraglib.simulator.fragment_simulator.ChromatinState method), 96

__init__() (pyfraglib.simulator.fragment_simulator.FragmentSimulator method), 101

__init__() (pyfraglib.simulator.fragment_simulator.NucleaseProfile method), 98

__init__() (pyfraglib.simulator.fragment_simulator.NucleosomeMap method), 95

__init__() (pyfraglib.simulator.fragment_simulator.SequenceContextGenerator method), 98

__init__() (pyfraglib.simulator.tissue_mixture_simulator.TissueMixtureSimulator method), 105

__init__() (pyfraglib.simulator.tissue_mixture_simulator.TissueProfile method), 105

A

add_custom_tissue() (pyfraglib.simulator.tissue_mixture_simulator.TissueMixtureSimulator method), 106

add_custom_tissue() (pyfraglib.simulator.TissueMixtureSimulator method), 91

append() (pyfraglib.fragment.FragmentCollection method), 59

append() (pyfraglib.fragment.FragmentList method), 56

B

bams_to_fragments() (pyfraglib.fragment.Fragment static method), 53

build_mutated_reads_set() (pyfraglib.fragment.Fragment static method), 51

C

chrom (pyfraglib.fragment.Fragment attribute), 49

chromatin_beta_params (pyfraglib.simulator.tissue_mixture_simulator.TissueProfile attribute), 104

chromatin_state (pyfraglib.simulator.fragment_simulator.FragmentSimulator attribute), 101

chromatin_state (pyfraglib.simulator.FragmentSimulator attribute), 85

ChromatinState (class in pyfraglib.simulator.fragment_simulator), 95

chromosome_maps_to_df() (in module pyfraglib.scores), 78

close() (pyfraglib.fragfile.FragFile method), 63

close() (pyfraglib.simulator.fragment_simulator.SequenceContextGenerator method), 100

