SignalFrame: Fast and Memory-Efficient Signal Quantification from BigWig Files for Large-Scale Genomic Region Analysis

Ryan Clark[1,2,3*]

[1]Howard Hughes Medical Institute and Program in Cellular and Molecular Medicine, Boston Children's Hospital, MA 02115, USA
[2]Department of Biological Chemistry and Molecular Pharmacology, Harvard Medical School, Boston, MA 02115, USA
[3]Lead Contact

*Correspondence:
Ryan.clark@childrens.harvard.edu

Abstract

Motivation

Quantifying signal intensity across genomic regions is a fundamental step in genomic data analysis, underpinning tasks such as enhancer detection from ChIP-seq, chromatin accessibility from ATAC-seq, gene expression from RNA-seq, and so much more. Despite the availability of tools such as BEDTools, deepTools, and pyBigWig, researchers face recurring challenges: high memory usage, slow runtimes on large files, difficulty working with multiple BigWig tracks in parallel, and lack of integration with downstream statistical workflows in Python. Researchers are frequently forced to write ad hoc scripts or convert formats just to perform simple signal extraction—often at the cost of excessive memory usage or runtime. A scalable, memory-efficient, and Python-native solution is needed to streamline these signal quantification tasks and support fast iteration in exploratory and production-scale genomics workflows.

Results

We present SignalFrame, a Python package for fast and scalable quantification of genomic signals from BigWig files across BED-defined regions. SignalFrame efficiently computes per-region signal values—including area under the curve (AUC), mean, max, and other statistics—across genomic intervals, with support for simultaneous extraction from multiple BigWig tracks and built-in optimization for performance. Outputs are returned as pandas DataFrames, enabling seamless integration with Python-based statistical and visualization workflows. Key features include dynamic merge-aware interval collapsing to minimize redundant BigWig queries, adaptive slack estimation for efficient region grouping, memory-efficient signal extraction via chunked access, and native support for multi-track comparative analysis. Designed for high-throughput workflows, SignalFrame enables rapid and reproducible signal quantification across millions of regions and large-scale genomic datasets.

1. Introduction

The growing volume of high-throughput sequencing data has made genome-wide signal tracks—typically stored in BigWig format—a cornerstone of functional genomics. These tracks quantify signals such as chromatin accessibility, histone modifications, transcription factor occupancy, and gene expression across the genome. In many workflows, including differential accessibility testing, motif enrichment, and epigenomic profiling, researchers must summarize signal values across predefined genomic regions stored in BED files. This is often accomplished by computing metrics such as area under the curve (AUC), mean, or maximum signal intensity within each interval.

Despite the ubiquity of this task, current tools for signal quantification face critical limitations. BEDTools[1], though widely adopted, is slow and memory-intensive at scale. pyBigWig[2] allows programmatic access in Python but lacks efficient support for batching, merging, or parallelizing queries. deepTools[3] offers high-quality visualization tools but is not optimized for extracting AUC across large numbers of intervals. These constraints become particularly acute in large-scale studies involving single-cell data, multi-condition experiments, or hundreds of epigenomic datasets, where performance bottlenecks and memory overhead can render certain analyses impractical.

To address these limitations, we developed SignalFrame, a fast and scalable Python package for extracting signal statistics from BigWig files across BED-defined regions. SignalFrame supports a broad range of per-region summary methods—including AUC, mean, max, median, standard deviation, and coverage—and performs automated, merge-aware interval batching to minimize redundant I/O operations. Adaptive slack estimation enables dynamic grouping of nearby intervals for speed improvements, while chunked access ensures memory efficiency. Outputs are returned as structured pandas DataFrames[4], facilitating seamless integration with Python-based statistical modeling, normalization routines, and signal visualization. Together, these features position SignalFrame as a flexible, high-performance toolkit for signal quantification in high-throughput genomics.

