Package 'cancerSimCraft'

March 5, 2025

Title Cancer Evolution Simulation Toolkit

Version 0.2.0

Description A package for simulating cancer genome evolution including copy number changes, structural variations, and whole genome duplications.

Depends R (>= 4.3.1)

Imports Biostrings,
igraph,
karyoploteR,
dplyr,
tidyr,
ggplot2,
reshape2,
IRanges,
ComplexHeatmap,
parallel

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

LazyData true

Contents

`acquire_clone_mutations` *Collect All Mutations for Each Clone in Tree*

Description

Gathers all mutations that occur along the evolutionary path from root to each clone in the phylogenetic tree. This includes mutations from all ancestral branches.

Usage

```
acquire_clone_mutations(mutation_table, tree)
```

Arguments

`mutation_table`	A data frame containing mutation information with columns: • edge_name - Tree edge identifier in format "parent_child" • other mutation-specific columns
`tree`	An igraph object representing the phylogenetic tree structure

Details

The function:

1. Identifies the root node and all other clones in the tree
2. For each clone:
   - Finds all edges in the path from root to clone
   - Collects mutations occurring on these edges
   - Returns mutations in their original table format

Value

A list where:

- Names are clone names (excluding root)
- Values are data frames containing all mutations affecting each clone, including those inherited from ancestors

See Also

`get_edges_between_clones`

Examples

```
## Not run:
# For a tree with path Root -> A -> B
clone_muts <- acquire_clone_mutations(mutation_table, tree)
# clone_muts$B will contain mutations from both Root->A and A->B edges

## End(Not run)
```

`calculate_window_sizes` *Calculate Sizes of Genomic Windows*

Description

Calculates the size (length in base pairs) of each genomic window from a vector of window identifiers in "chr_start_end" format.

Usage

```
calculate_window_sizes(genome_window_vector)
```

Arguments

`genome_window_vector`	A character vector of window identifiers in the format "chr_start_end" (e.g., "chr1_1000_2000")

Details

The function:

1. Splits each window identifier into its components
2. Extracts start and end positions
3. Calculates window size as the difference between end and start

Value

A numeric vector containing the size of each window in base pairs, calculated as (end position - start position)

Note

Assumes window identifiers are properly formatted with underscore separators and numeric start/end positions

Examples

```
## Not run:
windows <- c("chr1_1000_2000", "chr1_2000_5000", "chr2_1000_3000")
sizes <- calculate_window_sizes(windows)
# Returns: c(1000, 3000, 2000)

## End(Not run)
```

calc_clone_snv_num *Calculate Number of SNVs in a Clone*

Description

Calculates the expected number of single nucleotide variants (SNVs) accumulated in a clone over time using a simple linear model where the number of mutations is proportional to time, genome length, and a base mutation rate.

Usage

```
calc_clone_snv_num(time, genome_length, mutation_rate)
```

Arguments

time	Numeric value representing time (units should be consistent with mutation_rate)
genome_length	Numeric value representing the length of the genome in base pairs
mutation_rate	Numeric value representing the mutation rate per base pair per time unit

Details

The function implements a simple linear model where the number of mutations is directly proportional to time, genome length, and mutation rate. The result is rounded to the nearest integer.

Value

Integer number of mutations, rounded to the nearest whole number

Examples

```
# Calculate mutations for 100 time units, genome length of 1e6, and mutation rate of 1e-8
calc_clone_snv_num(100, 1e6, 1e-8)
```

check_alt_snp_match *Check Alternative Allele Matches Between SNP List and Simulated Genome*

Description

This function verifies that the alternate alleles (ALT) in the SNP list match the corresponding positions in the synthetic genome. It ensures that the SNPs have been correctly inserted into the synthetic genome for both maternal and paternal haplotypes. The function checks each segment (e.g., chromosome) and issues warnings if mismatches are found.

Usage

```
check_alt_snp_match(seg_names, snp_list, sim_genome)
```

Arguments

seg_names	Character vector of segment/chromosome names to check
snp_list	A nested list where first level contains segment names and second level contains data frames for different genotypes ('1|1', '1|0', '0|1') with POS and ALT columns
sim_genome	A nested list containing maternal and paternal sequences for each segment, structured as sim_genome$maternal[seg_name] and sim_genome$paternal[seg_name]

Details

For each segment and haplotype, the function:

1. Combines appropriate homozygous and heterozygous SNPs
2. Removes SNPs with duplicated positions
3. Extracts sequences at SNP positions from simulated genome
4. Compares extracted sequences with expected alternative alleles
5. Reports matches/mismatches via messages and warnings

Maternal haplotype checks include '1|1' and '1|0' variants, while paternal haplotype checks include '1|1' and '0|1' variants.

Value

No return value, called for side effects:

- Prints confirmation message for each matching segment and haplotype
- Issues warning if mismatches are found

Examples

```
## Not run:
# Example simulated genome
sim_genome <- list(
  maternal = list(chr1 = DNAStringSet("GCTGACTGACTG")),
  paternal = list(chr1 = DNAStringSet("ACTGACTCACTG"))
)

# Example SNP list
snp_list <- list(
  chr1 = list(
    "1|1" = data.frame(POS = 1, ALT = "G"),
    "1|0" = data.frame(POS = 5, ALT = "T"),
    "0|1" = data.frame(POS = 8, ALT = "C")
  )
)

check_alt_snp_match("chr1", snp_list, sim_genome)

## End(Not run)
```

`check_genome_chr_length` *Check Chromosome Lengths in Simulated Genome*

Description

Validates that the actual chromosome lengths in a simulated genome match their expected lengths. Checks both maternal and paternal copies of each chromosome.

Usage

```
check_genome_chr_length(clone_genome, expected_chr_lengths)
```

Arguments

`clone_genome`	A nested list containing genome sequences: • First level: haplotypes (maternal/paternal) • Second level: chromosomes with nucleotide sequences
`expected_chr_lengths`	A nested list containing expected chromosome lengths: • First level: haplotypes (maternal/paternal) • Second level: named numeric vector of chromosome lengths

Details

The function:

1. Checks each chromosome in both maternal and paternal copies
2. Compares actual sequence length to expected length
3. Generates informative messages about matches and mismatches
4. Useful for validating genome synthesis results

Value

A nested list where:

- First level: haplotypes (maternal/paternal)
- Second level: chromosomes
- Values are status messages indicating whether lengths match or not, including both expected and actual lengths when there's a mismatch

Examples

```
## Not run:
validation_results <- check_genome_chr_length(
  clone_genome = synthesized_genome,
  expected_chr_lengths = expected_lengths
)
# Access results for specific chromosomes
validation_results$maternal$chr1  # Check maternal chr1 status

## End(Not run)
```

`check_genome_mutations` *Validate Mutations in Simulated Genome*

Description

Verifies that mutations in the simulated genome match their expected alternative nucleotides, accounting for lost segments that should contain 'N's.

Usage

```
check_genome_mutations(clone_genome, clone_mutation_table, clone_seg_list)
```

Arguments

`clone_genome` A nested list containing genome sequences:
- First level: haplotypes (maternal/paternal)
- Second level: chromosomes with nucleotide sequences

`clone_mutation_table` A data frame containing mutation information:
- haplotype - Maternal or paternal copy
- chrom - Chromosome name
- pos - Mutation position
- alternative_nt - Expected nucleotide after mutation

`clone_seg_list` A nested list containing segment information:
- First level: haplotypes
- Second level: data frame with segment details including loss information

Details

The function:

1. Processes mutations chromosome by chromosome
2. Checks if mutations fall in lost segments (should be 'N')
3. Compares actual nucleotides to expected ones
4. Provides detailed output only for incorrect mutations
5. Handles special cases:
   - Mutations in lost segments
   - No mutations in a chromosome

Value

A nested list where:

- First level: haplotypes
- Second level: chromosomes
- Values are either:
  - Success message if all mutations are correct
  - Data frame of incorrect mutations with details for investigation

Examples

```
## Not run:
validation_results <- check_genome_mutations(
  clone_genome = synthesized_genome,
  clone_mutation_table = mutations,
  clone_seg_list = segments
)
# Check results for specific chromosome
validation_results$maternal$chr1

## End(Not run)
```

---

check_loss_segments *Check Nucleotide Content in Lost Segments*

Description

Validates that genomic regions marked as lost (CN_change = -1) contain only 'N' nucleotides in the simulated genome sequence. This ensures proper handling of deletion events.

Usage

```
check_loss_segments(clone_genome, clone_segments, verbose = TRUE)
```

Arguments

`clone_genome` A nested list containing genome sequences:

- First level: haplotypes (maternal/paternal)
- Second level: chromosomes with nucleotide sequences

`clone_segments` A nested list containing segment information:

- First level: haplotypes (maternal/paternal)
- Second level: data frame with columns:
  - chrom - Chromosome name
  - ori_start - Original start position
  - ori_end - Original end position
  - CN_change - Copy number change (-1 for losses)
  - seg_id - Segment identifier

Details

The function:

1. Identifies segments with CN_change = -1
2. For each lost segment:
   - Extracts the sequence
   - Counts nucleotide frequencies
   - Should show only 'N' nucleotides if loss is correctly implemented
3. Prints message if no loss segments found in a haplotype

Value

A list where:

- Names are in format "haplotype_segment_id"
- Values are nucleotide frequency counts (from Biostrings::alphabetFrequency)
- Empty list if no loss segments are found

Examples

```
## Not run:
# Check lost segments in a clone
loss_check <- check_loss_segments(
  clone_genome = synthesized_genome,
  clone_segments = segment_info
)
# Results should show only 'N' nucleotides in lost regions

## End(Not run)
```

`check_ref_snp_match` *Check Reference Genome and SNP List Compatibility*

Description

This function verifies that the reference nucleotides in the SNP list match the corresponding positions in the reference genome. It is used to ensure that the reference genome and SNP list are compatible before introducing SNPs into the genome.

Usage

```
check_ref_snp_match(seg_names, snp_list, ref_genome)
```

Arguments

`seg_names`	Character vector of segment/chromosome names to check
`snp_list`	A nested list where first level contains segment names and second level contains a data frame named 'all', which includes all SNPs having at least POS and REF columns
`ref_genome`	A DNAStringSet or similar object containing reference sequences, with names matching seg_names

Details

The function performs the following steps for each segment:

1. Creates an IRanges object from SNP positions
2. Extracts reference sequences at those positions
3. Compares extracted sequences with SNP reference alleles
4. Reports matches/mismatches via messages and warnings

Value

No return value, called for side effects:

- Prints confirmation message for each matching segment
- Issues warning if mismatches are found

Examples

```
## Not run:
# Example reference genome
ref_genome <- DNAStringSet(c(
  chr1 = "ACTGACTGACTG",
  chr2 = "GTCAGTCAGTCA"
))

# Example SNP list
snp_list <- list(
  chr1 = list(all = data.frame(POS = c(1,5), REF = c("A","C"))),
  chr2 = list(all = data.frame(POS = c(2,6), REF = c("T","T")))
)

check_ref_snp_match(c("chr1", "chr2"), snp_list, ref_genome)

## End(Not run)
```

---

create_cluster_cytoband_anno

Create Cytoband Annotations for Region Clusters

Description

Maps clustered genomic regions to their corresponding cytoband annotations for both maternal and paternal haplotypes. For each cluster of regions, the function collects and combines the cytoband annotations of all regions within that cluster.

Usage

```
create_cluster_cytoband_anno(region_clusters, cytoband_mapping)
```

Arguments

`region_clusters`
: A nested list structure containing clustered genomic regions. The list is organized by haplotype ("maternal"/"paternal"), then by chromosome, then by cluster number, containing region IDs as the innermost elements.

`cytoband_mapping`
: A list containing cytoband annotations for each genomic region. Each element should be named with a region ID corresponding to those in region_clusters.