close() (pyfraglib.simulator.SequenceContextGenerator method), 90

closed (pyfraglib.fragfile.FragFile property), 63

CodeUnreachableError, 42

count_bogus_fragments() (pyfraglib.fragment.FragmentList method), 56

count_endmotifs() (pyfraglib.fragment.FragmentList method), 57

count_mutated_fragments() (pyfraglib.fragment.FragmentList method), 56

create_chromosome_map() (in module pyfraglib.scores), 78

create_simulated() (pyfraglib.fragment.Fragment class method), 54

ctcf_sites (pyfraglib.simulator.fragment_simulator.ChromatinState attribute), 96

D

detect_cpus() (in module pyfraglib.core), 44

- [dffb_activity](#) ([pyfraglib.simulator.fragment_simulator.NucleaseProfile](#) attribute), 97
[dffb_activity](#) ([pyfraglib.simulator.NucleaseProfile](#) attribute), 87
[dffb_motif_preference](#) ([pyfraglib.simulator.fragment_simulator.NucleaseProfile](#) attribute), 98
[dffb_motif_preference](#) ([pyfraglib.simulator.NucleaseProfile](#) attribute), 88
[DI_FRACTION](#) ([pyfraglib.simulator.fragment_simulator.FragmentSimulator](#) attribute), 100
[DI_FRACTION](#) ([pyfraglib.simulator.FragmentSimulator](#) attribute), 85
[DI_NUC_MEAN](#) ([pyfraglib.simulator.fragment_simulator.FragmentSimulator](#) attribute), 100
[DI_NUC_MEAN](#) ([pyfraglib.simulator.FragmentSimulator](#) attribute), 85
[DI_NUC_STD](#) ([pyfraglib.simulator.fragment_simulator.FragmentSimulator](#) attribute), 100
[DI_NUC_STD](#) ([pyfraglib.simulator.FragmentSimulator](#) attribute), 85
[dnase1_activity](#) ([pyfraglib.simulator.fragment_simulator.NucleaseProfile](#) attribute), 97
[dnase1_activity](#) ([pyfraglib.simulator.NucleaseProfile](#) attribute), 87
[dnase1_motif_preference](#) ([pyfraglib.simulator.fragment_simulator.NucleaseProfile](#) attribute), 97
[dnase1_motif_preference](#) ([pyfraglib.simulator.NucleaseProfile](#) attribute), 87
[dnase113_activity](#) ([pyfraglib.simulator.fragment_simulator.NucleaseProfile](#) attribute), 97
[dnase113_activity](#) ([pyfraglib.simulator.NucleaseProfile](#) attribute), 87
[dnase113_motif_preference](#) ([pyfraglib.simulator.fragment_simulator.NucleaseProfile](#) attribute), 97
[dnase113_motif_preference](#) ([pyfraglib.simulator.NucleaseProfile](#) attribute), 87
- ## E
- [end3p](#) ([pyfraglib.fragment.Fragment](#) attribute), 49
[end5p](#) ([pyfraglib.fragment.Fragment](#) attribute), 49
[end_motifs_barplot\(\)](#) (in module [pyfraglib.stats](#)), 68
[end_pos](#) ([pyfraglib.fragment.Fragment](#) attribute), 49
[export_end_motifs_csv\(\)](#) (in module [pyfraglib.stats](#)), 69
[export_length_distribution_csv\(\)](#) (in module [pyfraglib.stats](#)), 68
- ## F
- [fail\(\)](#) (in module [pyfraglib.core](#)), 42
[find_skew_normal_mean_from_mode\(\)](#) (in module [pyfraglib.math](#)), 64
[fit_gmm\(\)](#) (in module [pyfraglib.math](#)), 66
[FragFile](#) (class in [pyfraglib.fragment](#)), 61
[Fragment](#) (class in [pyfraglib.fragment](#)), 48
[fragment_from_row\(\)](#) (in module [pyfraglib.fragment](#)), 61
[fragment_length_gmm\(\)](#) (in module [pyfraglib.lengths](#)), 71
[fragment_length_plot\(\)](#) (in module [pyfraglib.lengths](#)), 70
[fragment_size_distribution](#) ([pyfraglib.simulator.tissue_mixture_simulator.TissueProfile](#) attribute), 104
[FragmentCollection](#) (class in [pyfraglib.fragment](#)), 58
[FragmentCollection\(\)](#) ([pyfraglib.core.PyfragManager](#) method), 43
[FragmentList](#) (class in [pyfraglib.fragment](#)), 55
[fragments_per_chromosome_barplot\(\)](#) (in module [pyfraglib.stats](#)), 67
[FragmentSimulator](#) (class in [pyfraglib.simulator](#)), 84
[FragmentSimulator](#) (class in [pyfraglib.simulator.fragment_simulator](#)), 100
[from_bam\(\)](#) ([pyfraglib.fragment.Fragment](#) static method), 50
[from_bams\(\)](#) ([pyfraglib.fragment.Fragment](#) static method), 53
- ## G
- [gaussian_mixture\(\)](#) (in module [pyfraglib.math](#)), 64
[generate_nucleosome_map\(\)](#) ([pyfraglib.simulator.fragment_simulator.FragmentSimulator](#) static method), 101
[generate_nucleosome_map\(\)](#) ([pyfraglib.simulator.FragmentSimulator](#) static method), 85
[generate_sequence_context\(\)](#) ([pyfraglib.simulator.fragment_simulator.SequenceContextGenerator](#) method), 98
[generate_sequence_context\(\)](#) ([pyfraglib.simulator.SequenceContextGenerator](#) method), 88
[get_available_chromosomes\(\)](#) ([pyfraglib.simulator.fragment_simulator.SequenceContextGenerator](#) method), 99
[get_available_chromosomes\(\)](#) ([pyfraglib.simulator.SequenceContextGenerator](#) method), 89
[get_chromosome_length\(\)](#) (in module [pyfraglib.core](#)), 43
[get_chromosome_length\(\)](#) ([pyfraglib.simulator.fragment_simulator.SequenceContextGenerator](#) method), 99
[get_chromosome_length\(\)](#) ([pyfraglib.simulator.SequenceContextGenerator](#) method), 89
[get_end_motifs\(\)](#) ([pyfraglib.fragment.Fragment](#) method), 50
[get_fragment_list\(\)](#) ([pyfraglib.fragment.FragFile](#) method), 62
[get_logger\(\)](#) (in module [pyfraglib.core](#)), 42
[get_overlaps\(\)](#) ([pyfraglib.fragment.IntervalTable](#) method), 58
[goodness_of_fit_stats\(\)](#) (in module [pyfraglib.math](#)), 66

H

`hessian()` (in module `pyfraglib.math`), 65
`homogenize_contig_name()` (in module `pyfraglib.core`), 43
`homogenize_to_chrom_naming_convention()` (in module `pyfraglib.core`), 43

I

`insert()` (`pyfraglib.fragment.IntervalTable` method), 58
`IntervalTable` (class in `pyfraglib.fragment`), 57
`is_bogus` (`pyfraglib.fragment.Fragment` attribute), 49
`is_duplex()` (in module `pyfraglib.fragment`), 57
`is_forward` (`pyfraglib.fragment.Fragment` attribute), 49
`is_mutated` (`pyfraglib.fragment.Fragment` attribute), 49
`ITER_BATCH_SIZE` (`pyfraglib.fragfile.FragFile` attribute), 62