2. Methods

2.1 Region-Level Signal Extraction from BigWig Files

SignalFrame computes quantitative signal summaries from BigWig-formatted genomic tracks over user-defined regions. Input regions are specified in standard BED format, either as a file or a pandas.DataFrame[4] with 'chr', 'start', and 'end' columns. Using pyBigWig[2], the tool retrieves per-base signal values with built-in bounds checking to ensure valid chromosome coordinates. Supported summary statistics include AUC, mean, max, min, median, standard deviation, coverage (non-zero count), and non-zero mean. Regions are processed in chromosome-sorted order to minimize disk I/O, and results are returned as new columns in the original DataFrame, ready for downstream analysis in Python-based workflows.

2.2 Merge-Aware Interval Collapsing

To reduce redundant BigWig access in densely annotated regions, SignalFrame implements a merge-aware collapsing strategy. When multiple nearby regions are separated by short gaps, the

tool temporarily merges them before querying the BigWig file. After retrieving the signal from the merged span, SignalFrame maps the results back to each original interval based on its relative overlap. This merge-aware strategy significantly improves performance in high-density datasets. The merging threshold is determined automatically using a data-driven heuristic based on inter-region distances (see Section 2.3).

2.3 Adaptive Slack Estimation

The maximum allowable distance between adjacent regions for merging—termed the *slack*—is estimated automatically by SignalFrame. The algorithm computes the distribution of gaps between consecutive intervals and selects a slack value corresponding to 50% of the median positive gap, bounded within a user-safe range.

This adaptive slack estimation enables consistent performance across BED files with varying region densities, and the computed slack is applied independently per chromosome group to reflect local structure.

2.4 Multi-Track Signal Extraction

SignalFrame supports simultaneous extraction of signal values across multiple BigWig files using a single interface. When provided with a list of BigWig paths, the tool computes the specified summary statistic for each track independently while sharing interval batching and merging operations to maximize efficiency. Results are returned in a single DataFrame, with one row per region and one column per track-statistic pair, streamlining comparisons across experimental conditions, replicates, or time points, eliminating the need for external wrapper scripts.

2.5 Normalization and Enrichment Comparison

SignalFrame includes built-in methods for normalizing signal values and computing enrichment across tracks. Supported normalization options include length normalization, Z-score transformation, log2 scaling with a pseudocount, min-max scaling, and quantile normalization to a mean or median reference. For pairwise comparisons, users can compute absolute differences, fold changes, log2 fold changes, and percent changes. All operations are applied directly to the output DataFrame and integrate seamlessly with standard Python analysis tools.

2.6 Statistical Testing

SignalFrame provides built-in wrappers for common statistical tests on region-level signals. Users can perform unpaired t-tests or Mann–Whitney U tests[5] to compare two groups, as well as one-way or two-way ANOVA[6] to assess differences across categorical factors. These functions operate directly on the output DataFrame using user-supplied group labels, and return p-values or ANOVA tables for straightforward interpretation and integration with downstream analysis.

2.7 Visualization

SignalFrame offers tools for visualizing genomic signals from BigWig files, including region-specific line plots, stacked signal plots across BED-defined intervals, and distribution plots for group comparisons. Visualizations are built with matplotlib[7], support customization, and are Jupyter-compatible.

BigWig size	Tool	Time (10k/100k/1M) [s]	Peak Memory Usage [MB]	Correlation with SignalFrame
150 MB	SignalFrame	1.79 / 8.13 / 51.66	59 / 80 / 280	Ref
150 MB	PyBigWig	2.18 / 12.84 / 117.98	61 / 78 / 265	1.0
150 MB	Bedtools	6.35 / 8.28 / 26.49	9,490 / 9,490 / 9,490	1.0
150 MB	Deeptools	1.02 / 8.27 / 80.78	40 / 135 / 1,060	1.0
800 MB	SignalFrame	2.10 / 11.34 / 70.31	63 / 83 / 283	Ref
800 MB	PyBigWig	2.46 / 15.83 / 148.32	66 / 83 / 265	1.0
800 MB	Bedtools	31.42 / 37.16 / 93.66	48,850 / 48,850 / 48,850	1.0
800 MB	Deeptools	1.08 / 9.00 / 87.34	40 / 135 / 1,060	1.0
1600 MB	SignalFrame	1.80 / 7.98 / 52.62	72 / 91 / 291	Ref
1600 MB	PyBigWig	2.24 / 12.62 / 113.30	74 / 90 / 265	1.0
1600 MB	Bedtools	93.22 / 110.91 / 242.81	141,560 / 141,560 / 141,560	1.0
1600 MB	Deeptools	1.09 / 8.63 / 86.17	41 / 135 / 1,060	1.0