Details

The function processes clusters by:

1. Iterating through haplotypes (maternal/paternal)
2. For each chromosome in each haplotype
3. For each cluster in each chromosome
4. Retrieving and combining cytoband annotations for all regions in the cluster
5. Removing duplicate annotations using unique()

Value

A nested list with the same structure as region_clusters, where each cluster contains a data frame of unique cytoband annotations. The structure is: list[haplotype][chromosome][cluster] -> data frame of cytoband annotations

See Also

Related functions for genomic region analysis and cytoband mapping

Examples

```
## Not run:
# Assuming region_clusters and cytoband_mapping are already defined:
cluster_annotations <- create_cluster_cytoband_anno(
  region_clusters = region_clusters,
  cytoband_mapping = cytoband_mapping
)

# Access annotations for a specific cluster:
maternal_chr1_cluster1 <- cluster_annotations[["maternal"]][["chr1"]][[1]]

## End(Not run)
```

---

create_edge_event_table

Create Edge Event Table from Event Table and Tree

Description

This function generates an edge event table by combining information from an event table and a tree structure. It creates a table that summarizes events associated with each edge in the tree, including event labels and the number of events per edge.

Usage

```
create_edge_event_table(
  event_table,
  tree,
  anno_cols = c("haplotype_abbr", "region_name", "CN_change")
)
```

Arguments

event_table	A data frame containing copy number events with columns: • parent: Parent node ID • child: Child node ID • haplotype: Haplotype information ("maternal" or "paternal") • region_name: Region identifier • CN_change: Copy number change value • Additional annotation columns as specified in anno_cols
tree	A phylogenetic tree object that can be converted to an edge list using as_edgelist()
anno_cols	A character vector specifying which columns to use for creating event labels. Default is c("haplotype_abbr", "region_name", "CN_change")

Details

The function performs these steps:

1. Converts tree to edge list format
2. Creates abbreviated haplotype labels (M/P)
3. Combines annotation columns into event names
4. Groups events by edge
5. Creates multi-line labels for edges with multiple events
6. Orders results to match tree structure

Value

A data frame summarizing events for each edge in the tree. The returned data frame includes the following columns:

- `edge_name`: A unique identifier for each edge, constructed as `parent_child`.
- `edge_label`: A concatenated string of event labels for the edge, separated by newline characters.
- `n_events`: The number of events associated with the edge.

Note

Event labels are formatted as "HAPLOTYPE:REGION:CN_CHANGE" by default. Multiple events on the same edge are separated by newlines.

Examples

```
## Not run:
events <- data.frame(
  parent = c("A", "A"),
  child = c("B", "C"),
  haplotype = c("maternal", "paternal"),
  region_name = c("chr1p", "chr2q"),
  CN_change = c(1, -1)
)
tree <- your_tree_object  # Replace with actual tree
edge_events <- create_edge_event_table(events, tree)

## End(Not run)
```

`create_event` *Create a Genomic Event Record*

Description

Creates a record of a copy number change event Supports CNV and WGD events

Usage

```
create_event(
  haplotype,
  parent,
  child,
  region_name,
  CN_change,
  copy_index,
  event_type
)
```

Arguments

`haplotype`	Character string specifying the haplotype ("maternal" or "paternal")
`parent`	Character string identifying the parent cell
`child`	Character string identifying the child cell
`region_name`	Character string specifying the chromosome region (e.g., "chr1p", "chr2q")
`CN_change`	Numeric value indicating the copy number change
`copy_index`	Integer specifying which copy of the segment is affected
`event_type`	Character string describing the type of event

Value

A data frame containing a single row with the event information:

- haplotype: Origin of the segment
- parent: Parent cell identifier
- child: Child cell identifier
- region_name: Name of the affected region
- CN_change: Copy number change value
- copy_index: Index of the affected copy
- event_type: Type of genomic event

Examples

```
event <- create_event(
  haplotype = "maternal",
  parent = "C1",
  child = "C2",
  region_name = "chr1p",
  CN_change = 1,
```

```
  copy_index = 1,
  event_type = "CNV"
)
```

`create_initial_seg_list` *Create Initial Segment List from Chromosome Arm Table*

Description

Creates initial genomic segments for both maternal and paternal haplotypes from a chromosome arm table. Handles coordinate system conversion (0-based to 1-based) and initializes tracking metadata for each segment.

Usage

```
create_initial_seg_list(chr_arm_table)
```

Arguments

`chr_arm_table`	A data frame containing chromosome arm information with columns: • chrom: Chromosome name • start: Start position • end: End position • region_name: Identifier for the chromosome arm

Details

The function performs the following operations:

1. Checks and converts coordinates from 0-based to 1-based if necessary
2. Creates two copies of the input table (maternal and paternal)
3. Adds metadata columns for tracking segment evolution
4. Uses dplyr for efficient data manipulation

Value

A list with two elements ("maternal" and "paternal"), each containing a data frame of segments with the following columns:

- All original columns from chr_arm_table
- haplotype: "maternal" or "paternal"
- ref_start: Reference start position
- ref_end: Reference end position
- ori_start: Original start position
- ori_end: Original end position
- copy_index: Set to 1 for initial segments
- seg_id: Segment identifier (format: `<region_name>_1`)
- CN_change: Copy number change (initialized to 0)
- seg_source_edge: Set to "before_root"
- seg_source_event: Set to "base"

Note

Assumes input coordinates are either 0-based or 1-based. Automatically converts 0-based coordinates to 1-based.

Examples

```
## Not run:
chr_arms <- data.frame(
  chrom = c("chr1", "chr1"),
  start = c(0, 1000000),
  end = c(1000000, 2000000),
  region_name = c("chr1p", "chr1q")
)
segments <- create_initial_seg_list(chr_arms)

## End(Not run)
```

---

create_sc_tree_plot *Create a Cell Lineage Tree Visualization*

Description

Creates a dendrogram visualization of cell lineage data from simulation results, showing the relationships between cells and their clonal evolution over time.

Usage

```
create_sc_tree_plot(
  dynamic_sim_ob,
  clone_colors,
  title = "SC Dynamics Dendrogram"
)
```

Arguments

dynamic_sim_ob	List object returned by simulate_sc_dynamics function
clone_colors	Named vector of colors where names match clone types in the simulation
title	Character string for the plot title, defaults to "SC Dynamics Dendrogram"

Details

This function creates a dendrogram visualization of cell lineage data where:

1. The vertical axis represents time
2. Edges represent cells' lifespans
3. Edge colors indicate the clone type of each cell
4. Branch lengths correspond to cell lifetimes

The function first prepares the cell information data using the prepare_tree_data function, then creates a graph object and finally generates a dendrogram visualization using ggraph. The resulting plot has publication-ready styling with customized title, legend, and axis labels.

Value

A ggraph plot object displaying the cell lineage tree with customized styling

See Also

simulate_sc_dynamics, prepare_tree_data

Examples

```
# Define colors for each clone type
clone_colors <- c(A = "blue", B = "red")

# Create and display the lineage tree
tree_plot <- create_sc_tree_plot(dynamic_sim_ob, clone_colors, "Cell Lineage Evolution")
print(tree_plot)

# Save the plot
ggsave("cell_lineage_tree.png", tree_plot, width = 10, height = 8)
```

---

draw_chr_bar *Create Chromosome Bar Annotation for ComplexHeatmap*

Description

Creates a chromosome bar annotation track for ComplexHeatmap visualization, which includes alternating black and grey bars to distinguish chromosomes and chromosome number labels. The function handles both standard (22 autosomes + X, Y) and custom chromosome sets.

Usage

```
draw_chr_bar(window_data)
```

Arguments

window_data	A data frame containing genomic window information with at least one column named 'chr' (case-insensitive) containing chromosome identifiers (e.g., "chr1", "chr2", etc.).

Details

The function performs the following steps:

- Standardizes chromosome column names to lowercase
- Computes chromosome lengths and boundaries
- Creates alternating binary pattern for visual separation
- Calculates mean positions for chromosome labels
- Generates chromosome text annotations

Value

A HeatmapAnnotation object from the ComplexHeatmap package containing:

- A text annotation track with chromosome numbers
- A binary pattern track with alternating black and grey bars

Examples

```
## Not run:
# Create sample window data
window_data <- data.frame(
  chr = rep(paste0("chr", c(1:22, "X")), each = 100),
  start = 1:2300,
  end = 101:2400
)

# Generate chromosome bar annotation
chr_annotation <- draw_chr_bar(window_data)

# Use in ComplexHeatmap
library(ComplexHeatmap)
Heatmap(matrix_data,
        top_annotation = chr_annotation)

## End(Not run)
```

---

extend_blueprint_cn_for_equal_bin

Extend Blueprint Copy Number Data for Equal-Sized Bins

Description

This function extends a copy number (CN) data frame (`seg_dat`) to ensure that each genomic window is divided into equal-sized bins of a specified size (`bin_unit`). It replicates columns in the input data frame based on the size of each window and the desired bin size, creating a new data frame with expanded columns.

Usage

```
extend_blueprint_cn_for_equal_bin(seg_dat, window_sizes, bin_unit)
```

Arguments

`seg_dat`	A data frame of copy number segments where: • Rows represent clones/samples • Columns represent genomic windows • Values represent copy numbers
`window_sizes`	A numeric vector containing the size of each window in base pairs, must match the number of columns in seg_dat
`bin_unit`	The desired size of each bin in base pairs

Details

The function:

1. Calculates how many bins each window needs based on its size
2. Replicates copy number values to fill the required bins
3. Maintains data continuity while creating equal-sized segments

Value

A data frame with:

- Same number of rows as input
- Expanded columns based on window sizes and bin unit
- Column names as "original_window_name_binIndex"
- Original row names preserved

Note

Window sizes that aren't exact multiples of bin_unit will be rounded up to ensure complete coverage

Examples

```
## Not run:
seg_data <- data.frame(
  chr1_1000_3000 = c(2, 1),
  chr1_3000_4000 = c(1, 1),
  row.names = c("clone1", "clone2")
)
window_sizes <- c(2000, 1000)
bin_unit <- 1000

expanded <- extend_blueprint_cn_for_equal_bin(seg_data, window_sizes, bin_unit)
# Returns expanded data frame with bins of 1000bp each

## End(Not run)
```

---

find_identical_row_index

Find Index of an Identical Row in a Data Frame

Description

Searches a data frame for a row that is identical to the provided row and returns its index.

Usage

```
find_identical_row_index(row, dataframe)
```

Arguments

row	A data frame row or vector to search for
dataframe	The data frame to search within

Details

This function iterates through each row of the provided data frame and compares it with the target row using the all.equal function. The comparison ignores attributes to focus on the content of the data.

The function is particularly useful for finding specific mutations or cells in larger data frames when the exact row index is not known but the content is.

Value

An integer representing the index of the first identical row found, or NA if no identical row exists in the data frame

See Also

sim_snv_nt_sc

---

generate_blueprint_cn_profiles

Generate Blueprint Copy Number Profiles

Description

This function generates copy number (CN) profiles for clones based on their events and a genome-wide window vector. It initializes baseline CN profiles for paternal and maternal haplotypes and updates them according to the events (e.g., copy number changes, whole genome duplications) associated with each clone. The function handles both whole chromosome events and specific region events.