J

`jensen_shannon_divergence()` (in module `pyfraglib.math`), 66

L

`length` (`pyfraglib.fragment.Fragment` attribute), 49
`length()` (`pyfraglib.fragment.FragmentList` method), 56
`log_stats()` (in module `pyfraglib.stats`), 68
`logger`
 (`pyfraglib.simulator.fragment_simulator.FragmentSimulator`
 attribute), 101
`logger` (`pyfraglib.simulator.FragmentSimulator` attribute), 85

M

`MAX_FRAGMENT_SIZE`
 (`pyfraglib.simulator.fragment_simulator.FragmentSimulator`
 attribute), 101
`MAX_FRAGMENT_SIZE` (`pyfraglib.simulator.FragmentSimulator`
 attribute), 85
`MIN_FRAGMENT_SIZE`
 (`pyfraglib.simulator.fragment_simulator.FragmentSimulator`
 attribute), 100
`MIN_FRAGMENT_SIZE` (`pyfraglib.simulator.FragmentSimulator`
 attribute), 85
`mixture_cdf()` (in module `pyfraglib.math`), 65
`mixture_cdf_wrapper()` (in module `pyfraglib.math`), 65
module
 `pyfraglib.core`, 40
 `pyfraglib.fragfile`, 60
 `pyfraglib.fragment`, 44
 `pyfraglib.lengths`, 70
 `pyfraglib.math`, 63
 `pyfraglib.scores`, 73
 `pyfraglib.simulator`, 84
 `pyfraglib.simulator.fragment_simulator`, 92
 `pyfraglib.simulator.tissue_mixture_simulator`,
 102
 `pyfraglib.stats`, 67
`MONO_FRACTION`
 (`pyfraglib.simulator.fragment_simulator.FragmentSimulator`
 attribute), 100
`MONO_FRACTION` (`pyfraglib.simulator.FragmentSimulator`
 attribute), 85

`MONO_NUC_MEAN`
 (`pyfraglib.simulator.fragment_simulator.FragmentSimulator`
 attribute), 100
`MONO_NUC_MEAN` (`pyfraglib.simulator.FragmentSimulator`
 attribute), 85
`MONO_NUC_STD`
 (`pyfraglib.simulator.fragment_simulator.FragmentSimulator`
 attribute), 100
`MONO_NUC_STD` (`pyfraglib.simulator.FragmentSimulator`
 attribute), 85
`MONO_SKEW`
 (`pyfraglib.simulator.fragment_simulator.FragmentSimulator`
 attribute), 100
`MONO_SKEW` (`pyfraglib.simulator.FragmentSimulator` attribute),
 85
`motif_diversity()` (in module `pyfraglib.scores`), 76

N

`name`
 (`pyfraglib.simulator.tissue_mixture_simulator.TissueProfile`
 attribute), 104
`negative_log_likelihood()` (in module `pyfraglib.math`), 65
`nuclease_activities`
 (`pyfraglib.simulator.tissue_mixture_simulator.TissueProfile`
 attribute), 104
`NucleaseProfile` (class in `pyfraglib.simulator`), 86
`NucleaseProfile` (class in
 `pyfraglib.simulator.fragment_simulator`), 96
`nucleosome_map`
 (`pyfraglib.simulator.fragment_simulator.FragmentSimulator`
 attribute), 101
`nucleosome_map` (`pyfraglib.simulator.FragmentSimulator`
 attribute), 85
`NucleosomeMap` (class in
 `pyfraglib.simulator.fragment_simulator`), 94
`occupancy`
 (`pyfraglib.simulator.fragment_simulator.NucleosomeMap`
 attribute), 95
`open_regions`
 (`pyfraglib.simulator.fragment_simulator.ChromatinState`
 attribute), 96

P

`parse_bed_file()` (in module `pyfraglib.core`), 44
`PERIODICITY_AMPLITUDE`
 (`pyfraglib.simulator.fragment_simulator.FragmentSimulator`
 attribute), 100
`PERIODICITY_AMPLITUDE`
 (`pyfraglib.simulator.FragmentSimulator` attribute),
 85
`plot_gmm()` (in module `pyfraglib.math`), 66
`positions`
 (`pyfraglib.simulator.fragment_simulator.NucleosomeMap`
 attribute), 95
`precalc_size()` (in module `pyfraglib.scores`), 78
`pyfraglib.core`