Table 1. Benchmarking runtime and memory usage of SignalFrame versus existing tools for genomic signal extraction.

Runtime (in seconds) and peak memory usage (in MB or GB) are reported for SignalFrame, pyBigWig (Python script), bedtools (with AWK wrapper), and deeptools (via computeMatrix scale-regions). Three BigWig datasets were used: a small Treg ATAC-seq file (150MB), a medium H3K27ac ChIP-seq file (800MB), and a large FoxP3 ChIP-seq file (1.6GB, union of Redensky and Dixon). For each BigWig, signals were computed over 10k, 100k, and 1M genomic intervals sampled from the Simple Repeats track of the mm10 genome (UCSC Genome Browser). All tools produced identical signal outputs (Pearson = 1.0). Benchmarks were run on the Harvard O2 cluster using a shared networked file system. “Ref” indicates that SignalFrame was used as the reference for correlation comparisons.

3. Results

We benchmarked SignalFrame against three widely used tools—pyBigWig[2], bedtools[1], and deeptools[3]—for computing region-level signal from BigWig files. Benchmarks were run on a shared high-performance computing cluster with a networked file system (Harvard O2[8]), as tools like bedtools required over 100 GB of memory—making local execution infeasible for fair comparison. We used real datasets of varying size relating to the mm10 genome: a 150MB Treg

ATAC-seq file[9], an 800MB H3K4me3 ChIP-seq file[10], and a 1.6GB FoxP3 ChIP-seq file (union of Rudensky[9] and Dixon[11]). BED intervals were sampled from the Simple Repeats annotation based on Tandem Repeats Finder[12] to reflect realistic genomic regions in the mm10 genome.

Across all file sizes and region counts (10k, 100k, 1M), SignalFrame demonstrated consistently strong performance. For large-scale queries (1M regions), it outperformed pyBigWig and bedtools in runtime by up to 2.2× and 6.5×, respectively, while maintaining similar performance to deeptools. SignalFrame achieved this with low memory usage—comparable to pyBigWig, significantly lower than deeptools, and dramatically lower than bedtools, which consumed over 140GB for the largest dataset.

Although all four tools ultimately produced identical signal values (Pearson correlation = 1.0), each competitor required custom scripting to calculate signal over regions: a Python wrapper for pyBigWig, and shell-based AWK workflows for bedtools and deeptools. In contrast, SignalFrame is purpose-built for this task, requiring no additional scripting. The ability to scale to millions of regions with minimal memory, while preserving ease of use and correctness, makes SignalFrame a robust and practical solution for genomic signal extraction.

Data Availability

The software is available as a Python package via PyPI and can be installed using pip. Source code, documentation, and example usage are hosted on GitHub at https://github.com/clarkvd/SignalFrame. All code is released under an open-source license to support transparency and reproducibility.

Acknowledgements

The author thanks Xi (Dylan) Wang and members of the Hur lab for helpful feedback and discussion during the development of this project.