Usage

```
generate_blueprint_cn_profiles(
  clone_events,
  genome_window_vector,
  segment_info
)
```

Arguments

clone_events	A named list of data frames, where each data frame contains events for a clone: • event_type: Type of event (e.g., "WGD" for whole genome duplication) • region_name: Region identifier or chromosome name • CN_change: Copy number change value • haplotype: "maternal" or "paternal"
genome_window_vector	A character vector of window identifiers in format "chr_start_end"

`segment_info` A data frame containing segment information:

- region_name: Region identifier
- chrom: Chromosome name
- start: Start position
- end: End position

Details

The function processes events in these steps:

1. Initializes baseline copy numbers (1 for all regions)
2. For each clone and its events:
   - Handles WGD by doubling all copy numbers
   - Processes chromosome-wide events if region matches "chr0-9XY+"
   - Updates specific regions based on segment information
   - Applies copy number changes progressively
3. Maintains separate profiles for maternal and paternal haplotypes

Value

A list with two elements ("maternal" and "paternal"), each containing a data frame where:

- Rows represent clones
- Columns represent genomic windows
- Values represent copy numbers

Note

- Copy numbers cannot go below 0
- WGD events affect all regions simultaneously
- Chromosome-wide events are detected by regex pattern

Examples

```
## Not run:
clone_events <- list(
  clone1 = data.frame(
    event_type = c("CN", "WGD"),
    region_name = c("chr1_p", "genome"),
    CN_change = c(1, 0),
    haplotype = c("maternal", "maternal")
  ),
  clone2 = data.frame(...)
)
genome_windows <- c("chr1_0_1000000", "chr1_1000000_2000000")
segments <- data.frame(
  region_name = "chr1_p",
  chrom = "chr1",
  start = 0,
  end = 1000000
)
profiles <- generate_blueprint_cn_profiles(clone_events, genome_windows, segments)
```

```
## End(Not run)
```

`generate_chr_boundary_data` *Generate Chromosome Boundary Data*

Description

This function processes chromosome arm and sub-segment data to generate boundary information for specified chromosomes. It combines segments from both input tables, identifies unique boundaries, and creates windows for each chromosome. Additionally, it constructs a genome-wide vector of window identifiers.

Usage

```
generate_chr_boundary_data(chr_arm_table, sub_seg_list, chr_names)
```

Arguments

`chr_arm_table`	A data frame containing chromosome arm information: • chrom: Chromosome name • start: Start position • end: End position • region_name: Identifier for the chromosome arm
`sub_seg_list`	A list of data frames (by haplotype) containing sub-segment information: • chrom: Chromosome name • ref_start: Reference start position • ref_end: Reference end position • region_name: Segment identifier
`chr_names`	A character vector of chromosome names to process

Details

The function performs these steps for each chromosome:

1. Extracts and combines segments from chromosome arms and sub-segments
2. Generates windows based on unique boundary positions
3. Creates formatted window identifiers
4. Organizes results by chromosome

Windows are generated between each unique boundary point found in the combined segment data, ensuring all relevant genomic intervals are captured.

Value

A list containing three elements:

- segment_info: A list of data frames, one for each chromosome, containing combined segments from `chr_arm_table` and `sub_seg_list`.
- windows_info: A list of data frames, one for each chromosome, containing windows derived from the unique boundaries of the combined segments.
- genome_window_vector: Vector of formatted window strings ("chr_start_end")

Examples

```
## Not run:
chr_arms <- data.frame(
  chrom = c("chr1", "chr1"),
  start = c(1, 1000000),
  end = c(1000000, 2000000),
  region_name = c("chr1p", "chr1q")
)
sub_segs <- list(
  maternal = data.frame(...),
  paternal = data.frame(...)
)
chr_names <- c("chr1")
boundaries <- generate_chr_boundary_data(chr_arms, sub_segs, chr_names)

## End(Not run)
```

`generate_K2P_matrix` *Generate Kimura 2-Parameter (K2P) Substitution Matrix*

Description

Generate a Kimura 2-Parameter (K2P) Transition Matrix

This function generates a 5x5 transition probability matrix based on the Kimura 2-Parameter (K2P) model. The matrix includes transition rates for nucleotides (A, C, G, T) and an additional state (N), where transitions and transversions are governed by the parameters `alpha` and `beta`.

Usage

```
generate_K2P_matrix(alpha, beta)
```

Arguments

`alpha`	Numeric value representing the transition rate (rate of A<->G and C<->T mutations)
`beta`	Numeric value representing the transversion rate (rate of all other mutations between A, C, G, T)

Details

The K2P model assumes:

- Different rates for transitions (alpha) and transversions (beta)
- Equal base frequencies
- The 'N' state can only transition to itself
- No transitions are allowed between any nucleotide and 'N'

Matrix entries are normalized by (2*beta + alpha) to ensure proper probability distribution.

Value

A 5x5 matrix with row and column names (A, C, G, T, N) containing substitution probabilities. The matrix entries represent the probability of transitioning from the row nucleotide to the column nucleotide.

Examples

```
# Generate K2P matrix with transition rate 0.3 and transversion rate 0.1
K2P <- generate_K2P_matrix(alpha = 0.3, beta = 0.1)
```

---

generate_mock_window_data

Generate Mock Window Data

Description

Creates a simple data frame containing chromosome information extracted from genomic window identifiers (matrix column names in "chr_start_end" format).

Usage

```
generate_mock_window_data(matrix_colnames)
```

Arguments

matrix_colnames	A character vector of column names in the format "chr_start_end" (e.g., "chr1_1000_2000")

Details

This function creates a mock data frame containing chromosome information extracted from the column names of a matrix. The column names are expected to be in the format `chromosome_start_end`, and the function extracts the chromosome part to create the `chr` column in the resulting data frame.

The function:

1. Splits each column name by "_"
2. Extracts the first element (chromosome name)
3. Creates a data frame with these chromosome names

Value

A data frame with one column:

- chr: Character vector containing chromosome names extracted from input

Examples

```
## Not run:
col_names <- c("chr1_1000_2000", "chr1_2000_3000", "chr2_1000_2000")
window_data <- generate_mock_window_data(col_names)
# Returns data frame:
#   chr
# 1 chr1
# 2 chr1
# 3 chr2

## End(Not run)
```

---

get_clone_ancestors *Get Ancestor Nodes in Phylogenetic Tree*

Description

This function retrieves the ancestors of a specified node (clone) in a tree structure. It calculates the shortest path from the root node to the target node and returns the names of all ancestor nodes (excluding the target node itself).

Usage

```
get_clone_ancestors(tree, node)
```

Arguments

tree	An igraph tree object representing the phylogenetic structure
node	The target node for which to find ancestors

Details

The function:

1. Identifies the root node (assumes first vertex is root)
2. Finds shortest path from root to target node
3. Returns names of all nodes in the path

Value

A character vector containing the names of the ancestor nodes, ordered from the root to the immediate parent of the target node.

Note

Assumes the tree is properly rooted with root node as first vertex

Examples

```
## Not run:
library(igraph)
# Create a simple tree
tree <- make_tree(3, 2)
V(tree)$name <- c("root", "A", "B")

# Get ancestors of node "B"
ancestors <- get_clone_ancestors(tree, "B")
# Returns: c("root", "B")

## End(Not run)
```

`get_edges_between_clones` *Get Edge Names Between Two Nodes in Tree*

Description

Finds and returns the names of all edges in the path between two nodes in a phylogenetic tree, formatted as "parent_child" strings.

Usage

```
get_edges_between_clones(tree, upper_node, lower_node)
```

Arguments

`tree`	An igraph object representing the phylogenetic tree structure
`upper_node`	Character. Name of the starting (ancestor) node
`lower_node`	Character. Name of the ending (descendant) node

Details

The function:

1. Finds the shortest path between the two nodes
2. Identifies all edges along this path
3. Formats edge names as "parent_child"

Note: Assumes that the nodes are connected and upper_node is an ancestor of lower_node in the tree structure.

Value

A character vector containing edge names in the format "parent_child" for all edges in the path from upper_node to lower_node

Examples

```
## Not run:
# For a tree with path A -> B -> C
edges <- get_edges_between_clones(tree, "A", "C")
# Returns: c("A_B", "B_C")

## End(Not run)
```

`get_mutations_sc` *Get Mutations for Sampled Single Cells*

Description

Extracts and organizes mutation information for specified sampled cells, including mutations in their ancestral lineage, and identifies recurrent mutations.

Usage

```
get_mutations_sc(cell_info, mutation_info, sampled_cell_idx = NULL)
```

Arguments

`cell_info`	Data frame containing cell lineage information with columns: clone, parent, birth_time, death_time, cell_index
`mutation_info`	Data frame containing mutation information with columns: clone, cell_index, haplotype, chrom, pos, time
`sampled_cell_idx`	Vector of cell indices for which to retrieve mutation information

Details

This function reconstructs the complete mutation profile for each sampled cell by:

1. Identifying all ancestral cells for each sampled cell using get_sc_ancestors()
2. Collecting mutations from the sampled cell and all its ancestors
3. Identifying recurrent mutations (mutations that occur multiple times independently)
4. Grouping and processing recurrent mutations for tracking
5. Removing duplicate mutations from the final table

The function requires the following helper functions to be defined:

- get_sc_ancestors(): To identify ancestral cells
- identify_recurrent_mutations(): To find mutations that appear multiple times
- group_recurrent_mutations(): To group similar recurrent mutations
- process_recurrent_mutation(): To track and analyze recurrent mutations

Value

A list containing three elements:

- sampled_sc_mutations: List where each element corresponds to a sampled cell, containing a data frame of mutations for that cell and its ancestors
- sampled_mutation_table: Data frame containing all unique mutations across all sampled cells and their ancestors
- recurrent_mutation_tracker: List tracking recurrent mutations found across multiple cells or lineages

See Also

simulate_sc_dynamics, get_sc_ancestors

Examples

```
# Create sample cell_info and mutation_info data frames
cell_info <- data.frame(
  clone = c("A", "A", "B", "A", "B"),
  parent = c(NA, 1, 1, 2, 3),
  birth_time = c(0, 10, 10, 15, 20),
  death_time = c(10, NA, NA, NA, NA),
  cell_index = 1:5
)

mutation_info <- data.frame(
  clone = c("A", "A", "B", "A"),
  cell_index = c(1, 2, 3, 4),
  haplotype = c("hap1", "hap1", "hap2", "hap1"),
  chrom = c("chr1", "chr1", "chr2", "chr1"),
  pos = c(1000, 2000, 1500, 1000),
  time = c(5, 12, 15, 18)
)

# Get mutations for cells 4 and 5
mutations <- get_mutations_sc(cell_info, mutation_info, c(4, 5))

# View mutation summary
print(mutations$sampled_mutation_table)
```

---

get_sc_ancestors *Identify Ancestral Cells in a Lineage*

Description

Traces and returns all ancestral cell indices for a given cell by recursively following the parent relationships in the cell lineage information.

Usage

```
get_sc_ancestors(cell_info, cell_index)
```

Arguments

`cell_info`	Data frame containing cell lineage information with columns: cell_index, parent, and other attributes
`cell_index`	Integer specifying the cell index for which to find ancestors

Details

This function traverses the cell lineage tree upward from the specified cell, following the parent relationships until it reaches a cell with no parent (NA in the parent column). It builds and returns a vector of all ancestor cell indices encountered during this traversal.

The function is commonly used to reconstruct the complete lineage history of a cell, which is particularly useful for aggregating mutations or other heritable properties that are passed down through cell divisions.

Value

A numeric vector containing the cell indices of all ancestors of the specified cell, ordered from immediate parent to earliest ancestor

See Also

`get_mutations_sc`, `simulate_sc_dynamics`

`get_segment_info` *Retrieve Genomic Segment Information for a Mutation*

Description

Identifies and returns the genomic segment that contains a specific mutation, handling both regular segments and segments that have been lost through deletions.

Usage

```
get_segment_info(seg_list, mutation, in_loss_segment)
```

Arguments

`seg_list`	List structure containing segment information for different clones and haplotypes
`mutation`	Data frame row containing information about a single mutation, with columns: clone, haplotype, chrom, pos
`in_loss_segment`	Logical indicating whether the mutation is in a segment that has been lost through deletion

Details

This function locates the appropriate genomic segment for a mutation by:

1. Extracting relevant information (haplotype, chromosome, clone, position) from the mutation
2. Using different search criteria based on whether the segment is lost:
   - For lost segments: Matches chromosome and original coordinates (ori_start/ori_end) where CN_change is -1 and current coordinates (start/end) are NA
   - For normal segments: Matches chromosome and current coordinates (start/end)
3. Returns NULL if no matching segment is found

This function is primarily used by simulate_single_nt_change() to map mutations to their genomic context before simulating nucleotide changes.

