Package: bixverse (via r-universe)

October 14, 2026

Title Rust-accelerated standard bioinformatics, computational biology workflows

Version 0.7.0

Description Your favourite methods used in bioinformatics and computational biology made VERY fast via Rust. Contains various gene set enrichment methods, graph- based methods, co-expression module detection and much more. Additionally, has now a single cell analysis suite, enabling million cell scale analysis on local compute.

License MIT + file LICENSE

Encoding UTF-8

URL https://gregorlueg.github.io/bixverse

Roxygen list(markdown = TRUE)

Config/rextendr/version 0.5.0

Imports arrow, checkmate, cli, data.table, DBI, dplyr, duckdb (>= 1.5.5), fastcluster, ggplot2, igraph, jsonlite, lifecycle, magrittr, manifoldsR (>= 0.3.5), matrixStats, Matrix, mirai, patchwork, purrr, R.utils, R6, rextendr, rhdf5, rlang, S7, stringr, zeallot

Suggests airway, biomaRt, BiocParallel, bixverse.plots, cluster, e1071, edgeR, fgsea, GSVA, infotheo, knitr, limma, microbenchmark, mitch, methods, ontologyIndex, ontologySimilarity, qs2, quarto, singscore, scrapper, SingleCellExperiment, SummarizedExperiment, StatMatch, Seurat, tictoc, tinytest, furrr, future

SystemRequirements Cargo (Rust's package manager), rustc >= 1.90, xz

Depends R (>= 4.2)

LazyData true

Config/roxygen2/version 8.1.0

Config/pak/sysreqs cmake libglpk-dev libicu-dev libxml2-dev libssl-dev xz-utils libclang-dev

Repository https://gregorlueg.r-universe.dev

Date/Publication 2026-10-14 21:51:11 UTC

RemoteUrl https://github.com/GregorLueg/bixverse

RemoteRef main

RemoteSha d543471b7464b6dddc9a3e660abfca3252166593

Contents

%>>% . . . 15
add_adt_counts_sc . . . 15
add_new_metadata . . . 16
add_nhoods_info . . . 17
add_sc_new_obs . . . 18
add_snf_data_modality . . . 19
add_symphony_labels . . . 20
AnnDataParser . . . 21
apply_pipeline . . . 23
apply_pipeline_per_group . . . 24
archive_sc_exp . . . 25
as_ligand_target_influence . . . 26
assert_sc_state . . . 26
assign_sc_type . . . 27
aucell_sc . . . 29
batch_correction_bulk_dge . . . 30
bbknn_sc . . . 31
binarise_regulon_activity . . . 33
blitzgsea_calibrate . . . 34
bonsai_sc . . . 35
build_regulons . . . 37
build_symphony_ref . . . 39
bulk_dge_from_h5ad . . . 41
BulkCoExp . . . 42
BulkDge . . . 43
c_pca_plot_alphas . . . 44
calc_blitzgsea . . . 45
calc_diffusion_coordinates . . . 47
calc_fgsea . . . 48
calc_fgsea_simple . . . 49
calc_gsea_traditional . . . 51
calc_gsva . . . 52
calc_knn_metrics . . . 54
calc_manifold_metrics . . . 55
calc_meta_cell_compactness . . . 56
calc_meta_cell_purity . . . 57
calc_meta_cell_separation . . . 58
calc_mitch . . . 59
calc_sc_type_scores . . . 60

calc_singscore . . . 61
calc_singscore_multi . . . 62
calc_singscore_rank . . . 64
calc_ssgsea . . . 65
calculate_all_dges . . . 66
calculate_batch_asw_sc . . . 67
calculate_cell_type_asw_sc . . . 68
calculate_critical_value . . . 69
calculate_dge_hedges . . . 70
calculate_dge_limma . . . 71
calculate_diffusion_auc . . . 73
calculate_effect_size . . . 74
calculate_graph_connectivity_sc . . . 75
calculate_information_content . . . 76
calculate_integration_metrics_sc . . . 77
calculate_kbet_sc . . . 78
calculate_lisi_sc . . . 80
calculate_pca_adt_sc . . . 81
calculate_pca_bulk_dge . . . 82
calculate_pca_sc . . . 83
calculate_pcr_sc . . . 85
calculate_rpkm . . . 86
calculate_semantic_sim . . . 86
calculate_semantic_sim_mat . . . 87
calculate_semantic_sim_onto . . . 88
calculate_sparsity_stats . . . 89
calculate_tom . . . 90
calculate_tom_from_exp . . . 91
calculate_tpm . . . 93
calculate_wang_sim . . . 94
calculate_wang_sim_mat . . . 95
calculate_wang_sim_onto . . . 95
call_doublets_manual . . . 96
cell_cycle_genes . . . 97
cellphonedb_sc . . . 98
cellsweep_sc . . . 99
change_gene_identifier . . . 101
check_sc_state . . . 102
community_detection . . . 103
compute_expression_info_sc . . . 104
consensus_nmf_bulk . . . 106
consensus_nmf_sc . . . 107
constrained_page_rank . . . 109
constrained_page_rank_ls . . . 110
contrastive_pca . . . 112
contrastive_pca_processing . . . 113
cor_module_check_epsilon . . . 114
cor_module_coremo_clustering . . . 115

cor_module_coremo_cor_sign . . . 116
cor_module_coremo_eigengene . . . 117
cor_module_coremo_stability . . . 118
cor_module_graph_check_res . . . 119
cor_module_graph_final_modules . . . 120
cor_module_processing . . . 122
cor_module_tom . . . 123
demo_single_cells . . . 124
densmap_sc . . . 125
densne_sc . . . 127
detect_adt_isotypes . . . 128
dgrdl_grid_search . . . 129
dgrdl_result . . . 130
dialogue_sc . . . 132
diffcor_module_processing . . . 134
diffuse_seed_nodes . . . 135
doublet_detection_boost_sc . . . 136
download_cd34_data . . . 137
download_cistarget_hg38 . . . 138
download_demuxlet_pbmc . . . 139
download_dialogue_uc . . . 140
download_kang_pbmc . . . 141
download_marrow_cd34 . . . 142
download_pbmc_1k_5p . . . 142
download_pbmc_batches . . . 143
download_pbmc_totalseq_data . . . 144
download_pbmc3k . . . 144
download_pbmc8k . . . 145
download_thymus_ageing . . . 146
drop_cols_sc . . . 147
extract_binary_heatmap_data . . . 147
extract_dot_plot_data . . . 149
extract_embedding_data . . . 150
extract_feature_pair . . . 151
extract_feature_plot_data . . . 152
extract_gene_expression . . . 153
extract_gene_violin_data . . . 154
extract_milo_plot_data . . . 155
extract_paga_plot_data . . . 156
f1_score_confusion_mat . . . 158
fast_cluster_sc . . . 159
fast_ica_rust . . . 160
fast_ica_rust_helper . . . 161
fast_mnn_sc . . . 163
fgsea_go_elim . . . 165
fgsea_simple_go_elim . . . 166
filter_similarities . . . 167
find_all_markers_sc . . . 168

find_clusters_sc . . . 170
find_hvg_batch_aware_sc . . . 171
find_hvg_sc . . . 172
find_markers_sc . . . 174
find_neighbours_sc . . . 175
find_rbh_communities . . . 177
find_specific_markers_sc . . . 178
find_threshold_otsu . . . 180
fit_residuals_sc . . . 181
fix_meta_data_column . . . 182
flag_cells . . . 183
forceatlas2_sc . . . 184
gene_set_proportions_sc . . . 186
GeneOntologyElim . . . 187
generate_bt_meta_cells_sc . . . 188
generate_cellsweep_test_data . . . 189
generate_dialogue_test_data . . . 190
generate_gene_module_data . . . 192
generate_hotspot_membership . . . 193
generate_knn_sc . . . 193
generate_ligand_target_influence . . . 195
generate_personalisation_vec . . . 196
generate_rbh_graph . . . 197
generate_sc_knn . . . 198
generate_seacells_sc . . . 199
generate_single_cell_test_data . . . 201
generate_supercells_sc . . . 202
generate_wnn_graph_sc . . . 204
get_adt_feature_info . . . 206
get_adt_names . . . 207
get_adt_sample_info . . . 208
get_available_embeddings . . . 208
get_best_model . . . 209
get_best_run . . . 210
get_c_pca_factors . . . 211
get_c_pca_loadings . . . 212
get_cell_indices . . . 212
get_cell_info . . . 213
get_cell_names . . . 214
get_cell_ranger_params . . . 215
get_cellphonedb_db . . . 216
get_cells_to_keep . . . 217
get_centroids_sc . . . 217
get_cistarget_res . . . 218
get_cor_graph . . . 220
get_data . . . 221
get_dge_effect_sizes . . . 222
get_dge_limma_voom . . . 223

get_dge_list . . . 224
get_dge_qc_plot . . . 224
get_diagnostics . . . 225
get_diffcor_graph . . . 226
get_differential_abundance_res . . . 227
get_diffusion_perms . . . 228
get_diffusion_vector . . . 229
get_embedding . . . 230
get_epsilon_res . . . 231
get_factors . . . 231
get_feature_mat . . . 232
get_fpkm_counts . . . 233
get_gene_indices . . . 234
get_gene_lengths . . . 235
get_gene_names . . . 235
get_gene_names_from_idx . . . 236
get_go_data_human . . . 237
get_grid_search_res . . . 238
get_h . . . 238
get_h5ad_dimensions . . . 239
get_hotspot_membership . . . 240
get_hvg . . . 241
get_hvg_data_sc . . . 241
get_ica_stability_res . . . 242
get_index_cells . . . 243
get_inflection_point . . . 244
get_influence . . . 245
get_kmeans_clusters . . . 245
get_knn_dist . . . 246
get_knn_mat . . . 247
get_knn_obj . . . 248
get_magic . . . 249
get_marker_comparisons . . . 250
get_marker_summary . . . 250
get_meta_cell_purity . . . 251
get_metadata . . . 253
get_miloR_abundances_sc . . . 254
get_model_fit . . . 256
get_modules . . . 256
get_nmf_gene_loadings . . . 257
get_nmf_modules . . . 258
get_nmf_sample_activity . . . 258
get_nmf_stability . . . 259
get_ontology_ancestry . . . 260
get_outputs . . . 261
get_params . . . 261
get_pca_factors . . . 263
get_pca_loadings . . . 264

get_pca_singular_val . . . 265
get_pseudobulked_sc . . . 266
get_rbh_res . . . 267
get_residual_fit . . . 267
get_resolution_res . . . 268
get_results . . . 269
get_sc_available_features . . . 270
get_sc_cache_status . . . 271
get_sc_counts . . . 272
get_sc_obs . . . 273
get_sc_var . . . 273
get_scores . . . 274
get_sim_matrix . . . 275
get_snf_adjcacency_mat . . . 276
get_snf_final_mat . . . 277
get_snf_params . . . 277
get_snn_graph . . . 278
get_stability . . . 279
get_symphony_hvg_names . . . 280
get_symphony_labels . . . 281
get_symphony_loadings . . . 282
get_symphony_z_corr . . . 283
get_tf_to_gene . . . 284
get_top_terms . . . 285
get_tpm_counts . . . 285
get_w . . . 286
gse_go_elim_method . . . 287
gse_go_elim_method_list . . . 288
gse_hypergeometric . . . 289
gse_hypergeometric_list . . . 291
harmony_sc . . . 292
harmony_v2_sc . . . 293
hedges_g_dge . . . 294
hotspot_autocor_sc . . . 295
hotspot_gene_cor_sc . . . 297
ica_evaluate_comp . . . 299
ica_optimal_ncomp . . . 301
ica_processing . . . 302
ica_stabilised_results . . . 302
identify_tf_to_genes . . . 304
install_agent_skill . . . 306
knn_graph_label_propagation . . . 307
lda_k_sweep . . . 308
leiden_clustering . . . 310
ligand_activity_scores . . . 311
load_existing . . . 312
load_go_human_data . . . 313
load_h5ad . . . 313

load_h5ad_norm . . . 315
load_mtx . . . 317
load_multi_h5ad . . . 318
load_multi_mtx . . . 320
load_multi_tenx_h5 . . . 322
load_r_data . . . 323
load_sce . . . 325
load_seurat . . . 327
load_tenx_h5 . . . 328
map_symphony_query . . . 330
meld_sc . . . 331
merge_meta_cells . . . 333
merge_sc_experiments . . . 334
merge_subset_obs . . . 336
meta_cells_per_group . . . 337
MetaCells . . . 339
module_scores_sc . . . 340
modules_from_loadings . . . 341
nebula_mc . . . 342
nebula_sc . . . 344
NetworkDiffusions . . . 347
new_adt_counts_clr . . . 348
new_adt_counts_dsb . . . 349
new_sc_knn . . . 350
nmf_bulk . . . 351
nmf_k_sweep_bulk . . . 353
nmf_k_sweep_sc . . . 354
nmf_sc . . . 356
normalise_bulk_dge . . . 357
OntologySim . . . 358
ot_harmonic_score . . . 359
params_binary_heatmap . . . 359
params_blitzgsea . . . 361
params_boost . . . 362
params_bpca . . . 364
params_bulk_sparsity . . . 365
params_cistarget . . . 366
params_community_detection . . . 368
params_cor_graph . . . 369
params_coremo . . . 370
params_dgrdl . . . 371
params_dialogue_hlm . . . 372
params_dialogue_pmd . . . 373
params_dialogue_refine . . . 376
params_edger_ql . . . 377
params_fast_cluster_default . . . 378
params_graph_resolution . . . 379
params_gsea . . . 380

params_gsva . . . 381
params_hvg_defaults . . . 382
params_hvg_scran_defaults . . . 382
params_ica_general . . . 383
params_ica_ncomp . . . 384
params_ica_randomisation . . . 384
params_kmeans_defaults . . . 385
params_knn_defaults . . . 386
params_label_propagation . . . 387
params_lda . . . 388
params_ligand_target . . . 390
params_limma_voom . . . 391
params_meld . . . 393
params_module_membership . . . 394
params_nebula . . . 396
params_nmf_consensus . . . 398
params_nmf_hals . . . 399
params_norm_doublets_defaults . . . 400
params_pca_defaults . . . 401
params_ppca . . . 401
params_sc_apr . . . 402
params_sc_aucell . . . 403
params_sc_bbknn . . . 405
params_sc_bonsai . . . 406
params_sc_branch_selection . . . 408
params_sc_bt_metacells . . . 409
params_sc_cellphonedb . . . 410
params_sc_cellsweep . . . 411
params_sc_dsb . . . 414
params_sc_empty_droplets . . . 415
params_sc_fast_cluster . . . 416
params_sc_fastmnn . . . 418
params_sc_gene_trends . . . 419
params_sc_harmony . . . 421
params_sc_harmony_v2 . . . 423
params_sc_hotspot . . . 425
params_sc_hvg . . . 427
params_sc_knn . . . 428
params_sc_magic . . . 430
params_sc_miloR . . . 431
params_sc_min_quality . . . 432
params_sc_mtx_io . . . 433
params_sc_neighbours . . . 434
params_sc_palantir . . . 435
params_sc_pca . . . 437
params_sc_sctransform . . . 439
params_sc_seacells . . . 440
params_sc_seurat_cca . . . 442

params_sc_seurat_rpca . . . 444
params_sc_supercell . . . 445
params_sc_synthetic_cellsweep . . . 447
params_sc_synthetic_data . . . 448
params_sc_synthetic_data_adt . . . 450
params_sc_synthetic_dialogue . . . 451
params_sc_vision . . . 452
params_sc_wnn . . . 453
params_scdblfinder . . . 455
params_scenic . . . 458
params_scenic_binarise . . . 459
params_scenic_extra_trees_defaults . . . 460
params_scenic_gradient_boosting_defaults . . . 461
params_scenic_random_forest_defaults . . . 462
params_scrublet . . . 462
params_sctype_cells . . . 464
params_snf . . . 465
params_ssgsea . . . 466
params_symphony_map . . . 467
params_synthetic_bulk_rnaseq . . . 468
per_cell_qc_outlier . . . 471
permute_seed_nodes . . . 472
phate_sc . . . 473
plot.LdaKSweepResult . . . 474
plot.NmfKSweepResult . . . 475
plot_boxplot_normalization . . . 476
plot_epsilon_res . . . 477
plot_hvgs . . . 477
plot_ica_ncomp_params . . . 478
plot_ica_stability_individual . . . 479
plot_optimal_cuts . . . 480
plot_pca . . . 480
plot_pca_res . . . 481
plot_preprocessing_genes . . . 482
plot_preprocessing_outliers . . . 483
plot_rbf_impact . . . 484
plot_resolution_res . . . 484
plot_voom_normalization . . . 485
pre_process_sim_onto . . . 486
prepare_cell_markers . . . 487
preprocess_bulk_coexp . . . 487
preprocess_bulk_dge . . . 488
prescan_h5ad_files . . . 489
prescan_mtx_dirs . . . 490
prescan_tenx_h5_files . . . 492
prioritise_interactions . . . 493
process_go_data . . . 495
pseudobulk_dge_sc . . . 496

qc_bulk_dge . . . 497
RbhGraph . . . 498
read_h5ad_metadata . . . 500
read_h5ad_x_summary . . . 500
read_motif_annotation_file . . . 501
read_motif_ranking . . . 502
read_multi_tenx_h5_adt . . . 503
read_tenx_h5_adt . . . 504
read_tenx_h5_metadata . . . 505
relayout_bonsai . . . 506
remove_adt_isotypes . . . 507
remove_magic . . . 507
remove_residual_fit . . . 508
remove_samples . . . 509
rescue_cells . . . 510
reset_cells_to_keep . . . 510
robust_scale . . . 511
rs_2d_loess . . . 512
rs_batch_silhouette_width . . . 513
rs_blitzgsea_calibrate . . . 513
rs_blitzgsea_score . . . 514
rs_calc_norm_factors . . . 515
rs_cell_type_asw . . . 516
rs_cistarget . . . 517
rs_compare_knn . . . 518
rs_constrained_page_rank . . . 519
rs_constrained_page_rank_list . . . 520
rs_contrastive_pca . . . 521
rs_cor . . . 522
rs_cor_upper_triangle . . . 522
rs_cor2 . . . 523
rs_cos . . . 523
rs_count_zeroes . . . 524
rs_cov2cor . . . 524
rs_covariance . . . 525
rs_cpm . . . 525
rs_dense_to_upper_triangle . . . 526
rs_differential_cor . . . 526
rs_dist . . . 527
rs_edger_ql . . . 527
rs_fast_auc . . . 528
rs_fast_ica . . . 529
rs_fdr_adjustment . . . 530
rs_filter_by_expr . . . 530
rs_gene_trends . . . 531
rs_generate_ligand_target_influence . . . 532
rs_geom_elim_fgsea_simple . . . 533
rs_gower_dist . . . 534

rs_graph_connectivity . . . 535
rs_gse_geom_elim . . . 536
rs_gse_geom_elim_list . . . 537
rs_gsva . . . 538
rs_h5ad_data . . . 539
rs_hamming_dist . . . 540
rs_harmony . . . 540
rs_harmony_v2 . . . 541
rs_hedges_g . . . 541
rs_hypergeom_test . . . 542
rs_hypergeom_test_list . . . 543
rs_ica_iters . . . 544
rs_ica_iters_cv . . . 545
rs_jaccard_row_integers . . . 546
rs_kbet . . . 546
rs_knn_label_propagation . . . 547
rs_knn_mat_to_edge_list . . . 548
rs_leiden_clustering . . . 549
rs_ligand_activity_scores . . . 549
rs_limma_voom . . . 550
rs_lisi . . . 551
rs_mad_outlier . . . 552
rs_mc_aucell . . . 553
rs_mc_dialogue . . . 553
rs_mc_hotspot_autocor . . . 555
rs_mc_hotspot_gene_cor . . . 556
rs_mc_hvg . . . 557
rs_mc_pca . . . 559
rs_mc_scenic . . . 560
rs_mc_vision . . . 560
rs_mc_vision_with_autocorrelation . . . 561
rs_mitch_calc . . . 562
rs_mutual_info . . . 563
rs_nebula_mc . . . 564
rs_onto_semantic_sim . . . 565
rs_onto_semantic_sim_mat . . . 565
rs_onto_sim_wang . . . 566
rs_onto_sim_wang_mat . . . 567
rs_ot_harmonic_sum . . . 567
rs_page_rank . . . 568
rs_page_rank_parallel . . . 569
rs_pcr . . . 570
rs_phyper . . . 570
rs_pointwise_mutual_info . . . 571
rs_prcomp . . . 572
rs_prepare_whitening . . . 572
rs_random_svd . . . 573
rs_range_norm . . . 574

rs_rank_matrix_col . . . 574
rs_rank_matrix_col_stable . . . 575
rs_rbf_function . . . 575
rs_rbf_function_mat . . . 576
rs_rbh_cor . . . 576
rs_rbh_sets . . . 577
rs_remove_batch_effect . . . 578
rs_sc_knn . . . 579
rs_sc_knn_w_dist . . . 580
rs_sc_snn . . . 580
rs_set_similarity . . . 581
rs_set_similarity_list . . . 582
rs_set_similarity_list2 . . . 582
rs_singscore_multi . . . 583
rs_singscore_permutation_test . . . 584
rs_singscore_single . . . 585
rs_snf . . . 586
rs_sparse_dict_dgrdl . . . 586
rs_sparse_dict_dgrdl_grid_search . . . 587
rs_spatial_fdr . . . 589
rs_spectral_clustering . . . 589
rs_spectral_clustering_sim . . . 590
rs_ssgsea . . . 591
rs_tied_diffusion_parallel . . . 591
rs_tom . . . 593
rs_upper_triangle_to_dense . . . 593
rs_upper_triangle_to_sparse . . . 594
rs_voom_normalise . . . 595
rs_wnn . . . 596
run_bpca . . . 597
run_cell_qc . . . 598
run_cell_qc_fixed . . . 599
run_cistarget . . . 600
run_edger_ql . . . 601
run_gene_trends_sc . . . 603
run_lda . . . 605
run_limma_voom . . . 606
run_magic_sc . . . 607
run_paga_sc . . . 609
run_palantir_sc . . . 610
run_ppca . . . 612
run_snf . . . 613
save_h5ad . . . 614
save_sc_exp_to_disk . . . 615
sc_knn_to_nearest_neighbours . . . 616
sc_pipeline . . . 616
scdblfinder_sc . . . 617
scenic_gene_filter_sc . . . 618

scenic_grn_sc . . . 619
score_clusters . . . 621
scrublet_sc . . . 622
sct_corrected_counts_sc . . . 624
set_bonsai_embedding . . . 625
set_magic . . . 626
set_residual_fit . . . 627
set_sc_new_obs_col . . . 628
set_sc_new_obs_col_multiple . . . 628
set_sc_new_var_cols . . . 629
setnames_sc . . . 630
seurat_cca_sc . . . 631
seurat_rpca_sc . . . 633
SimilarityNetworkFusion . . . 634
simplify_hypergeom_res . . . 636
simulate_dropouts . . . 637
SingleCellCountData . . . 638
SingleCells . . . 645
SingleCellsMultiModal . . . 647
SingleCellsSubset . . . 648
stabilised_nmf_bulk . . . 649
stabilised_nmf_sc . . . 650
step_bbknn_sc . . . 652
step_clusters_sc . . . 653
step_fast_mnn_sc . . . 654
step_harmony_sc . . . 655
step_harmony_v2_sc . . . 656
step_hvg_sc . . . 657
step_metacells_sc . . . 658
step_neighbours_sc . . . 659
step_pca_sc . . . 660
stream_h5ad . . . 661
SymphonyReference . . . 662
synthetic_bulk_cor_matrix . . . 664
synthetic_c_pca_data . . . 665
synthetic_signal_matrix . . . 666
test_nhoods . . . 667
tf_to_genes_correlations . . . 669
tf_to_genes_motif_enrichment . . . 671
tied_diffusion . . . 673
to_snake_case . . . 675
top_genes_perc_sc . . . 675
transfer_labels_symphony . . . 676
tsne_sc . . . 678
umap_sc . . . 680
update_metadata_values . . . 681
upper_triangle_to_sparse . . . 682
upper_triangular_sym_mat . . . 683

validate_pipeline 685
vision_sc 686
vision_w_autocor_sc 687
write_cellranger_output 689
write_h5ad_sc 690
write_h5ad_sc_dense 691

Index **693**

`%>>%` *Append a step to a pipeline*

Description

`%>>%` chains pipeline steps. Either side can be a `ScStep` or a `ScPipeline`; the result is always a `ScPipeline`.

Usage

```
lhs %>>% rhs
```

Arguments

`lhs`	`ScPipeline` or `ScStep`.
`rhs`	`ScStep`.

Value

A `ScPipeline`.

Examples

```
# either side may be a step, the result is always a pipeline
step_hvg_sc(hvg_no = 30L) %>>% step_pca_sc(no_pcs = 10L)
```

`add_adt_counts_sc` *Add ADT counts to* `SingleCellsMultiModal`

Description

This method allows you to add ADT counts to a `SingleCellsMultiModal`. Assumes the transcriptomics data has already been ingested and the cells to keep are known. The method subsets the ADT counts to the kept cells, applies the requested normalisation (CLR or DSB), populates the `"var_adt"` table in the DuckDB, and attaches an `ADTCounts` to the class.

Usage

```
add_adt_counts_sc(object, adt_counts, method = c("clr", "dsb"), ...)
```

Arguments

object	SingleCellsMultiModal class.
adt_counts	Numeric matrix. Cells x features matrix of raw ADT counts.
method	String. One of c("clr", "dsb"). Normalisation method.
...	Additional arguments forwarded to the normalisation constructor. For method = "clr": seurat_clr, clean_clr_counts, percentile. For method = "dsb": empty_drops, isotype_names, dsb_params, scale_factor, seed, verbose. See new_adt_counts_clr() and new_adt_counts_dsb().

Value

Returns a SingleCellsMultiModal with the ADT data added.

Examples

```
# a CLR-normalised ADT layer on top of an ingested RNA modality
rna <- generate_single_cell_test_data()
adt <- generate_single_cell_test_data_adt()
dir <- tempfile("bixverse_mm")
dir.create(dir)
object <- load_r_data(
  SingleCellsMultiModal(dir_data = dir),
  counts = rna$counts,
  obs = rna$obs,
  var = rna$var,
  sc_qc_param = params_sc_min_quality(min_unique_genes = 5L),
  .verbose = FALSE
)
object <- add_adt_counts_sc(object, adt_counts = adt$counts, method = "clr")
get_adt_names(object)

unlink(dir, recursive = TRUE, force = TRUE)
```

---

add_new_metadata *Replace the meta data*

Description

This function will replace the meta data within the given object

Usage

```
add_new_metadata(object, new_metadata, ...)
```

Arguments

`object`	The class
`new_metadata`	data.table. The new meta data you wish to add.
`...`	Additional arguments to parse to the functions.

Value

The object with updated metadata.

Examples

```
# swap in a metadata table that carries an extra batch column
set.seed(42)
counts <- matrix(rpois(60, 20), nrow = 10, ncol = 6)
rownames(counts) <- sprintf("gene_%i", 1:10)
colnames(counts) <- sprintf("sample_%i", 1:6)
meta <- data.table::data.table(
  sample_id = colnames(counts),
  case_control = rep(c("case", "control"), each = 3)
)
object <- BulkDge(raw_counts = counts, meta_data = meta)
new_meta <- data.table::copy(meta)[, batch := rep(c("b1", "b2"), 3)]
object <- add_new_metadata(object, new_metadata = new_meta)
head(S7::prop(object, "meta_data"))
```

`add_nhoods_info` *Add neighbourhood info on majority cell type*

Description

This function adds cell type composition information to the `nhoods_info` slot within the `miloR` object. For each neighbourhood, it calculates the proportion of the majority cell type and identifies which cell type is most abundant. This is useful for annotating differential abundance results with the cellular composition of each neighbourhood.

Usage

```
add_nhoods_info(x, cell_info)

## S3 method for class 'miloR'
add_nhoods_info(x, cell_info)
```

Arguments

`x`	`miloR` object on which to tag on additional neighbourhood information.
`cell_info`	Character vector. Represents the cell type annotations you wish to add to the different neighbourhoods. Must be the same length as the number of cells (rows) in the nhoods matrix.

Value

Modified `miloR` object with updated `nhoods_info` containing `majority_celltype` and `majority_prop` columns.

Examples

```
# tag each neighbourhood with its majority cell type
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 500L,
    n_genes = 50L,
    n_samples = 6L,
    sample_bias = "even"
  )
)
milo <- get_miloR_abundances_sc(
  sc,
  sample_id_col = "sample_id",
  miloR_params = params_sc_miloR(k_refine = 10L),
  .verbose = FALSE
)
design_df <- data.frame(
  grp = rep(c("a", "b"), each = 3),
  row.names = sprintf("sample_%i", 1:6)
)
milo <- test_nhoods(milo, design = ~grp, design_df = design_df)
milo <- add_nhoods_info(milo, cell_info = get_sc_obs(sc)$cell_grp)
head(get_differential_abundance_res(milo))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`add_sc_new_obs` *Add an obs table derived from a method to the SingleCells.*

Description

Add an obs table derived from a method to the SingleCells.

Usage

```
add_sc_new_obs(object, obs_data)
```

Arguments

`object`	`SingleCells` class.
`obs_data`	data.table. A data.table you generated with `get_data()` on some sub class.

Value

The class with updated obs table in the DuckDB

Examples

```
# cluster memberships carry their own cell_idx, so they join straight on
sc <- demo_single_cells()
clusters <- fast_cluster_sc(sc, resolutions = 1.0, .verbose = FALSE)
sc <- add_sc_new_obs(sc, get_data(clusters))
head(get_sc_obs(sc), 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

add_snf_data_modality

Add a data modality for SNF generation

Description

This function will add a modality as an affinity matrix for subsequent fusion to the object. To note: all of the modalities need to have the same number of rows, i.e., samples!

Usage

```
add_snf_data_modality(object, data, data_name, params = NULL)
```

Arguments

`object`	The underlying class, see `SimilarityNetworkFusion()`.
`data`	matrix or data.table. The data to transform into adjacency data and add to the class. Any data supplied will be assumed to be samples x features. The provided data type can be a data.table (for categorical and/or mixed types) or a matrix (for continous types). If you provide a data.table, the function will assume the first column are the sample identifiers. Please ensure that setting.
`data_name`	String. The data modality name.
`params`	Optional List. If you wish to overwite the already set up parameters for SNF, see `params_snf()`. If `NULL`, the settings from within the object will be used. If not NULL, the new parameters will be used for this modality specifically and only for this modality!

Value

The class with added adjacency matrix for this data.

Examples

```
# add a categorical clinical modality to a continuous one
set.seed(42)
continuous <- matrix(rnorm(120), nrow = 12, ncol = 10)
rownames(continuous) <- sprintf("sample_%02i", 1:12)
colnames(continuous) <- sprintf("feature_%i", 1:10)
clinical <- data.table::data.table(
  sample_id = rownames(continuous),
  sex = factor(sample(c("M", "F"), 12, replace = TRUE)),
  stage = factor(sample(c("I", "II", "III"), 12, replace = TRUE))
)
object <- SimilarityNetworkFusion(
  data = continuous,
  data_name = "continuous",
  snf_params = params_snf(k = 3L)
)
object <- add_snf_data_modality(object, clinical, data_name = "clinical")
dim(get_snf_adjcacency_mat(object, name = "clinical"))
```

---

add_symphony_labels *Add labels to a Symphony reference post-hoc*

Description

Reads one or more obs columns from a `SingleCells` object and stores them in the reference's `labels` slot for use by `transfer_labels_symphony()`. The provided `sc_object` must have `cells_to_keep` matching the cells used at reference-build time; this is enforced by a length check against `nrow(z_corr)`.

Usage

```
add_symphony_labels(reference, sc_object, columns, overwrite = FALSE)
```

Arguments

`reference`	`SymphonyReference`.
`sc_object`	`SingleCells` to read labels from. Typically the same object passed to `build_symphony_ref()`.
`columns`	Character vector of obs column names.
`overwrite`	Boolean. If `TRUE`, existing label columns of the same name are replaced; otherwise an error is raised on collision.

Value

The `reference` with updated `labels`.

Examples

```
# attach obs labels to a reference built without them
ref <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 500L,
    n_genes = 50L,
    n_batches = 2L
  )
)
symphony_ref <- build_symphony_ref(
  ref,
  batch_column = "batch_index",
  hvg = get_hvg(ref) + 1L,
  harmony_params = params_sc_harmony(k = 10L),
  no_pcs = 10L,
  .verbose = FALSE
)
symphony_ref <- add_symphony_labels(
  symphony_ref,
  sc_object = ref,
  columns = "cell_grp"
)
head(get_symphony_labels(symphony_ref))

unlink(ref@dir_data, recursive = TRUE, force = TRUE)
```

`AnnDataParser` *Class for Anndata*

Description

This class helps dealing with h5ad objects from Python AnnData. You have several options will allow for retrieval of the underlying data.

Methods

Public methods:

- `AnnDataParser$new()`
- `AnnDataParser$get_obs_table()`
- `AnnDataParser$get_var_info()`
- `AnnDataParser$get_raw_counts()`
- `AnnDataParser$get_key_data()`
- `AnnDataParser$clone()`

`AnnDataParser$new()`: Initialises the Anndata Parser.

Usage:

```
AnnDataParser$new(h5_path)
```

Arguments:

`h5_path` String. Path to the h5 file.

Returns: Returns the initialised class.

`AnnDataParser$get_obs_table()`: Returns the observation table with all the data from the h5ad file.

Usage:

```
AnnDataParser$get_obs_table()
```

Returns: data.table. The found observations are returned. The pandas index will be named `sample_id`. Remaining columns (if found) will be returned as factors due to the way the data is stored in h5.

`AnnDataParser$get_var_info()`: Returns the variable table with all the data from the h5ad file.

Usage:

```
AnnDataParser$get_var_info()
```

Returns: data.table. The found observations are returned. The pandas index will be named `var_id`. Remaining columns (if found) will be returned as factors due to the way the data is stored in h5.

`AnnDataParser$get_raw_counts()`: Returns the counts that are stored in `X` slot of the anndata object.

Usage:

```
AnnDataParser$get_raw_counts()
```

Returns: Returns the count matrix with samples = columns and rows = features.

`AnnDataParser$get_key_data()`: Wrapper function that returns a list of the stored count data and the metadata found in the h5ad file.

Usage:

```
AnnDataParser$get_key_data()
```

Returns: List with following elements:

- metadata - metadata from the respective h5ad file
- var_info - metadata on the variables from the respective h5ad file.
- counts - counts that were found in the h5ad file.

`AnnDataParser$clone()`: The objects of this class are cloneable with this method.

Usage:

```
AnnDataParser$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# parse counts and obs back out of a small h5ad file
set.seed(42)
counts <- matrix(rpois(50, 5), nrow = 10, ncol = 5)
obs <- data.table::data.table(sample_id = sprintf("cell_%i", 1:10))
var <- data.table::data.table(var_id = sprintf("gene_%i", 1:5))
h5_path <- tempfile(fileext = ".h5ad")
write_h5ad_sc_dense(h5_path, counts, obs, var, .verbose = FALSE)
parser <- AnnDataParser$new(h5_path)
dim(parser$get_raw_counts())
head(parser$get_obs_table())
unlink(h5_path)
```

---

apply_pipeline *Apply a pipeline to a single cell object*

Description

Runs each step in order. The first step receives `object`; subsequent steps receive the result of the previous step. The chain is validated against the class of `object` before anything runs, see `validate_pipeline()`. Errors propagate; nothing is caught.

Usage

```
apply_pipeline(pipeline, object)
```

Arguments

`pipeline`	`ScPipeline`.
`object`	`SingleCells`, `SingleCellsSubset` or `MetaCells`. Dispatch happens inside each step's underlying generic, so the same pipeline works on any class its steps have methods for.

Value

The object after all steps have run.

Examples

```
# HVG then PCA, run in order on a freshly loaded object
sc <- demo_single_cells(prepped = FALSE)
p <- step_hvg_sc(hvg_no = 30L, .verbose = FALSE) %>>%
  step_pca_sc(no_pcs = 10L, .verbose = FALSE)
sc <- apply_pipeline(p, sc)
dim(get_pca_factors(sc))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

apply_pipeline_per_group *Apply a pipeline independently to each group of a* SingleCells *object*

Description

Splits `object` by `group_col`, applies `pipeline` to each subset, and returns a named list of processed `SingleCellsSubsets`. Useful for per-sample / per-cell-type re-analysis where the same chain (HVG, PCA, neighbours, clusters, ...) is run on each group, e.g. sample-pure metacell generation followed by an external merge.

Usage

```
apply_pipeline_per_group(
  pipeline,
  object,
  group_col,
  groups = NULL,
  progress = FALSE
)
```

Arguments

`pipeline`	`ScPipeline`.
`object`	`SingleCells`.
`group_col`	String. Column in obs used to split.
`groups`	Optional character vector. Restrict to these group values; if `NULL`, all unique values of `group_col` are used.
`progress`	Boolean. Shall big progress messages be printed to the console. Defaults to `FALSE`.

Value

Named list of processed objects, names being the group values. Usually `SingleCellsSubset`, or `MetaCells` if the pipeline ends on `step_metacells_sc()`, in which case `merge_meta_cells()` puts them back together.

Examples

```
# the same chain re-run inside each cell type
sc <- demo_single_cells(prepped = FALSE)
p <- sc_pipeline() %>>% step_hvg_sc(hvg_no = 20L, .verbose = FALSE)
res <- apply_pipeline_per_group(p, sc, group_col = "cell_grp")
names(res)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`archive_sc_exp` *Archive the on-disk counts for cold storage*

Description

Compresses `counts_cells.bin` into `counts.bxa` with zstd. The gene-based file is not archived, as it is a transpose of the cell-based one. Normalised values are only kept where they cannot be recomputed from the raw counts. `load_existing()` restores both binaries automatically when it finds only the archive. Save the in-memory data first with `save_sc_exp_to_disk()`.

Usage

```
archive_sc_exp(object, level = 3L, remove_bins = TRUE, .verbose = TRUE)
```

Arguments

`object`	`SingleCells` class.
`level`	Integer. zstd compression level, between 1 and 22. Higher levels compress more and take longer; decompression speed barely changes.
`remove_bins`	Boolean. Delete `counts_cells.bin` and `counts_genes.bin` once the archive is written.
`.verbose`	Boolean. Controls verbosity of the function.

Value

A list with `n_cells`, `nnz`, `n_norm_stored` and `archive_bytes`, invisibly.

Examples

```
sc <- demo_single_cells()
dir <- sc@dir_data
save_sc_exp_to_disk(sc, type = "rds")
archive_sc_exp(sc, .verbose = FALSE)
sc <- load_existing(SingleCells(dir_data = dir), .verbose = FALSE)

unlink(dir, recursive = TRUE, force = TRUE)
```

as_ligand_target_influence *Wrap an existing ligand-target matrix as a* LigandTargetInfluence

Description

Turns a regulatory potential matrix built elsewhere, for example with nichenetr's `construct_ligand_target_matri` into a `LigandTargetInfluence` object, so it can be scored with `ligand_activity_scores()`.