References

1. Quinlan, A. R. & Hall, I. M. BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics* **26**, 841-842, doi:10.1093/bioinformatics/btq033 (2010).
2. Ramírez, F. pyBigWig: Python interface for BigWig files. GitHub, https://github.com/deeptools/pyBigWig (2016).
3. Ramirez, F. *et al.* deepTools2: a next generation web server for deep-sequencing data analysis. *Nucleic Acids Res* **44**, W160-165, doi:10.1093/nar/gkw257 (2016).
4. McKinney, W. Data structures for statistical computing in Python. *Proceedings of the 9th Python in Science Conference* 51–56 (2010).
5. Virtanen, P. et al. SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nat. Methods* 17, 261–272, doi:10.1038/s41592-019-0686-2 (2020).
6. Seabold, S. & Perktold, J. Statsmodels: Econometric and statistical modeling with Python. *Proceedings of the 9th Python in Science Conference* 57–61 (2010).
7. Hunter, J. D. Matplotlib: A 2D graphics environment. *Comput. Sci. Eng.* 9, 90–95, doi:10.1109/MCSE.2007.55 (2007).

8. Harvard FAS Research Computing. O2 High Performance Compute Cluster. https://www.rc.fas.harvard.edu/ (accessed 2024).
9. Kitagawa, Y. *et al.* Guidance of regulatory T cell development by Satb1-dependent super-enhancer establishment. *Nat Immunol* **18**, 173-183, doi:10.1038/ni.3646 (2017).
10. ENCODE Project Consortium. File ENCFF649MFI from ChIP-seq of H3K4me3 in mouse hippocampus (ENCSR213YKJ). https://www.encodeproject.org/files/ENCFF649MFI/ (2022).
11. Samstein, R. M. *et al.* Foxp3 exploits a pre-existent enhancer landscape for regulatory T cell lineage specification. *Cell* **151**, 153-166, doi:10.1016/j.cell.2012.06.053 (2012).
12. Benson, G. Tandem repeats finder: a program to analyze DNA sequences. *Nucleic Acids Res.* 27, 573–580, doi:10.1093/nar/27.2.573 (1999).

Supplementary Data

PyBigWig

```
def load_bed(bed_path):
    df = pd.read_csv(bed_path, sep='\t', header=None)
    if df.shape[1] < 3:
        raise ValueError("BED file must have at least 3 columns")
    df.columns = ['chr', 'start', 'end'] + [f'col{i}' for i in range(4, df.shape[1] + 1)]
    return df

def get_naive_auc(bigwig_path, bed_df):
    bw = pyBigWig.open(bigwig_path)
    aucs = []
    pbar = tqdm(total=len(bed_df), desc="Computing naive AUC")

    for _, row in bed_df.iterrows():
        chrom, start, end = row['chr'], int(row['start']), int(row['end'])

        if chrom not in bw.chroms():
            aucs.append(0.0)
            pbar.update(1)
            continue

        chrom_len = bw.chroms()[chrom]
        start = max(0, start)
        end = min(end, chrom_len)

        if start >= end:
            aucs.append(0.0)
            pbar.update(1)
            continue

        try:
            values = bw.values(chrom, start, end, numpy=True)
            values = np.nan_to_num(values, nan=0.0)
            auc = np.sum(values)
        except RuntimeError:
            auc = 0.0

        aucs.append(auc)
        pbar.update(1)

    pbar.close()
    bw.close()
    return np.array(aucs)
```

SignalFrame:
/usr/bin/time -v python signalframe.py ATAC_tTreg.bw 10k_regions.bed
/usr/bin/time -v python signalframe.py ATAC_tTreg.bw 100k_regions.bed
/usr/bin/time -v python signalframe.py ATAC_tTreg.bw 1M_regions.bed
/usr/bin/time -v python signalframe.py H3K4me3 _ChIP.bw 10k_regions.bed

/usr/bin/time -v python signalframe.py H3K4me3 _ChIP.bw 100k_regions.bed
/usr/bin/time -v python signalframe.py H3K4me3 _ChIP.bw 1M_regions.bed
/usr/bin/time -v python signalframe.py FoxP3_ChIP.bw 10k_regions.bed
/usr/bin/time -v python signalframe.py FoxP3_ChIP.bw 100k_regions.bed
/usr/bin/time -v python signalframe.py FoxP3_ChIP.bw 1M_regions.bed