Value

A data frame row containing the segment information for the mutation, or NULL if no matching segment is found

See Also

simulate_single_nt_change

---

group_recurrent_mutations

Group Recurrent Mutations by Genomic Location

Description

Organizes recurrent mutations by grouping them according to their genomic location (haplotype, chromosome, and position).

Usage

```
group_recurrent_mutations(recurrent_mutations)
```

Arguments

recurrent_mutations	Data frame containing mutation information with columns: clone, cell_index, haplotype, chrom, pos, time

Details

This function takes a data frame of recurrent mutations and organizes them into groups based on their genomic coordinates. It:

1. Creates a unique key for each mutation combining haplotype, chromosome, and position
2. Splits the data frame into a list of smaller data frames, each containing all mutations that occurred at the same genomic location

This grouping is useful for analyzing patterns of mutations at specific sites and for further processing of recurrent mutations in evolutionary analyses.

Value

A list where each element is a data frame containing all mutations that occurred at the same genomic location. List names are constructed as "haplotype_chromosome_position".

See Also

identify_recurrent_mutations, process_recurrent_mutation

---

identify_recurrent_mutations *Identify Recurrent Mutations in Cell Lineage Data*

Description

Identifies mutations that occur multiple times at the same genomic location across different cells in a lineage.

Usage

```
identify_recurrent_mutations(cell_mutations)
```

Arguments

cell_mutations	Data frame containing mutation information with columns: clone, cell_index, haplotype, chrom, pos, time

Details

This function identifies recurrent mutations by:

1. Creating a unique key for each mutation based on its haplotype, chromosome, and position
2. Identifying duplicate keys, which indicate the same mutation occurring multiple times
3. Extracting and returning only the recurrent mutations

Value

A data frame containing only the recurrent mutations (mutations that appear more than once at the same genomic location)

See Also

get_mutations_sc, group_recurrent_mutations

`identify_region_clusters` *Identify Clusters of Highly Correlated Genomic Regions*

Description

Identifies clusters of adjacent genomic regions that show high correlation with each other. Uses a sliding window approach to find contiguous regions where the average pairwise correlation exceeds a specified threshold.

Usage

```
identify_region_clusters(cor_matrix, min_cluster_size, threshold)
```

Arguments

`cor_matrix`	A numeric correlation matrix where rows and columns represent genomic regions, with column names identifying the regions
`min_cluster_size`	Integer specifying the minimum number of regions required to form a cluster (must be >= 2)
`threshold`	Numeric value between -1 and 1 specifying the minimum average correlation required for regions to be considered part of the same cluster

Details

The function implements a dynamic window-based clustering algorithm:

- Starts with a window of minimum cluster size
- Calculates average correlation (excluding diagonal) within the window
- If average correlation exceeds threshold:
  - Extends window if not at matrix end
  - Finalizes cluster if at matrix end
- If correlation drops below threshold:
  - Finalizes cluster if window size >= minimum
  - Moves to next position if window size < minimum

Value

A list where each element represents a cluster. Each cluster contains:

- Named integer vector with positions of regions in the cluster
- Names of the vector correspond to the region identifiers from the input correlation matrix

See Also

Other clustering functions in the package for genomic analysis

Examples

```
# Create sample correlation matrix
set.seed(123)
n_regions <- 10
cor_mat <- matrix(runif(n_regions^2, -1, 1), nrow = n_regions)
cor_mat[upper.tri(cor_mat)] <- t(cor_mat)[upper.tri(cor_mat)]
diag(cor_mat) <- 1
colnames(cor_mat) <- paste0("region_", 1:n_regions)

# Find clusters
clusters <- identify_region_clusters(
  cor_matrix = cor_mat,
  min_cluster_size = 3,
  threshold = 0.7
)
```

---

initiate_seg_table *Initialize Segment Table for Chromosome Regions*

Description

Creates a table of chromosome segments dividing each chromosome into p (short arm), c (centromere), and q (long arm) regions. Each segment is initialized with base state properties including positions, copy numbers, and source information.

Usage

```
initiate_seg_table(chr_lengths, chr_names, centromere_length, haplotype)
```

Arguments

chr_lengths	A list containing chromosome lengths for a specific haplotype
chr_names	A vector of chromosome names (e.g., c("chr1", "chr2", "chr3"))
centromere_length	The length of the centromere region
haplotype	Character string specifying the haplotype ("maternal" or "paternal")

Value

A data frame containing segment information with columns:

- haplotype: Origin of the segment (maternal/paternal)
- chrom: Chromosome identifier
- ref_start, ref_end: Reference coordinates
- ori_start, ori_end: Original coordinates
- start, end: Current coordinates
- region_name: Region identifier (e.g., chr1p, chr1c, chr1q)
- copy_index: Copy number index

- seg_id: Unique segment identifier
- CN_change: Copy number change status
- seg_source_edge: Source edge information
- seg_source_event: Source event information

Examples

```
chr_lengths <- list(maternal = c(100, 200, 300))
chr_names <- c("chr1", "chr2", "chr3")
centromere_length <- 10
seg_table <- initiate_seg_table(chr_lengths, chr_names, centromere_length, "maternal")
```

---

insert_mutations *Insert Mutations into a Genome*

Description

This function inserts single nucleotide variants (SNVs) into a genome at specified positions. It groups mutations by haplotype and chromosome for efficient processing and modifies the genome sequence accordingly.

Usage

```
insert_mutations(genome, mutations)
```

Arguments

genome	List containing genome sequences organized by haplotype and chromosome
mutations	Data frame containing mutation details with columns: haplotype, chrom, pos, alternative_nt

Details

The function is designed to be efficient when processing large numbers of mutations by minimizing the number of times each sequence needs to be modified.

Value

The modified genome list with mutations incorporated

`insert_snps_to_genome` *Insert SNPs into Reference Genome*

Description

#' This function introduces single nucleotide polymorphisms (SNPs) into a reference genome based on a list of SNPs. The SNPs are categorized as homozygous (`1|1`) or heterozygous (`1|0` or `0|1`), and the function ensures that SNPs at duplicated positions are handled appropriately. The result is a synthetic genome with SNPs inserted for both maternal and paternal haplotypes.

Usage

```
insert_snps_to_genome(seg_names, snp_list, ref_genome)
```

Arguments

`seg_names`	Character vector of segment/chromosome names to process
`snp_list`	A nested list containing SNP information for each segment, with sublists for different genotypes ('1|1', '1|0', '0|1'), each containing data frames with at least POS and ALT columns
`ref_genome`	A DNAStringSet or similar object containing reference sequences, with names matching seg_names

Details

The function processes each segment as follows:

1. Combines homozygous SNPs with appropriate heterozygous SNPs for each haplotype
2. Identifies and removes SNPs with duplicated positions
3. Inserts alternative alleles into the reference sequence
4. Stores both modified sequences and SNP information

Maternal haplotypes receive '1|0' heterozygous variants, while paternal haplotypes receive '0|1' variants. Both haplotypes receive all homozygous alternative ('1|1') variants.

Value

A list with two components:

- sim_genome: List containing maternal and paternal haplotype sequences
- snp_info: List containing SNP information for each haplotype

Examples

```
## Not run:
# Example reference genome
ref_genome <- DNAStringSet(c(
  chr1 = "ACTGACTGACTG",
  chr2 = "GTCAGTCAGTCA"
))
```

```
# Example SNP list
snp_list <- list(
  chr1 = list(
    "1|1" = data.frame(POS = 1, ALT = "G"),
    "1|0" = data.frame(POS = 5, ALT = "T"),
    "0|1" = data.frame(POS = 8, ALT = "C")
  ),
  chr2 = list(
    "1|1" = data.frame(POS = 2, ALT = "A"),
    "1|0" = data.frame(POS = 6, ALT = "G"),
    "0|1" = data.frame(POS = 9, ALT = "T")
  )
)

result <- insert_snps_to_genome(c("chr1", "chr2"), snp_list, ref_genome)

## End(Not run)
```

`introduce_snv` *Introduce Single Nucleotide Variants into a Genome*

Description

Introduce single nucleotide variants (SNVs) into a genome sequence by processing mutations grouped by haplotype and chromosome.

Usage

```
introduce_snv(genome, mut_table, verbose = TRUE)
```

Arguments

`genome`	A nested list containing genome sequences: • First level: haplotypes (maternal/paternal) • Second level: chromosomes with nucleotide sequences
`mut_table`	A data frame containing mutation information with columns: • haplotype - Maternal or paternal copy • chrom - Chromosome name • pos - Position where mutation occurs • alternative_nt - Alternative nucleotide to introduce

Details

The function:

1. Groups mutations by haplotype and chromosome for efficient processing
2. Processes each group of mutations in a vectorized operation
3. Maintains the original genome structure while updating sequences

Note: Uses vectorized operations through replaceLetterAt for efficient mutation introduction

Value

A list containing:

- genome - Modified genome with introduced SNVs

See Also

replaceLetterAt

Examples

```
## Not run:
# Example mutation table
mutations <- data.frame(
  haplotype = "maternal",
  chrom = "chr1",
  pos = 100,
  alternative_nt = "A"
)
modified <- introduce_snv(genome, mutations)

## End(Not run)
```

---

introduce_snv_sc *Introduce Single Nucleotide Variants into a Single Cell Genome*

Description

Incorporates single nucleotide variants (SNVs) into a genome sequence, handling both regular mutations and recurrent mutations at the same genomic position.

Usage

```
introduce_snv_sc(input_genome, sc_mut_table_with_nt)
```

Arguments

input_genome	List containing genome sequences to be modified
sc_mut_table_with_nt	Data frame containing mutation details with columns: haplotype, chrom, pos, original_nt, alternative_nt, time

Details

This function introduces mutations into a genome sequence with special handling for recurrent mutations (multiple mutations at the same genomic position):

1. Identifies recurrent mutations by detecting duplicated positions across haplotype, chromosome, and position
2. Processes non-recurrent mutations first using the insert_mutations() helper function
3. For recurrent mutations:
    - Groups mutations by their genomic coordinates

- For each position with multiple mutations, creates a consolidated mutation that applies the cumulative effect (using the original nucleotide from the first mutation and the alternative nucleotide from the last mutation in the timeline)
- Applies these consolidated recurrent mutations to the genome

This approach ensures that the final genome correctly represents the cumulative effect of sequential mutations at the same position, rather than applying each mutation independently which could lead to incorrect results.

Value

A list containing the modified genome sequences with all mutations incorporated

See Also

`insert_mutations`, `synth_sc_genome`

---

merge_fasta_files *Merge Paternal and Maternal FASTA Files*

Description

This function merges two FASTA files (paternal and maternal) into a single output file, adding suffix identifiers to differentiate the sequences' origins.

Usage

```
merge_fasta_files(paternal_fa, maternal_fa, output_fa, tmp_dir = tempdir())
```

Arguments

`paternal_fa`	Character string. Path to the paternal FASTA file.
`maternal_fa`	Character string. Path to the maternal FASTA file.
`output_fa`	Character string. Path where the merged FASTA file will be written.
`tmp_dir`	Character string. Directory for temporary files. Default is tempdir().

Details

The function adds "_paternal" suffix to sequence headers in the paternal file and "_maternal" suffix to sequence headers in the maternal file before concatenating them into a single output file.

Value

No return value, called for side effect of creating the merged FASTA file.

Examples

```
## Not run:
merge_fasta_files(
  paternal_fa = "path/to/paternal.fa",
  maternal_fa = "path/to/maternal.fa",
  output_fa = "path/to/merged.fa"
)

## End(Not run)
```

`pick_action_cell` *Select a Clone Type for an Event*

Description

Selects a clone type on which an event (birth, death, etc.) will occur based on the population size and action rate of each clone type.