Usage

```
as_ligand_target_influence(x, ligands_as_cols = FALSE, params = NULL)
```

Arguments

`x`	Numeric matrix with unique row and column names.
`ligands_as_cols`	Boolean. Set to `TRUE` for a genes x ligands matrix, which is the nichenetr layout. Defaults to `FALSE` (ligands x genes).
`params`	Optional list. Parameters used to build `x`, kept for the record. Defaults to `NULL`.

Value

A `LigandTargetInfluence` object. Each ligand is its own seed.

Examples

```
# genes x ligands, as nichenetr returns it
ltm <- matrix(
  c(0.5, 0.0, 0.1, 0.4, 0.3, 0.1),
  nrow = 3,
  dimnames = list(c("G1", "G2", "G3"), c("L1", "L2"))
)
inf <- as_ligand_target_influence(ltm, ligands_as_cols = TRUE)
ligand_activity_scores(inf, gene_sets = list(set_A = "G1"))
```

assert_sc_state *Assert that cached artefacts still match the object's state*

Description

The hard tier. Called at the entry of anything that hands cached indices to Rust or writes a derived result back into the object, where a stale artefact means either a crash in the binding layer or silently mis-aligned biology.

Usage

```
assert_sc_state(
  x,
  artefacts,
  modality = "rna",
  .var.name = checkmate::vname(x),
  add = NULL
)
```

Arguments

`x`	`SingleCells`, `SingleCellsSubset`, `MetaCells` or `SingleCellsMultiModal` class.
`artefacts`	Character vector. Artefacts to check, either an artefact kind (`"pca"`, `"knn"`, `"snn"`, `"magic"`) or an embedding name (`"umap"`). May be modality qualified (`"adt:pca"`).
`modality`	String. Modality to resolve unqualified names against. One of `c("rna", "adt", "atac", "wnn")`.
`.var.name`	Name of the checked object to print in assertions. Defaults to the heuristic implemented in checkmate.
`add`	Collection to store assertion messages. See `checkmate::makeAssertCollection()`.

Value

Invisibly `x` when the assertion holds, otherwise an error.

`assign_sc_type` *Assign cell types per cell based on ScType*

Description

The cluster-level ScType path (`score_clusters()`) stamps one label on every cell of a cluster, so a minority population sharing a Leiden community with a bigger one gets absorbed without anything flagging it. This runs the scoring per cell instead: the score matrix is smoothed over the sNN graph via label spreading (Zhou et al.), then each cell takes its own argmax. Cells whose best score falls below `score_floor` come back as `NA`.

Pass `cluster_col` to also get the per-cluster composition (purity, entropy, runner-up cell type) and the hybrid assignment, where clusters at or above `purity_threshold` keep the blanket cluster-level call and mixed clusters fall back to the per-cell calls.

Usage

```
assign_sc_type(
  object,
  sc_type_res,
  cluster_col = NULL,
  sctype_cell_params = params_sctype_cells(),
  .verbose = TRUE
)
```

Arguments

object	SingleCells.
sc_type_res	ScTypeResults, see calc_sc_type_scores().
cluster_col	Optional string. Name of the obs column with the cluster assignment. If provided, the composition and hybrid assignment are returned on top of the per-cell calls.
sctype_cell_params	List. Output of params_sctype_cells().
.verbose	Boolean or integer. Controls verbosity.

Value

An ScTypeCellResults results class.

References

Ianevski et al., Nat Comm, 2022. Zhou et al., NIPS, 2004.

Examples

```
# per cell calls from the ScType scores
sc <- demo_single_cells()
markers <- data.table::data.table(
  cell_type = rep(sprintf("cell_type_%i", 1:3), each = 10),
  gene_id = sprintf("gene_%02d", 1:30)
)
cell_markers <- prepare_cell_markers(obj = sc, marker_df = markers)
scores <- calc_sc_type_scores(
  sc,
  cell_marker_list = cell_markers,
  .verbose = FALSE
)
res <- assign_sc_type(sc, sc_type_res = scores, .verbose = FALSE)
table(res$assignments)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`aucell_sc` *Calculate AUC scores (akin to AUCell)*

Description

Calculates an AUC-type score akin to AUCell across the gene sets, see Aibar et al. Three statistics are on offer, all consuming the same within-cell ranking but weighting it differently. `"recovery"` (default) is the recovery-curve AUC under a rank cutoff, i.e. the AUCell statistic of Aibar, et al., and is top-heavy: only genes inside the top `max_rank` of the cell contribute. `"wilcox"` is the AUC derived from the Mann-Whitney U statistic over the full ranking; its null sits at 0.5 for any gene set size, which makes it a good fit for pathway activity. `"ap"` is average precision, the most top-heavy of the three, but its null tracks the gene set prevalence, so raw values are not comparable across gene sets of different size unless `standardise` is on.

Usage

```
aucell_sc(
  object,
  gs_list,
  aucell_params = params_sc_aucell(),
  streaming = NULL,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells`, `MetaCells` (or potentially other) class.
`gs_list`	Named list. The elements have the gene identifiers of the respective gene sets.
`aucell_params`	List with the AUCell parameters, see `params_sc_aucell()` with the following elements: • auc_type - String. Which statistic to calculate. One of `c("recovery", "wilcox", "ap")`. `"recovery"` is the SCENIC one. • max_rank - Optional numeric. Rank cutoff for `"recovery"`. If `NULL`, the top 5% of the gene universe is used. Ignored by the other statistics. • standardise - Boolean. Shall each gene set's scores be z-scored across the cells.
`streaming`	Optional Boolean. Shall the data be streamed in. Useful for larger data sets where you wish to avoid loading in the whole data. If `NULL`, will automatically detect. Ignored when applied to `MetaCells`.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

AUCell results in form of a matrix that is cells x gene sets or as `ScMatrixRes` pending the input.

References

Aibar, et al., Nat Methods, 2017

Examples

```
# recovery curve AUC for two marker programmes
sc <- demo_single_cells()
gs_list <- list(
  type_1 = get_gene_names(sc)[1:10],
  type_2 = get_gene_names(sc)[11:20]
)
res <- aucell_sc(sc, gs_list = gs_list, .verbose = FALSE)
dim(res)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

batch_correction_bulk_dge

Run a linear batch correction

Description

Runs a linear batch correction over the data regressing out batch effects and adding `normalised_counts_corrected` to the object. Should these counts be found by `calculate_dge_hedges()`, they will be used for calculations of effect sizes based on Hedge's G.

Usage

```
batch_correction_bulk_dge(
  object,
  contrast_column,
  batch_col,
  scale_genes = FALSE,
  no_hvg_genes = 2500L
)
```

Arguments

`object`	The underlying class, see `BulkDge()`.
`contrast_column`	String. The contrast column in which the groupings are stored. Needs to be found in the meta_data within the properties.
`batch_col`	String. The column in which the batch effect groups can be found.

scale_genes Boolean. Shall the log(cpm) counts be scaled prior the PCA calculation. Defaults to FALSE.

no_hvg_genes Integer. Number of highly variable genes to include. Defaults to 2500.

Value

Returns the class with additional data added to the outputs.

Examples

```
# regress out a batch column and keep the corrected counts
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50),
  batch = rep(c("b1", "b2"), times = 50)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- qc_bulk_dge(object, group_col = "case_control", .verbose = FALSE)
object <- normalise_bulk_dge(
  object,
  group_col = "case_control",
  .verbose = FALSE
)
object <- calculate_pca_bulk_dge(object, no_hvg_genes = 500L)
object <- batch_correction_bulk_dge(
  object,
  contrast_column = "case_control",
  batch_col = "batch",
  no_hvg_genes = 500L
)
get_outputs(object)$normalised_counts_corrected[1:3, 1:3]
```

---

bbknn_sc *Run BBKNN*

Description

This function implements the batch-balanced k-nearest neighbour algorithm from Polański, et al. Briefly, the algorithm generates a KNN index on a per batch basis and identifies the neighbours of cells for each individual index. Subsequently, it leverages UMAP connectivity calculations to reduce spurious connections. For more details, please refer to Polański, et al.

Usage

```
bbknn_sc(
  object,
  batch_column,
```

```
  no_neighbours_to_keep = 5L,
  embd_to_use = "pca",
  no_embd_to_use = NULL,
  bbknn_params = params_sc_bbknn(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`batch_column`	String. The column with the batch information in the obs data of the class.
`no_neighbours_to_keep`	Integer. Maximum number of neighbours to keep from the BBKNN algorithm. Due to generating neighbours for each batch, there might be a large number of generated neighbours. This will only keep the top `no_neighbours_to_keep` neighbours. Defaults to `5L`.
`embd_to_use`	String. The embedding to use. Atm, the only option is `"pca"`.
`no_embd_to_use`	Optional integer. Number of embedding dimensions to use. If `NULL` all will be used.
`bbknn_params`	A list, please see `params_sc_bbknn()`. The list has the following parameters: • neighbours_within_batch - Integer. Number of neighbours to consider per batch. • set_op_mix_ratio - Numeric. Mixing ratio between union (1.0) and intersection (0.0). • local_connectivity - Numeric. UMAP connectivity computation parameter, how many nearest neighbours of each cell are assumed to be fully connected. • trim - Optional integer. Trim the neighbours of each cell to these many top connectivities. May help with population independence and improve the tidiness of clustering. If `NULL`, it defaults to `10 * neighbours_within_batch`. • knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.
`seed`	Integer. Random seed.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

The object with added kNN matrix based on BBKNN and the graph based on the returned connectivities of the algorithm.

References

Polański, et al., Bioinformatics, 2020

Examples

```
# batch balanced kNN, replacing the graph the demo object carries
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 600L, n_genes = 50L, n_batches = 3L
  )
)
sc <- bbknn_sc(
  remove_knn(sc),
  batch_column = "batch_index",
  no_neighbours_to_keep = 5L,
  .verbose = FALSE
)
dim(get_knn_mat(sc))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

binarise_regulon_activity

Binarise regulon activity into on/off calls

Description

The last SCENIC step. Each regulon gets its own threshold derived from the shape of its AUC distribution across cells, and a cell counts as on when its score sits strictly above that threshold.

The thresholds come back alongside the calls, so you can inspect them, override any that look wrong and re-apply the comparison yourself. SCENIC does the same, it writes them to an editable file between scoring and assignment.

Regulons flagged as not bimodal fell back to `mean + 2 * sd`. If most of your regulons land there, the AUC distributions are too flat to separate, which usually points at the scoring statistic rather than the thresholding. Check you used `auc_type = "recovery"`, see `params_sc_aucell()`.

Usage

```
binarise_regulon_activity(
  auc_matrix,
  binarise_params = params_scenic_binarise(),
  .verbose = TRUE
)
```

Arguments

auc_matrix	Numeric matrix of cells x regulons, or the ScMatrixRes returned by aucell_sc().
binarise_params	List. Output of params_scenic_binarise().
.verbose	Boolean. Controls verbosity of the function.

Value

A list with:

- binary - Logical matrix of cells x regulons, TRUE where the regulon is on.
- thresholds - data.table with one row per regulon, holding the threshold, whether it was bimodal and the number of cells called on.

References

Aibar, et al., Nat Methods, 2017

Examples

```
# on/off calls for a bimodal and a unimodal regulon
set.seed(7L)
auc <- cbind(
  regulon_a = c(rnorm(50, 0.1, 0.02), rnorm(50, 0.4, 0.02)),
  regulon_b = rnorm(100, 0.2, 0.05)
)
rownames(auc) <- sprintf("cell_%i", 1:100)
binarise_regulon_activity(auc, .verbose = FALSE)$thresholds
```

---

blitzgsea_calibrate *Calibrate the blitzGSEA null model for a signature*

Description

Draws random gene sets across a log-spaced grid of set sizes, fits gamma tails to the resulting enrichment scores and smooths the fitted parameters across sizes, see Lachmann, et al. No gene set library enters here, so one calibration serves every library you score against that signature. This is where nearly all of the runtime sits; scoring afterwards costs one gamma tail evaluation per pathway.

The returned null is a plain list, so it survives saveRDS() and can be cached against a signature.

Usage

```
blitzgsea_calibrate(stats, blitz_params = params_blitzgsea())
```

Arguments

stats	Named numeric vector. The gene level statistic. Sorted internally, so the order you hand it in does not matter.
blitz_params	List. The blitzGSEA parameters, see params_blitzgsea() wrapper function.

Value

An object of class BlitzGseaNull, a list with the following elements:

- anchor_sizes - Numeric vector. The anchor set sizes, ascending.
- shape_pos - Numeric vector. Smoothed positive-tail gamma shape.
- scale_pos - Numeric vector. Smoothed positive-tail gamma scale.
- shape_neg - Numeric vector. Smoothed negative-tail gamma shape.
- scale_neg - Numeric vector. Smoothed negative-tail gamma scale.
- pos_ratio - Numeric vector. Smoothed fraction of positive null scores at each anchor.
- ks_pos - Float. Mean goodness-of-fit p-value for the positive tail.
- ks_neg - Float. Mean goodness-of-fit p-value for the negative tail.
- centred - Boolean. Whether the signature was centred.
- n_genes - Integer. Size of the signature it was calibrated on.

References

Lachmann, et al., Bioinformatics, 2022

Examples

```
# calibrate a null once, to be reused across gene set libraries
set.seed(42L)
stats <- stats::setNames(rnorm(500), sprintf("gene_%03i", 1:500))
null_model <- blitzgsea_calibrate(
  stats,
  blitz_params = params_blitzgsea(permutations = 1000L, anchors = 10L)
)
null_model
```

---

bonsai_sc *Build a Bonsai tree over the cells*

Description

Bonsai reconstructs a tree over the cells in which every cell is a leaf and the internal nodes are inferred ancestral states, with branch lengths that carry the amount of change between them. Unlike a kNN graph it uses each measurement's error bar, which is what Sanity provides: the raw counts go through Sanity first, for posterior log fold changes with error bars, and Bonsai builds the tree on those. The tree is then laid out in 2D. For details, please refer to de Groot, et al. and Breda, et al.

No HVG selection needed. By default every gene goes in, streamed through Sanity in chunks, and only the genes with enough signal over their own noise (`min_signal_to_noise` in `params_sc_bonsai()`) are kept for the tree. That is the gene selection of the Bonsai paper, and it keeps memory at one chunk plus the survivors. Pass `hvg` to restrict the candidates.

Runtime grows a little faster than linearly with the number of cells. In bonsai-rs's own benchmarks the search took about 70 seconds at 10,000 cells and 210 seconds at 25,000 on ten cores. Sanity comes on top, linear in the number of genes it has to fit; on the CPU that is the larger share once all genes go in.

The object itself is not touched. Store the leaf coordinates with `set_bonsai_embedding()` if you want them next to the other embeddings.

On `MetaCells` every metacell is a leaf. Their raw counts are sums over their cells, and a sum of Poisson counts is Poisson, so Sanity treats a metacell exactly as it treats a cell: its total counts are the library size, and a bigger metacell gets tighter error bars, which Bonsai weights by. That is the way to large data: 100k cells in metacells of 50 is a 2,000-leaf tree. What you give up is any structure inside a metacell.

Usage

```
bonsai_sc(
  object,
  hvg = NULL,
  bonsai_params = params_sc_bonsai(),
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `MetaCells` class.
`hvg`	Optional integer. Restrict the candidate genes to these, e.g. the output of `get_hvg()` plus one. Please provide 1-indexed genes here! If `NULL`, every gene in the object is a candidate.
`bonsai_params`	List. See `params_sc_bonsai()`.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

A `BonsaiTree` S3 object with:

- nodes - data.table with `node`, `parent` (`NA` for the root), `branch`, `is_leaf`, `cell_id` (`NA` for the inferred ancestors), `n_cells` (the cells behind a leaf: 1 for single cells, the originating cells for a metacell; `NA` for the ancestors), `x` and `y`. The leaves come first, in the order of the object's cells or metacells.
- loglik - The loglikelihood of the final tree.
- steps - data.table with the loglikelihood after each search step and the step's wall time in seconds.
- timings - data.table with the wall time in seconds of each stage: `sanity`, `ingest`, `bonsai` (the whole search), `layout`, and `total`, the whole call as R saw it.
- genes_used - The genes the tree was built on.
- genes_dropped - The candidate genes left out: no counts in the cells, ill-conditioned Sanity posteriors, or a signal-to-noise ratio below `min_signal_to_noise`.
- cell_idx - The cells or metacells the tree was built over (0-indexed).
- layout, hyperbolic - The current layout.
- params - The parameters of the run.

References

de Groot, et al., Nat. Biotechnol., 2026; Breda, et al., Nat. Biotechnol., 2021.

Examples

```
# a tree over the demo cells, genes selected by their signal-to-noise
sc <- demo_single_cells(
  prepped = FALSE,
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 50L)
)
tree <- bonsai_sc(sc, .verbose = FALSE)
tree

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`build_regulons` *Build the final regulons*

Description

Turns the TF to gene table into the gene sets you hand to `aucell_sc()`. `tf_to_genes_motif_enrichment()` only flags which pairs survived the CisTarget leading edge, it does not remove anything, so this is the step that applies that filter.

Following SCENIC, the TF is added back to its own target list and regulons below `min_genes` are dropped. Note the CisTarget leading edge usually cuts hard: a module of a few hundred candidate targets typically ends up as a regulon of a few dozen.

Usage

```
build_regulons(
  x,
  use_leading_edge = TRUE,
  add_tf = TRUE,
  min_genes = 10L,
  mode = c("activating", "repressing", "both"),
  .verbose = TRUE
)

## S3 method for class 'ScenicGrn'
build_regulons(
  x,
  use_leading_edge = TRUE,
  add_tf = TRUE,
  min_genes = 10L,
  mode = c("activating", "repressing", "both"),
  .verbose = TRUE
)
```

Arguments

`x`	`ScenicGrn` object. You need to have run `tf_to_genes_motif_enrichment()` for the leading edge filter to do anything.
`use_leading_edge`	Boolean. Shall only the TF to gene pairs inside the CisTarget leading edge be kept. Defaults to `TRUE`.
`add_tf`	Boolean. Shall the TF be added to its own regulon. Defaults to `TRUE`, following SCENIC.
`min_genes`	Integer. Regulons with fewer genes than this are dropped. Defaults to `10L`, the SCENIC value.
`mode`	String. Which links to keep, based on the `cor_sign` column added by `tf_to_genes_correlations()`. One of `c("activating", "repressing", "both")`. With `"both"` the regulon names get a `"_pos"` or `"_neg"` suffix so the two stay distinct. Defaults to `"activating"`.
`.verbose`	Boolean. Controls verbosity of the function.

Value

A named list of character vectors, one per regulon.

References

Aibar, et al., Nat Methods, 2017

Examples

```
# turn the signed TF to gene table into regulon gene sets
sc <- demo_single_cells()
grn <- scenic_grn_sc(
  sc,
  tf_ids = sprintf("gene_%02d", 1:5),
  scenic_params = params_scenic(
    min_counts = 1L,
    learner_params = list(n_trees = 20L)
  ),
  .verbose = FALSE
)
grn <- identify_tf_to_genes(
  grn,
  method = "top_k",
  k_tfs = 3L,
  .verbose = FALSE
)
grn <- tf_to_genes_correlations(grn, object = sc, .verbose = FALSE)
lengths(build_regulons(
  grn,
  use_leading_edge = FALSE,
  min_genes = 3L,
  .verbose = FALSE
))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

build_symphony_ref *Build a Symphony reference from a SingleCells object*

Description

Runs sparse PCA over the provided HVGs, runs Harmony for batch correction, and compresses the result into the cached terms used at query time. The Harmony version is auto-detected from the class of `harmony_params`. Optionally snapshots one or more obs columns (e.g. cell type annotations) into the reference's `labels` slot for downstream label transfer.

Usage

```
build_symphony_ref(
  object,
  batch_column,
  additional_batch_columns = NULL,
  hvg,
  harmony_params = params_sc_harmony(),
  pca_params = params_sc_pca(),
  no_pcs = 30L,
```

```
  slim = FALSE,
  label_columns = NULL,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` (the reference).
`batch_column`	String. Primary batch column in the obs table.
`additional_batch_columns`	Optional character vector.
`hvg`	Integer vector. R-style 1-based HVG indices into the gene universe of `object`. Must be provided explicitly.
`harmony_params`	List. Output of `params_sc_harmony()` or `params_sc_harmony_v2()`.
`pca_params`	List. Output of `params_sc_pca()`.
`no_pcs`	Integer. Number of principal components.
`slim`	Boolean. If `TRUE`, drops `z_orig` and `r` from the returned reference. `z_corr` is always kept (needed for kNN label transfer).
`label_columns`	Optional character vector of obs column names to snapshot into the reference. Read in `cells_to_keep` order to align with `z_corr`. Required for `transfer_labels_symphony()` unless populated later via `add_symphony_labels()`.
`seed`	Integer.
`.verbose`	Boolean or integer.

Value

A SymphonyReference object.

Examples

```
# two batch reference, cell type labels snapshotted along the way
ref <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 500L,
    n_genes = 50L,
    n_batches = 2L,
    batch_effect_strength = "medium"
  )
)
symphony_ref <- build_symphony_ref(
  ref,
  batch_column = "batch_index",
  hvg = get_hvg(ref) + 1L,
  harmony_params = params_sc_harmony(k = 10L),
  no_pcs = 10L,
  label_columns = "cell_grp",
```

```
  .verbose = FALSE
)
symphony_ref

unlink(ref@dir_data, recursive = TRUE, force = TRUE)
```

`bulk_dge_from_h5ad` *Wrapper function to generate BulkDge object from h5ad*

Description

This is a helper function that can be used to create a `BulkDge` object (see `BulkDge()`) directly from h5ad objects.

Usage

```
bulk_dge_from_h5ad(h5_path, .verbose = TRUE)
```

Arguments

`h5_path`	String. Path to the h5ad object.
`.verbose`	Controls verbosity of the function

Value

`BulkDge` object.

Examples

```
# round trip a synthetic count matrix through a temporary h5ad
syn <- synthetic_bulk_cor_matrix()
h5_path <- tempfile(fileext = ".h5ad")
write_h5ad_sc_dense(
  f_path = h5_path,
  counts = t(syn$counts),
  obs = data.table::data.table(sample_id = colnames(syn$counts)),
  var = data.table::data.table(var_id = rownames(syn$counts)),
  .verbose = FALSE
)
object <- bulk_dge_from_h5ad(h5_path, .verbose = FALSE)
object

unlink(h5_path)
```

`BulkCoExp` *Bulk RNAseq co-expression modules*

Description

Class for applying various co-expression module detection methods on top of bulk RNAseq data.

Usage

```
BulkCoExp(raw_data, meta_data, variable_info = NULL)
```

Arguments

`raw_data`	The raw count matrix. Rows = samples, columns = features.
`meta_data`	data.table Metadata information on the samples. Expects to have a `sample_id` column.
`variable_info`	data.table. Metadata information on the features. This is an optional table.

Value

Returns the `BulkCoExp` class for further operations.

Properties

raw__data A numerical matrix of the provided raw data.

meta__data A data.table with the meta-information about the samples.

variable__info An optional data.table containing the variable info.

processed__data A list in which various types of processed data will be stored.

outputs A list in which key outputs will be stored.

params A (nested) list that will store all the parameters of the applied function.

final__results A data.table that will contain the final results.

Examples

```
# co-expression class over synthetic bulk counts (samples x genes)
syn <- synthetic_bulk_cor_matrix()
mat <- log1p(t(syn$counts))
meta <- data.table::data.table(sample_id = rownames(mat))
object <- BulkCoExp(raw_data = mat, meta_data = meta)
object
```

`BulkDge` *Bulk RNAseq differential gene expression class*

Description

Class for coordinating differential gene expression analyses with subsequent GSE in a structured format. The filtered counts, library sizes and normalisation factors are stored in the class; `get_dge_list()` turns them into an edgeR `DGEList` on demand.

Usage

```
BulkDge(
  raw_counts,
  meta_data,
  variable_info = NULL,
  alternative_gene_id = NULL
)
```

Arguments

`raw_counts`	matrix. The raw count matrix. Rows = genes, columns = samples. Note: this is different from the `BulkCoExp()` class!
`meta_data`	data.table. Metadata information on the samples. It expects to have a column sample_id and case_control column.
`variable_info`	data.table. Metadata information on the features. This is an optional table. Defaults to `NULL`.
`alternative_gene_id`	String. Optional alternative gene identifier to be used. Must be a column of variable_info!

Value

Returns the `BulkDge` class for further operations.

Properties

raw_counts A numerical matrix of the provided raw data.

meta_data A data.table with the meta-information about the samples.

variable_info An optional data.table containing the variable info.

outputs A list in which key outputs will be stored.

plots A list with the plots that are generated during subsequent QC steps.

params A (nested) list that will store all the parameters of the applied function.

final_results A list in which final results will be stored.

Examples

```
# DGE class over synthetic bulk counts (genes x samples)
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object
```

---

c_pca_plot_alphas *Plot various alphas for the contrastive PCA*

Description

This function will plot various alphas to highlight the most interesting alpha parameters akin to the implementation of contrastive PCA in Python.

Usage

```
c_pca_plot_alphas(
  object,
  label_column = NULL,
  min_alpha = 0.1,
  max_alpha = 100,
  n_alphas = 10L,
  .verbose = TRUE
)
```

Arguments

object	The underlying class, see BulkCoExp(). You need to apply contrastive_pca_processing() to the function for this method to work. Checkmate will raise errors otherwise.
label_column	An optional sample label column. Needs to exist in the meta_data of the BulkCoExp class.
min_alpha	Minimum alpha to test.
max_alpha	Maximum alpha to test.
n_alphas	Number of alphas to test. The function will generate a series of alphas from log(min_alpha) to log(max_alpha) to test out.
.verbose	Controls verbosity of function.

Value

A ggplot showing the impact of various alpha parameters on the samples in form of 2D plots.

References

Abid, et al., Nature Communications, 2018

Examples

```
# how the group separation responds to alpha
cpca_data <- synthetic_c_pca_data()
target <- t(cpca_data$target)
background <- t(cpca_data$background)
meta <- data.table::data.table(
  sample_id = rownames(target),
  grp = cpca_data$target_labels
)
obj <- BulkCoExp(target, meta)
obj <- preprocess_bulk_coexp(obj, .verbose = FALSE)
obj <- contrastive_pca_processing(obj, background, .verbose = FALSE)
c_pca_plot_alphas(obj, label_column = "grp", n_alphas = 6L, .verbose = FALSE)
```

---

calc_blitzgsea *Bixverse implementation of the blitzGSEA algorithm*

Description

Rust-based version of blitzGSEA, see Lachmann, et al. Instead of permuting per pathway, it calibrates a gamma null once for the signature and then reads each pathway's p-value straight off the fitted tail. That makes the cost independent of how many pathways you throw at it, which is where it pulls away from permutation methods on large libraries.

Hand back a `null_model` from `blitzgsea_calibrate()` to reuse a calibration across libraries. Left as `NULL`, one is calibrated on the fly.

Usage

```
calc_blitzgsea(
  stats,
  pathways,
  blitz_params = params_blitzgsea(),
  null_model = NULL
)
```

Arguments

`stats`	Named numeric vector. The gene level statistic.
`pathways`	List. A named list with each element containing the genes for this pathway.
`blitz_params`	List. The blitzGSEA parameters, see `params_blitzgsea()` wrapper function. This function generates a list containing: • min_size - Integer. Minimum size for the gene sets.

- max_size - Integer. Maximum size for the gene sets.
- permutations - Integer. Random gene sets per anchor size.
- anchors - Integer. Number of log-spaced anchor sizes.
- symmetric - Boolean. Pool both tails into a single gamma.
- centre - Boolean. Centre the signature before scoring.
- ks_test - Boolean. Run the goodness-of-fit diagnostic.
- seed - Float. Random seed.

`null_model` Optional `BlitzGseaNull` object from `blitzgsea_calibrate()`. Has to have been calibrated on the same signature and with the same `centre` setting. Defaults to `NULL`, in which case it is calibrated here.

Value

A data.table with the results from the blitzGSEA with the following columns:

- pathway_name - Character. The name of the pathway.
- es - Float. The enrichment score for this pathway.
- nes - Float. The normalised enrichment score, the signed normal quantile of the one-sided tail probability.
- pvals - Float. The two-sided p-value from the gamma approximation.
- fdr - Float. The Benjamini-Hochberg adjusted p-value.
- sidak - Float. The Sidak adjusted p-value.
- size - Integer. The size of the pathway after intersection.
- leading_edge - List of character vectors with the leading edge genes.

References

Lachmann, et al., Bioinformatics, 2022

Examples

```
# blitzGSEA reading p-values off a fitted gamma null
set.seed(42L)
stats <- stats::setNames(rnorm(500), sprintf("gene_%03i", 1:500))
stats[1:20] <- stats[1:20] + 2
pathways <- list(
  up_set = sprintf("gene_%03i", 1:20),
  random_set = sprintf("gene_%03i", 200:240)
)
res <- calc_blitzgsea(
  stats,
  pathways,
  blitz_params = params_blitzgsea(permutations = 1000L, anchors = 10L)
)
res[, c("pathway_name", "es", "nes", "pvals")]
```

`calc_diffusion_coordinates` *Calculate diffusion coordinates*

Description

To leverage the quality metrics from Persad, et al., we need the diffusion coordinates to then calculate if a cell is a dense or sparse region of the manifold, its compactness and separation to other meta cells. To do so, generate a diffusion map on the original data based on the approach of SEACells and add the data to the object, see Persad, et al.

Usage

```
calc_diffusion_coordinates(
  object,
  knn_data,
  n_dcs = 10L,
  k_density = 150L,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`MetaCells` class.
`knn_data`	`SingleCellNearestNeighbour` class. Contains the kNN graph from the original cells.
`n_dcs`	Integer. Number of diffusion coordinates to use. Defaults to `10L`.
`k_density`	Integer. The k-th neighbour to use for the density region estimation. Defaults to `150L`.
`seed`	Integer. Seed for reproducibility
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

The class with the diffusion map coordinates, density distance and region attached.

References

Persad, et al. Nat Biotechnol, 2023

Examples

```
# diffusion map off the source kNN graph, giving each meta cell a region
sc <- demo_single_cells()
mc <- generate_bt_meta_cells_sc(
  sc,
  sc_meta_cell_params = params_sc_bt_metacells(target_no_metacells = 50L),
  .verbose = FALSE
)
mc <- calc_diffusion_coordinates(
  mc,
  knn_data = get_knn_obj(sc),
  .verbose = FALSE
)
table(mc[["density_region"]]$density_region)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

calc_fgsea *Bixverse implementation of the fgsea algorithm*

Description

Rust-based version of the fgsea simple and multi-level algorithm. Initially, the simple method is run. For low p-values, the multi-level method is used to estimate lower p-values than possible just based on the permutations, see Korotkevich, et al.

Usage

```
calc_fgsea(
  stats,
  pathways,
  nperm = 1000L,
  gsea_params = params_gsea(),
  seed = 123L
)
```

Arguments

stats	Named numeric vector. The gene level statistic.
pathways	List. A named list with each element containing the genes for this pathway.
nperm	Integer. Number of permutation tests. Defaults to 1000L.
gsea_params	List. The GSEA parameters, see params_gsea() wrapper function. This function generates a list containing: • min_size - Integer. Minimum size for the gene sets. • max_size - Integer. Maximum size for the gene sets.

- gsea_param - Float. The GSEA parameter. Defaults to `1.0`.
- sample_size - Integer. Number of samples to iterate through for the multi-level implementation of fgsea.
- eps - Float. Boundary for calculating the p-value. Used for the multi-level implementation of fgsea.

`seed` Random seed for reproducibility.

Value

A data.table with the results from the GSEA with the following columns:

- es - Float. The enrichment score for this pathway.
- nes - Float. The normalised enrichment score for this pathway.
- pvals - Float. The p-value for this pathway.
- n_more_extreme - Integer. Number of permutation that had more extreme enrichment scores than the actual.
- size - Integer. The size of the pathway.
- pathway_name - Character. The name of the pathway.
- leading_edge - List of character vectors with the leading edge genes.
- fdr - Float. The adjusted pval.

References

Korotkevich, et al., bioRxiv

Examples

```
# fgsea with the multi-level refinement of low p-values
set.seed(42L)
stats <- stats::setNames(rnorm(500), sprintf("gene_%03i", 1:500))
stats[1:20] <- stats[1:20] + 2
pathways <- list(
  up_set = sprintf("gene_%03i", 1:20),
  random_set = sprintf("gene_%03i", 200:240)
)
res <- calc_fgsea(stats, pathways, nperm = 1000L)
res[, c("pathway_name", "es", "pvals")]
```

`calc_fgsea_simple` *Bixverse implementation of the simple fgsea algorithm*

Description

Rust-based version of the fgsea simple algorithm. This one is permutation- based and similar to the traditional implementation, but leverages some clear tricks to be way faster, see Korotkevich, et al.

Usage

```
calc_fgsea_simple(
  stats,
  pathways,
  nperm = 2000L,
  gsea_params = params_gsea(),
  seed = 123L
)
```

Arguments

`stats`	Named numeric vector. The gene level statistic.
`pathways`	List. A named list with each element containing the genes for this pathway.
`nperm`	Integer. Number of permutation tests. Defaults to `2000L`.
`gsea_params`	List. The GSEA parameters, see `params_gsea()` wrapper function. This function generates a list containing: • min_size - Integer. Minimum size for the gene sets. • max_size - Integer. Maximum size for the gene sets. • gsea_param - Float. The GSEA parameter. Defaults to `1.0`. • sample_size - Integer. Number of samples to iterate through for the multi-level implementation of fgsea. • eps - Float. Boundary for calculating the p-value. Used for the multi-level implementation of fgsea.
`seed`	Random seed for reproducibility.

Value

A data.table with the results from the GSEA with the following columns:

- es - Float. The enrichment score for this pathway.
- nes - Float. The normalised enrichment score for this pathway.
- pvals - Float. The p-value for this pathway.
- n_more_extreme - Integer. Number of permutation that had more extreme enrichment scores than the actual.
- size - Integer. The size of the pathway.
- pathway_name - Character. The name of the pathway.
- leading_edge - List of character vectors with the leading edge genes.
- fdr - Float. The adjusted pval.

References

Korotkevich, et al., bioRxiv

Examples

```
# fgsea simple against a signature with one enriched set
set.seed(42L)
stats <- stats::setNames(rnorm(500), sprintf("gene_%03i", 1:500))
stats[1:20] <- stats[1:20] + 2
pathways <- list(
  up_set = sprintf("gene_%03i", 1:20),
  random_set = sprintf("gene_%03i", 200:240)
)
res <- calc_fgsea_simple(stats, pathways, nperm = 1000L)
res[, c("pathway_name", "es", "pvals")]
```

---

calc_gsea_traditional

Bixverse implementation of the traditional GSEA algorithm

Description

Rust-based version of the traditional permutation-based GSEA algorithm.

Usage

```
calc_gsea_traditional(
  stats,
  pathways,
  nperm = 2000L,
  gsea_params = params_gsea(),
  seed = 123L
)
```

Arguments

`stats`	Named numeric vector. The gene level statistic.
`pathways`	List. A named list with each element containing the genes for this pathway.
`nperm`	Integer. Number of permutation tests. Defaults to `2000L`.
`gsea_params`	List. The GSEA parameters, see `params_gsea()` wrapper function. This function generates a list containing: • min_size - Integer. Minimum size for the gene sets. • max_size - Integer. Maximum size for the gene sets. • gsea_param - Float. The GSEA parameter. Defaults to `1.0`. • sample_size - Integer. Number of samples to iterate through for the multi-level implementation of fgsea. • eps - Float. Boundary for calculating the p-value. Used for the multi-level implementation of fgsea.
`seed`	Random seed for reproducibility.

Value

A data.table with the results from the GSEA with the following columns:

- es - Float. The enrichment score for this pathway.
- nes - Float. The normalised enrichment score for this pathway.
- pvals - Float. The p-value for this pathway.
- n_more_extreme - Integer. Number of permutation that had more extreme enrichment scores than the actual.
- size - Integer. The size of the pathway.
- pathway_name - Character. The name of the pathway.
- leading_edge - List of character vectors with the leading edge genes.
- fdr - Float. The adjusted pval.

Examples

```
# permutation GSEA against a signature with one enriched set
set.seed(42L)
stats <- stats::setNames(rnorm(500), sprintf("gene_%03i", 1:500))
stats[1:20] <- stats[1:20] + 2
pathways <- list(
  up_set = sprintf("gene_%03i", 1:20),
  random_set = sprintf("gene_%03i", 200:240)
)
res <- calc_gsea_traditional(stats, pathways, nperm = 1000L)
res[, c("pathway_name", "es", "pvals")]
```

---

calc_gsva *Bixverse implementation of GSVA*

Description

Implementation of the bixverse version of the gene set variation analysis (GSVA), see Hänzelmann, et al.

Usage

```
calc_gsva(
  exp,
  pathways,
  kernel = c("gaussian", "poisson", "none"),
  gaussian = deprecated(),
  gsva_params = params_gsva(),
  .verbose = FALSE
)
```

Arguments

`exp`	Numerical matrix. Rows represents the features, columns the features/genes.
`pathways`	List. A named list with each element containing the genes for this pathway.
`kernel`	String. One of `c("gaussian", "poisson", "none")`. The kernel to use.
`gaussian`	Boolean. If set to `TRUE` the Gaussian kernel will be used, if `FALSE` the Poisson will be used. [**Deprecated**]
`gsva_params`	List. The GSVA parameters, see `params_gsva()` wrapper function. This function generates a list containing: • tau - Float. The tau parameter of the algorithm. Large values will emphasise the tails more. Defaults to `1.0`. • min_size - Integer. Minimum size for the gene sets. • max_size - Integer. Maximum size for the gene sets. • max_diff - Boolean. Influences the scoring. If `TRUE` the difference will be used; if `FALSE`, the largest absolute value. • abs_rank - Boolean. If `TRUE` = pos-neg, `FALSE` = pos+neg for the internal calculations.
`.verbose`	Boolean. Controls verbosity.

Value

A matrix of shape pathways (that passed the thresholds) x samples.

References

see Hänzelmann, et al. Bmc Bioinformatics, 2013

Examples

```
# per-sample GSVA scores for two gene sets
set.seed(123L)
exp_mat <- matrix(
  rnorm(200 * 10),
  nrow = 200,
  dimnames = list(sprintf("gene_%03i", 1:200), sprintf("sample_%i", 1:10))
)
pathways <- list(
  set_a = sprintf("gene_%03i", 1:20),
  set_b = sprintf("gene_%03i", 50:80)
)
dim(calc_gsva(exp_mat, pathways))
```

`calc_knn_metrics` *Calculate recall at k and distance ratio*

Description

Helper function to compare the results of two `SingleCellNearestNeighbour` against each other. The first one can serve as a reference (ground truth) and you can compare against the second one.

Usage

```
calc_knn_metrics(ref_knn, query_knn)
```

Arguments

`ref_knn`	The reference `SingleCellNearestNeighbour`.
`query_knn`	The query `SingleCellNearestNeighbour`.

Value

A list with:

- matches - The intersecting indices between the reference and query kNN for each sample. In an ideal match up should be equal to k.
- distance_ratio - The distance ratio. Calculates `sum(dist_query) / sum(dist_ref)` per sample. Indicates how much worse the reference is.
- final_recall - The final recall across all samples.
- final_ratio - The final distance ratio across all samples.

Examples

```
# recall of an annoy index against the default one
set.seed(42L)
embd <- matrix(rnorm(500 * 10), nrow = 500)
rownames(embd) <- sprintf("cell_%03d", 1:500)
ref <- generate_sc_knn(embd, .verbose = FALSE)
query <- generate_sc_knn(
  embd,
  neighbours_params = params_sc_neighbours(
    knn = list(knn_method = "annoy")
  ),
  .verbose = FALSE
)
calc_knn_metrics(ref, query)$final_recall
```

`calc_manifold_metrics` *Calculate manifold metrics*

Description

This function will calculate the compactness and separation of your metacells on the manifold (defined by the diffusion map). You must have run `calc_diffusion_coordinates()` before calling this function. The idea is that compactness indicates how tight the metacell spans the manifold, whereas separation indicates how well the different metacells span the manifold.

Usage

```
calc_manifold_metrics(object)
```

Arguments

`object`	`MetaCells` class for which to calculate the different metrics.

Value

The class with the compactness and separation scores added.

References

Persad, et al. Nat Biotechnol, 2023

Examples

```
# compactness and separation on top of the diffusion coordinates
sc <- demo_single_cells()
mc <- generate_bt_meta_cells_sc(
  sc,
  sc_meta_cell_params = params_sc_bt_metacells(target_no_metacells = 50L),
  .verbose = FALSE
)
mc <- calc_diffusion_coordinates(
  mc,
  knn_data = get_knn_obj(sc),
  .verbose = FALSE
)
mc <- calc_manifold_metrics(mc)
head(mc[[c("compactness", "separation")]])

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`calc_meta_cell_compactness` *Meta cell compactness in a diffusion map*

Description

Scores a meta cell assignment for compactness, i.e. the mean variance across the diffusion components over the cells of each meta cell. Lower is better. Unlike `calc_manifold_metrics()`, this works on any assignment, so meta cells generated outside bixverse (or on a different object) can be scored against an existing diffusion map.