PyBigWig:
/usr/bin/time -v python pybigwig.py ATAC_tTreg.bw 10k_regions.bed
/usr/bin/time -v python pybigwig.py ATAC_tTreg.bw 100k_regions.bed
/usr/bin/time -v python pybigwig.py ATAC_tTreg.bw 1M_regions.bed
/usr/bin/time -v python pybigwig.py H3K4me3 _ChIP.bw 10k_regions.bed
/usr/bin/time -v python pybigwig.py H3K4me3 _ChIP.bw 100k_regions.bed
/usr/bin/time -v python pybigwig.py H3K4me3 _ChIP.bw 1M_regions.bed
/usr/bin/time -v python pybigwig.py FoxP3_ChIP.bw 10k_regions.bed
/usr/bin/time -v python pybigwig.py FoxP3_ChIP.bw 100k_regions.bed
/usr/bin/time -v python pybigwig.py FoxP3_ChIP.bw 1M_regions.bed

Bedtools:
/usr/bin/time -v bash -c 'bedtools intersect -a 10k_regions.bed -b ATAC_tTreg.bedgraph -wa -wb | awk -v OFMT="%.6f" '\"BEGIN {OFS="\t"} {overlap_start = ($2 > $5) ? $2 : $5; overlap_end = ($3 < $6) ? $3 : $6; overlap_len = (overlap_end > overlap_start) ? (overlap_end - overlap_start) : 0; auc = overlap_len * $7; key = $1 "\t" $2 "\t" $3; sum[key] += auc;} END {for (k in sum) print k, sum[k]}'\" > 10k_bedtools.bed'
/usr/bin/time -v bash -c 'bedtools intersect -a 100k_regions.bed -b ATAC_tTreg.bedgraph -wa -wb | awk -v OFMT="%.6f" '\"BEGIN {OFS="\t"} {overlap_start = ($2 > $5) ? $2 : $5; overlap_end = ($3 < $6) ? $3 : $6; overlap_len = (overlap_end > overlap_start) ? (overlap_end - overlap_start) : 0; auc = overlap_len * $7; key = $1 "\t" $2 "\t" $3; sum[key] += auc;} END {for (k in sum) print k, sum[k]}'\" > 100k_bedtools.bed'
/usr/bin/time -v bash -c 'bedtools intersect -a 1M_regions.bed -b ATAC_tTreg.bedgraph -wa -wb | awk -v OFMT="%.6f" '\"BEGIN {OFS="\t"} {overlap_start = ($2 > $5) ? $2 : $5; overlap_end = ($3 < $6) ? $3 : $6; overlap_len = (overlap_end > overlap_start) ? (overlap_end - overlap_start) : 0; auc = overlap_len * $7; key = $1 "\t" $2 "\t" $3; sum[key] += auc;} END {for (k in sum) print k, sum[k]}'\" > 1M_bedtools.bed'
/usr/bin/time -v bash -c 'bedtools intersect -a 10k_regions.bed -b H3K4me3 _ChIP.bedgraph -wa -wb | awk -v OFMT="%.6f" '\"BEGIN {OFS="\t"} {overlap_start = ($2 > $5) ? $2 : $5; overlap_end = ($3 < $6) ? $3 : $6; overlap_len = (overlap_end > overlap_start) ? (overlap_end - overlap_start) : 0; auc = overlap_len * $7; key = $1 "\t" $2 "\t" $3; sum[key] += auc;} END {for (k in sum) print k, sum[k]}'\" > 10k_bedtools.bed'
/usr/bin/time -v bash -c 'bedtools intersect -a 100k_regions.bed -b H3K4me3 _ChIP.bedgraph -wa -wb | awk -v OFMT="%.6f" '\"BEGIN {OFS="\t"} {overlap_start = ($2 > $5) ? $2 : $5; overlap_end = ($3 < $6) ? $3 : $6; overlap_len = (overlap_end > overlap_start) ? (overlap_end - overlap_start) : 0; auc = overlap_len * $7; key = $1 "\t" $2 "\t" $3; sum[key] += auc;} END {for (k in sum) print k, sum[k]}'\" > 100k_bedtools.bed'
/usr/bin/time -v bash -c 'bedtools intersect -a 1M_regions.bed -b H3K4me3 _ChIP.bedgraph -wa -wb | awk -v OFMT="%.6f" '\"BEGIN {OFS="\t"} {overlap_start = ($2 > $5) ? $2 : $5; overlap_end = ($3 < $6) ? $3 : $6; overlap_len = (overlap_end > overlap_start) ? (overlap_end -