Usage

```
pick_action_cell(population, action_rate)
```

Arguments

`population`	Named numeric vector containing the current population count for each clone type
`action_rate`	Named numeric vector containing the per-cell rate for the action for each clone type

Details

This function implements the clone type selection step in the Gillespie algorithm. The probability of selecting a particular clone type is proportional to its population multiplied by its action rate. This ensures that more abundant clones and clones with higher rates are more likely to be selected.

Value

A character string representing the name of the selected clone type

See Also

`sample_action`, `simulate_sc_dynamics`

Examples

```
# Example usage:
population <- c(Clone1 = 10, Clone2 = 5)
action_rate <- c(Clone1 = 0.1, Clone2 = 0.05)

selected_clone <- pick_action_cell(population, action_rate)
print(selected_clone)  # Possible output: "Clone1"
```

`pick_transition` *Select a Transition Event Between Clone Types*

Description

Selects a specific transition event between clone types based on the current population and edge transition rates, then updates the transition rate matrix to prevent the same transition from occurring again.

Usage

```
pick_transition(population, edge_transition_rates)
```

Arguments

`population`	Named numeric vector containing the current population count for each clone type
`edge_transition_rates`	Data frame with columns: parent, child, rate - specifying transition probabilities between clone types

Details

This function implements the transition selection step in the Gillespie algorithm for cell clone type transitions. The probability of selecting a particular transition is proportional to the population of the parent clone multiplied by the transition rate. After selection, the transition rate for the selected transition is set to zero, indicating it can only occur once in the simulation.

Value

A list containing:

- transition_clone: Character string representing the name of the target clone type (the clone to which the cell transitions)
- edge_transition_rates: Updated data frame of transition rates with the selected transition's rate set to zero

See Also

`sample_action`, `pick_action_cell`, `simulate_sc_dynamics`

Examples

```
# Population counts
pop <- c(A = 100, B = 50, C = 30)

# Define transition matrix
transitions <- data.frame(
  parent = c("A", "B", "A"),
  child = c("B", "C", "C"),
  rate = c(0.01, 0.02, 0.005)
)
```

```
# Select and process a transition event
result <- pick_transition(pop, transitions)
result$transition_clone  # The selected target clone
result$edge_transition_rates  # Updated transition matrix
```

`plot_event_tree` *Plot Event Tree*

Description

This function generates a visual representation of an event tree using the provided edge event table and tree structure. The tree is plotted with edges labeled by event information and nodes colored by their type (normal or tumor). The function returns both the graph object and the plot for further customization or analysis.

Usage

```
plot_event_tree(edge_event_table, tree)
```

Arguments

`edge_event_table`
: A data frame containing edge event information. This is typically generated by the `create_edge_event_table` function:
  - edge_name: Edge identifier ("parent_child" format)
  - edge_label: Event labels for the edge
  - n_events: Number of events on the edge

`tree`
: An igraph tree object representing the phylogenetic structure

Details

The function performs these steps:

1. Converts tree to edge list format
2. Creates node table with clone types (normal/tumor)
3. Merges edge event information with tree structure
4. Creates a dendrogram layout visualization with:
   - Edges labeled with copy number events
   - Nodes colored by clone type
   - Node labels showing clone names

Value

A list containing two elements:

- graph_ob: The igraph object with added event information
- tree_plot: A ggplot2 object showing the rendered tree

Note

The visualization uses:

- Node size of 20
- Text size of 5
- Grey edges
- 2.5mm label dodge for edge labels
- void theme for clean background

Examples

```
## Not run:
library(igraph)
library(ggraph)

# Create sample tree and edge events
tree <- make_tree(3, 2)
edge_events <- create_edge_event_table(your_events, tree)

# Create plot
result <- plot_event_tree(edge_events, tree)
print(result$tree_plot)

## End(Not run)
```

---

plot_population_history

Plot Population Dynamics from Simulation Results

Description

Creates a ggplot visualization of clone population dynamics over time from simulation results produced by the simulate_sc_dynamics function.

Usage

```
plot_population_history(
  simulation_result,
  clone_colors,
  title = "Population History"
)
```

Arguments

simulation_result	List object returned by simulate_sc_dynamics containing population_history
clone_colors	Named vector of colors where names match clone types in the simulation
title	Character string for the plot title, defaults to "Population History"

Details

This function transforms the population history data from wide to long format and creates a line plot showing how the population of each clone type changes over time. The plot includes custom styling for readability and aesthetic presentation.

The time points displayed on the x-axis correspond to the simulation steps without displaying the raw time values (using discrete scale with no labels).

Value

A ggplot object displaying population dynamics over time with customized styling

See Also

simulate_sc_dynamics

Examples

```
# Define colors for each clone type
clone_colors <- c(A = "blue", B = "red")

# Create and display the plot
p <- plot_population_history(sim_result, clone_colors, "Clone Evolution")
print(p)

# Save the plot
ggsave("population_dynamics.png", p, width = 8, height = 6)
```

---

plot_truth_heatmap *Plot Copy Number Heatmap with Clone Identity Annotations*

Description

Creates a ComplexHeatmap visualization of copy number data across genomic coordinates, with annotations for clone identities and chromosome locations. The function specifically handles integer copy number states and includes customizable color schemes.

Usage

```
plot_truth_heatmap(
  seg_dat,
  title,
  clone_identity_vector,
  clone_color_anno,
  chr_bar,
  integer_col,
  max_int = 8
)
```

Arguments

seg_dat	A numeric matrix containing segmented copy number data, where rows represent cells and columns represent genomic windows
title	Character string specifying the title for the heatmap
clone_identity_vector	A vector specifying clone assignments for each cell (row)
clone_color_anno	A named vector of colors for clone annotation
chr_bar	A HeatmapAnnotation object for chromosome visualization (typically created by draw_chr_bar function)
integer_col	Logical indicating whether to treat data as integer copy numbers (TRUE) or continuous values (FALSE)
max_int	Integer specifying the maximum copy number state to display. Values above this will be shown as NA (default: 8)

Details

The function creates a heatmap with the following features:

- Left annotation showing clone identities
- Top annotation showing chromosome boundaries
- Custom color scheme for integer copy numbers using ocean.balance palette
- NA values (including values > max_int) shown in yellow
- Non-clustered visualization preserving genomic order

Value

Draws a ComplexHeatmap visualization and invisibly returns the Heatmap object

See Also

draw_chr_bar for creating chromosome bar annotations

Examples

```
## Not run:
# Create sample data
seg_data <- matrix(sample(0:8, 1000, replace = TRUE), nrow = 20)
clone_ids <- rep(c("Clone1", "Clone2"), each = 10)
clone_colors <- c("Clone1" = "red", "Clone2" = "blue")

# Create chromosome bar annotation
chr_bar <- draw_chr_bar(window_data)

# Plot heatmap
plot_truth_heatmap(
  seg_dat = seg_data,
  title = "Copy Number Profile",
  clone_identity_vector = clone_ids,
  clone_color_anno = clone_colors,
  chr_bar = chr_bar,
  integer_col = TRUE,
```

```
  max_int = 8
)

## End(Not run)
```

`prepare_tree_data` *Prepare Cell Lineage Data for Tree Visualization*

Description

Transforms cell lineage information into a format suitable for phylogenetic tree visualization by creating edge and node tables with appropriate attributes.

Usage

```
prepare_tree_data(cell_info)
```

Arguments

`cell_info`	Data frame containing cell lineage information with columns: clone, parent, birth_time, death_time, cell_index

Details

This function processes cell lineage data to create a tree structure representation:

1. Replaces NA parents with "root_node" to establish a common ancestor
2. Assigns theoretical death times to living cells (cells with NA death_time)
3. Calculates edge lengths as the time between birth and death of each cell
4. Creates an edge table that defines connections between cells
5. Creates a node table with information about each cell and the root

The resulting data structure is compatible with tree visualization packages such as ggtree and can be used to create cell lineage visualizations.

Value

A list containing two data frames:

- edges: Data frame with columns parent, child, edge_label, edge_length, and child_clone
- nodes: Data frame with columns name, clone_type, and death_time

See Also

`simulate_sc_dynamics`, `plot_cell_tree`

Examples

```
# Example usage:
cell_info <- data.frame(
  parent = c(NA, 1, 1, 2),
  cell_index = c(1, 2, 3, 4),
  birth_time = c(0, 1, 1, 2),
  death_time = c(5, 3, NA, 4),
  clone = c("Clone1", "Clone1", "Clone2", "Clone2")
)

tree_data <- prepare_tree_data(cell_info)
print(tree_data$edges)  # Edge table
print(tree_data$nodes)  # Node table
```

---

process_cnv_events *Process and Expand Chromosome-Level CNV Events*

Description

Processes a table of genomic events, expanding chromosome-level CNV events into separate events for p and q arms while preserving other events.

Usage

```
process_cnv_events(event_table)
```

Arguments

event_table	A data frame of genomic events containing: • event_type: Type of event (e.g., "CNV", "WGD") • region_name: Region identifier or chromosome name in format "chrN" where N is the chromosome number • Other event-specific columns that will be preserved

Details

For each row in the input table:

1. If it's a CNV event with region_name matching "chr" followed by numbers:
   - Creates two new rows with "p" and "q" suffixes
   - Copies all other column values to both new rows
2. Otherwise keeps the original row unchanged

Value

A data frame with the same structure as input but with chromosome-level CNV events expanded into p and q arm events. For example:

- A CNV event for "chr1" becomes two events for "chr1p" and "chr1q"
- Non-CNV events or non-chromosome-level CNVs remain unchanged

Examples

```
## Not run:
events <- data.frame(
  event_type = c("CNV", "WGD", "CNV"),
  region_name = c("chr1", "genome", "chr1p"),
  CN_change = c(1, 0, -1),
  stringsAsFactors = FALSE
)

processed <- process_cnv_events(events)
# Returns:
#   event_type region_name CN_change
# 1       CNV       chr1p         1
# 2       CNV       chr1q         1
# 3       WGD      genome         0
# 4       CNV       chr1p        -1

## End(Not run)
```

`process_recurrent_mutation`

Process and Track Recurrent Mutations

Description

Updates a tracker structure that organizes and monitors recurrent mutations across different lineages, categorizing them as identical, inclusive, or branching.

Usage

```
process_recurrent_mutation(mutation, mutation_key, tracker)
```

Arguments

`mutation`	Data frame containing information about a specific set of recurrent mutations at the same genomic location
`mutation_key`	Character string uniquely identifying the genomic location of the mutation (typically in format "haplotype_chromosome_position")
`tracker`	List structure that tracks recurrent mutations by organizing them into sets based on their occurrence patterns

Details

This function processes recurrent mutations by comparing them to existing mutation sets in a tracker. It handles three scenarios:

1. If the mutation set already exists and is identical, it returns the tracker unchanged.
2. If the mutation set is a superset of an existing set, it updates the existing set with new mutations.
3. If the mutation set is distinct from existing sets, it creates a new branch in the tracker.

The function prints the relationship category for debugging and monitoring purposes.

Value

Updated tracker list with the mutation properly categorized and stored

See Also

group_recurrent_mutations, get_mutations_sc

`sample_action` *Sample a Gillespie Event Type*

Description

Randomly selects an event type (birth, death, or transition) based on the relative rates of each event in a Gillespie simulation algorithm.

Usage

```
sample_action(total_birth_rate, total_death_rate, total_transition_rate)
```

Arguments

`total_birth_rate`	Numeric value representing the sum of all birth rates in the system
`total_death_rate`	Numeric value representing the sum of all death rates in the system
`total_transition_rate`	Numeric value representing the sum of all transition rates in the system

Details

This function implements the event selection step of the Gillespie algorithm. The probability of selecting each event type is proportional to its rate relative to the total rate of all events combined.