Usage

```
calc_meta_cell_compactness(dcs, assignment)
```

Arguments

`dcs`	Numeric matrix. The diffusion map coordinates, cells x components, for example `S7::prop(object, "other_data")$dcs` after `calc_diffusion_coordinates()`. Its rows follow the `used_cells` of the kNN object the diffusion map was built from.
`assignment`	Atomic vector of length `nrow(dcs)` with the meta cell of each row of `dcs`. `NA` marks cells that belong to no meta cell. Overlapping meta cells (hdWGCNA-style) cannot be expressed this way; use `calc_manifold_metrics()` for those.

Value

Named numeric vector with one compactness value per meta cell, named and ordered like `split()` orders `assignment`.

References

Persad, et al., Nat Biotechnol, 2023

Examples

```
set.seed(1L)
dcs <- matrix(rnorm(200L), ncol = 2L)
assignment <- rep(sprintf("mc_%i", 1:10), each = 10L)
calc_meta_cell_compactness(dcs, assignment)
```

`calc_meta_cell_purity` *Calculate meta cell purity*

Description

A potential metric to see how well the meta cells are aggregated is their cell type purity. This helper function helps to plot the meta-cell purity based on annotated cell types. These can be also just unsupervised memberships to graph-based clustering, etc.

Usage

```
calc_meta_cell_purity(
  object,
  original_cell_type,
  add_additional_info = c("none", "top_label", "top_two_labels"),
  add_entropy = FALSE
)
```

Arguments

`object`	`MetaCells` class.
`original_cell_type`	Character vector. The original cell type annotations of the object the meta cells came from. Either in the row order of its full (unfiltered) obs table, i.e. `get_sc_obs(x)$<column>`, or of the QC-passing cells only, i.e. `get_sc_obs(x, filtered = TRUE)$<column>`. Which one you passed is inferred from the length, so a vector matching neither is an error rather than a silently wrong purity.
`add_additional_info`	String. Which label information to add on top of the purity. One of `c("none", "top_label", "top_two_labels")`. Defaults to `"none"`, i.e. the purity only.
`add_entropy`	Boolean. Shall the normalised Shannon entropy of the label distribution be added as a diversity measure. Defaults to `FALSE`.

Details

Ties for the top (or second) label resolve to whichever label sorts first, as the labels are factorised internally. The entropy is the Shannon entropy of the label distribution within a meta cell, divided by `log(<number of distinct labels in original_cell_type>)`, so it sits in `[0, 1]` and stays comparable between meta cells. It is `0` if there is only a single label in the data.

Value

The `MetaCells` with added columns to the observation table:

- mc_purity - Fraction of the meta cell's cells that carry the most abundant label. Always added.
- mc_top_label - Name of the most abundant label. Added for `add_additional_info %in% c("top_label", "top_two_labels")`.
- mc_second_label - Name of the second most abundant label, `NA` for a pure meta cell. Added for `add_additional_info = "top_two_labels"`.
- mc_second_frac - Fraction of the meta cell's cells carrying that second label, `0` for a pure meta cell. Added for `add_additional_info = "top_two_labels"`.
- mc_entropy - Normalised Shannon entropy of the label distribution, see details. Added for `add_entropy = TRUE`.

Examples

```
# purity of the meta cells against the cell types they were built from
sc <- demo_single_cells()
mc <- generate_bt_meta_cells_sc(
  sc,
  sc_meta_cell_params = params_sc_bt_metacells(target_no_metacells = 50L),
  .verbose = FALSE
)
cell_types <- sc[["cell_grp"]]$cell_grp
mc <- calc_meta_cell_purity(mc, original_cell_type = cell_types)
head(mc[[c("meta_cell_id", "mc_purity")]])

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

calc_meta_cell_separation

Meta cell separation in a diffusion map

Description

Scores a meta cell assignment for separation, i.e. the Euclidean distance from each meta cell centroid in diffusion space to the nearest other meta cell centroid. Higher is better. Unlike `calc_manifold_metrics()`, this works on any assignment, so meta cells generated outside bixverse (or on a different object) can be scored against an existing diffusion map.

Usage

```
calc_meta_cell_separation(dcs, assignment)
```

Arguments

`dcs`	Numeric matrix. The diffusion map coordinates, cells x components, for example `S7::prop(object, "other_data")$dcs` after `calc_diffusion_coordinates()`. Its rows follow the `used_cells` of the kNN object the diffusion map was built from.
`assignment`	Atomic vector of length `nrow(dcs)` with the meta cell of each row of `dcs`. `NA` marks cells that belong to no meta cell. Overlapping meta cells (hdWGCNA-style) cannot be expressed this way; use `calc_manifold_metrics()` for those.

Value

Named numeric vector with one separation value per meta cell, named and ordered like `split()` orders `assignment`. A single meta cell yields `Inf`.

References

Persad, et al., Nat Biotechnol, 2023

Examples

```
set.seed(1L)
dcs <- matrix(rnorm(200L), ncol = 2L)
assignment <- rep(sprintf("mc_%i", 1:10), each = 10L)
calc_meta_cell_separation(dcs, assignment)
```

`calc_mitch` *Calculate a mitch gene set enrichments on contrast*

Description

Rust-based version of the mitch multi-contrast enrichment, see Kaspi and Ziemann. Takes in a matrix representing the contrasts you wish to test against.

Usage

```
calc_mitch(contrast_mat, gene_set_list, min_size = 5L)
```

Arguments

`contrast_mat`	Numerical matrix. The rows represent the gene statistic per contrast and each column represents the contrast.
`gene_set_list`	Named list. Contains the pathways you wish to test against.
`min_size`	Integer. Minimum size of the gene set to be included.

Value

A data.table with the Mitch enrichment results.

References

Kaspi and Ziemann, Bmc Genomics, 2020

Examples

```
# multi-contrast enrichment over two contrasts
set.seed(10L)
contrast_mat <- matrix(
  rnorm(300 * 2),
  ncol = 2,
  dimnames = list(sprintf("gene_%03i", 1:300), c("contrast_a", "contrast_b"))
)
contrast_mat[1:30, ] <- contrast_mat[1:30, ] + 1.5
gene_sets <- list(
  hit_set = sprintf("gene_%03i", 1:30),
  bg_set = sprintf("gene_%03i", 100:150)
)
res <- calc_mitch(contrast_mat, gene_sets)
res[, c("pathway_names", "manova_pval", "s_dist")]
```

---

calc_sc_type_scores *Calculate ScType scores per cell*

Description

Implements the approach from

Usage

```
calc_sc_type_scores(
  object,
  cell_marker_list,
  sensitivity = TRUE,
  weight_floor = NULL,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsMultiModal`
`cell_marker_list`	A list, see `prepare_cell_markers()`.
`sensitivity`	Boolean. Shall shared marker genes be downweighted (like in the original reference). Defaults to `TRUE`.
`weight_floor`	Optional numeric. A value between 0 to 1 and sets the floor for the weights if `sensitivity = TRUE`. If not provided, defaults to `0.1`.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

An `ScTypeResults` results class

Examples

```
# ScType scores against the planted marker genes
sc <- demo_single_cells()
markers <- data.table::data.table(
  cell_type = rep(sprintf("cell_type_%i", 1:3), each = 10),
  gene_id = sprintf("gene_%02d", 1:30)
)
cell_markers <- prepare_cell_markers(obj = sc, marker_df = markers)
res <- calc_sc_type_scores(
  sc,
  cell_marker_list = cell_markers,
  .verbose = FALSE
)
dim(get_scores(res))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`calc_singscore` *Bixverse implementation of singscore (single gene set)*

Description

Scores all samples against one up-regulated gene set with an optional paired down-regulated set. When `n_permutations > 0`, additionally runs a permutation test and includes empirical p-values.

Usage

```
calc_singscore(
  ranks,
  up_set,
  down_set = NULL,
  center_score = TRUE,
  known_direction = TRUE,
  n_permutations = 0L,
  seed = 42L
)
```

Arguments

`ranks`	Numerical matrix. Output of `calc_singscore_rank()`.
`up_set`	Character vector. Gene names of the up-regulated set.
`down_set`	Character vector or NULL. Optional paired down-regulated set.

`center_score` Boolean. Centre scores around 0. Ignored when `known_direction = FALSE`.

`known_direction` Boolean. Whether the up-set direction is known. Becomes irrelevant when `down_set` is also provided.

`n_permutations` Integer. Number of permutations. `0` disables the permutation test.

`seed` Integer. RNG seed for the permutation test.

Value

A data.table with one row per sample. Columns: `total_score`, `total_dispersion`, optionally `up_score`, `up_dispersion`, `down_score`, `down_dispersion`, and (when permutations are run) `pval`. With permutations, the null distribution is attached as `attr(., "null_distribution")`.

References

Foroutan et al., BMC Bioinformatics, 2018.

Examples

```
# score every sample against one up-regulated gene set
set.seed(123L)
exp_mat <- matrix(
  rnorm(200 * 10),
  nrow = 200,
  dimnames = list(sprintf("gene_%03i", 1:200), sprintf("sample_%i", 1:10))
)
ranks <- calc_singscore_rank(exp_mat)
head(calc_singscore(ranks, up_set = sprintf("gene_%03i", 1:20)))
```

`calc_singscore_multi` *Bixverse implementation of singscore (multiple gene sets)*

Description

Scores all samples against many up-regulated gene sets with optional paired down-regulated sets. Down sets are paired with up sets by name; unmatched names are dropped.

Usage

```
calc_singscore_multi(
  ranks,
  up_pathways,
  down_pathways = NULL,
  center_score = TRUE,
  known_direction = TRUE,
```

```
  min_size = 1L,
  max_size = 500L
)
```

Arguments

ranks	Numerical matrix. Output of calc_singscore_rank().
up_pathways	Named list of character vectors. Up-regulated gene sets.
down_pathways	Named list or NULL. Paired down-regulated gene sets.
center_score	Boolean. Centre scores around 0. Ignored when known_direction = FALSE.
known_direction	Boolean. Whether the up-set direction is known. Becomes irrelevant when down_set is also provided.
min_size	Integer. Minimum gene-set size after dropping missing genes.
max_size	Integer. Maximum gene-set size.

Value

A named list with two matrices, scores and dispersions, each of shape gene_sets × samples.

References

Foroutan et al., BMC Bioinformatics, 2018.

Examples

```
# score every sample against several up-regulated gene sets
set.seed(123L)
exp_mat <- matrix(
  rnorm(200 * 10),
  nrow = 200,
  dimnames = list(sprintf("gene_%03i", 1:200), sprintf("sample_%i", 1:10))
)
ranks <- calc_singscore_rank(exp_mat)
pathways <- list(
  set_a = sprintf("gene_%03i", 1:20),
  set_b = sprintf("gene_%03i", 50:80)
)
res <- calc_singscore_multi(ranks, up_pathways = pathways)
dim(res$scores)
```

`calc_singscore_rank` *Rank an expression matrix for singscore*

Description

Column-ranks an expression matrix. If `stable_genes` is provided, uses the stable-genes ranking method (unit-normalised ranks against a small set of stable genes); otherwise standard column ranks. Sets `attr(., "stable")` so downstream singscore functions know which bounds formula to use.

Usage

```
calc_singscore_rank(exp, stable_genes = NULL)
```

Arguments

`exp`	Numerical matrix. Rows = genes, columns = samples.
`stable_genes`	Character vector or NULL. Gene names of stable genes. Defaults to `NULL` (standard ranking).

Value

A rank matrix with the same shape as `exp` and `attr(., "stable")` set to `TRUE` or `FALSE`.

References

1.) Foroutan et al., BMC Bioinformatics, 2018.; 2.) Bhuva, et al., Nucleic Acids Res., 2020

Examples

```
# column ranks of an expression matrix, ready for singscore
set.seed(123L)
exp_mat <- matrix(
  rnorm(200 * 10),
  nrow = 200,
  dimnames = list(sprintf("gene_%03i", 1:200), sprintf("sample_%i", 1:10))
)
ranks <- calc_singscore_rank(exp_mat)
dim(ranks)
```

`calc_ssgsea` *Bixverse implementation of ssGSEA*

Description

Implementation of the bixverse version of the single sample gene set enrichment analysis (ssGSEA), see Barbie et al.

Usage

```
calc_ssgsea(exp, pathways, ssgsea_params = params_ssgsea(), .verbose = FALSE)
```

Arguments

`exp`	Numerical matrix. Rows represents the features, columns the features/genes.
`pathways`	List. A named list with each element containing the genes for this pathway.
`ssgsea_params`	List. The GSVA parameters, see `params_ssgsea()` wrapper function. This function generates a list containing: • alpha - Float. The exponent defining the weight of the tail in the random walk performed by ssGSEA. • min_size - Integer. Minimum size for the gene sets. • max_size - Integer. Maximum size for the gene sets. • normalise - Boolean. Shall the scores be normalised.
`.verbose`	Boolean. Controls verbosity.

Value

A matrix of shape pathways (that passed the thresholds) x samples.

References

Barbie et al., Nature, 2009

Examples

```
# per-sample ssGSEA scores for two gene sets
set.seed(123L)
exp_mat <- matrix(
  rnorm(200 * 10),
  nrow = 200,
  dimnames = list(sprintf("gene_%03i", 1:200), sprintf("sample_%i", 1:10))
)
pathways <- list(
  set_a = sprintf("gene_%03i", 1:20),
  set_b = sprintf("gene_%03i", 50:80)
)
round(calc_ssgsea(exp_mat, pathways)[, 1:3], 3)
```

`calculate_all_dges` *Calculate all possible DGE variants (DEPRECATED!)*

Description

This is a deprecated method and will raise an error. Please use `calculate_dge_limma()` and `calculate_dge_hedges()`.

Usage

```
calculate_all_dges(
  object,
  contrast_column,
  contrast_list = NULL,
  filter_column = NULL,
  co_variates = NULL,
  small_sample_correction = NULL,
  ...,
  .verbose = TRUE
)
```

Arguments

`object`	The underlying class, see `BulkDge()`.
`contrast_column`	String. The contrast column in which the groupings are stored. Needs to be found in the meta_data within the properties.
`contrast_list`	Optional string vector. A vectors that contains the contrast formatted as `"contrast1-contrast2"`. Default `NULL` will create all possible contrast automatically.
`filter_column`	Optional String. If there is a column you wish to use as sub groupings, this can be provided here. An example could be different sampled tissues and you wish to run the DGE analyses within each tissue separately.
`co_variates`	Optional string vector. Any co-variates you wish to consider during the Limma Voom modelling.
`small_sample_correction`	Can be NULL (automatic determination if a small sample size correction should be applied) or a Boolean.
`...`	Additional parameters that used to go to limma's `eBayes()` or `voom()`.
`.verbose`	Controls verbosity of the function.

Value

Throws an error, as it is deprecated.

`calculate_batch_asw_sc` *Calculate batch average silhouette width*

Description

Computes the average silhouette width (ASW) on batch labels using pairwise Euclidean distances in the embedding space. For each cell, the function estimates the mean distance to cells of the same batch (a) and the mean distance to cells of the nearest other batch (b), then computes s = (b - a) / max(a, b).

Values near 0 indicate good batch mixing, values near 1 indicate batch separation, and negative values suggest overcorrection. This metric is best suited for embedding-based correction methods (e.g. Harmony, fastMNN). For graph-based methods like BBKNN, consider using `calculate_lisi_sc()` instead.

Usage

```
calculate_batch_asw_sc(
  object,
  batch_column,
  embd_to_use = "pca",
  max_cells = 5000L,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`batch_column`	String. The column with the batch information in the obs data of the class.
`embd_to_use`	String. Which embedding to compute the ASW on. One of `c("pca", "harmony", "mnn")`. Defaults to `"pca"`.
`max_cells`	Integer or `NULL`. If not `NULL`, subsample to this many cells for performance. The pairwise distance computation is O(n^2), so subsampling is recommended for large datasets. Defaults to `5000L`.
`seed`	Integer. Seed for subsampling reproducibility.
`.verbose`	Boolean. Controls verbosity of the function.

Value

A `BatchSilhouetteScores` object with the following elements

- per_cell - Per-cell silhouette scores in `[-1, 1]`.
- mean_asw - Mean silhouette width across all cells.

- median_asw - Median silhouette width across all cells.
- n_batches - Number of batches in the data.
- embedding_used - Which embedding the ASW was computed on.

Examples

```
# batch silhouette width on the PCA embedding
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 600L, n_genes = 50L, n_batches = 3L
  )
)
calculate_batch_asw_sc(
  sc,
  batch_column = "batch_index",
  .verbose = FALSE
)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

calculate_cell_type_asw_sc

Calculate cell type average silhouette width

Description

Average silhouette width on cell type labels in the embedding, rescaled to `[0, 1]` via `(s + 1) / 2` as in scIB. Higher values mean cell types stay separated after correction. Counterpart to `calculate_batch_asw_sc()` on the biology side.

Usage

```
calculate_cell_type_asw_sc(
  object,
  cell_type_column,
  embd_to_use = "pca",
  max_cells = 5000L,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`cell_type_column`	String. The column with the cell type labels in the obs data of the class.
`embd_to_use`	String. Which embedding to compute the ASW on. Defaults to `"pca"`.

max_cells	Integer or NULL. If not NULL, subsample to this many cells for performance. Defaults to 5000L.
seed	Integer. Seed for subsampling reproducibility.
.verbose	Boolean. Controls verbosity of the function.

Value

A CellTypeAswScores object with the following elements

- per_cell - Per-cell rescaled silhouette scores in [0, 1].
- mean_asw - Mean rescaled silhouette width.
- median_asw - Median rescaled silhouette width.
- n_cell_types - Number of cell types.
- embedding_used - Which embedding the ASW was computed on.

References

Luecken, et al., Nat. Methods, 2022

Examples

```
# cell type silhouette width on the PCA embedding
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 600L, n_genes = 50L, n_batches = 3L
  )
)
calculate_cell_type_asw_sc(
  sc,
  cell_type_column = "cell_grp",
  .verbose = FALSE
)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

calculate_critical_value

Calculates the critical value

Description

This function calculates the critical value for a given ontology similarity matrix.

Usage

```
calculate_critical_value(x, alpha, permutations = 100000L, seed = 10101L)
```

Arguments

x	Numerical matrix or OntologySim class, see OntologySim(). This function tends to be slower on matrices compared to the OntologySim class.
alpha	Float. The alpha value. For example, 0.001 would mean that the critical value is smaller than 0.1 percentile of the random permutations.
permutations	Number of random permutations.
seed	Integer. For reproducibility purposes

Value

The critical value.

Examples

```
# critical value of a Wang similarity matrix at alpha 0.1
onto <- data.table::data.table(
  parent = c("a", "b", "b", "b", "c"),
  child = c("b", "c", "d", "e", "f"),
  type = c("part_of", "part_of", "part_of", "is_a", "is_a")
)
weights <- c(part_of = 0.8, is_a = 0.6)
sim_mat <- calculate_wang_sim_mat(onto, weights = weights)
calculate_critical_value(sim_mat, alpha = 0.1)
```

---

calculate_dge_hedges *Calculates the Hedge's G effect size*

Description

This function will calculate the Hedge's G effect size on the normalised counts. Should batch-corrected counts be found, these will be used. At a minimum you will need to provide contrast_column that can be found in the meta-data. If you do not provide a vector of contrasts that you wish to test for, every permutation of groups represented in that column will be tested against each other.

Usage

```
calculate_dge_hedges(
  object,
  contrast_column,
  contrast_list = NULL,
  filter_column = NULL,
  .verbose = TRUE
)
```

Arguments

`object`	The underlying class, see `BulkDge()`.
`contrast_column`	String. The contrast column in which the groupings are stored. Needs to be found in the meta_data within the properties.
`contrast_list`	Optional string vector. A vectors that contains the contrast formatted as `"contrast1-contrast2"`. Default `NULL` will create all possible contrast automatically.
`filter_column`	Optional String. If there is a column you wish to use as sub groupings, this can be provided here. An example could be different sampled tissues and you wish to run the DGE analyses within each tissue separately in the data.
`.verbose`	Controls verbosity of the function.

Value

Returns the class with additional data added to the outputs.

Examples

```
# effect sizes over every contrast in the case_control column
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- qc_bulk_dge(object, group_col = "case_control", .verbose = FALSE)
object <- normalise_bulk_dge(
  object,
  group_col = "case_control",
  .verbose = FALSE
)
object <- calculate_dge_hedges(
  object,
  contrast_column = "case_control",
  .verbose = FALSE
)
head(get_dge_effect_sizes(object))
```

`calculate_dge_limma` *Calculates the Limma Voom DGE*

Description

This function will apply the Limma Voom DGE workflow. At a minimum you will need to provide `contrast_column` that can be found in the meta-data. If you do not provide a vector of contrasts that you wish to test for, every permutation of groups represented in that column will be tested against each other.

Usage

```
calculate_dge_limma(
  object,
  contrast_column,
  contrast_list = NULL,
  filter_column = NULL,
  co_variates = NULL,
  limma_params = params_limma_voom(),
  .verbose = TRUE
)
```

Arguments

`object`	The underlying class, see `BulkDge()`.
`contrast_column`	String. The contrast column in which the groupings are stored. Needs to be found in the meta_data within the properties.
`contrast_list`	Optional string vector. A vectors that contains the contrast formatted as `"contrast1-contrast2"`. Default `NULL` will create all possible contrast automatically.
`filter_column`	Optional String. If there is a column you wish to use as sub groupings, this can be provided here. An example could be different sampled tissues and you wish to run the DGE analyses within each tissue separately in the data.
`co_variates`	Optional string vector. Any co-variates you wish to consider during the Limma Voom modelling.
`limma_params`	List. The limma parameters, see `params_limma_voom()`.
`.verbose`	Controls verbosity of the function.

Value

Returns the class with additional data added to the outputs.

Examples

```
# limma voom over every contrast in the case_control column
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- qc_bulk_dge(object, group_col = "case_control", .verbose = FALSE)
object <- normalise_bulk_dge(
  object,
  group_col = "case_control",
  .verbose = FALSE
)
```

```
object <- calculate_dge_limma(
  object,
  contrast_column = "case_control",
  .verbose = FALSE
)
head(get_dge_limma_voom(object))
```

`calculate_diffusion_auc` *Calculate the AUROC for a diffusion score*

Description

This functions can take a given `NetworkDiffusions` object and calculates an AUC and generates a Z-score based on random permutation of `random_aucs` for test for statistical significance if desired.

Usage

```
calculate_diffusion_auc(
  object,
  hit_nodes,
  auc_iters = 10000L,
  random_aucs = 1000L,
  permutation_test = FALSE,
  seed = 42L
)
```

Arguments

`object`	`NetworkDiffusions` object. The underlying class `NetworkDiffusions()`.
`hit_nodes`	String vector. Which nodes in the graph are considered a 'hit'.
`auc_iters`	Integer. How many iterations to run to approximate the AUROC.
`random_aucs`	Integer. How many random AUROCs to calculate to estimate the Z-score. Only of relevance if permutation test is set to `TRUE`.
`permutation_test`	Boolean. Shall a permutation based Z-score be calculated.
`seed`	Integer. Random seed.

Value

List with AUC and Z-score as the two named elements if permutations test set to TRUE; otherwise just the AUC.

Examples

```
# AUROC of the diffusion score against two known hit nodes
set.seed(42)
g <- igraph::sample_pa(15, directed = FALSE)
edges <- data.table::setDT(igraph::as_data_frame(g))[, `:=`(
  from = sprintf("node_%i", from),
  to = sprintf("node_%i", to)
)]
object <- NetworkDiffusions(edges, weighted = FALSE, directed = FALSE)
object <- diffuse_seed_nodes(object, c(node_1 = 1, node_3 = 1), "max")
calculate_diffusion_auc(object, hit_nodes = c("node_2", "node_4"))
```

---

calculate_effect_size

Calculate the Hedge's G effect between two matrices

Description

This function takes two matrices in and calculate on a per column basis the Hedge's G effect size and the standard error. These results can be subsequently used for meta-analyses or other approaches.

Usage

```
calculate_effect_size(
  mat_a,
  mat_b,
  small_sample_correction = NULL,
  .verbose = TRUE
)
```

Arguments

mat_a	Numerical matrix. Contains the values for group a. Assumes that rows = samples, and columns = features.
mat_b	Numerical matrix. Contains the values for group b.
small_sample_correction	Can be NULL (automatic determination if a small sample size correction should be applied) or Boolean.
.verbose	Boolean that controls verbosity of the function.

Value

x, robustly scaled.

Examples

```
# Hedge's G between two groups of ten samples
set.seed(42)
mat_a <- matrix(rnorm(100), nrow = 10, ncol = 10)
mat_b <- matrix(rnorm(100, mean = 1), nrow = 10, ncol = 10)
colnames(mat_a) <- colnames(mat_b) <- sprintf("gene_%i", 1:10)
res <- calculate_effect_size(mat_a, mat_b, .verbose = FALSE)
head(res$effect_sizes)
```

---

calculate_graph_connectivity_sc

Calculate the graph connectivity per cell type

Description

For each cell type, restricts the kNN graph to that cell type and takes the fraction of its cells in the largest connected component. A cell type split across batches after correction falls apart into several components and scores low. 1 means every cell type is one connected piece.

Usage

```
calculate_graph_connectivity_sc(object, cell_type_column, .verbose = TRUE)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`cell_type_column`	String. The column with the cell type labels in the obs data of the class.
`.verbose`	Boolean. Controls verbosity of the function.

Value

A `GraphConnectivityScores` object with the following elements

- per_cell_type - Named numeric. Connectivity per cell type.
- mean_connectivity - Mean connectivity across cell types.
- median_connectivity - Median connectivity across cell types.

References

Luecken, et al., Nat. Methods, 2022

Examples

```
# connectivity of each cell type in the kNN graph
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 600L, n_genes = 50L, n_batches = 3L
  )
)
calculate_graph_connectivity_sc(sc, cell_type_column = "cell_grp")

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

calculate_information_content

Calculate the information content for each ontology term

Description

this function will calculate the information content of each provided term based on a list of ancestors, which is a named list of terms with ancestor identifiers as their values. Can be calculated using `get_ontology_ancestry()`. The information content is calculated as `-log2(number descendant/total terms in the ontology)`. More information can be found here.

Usage

```
calculate_information_content(ancestor_list)
```

Arguments

`ancestor_list`	List. Named list of terms with ancestor identifiers as their values

Value

A named list of each term and their information content as values.

Examples

```
# information content from the descendant counts of a toy ontology
onto <- data.table::data.table(
  parent = c("a", "b", "b", "b", "c"),
  child = c("b", "c", "d", "e", "f")
)
ancestry <- get_ontology_ancestry(onto)
unlist(calculate_information_content(ancestry$descendants))
```

`calculate_integration_metrics_sc` *Calculate a summary of integration metrics*

Description

Runs the batch mixing and (if cell type labels are given) the biological conservation metrics in one go and returns one row per call, so results across correction methods can be `rbind`-ed into one table. Every column is on `[0, 1]` (PCR comparison can go negative) and higher is better, the scIB convention:

Batch mixing:

- kbet_accept - `1 -` kBET rejection rate.
- batch_asw - `mean(1 - |s|)` over the per-cell batch silhouettes.
- ilisi - Normalised iLISI.
- pcr_comparison - `(pre - post) / pre` of the batch PCR.

Biological conservation:

- clisi - Normalised cLISI.
- cell_type_asw - Rescaled cell type silhouette width.
- graph_connectivity - Mean graph connectivity over cell types.

The kNN metrics read the kNN graph currently stored in the object, so recompute the neighbours on the corrected embedding first. Embedding metrics are `NA` if `embd_to_use = NULL` (e.g. BBKNN, which only returns a graph).

Usage

```
calculate_integration_metrics_sc(
  object,
  batch_column,
  cell_type_column = NULL,
  embd_to_use = "pca",
  max_cells = 5000L,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`batch_column`	String. The column with the batch information in the obs data of the class.
`cell_type_column`	Optional string. The column with the cell type labels. If `NULL`, the conservation metrics are `NA`.

embd_to_use	Optional string. The embedding for ASW and PCR. Defaults to "pca".
max_cells	Integer or NULL. Subsampling for the silhouette widths. Defaults to 5000L.
seed	Integer. Seed for subsampling reproducibility.
.verbose	Boolean. Controls verbosity of the function.

Value

A one-row data.table with the columns embedding, kbet_accept, batch_asw, ilisi, pcr_comparison, clisi, cell_type_asw and graph_connectivity.

References

Luecken, et al., Nat. Methods, 2022

Examples

```
# all metrics on the uncorrected PCA
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 600L, n_genes = 50L, n_batches = 3L
  )
)
calculate_integration_metrics_sc(
  sc,
  batch_column = "batch_index",
  cell_type_column = "cell_grp",
  .verbose = FALSE
)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

calculate_kbet_sc *Calculate kBET scores*

Description

This function calculates the k-nearest neighbour batch-effect test (kBET). Briefly, the function leverages a Chi-Square statistic to calculate the differences in batch proportions observed in the neighbourhood of a given cell with the overall batch proportions. If the test is significant for that cell it indicates poor mixing for that cell specifically. Large number of positive tests indicate bad mixing overall. For more details, please see Büttner et al.

Usage

```
calculate_kbet_sc(object, batch_column, threshold = 0.05, .verbose = TRUE)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`batch_column`	String. The column with the batch information in the obs data of the class.
`threshold`	Numeric. Number between 0 and 1. Below this threshold, the test is considered significant. Defaults to `0.05`.
`.verbose`	Boolean. Controls verbosity of the function.

Value

A `KbetScores` object with the following elements

- kbet_score - Proportion of significant tests over all cells. 0 indicates perfect mixing, 1 indicates no mixing between batches.
- significant_tests - Logical vector indicating for which cells the test was below the threshold.
- p_values - The p-values from the Chi-Square test.
- chi_square_stats - Per-cell Chi-Square statistics.
- mean_chi_square - Mean Chi-Square statistic across all cells.
- median_chi_square - Median Chi-Square statistic across all cells.
- threshold - The significance threshold used.
- n_batches - Number of batches in the data.

References

Büttner, et al., Nat. Methods, 2019

Examples

```
# kBET rejection rate over three batches
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 600L, n_genes = 50L, n_batches = 3L
  )
)
calculate_kbet_sc(sc, batch_column = "batch_index", .verbose = FALSE)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`calculate_lisi_sc` *Calculate LISI scores (iLISI or cLISI)*

Description

Computes the Local Inverse Simpson's Index (LISI) on the kNN graph: the effective number of labels in each cell's neighbourhood. On batch labels this is iLISI, where higher means better mixing. On cell type labels it is cLISI, where lower means cell types stay apart. Unlike kBET, LISI does not compare against global proportions, so it also works on graph-based corrections like BBKNN.

The normalised score follows scIB and lands in `[0, 1]`, higher is better for both: iLISI as `(median - 1) / (n - 1)`, cLISI as `(n - median) / (n - 1)`.

Usage

```
calculate_lisi_sc(
  object,
  label_column,
  type = c("batch", "cell_type"),
  weighted = FALSE,
  perplexity = 30,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`label_column`	String. The column with the batch or cell type labels in the obs data of the class.
`type`	String. One of `c("batch", "cell_type")`. Decides which normalised score is reported. Defaults to `"batch"`.
`weighted`	Boolean. Weight the neighbours with a perplexity-calibrated Gaussian kernel on the kNN distances, as in Korsunsky et al. If `FALSE`, all neighbours count equally. Defaults to `FALSE`.
`perplexity`	Numeric. Perplexity for the weighted version. Defaults to `30`.
`.verbose`	Boolean. Controls verbosity of the function.

Value

A `LisiScores` object with the following elements

- per_cell - Per-cell LISI scores in `[1, n_labels]`.
- mean_lisi - Mean LISI across all cells.
- median_lisi - Median LISI across all cells.
- lisi_norm - The normalised score in `[0, 1]`, higher is better.
- n_labels - Number of distinct labels.
- type - `"batch"` (iLISI) or `"cell_type"` (cLISI).

References

Korsunsky, et al., Nat. Methods, 2019; Luecken, et al., Nat. Methods, 2022

Examples

```
# iLISI over the kNN graph
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 600L, n_genes = 50L, n_batches = 3L
  )
)
calculate_lisi_sc(
  sc,
  label_column = "batch_index",
  .verbose = FALSE
)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

calculate_pca_adt_sc *Calculate the PCA on top of the normalised ADT counts*

Description

This function will run PCA - via (randomised) SVD - on the normalised counts and add the PCA results to the ScCache for the ADT counts.

Usage

```
calculate_pca_adt_sc(
  object,
  no_pcs,
  features = NULL,
  randomised_svd = FALSE,
  seed = 42L
)
```

Arguments

`object`	`SingleCellsMultiModal` class with ADT counts added.
`no_pcs`	Integer. Number of PCs to calculate.
`features`	Optional string vector. If you want to subset to a specific set of ADT probes (for example to exclude isotypes).
`randomised_svd`	Boolean. Shall randomised SVD be used. Faster, but less precise.
`seed`	Integer. Controls reproducibility. Only relevant if `randomised_svd = TRUE`.

Value

The function will add the PCA factors, loadings and singular values for the ADT data to the object.

Examples

```
# PCA over the CLR-normalised protein counts
rna <- generate_single_cell_test_data()
adt <- generate_single_cell_test_data_adt()
dir <- tempfile("bixverse_mm")
dir.create(dir)
object <- load_r_data(
  SingleCellsMultiModal(dir_data = dir),
  counts = rna$counts,
  obs = rna$obs,
  var = rna$var,
  sc_qc_param = params_sc_min_quality(min_unique_genes = 5L),
  .verbose = FALSE
)
object <- add_adt_counts_sc(object, adt_counts = adt$counts, method = "clr")
object <- calculate_pca_adt_sc(object, no_pcs = 10L)
dim(get_pca_factors(object, modality = "adt"))

unlink(dir, recursive = TRUE, force = TRUE)
```

---

calculate_pca_bulk_dge

Calculate PCA on the expression.

Description

Calculates the principal component on top of the filtered count matrix and adds the information of the first 10 principal components to the outputs.

Usage

```
calculate_pca_bulk_dge(
  object,
  scale_genes = FALSE,
  pcs = 10L,
  no_hvg_genes = 2500L
)
```

Arguments

`object`	The underlying class, see `BulkDge()`.
`scale_genes`	Boolean. Shall the log(cpm) counts be scaled prior the PCA calculation. Defaults to `FALSE`.

pcs	Integer. Number of PCs to return and add to the outputs slot.
no_hvg_genes	Integer. Number of highly variable genes to include. Defaults to 2500.

Value

Returns the class with additional data added to the outputs.

Examples

```
# PCA over the most variable genes of the normalised counts
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- qc_bulk_dge(object, group_col = "case_control", .verbose = FALSE)
object <- normalise_bulk_dge(
  object,
  group_col = "case_control",
  .verbose = FALSE
)
object <- calculate_pca_bulk_dge(object, no_hvg_genes = 500L)
head(get_outputs(object)$pca)
```

---

calculate_pca_sc *Run PCA for single cell*

Description

This function will run PCA on the detected highly variable genes. The solver is set via `svd_solver` in `params_sc_pca()`; the default builds the gene x gene cross-product from the sparse data, which is exact and the fastest option for a few thousand HVGs.

Usage

```
calculate_pca_sc(
  object,
  no_pcs,
  pca_params = params_sc_pca(),
  sparse_svd = NULL,
  hvg = NULL,
  seed = 42L,
  residuals = FALSE,
  .verbose = TRUE
)
```

Arguments

object	SingleCells, MetaCells (or potentially other) class.
no_pcs	Integer. Number of PCs to calculate.
pca_params	Named list. Controls the parameters to be used for the PCA calculation which is single cell-specific, see params_sc_pca()
sparse_svd	Optional boolean. Solve on the sparse data with implicit centring and scaling instead of materialising the dense scaled matrix. NULL picks the sparse path, which is faster and lighter on memory for every solver, or the dense one for residuals = TRUE. FALSE uses the dense path, but only below 500,000 cells. Not used for MetaCells.
hvg	Optional integer. If you want to provide your own HVG genes. Otherwise, the function will default to what is found in get_hvg(). Please provide 1-indexed genes here! If you provide these, the internal HVG will be overwritten.
seed	Integer. Controls reproducibility. Only relevant for svd_solver = "randomised" and the Lanczos start vector.
residuals	Boolean. Run the PCA on the Pearson residuals of a model fitted with fit_residuals_sc() instead of the stored normalised layer. Needs params_sc_pca(normalise_ = FALSE, clr = FALSE), since the residuals already carry the signal as variance, and does not support sparse_svd: a residual column is dense even where the counts are not. On this dense path the default svd_solver = "covariance" pays for a full gene x gene cross-product and is slower than svd_solver = "randomised", so pick the latter in params_sc_pca() when speed matters. Not supported for MetaCells.
.verbose	Boolean or integer. Controls verbosity and returns run times. FALSE -> quiet, TRUE or 1L -> normal verbosity, 2L -> detailed verbosity.

Value

The function will add the PCA factors, loadings and singular values to the object cache in memory.

Examples

```
# PCA on the highly variable genes
sc <- demo_single_cells(prepped = FALSE)
sc <- find_hvg_sc(sc, hvg_no = 30L, .verbose = FALSE)
sc <- calculate_pca_sc(sc, no_pcs = 10L, .verbose = FALSE)
dim(get_pca_factors(sc))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`calculate_pcr_sc` *Calculate the principal component regression on batch*

Description

Regresses each embedding dimension on the batch labels and weights the per-dimension R-squared by the variance of that dimension. On its own the number says how much of the embedding variance batch explains. For a corrected embedding, the function also runs it on the uncorrected PCA and reports the scIB comparison `(pre - post) / pre`: 1 means the batch variance is gone, 0 means nothing changed, negative means it got worse.