- module, [40](#)
- pyfraglib.fragfile
 - module, [60](#)
- pyfraglib.fragment
 - module, [44](#)
- pyfraglib.lengths
 - module, [70](#)
- pyfraglib.math
 - module, [63](#)
- pyfraglib.scores
 - module, [73](#)
- pyfraglib.simulator
 - module, [84](#)
- pyfraglib.simulator.fragment_simulator
 - module, [92](#)
- pyfraglib.simulator.tissue_mixture_simulator
 - module, [102](#)
- pyfraglib.stats
 - module, [67](#)
- PyfraglibException, [42](#)
- PyfragManager (class in pyfraglib.core), [42](#)

R

- read_gmm_config() (in module pyfraglib.math), [65](#)
- record_names() (pyfraglib.fragment.FragmentCollection method), [59](#)

S

- score_line_plot() (in module pyfraglib.scores), [81](#)
- sequence_generator
 - (pyfraglib.simulator.fragment_simulator.FragmentSimulator attribute), [101](#)
- sequence_generator
 - (pyfraglib.simulator.FragmentSimulator attribute), [85](#)
- SequenceContextGenerator (class in pyfraglib.simulator), [88](#)
- SequenceContextGenerator (class in pyfraglib.simulator.fragment_simulator), [98](#)
- shannon_entropy() (in module pyfraglib.core), [43](#)
- simpson_index() (in module pyfraglib.core), [44](#)
- simulate_cancer_progression()
 - (pyfraglib.simulator.tissue_mixture_simulator.TissueMixtureSimulator method), [107](#)
- simulate_cancer_progression()
 - (pyfraglib.simulator.TissueMixtureSimulator method), [92](#)
- simulate_fragments()
 - (pyfraglib.simulator.fragment_simulator.FragmentSimulator method), [101](#)
- simulate_fragments()
 - (pyfraglib.simulator.FragmentSimulator method), [86](#)
- simulate_tissue_mixture()
 - (pyfraglib.simulator.tissue_mixture_simulator.TissueMixtureSimulator method), [106](#)
- simulate_tissue_mixture()
 - (pyfraglib.simulator.TissueMixtureSimulator method), [91](#)

- smooth_wps() (in module pyfraglib.scores), [79](#)
- start_pos (pyfraglib.fragment.Fragment attribute), [49](#)

T

- task0() (in module pyfraglib.fragment), [54](#)
- task1() (in module pyfraglib.fragment), [55](#)
- tf_binding_sites
 - (pyfraglib.simulator.fragment_simulator.ChromatinState attribute), [96](#)
- TissueMixtureSimulator (class in pyfraglib.simulator), [90](#)
- TissueMixtureSimulator (class in pyfraglib.simulator.tissue_mixture_simulator), [105](#)
- TissueProfile (class in pyfraglib.simulator.tissue_mixture_simulator), [103](#)
- to_frag_file() (pyfraglib.fragment.FragmentList method), [56](#)
- to_frag_files() (pyfraglib.fragment.FragmentCollection method), [59](#)
- to_interval_table() (pyfraglib.fragment.FragmentList method), [57](#)
- TRI_NUC_MEAN
 - (pyfraglib.simulator.fragment_simulator.FragmentSimulator attribute), [100](#)
- TRI_NUC_MEAN (pyfraglib.simulator.FragmentSimulator attribute), [85](#)
- TRI_NUC_STD
 - (pyfraglib.simulator.fragment_simulator.FragmentSimulator attribute), [100](#)
- TRI_NUC_STD (pyfraglib.simulator.FragmentSimulator attribute), [85](#)

W

- windowed_protection_score() (in module pyfraglib.scores), [76](#)
- windowed_protection_score_fast() (in module pyfraglib.scores), [77](#)
- WPS_DETREND_WIN (in module pyfraglib.scores), [75](#)
- WPS_NUCL_PERIOD (in module pyfraglib.scores), [76](#)
- WPS_PEAK_MIN_DIST (in module pyfraglib.scores), [75](#)
- wps_power_spectrum_plot() (in module pyfraglib.scores), [83](#)
- WPS_PROM_FRAC (in module pyfraglib.scores), [76](#)
- wps_similarity_metrics() (in module pyfraglib.scores), [80](#)
- WPS_SMOOTH_POLY (in module pyfraglib.scores), [75](#)
- WPS_SMOOTH_WIN (in module pyfraglib.scores), [75](#)
- write_gmm_params() (in module pyfraglib.lengths), [72](#)