Value

A character string: either "birth", "death", or "transition" indicating the selected event

See Also

simulate_sc_dynamics

Examples

```
# Example usage:
total_birth_rate <- 0.5
total_death_rate <- 0.3
total_transition_rate <- 0.2

action <- sample_action(total_birth_rate, total_death_rate, total_transition_rate)
print(action)  # Possible output: "birth"
```

`sample_cells` *Sample Living Cells from Simulation Results*

Description

Randomly samples a specified number of living cells from the cell lineage information produced by a dynamics simulation.

Usage

```
sample_cells(cell_info, num_samples)
```

Arguments

`cell_info`	Data frame containing cell lineage information with columns: clone, parent, birth_time, death_time, cell_index, clone_index
`num_samples`	Integer specifying the number of cells to sample

Details

This function first filters the cell_info data frame to identify living cells (those with NA in death_time), then randomly samples the specified number of cells from this subset.

If the number of requested samples exceeds the number of living cells, the function issues a warning and returns all living cells instead.

Value

A data frame containing the cell_info rows for the randomly sampled cells

See Also

`simulate_sc_dynamics`

Examples

```
# Example usage:
cell_info <- data.frame(
  clone = c("Clone1", "Clone1", "Clone2"),
  death_time = c(NA, 5, NA),
  cell_index = c(1, 2, 3)
)
num_samples <- 2

sampled_cells <- sample_cells(cell_info, num_samples)
print(sampled_cells)  # Sampled live cells
```

`segments_sanity_check` *Perform Sanity Checks on Genomic Segments*

Description

This function performs a sanity check on chromosome segments to ensure that the total length of the segments matches the expected chromosome lengths for both maternal and paternal haplotypes. It checks each chromosome and haplotype combination and reports any discrepancies.

Usage

```
segments_sanity_check(segment_list, chr_lengths)
```

Arguments

`segment_list`	A list containing two data frames (maternal and paternal), each with columns: • chrom - Chromosome name • ref_start - Segment start position • ref_end - Segment end position • CN_change - Copy number change (-1 indicates segments to exclude)
`chr_lengths`	A list containing two named numeric vectors (maternal and paternal) where names are chromosome identifiers and values are chromosome lengths

Details

The function performs the following checks for each chromosome in both haplotypes:

1. Excludes segments with CN_change == -1
2. Sums the lengths of remaining segments (ref_end - ref_start + 1)
3. Compares total segment length with expected chromosome length
4. Records any mismatches in the returned list

Value

A list where:

- Names are concatenated strings of haplotype and chromosome (e.g., "maternal chr1")
- Values are error messages for failed checks
- Empty list indicates all checks passed

See Also

Related functions for segment manipulation and validation

Examples

```
## Not run:
# Create sample segment list
segments <- list(
  maternal = data.frame(
    chrom = c("chr1", "chr1"),
    ref_start = c(1, 101),
    ref_end = c(100, 200),
    CN_change = c(0, 1)
  ),
  paternal = data.frame(
    chrom = c("chr1", "chr1"),
    ref_start = c(1, 101),
    ref_end = c(100, 200),
    CN_change = c(0, 0)
  )
)

# Create sample chromosome lengths
chr_lengths <- list(
  maternal = c(chr1 = 200),
  paternal = c(chr1 = 200)
)

# Run sanity checks
results <- segments_sanity_check(segments, chr_lengths)
if (length(results) == 0) {
  print("All checks passed")
} else {
  print("Some checks failed:")
  print(results)
}

## End(Not run)
```

---

select_cell_index *Select a Specific Cell from a Clone Population*

Description

Randomly selects a specific cell from the living cells of a given clone type based on the cell lineage information.

Usage

```
select_cell_index(cell_info, clone_type)
```

Arguments

cell_info	Data frame containing cell lineage information with columns: clone, parent, birth_time, death_time, cell_index, clone_index
clone_type	Character string specifying the clone type from which to select a cell

Details

This function filters the cell_info data frame to find all living cells (those without a death_time) of the specified clone type. If there is only one such cell, it is returned directly. If there are multiple cells, one is chosen randomly with equal probability.

Living cells are identified as those where death_time is NA in the cell_info data frame.

Value

An integer representing the selected cell's unique index

Example usage:

cell_info <- data.frame(clone = c("Clone1", "Clone1", "Clone2"), death_time = c(NA, 5, NA), cell_index = c(1, 2, 3)) clone_type <- "Clone1"

selected_cell_index <- select_cell_index(cell_info, clone_type) print(selected_cell_index) # Possible output: 1

See Also

pick_action_cell, simulate_sc_dynamics

`simulate_read_dynamic_sc` *Simulate Single-cell Level (dynamics incorporated) Sequencing Reads Using ART in Parallel*

Description

Simulates next-generation sequencing reads from input FASTA files in parallel using the ART sequencing simulator (Huang et al., Bioinformatics 2012). This function supports both single-end and paired-end read generation.

Usage

```
simulate_read_dynamic_sc(
  fasta_inputs,
  output_prefixes,
  readLen = 100,
  depth = 30,
  artPath = "art_illumina",
  seqSys = "HS25",
  paired = TRUE,
  numCores = 2,
  otherArgs = ""
)
```

Arguments

fasta_inputs	Character vector of paths to input FASTA files
output_prefixes	Character vector of output prefixes for generated read files
readLen	Integer specifying the length of simulated reads in base pairs (default: 100)
depth	Numeric value specifying the desired sequencing depth/coverage (default: 30)
artPath	Character string specifying the path to the ART executable (default: "art_illumina")
seqSys	Character string specifying the sequencing system to simulate (default: "HS25" for HiSeq 2500)
paired	Logical indicating whether to generate paired-end reads (default: TRUE)
numCores	Integer specifying the number of CPU cores to use for parallel processing (default: 2)
otherArgs	Character string with additional arguments to pass to ART (default: "")

Details

This function provides a parallel interface to the ART sequencing simulator by:

1. Defining an internal function `simulateOne()` that constructs and executes the ART command for a single FASTA input
2. Using `mclapply()` to run multiple simulations in parallel
3. Redirecting ART's output to log files

For paired-end reads (when paired=TRUE), the function sets a mean fragment length of 200bp with a standard deviation of 10bp. For single-end reads, these parameters are omitted.

The function requires that the ART executable is installed and accessible.

Value

None. The function generates sequencing read files at the locations specified by output_prefixes and prints completion messages.

See Also

simulate_sc_dynamic_reads_for_batches

Examples

```
## Not run:
# Simulate single-end reads for 3 genomes
simulate_read_dynamic_sc(
  fasta_inputs = c("data/genome1.fa", "data/genome2.fa", "data/genome3.fa"),
  output_prefixes = c("results/sim1", "results/sim2", "results/sim3"),
  readLen = 150,
  depth = 30,
  artPath = "/usr/local/bin/art_illumina",
  seqSys = "HS25",
  paired = FALSE,
  numCores = 3
)
```

```
# Simulate paired-end reads with custom ART arguments
simulate_read_dynamic_sc(
  fasta_inputs = c("data/genome1.fa", "data/genome2.fa"),
  output_prefixes = c("results/paired1", "results/paired2"),
  paired = TRUE,
  otherArgs = "--noALN --rndSeed 42"
)

## End(Not run)
```

`simulate_read_sc` *Simulate Clonal-level Single-Cell Sequencing Reads*

Description

This function uses the ART read simulator (Huang et al., Bioinformatics 2012) to generate synthetic sequencing reads from a reference FASTA file for multiple single cells in parallel. It supports both paired-end and single-end reads and allows for parallel execution across multiple cores. The function generates FASTQ files for each cell and logs the ART output.

Usage

```
simulate_read_sc(
  fasta_input,
  output_prefixes,
  readLen = 100,
  depth = 1,
  artPath = "art_illumina",
  seqSys = "HS25",
  paired = TRUE,
  numCores = 2,
  otherArgs = ""
)
```

Arguments

`fasta_input`	Character string. Path to the input FASTA file.
`output_prefixes`	Character vector. Prefixes for output files, one per simulated cell.
`readLen`	Numeric. Length of the simulated reads in base pairs. Default is 100.
`depth`	Numeric. Sequencing coverage depth. Default is 1.
`artPath`	Character string. Path to the ART Illumina executable. Default is "art_illumina".
`seqSys`	Character string. Sequencing system to simulate. Default is "HS25" (HiSeq 2500).
`paired`	Logical. Whether to generate paired-end reads (TRUE) or single-end reads (FALSE). Default is TRUE.
`numCores`	Numeric. Number of cores to use for parallel processing. Default is 2.
`otherArgs`	Character string. Additional arguments to pass to ART Illumina. Default is "".

Details

This function creates synthetic sequencing reads by calling ART Illumina for each output prefix in parallel using mclapply. For paired-end reads, it sets a mean fragment length of 200bp with a standard deviation of 10bp. Simulation output and errors are redirected to log files.

Value

No return value, called for side effects of generating simulated read files and printing completion messages.

Examples

```
## Not run:
# Simulate reads for 3 cells with default parameters
simulate_read_sc(
  fasta_input = "reference.fa",
  output_prefixes = c("cell1_", "cell2_", "cell3_")
)

# Simulate single-end reads with custom parameters
simulate_read_sc(
  fasta_input = "reference.fa",
  output_prefixes = c("cell1_", "cell2_"),
  readLen = 150,
  depth = 0.01,
  paired = FALSE,
  ...
)

## End(Not run)
```

---

simulate_sc_dynamics *Simulate Single-Cell Dynamics with Mutations*

Description

Simulates cell population dynamics using a Gillespie algorithm, tracking births, deaths, transitions between cell types, and mutations on chromosomes. This function models cell growth with logistic constraints and generates detailed lineage information.

Usage

```
simulate_sc_dynamics(
  initial_population,
  max_steps,
  intrinsic_birth_rates,
  intrinsic_death_rates,
  edge_transition_rates,
  clone_capacity,
  chr_lengths,
  mutation_rate
)
```

Arguments

`initial_population`	Named numeric vector specifying the initial count for each clone type
`max_steps`	Maximum number of simulation steps to perform
`intrinsic_birth_rates`	Named numeric vector of birth rates for each clone type
`intrinsic_death_rates`	Named numeric vector of death rates for each clone type
`edge_transition_rates`	Data frame with columns: parent, child, rate - specifying transitions between clone types
`clone_capacity`	Numeric value representing the carrying capacity of the system of each clone
`chr_lengths`	Nested list structure defining chromosome lengths for each clone and haplotype
`mutation_rate`	Probability of mutation occurring during cell division

Details

The simulation implements a Gillespie algorithm with three possible events:

1. Birth: A cell divides into two daughter cells with possible mutations
2. Death: A cell dies and is removed from the population
3. Transition: A cell changes from one clone type to another

Birth rates are modulated by logistic growth constraints based on total population and clone capacity. The simulation tracks detailed lineage information including parent-child relationships, birth and death times for each cell, and mutation events.

Value

A list containing:

- population_history: Data frame tracking population counts over time
- time_history: Vector of time points corresponding to population_history
- cell_info: Data frame with detailed lineage information for each cell
- mutation_info: Data frame recording mutation events that occurred during simulation

Note

The simulation stops when either max_steps is reached or the total event rate becomes zero.

See Also

`sample_action`, `pick_action_cell`, `select_cell_index`, `pick_transition`

`simulate_sc_dynamic_reads_for_batches` *Simulate Sequencing Reads for Single Cells in Batches*

Description

Simulates sequencing reads for multiple single cells in batches by synthesizing cell-specific genomes with mutations and generating sequencing reads using the ART simulator. This function processes cells in batches to manage memory usage.