Usage

```
calculate_pcr_sc(object, batch_column, embd_to_use = "pca", .verbose = TRUE)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`batch_column`	String. The column with the batch information in the obs data of the class.
`embd_to_use`	String. Which embedding to compute the PCR on. Defaults to `"pca"`.
`.verbose`	Boolean. Controls verbosity of the function.

Value

A `PcrScores` object with the following elements

- pcr - Variance-weighted R-squared of batch on `embd_to_use`.
- pcr_pca - The same on the uncorrected PCA.
- pcr_comparison - `(pcr_pca - pcr) / pcr_pca`. `NA` if `embd_to_use = "pca"`.
- var_explained - Variance per embedding dimension.
- r_squared - Batch R-squared per embedding dimension.
- embedding_used - Which embedding the PCR was computed on.

References

Büttner, et al., Nat. Methods, 2019; Luecken, et al., Nat. Methods, 2022

Examples

```
# share of PCA variance explained by batch
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 600L, n_genes = 50L, n_batches = 3L
  )
)
```

```
calculate_pcr_sc(sc, batch_column = "batch_index")

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`calculate_rpkm` *RPKM calculation*

Description

RPKM calculation

Usage

```
calculate_rpkm(counts, gene_lengths)
```

Arguments

`counts`	Numeric matrix. Count matrix (gene x sample)
`gene_lengths`	Named vector. Named vector with gene lengths.

Value

RPKM-normalised matrix.

Examples

```
# RPKM over synthetic counts with flat 2kb gene lengths
syn <- synthetic_bulk_cor_matrix()
gene_lengths <- stats::setNames(
  rep(2000, nrow(syn$counts)),
  rownames(syn$counts)
)
rpkm <- calculate_rpkm(syn$counts, gene_lengths)
rpkm[1:3, 1:3]
```

`calculate_semantic_sim`
Calculate the Resnik or Lin semantic similarity

Description

This function calculates the semantic similarities based on Resnik or Lin similarity for a given ontology. Has also the option to calculate a combined version of the two.

Usage

```
calculate_semantic_sim(
  terms,
  similarity_type,
  ancestor_list,
  ic_list,
  add_self = FALSE
)
```

Arguments

terms	Vector of strings. The terms in the ontology you wish to screen.
similarity_type	String. One of `c("resnik", "lin", "combined")`.
ancestor_list	List. Names being the term and the elements in the list the names of the ancestors, see `get_ontology_ancestry()`.
ic_list	List. The names being the term and the elements the information content of this given term. Needs to be a single float! See `calculate_information_content()`.
add_self	Boolean. Shall self-similarities be added. Defaults to `FALSE`.

Value

A data.table with the calculated similarities.

Examples

```
# Lin similarity for a subset of terms only
onto <- data.table::data.table(
  parent = c("a", "b", "b", "b", "c"),
  child = c("b", "c", "d", "e", "f")
)
ancestry <- get_ontology_ancestry(onto)
ic <- calculate_information_content(ancestry$descendants)
calculate_semantic_sim(
  terms = c("c", "d", "f"),
  similarity_type = "lin",
  ancestor_list = ancestry$ancestors,
  ic_list = ic
)
```

---

calculate_semantic_sim_mat

Calculate the Resnik or Lin semantic similarity matrix

Description

This function calculates the semantic similarities based on Resnik or Lin similarity for a given ontology. Has also the option to calculate a combined version of the two. Returns the full matrix of the data.

Usage

```
calculate_semantic_sim_mat(similarity_type, ancestor_list, ic_list)
```

Arguments

similarity_type	String. One of `c("resnik", "lin", "combined")`. The Resnik similarity is automatically rescaled between 0 and 1 (dividing the values by maximum information content observed).
ancestor_list	List. Names being the term and the elements in the list the names of the ancestors, see `get_ontology_ancestry()`.
ic_list	List. The names being the term and the elements the information content of this given term. Needs to be a single float! See `calculate_information_content()`.

Value

The symmetric similarity matrix for the specified data from the ontology.

Examples

```
# Resnik similarity matrix over a toy ontology
onto <- data.table::data.table(
  parent = c("a", "b", "b", "b", "c"),
  child = c("b", "c", "d", "e", "f")
)
ancestry <- get_ontology_ancestry(onto)
ic <- calculate_information_content(ancestry$descendants)
round(calculate_semantic_sim_mat("resnik", ancestry$ancestors, ic), 3)
```

---

calculate_semantic_sim_onto

Calculate the Resnik or Lin semantic similarity for an ontology.

Description

This function calculates the specified semantic similarities for the whole ontology and adds it to the class.

Usage

```
calculate_semantic_sim_onto(
  object,
  sim_type = c("resnik", "lin", "combined"),
  .verbose = TRUE
)
```

Arguments

`object`	`OntologySim` class. See `OntologySim()`.
`sim_type`	String. One of `c("resnik", "lin", "combined")`.
`.verbose`	Boolean. Controls the verbosity of the function.

Value

The class with added semantic similarities to the properties.

Examples

```
# Resnik similarities for the whole ontology stored in the class
onto <- data.table::data.table(
  parent = c("a", "b", "b", "b", "c"),
  child = c("b", "c", "d", "e", "f"),
  type = c("part_of", "part_of", "part_of", "is_a", "is_a")
)
onto_obj <- pre_process_sim_onto(
  OntologySim(onto, .verbose = FALSE),
  .verbose = FALSE
)
calculate_semantic_sim_onto(onto_obj, sim_type = "resnik", .verbose = FALSE)
```

`calculate_sparsity_stats` *Helper function to calculate the induced sparsity*

Description

Helper function to calculate the induced sparsity

Usage

```
calculate_sparsity_stats(object, no_exp_bins = 10L)
```

Arguments

`object`	`synthetic_bulk_data` object. You need to have run `simulate_dropouts()` for this function to work.
`no_exp_bins`	Integer. Number of expression bins to check. Defaults to `10L`.

Value

A list with various statistics about the sparsity

- original_sparsity - Original proportion of zeroes in the counts.
- final_sparsity - Sparsity after applying `simulate_dropouts()`.

- added_sparsity - Added sparsity.
- gene_sparsity_mean - Mean sparsity for the genes.
- gene_sparsity_sd - SD sparsity form the genes.
- sample_sparsity_mean - Mean sparsity for the genes.
- sample_sparsity_sd - SD sparsity for the genes.
- dropout_by_expression - Dropout per expression bin level.

Examples

```
# how much sparsity the dropout simulation actually added
syn <- synthetic_bulk_cor_matrix()
syn <- simulate_dropouts(syn, params_bulk_sparsity())
stats <- calculate_sparsity_stats(syn)
stats$added_sparsity
stats$dropout_by_expression
```

`calculate_tom` *Calculate the TOM from a correlation matrix*

Description

Calculate the TOM from a correlation matrix

Usage

```
calculate_tom(cor_mat, signed, version = c("v1", "v2"))
```

Arguments

`cor_mat`	Numerical matrix. The symmetric correlation matrix.
`signed`	Boolean. Do you want to calculate the signed version. If set to `FALSE`, the absolute correlation coefficients will be used.
`version`	String. One of `c("v1", "v2")`. Defaults to `"v1"`.

Details

Calculates the topological overlap matrix from a correlation matrix. The TOM is defined as:

Unsigned, v1:

$$TOM_{ij} = \frac{a_{ij} + \sum_{k \neq i,j} a_{ik} a_{kj}}{\min(k_i, k_j) + 1 - a_{ij}}$$

Signed, v1:

$$TOM_{ij} = \frac{a_{ij} + \sum_{k \neq i,j} a_{ik} a_{kj}}{\min(k_i, k_j) + 1 - |a_{ij}|}$$

Unsigned, v2:

$$TOM_{ij} = 0.5 \left(a_{ij} + \frac{\sum_{k \neq i,j} a_{ik} a_{kj}}{\min(k_i, k_j) + a_{ij}} \right)$$

Signed, v2:

$$TOM_{ij} = 0.5 \left(a_{ij} + \frac{\sum_{k \neq i,j} a_{ik} a_{kj}}{\min(k_i, k_j) + |a_{ij}|} \right)$$

where a_{ij} is the affinity between nodes i and j, and $k_i = \sum_j a_{ij}$ is the connectivity of node i. For signed networks, connectivity is calculated as $k_i = \sum_j |a_{ij}|$.

Version 2 uses a different normalisation approach that scales the shared neighbour contribution separately before combining it with the direct connection strength.

Value

A symmetric matrix of the same dimensions as `cor_mat` containing the topological overlap measures.

Examples

```
# unsigned TOM from a small correlation matrix
set.seed(42)
cor_mat <- cor(matrix(rnorm(200), nrow = 20, ncol = 10))
tom <- calculate_tom(cor_mat, signed = FALSE)
dim(tom)
```

`calculate_tom_from_exp` *Calculate the TOM from an expression matrix*

Description

Calculate the TOM from an expression matrix

Usage

```
calculate_tom_from_exp(x, signed, version, cor_method)
```

Arguments

`x`	Numerical matrix. The expression matrix. Assumes that columns are the genes, and rows the samples.
`signed`	Boolean. Do you want to calculate the signed version. If set to `FALSE`, the absolute correlation coefficients will be used.
`version`	String. One of `c("v1", "v2")`. Defaults to `"v1"`
`cor_method`	String. One of `c("pearson", spearman)`. Defaults to `"pearson"`.

Details

Calculates the topological overlap matrix from an expression matrix. It will first calculate the specified correlation matrix and then generate the TOM. The TOM is defined as:

Unsigned, v1:

$$TOM_{ij} = \frac{a_{ij} + \sum_{k \neq i,j} a_{ik} a_{kj}}{\min(k_i, k_j) + 1 - a_{ij}}$$

Signed, v1:

$$TOM_{ij} = \frac{a_{ij} + \sum_{k \neq i,j} a_{ik} a_{kj}}{\min(k_i, k_j) + 1 - |a_{ij}|}$$

Unsigned, v2:

$$TOM_{ij} = 0.5 \left(a_{ij} + \frac{\sum_{k \neq i,j} a_{ik} a_{kj}}{\min(k_i, k_j) + a_{ij}} \right)$$

Signed, v2:

$$TOM_{ij} = 0.5 \left(a_{ij} + \frac{\sum_{k \neq i,j} a_{ik} a_{kj}}{\min(k_i, k_j) + |a_{ij}|} \right)$$

where a_{ij} is the affinity between nodes i and j, and $k_i = \sum_j a_{ij}$ is the connectivity of node i. For signed networks, connectivity is calculated as $k_i = \sum_j |a_{ij}|$.

Version 2 uses a different normalisation approach that scales the shared neighbour contribution separately before combining it with the direct connection strength.

Value

The topological overlap matrix.

Examples

```
# TOM straight from a samples x genes expression matrix
set.seed(42)
x <- matrix(rnorm(200), nrow = 20, ncol = 10)
tom <- calculate_tom_from_exp(
  x,
  signed = FALSE,
  version = "v1",
  cor_method = "pearson"
)
dim(tom)
```

---

calculate_tpm *TPM calculation*

Description

TPM calculation

Usage

```
calculate_tpm(counts, gene_lengths)
```

Arguments

`counts`	Numeric matrix. Count matrix (gene x sample)
`gene_lengths`	Named vector. Named vector with gene lengths.

Value

TPM-normalised matrix.

Examples

```
# TPM over synthetic counts with flat 2kb gene lengths
syn <- synthetic_bulk_cor_matrix()
gene_lengths <- stats::setNames(
  rep(2000, nrow(syn$counts)),
  rownames(syn$counts)
)
tpm <- calculate_tpm(syn$counts, gene_lengths)
colSums(tpm)[1:3]
```

`calculate_wang_sim` *Calculate the Wang similarities between terms*

Description

This function calculates the Wang similarity for a set of temrs, based on the DAG for a given ontology.

Usage

```
calculate_wang_sim(terms, parent_child_dt, weights, add_self = FALSE)
```

Arguments

`terms`	String vector. The terms for which to calculate the Wang similarity.
`parent_child_dt`	data.table. The data.table with column parent and child. You also need to have a type column for the Wang similarity to provide the weights for the relationships.
`weights`	Named numeric. The relationship of type to weight for this specific edge. For example `c("part_of" = 0.8, "is_a" = 0.6)`.
`add_self`	Boolean. Shall self-similarities be added. Defaults to `FALSE`.

Value

A data.table with the calculated similarities.

Examples

```
# Wang similarity for a subset of terms only
onto <- data.table::data.table(
  parent = c("a", "b", "b", "b", "c"),
  child = c("b", "c", "d", "e", "f"),
  type = c("part_of", "part_of", "part_of", "is_a", "is_a")
)
weights <- c("part_of" = 0.8, "is_a" = 0.6)
calculate_wang_sim(c("c", "d", "f"), onto, weights = weights)
```

`calculate_wang_sim_mat`

Calculate the Wang similarity matrix

Description

This function calculates the Wang similarity, based on the DAG for a given ontology. This function will return the full similarity matrix.

Usage

```
calculate_wang_sim_mat(parent_child_dt, weights)
```

Arguments

`parent_child_dt`	data.table. The data.table with column parent and child. You also need to have a type column for the Wang similarity to provide the weights for the relationships.
`weights`	Named numeric. The relationship of type to weight for this specific edge. For example `c("part_of" = 0.8, "is_a" = 0.6)`.

Value

The symmetric Wang similarity matrix.

Examples

```
# Wang similarity matrix with relationship-specific weights
onto <- data.table::data.table(
  parent = c("a", "b", "b", "b", "c"),
  child = c("b", "c", "d", "e", "f"),
  type = c("part_of", "part_of", "part_of", "is_a", "is_a")
)
weights <- c("part_of" = 0.8, "is_a" = 0.6)
round(calculate_wang_sim_mat(onto, weights = weights), 3)
```

`calculate_wang_sim_onto`

Calculate the Wang similarity for an ontology.

Description

This function calculates the Wang similarity for the whole ontology and adds it to the class in a memory-efficient format for subsequent usage.

Usage

```
calculate_wang_sim_onto(object, weights, .verbose = TRUE)
```

Arguments

`object`	`OntologySim` class. See `OntologySim()`.
`weights`	Named numeric. The relationship of type to weight for this specific edge. For example `c("part_of" = 0.8, "is_a" = 0.6)`.
`.verbose`	Boolean. Controls the verbosity of the function.

Value

The class with added semantic similarities to the properties.

Examples

```
# Wang similarities for the whole ontology stored in the class
onto <- data.table::data.table(
  parent = c("a", "b", "b", "b", "c"),
  child = c("b", "c", "d", "e", "f"),
  type = c("part_of", "part_of", "part_of", "is_a", "is_a")
)
onto_obj <- OntologySim(onto, .verbose = FALSE)
calculate_wang_sim_onto(
  onto_obj,
  weights = c(part_of = 0.8, is_a = 0.6),
  .verbose = FALSE
)
```

`call_doublets_manual` *Manually readjust Scrublet doublet call thresholds*

Description

Updates doublet calls and associated summary statistics in a `ScrubletRes` object using a user-supplied threshold. Intended for use after visual inspection of the score histograms via `plot.ScrubletRes()`.

Usage

```
call_doublets_manual(
  scrublet_res,
  threshold,
  for_sample = NULL,
  .verbose = TRUE
)
```

Arguments

scrublet_res	A ScrubletRes object.
threshold	Numeric in [0, 1]. The new threshold to apply.
for_sample	Optional character. For grouped results, the name of the group to update. Defaults to the first group if NULL. Ignored for ungrouped results.
.verbose	Logical. If TRUE, prints updated rate summaries to the console.

Value

The ScrubletRes object with updated predicted_doublets, z_scores, threshold, detected_doublet_rate, detectable_doublet_fraction, and overall_doublet_rate.

Examples

```
# move the automatic threshold after eyeballing the histograms
sc <- demo_single_cells(prepped = FALSE)
res <- scrublet_sc(
  sc,
  scrublet_params = params_scrublet(
    pca = list(no_pcs = 10L),
    hvg = list(min_gene_var_pctl = 0.0),
    n_bins = 20L
  ),
  .verbose = FALSE
)
call_doublets_manual(res, threshold = 0.3, .verbose = FALSE)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

cell_cycle_genes *Cell cycle genes*

Description

(Human) cell cycle genes for scoring in single cell, please see module_scores_sc() and Tirosh et al, Science (2016). These genes are based on the updated version in Seurat.

Usage

```
cell_cycle_genes
```

Format

cell_cycle_genes:
A data.table

hgnc_symbol The (HGNC) gene symbol

ensembl_gene_id The corresponding ensembl identifiers.

set To which cell cycle phase the gene belongs

`cellphonedb_sc` *CellPhoneDB ligand-receptor analysis*

Description

Runs the CellPhoneDB v5 analysis on a `SingleCells` object. For every interaction and every ordered (sender, receiver) cluster pair, partner a is measured in the sender and partner b in the receiver. The interaction mean is the average of the two partner means, zero if either is not expressed. A complex takes the minimum over its subunits. The gate requires both partners to be expressed in more than `threshold` of their cluster's cells.

Three methods, as in CellPhoneDB:

- `"statistical"` - Permutes the cluster labels and reports `p = #(perm > real) / n_perm`, 1 where the mean is zero or the gate fails.
- `"degs"` - No permutations. The gate also needs partner a to be differentially expressed in the sender or partner b in the receiver, as given by `deg_table`.
- `"simple"` - Means and the expression gate only.

Reads the normalised layer of the L/R genes only, once. Interactions with any subunit missing from the object are dropped with a warning.

Usage

```
cellphonedb_sc(
  object,
  celltype_colname,
  lr_db,
  method = c("statistical", "degs", "simple"),
  deg_table = NULL,
  senders = NULL,
  receivers = NULL,
  gene_id_col = "gene_id",
  params = params_sc_cellphonedb(),
  .verbose = TRUE
)
```

Arguments

`object`	A `SingleCells` object.
`celltype_colname`	String. Name of the cluster column in `obs`. Cells with a missing label are ignored.
`lr_db`	data.table. The ligand-receptor database, see `get_cellphonedb_db()`. Needs `interaction_id`, `partner_a`, `partner_b` and the subunit list columns `genes_a` and `genes_b`.
`method`	String. One of `c("statistical", "degs", "simple")`.

`deg_table` Optional data.table with the columns `cluster_id` and `gene`, holding the differentially expressed genes per cluster, e.g. a filtered `find_all_markers_sc()` result. Required for `method = "degs"`.

`senders`, `receivers` Optional character vectors. Restrict the tested pairs to these sender and receiver clusters. `NULL` uses all clusters.

`gene_id_col` String. The var column holding the gene symbols used in `lr_db`. A duplicated symbol resolves to its first gene. Defaults to `"gene_id"`.

`params` List. See `params_sc_cellphonedb()`.

`.verbose` Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

A data.table with one row per interaction and cluster pair:

- interaction_id, partner_a, partner_b - From `lr_db`.
- sender, receiver - The cluster pair.
- mean - The interaction mean.
- pval - Permutation p-value, `NA` unless `method = "statistical"`.
- gate - The expression gate, or the DEG gate for `method = "degs"`.

References

Efremova et al., Nat Protoc, 2020; Troulé et al., Nat Protoc, 2025.

`cellsweep_sc` *Remove ambient and bulk contamination with CellSweep*

Description

Fits the CellSweep multinomial mixture model and writes a new `SingleCells` object holding the denoised counts. Every observed count is split three ways by EM: ambient contamination drawn from a per-emulsion profile, bulk contamination drawn from a global profile, and true expression drawn from the barcode's cell-type profile. Subtracting the first two gives the denoised matrix.

Usage

```
cellsweep_sc(
  target,
  input,
  celltype_column,
  sample_column,
  empty_params,
  cellsweep_params = params_sc_cellsweep(),
```

```
  sc_qc_param = params_sc_min_quality(),
  csc_mem_gb = NULL,
  streaming = deprecated(),
  batch_size = deprecated(),
  max_genes_in_memory = deprecated(),
  cell_batch_size = deprecated(),
  .verbose = TRUE
)
```

Arguments

`target`	`SingleCells`. A fresh object pointing at the directory the denoised counts should be written to.
`input`	`SingleCells`. The raw object, with the empty droplets still in it.
`celltype_column`	String. Obs column with the cell-type labels.
`sample_column`	String. Obs column identifying the emulsion. One independent fit per level.
`empty_params`	List. See `params_sc_empty_droplets()`. Required: there is no safe default, since the recommended `method = "supplied"` needs the name of the obs column holding the mask.
`cellsweep_params`	List. See `params_sc_cellsweep()`.
`sc_qc_param`	List. See `params_sc_min_quality()`. Only `target_size` is used, to scale the new normalised layer.
`csc_mem_gb`	Optional numeric. Memory in GB for the buffers of the cell-to-gene (CSR to CSC) conversion, at 10 bytes per non-zero. `NULL` (default) converts in one pass and holds the whole matrix. Set a cap for large data sets; every extra phase re-reads the cell file once.
`streaming`, `batch_size`, `max_genes_in_memory`, `cell_batch_size`	Replaced by `csc_mem_gb` and ignored. [**Deprecated**]
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Details

Two things about where this sits in the workflow, because both are easy to get wrong.

It runs after annotation, not before. The model subtracts against cell-type profiles, so it needs the labels on input. The chain is: ingest the raw barcodes, mask and cluster and annotate as usual, then `cellsweep_sc()`, then redo feature selection and reduction on the clean counts.

It needs the empty droplets. The ambient profile is estimated from their pooled counts, and they stay in the EM with their contamination fraction pinned at 1. That means the object has to have been ingested permissively:

```
obj <- load_mtx(
  obj,
  sc_mtx_io_param = get_cell_ranger_params(path),
  sc_qc_param = params_sc_min_quality(
    min_unique_genes = 0L, min_lib_size = 0L, min_cells = 0L
  )
)
```

The load-time cutoffs are irreversible, so the defaults (`min_lib_size = 250L`) delete exactly the barcodes CellSweep trains on. This function checks for that and errors rather than fitting a garbage profile.

One EM fit per sample, since the ambient profile is a property of a single emulsion. `sample_column` is required for that reason; pooling samples into one ambient profile is wrong even when it runs.

Barcodes partition three ways: empty droplets, annotated barcodes that passed QC, and everything else. The third group is excluded from the fit and does not appear in the output.

The denoised counts land in a new directory. The raw layer takes the stochastically rounded values so the negative binomial methods downstream still see integers, and the normalised layer keeps the float magnitudes. Entries whose denoised value rounds to zero are dropped from both layers: the two share one index set, so keeping them would put explicit zeros in the raw layer and `library_size` would stop being the sum of what is stored, breaking every consumer that computes a fraction of the library.

Value

The `target` object, populated with the denoised counts. The fit lands in obs as `cellsweep_alpha`, `cellsweep_z`, `cellsweep_beta`, `cellsweep_ll` and `cellsweep_converged`, and in var as `cellsweep_ambient`.

References

Sullivan et al., CellSweep, 2025

`change_gene_identifier` *Change the primary gene identifier of BulkDge*

Description

Changes the primary gene identifier in the BulkDge class. To do so, you need to either provide a `variable_info` data.table with the alternative gene identifier you wish to use or it exists already in the object itself. If it exists in the object, that variable_info will be used.

Usage

```
change_gene_identifier(object, alternative_gene_id, variable_info = NULL)
```

Arguments

`object`	`BulkDge` class, see `BulkDge()`.
`alternative_gene_id`	String. The column containing the alternative gene identifier. Must be present in the provided `variable_info` data.table or within the class attributes.
`variable_info`	Optional data.table with variable information. If `variable_info` is in an attribute of the class, that one will be used.

Value

The class with modified primary gene identifier.

Examples

```
# swap the row names over to a symbol column of the variable info
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(sample_id = colnames(syn$counts))
var_info <- data.table::data.table(
  var_id = rownames(syn$counts),
  symbol = sprintf("SYM%i", seq_len(nrow(syn$counts)))
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- change_gene_identifier(object, "symbol", var_info)
head(rownames(S7::prop(object, "raw_counts")))
```

`check_sc_state` *Check that cached artefacts still match the object's state*

Description

Returns `TRUE` when every requested artefact is fresh, otherwise a message describing the first problem. An artefact that is absent, or that predates provenance stamping, passes: unknown provenance is not the same as known bad.

Named in snake case rather than the `checkXxx` camel case of the other checkmate extensions in this package, because it validates object state rather than a `params_*()` list and sits alongside `get_sc_cache_status()`.

Usage

```
check_sc_state(x, artefacts, modality = "rna")
```

Arguments

x	SingleCells, SingleCellsSubset, MetaCells or SingleCellsMultiModal class.
artefacts	Character vector. Artefacts to check, either an artefact kind ("pca", "knn", "snn", "magic") or an embedding name ("umap"). May be modality qualified ("adt:pca").
modality	String. Modality to resolve unqualified names against. One of c("rna", "adt", "atac", "wnn").

Value

TRUE if the check was successful, otherwise a string describing the failure.

---

community_detection *Identify privileged communities based on a given diffusion vector*

Description

Detects privileged communities after a diffusion based on seed nodes.

Usage

```
community_detection(
  object,
  community_params = params_community_detection(),
  seed = 42L,
  .verbose = FALSE,
  .max_iters = 100L
)
```

Arguments

object	NetworkDiffusions object. The underlying class NetworkDiffusions().
community_params	List. Parameters for the community detection within the reduced network, see params_community_detection(). A list with the following items: • max_nodes - Integer. Number of maximum nodes per community. Larger communities will be recursively subclustered. • min_nodes - Integer. Minimum number of nodes per community. • min_seed_nodes - Integer. Minimum number of seed genes that have to be found in a given community. • initial_res - Float. Initial resolution parameter for the Leiden clustering.

- threshold_type - String. One of `c("prop_based", "pval_based")`. You can chose to include a certain proportion of the network (like in the original paper) with the highest diffusion scores, or use p-values based on permutations. Defaults to `"prop_based"`.
- network_threshold - Float. The proportion of the network to include. Used if `threshold_type = "prop_based"`.
- pval_threshold - Float. The maximum p-value for nodes to be included. Used if `threshold_type = "pval_based"`.

`seed` Random seed.

`.verbose` Controls the verbosity of the function.

`.max_iters` Controls how many iterations shall be tried for the sub-clustering. To note, in each iteration of the sub-clustering, the resolution parameter is increased by 0.05, to identify more granular communities within the sub communities.

Value

The class with added diffusion community detection results (if any could be identified with the provided parameters).

Examples

```
# privileged communities in the heated part of the network
set.seed(42)
g <- igraph::sample_pa(15, directed = FALSE)
edges <- data.table::setDT(igraph::as_data_frame(g))[, `:=`(
  from = sprintf("node_%i", from),
  to = sprintf("node_%i", to)
)]
object <- NetworkDiffusions(edges, weighted = FALSE, directed = FALSE)
object <- diffuse_seed_nodes(object, c(node_1 = 1, node_3 = 1), "max")
object <- permute_seed_nodes(object, perm_iters = 100L, .verbose = FALSE)
object <- community_detection(
  object,
  community_params = params_community_detection(
    min_seed_nodes = 0L,
    min_nodes = 2L
  )
)
head(get_results(object))
```

---

compute_expression_info_sc

Compute per-cluster mean expression and expressing fraction for a gene set

---

Description

Thin R wrapper around the Rust `compute_cluster_expression_stats` routine. Streams gene chunks from the on-disk store and aggregates expression across user-supplied cell clusters. Cells outside any cluster are ignored.

If `condition_colname` and `condition_oi` are supplied, only cells from that condition contribute to the aggregation.

Usage

```
compute_expression_info_sc(
  object,
  celltype_colname,
  genes,
  condition_colname = NULL,
  condition_oi = NULL
)
```

Arguments

`object`	A `SingleCells` object.
`celltype_colname`	Name of the cluster column in `obs`.
`genes`	Character vector of gene IDs to aggregate over.
`condition_colname`	Optional. Name of a condition column in `obs`.
`condition_oi`	Optional. Value of `condition_colname` to subset to.

Value

A long `data.table` with columns `cluster_id`, `gene`, `avg_expr`, `frac_expr`.

Examples

```
# mean expression and expressing fraction per planted cell type
sc <- demo_single_cells()
res <- compute_expression_info_sc(
  sc,
  celltype_colname = "cell_grp",
  genes = get_gene_names(sc)[1:5]
)
head(res)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`consensus_nmf_bulk` *Run consensus NMF on a BulkCoExp*

Description

Runs `n_runs` HALS-NMF restarts, pools their components, drops unstable ones by local density, k-means clusters the survivors into `k` groups and refits the partner factor against the per-cluster median. This is cNMF: the answer is the structure the restarts agree on, and the mean silhouette of those clusters (`stability`) tells you how much they agreed.

Prefer this over `stabilised_nmf_bulk()`, which just picks the lowest-loss restart and tells you nothing about whether that restart is reproducible.

Usage

```
consensus_nmf_bulk(
  object,
  k,
  n_runs = 30L,
  preprocessing = c("none", "sd", "sqrt_sd"),
  nmf_hals_params = params_nmf_hals(),
  nmf_consensus_params = params_nmf_consensus(),
  membership_params = params_module_membership(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	The class, see `BulkCoExp()`. Ideally, you should run `preprocess_bulk_coexp()` before applying this function. The NMF requires non-negative input, so the pre-processing should either skip scaling or use a strictly non-negative representation.
`k`	Integer. Number of latent factors (modules). Must be at least 2.
`n_runs`	Integer. Number of random restarts. Must be at least 2.
`preprocessing`	String. One of `c("none", "sd", "sqrt_sd")`. Applied inside the Rust kernel to the input matrix before the HALS updates.
`nmf_hals_params`	List. Output of `params_nmf_hals()`. The `nmf_init` field is ignored, restarts always use random initialisation.
`nmf_consensus_params`	List. Output of `params_nmf_consensus()`. Controls the clustering of the pooled components.
`membership_params`	List. Controls how the gene loadings are turned into module membership, see `params_module_membership()`.

`seed`	Integer. Base random seed. Restart `i` uses `seed + i`, and the k-means step is seeded from it too.
`.verbose`	Boolean or integer `0L`/`1L`/`2L`. Controls verbosity.

Details

Use `nmf_k_sweep_bulk()` first if you do not already know `k`.

The density filter is the part that bites. Components whose mean cosine distance to their neighbours exceeds `density_threshold` are dropped, and if that leaves fewer than `k` survivors the fit errors rather than returning something degenerate. With few restarts the filter is jumpy, so either raise `n_runs` or set `density_threshold = 2` to switch it off.

Value

The class with `final_results` populated from the consensus fit, with the clustering diagnostics under `diagnostics` and the fit parameters under `params$nmf_fit`.

References

Kotliar et al., eLife, 2019

Examples

```
# consensus over ten restarts
syn <- generate_gene_module_data(n_samples = 24L, n_genes = 60L)
# NMF needs a non-negative matrix
mat <- syn$data - min(syn$data)
obj <- BulkCoExp(mat, syn$meta_data)
obj <- preprocess_bulk_coexp(
  obj, hvg = NULL, scaling = FALSE, .verbose = FALSE
)
obj <- consensus_nmf_bulk(obj, k = 4L, n_runs = 10L, .verbose = FALSE)
get_nmf_stability(obj)$stability
```

`consensus_nmf_sc` *Run consensus NMF on single cell or meta cell data*

Description

Runs `n_runs` HALS NMF restarts, pools their components, drops unstable ones by local density, k-means clusters the survivors into `k` groups and refits the partner factor against the per-cluster median. This is cNMF: the answer is the structure the restarts agree on, and the mean silhouette of those clusters (`stability`) says how much they agreed.

Usage

```
consensus_nmf_sc(
  object,
  k,
  cell_ids = NULL,
  gene_ids = NULL,
  preprocessing = "none",
  use_second_layer = TRUE,
  nmf_hals_params = params_nmf_hals(),
  nmf_consensus_params = params_nmf_consensus(),
  n_runs = 30L,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `MetaCells` class.
`k`	Integer. Number of latent factors. Must be at least 2.
`cell_ids`	Optional character. Cell ids (or meta cell ids) to restrict the NMF to. If `NULL`, uses `get_cells_to_keep()` for `SingleCells` and all meta cells for `MetaCells`.
`gene_ids`	Optional character. Gene ids to restrict the NMF to. If `NULL`, uses `get_hvg()` on the object.
`preprocessing`	String. One of `c("none", "sd", "sqrt_sd")`.
`use_second_layer`	Boolean. If `TRUE`, runs NMF on the normalised counts (recommended); if `FALSE`, on the raw counts.
`nmf_hals_params`	List, see `params_nmf_hals()`. The `nmf_init` field is ignored, restarts always use random initialisation.
`nmf_consensus_params`	List, see `params_nmf_consensus()`.
`n_runs`	Integer. Number of random restarts. Must be at least 2.
`seed`	Integer. Base random seed. Restart `i` uses `seed + i`, and the k-means step is seeded from it too.
`.verbose`	Boolean or integer. Verbosity.

Details

Use `nmf_k_sweep_sc()` first if you do not already know `k`.

Memory is the thing to watch. The restarts are dense and all live at once: roughly `n_cells * k * n_runs` plus `n_runs * k * n_genes` floats on top of the counts. At 200k cells, `k = 20` and `n_runs = 50` that is a few hundred megabytes before anything else, which is exactly the regime where running on `MetaCells` instead is the honest answer.

The density filter is the part that bites. If it leaves fewer than `k` components the fit errors rather than returning something degenerate. With few restarts the filter is jumpy, so either raise `n_runs` or set `density_threshold = 2` in `params_nmf_consensus()` to switch it off.

Value

A `ConsensusNmfResult` object.

References

Kotliar et al., eLife, 2019

Examples

```
# ten restarts pooled, density filter off on data this small
sc <- demo_single_cells()
res <- consensus_nmf_sc(
  sc,
  k = 3L,
  n_runs = 10L,
  nmf_consensus_params = params_nmf_consensus(density_threshold = 2),
  .verbose = FALSE
)
res

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

constrained_page_rank

Constrained personalised page rank

Description

This function implements a constrained personalised page-rank. You can define `sink_nodes` or `sink_edges`. In the case of the former, the vist of a sink_node (via a type attribute in the igraph) will automatically cause the surfer to reset. In the case of the latter, the traversal of a sink_edge is allowed, however, subsequently, the surfer will be reseted. This function can be useful for deriving PPR profiles under constraints in heterogenous graphs and has been inspired by Ruiz, et al.

Usage

```
constrained_page_rank(
  graph,
  personalisation_vector,
  sink_nodes = NULL,
  sink_edges = NULL
)
```

Arguments

`graph`	igraph. This one needs to be directed and weighted and have the node attribute `type` defining the node type and the edge attribute `type` defining the edge type.
`personalisation_vector`	Numerical vector. The personalisation vector for the reset of the personalised page rank. The function will automatically normalise it.
`sink_nodes`	Optional String vector. The node types that should force a reset.
`sink_edges`	Optional String vector. The edge types after which there should be a forced reset.

Value

The constrained personalised page rank

References

Ruiz, et al., Nat Commun, 2021

Examples

```
# personalised page rank from a receptor over a small signalling graph
nodes <- data.frame(
  name = c("rec_a", "kin_b", "tf_c", "gene_d"),
  type = c("receptor", "kinase", "tf", "target_gene")
)
edges <- data.frame(
  from = c("rec_a", "kin_b", "tf_c"),
  to = c("kin_b", "tf_c", "gene_d"),
  weight = rep(1, 3),
  type = c("activation", "phosphorylation", "tf_activation")
)
g <- igraph::graph_from_data_frame(edges, vertices = nodes, directed = TRUE)
constrained_page_rank(g, personalisation_vector = c(1, 0, 0, 0))
```

`constrained_page_rank_ls` *Constrained personalised page rank over a list*

Description

This function implements a personalised constrained page-rank over a list of personalisation vectors for the same network. You can define `sink_nodes` or `sink_edges`. In the case of the former, the vist of a sink_node (via a type attribute in the igraph) will automatically cause the surfer to reset. In the case of the latter, the traversal of a sink_edge is allowed, however, subsequently, the surfer will be reseted. This function can be useful for deriving PPR profiles under constraints in heterogenous graphs and has been inspired by Ruiz, et al.

Usage

```
constrained_page_rank_ls(
  graph,
  personalisation_list,
  sink_nodes = NULL,
  sink_edges = NULL
)
```

Arguments

graph	igraph. This one needs to be directed and weighted and have the node attribute `type` defining the node type and the edge attribute `type` defining the edge type.
personalisation_list	A list of numerical vectors to use for the personalisation.
sink_nodes	Optional String vector. The node types that should force a reset.
sink_edges	Optional String vector. The edge types after which there should be a forced reset.

Value

A list with the constrained personalised page rank values.

References

Ruiz, et al., Nat Commun, 2021

Examples

```
# two personalisation vectors over the same signalling graph
nodes <- data.frame(
  name = c("rec_a", "kin_b", "tf_c", "gene_d"),
  type = c("receptor", "kinase", "tf", "target_gene")
)
edges <- data.frame(
  from = c("rec_a", "kin_b", "tf_c"),
  to = c("kin_b", "tf_c", "gene_d"),
  weight = rep(1, 3),
  type = c("activation", "phosphorylation", "tf_activation")
)
g <- igraph::graph_from_data_frame(edges, vertices = nodes, directed = TRUE)
res <- constrained_page_rank_ls(
  g,
  personalisation_list = list(a = c(1, 0, 0, 0), b = c(0, 1, 0, 0))
)
res$a
```

`contrastive_pca` *Apply contrastive PCA.*

Description

Applies the contrastive PCA algorithm given a specified alpha and a number of contrastive principal components to extract.

Usage

```
contrastive_pca(object, alpha, no_pcs)
```

Arguments

`object`	The underlying class, see `BulkCoExp()`.
`alpha`	Alpha parameter to use.
`no_pcs`	Number of contrastive PCs to generate.

Value

`BulkCoExp` with additional data in the slots

References

Abid, et al., Nature Communications, 2018

Examples

```
# five contrastive PCs at alpha 2.5
cpca_data <- synthetic_c_pca_data()
target <- t(cpca_data$target)
background <- t(cpca_data$background)
meta <- data.table::data.table(sample_id = rownames(target))
obj <- BulkCoExp(target, meta)
obj <- preprocess_bulk_coexp(obj, .verbose = FALSE)
obj <- contrastive_pca_processing(obj, background, .verbose = FALSE)
obj <- contrastive_pca(obj, alpha = 2.5, no_pcs = 5L)
dim(get_c_pca_factors(obj))
```

`contrastive_pca_processing` *Prepare class for contrastive PCA*

Description

This function will prepare the `BulkCoExp` for subsequent usage of the contrastive PCA functions. This is based on the work of Abid, et al.

Usage

```
contrastive_pca_processing(
  object,
  background_matrix,
  scale = FALSE,
  .verbose = TRUE
)
```

Arguments

`object`	The underlying class, see `BulkCoExp()`.
`background_matrix`	Numeric matrix. The background matrix you wish to remove. You should apply any data transformation to this matrix, too!
`scale`	Boolean. Shall the data be scaled. Defaults to FALSE.
`.verbose`	Boolean. Controls verbosity of the function.

Value

`BulkCoExp` with the needed data for contrastive PCA in the properties of the class.

References

Abid, et al., Nature Communications, 2018

Examples

```
# covariance matrices of the target and the background
cpca_data <- synthetic_c_pca_data()
target <- t(cpca_data$target)
background <- t(cpca_data$background)
meta <- data.table::data.table(sample_id = rownames(target))
obj <- BulkCoExp(target, meta)
obj <- preprocess_bulk_coexp(obj, .verbose = FALSE)
obj <- contrastive_pca_processing(obj, background, .verbose = FALSE)
dim(obj@processed_data$target_covar)
```

`cor_module_check_epsilon` *Iterate through different epsilon parameters*

Description

This functions iterates through a set of provided epsilons and checks for each one to what extend the underlying affinity matrix will follow a power law distribution.

Usage

```
cor_module_check_epsilon(
  object,
  rbf_func = c("bump", "gaussian", "inverse_quadratic"),
  epsilons = c(0.25, seq(from = 0.5, to = 10, by = 0.5)),
  .verbose = TRUE
)
```

Arguments

`object`	The class, see `BulkCoExp()`. You need to run `cor_module_processing()` before running this function.
`rbf_func`	The type of RBF function you want to apply. A choice of `c('bump', 'gaussian', 'inverse_quadratic')`.
`epsilons`	Vector of floats. The different epsilon parameters you would like to run.
`.verbose`	Boolean. Controls verbosity of the function.

Value

The class with added data to the properties for subsequent usage.

Examples

```
# scan epsilons of a Gaussian RBF for scale-free topology
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- cor_module_processing(obj, cor_method = "spearman", .verbose = FALSE)
obj <- cor_module_check_epsilon(
  obj, rbf_func = "gaussian", .verbose = FALSE
)
head(get_epsilon_res(obj))
```

`cor_module_coremo_clustering` *Generates CoReMo-based gene modules*

Description

This function creates gene modules, based on the framework from Srivastava et al., 2018. Briefly, it applies an RBF function to the correlation matrix to reduce the impact of weak correlations, leverages hierarchical clustering for clustering the genes. It optimises the R2 (cor^2) within the clusters to identify the optimal cut. Gene modules with low R2 are being considered as the 'junk module'.

Usage

```
cor_module_coremo_clustering(
  object,
  span = 0.1,
  coremo_params = params_coremo(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	The class, see `BulkCoExp()`.
`span`	Float. Defines the span parameter for the loess function to identify the inflection point. Defaults to `0.1`. Must be between 0.1 and 1.
`coremo_params`	List. Parameters for the generation of the CoReMo modules, see `params_coremo()`. Contains: • epsilon - Float. The epsilon parameter for the RBF. You can optimise that one with `cor_module_check_epsilon()`. Defaults to `2`. • k_min - Integer. Minimum number of cuts. Defaults to `2L`. • k_max - Integer. Maximum number of cuts. Defaults to `150L`. • min_size - Optional integer. Minimum size of the clusters. If provided, smaller clusters will be merged by eigengene similarity. • junk_module_threshold - Float. Minimum R2 median value for a module to not be considered a junk module. Defaults to `0.05`. • rbf_func - String. Type of RBF function to apply. Defaults to `"gaussian"`. • cor_method - String. Type of correlation method to use for merging the smaller cluster. Defaults to `"spearman"`.
`seed`	Integer. Random seed for reproducibility purposes.
`.verbose`	Boolean. Controls verbosity of the function.