overlap_start) : 0; auc = overlap_len * $7; key = $1 "\t" $2 "\t" $3; sum[key] += auc;} END {for (k in sum) print k, sum[k]}'\" > 1M_bedtools.bed'
/usr/bin/time -v bash -c 'bedtools intersect -a 10k_regions.bed -b FoxP3_ChIP.bedgraph -wa -wb | awk -v OFMT="%.6f" '\"BEGIN {OFS="\t"} {overlap_start = ($2 > $5) ? $2 : $5; overlap_end = ($3 < $6) ? $3 : $6; overlap_len = (overlap_end > overlap_start) ? (overlap_end - overlap_start) : 0; auc = overlap_len * $7; key = $1 "\t" $2 "\t" $3; sum[key] += auc;} END {for (k in sum) print k, sum[k]}'\" > 10k_bedtools.bed'
/usr/bin/time -v bash -c 'bedtools intersect -a 100k_regions.bed -b FoxP3_ChIP.bedgraph -wa -wb | awk -v OFMT="%.6f" '\"BEGIN {OFS="\t"} {overlap_start = ($2 > $5) ? $2 : $5; overlap_end = ($3 < $6) ? $3 : $6; overlap_len = (overlap_end > overlap_start) ? (overlap_end - overlap_start) : 0; auc = overlap_len * $7; key = $1 "\t" $2 "\t" $3; sum[key] += auc;} END {for (k in sum) print k, sum[k]}'\" > 100k_bedtools.bed'
/usr/bin/time -v bash -c 'bedtools intersect -a 1M_regions.bed -b FoxP3_ChIP.bedgraph -wa -wb | awk -v OFMT="%.6f" '\"BEGIN {OFS="\t"} {overlap_start = ($2 > $5) ? $2 : $5; overlap_end = ($3 < $6) ? $3 : $6; overlap_len = (overlap_end > overlap_start) ? (overlap_end - overlap_start) : 0; auc = overlap_len * $7; key = $1 "\t" $2 "\t" $3; sum[key] += auc;} END {for (k in sum) print k, sum[k]}'\" > 1M_bedtools.bed'