Usage

```
simulate_sc_dynamic_reads_for_batches(
  sampled_cell_idx,
  dynamics_ob,
  sc_founder_genomes,
  all_sampled_sc_mutations,
  sc_mut_table_with_nt,
  sim_updates,
  fa_dir,
  output_dir,
  art_path,
  n_cores = 4,
  depth = 30,
  readLen = 150,
  otherArgs = "",
  keep_genome_files = FALSE,
  batch_size = 10
)
```

Arguments

`sampled_cell_idx`	Vector of cell indices for which to simulate reads
`dynamics_ob`	List object returned by simulate_sc_dynamics containing cell lineage information
`sc_founder_genomes`	List containing genome information for clone founders with elements: all_node_genomes, child_node_founders_df
`all_sampled_sc_mutations`	List containing mutation information for sampled cells
`sc_mut_table_with_nt`	Data frame containing mutation details with nucleotide changes
`sim_updates`	List containing updated simulation information including all_node_segments
`fa_dir`	Character string specifying the directory to store FASTA files
`output_dir`	Character string specifying the directory for output read files
`art_path`	Character string specifying the path to the ART sequencing simulator executable
`n_cores`	Integer specifying the number of CPU cores to use for parallel processing (default: 4)

depth	Numeric value specifying the desired sequencing depth/coverage (default: 30)
readLen	Integer specifying the length of simulated reads in base pairs (default: 150)
otherArgs	Character string with additional arguments to pass to ART (default: "")
keep_genome_files	Logical indicating whether to retain the temporary FASTA genome files after simulation (default: FALSE)
batch_size	Integer specifying the number of cells to process in each batch (default: 10)

Details

This function performs the following steps for each batch of cells:

1. For each cell in the batch:
   - Extracts cell lineage information from dynamics_ob
   - Synthesizes a cell-specific genome with mutations using synth_sc_genome()
   - Performs checks on the generated genome using check_genome_mutations()
   - Validates the sanity check results with validate_mutation_check()
   - Combines maternal and paternal haplotypes into a single FASTA file
   - Writes the merged genome to disk
2. Simulates sequencing reads for all cells in the batch in parallel using simulate_read_dynamic_sc()
3. Optionally removes temporary genome files to save disk space

Processing cells in batches helps manage memory usage when dealing with large numbers of cells. The function relies on external functions like synth_sc_genome(), check_genome_mutations(), validate_mutation_check(), and simulate_read_dynamic_sc().

Value

A list where each element contains mutation check results for a cell, with the cell index as the name of each element

See Also

synth_sc_genome, check_genome_mutations, validate_mutation_check, simulate_read_dynamic_sc

Examples

```
## Not run:
# Assuming you have already run a simulation and have necessary objects
mutation_checks <- simulate_sc_dynamic_reads_for_batches(
  sampled_cell_idx = c(50, 51, 52, 53, 54),
  dynamics_ob = sc_dynamics_result,
  sc_founder_genomes = founder_genomes,
  all_sampled_sc_mutations = sampled_mutations,
  sc_mut_table_with_nt = mutations_with_nt,
  sim_updates = simulation_updates,
  fa_dir = "data/genomes/",
  output_dir = "results/reads/",
  art_path = "/usr/local/bin/art_illumina",
  n_cores = 4,
  depth = 30,
  readLen = 150,
  keep_genome_files = FALSE,
```

```
  batch_size = 2
)

# Check results for a specific cell
print(mutation_checks[["50"]])

## End(Not run)
```

`simulate_single_nt_change` *Simulate Nucleotide Change for a Single Mutation*

Description

Simulates the specific nucleotide change for a single mutation based on genomic context, handling both regular and recurrent mutations, as well as mutations in lost segments.

Usage

```
simulate_single_nt_change(
  single_mutation,
  seg_list,
  genome_sequence,
  nt_transition_matrix,
  recurrent = FALSE,
  original_nt = NA
)
```

Arguments

`single_mutation`	Data frame row containing information about a single mutation
`seg_list`	List structure containing segment information for genome regions
`genome_sequence`	List of DNA sequences representing the reference genome
`nt_transition_matrix`	Matrix specifying nucleotide transition probabilities
`recurrent`	Logical indicating whether this is a recurrent mutation (default: FALSE)
`original_nt`	Character specifying the original nucleotide for recurrent mutations (required when recurrent = TRUE, default: NA)

Details

This function simulates nucleotide-level details for a single mutation by:

1. Identifying the genomic segment containing the mutation
2. Determining if the segment has been lost through deletion or other structural variants
3. For mutations in lost segments:
    - Setting both original and alternative nucleotides to "N"

4. For mutations in non-lost segments:
    - For regular mutations: Looking up the original nucleotide from the reference genome and sampling an alternative nucleotide based on the transition matrix
    - For recurrent mutations: Using the provided original nucleotide and sampling an alternative nucleotide based on the transition matrix

The function requires the helper function get_segment_info() to map the mutation coordinates to the appropriate genomic segment.

Value

The updated single_mutation row with additional fields:

- seg_id: Segment identifier for the mutation
- ref_pos: Reference position within the segment
- original_nt: Original nucleotide at the mutation site
- alternative_nt: Mutated nucleotide
- processed: Set to TRUE indicating the mutation has been processed

See Also

get_segment_info, sim_snv_nt_sc

---

sim_clonal_mutation_nt

Simulate Nucleotide Changes for Clonal Mutations

Description

Simulates nucleotide changes for mutations in a clonal evolutionary tree, handling both regular and recurrent mutations. Takes into account chromosome segment information and possible loss events.

Usage

```
sim_clonal_mutation_nt(
  genome_sequence,
  seg_list,
  mutation_info,
  nt_transition_matrix,
  tree
)
```

Arguments

`genome_sequence`
: A nested list containing reference genome sequences:
    - First level: haplotypes (maternal/paternal)
    - Second level: chromosomes
    - Each chromosome contains nucleotide sequence

`seg_list` A nested list containing segment information for each clone and haplotype, including coordinates and copy number changes.

`mutation_info` A data frame containing mutation locations with columns:

- clone - Clone name where mutation occurs
- haplotype - Maternal or paternal copy
- chrom - Chromosome name
- pos - Position on chromosome

`nt_transition_matrix` A 5x5 matrix containing nucleotide transition probabilities

`tree` An igraph object representing the phylogenetic tree structure

Details

The function processes mutations in two categories:

1. Regular mutations (first occurrence at a position):
   - Identifies segment containing the mutation
   - Handles lost segments (marked with 'N')
   - Samples alternative nucleotide based on transition matrix
2. Recurrent mutations (at previously mutated positions):
   - Finds previous mutation in ancestor clones
   - Uses previous alternative as new original nucleotide
   - Samples new alternative based on transition matrix

Value

An updated mutation_info data frame with additional columns:

- seg_id - Segment identifier where mutation occurs
- ref_pos - Position in reference coordinates
- original_nt - Original nucleotide
- alternative_nt - Mutated nucleotide

See Also

`get_segment_info`, `get_clone_ancestors`

`sim_clonal_mutation_pos` *Simulate Random Mutation Positions in a Clonal Tree*

Description

Simulates random mutation positions along chromosomes for each edge in a phylogenetic tree. Mutations are distributed across chromosomes proportionally to their lengths, and can occur on either maternal or paternal haplotypes.

Usage

```
sim_clonal_mutation_pos(tree, chr_lengths, mutation_number)
```

Arguments

`tree`	An igraph object representing the phylogenetic tree structure.
`chr_lengths`	A nested list containing chromosome lengths for each clone and haplotype: • First level: clone names • Second level: haplotypes (maternal/paternal) • Third level: named numeric vector of chromosome lengths
`mutation_number`	A named numeric vector specifying the number of mutations to simulate for each edge in the tree. Names should be in format "parent_child".

Details

The function processes the tree in depth-first search order, starting from the first tumor clone (excluding root). For each edge, it simulates the specified number of mutations by:

1. Randomly selecting a haplotype (maternal/paternal)
2. Selecting a chromosome with probability proportional to its length
3. Randomly selecting a position within the chosen chromosome

Value

A data frame containing simulated mutation information:

- clone - Name of the clone where mutation occurs
- edge_name - Tree edge identifier (parent_child)
- haplotype - Maternal or paternal haplotype
- chrom - Chromosome where mutation occurs
- pos - Position of mutation on the chromosome

`sim_snv_nt_sc` *Simulate Single Nucleotide Variants for Single Cell Data*

Description

Simulates nucleotide-level details for single nucleotide variants (SNVs) in single cell mutation data, handling both regular and recurrent mutations with appropriate nucleotide changes.

Usage

```
sim_snv_nt_sc(
  genome_sequence,
  seg_list,
  mutation_table,
  recurrent_mutation_tracker = NA,
  nt_transition_matrix
)
```

Arguments

genome_sequence	List of DNA sequences representing the reference genome
seg_list	Data frame or list containing genomic segment information with mapping between chromosome coordinates and segment identifiers
mutation_table	Data frame containing mutation information with columns: clone, cell_index, haplotype, chrom, pos, time
recurrent_mutation_tracker	List structure tracking recurrent mutations, organized by genomic location and mutation sets (default is NA for no recurrent mutations)
nt_transition_matrix	Matrix specifying nucleotide transition probabilities for different mutation types

Details

This function simulates the nucleotide-level details of mutations by:

1. Extending the mutation table with columns for segment ID, reference position, original nucleotide, alternative nucleotide, and processing status
2. Processing recurrent mutations first, maintaining proper nucleotide changes across mutation sets (later mutations in a set build upon earlier ones)
3. Processing regular (non-recurrent) mutations

For recurrent mutations, the function ensures that:

- The first mutation in a set is processed normally
- Subsequent mutations in the set use the alternative nucleotide from the previous mutation as their original nucleotide

The function relies on helper functions:

- find_identical_row_index(): To locate specific mutations in the table
- simulate_single_nt_change(): To determine nucleotide changes for individual mutations

Value

An extended mutation_table data frame with additional columns:

- seg_id: Segment identifier for the mutation
- ref_pos: Reference position within the segment
- original_nt: Original nucleotide at the mutation site
- alternative_nt: Mutated nucleotide
- processed: Logical flag indicating whether the mutation has been processed

See Also

simulate_single_nt_change, find_identical_row_index, process_recurrent_mutation

`subset_interested_profiles` *Subset and Transform Haplotype Copy Number Profiles for Regions of Interest*

Description

Extracts copy number data for specific regions of interest from a haplotype-specific copy number profile and returns a transformed matrix with the same dimensions as the original data, where only the regions of interest contain values.

Usage

```
subset_interested_profiles(
  haplotype_cn,
  haplotype,
  chr,
  interested_region_indices
)
```

Arguments

`haplotype_cn`	A list containing haplotype-specific copy number matrices
`haplotype`	Character string specifying which haplotype to analyze (must be a name in the haplotype_cn list)
`chr`	Character string specifying the chromosome of interest (e.g., "chr1", "chrX")
`interested_region_indices`	Numeric vector of indices specifying which regions within the chromosome to analyze

Details

The function performs these steps:

- Identifies regions in the specified chromosome
- Extracts copy number data for regions of interest
- Creates a new matrix with same dimensions as original data
- Fills in only the specified regions, leaving others as NA

Value

A numeric matrix with the same dimensions as the original copy number matrix, where:

- Only specified regions of interest contain copy number values
- All other positions contain NA
- Column names are preserved from the original matrix

See Also

Other functions for manipulating copy number profiles in the package

Examples

```
# Create sample haplotype copy number data
sample_cn <- list(
  hap1 = matrix(1:20, nrow = 2, ncol = 10,
                dimnames = list(NULL,
                paste0("chr1_", 1:10))),
  hap2 = matrix(2:21, nrow = 2, ncol = 10,
                dimnames = list(NULL,
                paste0("chr1_", 1:10)))
)

# Extract regions of interest
subset_data <- subset_interested_profiles(
  haplotype_cn = sample_cn,
  haplotype = "hap1",
  chr = "chr1",
  interested_region_indices = 2:4
)
```

---

sub_seg_from_cytoband_anno

Create Genomic Segments from Cytoband Annotations

Description

Converts clustered cytoband annotations into genomic segments by identifying the spanning regions for each cluster. Creates segments for both maternal and paternal haplotypes with additional metadata for downstream analysis.