Value

The class with added data to the properties for subsequent usage.

References

Srivastava, et al., Nat. Commun., 2018

Examples

```
# CoReMo clustering with the default RBF and cut range
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- cor_module_processing(obj, cor_method = "spearman", .verbose = FALSE)
obj <- cor_module_coremo_clustering(obj, .verbose = FALSE)
head(obj@outputs$final_modules)
```

---

cor_module_coremo_cor_sign

Split CoReMo modules by correlation sign

Description

This function will split the identified modules by their correlation sign. In certain cases, positive and negatively correlated genes can be part of the same module. This function will annotate them with `"_pos"` and `"_neg"` respectively.

Usage

```
cor_module_coremo_cor_sign(object, min_stability = NULL, .verbose = TRUE)
```

Arguments

`object`	The class, see `BulkCoExp()`.
`min_stability`	Optional float. The minimum stability for the gene you wish to filter for based on the leave-one-out resampling. If `NULL`, no filtering will be applied.
`.verbose`	Boolean. Controls verbosity of the function.

Value

The class with updated correlation module names.

References

Srivastava, et al., Nat. Commun., 2018; Francois, Romagnolo, et al., Nat. Commun., 2024.

Examples

```
# split the modules into positively and negatively correlated genes
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- cor_module_processing(obj, cor_method = "spearman", .verbose = FALSE)
obj <- cor_module_coremo_clustering(obj, .verbose = FALSE)
obj <- cor_module_coremo_cor_sign(obj, .verbose = FALSE)
table(obj@outputs$final_modules$sign)
```

---

cor_module_coremo_eigengene

Calculate Eigengenes for CoReMo modules

Description

This function will calculate the eigengene values for the modules on a per sample basis and add correlations of the gene expression of a given gene within the module to its eigengene.

Usage

```
cor_module_coremo_eigengene(object, min_stability = NULL, .verbose = TRUE)
```

Arguments

`object`	The class, see `BulkCoExp()`.
`min_stability`	Optional float. The minimum stability for the gene you wish to filter for based on the leave-one-out resampling. If `NULL`, no filtering will be applied.
`.verbose`	Boolean. Controls verbosity of the function.

Value

The class with added correlations to the modules and the values for a given eigengene per sample as a data.table.

References

Srivastava, et al., Nat. Commun., 2018; Francois, Romagnolo, et al., Nat. Commun., 2024.

Examples

```
# eigengenes per module and the gene to eigengene correlations
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- cor_module_processing(obj, cor_method = "spearman", .verbose = FALSE)
obj <- cor_module_coremo_clustering(obj, .verbose = FALSE)
obj <- cor_module_coremo_eigengene(obj, .verbose = FALSE)
head(get_modules(get_results(obj)))
```

`cor_module_coremo_stability` *Assesses CoReMo-based gene module stability*

Description

The function assesses the stability of the CoReMo modules, leveraging a leave-one-out sample method. In each iteration, one of the samples is left out and the clustering is redone. Subsequently, the Jaccard similarity (as a surrogate for membership stability) is calculated for each feature across the different resamplings and added to the `final_modules` data.table.

Usage

```
cor_module_coremo_stability(object, chunk_size = 15L, .verbose = TRUE)
```

Arguments

`object`	The class, see `BulkCoExp()`.
`chunk_size`	Integer. Chunk size in which to process the data. Defaults to `15L`, i.e., 15 samples are being processed in one go. You can use bigger values here, but be aware of memory pressure.
`.verbose`	Boolean. Controls verbosity of the function.

Value

The class with added data to the properties for subsequent usage.

References

Srivastava, et al., Nat. Commun., 2018; Francois, Romagnolo, et al., Nat. Commun., 2024.

Examples

```
# leave-one-out stability of the CoReMo modules
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- cor_module_processing(obj, cor_method = "spearman", .verbose = FALSE)
obj <- cor_module_coremo_clustering(obj, .verbose = FALSE)
obj <- cor_module_coremo_stability(obj, .verbose = FALSE)
summary(obj@outputs$final_modules$stability)
```

`cor_module_graph_check_res` *Iterate through Leiden resolutions for graph-based community detection.*

Description

This function will identify gene modules based on affinity graphs from the single correlation or differential correlation methods. Briefly, in the case of single correlation, the graph is generated based on the absolute correlation coefficients that are subjected to a Gaussian affinity kernel. This reduces spurious correlations and leaves a sparsely connected graph. In the case of differential correlations, the graph is generated based on significant differential correlations if one of the two correlations reached the defined minimum thresholds.
Subsequently, Leiden community detection is applied on the respective graph through a range of resolutions that the user can define. The function then returns meta information about the resolutions (which can also be plotted) to identify the best suitable resolution parameter to identify co-expression modules.

Usage

```
cor_module_graph_check_res(
  object,
  resolution_params = params_graph_resolution(),
  graph_params = params_cor_graph(),
  random_seed = 123L,
  min_genes = 10L,
  parallel = TRUE,
  max_workers = NULL,
  .verbose = TRUE
)
```

Arguments

`object` The class, see `BulkCoExp()`.

`resolution_params`
List. Parameters for the resolution search, see `params_graph_resolution()`. Contains:

- min_res - Float. Minimum resolution to test.
- max_res - Float. Maximum resolution to test.
- number_res - Integer. Number of resolutions to test between the `max_res` and `min_res`.

`graph_params` List. Parameters for the generation of the (differential) correlation graph, see `params_cor_graph()`. Contains:

- Epsilon - Defines the epsilon parameter for the radial basis function. Defaults to 1, but should be ideally optimised.

- min_cor - Float. Minimum absolute correlation that needs to be observed in either data set. Only relevant for differential correlation-based graphs.
- fdr_threshold - Float. Maximum FDR for the differential correlation p-value.
- verbose - Boolean. Controls verbosity of the graph generation.

`random_seed` Integer. Random seed.

`min_genes` Integer. Minimum number of genes that should be in a community.

`parallel` Boolean. Parallelise the Leiden clustering.

`max_workers` Optional Integer. Number of cores to use if parallel is set to `TRUE`. If set to `NULL` it will automatically detect the number of cores.

`.verbose` Controls the verbosity of the function.

Value

The class with added data to the properties.

Examples

```
# scan Leiden resolutions on the correlation graph
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- cor_module_processing(obj, cor_method = "spearman", .verbose = FALSE)
obj <- cor_module_graph_check_res(
  obj,
  resolution_params = params_graph_resolution(number_res = 5L),
  parallel = FALSE,
  .verbose = FALSE
)
head(get_resolution_res(obj))
```

---

cor_module_graph_final_modules

Identify correlation-based gene modules via graphs

Description

This function leverages graph-based clustering to identify gene co-expression modules. The class has the option to sub-cluster large communities within their respective sub graphs, akin to the approach taken by Barrio-Hernandez, et al.

Usage

```
cor_module_graph_final_modules(
  object,
  resolution = NULL,
  min_size = 10L,
  max_size = 500L,
  subclustering = TRUE,
  random_seed = 123L,
  .graph_params = params_cor_graph(),
  .max_iters = 100L,
  .verbose = TRUE
)
```

Arguments

`object`	The class, see `BulkCoExp()`.
`resolution`	The Leiden resolution parameter you wish to use. If NULL, it will use the optimal one identified by `cor_module_graph_check_res()`. If nothing can be found, will default to 1.
`min_size`	Integer. Minimum size of the communities.
`max_size`	Integer. Maximum size of the communities.
`subclustering`	Boolean. Shall after a first clustering communities that are too large be further sub clustered. Defaults to `TRUE`.
`random_seed`	Integer. Random seed.
`.graph_params`	List. Parameters for the generation of the (differential) correlation graph, see `params_cor_graph()`. Contains: • Epsilon - Defines the epsilon parameter for the radial basis function. Defaults to 2, but should be ideally optimised. • min_cor - Float. Minimum absolute correlation that needs to be observed in either data set. Only relevant for differential correlation-based graphs. • fdr_threshold - Float. Maximum FDR for the differential correlation p-value. • verbose - Boolean. Controls verbosity of the graph generation. This parameter is only relevant if you did *not* run `cor_module_graph_check_res()`.
`.max_iters`	Integer. If sub clustering is set to `TRUE`, what shall be the maximum number of iterations. Defaults to 100L.
`.verbose`	Boolean. Controls the verbosity of the function.

Value

The class with added data to the properties.

References

Barrio-Hernandez, et al., Nat Genet, 2023.

Examples

```
# finalise the communities at the resolution picked by the scan
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- cor_module_processing(obj, cor_method = "spearman", .verbose = FALSE)
obj <- cor_module_graph_check_res(
  obj,
  resolution_params = params_graph_resolution(number_res = 5L),
  parallel = FALSE,
  .verbose = FALSE
)
obj <- cor_module_graph_final_modules(obj, .verbose = FALSE)
head(get_modules(get_results(obj)))
```

---

cor_module_processing

Prepare correlation-based module detection

Description

This function will calculate the correlation coefficients between the genes, using the highly variable genes (if available, otherwise the function will use the raw data). The data will be stored in a memory-efficient format in the properties of the class.

Usage

```
cor_module_processing(
  object,
  cor_method = c("pearson", "spearman"),
  .verbose = TRUE
)
```

Arguments

`object`	The class, see `BulkCoExp()`. Ideally, you should run `preprocess_bulk_coexp()` before applying this function.
`cor_method`	String. Option of `c("pearson", "spearman")`.
`.verbose`	Boolean. Controls verbosity of the function.

Value

The class with added data to the properties for subsequent usage.

Examples

```
# spearman correlations over the 300 most variable genes
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- cor_module_processing(obj, cor_method = "spearman", .verbose = FALSE)
dim(obj@processed_data$correlation_res$get_sym_matrix(
  .verbose = FALSE
))
```

---

cor_module_tom *Update the correlation matrix to a TOM*

Description

This function will update the correlation matrix to a topological overlap matrix. It defaults to "v2" and the signed version, please see calculate_tom() for details.

Usage

```
cor_module_tom(object, signed = TRUE, version = c("v2", "v1"), .verbose = TRUE)
```

Arguments

object	The class, see BulkCoExp(). You need to have applied cor_module_processing() before applying this function.
signed	Boolean. Do you want to use the signed or unsigned version. Defaults to TRUE.
version	String. One of c("v2", "v1"). Defaults to "v2".
.verbose	Boolean. Controls verbosity of the function.

Value

The class with added data to the properties for subsequent usage.

Examples

```
# replace the stored correlations with a signed topological overlap
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- cor_module_processing(obj, cor_method = "spearman", .verbose = FALSE)
obj <- cor_module_tom(obj, signed = TRUE, version = "v2", .verbose = FALSE)
obj@params$correlation_params$TOM
```

`demo_single_cells` *Ready-made* `SingleCells` *object for examples and tests*

Description

Wires `generate_single_cell_test_data()` into a `SingleCells` object on disk in one call, so examples do not have to repeat the whole ingestion dance. The default is deliberately tiny (500 cells x 50 genes) and the quality thresholds are loose enough that every cell survives. This is synthetic data for demonstration and testing, not something to analyse.

Usage

```
demo_single_cells(
  dir = tempfile("bixverse_demo"),
  prepped = TRUE,
  syn_data_params = params_sc_synthetic_data(n_cells = 500L, n_genes = 50L),
  hvg_no = 30L,
  no_pcs = 10L,
  k = 15L,
  seed = 42L,
  .verbose = FALSE
)
```

Arguments

`dir`	String. Directory to hold the object. Created if it does not exist. Defaults to a fresh path under the session `tempdir()`. Remove it with `unlink(dir, recursive = TRUE)` when you are done.
`prepped`	Boolean. Run the standard HVG -> PCA -> kNN chain before returning? Defaults to `TRUE`.
`syn_data_params`	List. Parameters for the synthetic data, see `params_sc_synthetic_data()`.
`hvg_no`	Integer. Number of highly variable genes, `prepped` only.
`no_pcs`	Integer. Number of principal components, `prepped` only.
`k`	Integer. Number of nearest neighbours, `prepped` only.
`seed`	Integer. Seed for the data generation.
`.verbose`	Boolean. Controls verbosity of the function.

Value

A `SingleCells` object backed by `dir`.

Examples

```
# a prepped object, ready for clustering
sc <- demo_single_cells()
sc <- find_clusters_sc(sc, res = 1.0)
sc

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

densmap_sc *Run densMAP on a SingleCells/MetaCells object*

Description

Wrapper around `manifoldsR::densmap()` for the `SingleCells` and `MetaCells` classes. densMAP is UMAP plus a density-preserving term: a tight population stays tight in the embedding and a diffuse one stays diffuse. With plain UMAP the relative size of a cluster on the plot tells you nothing, with densMAP it does. Setting `lambda = 0` in `manifoldsR::params_densmap()` gives you back plain UMAP.

Neighbour handling is the same as in `umap_sc()`: with `use_knn = TRUE` (the default) the cached kNN graph is reused, otherwise neighbours are computed from the chosen embedding.

Usage

```
densmap_sc(
  object,
  use_knn = TRUE,
  embd_to_use = "pca",
  slot_name = "densmap",
  no_embd_to_use = NULL,
  modality = c("rna", "adt", "wnn"),
  n_dim = 2L,
  k = 15L,
  min_dist = 0.5,
  spread = 1,
  knn_method = c("kmknn", "hnsw", "annoy", "nndescent", "balltree", "ivf", "exhaustive"),
  nn_params = manifoldsR::params_nn(),
  umap_params = manifoldsR::params_umap(),
  dens_params = manifoldsR::params_densmap(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells`, `MetaCells` class.

use_knn	Boolean. Use the kNN graph found in the object. Defaults to TRUE. If not available, will default to the embedding.
embd_to_use	String. The embedding to use for densMAP. Must be available in the object.
slot_name	String. The name of this embedding within the object. Defaults to "densmap".
no_embd_to_use	Optional integer. Number of embedding dimensions to use. If NULL all will be used.
modality	String. On which modality to run densMAP. One of c("rna", "adt", "wnn"). The two latter options are only available for multi-modal versions with the added data.
n_dim	Integer. Number of densMAP dimensions. Defaults to 2L.
k	Integer. Number of nearest neighbours. Defaults to 15L.
min_dist	Numeric. Minimum distance between embedded points. Defaults to 0.5.
spread	Numeric. Effective scale of embedded points. Defaults to 1.0.
knn_method	String. Approximate nearest neighbour algorithm. One of c("kmknn", "hnsw", "annoy", "nndescent", "balltree", "ivf", "exhaustive"). Defaults to "kmknn". Only used when neighbours are computed from the embedding.
nn_params	Named list. See manifoldsR::params_nn().
umap_params	Named list. See manifoldsR::params_umap().
dens_params	Named list. The density knobs, see manifoldsR::params_densmap().
seed	Integer. For reproducibility.
.verbose	Boolean. Controls verbosity.

Value

The object with a "densmap" embedding added.

References

Narayan, Berger & Cho, Nat. Biotechnol., 2021

Examples

```
# densMAP off the cached kNN graph
sc <- demo_single_cells()
sc <- densmap_sc(sc, .verbose = FALSE)
dim(get_embedding(sc, "densmap"))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`densne_sc` *Run den-SNE on a SingleCells/MetaCells object*

Description

Wrapper around `manifoldsR::densne()` for the `SingleCells` and `MetaCells` classes. den-SNE is t-SNE plus a density-preserving term: a tight population stays tight in the embedding and a diffuse one stays diffuse. Plain t-SNE inflates dense clusters and shrinks sparse ones, so relative cluster sizes on the plot mean nothing; with den-SNE they do. Setting `lambda = 0` in `manifoldsR::params_densne()` gives you back plain t-SNE.

Neighbour handling, `approx_type` and the `k` versus `perplexity` caveats are the same as in `tsne_sc()`.

Usage

```
densne_sc(
  object,
  use_knn = FALSE,
  embd_to_use = "pca",
  slot_name = "densne",
  no_embd_to_use = NULL,
  modality = c("rna", "adt", "wnn"),
  n_dim = 2L,
  perplexity = 10,
  approx_type = c("bh", "fft", "fft_3k"),
  knn_method = c("kmknn", "hnsw", "annoy", "nndescent", "balltree", "ivf", "exhaustive"),
  nn_params = manifoldsR::params_nn(),
  tsne_params = manifoldsR::params_tsne(),
  dens_params = manifoldsR::params_densne(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells`, `MetaCells` class.
`use_knn`	Boolean. Use the kNN graph found in the object. Defaults to `FALSE`. If not available, will default to the embedding.
`embd_to_use`	String. The embedding to use for den-SNE. Must be available in the object.
`slot_name`	String. The name of this embedding within the object. Defaults to `"densne"`.
`no_embd_to_use`	Optional integer. Number of embedding dimensions to use. If `NULL` all will be used.

modality	String. On which modality to run den-SNE. One of c("rna", "adt", "wnn"). The two latter options are only available for multi-modal versions with the added data.
n_dim	Integer. Number of den-SNE dimensions. Currently only 2L is supported. Defaults to 2L.
perplexity	Numeric. Perplexity parameter. Typical values between 5 and 50. Defaults to 10.0.
approx_type	String. Approximation method. One of "bh" (Barnes-Hut), "fft" or "fft_3k". Defaults to "bh".
knn_method	String. Approximate nearest neighbour algorithm. One of c("kmknn", "hnsw", "annoy", "nndescent", "balltree", "ivf", "exhaustive"). Defaults to "kmknn". Only used when neighbours are computed from the embedding.
nn_params	Named list. See manifoldsR::params_nn().
tsne_params	Named list. See manifoldsR::params_tsne().
dens_params	Named list. The density knobs, see manifoldsR::params_densne().
seed	Integer. For reproducibility.
.verbose	Boolean. Controls verbosity.

Value

The object with a "densne" embedding added.

References

Narayan, Berger & Cho, Nat. Biotechnol., 2021

Examples

```
# Barnes-Hut den-SNE on the PCA factors
sc <- demo_single_cells()
sc <- densne_sc(sc, .verbose = FALSE)
dim(get_embedding(sc, "densne"))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

detect_adt_isotypes *Detect likely isotype-control features by name pattern*

Description

Detect likely isotype-control features by name pattern

Usage

```
detect_adt_isotypes(feature_names, pattern = "isotype")
```

Arguments

`feature_names`	Character. ADT feature names (matrix colnames).
`pattern`	String. Case-insensitive regex. Defaults to `"isotype"`.

Value

Character vector of matching names, for inspection before passing to `add_adt_counts_sc()` as `isotype_names`.

Examples

```
# find the isotype controls among the ADT features
features <- c("CD3", "CD19", "IgG1_isotype", "IgG2a_isotype")
detect_adt_isotypes(features)
```

`dgrdl_grid_search` *Grid search over DGRDL parameters*

Description

This function allows you to quickly iterate over different initial seeds, number of neighbours for the KNN graph and dictionary sizes to identify optimal hyperparameters for your DGRDL run.

Usage

```
dgrdl_grid_search(
  object,
  neighbours_vec,
  dict_size_vec,
  seed_vec,
  dgrdl_params = params_dgrdl(),
  .verbose = TRUE
)
```

Arguments

`object`	The class, see `BulkCoExp()`. Ideally, you should run `preprocess_bulk_coexp()` before applying this function.
`neighbours_vec`	Integer vector. The different k nearest neighbours to test.
`dict_size_vec`	Integer vector. The different dictionary sizes to test for.
`seed_vec`	Integer vector. The different initial seeds to test for the dictionary generation.
`dgrdl_params`	List. Output of `params_dgrdl()`:

- sparsity - Integer. Sparsity constraint (max non-zero coefficients per signal)
- dict size - Integer. Will be ignored by this function and the `dict_size_vec` vector will be used.
- alpha - Float. Sample context regularisation weight.
- beta - Float. Feature effect regularisation weight.
- max_iter - Integer. Maximum number of iterations for the main algorithm.
- k_neighbours - Integer. Will be ignored by this function and the `neighbours_vec` will be used.
- admm_iter - Integer. ADMM iterations for sparse coding.
- rho - Float. ADMM step size.

`.verbose` Boolean. Controls verbosity of the function.

Value

`BulkCoExp` with the grid search results added to the class.

References

Pan et al., Cell Syst, 2022

Examples

```
# small grid over neighbours and dictionary sizes
syn <- generate_gene_module_data(n_samples = 24L, n_genes = 60L)
obj <- BulkCoExp(syn$data, syn$meta_data)
obj <- preprocess_bulk_coexp(obj, hvg = NULL, .verbose = FALSE)
obj <- dgrdl_grid_search(
  obj,
  neighbours_vec = c(3L, 5L),
  dict_size_vec = c(4L, 6L),
  seed_vec = 123L,
  .verbose = FALSE
)
get_grid_search_res(obj)
```

`dgrdl_result` *Run DGRDL with the specified parameters*

Description

Runs the DGRDL algorithm from Pan et al., with the specified hyperparamters. To determine the hyperparameters, you can use `dgrdl_grid_search()`.

Usage

```
dgrdl_result(
  object,
  dgrdl_params = params_dgrdl(),
  membership_params = params_module_membership(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	The class, see `BulkCoExp()`. Ideally, you should run `preprocess_bulk_coexp()` before applying this function.
`dgrdl_params`	List. Output of `params_dgrdl()`: • sparsity - Integer. Sparsity constraint (max non-zero coefficients per signal) • dict size - Integer. The dictionary size. • alpha - Float. Sample context regularisation weight. • beta - Float. Feature effect regularisation weight. • max_iter - Integer. Maximum number of iterations for the main algorithm. • k_neighbours - Integer. Number of neighbours for the KNN graph for the feature and sample Laplacian. • admm_iter - Integer. ADMM iterations for sparse coding. • rho - Float. ADMM step size.
`membership_params`	List. Controls how the atom loadings are turned into module membership, see `params_module_membership()`. Membership is not exclusive: a gene active in several atoms appears in several modules, and a gene in no tail appears in none.
`seed`	Integer. Seed for the initialisation of the dictionary.
`.verbose`	Boolean. Controls verbosity of the function.

References

Pan et al., Cell Syst, 2022

Examples

```
# fit DGRDL with a six-atom dictionary
syn <- generate_gene_module_data(n_samples = 24L, n_genes = 60L)
obj <- BulkCoExp(syn$data, syn$meta_data)
obj <- preprocess_bulk_coexp(obj, hvg = NULL, .verbose = FALSE)
obj <- dgrdl_result(
  obj,
  dgrdl_params = params_dgrdl(dict_size = 6L, k_neighbours = 3L),
  .verbose = FALSE
```

```
)
head(get_modules(get_results(obj)))
```

`dialogue_sc` *Find multicellular programmes with DIALOGUE*

Description

DIALOGUE looks for programmes of cell-type-specific genes whose activity covaries across the samples that several cell types share. Where co-expression modules ask what varies together *within* a cell type, this asks what varies together *between* them, sample by sample.

Usage

```
dialogue_sc(
  object,
  cell_type_col,
  sample_col,
  features,
  quality_col = NULL,
  gene_ids = NULL,
  pmd_params = params_dialogue_pmd(),
  hlm_params = params_dialogue_hlm(),
  refine_params = params_dialogue_refine(),
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells`, `SingleCellsSubset` or `MetaCells` class.
`cell_type_col`	String. Column in the obs table holding the cell type labels.
`sample_col`	String. Column in the obs table holding the sample labels. The random intercept in stage two is over these. For `MetaCells` the meta cells must have been built *within* samples, otherwise the level is not well-defined.
`features`	Named list of numeric matrices, one per cell type. Names must match the levels in `cell_type_col`, row names must cover that cell type's cells, and each needs at least two columns. Rows are matched by name, not by position.
`quality_col`	Optional string. Column in the obs table to use as the cell quality covariate. If `NULL`, defaults to the z-scored log library size.
`gene_ids`	Optional character. Genes to consider when building signatures. If `NULL`, uses `get_hvg()` on the object.
`pmd_params`	List, see `params_dialogue_pmd()`.
`hlm_params`	List, see `params_dialogue_hlm()`.
`refine_params`	List, see `params_dialogue_refine()`.
`.verbose`	Boolean or integer. Verbosity.

Details

The algorithm works in three stages. Each cell type's features are collapsed to one row per sample and put through a sparse multi-CCA, giving every programme a weight vector per cell type plus a provisional gene signature. Then, for every ordered pair of cell types and every candidate gene, a mixed model asks whether a cell's own programme score tracks the partner's expression of that gene in the same sample. Finally the partners are meta-analysed and the scores refit onto the surviving genes by non-negative least squares.

`features` is mandatory and there is no default. DIALOGUE does not compute it: it is whatever low-dimensional description of each cell type you trust, and the method is only as good as that choice. Do *not* hand it a slice of a global PCA. Those components mostly carry between-cell-type identity, which is near-constant inside one cell type and gets dropped by the ANOVA filter, leaving the decomposition to work off whatever is left. Run a PCA per cell type instead, which for `SingleCells` means `SingleCellsSubset()` followed by `calculate_pca_sc()` on each subset.

The method is unforgiving about study design, and the failure modes are errors rather than bad answers. It needs at least two cell types, at least five samples present in *every* cell type, and enough cells per sample per cell type to clear `abn_c` in `params_dialogue_pmd()`. A handful of samples with thousands of cells each is the regime it was built for; many samples with a dozen cells each is not.

Value

A `DialogueResult` object.

References

Jerby-Arnon & Regev, Nature Biotechnology, 2022

Examples

```
# a planted multicellular programme recovered across three cell types
data <- generate_dialogue_test_data()
dir <- tempfile("bixverse_dlg")
dir.create(dir)
object <- load_r_data(
  SingleCells(dir_data = dir),
  counts = data$counts,
  obs = data$obs,
  var = data$var,
  sc_qc_param = params_sc_min_quality(
    min_unique_genes = 10L,
    min_lib_size = 50L,
    min_cells = 10L
  ),
  .verbose = FALSE
)
res <- dialogue_sc(
  object,
  cell_type_col = "cell_grp",
  sample_col = "sample_id",
```

```
  features = data$features,
  gene_ids = data$var$gene_id,
  pmd_params = params_dialogue_pmd(k = 2L, n_permutations = 20L),
  .verbose = FALSE
)
res

unlink(dir, recursive = TRUE, force = TRUE)
```

`diffcor_module_processing`
Prepare differential correlation-based module detection

Description

This function will calculate the differential correlation between the stored data set in the class and another background data set. To do so, it uses a Fisher transformation of the correlation coefficients and calculates a Z score based on the delta. The function will automatically subset into shared features between the two data sets.

Usage

```
diffcor_module_processing(
  object,
  background_mat,
  cor_method = c("pearson", "spearman"),
  .verbose = TRUE
)
```

Arguments

`object`	The class, see `BulkCoExp()`. Ideally, you should run `preprocess_bulk_coexp()` before applying this function.
`background_mat`	Numerical matrix. The background data set.
`cor_method`	String. Option of `c("pearson", "spearman")`.
`.verbose`	Boolean. Controls verbosity of the function.

Value

The class with added data to the properties for subsequent usage.

Examples

```
# differential correlation of two sample groups of the same matrix
sig <- synthetic_signal_matrix()
mat <- t(sig$mat)
target <- mat[sig$group %in% c("group1", "group2"), ]
background <- mat[sig$group == "group3", ]
meta <- data.table::data.table(sample_id = rownames(target))
obj <- BulkCoExp(target, meta)
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- diffcor_module_processing(
  obj, background, cor_method = "pearson", .verbose = FALSE
)
obj@params$correlation_params$no_intersecting_features
```

`diffuse_seed_nodes` *Diffuse seed genes over a network*

Description

This function takes a diffusion vector and leverages personalised page-rank diffusion to identify influential nodes. These can be used subsequently for community detection or check AUROC values given a set of genes.

Usage

```
diffuse_seed_nodes(
  object,
  diffusion_vector,
  summarisation = c("max", "mean", "harmonic_sum")
)
```

Arguments

`object`	`NetworkDiffusions` object. The underlying class `NetworkDiffusions()`.
`diffusion_vector`	Named nuermic. A named vector with values to use for the reset parameter in the personalised page-rank diffusion. Names should represent node names of the graph.
`summarisation`	String. If there are duplicated names in the `diffusion_vector` how to summarise the scores.

Value

The class with added diffusion score based on a single set of seed genes. Additionally, the seed genes are stored in the class.

Examples

```
# personalised page-rank diffusion from three seed nodes
set.seed(42)
g <- igraph::sample_pa(15, directed = FALSE)
edges <- data.table::setDT(igraph::as_data_frame(g))[, `:=`(
  from = sprintf("node_%i", from),
  to = sprintf("node_%i", to)
)]
object <- NetworkDiffusions(edges, weighted = FALSE, directed = FALSE)
object <- diffuse_seed_nodes(
  object,
  c(node_1 = 1, node_3 = 1, node_10 = 1),
  summarisation = "max"
)
head(get_diffusion_vector(object))
```

---

doublet_detection_boost_sc

Doublet detection with boosted doublet classification

Description

This function implements the boosted doublet detection. It generates through several iterations simulated doublets, generate kNN graphs, runs Louvain clustering and assesses how often an observed cells clsuters together with the simulated doublets.

Usage

```
doublet_detection_boost_sc(
  object,
  boost_params = params_boost(),
  cells_to_use = NULL,
  group_by = NULL,
  seed = 42L,
  streaming = NULL,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`boost_params`	A list with the final scrublet parameters, see `params_boost()` for full details.
`cells_to_use`	Optional string. Names of the cells to use for the run of the boosted doublet detection. Useful when you wish to run doublet detection on individual batches within your data. The object returned will be specifically using these cells.

group_by	Optional grouping variable. Useful if you want to run the method on a per-sample basis.
seed	Integer. Random seed.
streaming	Optional Boolean. Shall the data be streamed in. Useful for larger data sets where you wish to avoid loading in the whole data. If NULL, will automatically detect.
.verbose	Boolean or integer. Controls verbosity and returns run times. FALSE -> quiet, TRUE or 1L -> normal verbosity, 2L -> detailed verbosity.

Value

A boost_res class that has with the following items:

- predicted_doublets - Boolean vector indicating which observed cells predicted as doublets (TRUE = doublet, FALSE = singlet).
- doublet_scores_obs - Numerical vector with the likelihood of being a doublet for the observed cells.
- voting_avg - Numerical vector with the average voting score.