Deeptools:
/usr/bin/time -v multiBigwigSummary BED-file --bwfiles ATAC_tTreg.bw --BED 10k_regions.bed --outFileName deeptools_10k.npz --outRawCounts deeptools_10k.tmp.tsv && awk 'BEGIN {OFS="\t"} !/^#/ {region_len = $3 - $2; auc = $4 * region_len; print $1, $2, $3, $4, region_len, auc}' deeptools_10k.tmp.tsv > deeptools_10k_with_auc.tsv && rm deeptools_10k.tmp.tsv
/usr/bin/time -v multiBigwigSummary BED-file --bwfiles ATAC_tTreg.bw --BED 100k_regions.bed --outFileName deeptools_100k.npz --outRawCounts deeptools_100k.tmp.tsv && awk 'BEGIN {OFS="\t"} !/^#/ {region_len = $3 - $2; auc = $4 * region_len; print $1, $2, $3, $4, region_len, auc}' deeptools_100k.tmp.tsv > deeptools_100k_with_auc.tsv && rm deeptools_100k.tmp.tsv
/usr/bin/time -v multiBigwigSummary BED-file --bwfiles ATAC_tTreg.bw --BED 1M_regions.bed --outFileName deeptools_1M.npz --outRawCounts deeptools_1M.tmp.tsv && awk 'BEGIN {OFS="\t"} !/^#/ {region_len = $3 - $2; auc = $4 * region_len; print $1, $2, $3, $4, region_len, auc}' deeptools_1M.tmp.tsv > deeptools_1M_with_auc.tsv && rm deeptools_1M.tmp.tsv
/usr/bin/time -v multiBigwigSummary BED-file --bwfiles H3K4me3 _ChIP.bw --BED 10k_regions.bed --outFileName deeptools_10k.npz --outRawCounts deeptools_10k.tmp.tsv && awk 'BEGIN {OFS="\t"} !/^#/ {region_len = $3 - $2; auc = $4 * region_len; print $1, $2, $3, $4, region_len, auc}' deeptools_10k.tmp.tsv > deeptools_10k_with_auc.tsv && rm deeptools_10k.tmp.tsv
/usr/bin/time -v multiBigwigSummary BED-file --bwfiles H3K4me3 _ChIP.bw --BED 100k_regions.bed --outFileName deeptools_100k.npz --outRawCounts deeptools_100k.tmp.tsv && awk 'BEGIN {OFS="\t"} !/^#/ {region_len = $3 - $2; auc = $4 * region_len; print $1, $2, $3, $4, region_len, auc}' deeptools_100k.tmp.tsv > deeptools_100k_with_auc.tsv && rm deeptools_100k.tmp.tsv
/usr/bin/time -v multiBigwigSummary BED-file --bwfiles H3K4me3 _ChIP.bw --BED 1M_regions.bed --outFileName deeptools_1M.npz --outRawCounts deeptools_1M.tmp.tsv &&

```
awk 'BEGIN {OFS="\t"} !/^#/ {region_len = $3 - $2; auc = $4 * region_len; print $1, $2, $3, $4, region_len, auc}' deeptools_1M.tmp.tsv > deeptools_1M_with_auc.tsv && rm deeptools_1M.tmp.tsv
/usr/bin/time -v multiBigwigSummary BED-file --bwfiles FoxP3_ChIP.bw --BED 10k_regions.bed --outFileName deeptools_10k.npz --outRawCounts deeptools_10k.tmp.tsv && awk 'BEGIN {OFS="\t"} !/^#/ {region_len = $3 - $2; auc = $4 * region_len; print $1, $2, $3, $4, region_len, auc}' deeptools_10k.tmp.tsv > deeptools_10k_with_auc.tsv && rm deeptools_10k.tmp.tsv
/usr/bin/time -v multiBigwigSummary BED-file --bwfiles FoxP3_ChIP.bw --BED 100k_regions.bed --outFileName deeptools_100k.npz --outRawCounts deeptools_100k.tmp.tsv && awk 'BEGIN {OFS="\t"} !/^#/ {region_len = $3 - $2; auc = $4 * region_len; print $1, $2, $3, $4, region_len, auc}' deeptools_100k.tmp.tsv > deeptools_100k_with_auc.tsv && rm deeptools_100k.tmp.tsv
/usr/bin/time -v multiBigwigSummary BED-file --bwfiles FoxP3_ChIP.bw --BED 1M_regions.bed --outFileName deeptools_1M.npz --outRawCounts deeptools_1M.tmp.tsv && awk 'BEGIN {OFS="\t"} !/^#/ {region_len = $3 - $2; auc = $4 * region_len; print $1, $2, $3, $4, region_len, auc}' deeptools_1M.tmp.tsv > deeptools_1M_with_auc.tsv && rm deeptools_1M.tmp.tsv
```