Usage

```
sub_seg_from_cytoband_anno(cluster_anno)
```

Arguments

cluster_anno	A nested list containing cytoband annotations organized by haplotype (maternal/paternal), chromosome, and cluster. Each cluster contains a data frame of cytoband information including chromStart and chromEnd positions.

Details

For each cluster in the input annotations, the function:

1. Identifies the minimum start and maximum end positions of all cytobands
2. Creates a unique region name using haplotype, chromosome, and cluster index
3. Initializes metadata fields for downstream analysis
4. Combines all segments into haplotype-specific data frames

The resulting segments serve as a baseline for further genomic analyses, with fields prepared for tracking copy number changes and segment evolution.

Value

A list with two elements (maternal and paternal), each containing a data frame of genomic segments with the following columns:

- chrom: Chromosome name
- start: Starting position of the segment
- end: Ending position of the segment
- region_name: Unique identifier (format: "sub_haplotype_chrom_clusterIndex")
- haplotype: Maternal or paternal
- ref_start: Reference start position
- ref_end: Reference end position
- ori_start: Original start position
- ori_end: Original end position
- copy_index: Copy number index (initialized to 0)
- seg_id: Segment identifier (initialized to 0)
- CN_change: Copy number change (initialized to 0)
- seg_source_edge: Source edge info (initialized to "root")
- seg_source_event: Source event info (initialized to "base")

See Also

Related functions for genomic segment analysis and manipulation

Examples

```
## Not run:
# Assuming cluster_anno is already defined:
segments <- sub_seg_from_cytoband_anno(cluster_anno)

# Access maternal segments:
maternal_segs <- segments[["maternal"]]

## End(Not run)
```

---

synth_clone_genome *Synthesize Clone Genome Based on Events and Mutations*

Description

Constructs a clone's genome by applying sequential genomic alterations (CNVs, WGDs) and mutations (SNVs) starting from a nearest ancestral genome. Processes events along the evolutionary path between the nearest ancestor and target clone.

Usage

```
synth_clone_genome(
  target_clone,
  nearest_genome,
  nearest_clone,
  tree,
  seg_list,
  mut_table,
  verbose = TRUE
)
```

Arguments

target_clone	Character. Name of the clone whose genome is to be synthesized.
nearest_genome	Nested list containing the nearest ancestor's genome sequences: • First level: haplotypes (maternal/paternal) • Second level: chromosomes with nucleotide sequences
nearest_clone	Character. Name of the nearest ancestor clone.
tree	igraph object. Phylogenetic tree structure.
seg_list	Nested list containing segment information for all clones: • First level: clone names • Second level: haplotypes • Each haplotype contains segment information (data frame)
mut_table	Data frame containing mutation information with columns: • edge_name - Tree edge identifier • other mutation-specific columns required by introduce_snv

Details

The function:

1. Identifies edges between nearest ancestor and target clone
2. For each edge:
   - Processes segment changes (copy number variations)
   - Handles segment losses (marks with "N")
   - Adds new segments for gains
   - Applies SNVs if present
3. Maintains separate tracking for maternal and paternal haplotypes

Value

A nested list containing the synthesized genome with the same structure as nearest_genome, but updated with all genomic changes.

See Also

get_edges_between_clones, introduce_snv

`synth_sc_founder_genome` *Synthesize Founder Genome with Structural Variants*

Description

Incorporates structural variants (deletions, duplications) into a genome during the transition from parent to child clone in a cell lineage tree.

Usage

```
synth_sc_founder_genome(child_node, parent_node, pre_child_genome, seg_list)
```

Arguments

`child_node`	Character string specifying the name of the child clone
`parent_node`	Character string specifying the name of the parent clone
`pre_child_genome`	List containing the genome sequences after SNVs have been incorporated
`seg_list`	List structure containing segment information for all clones

Details

This function applies structural variants to a genome during clone evolution by:

1. Starting with a genome that already has SNVs incorporated (pre_child_genome)
2. Identifying segments that are associated with the specific transition edge between parent and child clones
3. Processing each relevant segment based on its copy number change (CN_change):
   - For deletions (CN_change = -1): Replaces the original sequence with "N" characters
   - For duplications (CN_change >= 1): Copies the sequence and adds it to the end of the chromosome

The function works on each haplotype (maternal/paternal) separately and modifies only chromosomes that have structural variant events associated with the specific parent-to-child clone transition.

This function is typically called after SNVs have been incorporated into the genome but before cell-specific mutations are added, representing the genetic changes that define a new clone's emergence.

Value

A list containing the modified genome with structural variants incorporated

See Also

`synth_sc_tree_founder_genomes`, `introduce_snv_sc`

synth_sc_genome *Synthesize Single-Cell Genome*

Description

This function synthesizes a single-cell genome by introducing single nucleotide variants (SNVs) into a backbone genome. It retrieves cell-specific mutations, filters mutations that occurred after the founder cell, and introduces the mutations into the genome.

Usage

```
synth_sc_genome(
  cell_index,
  backbone_genome,
  sc_mut_list,
  mut_table_with_nt,
  founder_cell_index
)
```

Arguments

cell_index	Integer specifying the index of the cell for which to synthesize a genome
backbone_genome	List containing the reference genome sequences of the cell's clone
sc_mut_list	List where each element corresponds to a cell's mutations, with cell indices as names
mut_table_with_nt	Data frame containing mutation details with nucleotide changes
founder_cell_index	Integer specifying the founder cell index of the clone

Value

A list containing two elements:

- sc_genome: The synthesized single cell genome with mutations incorporated
- cell_mut_with_nt: Data frame containing the cell-specific mutations with nucleotide details

See Also

find_identical_row_index, introduce_snv_sc, simulate_sc_dynamic_reads_for_batches

`synth_sc_tree_founder_genomes` *Synthesize Founder Genomes for a Cell Lineage Tree*

Description

Synthesizes founder genomes for each clone in a cell lineage tree by traversing the tree from root to leaves, incorporating appropriate mutations and structural variants at each step.

Usage

```
synth_sc_tree_founder_genomes(
  tree,
  root_genome,
  seg_list,
  mutation_info,
  cell_info
)
```

Arguments

`tree`	An igraph object representing the cell lineage tree with vertices as clone types
`root_genome`	List containing the genome sequences for the root clone
`seg_list`	List structure containing segment information for all clones
`mutation_info`	Data frame containing mutation details for all cells
`cell_info`	Data frame containing cell lineage information

Details

The function builds genomes sequentially following the evolutionary relationships defined in the tree structure, ensuring that each clone's genome properly inherits all modifications from its ancestors plus its unique changes.

Value

A list containing four elements:

- all_node_genomes: List of genome sequences for each clone in the tree
- child_node_founders_df: Data frame containing information about the founder cells for each clone
- child_node_founder_mutations: List of all mutations in founder cells, organized by clone
- child_node_founder_mutations_filtered: List of mutations that occurred between parent and child nodes

See Also

`synth_sc_founder_genome`, `introduce_snv_sc`, `get_mutations_sc`

Examples

```
## Not run:
# Assuming you have a cell lineage tree and other required objects
founder_genomes <- synth_sc_tree_founder_genomes(
  tree = clone_tree,
  root_genome = reference_genome,
  seg_list = all_segments,
  mutation_info = mutations,
  cell_info = cell_lineage
)
## End(Not run)
```

---

synth_tree_genomes *Synthesize Genomes for All Clones in Phylogenetic Tree*

Description

Generates complete genome sequences for all clones in a phylogenetic tree, starting from the root genome and applying sequential genomic alterations following the tree structure.

Usage

```
synth_tree_genomes(tree, root_genome, seg_list, mut_table)
```

Arguments

`tree`	An igraph object representing the phylogenetic tree structure
`root_genome`	A nested list containing the initial genome sequences: • First level: haplotypes (maternal/paternal) • Second level: chromosomes with nucleotide sequences
`seg_list`	A nested list containing segment information for all clones: • First level: clone names • Second level: haplotypes • Each haplotype contains segment information (data frame)
`mut_table`	A data frame containing mutation information for all clones

Details

The function:

1. Identifies the root node of the tree
2. Processes nodes in depth-first search order
3. For each node:
   - Identifies its parent node
   - Synthesizes its genome based on parent's genome
   - Applies all genomic changes along the branch
4. Tracks progress with print statements

Value

A list where:

- Names are clone names from the tree
- Values are genome sequences for each clone

See Also

synth_clone_genome, dfs

`update_sim_from_event_table` *Update Simulation from Event Table*

Description

This function updates a simulation based on an event table, which contains information about genomic events such as Whole Genome Duplications (WGD), Copy Number Variations (CNV), and sub-CNVs. It traverses a tree structure and applies the events to the segments and chromosome lengths accordingly. Events are processed in depth-first search order from root to leaves.

Usage

```
update_sim_from_event_table(
  tree,
  event_table,
  initial_chr_arm_seg_list,
  initial_sub_seg_list = NULL,
  initial_chr_lengths
)
```

Arguments

`tree`	An igraph object representing the phylogenetic tree structure
`event_table`	A data frame containing events with columns: • parent - parent node name • child - child node name • event_type - type of event ("WGD", "CNV", or "sub_CNV") • other event-specific columns
`initial_chr_arm_seg_list`	List containing initial chromosome arm segment information
`initial_sub_seg_list`	List containing initial sub-segment information. Optional, required only if sub_CNV events are present in event_table
`initial_chr_lengths`	Vector or list containing initial chromosome lengths

Value

A list containing:

- all_node_events - List of events occurring on each tree edge
- all_node_segments - List of segment states for each node
- all_node_chr_lengths - List of chromosome lengths for each node

See Also

update_wgd_seg, update_cnv_seg, update_sub_seg

---

validate_mutation_check

Validate Genome Mutation Check Results

Description

Validates the output from mutation checking functions to ensure all mutations were correctly introduced into the genome.

Usage

```
validate_mutation_check(check_output)
```

Arguments

check_output	A nested list structure containing mutation check results, typically returned by check_genome_mutations()

Details

This function examines the output from a mutation checking function to verify that all mutations were correctly introduced into the genome. It:

This function is typically used as a safeguard in genome simulation pipelines to ensure the integrity of the mutation introduction process.

Value

Logical value: TRUE if all mutations were correctly introduced, FALSE if any validation failures were detected

See Also

check_genome_mutations, simulate_sc_dynamic_reads_for_batches

`vcf_to_snp_list` *Convert Phased VCF Table to SNP Lists by Genotype*

Description

Takes a phased VCF table and converts it into a list of filtered SNP data frames, separated by genotype categories (homozygous alternative and heterozygous variants).

Usage

```
vcf_to_snp_list(phased_vcf_table, sample_name)
```

Arguments

`phased_vcf_table`	A data frame containing phased VCF data with at least the following columns: CHROM, POS, REF, ALT, and a sample-specific genotype column
`sample_name`	Character string specifying the name of the sample column in the VCF table

Details

The function filters for single-nucleotide variants only (where REF and ALT are single characters) and separates the genotype field (GT) from the dosage field (DS) in the sample column.

Value

A list containing four data frames:

- all: All SNPs (single-nucleotide variants only)
- 1|1: Homozygous alternative variants
- 1|0: Heterozygous variants with alternative allele on first haplotype
- 0|1: Heterozygous variants with alternative allele on second haplotype

Examples

```
vcf_df <- data.frame(
  CHROM = c("chr1", "chr1"),
  POS = c(1000, 2000),
  REF = c("A", "C"),
  ALT = c("G", "T"),
  sample1 = c("1|0:0.5", "0|1:0.5")
)
snp_lists <- vcf_to_snp_list(vcf_df, "sample1")
```

Index