Examples

```
# boosted doublet detection over five iterations
sc <- demo_single_cells(prepped = FALSE)
doublet_detection_boost_sc(
  sc,
  boost_params = params_boost(
    hvg = list(min_gene_var_pctl = 0.0),
    pca = list(no_pcs = 10L),
    n_iters = 5L
  ),
  .verbose = FALSE
)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

download_cd34_data *Download the CD34 example data from SEACells*

Description

This function downloads the CD34 data set from the SEACells paper into the temperorary directory.

Usage

```
download_cd34_data(quiet = FALSE)
```

Arguments

quiet	Boolean. If the download shall be quiet.

Value

String. The path to CD34 SEACells data set.

References

Persad, et al., Nat. Biotechnol., 2023

Examples

```
## Not run:
# pulls the archive into the session tempdir()
path <- download_cd34_data()
get_h5ad_dimensions(path)$dims

## End(Not run)
```

---

download_cistarget_hg38

Download CisTarget reference files for human (hg38)

Description

Download CisTarget reference files for human (hg38)

Usage

```
download_cistarget_hg38(
  cache_dir = tools::R_user_dir("bixverse", which = "cache"),
  overwrite = FALSE
)
```

Arguments

cache_dir	String. Directory to cache the files. Defaults to a package-specific user cache directory.
overwrite	Logical. Re-download even if files already exist.

Value

Named list with paths: rankings and motif_annotations.

Examples

```
## Not run:
# fetch the hg38 rankings and motif annotations (large, network)
paths <- download_cistarget_hg38()
paths$rankings

## End(Not run)
```

`download_demuxlet_pbmc` *Download PBMCs with demuxlet doublet information*

Description

This function downloads a PBMC data set with demuxlet information to test doublet detection methods.

Usage

```
download_demuxlet_pbmc(quiet = FALSE)
```

Arguments

`quiet`	Boolean. If the download shall be quiet.

Value

String. The path to the extracted doublet detection data.

Examples

```
## Not run:
# pulls the archive into the session tempdir()
path <- download_demuxlet_pbmc()
list.files(path)

## End(Not run)
```

`download_dialogue_uc` *Download the ulcerative colitis example data for DIALOGUE*

Description

This function downloads the ulcerative colitis subset used by the DIALOGUE paper into the temporary directory. 5,374 cells across five cell subtypes and 30 donors, with the inflammation status per biopsy. Enough samples per cell type for `dialogue_sc()`, which most single sample example data sets are not.

Usage

```
download_dialogue_uc(quiet = FALSE)
```

Arguments

`quiet`	Boolean. If the download shall be quiet.

Details

The published matrix holds `log2(TPM / 10 + 1)`, which is what Smillie, et al. released; the raw counts were never published. The file served here has been linearised back to the TPM/10 scale and rounded, so it loads through `load_h5ad()` like any count matrix and gets bixverse's own log CPM on the way in. See `data-raw/dialogue_uc.R` for the preparation.

Value

String. The path to the ulcerative colitis h5ad file.

References

Smillie, et al., Cell, 2019; Jerby-Arnon and Regev, Nat. Biotechnol., 2022

Examples

```
## Not run:
# pulls the archive into the session tempdir()
path <- download_dialogue_uc()
get_h5ad_dimensions(path)$dims

## End(Not run)
```

`download_kang_pbmc` *Download the Kang, et al. IFN-beta stimulated PBMC data*

Description

Downloads the Kang, et al. demultiplexed PBMC experiment as a `SingleCellExperiment`. 8 lupus donors, PBMCs, control against IFN-beta stimulated in vitro, roughly 29k cells before quality control.

The design is paired within donor, which is what makes it the example for `nebula_sc()`: the donor is a genuine random effect and every donor contributes to both arms. Load it with `load_sce()`.

Usage

```
download_kang_pbmc(quiet = FALSE)
```

Arguments

`quiet`	Boolean. If the download shall be quiet.

Details

Sourced from the `muscData` Bioconductor package, itself from GEO accession `GSE96583`. See `data-raw/zenodo_sce_datasets.R` for the preparation. The `multiplets` column marks doublets and ambiguous droplets, which are worth dropping before any modelling.

Served from the `bixverse-data` GitHub release. There is no Zenodo fallback for this one yet.

Value

String. The path to the qs2 file holding the `SingleCellExperiment`.

References

Kang, et al., Nat. Biotechnol., 2018

Examples

```
## Not run:
# pulls the archive into the session tempdir()
path <- download_kang_pbmc()
sce <- qs2::qs_read(path)

## End(Not run)
```

download_marrow_cd34 *Download the marrow CD34 example data from Palantir*

Description

This function downloads the bone marrow CD34 data set from the Palantir paper into the temporary directory.

Usage

```
download_marrow_cd34(quiet = FALSE)
```

Arguments

quiet	Boolean. If the download shall be quiet.

Value

String. The path to the marrow CD34 data set.

References

Setty, et al., Nat. Biotechnol., 2019

Examples

```
## Not run:
# pulls the archive into the session tempdir()
path <- download_marrow_cd34()
get_h5ad_dimensions(path)$dims

## End(Not run)
```

download_pbmc_1k_5p *Download the raw PBMC 1k 5' matrix from 10x Genomics*

Description

Downloads the unfiltered `raw_feature_bc_matrix.h5` of the 10x Genomics 5' PBMC 1k run, straight from the 10x CDN rather than the bixverse-data mirror. All 737,280 barcodes are in there, empty droplets included, which is what `cellsweep_sc()` needs. The file also carries 19 Antibody Capture features next to the 36,601 genes, so load it with `load_tenx_h5(feature_type = "Gene Expression")`.

Usage

```
download_pbmc_1k_5p(quiet = FALSE)
```

Arguments

quiet	Boolean. If the download shall be quiet.

Value

String. The path to the downloaded h5 file.

Examples

```
## Not run:
# pulls roughly 17MB into the session tempdir()
h5_path <- download_pbmc_1k_5p()
read_tenx_h5_metadata(h5_path)$dims

## End(Not run)
```

---

download_pbmc_batches

Download two different PBMC data sets for batch correction testing

---

Description

This function downloads two different h5ad files for testing batch correction methods into the temporary directory.

Usage

```
download_pbmc_batches(quiet = FALSE)
```

Arguments

quiet	Boolean. If the download shall be quiet.

Value

String. The path to the directory with the PBMC h5ad files.

Examples

```
## Not run:
# pulls the archive into the session tempdir()
path <- download_pbmc_batches()
list.files(path)

## End(Not run)
```

download_pbmc_totalseq_data *Download the PBMC TotalSeq data with ADT counts*

Description

This function downloads the PBMC TotalSeq data set.

Usage

```
download_pbmc_totalseq_data(quiet = FALSE)
```

Arguments

quiet	Boolean. If the download shall be quiet.

Value

String. The path to the TotalSeq data.

Examples

```
## Not run:
# pulls the archive into the session tempdir()
path <- download_pbmc_totalseq_data()
read_tenx_h5_metadata(path)$feature_types

## End(Not run)
```

download_pbmc3k *Download PBMC3K data from Zenodo*

Description

This function downloads the PBMC3K dataset from 10x Genomics (uploaded on Zenodo) and extracts it and returns the (temporary) paths.

Usage

```
download_pbmc3k(quiet = FALSE)
```

Arguments

quiet	Boolean. If the download shall be quiet.

Value

String. The path to the extracted PBMC3K data.

Examples

```
## Not run:
# pulls the archive into the session tempdir()
path <- download_pbmc3k()
list.files(path)

## End(Not run)
```

`download_pbmc8k` *Download PBMC8K data from Zenodo*

Description

This function downloads the PBMC8k dataset from 10x Genomics (uploaded on Zenodo) and extracts it and returns the (temporary) paths.

Usage

```
download_pbmc8k(quiet = FALSE)
```

Arguments

`quiet`	Boolean. If the download shall be quiet.

Value

String. The path to the extracted PBMC8K data.

Examples

```
## Not run:
# pulls the archive into the session tempdir()
path <- download_pbmc8k()
list.files(path)

## End(Not run)
```

`download_thymus_ageing` *Download the Baran-Gale, et al. ageing thymus data*

Description

Downloads the mouse ageing thymus droplet experiment as a `SingleCellExperiment`. Roughly 69k thymic epithelial cells across three ages, with real change in cell type proportions between them.

That compositional change is what makes it the example for `get_miloR_abundances_sc()` and `meld_sc()`. A stimulation experiment moves cells in embedding space without moving the proportions, so almost every neighbourhood comes out significant and the result says nothing. Load it with `load_sce()`.

Usage

```
download_thymus_ageing(quiet = FALSE)
```

Arguments

`quiet`	Boolean. If the download shall be quiet.

Details

Sourced from the `MouseThymusAgeing` Bioconductor package. See `data-raw/zenodo_sce_datasets.R` for the preparation. The object carries no gene names, so `load_sce()` falls back to the Ensembl identifiers in the `rowData`, and 336 genes with no annotation at all get a generated identifier.

Served from the `bixverse-data` GitHub release. There is no Zenodo fallback for this one yet. At 430 MB it is the largest of the example datasets, so expect the first call to take a while.

Value

String. The path to the qs2 file holding the `SingleCellExperiment`.

References

Baran-Gale, et al., Development, 2020

Examples

```
## Not run:
# pulls the archive into the session tempdir()
path <- download_thymus_ageing()
sce <- qs2::qs_read(path)

## End(Not run)
```

`drop_cols_sc` *Drop columns from the obs or var table*

Description

Drops the named columns from the obs or var table of single cell-related classes. Protected identifier and bookkeeping columns (`cell_idx`, `cell_id`, `to_keep` for obs; `gene_idx`, `gene_id` for var) are refused with a warning. Columns that do not exist trigger a warning and are skipped.

Usage

```
drop_cols_sc(object, table = c("obs", "var"), cols)
```

Arguments

`object`	`SingleCells` (or other compatible) class.
`table`	String. One of `c("obs", "var")`.
`cols`	Character vector. Column names to drop.

Value

Invisible self.

Examples

```
# drop a column that is no longer needed
sc <- demo_single_cells(prepped = FALSE)
sc <- drop_cols_sc(sc, table = "obs", cols = "batch_index")
names(get_sc_obs(sc))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`extract_binary_heatmap_data`
Extract plot-ready data for a binary heatmap

Description

Filters, orders and optionally bins a logical samples x features matrix, e.g. the regulon on/off calls from `binarise_regulon_activity()`, so it can be drawn as a single raster with `bixverse.plots::plot_binary_heatmap()`.

Features are clustered within their group on the Jaccard distance, so shared absences do not count as similarity. Samples are clustered within their group on the Hamming distance. Groups larger than `max_cluster_n` are instead ordered by barycentre, the mean

plot position of the features a sample has on, which avoids the quadratic distance matrix. With more samples than `max_cols`, consecutive samples within a group are collapsed into roughly `max_cols` bins that hold the fraction of samples on. Every group keeps at least one bin.

Usage

```
extract_binary_heatmap_data(
  binary_mat,
  sample_groups = NULL,
  feature_groups = NULL,
  heatmap_params = params_binary_heatmap(),
  .verbose = TRUE
)
```

Arguments

`binary_mat`	Logical matrix of samples x features with unique row and column names.
`sample_groups`	Optional named character vector or factor mapping every row name of `binary_mat` to a group. Factor levels set the group order, otherwise groups are sorted.
`feature_groups`	Optional named character vector or factor mapping every column name of `binary_mat` to a group. Factor levels set the group order, otherwise groups are sorted.
`heatmap_params`	List. Output of `params_binary_heatmap()`.
`.verbose`	Boolean. Controls verbosity of the function.

Value

An object of class `BinaryHeatmapData`, a list with:

- mat - Numeric matrix of features x columns in plot order. Values are 0/1, or the fraction of samples on if binned.
- col_annot - data.table with one row per column: `col_idx`, `group` (factor) and `n_samples` in that column.
- row_annot - data.table with one row per feature: `row_idx`, `feature` and `group` (factor).
- binned - Boolean. Whether the samples were binned.

Without groups, `group` is a single level `"all"`.

References

Aibar, et al., Nat Methods, 2017

Examples

```
set.seed(7L)
binary_mat <- matrix(runif(200 * 30) > 0.7, nrow = 200)
dimnames(binary_mat) <- list(
  sprintf("cell_%i", 1:200),
  sprintf("regulon_%i", 1:30)
)
sample_groups <- setNames(rep(c("a", "b"), each = 100), rownames(binary_mat))
res <- extract_binary_heatmap_data(
  binary_mat,
  sample_groups = sample_groups,
  .verbose = FALSE
)
dim(res$mat)
```

---

extract_dot_plot_data

Extract grouped gene statistics for dot plots

Description

Extracts per-group mean expression and percentage of expressing cells for a set of genes. Returns a long-format data.table suitable for dot plots.

Usage

```
extract_dot_plot_data(
  object,
  features,
  grouping_variable,
  scale_exp = TRUE,
  modality = c("rna", "adt")
)
```

Arguments

`object`	A single cell class.
`features`	Character vector. Gene IDs to extract.
`grouping_variable`	String. Column name in the obs table to group by.
`scale_exp`	Boolean. Whether to min-max scale mean expression per gene.
`modality`	String. One of `c("rna", "adt")`. ADT is only available for `SingleCellsMultiModal`.

Value

A data.table with columns: gene, group, mean_exp, scaled_exp and pct_exp.

Examples

```
# mean expression and expressing fraction per cell group
sc <- demo_single_cells()
dt <- extract_dot_plot_data(
  sc,
  features = get_gene_names(sc)[1:5],
  grouping_variable = "cell_grp"
)
head(dt, 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

extract_embedding_data

Extract embedding coordinates for plotting

Description

Pulls an embedding into a long data.table with standardised coordinate columns (`dim_1`, `dim_2`, ...) and, optionally, observation metadata. The embedding name is stored as an `embedding` attribute for axis labelling.

Usage

```
extract_embedding_data(object, embedding, obs_cols = NULL, ...)
```

Arguments

`object`	A single cell class.
`embedding`	String. Name of the embedding (e.g. `"umap"`, `"pca"`).
`obs_cols`	Optional character vector. Obs columns to attach.
`...`	Additional arguments forwarded to `get_embedding()` (e.g. `modality`).

Value

A data.table with `cell_id`, `dim_*` columns and any requested obs columns.

Examples

```
# PCA coordinates with a cell annotation riding along
sc <- demo_single_cells()
dt <- extract_embedding_data(sc, "pca", obs_cols = "cell_grp")
head(dt[, c("cell_id", "dim_1", "dim_2", "cell_grp")])

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`extract_feature_pair` *Extract a pair of features for scatter / hex plots*

Description

Extracts two features into a wide data.table with `feature_1` and `feature_2` value columns, ready for a scatter or hex plot. Each feature may carry a `_rna` or `_adt` suffix to choose its modality independently (e.g. `"ENSG00000167286_rna"` against `"CD3_adt"`); features without a suffix fall back to `modality`. For `SingleCells` / `MetaCells` only RNA exists, so an `_adt` feature there errors via `extract_gene_expression()`.

Usage

```
extract_feature_pair(
  object,
  feature_1,
  feature_2,
  obs_cols = NULL,
  scale = FALSE,
  clip = NULL,
  modality = c("rna", "adt"),
  layer = c("norm", "magic")
)
```

Arguments

`object`	A single cell class.
`feature_1`	String. First feature, optionally `_rna` / `_adt` suffixed.
`feature_2`	String. Second feature, optionally `_rna` / `_adt` suffixed.
`obs_cols`	Optional character vector. Obs columns to attach (e.g. to colour the scatter).
`scale`	Boolean. Whether to z-score the expression values per feature.
`clip`	Optional numeric. Clip z-scores if `scale = TRUE`.
`modality`	String. Fallback modality for unsuffixed features. One of `c("rna", "adt")`.
`layer`	String. One of `c("norm", "magic")`, forwarded to `extract_gene_expression()`. Applies to both features, and an `_adt` suffixed one will error under `"magic"`.

Value

A data.table with `cell_id`, `feature_1`, `feature_2` and any requested obs columns. The original feature labels are stored in a `features` attribute as `c(feature_1, feature_2)`.

Examples

```
# two genes side by side, ready for a scatter
sc <- demo_single_cells(prepped = FALSE)
genes <- get_gene_names(sc)[1:2]
head(extract_feature_pair(sc, genes[1], genes[2], obs_cols = "cell_grp"))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

extract_feature_plot_data

Extract per-cell expression mapped onto an embedding

Description

Combines `extract_gene_expression()` with `extract_embedding_data()` and melts to long format, ready for faceted feature plots. The expression source and the embedding source are chosen independently via `expr_modality` and `embd_modality`, so you can colour an embedding from one modality by expression from another (e.g. RNA expression on an ADT-derived UMAP, or either modality on a WNN embedding). All sources key on the same kept-cell barcodes, so the merge stays aligned regardless of the chosen combination.

Usage

```
extract_feature_plot_data(
  object,
  features,
  embedding,
  scale = FALSE,
  clip = NULL,
  obs_col = NULL,
  expr_modality = c("rna", "adt"),
  embd_modality = c("rna", "adt", "wnn"),
  layer = c("norm", "magic"),
  ...
)
```

Arguments

`object`	A single cell class.
`features`	Character vector. Gene/feature IDs to extract, taken from `expr_modality`.
`embedding`	String. Name of the embedding.
`scale`	Boolean. Whether to z-score the expression values.
`clip`	Optional numeric. Clip z-scores if `scale = TRUE`.
`obs_col`	Optional character vector. Obs columns to attach.
`expr_modality`	String. Modality the expression is pulled from. One of `c("rna", "adt")`.

`embd_modality` String. Modality the embedding is pulled from. One of `c("rna", "adt", "wnn")`. Use `"wnn"` for WNN-derived embeddings.

`layer` String. One of `c("norm", "magic")`, forwarded to `extract_gene_expression()`. Use `"magic"` to colour the embedding by the imputed layer `run_magic_sc()` wrote.

`...` Additional arguments forwarded to `extract_embedding_data()` and onward to `get_embedding()`. Do not pass `modality` here; the embedding modality is set via `embd_modality` and passing it again will error.

Value

A long data.table with `cell_id`, `dim_*`, `gene` and `expression`.

Examples

```
# two genes melted onto the PCA coordinates
sc <- demo_single_cells()
dt <- extract_feature_plot_data(
  sc,
  features = get_gene_names(sc)[1:2],
  embedding = "pca"
)
head(dt[, c("cell_id", "dim_1", "dim_2", "gene", "expression")])

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

extract_gene_expression

Extract normalised gene expression for plotting

Description

Extracts dense normalised (log1p) expression values for a set of genes, optionally with additional observation metadata columns.

Usage

```
extract_gene_expression(
  object,
  features,
  obs_cols = NULL,
  scale = FALSE,
  clip = NULL,
  modality = c("rna", "adt"),
  layer = c("norm", "magic")
)
```

Arguments

`object`	A single cell class.
`features`	Character vector. Gene IDs to extract.
`obs_cols`	Optional character vector. Column names from the obs table to include.
`scale`	Boolean. Whether to z-score the expression values.
`clip`	Optional numeric. If `scale = TRUE`, clip z-scores to `[-clip, clip]`.
`modality`	String. One of `c("rna", "adt")`. ADT is only available for `SingleCellsMultiModal`.
`layer`	String. One of `c("norm", "magic")`. With `"magic"` the values come from the imputed layer `run_magic_sc()` wrote, which only holds the genes it was asked for. Imputation inflates gene-gene correlation, so this is for looking at things, not for measuring them. Note that `extract_dot_plot_data()` deliberately has no such argument: group means of imputed values are exactly the quantity MAGIC manufactures.

Value

A data.table with a `cell_id` column, one column per gene, and any requested obs columns.

Examples

```
# normalised expression of three genes with a cell annotation
sc <- demo_single_cells()
dt <- extract_gene_expression(
  sc,
  features = get_gene_names(sc)[1:3],
  obs_cols = "cell_grp"
)
head(dt, 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

extract_gene_violin_data

Extract per-cell expression grouped for violin plots

Description

Combines `extract_gene_expression()` with a grouping obs column and melts to long format, ready for stacked (one gene per row) violin plots.

Usage

```
extract_gene_violin_data(
  object,
  features,
  grouping_variable,
```

```
  scale = FALSE,
  clip = NULL,
  modality = c("rna", "adt"),
  layer = c("norm", "magic")
)
```

Arguments

`object`	A single cell class.
`features`	Character vector. Gene IDs to extract.
`grouping_variable`	String. Obs column to group by.
`scale`	Boolean. Whether to z-score the expression values.
`clip`	Optional numeric. Clip z-scores if `scale = TRUE`.
`modality`	String. One of `c("rna", "adt")`.
`layer`	String. One of `c("norm", "magic")`, forwarded to `extract_gene_expression()`.

Value

A long data.table with `cell_id`, `group`, `gene` and `expression`. `gene` is an ordered factor following `features`.

Examples

```
# long format expression grouped by cell type
sc <- demo_single_cells(prepped = FALSE)
head(extract_gene_violin_data(
  sc,
  features = get_gene_names(sc)[1:2],
  grouping_variable = "cell_grp"
))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

extract_milo_plot_data

Extract the miloR neighbourhood graph positioned on an embedding

---

Description

Places every tested neighbourhood of a `get_miloR_abundances_sc()` result at the embedding coordinate of its index cell, as Milo's `plotNhoodGraphDA()` does, and returns the neighbourhood graph as node and edge tables. Two neighbourhoods are connected by the number of cells they share.

The logFC is returned as is. Masking non-significant neighbourhoods is left to the plot, `is_sig` carries the call.

Usage

```
extract_milo_plot_data(
  object,
  milo_res,
  embedding = "umap",
  alpha = 0.1,
  overlap = 1L,
  ...
)
```

Arguments

`object`	A single cell class. The one the miloR result was built on.
`milo_res`	`miloR` class. Needs to have been through `test_nhoods()`.
`embedding`	String. Name of the embedding to position the nodes in.
`alpha`	Numeric. Spatial FDR threshold for `is_sig`. Defaults to `0.1`.
`overlap`	Integer. Minimum number of shared cells for an edge. Defaults to `1L`.
`...`	Additional arguments forwarded to `extract_embedding_data()` and onward to `get_embedding()` (e.g. `modality`).

Value

A list with the embedding stored as an `embedding` attribute and

- nodes - data.table with `Nhood`, `dim_1`, `dim_2`, `size` (cells in the neighbourhood), every column of the differential abundance results and `is_sig`.
- edges - data.table with `from`, `to`, `weight` (shared cells) and the segment coordinates `x`, `y`, `xend`, `yend`.

References

Dann, et al., Nat Biotechnol, 2022

---

extract_paga_plot_data

Extract the PAGA graph positioned on an embedding

Description

Puts every cluster of a `run_paga_sc()` result at the centroid of its cells in an embedding and returns the abstracted graph as node and edge tables. A node-link plot drawn from these sits where the reader already knows the biology is, which a free layout of the abstracted graph does not.

The abstracted graph is close to complete on real data, so `threshold` is doing real work. Drop it to zero and you get a hairball. `tree_only` is the shortcut to the backbone.

Usage

```
extract_paga_plot_data(
  object,
  paga_res,
  embedding = "umap",
  cluster_col = NULL,
  threshold = 0.01,
  tree_only = FALSE,
  centroid = c("median", "mean"),
  node_stat_col = NULL,
  ...
)
```

Arguments

`object`	A single cell class.
`paga_res`	`PagaRes` class. The output of `run_paga_sc()`, run on this object.
`embedding`	String. Name of the embedding to position the nodes in.
`cluster_col`	Optional string. The obs column holding the clustering. Defaults to the one the PAGA run used, and errors if a different one is given: nodes from one clustering with edges from another produce a plausible looking graph that is wrong.
`threshold`	Numeric. Edges below this connectivity are dropped. Defaults to `0.01`.
`tree_only`	Boolean. Use the maximum spanning forest rather than the full abstracted graph. Defaults to `FALSE`.
`centroid`	String. One of `c("median", "mean")`. How a cluster's position is summarised. Median by default, since embeddings throw stragglers that drag a mean off its cluster.
`node_stat_col`	Optional string. A numeric obs column to summarise per cluster with the same statistic, e.g. `"palantir_pseudotime"`, giving a `stat` column to colour the nodes by.
`...`	Additional arguments forwarded to `extract_embedding_data()` and onward to `get_embedding()` (e.g. `modality`).

Value

A list with the embedding stored as an `embedding` attribute and

- nodes - data.table with `cluster` (a factor in graph order), `dim_1`, `dim_2`, `n_cells` and, when `node_stat_col` is given, `stat`. Clusters holding no cells are dropped, as they have no position.
- edges - data.table with `from`, `to`, `weight` and the coordinates `x`, `y`, `xend`, `yend` of both ends, ready for a segment layer. Each edge appears once, despite the graph being stored symmetrically.

References

Wolf, et al., Genome Biol., 2019.

Examples

```
# the abstracted graph placed on the PCA coordinates
sc <- demo_single_cells()
sc <- find_clusters_sc(sc, res = 1.0)
paga <- run_paga_sc(sc, cluster_col = "leiden_clustering", .verbose = FALSE)
extract_paga_plot_data(sc, paga, embedding = "pca")$nodes

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

f1_score_confusion_mat

F1 scores on top of a confusion matrix

Description

Helper function to check for expected clustering vs actual clustering.

Usage

```
f1_score_confusion_mat(clusters_a, clusters_b)
```

Arguments

clusters_a	String or factor. The clustering of algorithm 1.
clusters_b	String or factor. The clustering of algorithm 2.

Value

Named vector with the F1 scores between the two clustering algorithms.

Examples

```
# agreement between two clusterings of the same six samples
clusters_a <- c("c1", "c1", "c2", "c2", "c3", "c3")
clusters_b <- c("x", "x", "y", "y", "z", "x")
f1_score_confusion_mat(clusters_a, clusters_b)
```

`fast_cluster_sc` *Run fast Louvain clustering on a SingleCells object*

Description

Runs k-means on the chosen embedding, builds a kNN graph on the centroids, applies Louvain clustering and propagates memberships back to the cells. Optionally runs a grid over multiple seeds and returns stability statistics.

Usage

```
fast_cluster_sc(
  object,
  embd_to_use = "pca",
  no_embd_to_use = NULL,
  resolutions = c(2, 1, 0.5),
  km_type = c("kmeans", "minibatch"),
  n_centroids = NULL,
  fc_params = params_sc_fast_cluster(),
  snn = TRUE,
  return_kmeans = FALSE,
  grid_search = FALSE,
  no_seeds = 10L,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`embd_to_use`	String. Embedding name. Defaults to `"pca"`.
`no_embd_to_use`	Optional integer. Number of dimensions to keep.
`resolutions`	Numeric vector. Louvain resolutions.
`km_type`	String. One of `c("kmeans", "minibatch")`. The former runs standard k-means, the latter a mini-batch version that can be useful for large data sets.
`n_centroids`	Optional integer. Number of k-means centroids. Defaults to `sqrt(n_cells)` Rust-side if `NULL`.
`fc_params`	List. Output of `params_sc_fast_cluster()`.
`snn`	Boolean. Convert kNN to sNN.
`return_kmeans`	Boolean. Return k-means assignments and centroids.
`grid_search`	Boolean. Run multi-seed grid version.
`no_seeds`	Integer. Number of additional seeds (only used when `grid_search = TRUE`).

seed	Integer. Reproducibility.
.verbose	Boolean or integer. Controls verbosity and returns run times. FALSE -> quiet, TRUE or 1L -> normal verbosity, 2L -> detailed verbosity.

Value

SingleCellFastClusters S3 object with:

memberships data.table with cell_idx and one column per resolution (res_<value>).

stats data.table of grid statistics, or NULL.

k_means_cluster Integer vector of k-means assignments, or NULL.

centroids Numeric matrix of centroids, or NULL.

resolutions Resolutions used.

with cell_indices stored as an attribute (0-indexed).

Examples

```
# k-means centroids, Louvain on top, memberships back to the cells
sc <- demo_single_cells()
res <- fast_cluster_sc(
  sc,
  resolutions = c(1.0, 0.5),
  n_centroids = 30L,
  .verbose = FALSE
)
head(get_data(res))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

fast_ica_rust *Fast ICA via Rust*

Description

This functions is a wrapper over the Rust implementation of fastICA and the generation of the pre-processed data and pre-whitening matrix. It has the same two options c("logcosh", "exp") to run ICA in parallel modus. You can control the parameters of ICA via ica_params.

Usage

```
fast_ica_rust(
  X,
  n_icas,
  ica_fun = c("logcosh", "exp"),
  ica_params = params_ica_general(),
  fast_svd = TRUE,
  seed = NULL
)
```

Arguments

`X`	Numeric matrix. The data on which you want to run fastICA.
`n_icas`	Integer. Number of independent components to recover.
`ica_fun`	String, element of `c("logcosh", "exp")`.
`ica_params`	List. The ICA parameters, see `params_ica_general()` wrapper function. This function generates a list containing: • maxit - Integer. Maximum number of iterations for ICA. • alpha - Float. The alpha parameter for the logcosh version of ICA. Should be between 1 to 2. • max_tol - Maximum tolerance of the algorithm. • verbose - Controls verbosity of the function.
`fast_svd`	Boolean. Shall the randomised SVD be used. This is faster, but less precise.
`seed`	Integer. Seed to ensure reproducible results.

Value

A list containing:

- w The mixing matrix w.
- A ICA results matrix A.
- S ICA results matrix S.
- converged Boolean indicating if algorithm converged.

Examples

```
# recover two mixed sources
sources <- cbind(sin((1:1000) / 20), rep(((1:200) - 100) / 100, 5))
mixed <- sources %*% matrix(c(0.291, 0.6557, -0.5439, 0.5572), 2, 2)
ica_res <- fast_ica_rust(mixed, n_icas = 2L, ica_fun = "logcosh", seed = 42L)
max(abs(cor(sources[, 1], t(ica_res$S))))
```

`fast_ica_rust_helper` *Fast ICA via Rust from processed data*

Description

This functions is a wrapper over the Rust implementation of fastICA expected already the pre-processed matrix and the pre-whitening matrix. It has the same two options `c("logcosh", "exp")` to run ICA in parallel modus. You can control the parameters of ICA via `ica_params`.

Usage

```
fast_ica_rust_helper(
  X,
  K,
  n_icas,
  ica_fun = c("logcosh", "exp"),
  seed = NULL,
  ica_params = params_ica_general()
)
```

Arguments

X	Numeric matrix. Processed data. Output of rs_prepare_whitening().
K	The K matrix. Pre-whitening matrix. Output of rs_prepare_whitening().
n_icas	Integer. Number of independent components to recover.
ica_fun	String, element of c("logcosh", "exp").
seed	Integer. Seed to ensure reproducible results.
ica_params	List. The ICA parameters, see params_ica_general() wrapper function. This function generates a list containing: • maxit - Integer. Maximum number of iterations for ICA. • alpha - Float. The alpha parameter for the logcosh version of ICA. Should be between 1 to 2. • max_tol - Maximum tolerance of the algorithm. • verbose - Controls verbosity of the function.

Value

A list containing:

- w The mixing matrix w.
- A ICA results matrix A.
- S ICA results matrix S.
- converged Boolean indicating if algorithm converged.

Examples

```
# same run, but with the whitening done up front
sources <- cbind(sin((1:1000) / 20), rep(((1:200) - 100) / 100, 5))
mixed <- sources %*% matrix(c(0.291, 0.6557, -0.5439, 0.5572), 2, 2)
whitened <- rs_prepare_whitening(
  x = mixed,
  fast_svd = TRUE,
  seed = 42L,
  rank = NULL,
  oversampling = NULL,
  n_power_iter = NULL
)
```

```
ica_res <- fast_ica_rust_helper(
  X = whitened$x,
  K = whitened$k,
  n_icas = 2L,
  ica_fun = "logcosh",
  seed = 42L
)
dim(ica_res$S)
```

`fast_mnn_sc` *Run fastMNN*

Description

This function implements the fast mutual nearest neighbour (MNN) from Haghverdi, et al. This version works on the PCA embedding and generates an embedding only and not a fully corrected count matrix. The function will iterate through the batches, identify the MNN and generate correction vectors and generate a corrected embedding which is added to the function.

Usage

```
fast_mnn_sc(
  object,
  batch_column,
  batch_hvg_genes,
  fastmnn_params = params_sc_fastmnn(),
  use_precomputed_pca = FALSE,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`batch_column`	String. The column with the batch information in the obs data of the class.
`batch_hvg_genes`	Integer vector. These are the highly variable genes, identified by a batch-aware method. Please refer to `find_hvg_batch_aware_sc()` for more details. These genes have to be 0-indexed!
`fastmnn_params`	A list, please see `params_sc_fastmnn()`. The list has the following parameters: • sigma - Numeric. Bandwidth of the Gaussian smoothing kernel (as proportion of space radius).

- cos_norm - Logical. Apply cosine normalisation before computing distances.
- var_adj - Logical. Apply variance adjustment to avoid kissing effects.
- no_pcs - Integer. Number of PCs to use for MNN calculations.
- knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.
- pca - List of PCA parameters, see `params_sc_pca()` for available parameters and their defaults.

`use_precomputed_pca`
: Boolean. Should the PCA in the object be used if found. If you decide to do this, make sure that you have run the PCA on the batch-aware HVG ideally.

`seed`
: Integer. Random seed.

`.verbose`
: Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

The object with the added fastMNN embeddings to the object.

References

Haghverdi, et al., Nat Biotechnol, 2018

Examples

```
# fastMNN over batch aware highly variable genes
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 600L, n_genes = 50L, n_batches = 3L
  )
)
hvg <- find_hvg_batch_aware_sc(
  sc, hvg_no = 20L, batch_column = "batch_index", .verbose = FALSE
)
sc <- fast_mnn_sc(
  sc,
  batch_column = "batch_index",
  batch_hvg_genes = hvg$hvg_gene_idx,
  fastmnn_params = params_sc_fastmnn(
    no_pcs = 10L,
    knn = list(k = 5L)
  ),
  .verbose = FALSE
)
dim(get_embedding(sc, "mnn"))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`fgsea_go_elim` *Run GO enrichment with elimination method over a continuous vectors*

Description

This method takes the GeneOntologyElim and a vector of gene level statistics to perform fgsea (multi-level) leveraging ontological information. It starts at the lowest levels of the ontology and tests if there is significant enrichment for any GO terms. If the threshold of the p-value is below the elimination threshold, the genes from this term will be removed from all its ancestors. The function then proceeds to the next level of the ontology and repeats the process. Subsequently, it leverages the multi-level method to estimate lower p-values for significant terms, see Korotkevich, et al.

Usage

```
fgsea_go_elim(
  object,
  stats,
  elim_threshold = 0.05,
  nperm = 2000L,
  gsea_params = params_gsea(max_size = 2000L),
  seed = 42L
)
```

Arguments

`object`	The underlying class, see `GeneOntologyElim()`.
`stats`	Named numeric vector. The gene level statistic.
`elim_threshold`	Float. Threshold from which p-value onwards the elimination on the ancestors shall be conducted.
`nperm`	Integer. Number of permutation tests. Defaults to `2000L`
`gsea_params`	List. The GSEA parameters, see `params_gsea()` wrapper function. This function generates a list containing: • min_size - Integer. Minimum size for the gene sets. • max_size - Integer. Maximum size for the gene sets. • gsea_param - Float. The GSEA parameter. Defaults to `1.0`. • sample_size - Integer. Number of samples to iterate through for the multi-level implementation of fgsea. • eps - Float. Boundary for calculating the p-value. Used for the multi-level implementation of fgsea.
`seed`	Random seed for reproducibility.

Value

data.table with enrichment results.

References

Korotkevich, et al., bioRxiv

Examples

```
# human GO terms with at least 25 genes
go_obj <- GeneOntologyElim(
  get_go_data_human(.verbose = FALSE),
  min_genes = 25L
)
genes <- unique(unlist(S7::prop(go_obj, "go_to_genes")))
set.seed(1L)
stats <- stats::setNames(rnorm(length(genes)), genes)
stats[1:200] <- stats[1:200] + 2
res <- fgsea_go_elim(go_obj, stats = stats, nperm = 1000L)
head(res, 3)
```

---

fgsea_simple_go_elim *Run GO enrichment with elimination with fgsea simple*

Description

This method takes the GeneOntologyElim and a vector of gene level statistics to perform fgsea (simple) leveraging ontological information. It starts at the lowest levels of the ontology and tests if there is significant enrichment for any GO terms. If the threshold of the p-value is below the elimination threshold, the genes from this term will be removed from all its ancestors. The function then proceeds to the next level of the ontology and repeats the process.

Usage

```
fgsea_simple_go_elim(
  object,
  stats,
  elim_threshold = 0.05,
  nperm = 2000L,
  gsea_params = params_gsea(max_size = 2000L),
  seed = 42L
)
```

Arguments

object	The underlying class, see GeneOntologyElim().
stats	Named numeric vector. The gene level statistic.
elim_threshold	Float. Threshold from which p-value onwards the elimination on the ancestors shall be conducted.
nperm	Integer. Number of permutation tests. Defaults to 2000L
gsea_params	List. The GSEA parameters, see params_gsea() wrapper function. This function generates a list containing: • min_size - Integer. Minimum size for the gene sets. • max_size - Integer. Maximum size for the gene sets. • gsea_param - Float. The GSEA parameter. Defaults to 1.0. • sample_size - Integer. Number of samples to iterate through for the multi-level implementation of fgsea. • eps - Float. Boundary for calculating the p-value. Used for the multi-level implementation of fgsea.
seed	Random seed for reproducibility.

Value

data.table with enrichment results.

Examples

```
# human GO terms with at least 25 genes
go_obj <- GeneOntologyElim(
  get_go_data_human(.verbose = FALSE),
  min_genes = 25L
)
genes <- unique(unlist(S7::prop(go_obj, "go_to_genes")))
set.seed(1L)
stats <- stats::setNames(rnorm(length(genes)), genes)
stats[1:200] <- stats[1:200] + 2
res <- fgsea_simple_go_elim(go_obj, stats = stats, nperm = 1000L)
head(data.table::setorder(res, pvals), 3)
```

---

filter_similarities *Filter the calculated similarities*

Description

This function calculates the critical value, see calculate_critical_value() and filters subsequently all the term pairs to the ones with a value critical value.

Usage

```
filter_similarities(
  object,
  alpha,
  permutations = 100000L,
  seed = 10101L,
  .verbose = TRUE
)
```

Arguments

`object`	`OntologySim` class. See `OntologySim()`.
`alpha`	Float. The alpha value. For example, 0.001 would mean that the critical value is smaller than 0.1 percentile of the random permutations.
`permutations`	Number of random permutations.
`seed`	Integer. For reproducibility purposes.
`.verbose`	Boolean. Controls verbosity of the function.

Value

The class with filtered results added to the respective slot.

Examples

```
# keep only term pairs above the permutation-derived critical value
onto <- data.table::data.table(
  parent = c("a", "b", "b", "b", "c"),
  child = c("b", "c", "d", "e", "f"),
  type = c("part_of", "part_of", "part_of", "is_a", "is_a")
)
onto_obj <- OntologySim(onto, .verbose = FALSE)
onto_obj <- calculate_wang_sim_onto(
  onto_obj,
  weights = c(part_of = 0.8, is_a = 0.6),
  .verbose = FALSE
)
onto_obj <- filter_similarities(onto_obj, alpha = 0.1, .verbose = FALSE)
head(get_results(onto_obj))
```

`find_all_markers_sc` *Find all markers*

Description

This function can be used to run differential gene expression for every group of an unsupervised clustering method for example. You specify a column and the function calculates differential gene expression of the first cluster vs. everything else, second cluster vs. everything else, etc. All groups are tested in a single pass over the genes, so memory stays bounded however many cells there are.

Usage

```
find_all_markers_sc(
  object,
  column_of_interest,
  method = "wilcox",
  alternative = c("greater", "less", "twosided"),
  min_prop = 0.05,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`column_of_interest`	String. The column you wish to use to identify the markers between all combination. Needs to be in the obs table
`method`	String. Which method to use for the calculations of the DGE. At the moment the only option is `"wilcox"`, but the parameter is reserved for future features.
`alternative`	String. Test alternative. One of `c("twosided", "greater", "less")`. This function will default to `"greater"`, i.e., genes upregulated in the group.
`min_prop`	Numeric. The minimum proportion of cells that need to express the gene to be tested in the group or in the rest.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

data.table with the DGE results from the test.

Examples

```
# one versus rest across the Leiden clusters
sc <- demo_single_cells()
sc <- find_clusters_sc(sc, res = 1.0)
res <- find_all_markers_sc(
  sc,
  column_of_interest = "leiden_clustering",
  .verbose = FALSE
)
head(res)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`find_clusters_sc` *Graph-based clustering of cells on the sNN graph*

Description

This function will apply Leiden clustering on the sNN graph with the given resolution and add a column to the obs table. Runs in Rust via `leiden_clustering()`; the labels depend on `seed` only, `0` is the largest cluster.

Usage

```
find_clusters_sc(
  object,
  cluster_algorithm = c("leiden", "louvain"),
  res = 1,
  name = "leiden_clustering",
  modality = c("rna", "adt", "wnn"),
  seed = 42L
)
```

Arguments

`object`	`SingleCells`, `MetaCells` (or potentially other) class.
`cluster_algorithm`	String. One of `c("leiden", "louvain")`.
`res`	Numeric. The modularity resolution parameter, see `leiden_clustering()`.
`name`	String. The name to add to the obs table in the DuckDB.
`modality`	String. On which modality to run the UMAP. One of `c("rna", "adt", "wnn")`. The two latter options are only available for multi-modal versions with the added data.
`seed`	Integer. For reproducibility.

Value

The object with added clustering in the obs table.

Examples

```
# Leiden on the cached sNN graph
sc <- demo_single_cells()
sc <- find_clusters_sc(sc, res = 1.0, name = "clusters")
table(get_sc_obs(sc)$clusters)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`find_hvg_batch_aware_sc` *Identify HVGs (batch aware)*

Description

This is a helper function to identify highly variable genes in a batch-aware manner. At the moment the implementation has only the VST-based version (known as Seurat v3). The other methods will be implemented in the future. This function will calculate the HVG per given experimental batch and you can choose the way to combine them. The choices are union (of Top x HVG per batch), based on the average variance per batch or only take genes that are amongst the Top X HVG in all batches. Important. The function returns 0-indices for the genes!

Usage

```
find_hvg_batch_aware_sc(
  object,
  batch_column,
  hvg_no = 2000L,
  gene_comb_method = c("union", "average", "intersection"),
  hvg_params = params_sc_hvg(),
  streaming = NULL,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`batch_column`	String. The column name of the batch column in the obs table.
`hvg_no`	Integer. Number of highly variable genes to include. Defaults to `2000L`.
`gene_comb_method`	String. One of `c("union", "average", "intersection")`. The method to combine the HVG across the different batches. Defaults to `"union"`.
`hvg_params`	List, see `params_sc_hvg()`. This list contains • method - Which method to use. One of `c("vst", "meanvarbin", "dispersion", "scran")` • loess_span - The span for the loess function to standardise the variance (`"vst"`), or of the lowess trend (`"scran"`) • num_bin - Integer. Not yet implemented. • bin_method - String. One of `c("equal_width", "equal_freq")`. Not implemented yet. • mean_filter, min_mean, transform, use_min_width, min_width, min_window_count - The `"scran"` trend parameters, see `params_hvg_scran_defaults()`

streaming	Optional Boolean. Shall the data be streamed in. Useful for larger data sets where you wish to avoid loading in the whole data. If NULL, will automatically detect.
.verbose	Boolean or integer. Controls verbosity and returns run times. FALSE -> quiet, TRUE or 1L -> normal verbosity, 2L -> detailed verbosity.

Value

This function will return a list with:

- hvg_genes - The gene names of the HVGs.
- hvg_gene_idx - The (0-index) gene features.
- hvg_data - data.table with the detailed information of the variance per batch.

Examples

```
# highly variable genes taken as the union over the batches
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 600L, n_genes = 50L, n_batches = 3L
  )
)
hvg <- find_hvg_batch_aware_sc(
  sc,
  hvg_no = 20L,
  batch_column = "batch_index",
  gene_comb_method = "union",
  .verbose = FALSE
)
head(hvg$hvg_genes)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

find_hvg_sc *Identify HVGs*

Description

This is a helper function to identify highly variable genes for SingleCells (using the Rust-based streaming of data) or MetaCells.

Small cell selections (up to 10% of the stored cells and at most 100k cells, e.g. most SingleCellsSubset objects) are read from the cell-major file into memory rather than sweeping the whole gene-major file, which makes the cost follow the selection instead of the store.

Usage

```
find_hvg_sc(
  object,
  hvg_no = 2000L,
  hvg_params = params_sc_hvg(),
  streaming = NULL,
  .verbose = TRUE
)
```

Arguments

object	SingleCells, MetaCells (or potentially other) class.
hvg_no	Integer. Number of highly variable genes to include. Defaults to 2000L.
hvg_params	List, see params_sc_hvg(). This list contains • method - Which method to use. One of c("vst", "meanvarbin", "dispersion", "scran", "residual") • loess_span - The span for the loess function to standardise the variance ("vst"), or of the lowess trend ("scran") • num_bin - Integer. Not yet implemented. • bin_method - String. One of c("equal_width", "equal_freq"). Not implemented yet. • mean_filter, min_mean, transform, use_min_width, min_width, min_window_count - The "scran" trend parameters, see params_hvg_scran_defaults()
streaming	Optional Boolean. Shall the data be streamed in. Useful for larger data sets where you wish to avoid loading in the whole data. If NULL, will automatically detect. Not used for MetaCells.
.verbose	Boolean or integer. Controls verbosity and returns run times. FALSE -> quiet, TRUE or 1L -> normal verbosity, 2L -> detailed verbosity.

Value

It will add the per-gene HVG statistics to the var table: mean, var, var_exp and var_std for "vst"; mean, dispersion, dispersion_scaled and bin for "meanvarbin" and "dispersion"; scran_mean, scran_var, scran_fitted and scran_residual (log2 scale) for "scran".

Examples

```
# the twenty most variable genes by the vst method
sc <- demo_single_cells(prepped = FALSE)
sc <- find_hvg_sc(sc, hvg_no = 20L, .verbose = FALSE)
length(get_hvg(sc))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`find_markers_sc` *Calculate DGE between two cell groups*

Description

This function can be used to calculate differentially expressed genes between two groups in the single cell data. At the moment, it has only an implementation for the Wilcox-based rank statistic.

Usage

```
find_markers_sc(
  object,
  cells_1,
  cells_2,
  method = c("wilcox"),
  alternative = c("twosided", "greater", "less"),
  min_prop = 0.05,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`cells_1`	String. The names of the cells in group 1. Need to be part of the cell names in the object, see `get_cell_names()`.
`cells_2`	String. The names of the cells in group 2. Need to be part of the cell names in the object, see `get_cell_names()`.
`method`	String. Which method to use for the calculations of the DGE. At the moment the only option is `"wilcox"`, but the parameter is reserved for future features.
`alternative`	String. Test alternative. One of `c("twosided", "greater", "less")`. Function will default to `"twosided"`.
`min_prop`	Numeric. The minimum proportion of cells that need to express the gene to be tested in any of the two groups.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

data.table with the DGE results from the test.

Examples

```
# Wilcoxon test between two of the planted cell types
sc <- demo_single_cells()
obs <- get_sc_obs(sc)
res <- find_markers_sc(
  sc,
  cells_1 = obs$cell_id[obs$cell_grp == "cell_type_1"],
  cells_2 = obs$cell_id[obs$cell_grp == "cell_type_2"],
  .verbose = FALSE
)
head(res)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

find_neighbours_sc *Find the neighbours for single cell.*

Description

This function will generate the kNNs based on a given embedding. Available algorithms are:

- `kmknn` - An exact kNN search that leverages k-means clustering under the hood to prune out data points. The default setting.
- `exhaustive` - An exhaustive, flat index. On smaller data sets often faster than the approximate nearest neighbour search algorithms.
- `hnsw` - Hierarchical Navigable Small World. A graph-based approximate nearest neighbour search algorithm; works well on large data sets. A benign race condition is leveraged during index build, making the build non-deterministic. Bigger impact on smaller data sets.
- `nndescent` - Nearest neighbour descent. Leverages concepts from `PyNNDescent` and works well on very large data sets similar to `hnsw`. Set `extract_knn = TRUE` in the kNN parameters to hand back the descent graph directly instead of beam searching it. That drops the query pass altogether, so it is much faster, but recall goes down a little.
- `ivf` - Inverted file index. Uses first k-means clustering to identify Voronoi cells and leverages these during querying. Works well on large data sets with high dimensionality and when you need to return large number of neighbours.
- `annoy` - Approximate nearest neighbours Oh Yeah. Tree-based index, used across different R single cell packages (Seurat, SCE). This version is purely memory-based.

Subsequently, the kNN graph will be additionally transformed into a shared nearest neighbour graph for clustering methods.

Usage

```
find_neighbours_sc(
  object,
  embd_to_use = "pca",
  no_embd_to_use = NULL,
  modality = c("rna", "adt"),
  neighbours_params = params_sc_neighbours(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells`, `MetaCells` (or potentially other) class.
`embd_to_use`	String. The embedding to use. Whichever you chose, it needs to be part of the object.
`no_embd_to_use`	Optional integer. Number of embedding dimensions to use. If `NULL` all will be used.
`modality`	String. One of `c("rna", "adt")`. You can only use `"adt"` on `SingleCellsMultiModal` class.
`neighbours_params`	List. Output of `params_sc_neighbours()`. A list with the following items: • full_snn - Boolean. Shall the full shared nearest neighbour graph be generated that generates edges between all cells instead of between only neighbours. • pruning - Numeric. Weights below this threshold will be set to 0 in the generation of the sNN graph. • snn_similarity - String. One of `c("rank", "jaccard")`. Defines how the weight from the SNN graph is calculated. For details, please see `params_sc_neighbours()`. • knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.
`seed`	Integer. For reproducibility.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

The object with added KNN matrix.

Examples

```
# kNN and the sNN graph on top of the PCA
sc <- demo_single_cells(prepped = FALSE)
sc <- find_hvg_sc(sc, hvg_no = 30L, .verbose = FALSE)
```

```
sc <- calculate_pca_sc(sc, no_pcs = 10L, .verbose = FALSE)
sc <- find_neighbours_sc(
  sc,
  neighbours_params = params_sc_neighbours(knn = list(k = 15L)),
  .verbose = FALSE
)
dim(get_knn_mat(sc))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`find_rbh_communities` *Find RBH communities*

Description

This function will identify communities in the reciprocal best hit (RBH) graph. It will iterate through resolutions and add the results to the class. Additionally, a column will be added that signifies the resolution with the best modularity.

Usage

```
find_rbh_communities(
  object,
  resolution_params = params_graph_resolution(),
  max_workers = NULL,
  parallel = TRUE,
  random_seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	The underlying class, see `RbhGraph()`.
`resolution_params`	List. Parameters for the resolution search, see `params_graph_resolution()`. Contains: • min_res - Float. Minimum resolution to test. • max_res - Float. Maximum resolution to test. • number_res - Integer. Number of resolutions to test between the `max_res` and `min_res`.
`max_workers`	Integer. Number of maximum cores to use. Defaults to half of the identified cores (to a maximum of 8).
`parallel`	Boolean. Shall the resolution search be in parallel.
`random_seed`	Integer. Random seed for reproducibility.
`.verbose`	Boolean. Controls verbosity of the function.

Value

The class with added community detection results.

Examples

```
# Leiden communities across a resolution sweep of the RBH graph
set.seed(123)
modules <- data.table::data.table(
  origin = rep(c("set_a", "set_b"), each = 20),
  module = rep(c("m1", "m2", "m3", "m4"), each = 10),
  gene = unlist(replicate(4, sample(letters, 10), simplify = FALSE))
)
object <- RbhGraph(
  modules,
  rbh_type = "set",
  dataset_col = "origin",
  module_col = "module",
  value_col = "gene"
)
object <- generate_rbh_graph(object, minimum_similarity = 0)
object <- find_rbh_communities(object, parallel = FALSE, .verbose = FALSE)
head(get_results(object))
```

---

find_specific_markers_sc

Find markers that are specific to a cell group

Description

This function scores one group of cells against every other group of a column separately, instead of against all of them pooled together. That difference matters for markers. A pooled test is dominated by whichever rival contributes the most cells, so a gene that is high in the reference and just as high in one small rival still comes out looking like a clean marker. Here the gene has to hold up against every rival, and the per-gene summaries (`min_auroc`, `median_auroc`, `min_rank`) tell you whether it does.

Leave `reference_group` as `NULL` to run every group of the column as the reference, or name one group to only get that arm. Either way, everything is calculated in a single pass over the genes.

The summaries rank on AUROC rather than the p-value on purpose. Group sizes vary a lot in practice and p-values scale with the group sizes, so a large rival would otherwise crowd out a small one regardless of effect size.

Usage

```
find_specific_markers_sc(
  object,
  column_of_interest,
  reference_group = NULL,
```

```
  method = "wilcox",
  alternative = c("greater", "less", "twosided"),
  min_prop = 0.05,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`column_of_interest`	String. The column in the obs table holding the groups, e.g. a cell type annotation or a clustering.
`reference_group`	Optional string. The group to use as the reference. If `NULL`, every group of `column_of_interest` is used as the reference in turn.
`method`	String. Which method to use for the calculations of the DGE. At the moment the only option is `"wilcox"`, but the parameter is reserved for future features.
`alternative`	String. Test alternative. One of `c("twosided", "greater", "less")`. This function will default to `"greater"`, i.e., genes upregulated in the reference group.
`min_prop`	Numeric. The minimum proportion of cells that need to express the gene in at least one of the groups. Applied once, globally, so every comparison's FDR is calculated over the same gene set.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

A `ScSpecificMarkers` class with the following elements

- summary - data.table. Per gene and reference group, the summaries across all rivals: `prop_ref`, `median_auroc`, `min_auroc`, `mean_auroc`, `max_auroc`, `worst_rival` (the rival achieving `min_auroc`), `min_rank` (best AUROC rank the gene reaches against any single rival), the Simes-combined p-value with its FDR, and the maximum p-value with its FDR.
- per_comparison - data.table. Per gene, reference group and rival, the underlying `auroc`, `lfc`, `prop_ref`, `prop_rival`, `z_scores`, `p_values` and `fdr`.
- params - List. The parameters the run used.

References

Soneson and Robinson, Nat Methods, 2018; Lun, et al., F1000Research, 2016

Examples

```
# markers that hold up against every rival cell type
sc <- demo_single_cells()
res <- find_specific_markers_sc(
  sc,
  column_of_interest = "cell_grp",
  .verbose = FALSE
)
head(res$summary)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

find_threshold_otsu *Find a threshold via the Otsu method*

Description

Find a threshold via the Otsu method

Usage

```
find_threshold_otsu(x, bins = 100L)
```

Arguments

x	Numerical vector. The vector for which to find the threshold via the Otsu method
bins	Integer. Number of bins to use.

Value

The threshold.

Examples

```
# threshold separating two Gaussian modes
set.seed(42)
find_threshold_otsu(c(rnorm(100), rnorm(100, mean = 5)))
```

`fit_residuals_sc` *Fit a residual model for single cell data*

Description

Fits scTransform (v2) or the analytic Pearson residual model over the cells currently kept. The fit lands in the cache, and `find_hvg_sc()` with `params_sc_hvg(method = "residual")`, `calculate_pca_sc()` with `residuals = TRUE` and `sct_corrected_counts_sc()` all read it from there.

Unlike the log-normalised layer, which is computed once at ingestion and written to disk, nothing is precomputed here: the residual rows are regenerated on demand by whatever consumes the fit.

Small cell selections (up to 10% of the stored cells and at most 100k cells) are read from the cell-major file into memory for the fit, the residual HVG selection and the residual PCA, instead of sweeping the whole gene-major file each time.

With `group_column` one model is fitted per group. That is what a multi-sample experiment wants, since each sample keeps its own depth and composition, and it changes how the residual HVG selection behaves, see `params_sc_hvg()`.

For `MetaCells` the counts are summed UMIs, so the negative binomial still applies, but at a much greater depth than a single cell. Prefer `method = "analytic_pearson"` there, and revisit the defaults of `params_sc_sctransform()`: `n_genes` and `n_cells` are sized for raw cells, of which there are usually far more than meta cells.

Usage

```
fit_residuals_sc(
  object,
  method = c("sctransform", "analytic_pearson"),
  group_column = NULL,
  covariate_columns = NULL,
  residual_params = NULL,
  gene_batch_size = NULL,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells`, `SingleCellsSubset` or `MetaCells` class.
`method`	String. One of `c("sctransform", "analytic_pearson")`. scTransform fits a negative binomial per gene and regularises the parameters; analytic Pearson is the closed-form alternative with one shared dispersion, which is much cheaper.
`group_column`	String or `NULL`. Column in the observation table to fit separate models over, usually the sample. `NULL` fits one model.

`covariate_columns`
: Character vector or `NULL`. Numeric columns in the observation table to add to the design. scTransform only. The library size is never a covariate, it enters as a fixed offset.

`residual_params`
: List or `NULL`. Parameters, see `params_sc_sctransform()` or `params_sc_apr()`. `NULL` takes the defaults for `method`.

`gene_batch_size`
: Integer or `NULL`. Genes held in memory per batch.

`seed`
: Integer. Random seed for the step-1 subsample.

`.verbose`
: Boolean or Integer. Controls verbosity.

Value

The object with the fitted model attached.

References

Choudhary and Satija, Genome Biology, 2022; Lause, Berens and Kobak, Genome Biology, 2021.

Examples

```
# analytic Pearson residuals over every kept cell
sc <- demo_single_cells(prepped = FALSE)
sc <- fit_residuals_sc(sc, method = "analytic_pearson", .verbose = FALSE)
get_residual_fit(sc)$n_groups

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`fix_meta_data_column` *Helper to fix meta-data columns to be R conform*

Description

This function will update the specified columns in the metadata of an `BulkDge()` or `BulkCoExp()` to be conform with R standard naming conventions. This is useful to do before running DGE methods as they expect standardised names.

Usage

```
fix_meta_data_column(object, col_names, ...)
```

Arguments

`object`
: The underlying object, either `BulkCoExp()` or `BulkDge()`.

`col_names`
: Character vector. The columns to fix.

`...`
: Additional arguments to parse to the functions.

Value

Returns the object with the respective metadata columns updated.

Examples

```
# make a contrast column safe for model.matrix()
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case 1", "control 1"), each = 50)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- fix_meta_data_column(object, "case_control")
unique(get_metadata(object)$case_control)
```

---

flag_cells *Add hard-threshold flags to a CellQc object*

Description

Unions new hard flags with existing ones per metric. Set `reset = TRUE` to clear existing hard flags first.

Usage

```
flag_cells(x, hard_thresholds, reset = FALSE)
```

Arguments

`x`	A `CellQc` object.
`hard_thresholds`	Named list. See `run_cell_qc`.
`reset`	Logical. Clear existing hard flags before applying.

Value

Updated `CellQc`.

Examples

```
# add a hard mitochondrial cut on top of the MAD flags
set.seed(42L)
metrics <- list(
  lib_size = c(rnorm(99, 1000, 100), 50),
  pct_mt = runif(100, 0, 20)
)
qc <- run_cell_qc(
  metrics,
  cells_to_keep = 0:99,
```

```
  directions = c(lib_size = "below", pct_mt = "above")
)
flag_cells(qc, list(pct_mt = c(upper = 15)))
```

`forceatlas2_sc` *Run ForceAtlas2 on a SingleCells/MetaCells object*

Description

Wrapper around `manifoldsR::forceatlas2()` and `manifoldsR::forceatlas2_from_graph()` for the `SingleCells` and `MetaCells` classes. ForceAtlas2 is a force-directed graph layout: every edge pulls, every pair of cells pushes, and gravity keeps disconnected components from drifting off. It is what scanpy's `draw_graph` does.

`graph` picks what gets laid out:

- `"knn"` - The kNN graph (cached if `use_knn = TRUE`, otherwise built from the chosen embedding), turned into the UMAP fuzzy union graph first. This matches scanpy.
- `"snn"` - The sNN graph from `find_neighbours_sc()`, i.e. the same graph the Leiden/Louvain clustering runs on. `use_knn`, `embd_to_use`, `no_embd_to_use`, `k`, `knn_method` and `nn_params` do not apply here, and neither do the graph and initialisation knobs in `fa2_params`. Pass `init_embd` to start from an existing 2D embedding, otherwise the layout starts from random positions.

Usage

```
forceatlas2_sc(
  object,
  graph = c("knn", "snn"),
  use_knn = TRUE,
  embd_to_use = "pca",
  slot_name = "fa2",
  no_embd_to_use = NULL,
  init_embd = NULL,
  modality = c("rna", "adt", "wnn"),
  k = 15L,
  knn_method = c("kmknn", "hnsw", "annoy", "nndescent", "balltree", "ivf", "exhaustive"),
  nn_params = manifoldsR::params_nn(),
  fa2_params = manifoldsR::params_fa2(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells`, `MetaCells` class.
`graph`	String. Which graph to lay out. One of `c("knn", "snn")`. Defaults to `"knn"`.

`use_knn`	Boolean. Use the kNN graph found in the object. Defaults to `TRUE`. If not available, will default to the embedding. `"knn"` only.
`embd_to_use`	String. The embedding to build the kNN graph from. Must be available in the object. `"knn"` only.
`slot_name`	String. The name of this embedding within the object. Defaults to `"fa2"`.
`no_embd_to_use`	Optional integer. Number of embedding dimensions to use. If `NULL` all will be used. `"knn"` only.
`init_embd`	Optional string. Name of a stored 2D embedding, e.g. `"umap"`, to initialise the layout with. `"snn"` only. Defaults to `NULL`.
`modality`	String. On which modality to run ForceAtlas2. One of `c("rna", "adt", "wnn")`. The two latter options are only available for multi-modal versions with the added data.
`k`	Integer. Number of nearest neighbours. Defaults to `15L`. `"knn"` only.
`knn_method`	String. Approximate nearest neighbour algorithm. One of `c("kmknn", "hnsw", "annoy", "nndescent", "balltree", "ivf", "exhaustive")`. Defaults to `"kmknn"`. Only used when neighbours are computed from the embedding. `"knn"` only.
`nn_params`	Named list. See `manifoldsR::params_nn()`. `"knn"` only.
`fa2_params`	Named list. See `manifoldsR::params_fa2()`.
`seed`	Integer. For reproducibility.
`.verbose`	Boolean. Controls verbosity.

Value

The object with a `"fa2"` embedding added.

References

Jacomy, et al., PLoS ONE, 2014

Examples

```
# ForceAtlas2 on the cached kNN graph, then on the sNN graph
sc <- demo_single_cells()
sc <- forceatlas2_sc(sc, .verbose = FALSE)
sc <- forceatlas2_sc(
  sc,
  graph = "snn",
  slot_name = "fa2_snn",
  .verbose = FALSE
)
dim(get_embedding(sc, "fa2_snn"))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`gene_set_proportions_sc` *Calculate the proportions of reads for specific gene sets*

Description

This is a helper function that calculates proportions of reads belonging to given gene sets. This can be used for example for the calculation of percentage mitochondrial reads per cell. These will be automatically added to the obs table

Usage

```
gene_set_proportions_sc(
  object,
  gene_set_list,
  streaming = NULL,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`gene_set_list`	A named list with each element containing the gene identifiers of that set. These should be the same as `get_gene_names(object)`!
`streaming`	Optional Boolean. Shall the data be streamed in. Useful for larger data sets where you wish to avoid loading in the whole data. If `NULL`, will automatically detect.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

It will add the columns based on the names in the `gene_set_list` to the obs table.

Examples

```
# read proportion of a gene set, the mitochondrial percentage pattern
sc <- demo_single_cells(prepped = FALSE)
sc <- gene_set_proportions_sc(
  sc,
  gene_set_list = list(set_a = c("gene_01", "gene_02", "gene_03")),
  .verbose = FALSE
)
head(unlist(sc[["set_a"]]))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`GeneOntologyElim` *Gene Ontology data*

Description

This class is used to store the gene ontology information for usage in GSE elimination methods.

Usage

```
GeneOntologyElim(go_data_dt, min_genes)
```

Arguments

`go_data_dt`	A data.table that contains the gene ontology information. This can be extract with for example `get_go_data_human()`.
`min_genes`	data.frame. Meta-data information in form of a data.frame.

Value

Returns the class for subsequent usage.

Properties

go_info data.table. Contains the gene ontology identifiers and names.

go_to_genes List. Contains the genes within each gene ontology term.

ancestry List. Contains the ancestors for each gene ontology term.

levels List. Which gene ontology terms sit at which level.

min_genes Integer, the minimum genes in the gene ontology term to conduct the test.

Examples

```
# human GO restricted to terms with at least 25 genes
go_obj <- GeneOntologyElim(
  get_go_data_human(.verbose = FALSE),
  min_genes = 25L
)
go_obj
```

`generate_bt_meta_cells_sc` *Generate meta cells based on hdWGCNA and return a* `MetaCells` *object*

Description

This function implements the meta cell aggregation from Morabito, et al. The generation of metacells is a useful approach for subsequent application of for example correlation-based methods to identify co-regulated genes.

Usage

```
generate_bt_meta_cells_sc(
  object,
  sc_meta_cell_params = params_sc_bt_metacells(),
  regenerate_knn = FALSE,
  embd_to_use = "pca",
  no_embd_to_use = NULL,
  cells_to_use = NULL,
  target_size = 1e+05,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object` `SingleCells` or `SingleCellsSubset` class.

`sc_meta_cell_params`
List. Output of `params_sc_bt_metacells()`. A list with the following items:

- max_shared - Maximum number of allowed shared neighbours for the meta cell to be considered.
- target_no_metacells - Number of target meta cells you wish to reach.
- max_iter - Maximum number of iterations you want to use for the algorithm.
- knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.

`regenerate_knn`
Boolean. Shall a kNN graph be regenerated.

`embd_to_use` String. The embedding to use. Only relevant if you set regenerate_knn to `TRUE`.

`no_embd_to_use`
Optional integer. Number of embedding dimensions to use. If `NULL` all will be used. Only relevant if you set regenerate_knn to `TRUE`.

cells_to_use	Optional string. Names of the cells to use for the generation of the meta-cells. If provided, this function will regenerate the kNN graph no matter what.
target_size	Numeric. The library target size to normalise the meta cells to.
seed	Integer. Seed for reproducibility.
.verbose	Boolean or integer. Controls verbosity and returns run times. FALSE -> quiet, TRUE or 1L -> normal verbosity, 2L -> detailed verbosity.

Value

A MetaCells() with the data generated by this meta cell aggregation method.

References

Morabito, et al. Cell Rep Methods, 2023

Examples

```
# 50 meta cells off the demo object
sc <- demo_single_cells()
mc <- generate_bt_meta_cells_sc(
  sc,
  sc_meta_cell_params = params_sc_bt_metacells(
    target_no_metacells = 50L,
    knn = list(k = 10L)
  ),
  .verbose = FALSE
)
mc

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

generate_cellsweep_test_data

Single cell test data with a planted ambient profile

Description

This function generates synthetic data for CellSweep test purposes. Every real barcode is a two-component multinomial: a planted fraction alpha of its library comes from the soup, the rest from its own cell type profile. Empty droplets are pure soup at a much smaller library size, which is what the ambient profile is estimated off. Real barcodes come first, the empty droplets after.

Usage

```
generate_cellsweep_test_data(
  syn_data_params = params_sc_synthetic_cellsweep(),
  seed = 42L
)
```

Arguments

`syn_data_params`	List. Contains the parameters for the generation of the synthetic data, see: `params_sc_synthetic_cellsweep()`.
`seed`	Integer. The seed for the generation of the synthetic data.

Details

The empty droplets carry no cell type label, which is exactly what `cellsweep_sc()` keys off: barcodes that are neither empty nor annotated are excluded from the fit. Load the counts with a fully permissive `params_sc_min_quality()`, otherwise the ingest deletes the empty droplets the model trains on.

Value

List with the following items

- counts - dgRMatrix with cells x genes.
- obs - data.table with `cell_id`, `cell_grp` (`NA` for the empty droplets), `sample_id`, `is_empty` and `alpha_true` (`NA` for the empty droplets).
- var - data.table that contains the gene information.
- ambient_true - Numeric vector. The soup the empty droplets were drawn from, named by gene and summing to one.
- celltype_profiles_true - Numeric matrix of cell types x genes. Each row sums to one.

Examples

```
# a small synthetic experiment with a planted soup
data <- generate_cellsweep_test_data(
  syn_data_params = params_sc_synthetic_cellsweep(
    n_real = 60L,
    n_empty = 200L,
    n_genes = 60L
  )
)
dim(data$counts)
head(data$obs, 3)
```

---

generate_dialogue_test_data

Single cell test data with a planted multicellular programme

Description

Generates synthetic data DIALOGUE should be able to solve. Every cell type gets its own noise and its own sample-level nuisance factors; only the first feature column and the planted genes carry a shared per-sample latent, so anything found beyond that is spurious.

Usage

```
generate_dialogue_test_data(
  syn_data_params = params_sc_synthetic_dialogue(),
  seed = 42L
)
```

Arguments

syn_data_params	List. Contains the parameters for the generation of the synthetic data, see: params_sc_synthetic_dialogue().
seed	Integer. The seed for the generation of the synthetic data.

Details

Same generator the Rust integration tests use, so the planted structure cannot drift between the two suites. What differs is the scale: R takes the raw count layer and lets the normal ingestion path log-normalise it, where the Rust tests feed the planted layer straight in.

Cells are laid out contiguously by cell type and, within a cell type, by sample, so every cell type sees every sample. That is the easy case for the method and is what a fixture should be.

Value

List with the following items

- counts - dgRMatrix with cells x genes of raw counts.
- obs - data.table with `cell_id`, `cell_grp`, `sample_id` and `cell_quality`. The last is pure noise, independent of the planted programme: hand it to `dialogue_sc(quality_col = "cell_quality")` so the covariate does not carry the signal you are trying to find.
- var - data.table with the gene information.
- features - Named list of numeric matrices, one per cell type, with the cell identifiers as row names. Hand this straight to dialogue_sc().
- latent - Numeric vector. The per-sample latent the planted programme follows, named by sample.
- planted - Named list of character vectors. The planted gene identifiers per cell type.

References

Jerby-Arnon & Regev, Nature Biotechnology, 2022

Examples

```
# synthetic data with a multicellular programme planted across samples
data <- generate_dialogue_test_data()
names(data$features)
head(data$planted[[1]])
```

`generate_gene_module_data` *Generates synthetic gene module data.*

Description

Generates an artifical matrix of active modules (for a maximum of 8) in samples being active. Also allows for overlap between the modules. Designed to test some of the algorithms.

Usage

```
generate_gene_module_data(
  n_samples = 24L,
  n_genes = 60L,
  n_modules = 4L,
  overlap_fraction = 0.25,
  seed = 10101L
)
```

Arguments

`n_samples`	Integer. Number of samples.
`n_genes`	Integer. Number of genes.
`n_modules`	Integer. Number of active gene modules. To a maximum of 10.
`overlap_fraction`	Float. How much the same modules can be active in the same samples.
`seed`	Integer. Initial random seed for generation of the synthetic data. Default: 10101L.

Value

A `synthetic_matrix_modules` class with the following items:

- data - The data matrix.
- meta_data - The sample metadata.

Examples

```
# four partially overlapping modules over 24 samples
mods <- generate_gene_module_data(
  n_samples = 24L,
  n_genes = 60L,
  n_modules = 4L
)
dim(mods$data)
head(mods$meta_data)
```

generate_hotspot_membership *Identify hotspot gene clusters*

Description

Identify hotspot gene clusters

Usage

```
generate_hotspot_membership(x, fdr_threshold = 0.05, min_size = 10L)

## S3 method for class 'Hotspot'
generate_hotspot_membership(x, fdr_threshold = 0.05, min_size = 10L)
```

Arguments

`x`	An object to generate the hotspot gene clusters for.
`fdr_threshold`	Numeric. The maximum FDR for a given gene-gene local correlation to be included.
`min_size`	Integer. Minimum cluster size.

Examples

```
# cluster the local gene-gene correlations into modules
sc <- demo_single_cells()
hs <- hotspot_gene_cor_sc(sc, .verbose = FALSE)
generate_hotspot_membership(hs)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

generate_knn_sc *Generate a* `SingleCellNearestNeighbour` *from a single cell class*

Description

This function will generate the kNNs based on a given embedding. Available algorithms are:

- `kmknn` - An exact kNN search that leverages k-means clustering under the hood to prune out data points. The default setting.
- `exhaustive` - An exhaustive, flat index. On smaller data sets often faster than the approximate nearest neighbour search algorithms.

- `hnsw` - Hierarchical Navigable Small World. A graph-based approximate nearest neighbour search algorithm; works well on large data sets. A benign race condition is leveraged during index build, making the build non-deterministic. Bigger impact on smaller data sets.
- `nndescent` - Nearest neighbour descent. Leverages concepts from `PyNNDescent` and works well on very large data sets similar to `hnsw`. Set `extract_knn = TRUE` in the kNN parameters to hand back the descent graph directly instead of beam searching it. That drops the query pass altogether, so it is much faster, but recall goes down a little.
- `ivf` - Inverted file index. Uses first k-means clustering to identify Voronoi cells and leverages these during querying. Works well on large data sets with high dimensionality and when you need to return large number of neighbours.
- `annoy` - Approximate nearest neighbours Oh Yeah. Tree-based index, used across different R single cell packages (Seurat, SCE). This version is purely memory-based.

Usage

```
generate_knn_sc(
  object,
  embd_to_use = "pca",
  cells_to_use = NULL,
  no_embd_to_use = NULL,
  neighbours_params = params_sc_neighbours(),
  seed = 42L,
  .validate_index = TRUE,
  .verbose = TRUE
)
```

Arguments

`object` `SingleCells` class.

`embd_to_use` String. The embedding to use. Whichever you chose, it needs to be part of the object.

`cells_to_use` String. Optional cell names to include in the generation of the kNN graph. If `NULL` all (filtered) cells in the object will be used.

`no_embd_to_use` Optional integer. Number of embedding dimensions to use. If `NULL` all will be used.

`neighbours_params` List. Output of `params_sc_neighbours()`. A list with the following items:

- full_snn - Boolean. Shall the full shared nearest neighbour graph be generated that generates edges between all cells instead of between only neighbours. Not used for this function.
- pruning - Numeric. Weights below this threshold will be set to 0 in the generation of the sNN graph. Not used for this function.

- snn_similarity - String. One of `c("rank", "jaccard")`. Defines how the weight from the SNN graph is calculated. For details, please see `params_sc_neighbours()`. Not used for this function.
- knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.

`seed` Integer. For reproducibility.

`.validate_index`
Boolean. Shall an exhaustive search against a subset of cells be run to validate the approximate nearest neighbour index.

`.verbose` Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

Initialised `sc_knn` with the kNN data.

Examples

```
# a standalone kNN object off the PCA embedding
sc <- demo_single_cells()
knn <- generate_knn_sc(sc, .validate_index = FALSE, .verbose = FALSE)
dim(get_knn_mat(knn))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

generate_ligand_target_influence

Generate the ligand to target influence matrix

Description

Computes the NicheNet-style ligand to target gene influence matrix. Builds the gene universe as the union of all symbols across both networks, remaps to 0-indexed integer node IDs for the Rust side, and wraps the resulting matrix in a `LigandTargetInfluence` object.

Usage

```
generate_ligand_target_influence(
  ligand_seeds,
  ppi_network,
  grn_network,
  params = params_ligand_target()
)
```

Arguments

`ligand_seeds`	List of character vectors. Each element is one ligand symbol or a group treated as one complex (e.g. `list(c("TGFB1", "TGFB2"))`). Optionally named; if unnamed, row names default to the symbols joined with `+`.
`ppi_network`	data.table with columns `from`, `to`, `weight` (character, character, numeric). Protein-protein / signalling layer.
`grn_network`	data.table with columns `from`, `to`, `weight` (character, character, numeric). Gene regulatory layer.
`params`	List. As returned by `params_ligand_target()`.

Value

A `LigandTargetInfluence` object.

Examples

```
# two disjoint signalling components over a toy network
ppi <- data.table::data.table(
  from = c("L1", "SIG1", "L2", "SIG2"),
  to = c("SIG1", "TF1", "SIG2", "TF2"),
  weight = 1.0
)
grn <- data.table::data.table(
  from = rep(c("TF1", "TF2"), each = 3),
  to = c("G1", "G2", "G3", "G4", "G5", "G6"),
  weight = 1.0
)
inf <- generate_ligand_target_influence(
  ligand_seeds = list(L1 = "L1", L2 = "L2"),
  ppi_network = ppi,
  grn_network = grn,
  params = params_ligand_target(ltf_cutoff = 0)
)
inf
```

`generate_personalisation_vec` *Helper function to create personalisation vectors*

Description

Helper function to create personalisation vectors

Usage

```
generate_personalisation_vec(graph, node_weights)
```

Arguments

`graph`	igraph. The graph for which to produce the personalisation vector.
`node_weights`	Named numeric. The names represent the nodes and the values the strength of the reset.

Value

The personalisation vector for subsequent usage in page-rank

Examples

```
# reset weight split across two seed nodes, normalised to sum to one
g <- igraph::graph_from_data_frame(
  data.frame(from = c("a", "b", "c"), to = c("b", "c", "d")),
  directed = TRUE
)
generate_personalisation_vec(g, node_weights = c(a = 3, c = 1))
```

`generate_rbh_graph` *Generate an RBH graph.*

Description

This function will generate an RBH graph based on set similarity between gene modules. You have the option to use an overlap coefficient instead of Jaccard similarity and to specify a minimum similarity.

Usage

```
generate_rbh_graph(
  object,
  minimum_similarity,
  k_best = 1L,
  overlap_coefficient = FALSE,
  spearman = FALSE
)
```

Arguments

`object`	The underlying class, see `RbhGraph()`.
`minimum_similarity`	The minimum similarity to create an edge.
`k_best`	Integer. Number of best neighbours to consider. If set to `1L`, this behaves as the traditional reciprocal best hit. If you set this to `3L` you consider edges if the modules is in the top 3 best modules by similarity for each other.

overlap_coefficient
: Boolean. Shall the overlap coefficient be used instead of Jaccard similarity. Only relevant if the underlying class is set to set similarity.

spearman
: Boolean. Shall Spearman correlation be used. Only relevant if the underlying class is set to correlation-based similarity.

Value

The class with added properties.

Examples

```
# Jaccard-based reciprocal best hits between two module sets
set.seed(123)
modules <- data.table::data.table(
  origin = rep(c("set_a", "set_b"), each = 20),
  module = rep(c("m1", "m2", "m3", "m4"), each = 10),
  gene = unlist(replicate(4, sample(letters, 10), simplify = FALSE))
)
object <- RbhGraph(
  modules,
  rbh_type = "set",
  dataset_col = "origin",
  module_col = "module",
  value_col = "gene"
)
object <- generate_rbh_graph(object, minimum_similarity = 0)
head(get_rbh_res(object))
```

---

generate_sc_knn *Generate a new SingleCellNearestNeighbour from data*

Description

Generate a new SingleCellNearestNeighbour from data

Usage

```
generate_sc_knn(
  data,
  neighbours_params = params_sc_neighbours(),
  seed = 42L,
  .validate_index = FALSE,
  .verbose = TRUE
)
```

Arguments

`data` Numerical matrix. Samplex x features. The embedding matrix from which to generate the kNN data.

`neighbours_params`
List. Output of `params_sc_neighbours()`. A list with the following items:

- full_snn - Boolean. Shall the full shared nearest neighbour graph be generated that generates edges between all cells instead of between only neighbours. (Not used in this function.)
- pruning - Numeric. Weights below this threshold will be set to 0 in the generation of the sNN graph. (Not used in this function.)
- snn_similarity - String. One of `c("rank", "jaccard")`. Defines how the weight from the SNN graph is calculated. For details, please see `params_sc_neighbours()`. (Not used in this function.)
- knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.

`seed` Integer. Random seed for reproducibility.

`.validate_index`
Boolean. Shall an exhaustive search against a subset of cells be run to validate the approximate nearest neighbour index.

`.verbose` Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

The `SingleCellNearestNeighbour` for downstream usage.

Examples

```
# kNN over a random embedding; the rows have to carry cell names
set.seed(42L)
embd <- matrix(rnorm(500 * 10), nrow = 500)
rownames(embd) <- sprintf("cell_%03d", 1:500)
knn <- generate_sc_knn(embd, .verbose = FALSE)
dim(get_knn_mat(knn))
```

---

generate_seacells_sc *Generate meta cells based on SEACells and return a* `MetaCells` *object*

Description

This function implements the meta cell aggregation from Persad et al., and returns the resulting SEACells. Compared to other algorithms, a kernel archetype analysis is used to identify the metacells. For details, please refer to the publication.

Usage

```
generate_seacells_sc(
  object,
  seacell_params = params_sc_seacells(),
  embd_to_use = "pca",
  no_embd_to_use = NULL,
  cells_to_use = NULL,
  regenerate_knn = FALSE,
  target_size = 1e+05,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object` `SingleCells` or `SingleCellsSubset` class.

`seacell_params`
List. Output of `params_sc_seacells()`. A list with the following items:

- n_sea_cells - Number of SEA cells to detect.
- max_fw_iters - Maximum iterations for the Franke-Wolfe algorithm per matrix update.
- convergence_epsilon - Convergence threshold. Algorithm stops when RSS change < epsilon * RSS(0).
- max_iter - Maximum iterations to run SEACells for.
- min_iter - Minimum iterations to run SEACells for.
- greedy_threshold - Maximum number of cells before defaulting to rapid random selection of archetypes.
- graph_building - Graph building method.
- pruning - Boolean. Shall small values be pruned during the Franke-Wolfe iterations.
- pruning_threshold - The threshold below which pruning shall be applied during Franke-Wolfe iterations.
- knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.

`embd_to_use` String. The embedding to use. Atm, the only option is `"pca"`. Only relevant if you set regenerate_knn to `TRUE`.

`no_embd_to_use`
Optional integer. Number of embedding dimensions to use. If `NULL` all will be used. Only relevant if you set regenerate_knn to `TRUE`.

`cells_to_use` Optional string. Names of the cells to use for the generation of the SEACells.

`regenerate_knn`
Boolean. Shall a kNN graph be regenerated. If not, the internal one will be used.

`target_size` Numeric. The library target size to normalise the meta cells to.

`seed`	Integer. Seed for reproducibility.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

A `MetaCells()` with the data generated by this meta cell aggregation method.

References

Persad, et al. Nat Biotechnol, 2023

Examples

```
# archetypal analysis based meta cells
sc <- demo_single_cells()
mc <- generate_seacells_sc(
  sc,
  seacell_params = params_sc_seacells(
    n_sea_cells = 50L,
    min_iter = 5L,
    knn = list(k = 10L)
  ),
  .verbose = FALSE
)
mc

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

generate_single_cell_test_data

Single cell test data

Description

This function generates synthetic data for single cell test purposes. These data can be used for testing functionality of various single cell functions.

Usage

```
generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(),
  seed = 42L
)
```

Arguments

`syn_data_params`
: List. Contains the parameters for the generation of synthetic data, see: `params_sc_synthetic_data()`. Has the following elements:

  - n_cells - Integer. Number of cells.
  - n_genes - Integer. Number of genes.
  - marker_genes - List. A nested list that indicates which gene indices are markers for which cell.
  - n_batches - Integer. Number of batches.
  - batch_effect_strength - String. Indicates the strength of the batch effect to add.

`seed`
: Integer. The seed for the generation of the seed data.

Value

List with the following items

- counts - dgRMatrix with cells x genes.
- obs - data.table that contains the cell information.
- var - data.table that contains the var information.

Examples

```
# a small synthetic experiment with three planted cell types
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
dim(data$counts)
head(data$obs, 3)
```

`generate_supercells_sc` *Generate SuperCells and return a* `MetaCells` *object*

Description

This function implements the meta cell aggregation from Bilous, et al. The core idea is to use the walktrap community detection on the kNN graph. For details, please refer to the paper.

Usage

```
generate_supercells_sc(
  object,
  sc_supercell_params = params_sc_supercell(),
  regenerate_knn = FALSE,
  embd_to_use = "pca",
  no_embd_to_use = NULL,
  cells_to_use = NULL,
  target_size = 1e+05,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`sc_supercell_params`	List. Output of `params_sc_supercell()`. A list with the following items: • walk_length - Integer. Walk length for the Walktrap algorithm. • graining_factor - Numeric. Graining level of data (proportion of number of single cells in the initial dataset to the number of metacells in the final dataset) • use_kernel - Boolean. Shall a kernel be applied. • kith_neighbour - Optional integer. Which neighbour to use for the sigma estimation. • knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.
`regenerate_knn`	Boolean. Shall a kNN graph be regenerated.
`embd_to_use`	String. The embedding to use. Only relevant if you set regenerate_knn to `TRUE`.
`no_embd_to_use`	Optional integer. Number of embedding dimensions to use. If `NULL` all will be used. Only relevant if you set regenerate_knn to `TRUE`.
`cells_to_use`	Optional string. Names of the cells to use for the generation of the meta-cells. If provided, this function will regenerate the kNN graph no matter what.
`target_size`	Numeric. The library target size to normalise the meta cells to.
`seed`	Integer. Seed for reproducibility.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

A `MetaCells()` with the data generated by this meta cell aggregation method.

References

Bilous, et al. BMC Bioinform., 2022

Examples

```
# walktrap based super cells at a graining factor of 10
sc <- demo_single_cells()
mc <- generate_supercells_sc(
  sc,
  sc_supercell_params = params_sc_supercell(
    graining_factor = 10,
    knn = list(k = 10L)
  ),
  regenerate_knn = TRUE,
  .verbose = FALSE
)
mc

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`generate_wnn_graph_sc` *Generate a weighted nearest neighbour (WNN) graph*

Description

This function implements the approach from Hao et al., to generate a weighted nearest neighbour graph given two 'omics modalities (for example RNA and ADT). Per-cell modality weights are computed by comparing within- and cross-modality neighbourhood distances, and these weights are used to fuse the two modalities into a single multimodal kNN graph.

The per-modality kNN graphs are always recomputed internally from the chosen embeddings at `knn_range` neighbours (see `params_sc_wnn()`); the kNN graphs previously stored on the object via `find_neighbours_sc()` are not reused, as WNN requires a larger candidate pool than the default neighbour search.

The result is stored in the object's `other_data` under `"wnn"`, containing the fused kNN graph as a `SingleCellNearestNeighbour` (with a kernelised pseudo-distance metric), the sNN graph as an igraph and a table of per-cell modality weights. This graph can subsequently be used for clustering and 2D embedding via the `"wnn"` modality.

Usage

```
generate_wnn_graph_sc(
  object,
  modality_1 = "rna",
  modality_2 = "adt",
  embd_to_use_1 = "pca",
```

```
  embd_to_use_2 = "pca",
  no_embd_to_use_1 = NULL,
  no_embd_to_use_2 = NULL,
  wnn_params = params_sc_wnn(),
  full_snn = TRUE,
  pruning = 1/15,
  snn_similarity = "jaccard",
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCellsMultiModal` class to which to add the WNN.
`modality_1`	String. First modality. Defaults to `"rna"`.
`modality_2`	String. Second modality. Defaults to `"adt"`.
`embd_to_use_1`	String. The embedding to use for the first modality. Must be available in the object for `modality_1`. Defaults to `"pca"`.
`embd_to_use_2`	String. The embedding to use for the second modality. Must be available in the object for `modality_2`. Defaults to `"pca"`.
`no_embd_to_use_1`	Optional integer. Number of embedding dimensions to use for `embd_to_use_1`. If `NULL`, all will be used.
`no_embd_to_use_2`	Optional integer. Number of embedding dimensions to use for `embd_to_use_2`. If `NULL`, all will be used.
`wnn_params`	Named list. Controls the parameters for the WNN generation, see `params_sc_wnn()`.
`full_snn`	Boolean. Shall the full shared nearest neighbour graph be generated that generates edges between all cells instead of between only neighbours.
`pruning`	Numeric. Weights below this threshold will be set to 0 in the generation of the sNN graph. Seurat uses for example 1/15 with k = 20.
`snn_similarity`	String. One of `c("rank", "jaccard")`. The Jaccard similarity calculates the Jaccard between the neighbours, whereas the rank method calculates edge weights based on the ranking of shared neighbours. For the rank method, the weight is determined by finding the shared neighbour with the lowest combined rank across both cells, where lower-ranked (closer) shared neighbours result in higher edge weights Both methods produce weights normalised to the range `[0, 1]`.
`seed`	Integer. For reproducibility.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

The `SingleCellsMultiModal` object with the WNN graph and per-cell modality weights added to `other_data[["wnn"]]`.

References

Hao et al., Cell, 2021

Examples

```
# a fused RNA + ADT graph, both modalities reduced with PCA first
rna <- generate_single_cell_test_data()
adt <- generate_single_cell_test_data_adt()
dir <- tempfile("bixverse_mm")
dir.create(dir)
object <- load_r_data(
  SingleCellsMultiModal(dir_data = dir),
  counts = rna$counts,
  obs = rna$obs,
  var = rna$var,
  sc_qc_param = params_sc_min_quality(min_unique_genes = 5L),
  .verbose = FALSE
)
object <- add_adt_counts_sc(object, adt_counts = adt$counts, method = "clr")
object <- find_hvg_sc(object, hvg_no = 30L, .verbose = FALSE)
object <- calculate_pca_sc(object, no_pcs = 15L, .verbose = FALSE)
object <- calculate_pca_adt_sc(object, no_pcs = 10L)
object <- generate_wnn_graph_sc(object, .verbose = FALSE)
get_snn_graph(object, modality = "wnn")

unlink(dir, recursive = TRUE, force = TRUE)
```

---

get_adt_feature_info *Get the ADT feature info*

Description

Returns the number of cells expressing a given ADT

Usage

```
get_adt_feature_info(x)

## S3 method for class 'ADTCounts'
get_adt_feature_info(x)
```

Arguments

x	The object from which to get the ADT feature information

Value

A data.table with the ADT feature information

Examples

```
# number of cells expressing each protein
adt <- generate_single_cell_test_data_adt()
cell_info <- stats::setNames(
  seq_len(nrow(adt$counts)),
  rownames(adt$counts)
)
adt_clr <- new_adt_counts_clr(adt$counts, cell_info = cell_info)
head(get_adt_feature_info(adt_clr))
```

---

get_adt_names *Get the ADT feature names*

Description

Get the main ADT feature names

Usage

```
get_adt_names(x)

## S3 method for class 'ADTCounts'
get_adt_names(x)
```

Arguments

x	An object to get the gene names from.

Value

The primary ADT feature identifiers stored in the class.

Examples

```
# protein identifiers held by an `ADTCounts`
adt <- generate_single_cell_test_data_adt()
cell_info <- stats::setNames(
  seq_len(nrow(adt$counts)),
  rownames(adt$counts)
)
adt_clr <- new_adt_counts_clr(adt$counts, cell_info = cell_info)
head(get_adt_names(adt_clr))
```

get_adt_sample_info *Get the ADT sample info*

Description

Returns sample information in terms of ADT capture, number of features, etc.

Usage

```
get_adt_sample_info(x)

## S3 method for class 'ADTCounts'
get_adt_sample_info(x)
```

Arguments

x	The object from which to get the ADT sample information

Value

A data.table with the ADT sample information

Examples

```
# per-cell ADT capture: non-zero proteins and library size
adt <- generate_single_cell_test_data_adt()
cell_info <- stats::setNames(
  seq_len(nrow(adt$counts)),
  rownames(adt$counts)
)
adt_clr <- new_adt_counts_clr(adt$counts, cell_info = cell_info)
head(get_adt_sample_info(adt_clr))
```

get_available_embeddings *Get the available embeddings*

Description

Returns the available embedding as names from the class. This function is used for the single cell-related classes and methods.

Usage

```
get_available_embeddings(x, ...)

## S7 method for class <bixverse::MetaCells>
get_available_embeddings(x, ...)

## S3 method for class 'ScCache'
get_available_embeddings(x, ...)

## S7 method for class <bixverse::SingleCells>
get_available_embeddings(x, ...)

## S7 method for class <bixverse::SingleCellsSubset>
get_available_embeddings(x, ...)
```

Arguments

x	An object to get embedding from
...	Other parameters.

Value

Get the names of the available embeddings.

Examples

```
# what is in the cache to plot against
sc <- demo_single_cells()
get_available_embeddings(sc)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_best_model *Get the selected model from an LDA topic count sweep*

Description

Returns the fit at `best_k`, or at a `k` you name, without refitting. The sweep keeps every model it fitted.

Usage

```
get_best_model(x, k = NULL)

## S3 method for class 'LdaKSweepResult'
get_best_model(x, k = NULL)
```

Arguments

x	LdaKSweepResult object.
k	Optional integer. The topic count to extract. If NULL, uses the best_k the sweep selected.

Details

best_k is never below five, see lda_k_sweep(). Pass k explicitly if the raw metrics point somewhere the selection could not go.

Value

An LdaResult.

Examples

```
# pull the selected fit out of a sweep without refitting
set.seed(42L)
corpus <- matrix(rbinom(200L * 40L, 1L, 0.05), nrow = 200L, ncol = 40L)
corpus[1:100, 1:10] <- rbinom(1000L, 1L, 0.6)
corpus[101:200, 11:20] <- rbinom(1000L, 1L, 0.6)
colnames(corpus) <- sprintf("term_%02d", 1:40)
sweep_res <- lda_k_sweep(corpus > 0, k_range = 5:7, .verbose = FALSE)
dim(get_best_model(sweep_res))
```

---

get_best_run *Get the best run from a stabilised NMF result*

Description

Extracts the run with the lowest reconstruction loss and returns it as a single-run NmfResult for downstream methods.

Usage

```
get_best_run(x)

## S3 method for class 'StabilisedNmfResult'
get_best_run(x)
```

Arguments

x	StabilisedNmfResult object.

Value

An NmfResult containing the W/H of the best run.

Examples

```
# the restart with the lowest reconstruction loss
sc <- demo_single_cells()
res <- stabilised_nmf_sc(sc, k = 5L, n_runs = 5L, .verbose = FALSE)
get_best_run(res)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_c_pca_factors *Get the contrastive PCA factors*

Description

Getter function for the feature factors of the contrastive PCA

Usage

```
get_c_pca_factors(object)
```

Arguments

object	The underlying class, see BulkCoExp().

Value

The sample scores of the contrastive PCA run. If not found, returns a warning and NULL.

Examples

```
# sample scores of the contrastive components
cpca_data <- synthetic_c_pca_data()
target <- t(cpca_data$target)
background <- t(cpca_data$background)
meta <- data.table::data.table(sample_id = rownames(target))
obj <- BulkCoExp(target, meta)
obj <- preprocess_bulk_coexp(obj, .verbose = FALSE)
obj <- contrastive_pca_processing(obj, background, .verbose = FALSE)
obj <- contrastive_pca(obj, alpha = 2.5, no_pcs = 5L)
head(get_c_pca_factors(obj)[, 1:3])
```

get_c_pca_loadings *Get the contrastive PCA loadings*

Description

Getter function for the feature loadings of the contrastive PCA

Usage

```
get_c_pca_loadings(object)
```

Arguments

object	The underlying class, see BulkCoExp().

Value

The feature loadings of the contrastive PCA run. If not found, returns a warning and NULL.

Examples

```
# feature loadings of the contrastive components
cpca_data <- synthetic_c_pca_data()
target <- t(cpca_data$target)
background <- t(cpca_data$background)
meta <- data.table::data.table(sample_id = rownames(target))
obj <- BulkCoExp(target, meta)
obj <- preprocess_bulk_coexp(obj, .verbose = FALSE)
obj <- contrastive_pca_processing(obj, background, .verbose = FALSE)
obj <- contrastive_pca(obj, alpha = 2.5, no_pcs = 5L)
head(get_c_pca_loadings(obj)[, 1:3])
```

get_cell_indices *Get the index position for a gene*

Description

Returns the index for a given gene based on the internal gene mapping. This is used for the single cell-related classes and methods.

Usage

```
get_cell_indices(x, cell_ids, rust_index)

## S7 method for class <bixverse::MetaCells>
get_cell_indices(x, cell_ids, rust_index)

## S3 method for class 'ScMap'
get_cell_indices(x, cell_ids, rust_index)

## S7 method for class <bixverse::SingleCells>
get_cell_indices(x, cell_ids, rust_index)

## S7 method for class <bixverse::SingleCellsSubset>
get_cell_indices(x, cell_ids, rust_index)
```

Arguments

x	An object to get the gene index from.
cell_ids	String vector. The cell ids to search for.
rust_index	Bool. Shall rust-based indexing be returned.

Value

The indices of the cells

Examples

```
# R-based positions of two cells
sc <- demo_single_cells(prepped = FALSE)
get_cell_indices(sc, c("cell_001", "cell_002"), rust_index = FALSE)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_cell_info *Get the cell idx (R-based) and cell names*

Description

Returns the cell indices (R-based) and the cell names (usually barcodes) from the object for further downstream usage.

Usage

```
get_cell_info(x, filtered = TRUE)

## S3 method for class 'ScMap'
get_cell_info(x, filtered = TRUE)
```

```
## S7 method for class <bixverse::SingleCells>
get_cell_info(x, filtered = TRUE)
```

Arguments

x	An object to get the cell info from
filtered	Boolean. If TRUE, only the cells to keep will be returned.

Value

A named vector with elements -> cell_idx, names -> cell_names.

Examples

```
# cell indices carrying the barcodes as names
sc <- demo_single_cells(prepped = FALSE)
head(get_cell_info(sc), 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_cell_names *Get the cell names*

Description

Returns the cell names (usually barcodes).

Usage

```
get_cell_names(x, filtered = FALSE)

## S3 method for class 'ScMap'
get_cell_names(x, filtered = FALSE)

## S7 method for class <bixverse::SingleCells>
get_cell_names(x, filtered = FALSE)

## S7 method for class <bixverse::SingleCellsSubset>
get_cell_names(x, filtered = FALSE)
```

Arguments

x	An object to get the cell names from.
filtered	Boolean. Shall, if found only the cell names of the cells_to_keep be returned (see set_cells_to_keep(). Defaults to FALSE

Value

The cell names (barcodes)

Examples

```
# barcodes of the cells that passed quality control
sc <- demo_single_cells(prepped = FALSE)
head(get_cell_names(sc, filtered = TRUE), 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_cell_ranger_params

Helper to generate cell ranger input parameters

Description

Resolves the three files a Cell Ranger MTX directory holds and wraps them into the parameter list `load_mtx()` wants. Handles the v2 naming (`genes.tsv`) and the v3 one (`features.tsv`), and the `.csv` variant that `write_cellranger_output()` can emit.

Usage

```
get_cell_ranger_params(dir_data, cells_as_rows = FALSE, has_hdr = FALSE)
```

Arguments

`dir_data`	String. The directory with the Cell Ranger outputs
`cells_as_rows`	Boolean. Are the cells the rows of the matrix? Cell Ranger writes genes x cells, so this defaults to `FALSE`. Set to `TRUE` for output of `write_cellranger_output()` written with `rows = "cells"`.
`has_hdr`	Boolean. Do the barcode and feature files carry a header row? Cell Ranger writes none, so this defaults to `FALSE`. `write_cellranger_output()` does write one.

Value

A list based on `params_sc_mtx_io()`.

Examples

```
# round trip through the package's own writer
dir <- tempfile("cellranger")
dir.create(dir)
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 50L, n_genes = 40L)
)
write_cellranger_output(
```

```
  f_path = dir,
  counts = data$counts,
  obs = data$obs,
  var = data$var,
  rows = "genes",
  format_type = "tsv",
  .verbose = FALSE
)
str(get_cell_ranger_params(dir, has_hdr = TRUE))

unlink(dir, recursive = TRUE, force = TRUE)
```

`get_cellphonedb_db` *Download the CellPhoneDB ligand-receptor database*

Description

Fetches the CellPhoneDB v5.0.0 database from the cellphonedb-data repository and flattens it to one row per interaction. Each partner is resolved to the gene symbols of its subunits: one gene for a single protein, several for a complex. The download is cached in `dir`.

Usage

```
get_cellphonedb_db(dir = tempdir(), .verbose = TRUE)
```

Arguments

`dir`	String. Directory to store and look for `cellphonedb.zip`. Defaults to `tempdir()`.
`.verbose`	Boolean. Controls verbosity of the download.

Value

A data.table with one row per interaction and the columns:

- interaction_id - The CellPhoneDB interaction identifier.
- partner_a, partner_b - The partner names: the gene symbol for a single protein, the complex name otherwise.
- genes_a, genes_b - List columns with the gene symbols of the subunits.
- is_complex_a, is_complex_b - Is the partner a complex.
- directionality - E.g. `"Ligand-Receptor"`.
- classification - The signalling classification.

References

Troulé et al., Nat Protoc, 2025.

get_cells_to_keep *Get the cells to keep*

Description

Returns the indices of the cells that survived quality control. These are stored 0-indexed for Rust, so add one before using them in R.

Usage

```
get_cells_to_keep(x)

## S3 method for class 'ScMap'
get_cells_to_keep(x)

## S7 method for class <bixverse::SingleCells>
get_cells_to_keep(x)

## S7 method for class <bixverse::SingleCellsSubset>
get_cells_to_keep(x)
```

Arguments

x	An object from which to get the cells to keep from. These are 0-indexed.

Value

Integer vector with 0-indices of the cells to keep.

Examples

```
# 0-based for Rust, so add one before indexing in R
sc <- demo_single_cells(prepped = FALSE)
head(get_cells_to_keep(sc) + 1, 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

get_centroids_sc *Get k-means centroids from a fast cluster result*

Description

Get k-means centroids from a fast cluster result

Usage

```
get_centroids_sc(x)

## S3 method for class 'SingleCellFastClusters'
get_centroids_sc(x)
```

Arguments

x	SingleCellFastClusters object.

Examples

```
# the k-means centroids the graph clustering was built on
sc <- demo_single_cells()
res <- fast_cluster_sc(
  sc,
  resolutions = 1.0,
  n_centroids = 30L,
  return_kmeans = TRUE,
  .verbose = FALSE
)
dim(get_centroids_sc(res))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_cistarget_res *Extract the TF to gene data from the ScenicGrn object*

Description

Extract the TF to gene data from the ScenicGrn object

Usage

```
get_cistarget_res(x)

## S3 method for class 'ScenicGrn'
get_cistarget_res(x)
```

Arguments

x	ScenicGrn object from which to extract the TF to gene data.table.

Value

data.table with TF to gene information

Examples

```
# CisTarget against a synthetic ranking database
sc <- demo_single_cells()
tfs <- sprintf("gene_%02d", 1:5)
grn <- scenic_grn_sc(
  sc,
  tf_ids = tfs,
  scenic_params = params_scenic(
    min_counts = 1L,
    learner_params = list(n_trees = 20L)
  ),
  .verbose = FALSE
)
grn <- identify_tf_to_genes(
  grn,
  method = "top_k",
  k_tfs = 3L,
  .verbose = FALSE
)
genes <- get_gene_names(sc)
targets <- split(get_tf_to_gene(grn)$gene, get_tf_to_gene(grn)$tf)
rankings <- matrix(
  vapply(
    tfs,
    function(tf) {
      as.integer(rank(!(genes %in% targets[[tf]]), ties.method = "first"))
    },
    integer(length(genes))
  ),
  nrow = length(genes),
  dimnames = list(genes, sprintf("motif_%s", tfs))
)
annot <- data.table::data.table(
  motif = sprintf("motif_%s", tfs),
  TF = tfs,
  annotationSource = factor(
    c(rep("directAnnotation", 4), "inferredBy_MotifSimilarity")
  )
)
grn <- tf_to_genes_motif_enrichment(
  grn,
  motif_rankings = rankings,
  annot_data = annot,
  cis_target_params = params_cistarget(
    auc_threshold = 1,
    nes_threshold = 1,
    high_conf_cats = "directAnnotation",
    low_conf_cats = "inferredBy_MotifSimilarity"
  ),
  .verbose = FALSE
)
get_cistarget_res(grn)[, c("gs_name", "motif", "nes")]
```

```
unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`get_cor_graph` *Get correlation-based graph*

Description

Helper function to get a correlation-based igraph from the class

Usage

```
get_cor_graph(object, epsilon, .verbose)
```

Arguments

`object`	The class, see `BulkCoExp()`.
`epsilon`	Float. The epsilon parameter for the RBF function, in this case the bump function.
`.verbose`	Boolean. Controls verbosity of the function.

Value

A list with the following elements:

- graph - The igraph
- params - A list that contains the parameters of the graph generation and general graph information (node, edge numbers).

Examples

```
# igraph from the stored correlations at a fixed epsilon
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- cor_module_processing(obj, cor_method = "spearman", .verbose = FALSE)
graph_res <- get_cor_graph(obj, epsilon = 2, .verbose = FALSE)
graph_res$params
```

`get_data` *Get the ready obs data from various sub method*

Description

Helper method that creates data.tables with cell indices which were used in the given analysis + the values that are to be added to the obs table in the DuckDB.

Usage

```
get_data(x, columns = NULL, ...)

## S3 method for class 'CellQc'
get_data(x, ...)

## S3 method for class 'ScListRes'
get_data(x, ...)

## S3 method for class 'ScMatrixRes'
get_data(x, columns = NULL, ...)

## S3 method for class 'ScrubletRes'
get_data(x, ...)

## S3 method for class 'BoostRes'
get_data(x, ...)

## S3 method for class 'ScDblFinderRes'
get_data(x, ...)

## S3 method for class 'SingleCellFastClusters'
get_data(x, ...)

## S3 method for class 'NmfResult'
get_data(x, ...)

## S3 method for class 'ConsensusNmfResult'
get_data(x, ...)
```

Arguments

`x`	An object to get the data from.
`columns`	Optional string. For some of the functions you can decide to only extract specific columns.
`...`	Other parameters

Value

Returns a data.table with a cell_idx column for the cells included in the analysis and additional columns to be added to the obs table.

Examples

```
# the cell indices and cluster memberships a run produced
sc <- demo_single_cells()
res <- fast_cluster_sc(sc, resolutions = 1.0, .verbose = FALSE)
head(get_data(res), 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_dge_effect_sizes *Return the effect size results*

Description

Getter function to extract the Effect size results from the `BulkDge()` class.

Usage

```
get_dge_effect_sizes(object)
```

Arguments

`object`	`BulkDge` class.

Value

Returns the effect size results. (If found.)

Examples

```
# Hedge's G effect sizes per contrast
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- qc_bulk_dge(object, group_col = "case_control", .verbose = FALSE)
object <- normalise_bulk_dge(
  object,
  group_col = "case_control",
  .verbose = FALSE
)
object <- calculate_dge_hedges(
  object,
```

```
  contrast_column = "case_control",
  .verbose = FALSE
)
head(get_dge_effect_sizes(object))
```

`get_dge_limma_voom` *Return the Limma Voom results*

Description

Getter function to extract the Limma Voom results from the `BulkDge()` class.

Usage

```
get_dge_limma_voom(object)
```

Arguments

`object`	`BulkDge` class.

Value

Returns the Limma Voom results. (If found.)

Examples

```
# topTable results for every contrast that was fitted
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- qc_bulk_dge(object, group_col = "case_control", .verbose = FALSE)
object <- normalise_bulk_dge(
  object,
  group_col = "case_control",
  .verbose = FALSE
)
object <- calculate_dge_limma(
  object,
  contrast_column = "case_control",
  .verbose = FALSE
)
head(get_dge_limma_voom(object))
```

get_dge_list *Return the DGEList*

Description

Builds an edgeR `DGEList` from the filtered counts, library sizes and normalisation factors stored in the `BulkDge()` class. bixverse does not need edgeR itself, so this requires edgeR to be installed. Normalisation factors are one until `normalise_bulk_dge()` has run.

Usage

```
get_dge_list(object)
```

Arguments

`object`	`BulkDge` class.

Value

An edgeR `DGEList`, or `NULL` with a warning if `qc_bulk_dge()` has not been run.

Examples

```
# the edgeR DGEList built during QC
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- qc_bulk_dge(object, group_col = "case_control", .verbose = FALSE)
if (requireNamespace("edgeR", quietly = TRUE)) {
  dim(get_dge_list(object))
}
```

get_dge_qc_plot *Return QC plots*

Description

Getter function to extract the QC plots from the `BulkDge()` class. These are added when you run for example `qc_bulk_dge()` and `normalise_bulk_dge()`. You can either leave the plot choice as `NULL` and provide input when prompted, or you provide the name. The possible plots that might be in the class

- p1_nb_genes_cohort Proportion of non-zero genes for the samples in the respective cohorts (added after using `qc_bulk_dge()`).

- p2_outliers An outlier plot based on the data from p1, added after using `qc_bulk_dge()`.
- p3_voom_normalization Initial Voom normalisation plot after filtering lowly expressed genes. Added after using `normalise_bulk_dge()`.
- p4_boxplot_normalization Expression levels after normalisation. Added after using `normalise_bulk_dge()`.
- p5_pca_case_control A PCA plot with the chosen case control category. Added if `calculate_pca_bulk_dge()` is run.
- p6_batch_correction_plot A PCA plot pre and post batch correction with the case-control category overlayed. Added if `batch_correction_bulk_dge()` is run.

Usage

```
get_dge_qc_plot(object, plot_choice = NULL)
```

Arguments

`object`	`BulkDge` class.
`plot_choice`	Optional string or integer. Index or name of the plate.

Value

Returns the DGEList stored in the class.

Examples

```
# pull a named QC plot back off the object
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- qc_bulk_dge(object, group_col = "case_control", .verbose = FALSE)
get_dge_qc_plot(object, plot_choice = "p1_nb_genes_cohort")
```

`get_diagnostics` *Get diagnostics from a BulkModuleResult*

Description

Returns one diagnostic by key, or the whole named list if `which` is `NULL`. Contents are method-specific: CoReMo stores `stability` and `cluster_quality`; Leiden stores `resolution_used` and `modularity`; ICA stores `ica_meta` and `stability_scores`; DGRDL stores `feature_laplacian`, `sample_laplacian`, and the grid-search table; NMF stores `final_loss`, `n_iter`, `converged`, and (for stabilised runs) `losses`, `best_idx`.

Usage

```
get_diagnostics(object, which = NULL)
```

Arguments

`object`	A `BulkModuleResult`.
`which`	String or `NULL`. Name of the diagnostic to return. If `NULL`, returns the whole list.

Value

The requested diagnostic, the named list, or `NULL` (with warning) if `which` is not among the stored diagnostic keys.

Examples

```
# convergence diagnostics of an NMF fit
syn <- synthetic_bulk_cor_matrix()
mat <- log1p(t(syn$counts))
meta <- data.table::data.table(sample_id = rownames(mat))
object <- BulkCoExp(raw_data = mat, meta_data = meta)
object <- preprocess_bulk_coexp(object, hvg = 500L, .verbose = FALSE)
object <- nmf_bulk(object, k = 3L, .verbose = FALSE)
res <- S7::prop(object, "final_results")
names(get_diagnostics(res))
get_diagnostics(res, "converged")
```

`get_diffcor_graph` *Get differential correlation-based graph*

Description

Helper function to get a differential correlation-based igraph from the class

Usage

```
get_diffcor_graph(object, min_cor = 0.2, fdr_threshold = 0.05, .verbose = TRUE)
```

Arguments

`object`	The class, see `BulkCoExp()`.
`min_cor`	Float. The minimum absolute correlation that needs to be present in either data set.
`fdr_threshold`	Float. The maximum FDR that is tolerated for the generation of the graph.
`.verbose`	Boolean. Controls the verbosity of the function.

Value

A list with the following elements:

- graph - The igraph
- params - A list that contains the parameters of the graph generation and general graph information (node, edge numbers).

Examples

```
# igraph from the differential correlations
sig <- synthetic_signal_matrix()
mat <- t(sig$mat)
target <- mat[sig$group %in% c("group1", "group2"), ]
background <- mat[sig$group == "group3", ]
meta <- data.table::data.table(sample_id = rownames(target))
obj <- BulkCoExp(target, meta)
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- diffcor_module_processing(
  obj, background, cor_method = "pearson", .verbose = FALSE
)
graph_res <- get_diffcor_graph(obj, .verbose = FALSE)
graph_res$params
```

---

get_differential_abundance_res

Get the differential abundance results

Description

Get the differential abundance results

Usage

```
get_differential_abundance_res(x)

## S3 method for class 'miloR'
get_differential_abundance_res(x)
```

Arguments

x	An object from which to get the differential abundance results from.

Value

The differential abundance results stored in the object if found.

Examples

```
# neighbourhood level differential abundance table
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 500L,
    n_genes = 50L,
    n_samples = 6L,
    sample_bias = "even"
  )
)
milo <- get_miloR_abundances_sc(
  sc,
  sample_id_col = "sample_id",
  miloR_params = params_sc_miloR(k_refine = 10L),
  .verbose = FALSE
)
design_df <- data.frame(
  grp = rep(c("a", "b"), each = 3),
  row.names = sprintf("sample_%i", 1:6)
)
milo <- test_nhoods(milo, design = ~grp, design_df = design_df)
head(get_differential_abundance_res(milo))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_diffusion_perms *Get the diffusion permutations*

Description

Returns the diffusion Z-scores if you ran `permute_seed_nodes()`.

Usage

```
get_diffusion_perms(object)
```

Arguments

`object`	The underlying class `NetworkDiffusions()`.

Value

The diffusion Z scores if found. Otherwise `NULL`.

Examples

```
# Z-scores from node-degree adjusted permutations
set.seed(42)
g <- igraph::sample_pa(15, directed = FALSE)
edges <- data.table::setDT(igraph::as_data_frame(g))[, `:=`(
```

```
  from = sprintf("node_%i", from),
  to = sprintf("node_%i", to)
)]
object <- NetworkDiffusions(edges, weighted = FALSE, directed = FALSE)
object <- diffuse_seed_nodes(object, c(node_1 = 1, node_3 = 1), "max")
object <- permute_seed_nodes(object, perm_iters = 100L, .verbose = FALSE)
head(get_diffusion_perms(object))
```

`get_diffusion_vector` *Get the diffusion vector*

Description

Returns the diffusion vector if you ran `tied_diffusion()` or `diffuse_seed_nodes()`.

Usage

```
get_diffusion_vector(object)
```

Arguments

`object`	The underlying class `NetworkDiffusions()`.

Value

The diffusion vector if found. If you did not run either diffusion functions, it will return `NULL` and a warning.

Examples

```
# diffusion scores after a single seed node diffusion
set.seed(42)
g <- igraph::sample_pa(15, directed = FALSE)
edges <- data.table::setDT(igraph::as_data_frame(g))[, `:=`(
  from = sprintf("node_%i", from),
  to = sprintf("node_%i", to)
)]
object <- NetworkDiffusions(edges, weighted = FALSE, directed = FALSE)
object <- diffuse_seed_nodes(object, c(node_1 = 1, node_3 = 1), "max")
head(get_diffusion_vector(object))
```

`get_embedding` *Get the embedding*

Description

General wrapper function that can be used to pull out any embedding stored in the class. This function is used for the single cell-related classes and methods.

Usage

```
get_embedding(x, embd_name, ...)

## S7 method for class <bixverse::MetaCells>
get_embedding(x, embd_name, ...)

## S3 method for class 'ScCache'
get_embedding(x, embd_name, ...)

## S7 method for class <bixverse::SingleCells>
get_embedding(x, embd_name, ...)

## S7 method for class <bixverse::SingleCellsSubset>
get_embedding(x, embd_name, ...)
```

Arguments

`x`	An object to get embedding from
`embd_name`	String. The name of the embedding to return. The function will throw an error if the embedding does not exist.
`...`	Other parameters.

Value

Get the specified embeddings from the object (if found).

Examples

```
# any stored embedding, by name
sc <- demo_single_cells()
dim(get_embedding(sc, "pca"))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

get_epsilon_res *Return the epsilon data*

Description

Getter function to extract the `epsilon param ~ power law goodness of fit` data from the `BulkCoExp()` class.

Usage

```
get_epsilon_res(object)
```

Arguments

`object`	`BulkCoExp` class.

Value

Returns the epsilon data. (If found. Otherwise `NULL`).

Examples

```
# scale-free fit of the affinity matrix across epsilons
syn <- synthetic_bulk_cor_matrix()
mat <- log1p(t(syn$counts))
meta <- data.table::data.table(sample_id = rownames(mat))
object <- BulkCoExp(raw_data = mat, meta_data = meta)
object <- preprocess_bulk_coexp(object, hvg = 200L, .verbose = FALSE)
object <- cor_module_processing(
  object,
  cor_method = "pearson",
  .verbose = FALSE
)
object <- cor_module_check_epsilon(
  object,
  rbf_func = "gaussian",
  .verbose = FALSE
)
head(get_epsilon_res(object))
```

get_factors *Get factor matrices from a BulkModuleResult*

Description

Returns one factor matrix by key, or the whole named list of factor matrices if `which` is `NULL`. Keys are method-specific; see the `factors` argument of `new_bulk_module_result()` for the shared vocabulary.

Usage

```
get_factors(object, which = NULL)
```

Arguments

object	A BulkModuleResult.
which	String or NULL. Name of the factor matrix to return. If NULL, returns the whole list.

Value

A matrix, the named list of matrices, or NULL (with warning) if which is not among the stored factor keys.

Examples

```
# gene loadings and sample activities of an NMF fit
syn <- synthetic_bulk_cor_matrix()
mat <- log1p(t(syn$counts))
meta <- data.table::data.table(sample_id = rownames(mat))
object <- BulkCoExp(raw_data = mat, meta_data = meta)
object <- preprocess_bulk_coexp(object, hvg = 500L, .verbose = FALSE)
object <- nmf_bulk(object, k = 3L, .verbose = FALSE)
res <- S7::prop(object, "final_results")
names(get_factors(res))
dim(get_factors(res, "gene_loadings"))
```

---

get_feature_mat *Get the feature matrix used for the classifier*

Description

Get the feature matrix used for the classifier

Usage

```
get_feature_mat(x, ...)

## S3 method for class 'ScDblFinderRes'
get_feature_mat(x, ...)
```

Arguments

x	An object to get the feature matrix from. This will only include the values of the observed cells.
...	Additional parameters to forward to the method.

Examples

```
# the features the scDblFinder classifier was trained on
sc <- demo_single_cells(prepped = FALSE)
res <- scdblfinder_sc(
  sc,
  scdblfinder_params = params_scdblfinder(
    pca = list(no_pcs = 10L),
    n_genes = 25L,
    cxds_genes = 25L
  ),
  return_features = TRUE,
  .verbose = FALSE
)
dim(get_feature_mat(res))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_fpkm_counts *Return the FPKM-normalised counts*

Description

Getter function to extract the FPKM-normalised counts from the `BulkDge()` class.

Usage

```
get_fpkm_counts(object)
```

Arguments

`object`	`BulkDge` class.

Value

Returns the FPKM-normalised counts. (If found.)

Examples

```
# FPKM counts, which normalise_bulk_dge() only computes on request
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
gene_lengths <- stats::setNames(
  rep(2000, nrow(syn$counts)),
  rownames(syn$counts)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- qc_bulk_dge(object, group_col = "case_control", .verbose = FALSE)
```

```
object <- normalise_bulk_dge(
  object,
  group_col = "case_control",
  calc_fpkm = TRUE,
  gene_lengths = gene_lengths,
  .verbose = FALSE
)
get_fpkm_counts(object)[1:3, 1:3]
```

`get_gene_indices` *Get the index position for a gene*

Description

Returns the index for a given gene based on the internal gene mapping. This is used for the single cell-related classes and methods.

Usage

```
get_gene_indices(x, gene_ids, rust_index)

## S7 method for class <bixverse::MetaCells>
get_gene_indices(x, gene_ids, rust_index)

## S3 method for class 'ScMap'
get_gene_indices(x, gene_ids, rust_index)

## S7 method for class <bixverse::SingleCells>
get_gene_indices(x, gene_ids, rust_index)

## S7 method for class <bixverse::SingleCellsSubset>
get_gene_indices(x, gene_ids, rust_index)
```

Arguments

`x`	An object to get the gene index from.
`gene_ids`	String vector. The gene ids to search for.
`rust_index`	Bool. Shall Rust-based indexing be returned.

Value

The indices of the genes

Examples

```
# R-based positions of two genes
sc <- demo_single_cells(prepped = FALSE)
get_gene_indices(sc, c("gene_01", "gene_02"), rust_index = FALSE)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

get_gene_lengths *Get the gene lengths*

Description

Get the gene lengths

Usage

```
get_gene_lengths(x, species = c("human", "mouse", "rat"), ...)

## S7 method for class <bixverse::BulkCoExp>
get_gene_lengths(x, species = c("human", "mouse", "rat"), ...)
```

Arguments

x	Object to extract gene lengths from. Can be a matrix or BulkCoExp object.
species	String. One of `c("human", "mouse", "rat")`.
...	Additional parameters passed to methods.

Examples

```
## Not run:
# median transcript length per Ensembl gene, queried from Ensembl
counts <- matrix(
  1:4,
  nrow = 2,
  dimnames = list(c("ENSG00000141510", "ENSG00000012048"), c("s1", "s2"))
)
get_gene_lengths(counts, species = "human")

## End(Not run)
```

get_gene_names *Get the gene names*

Description

Get the main gene names (for example symbols or Ensembl identifiers).

Usage

```
get_gene_names(x)

## S3 method for class 'ScMap'
get_gene_names(x)

## S7 method for class <bixverse::SingleCells>
get_gene_names(x)

## S7 method for class <bixverse::SingleCellsSubset>
get_gene_names(x)
```

Arguments

x	An object to get the gene names from.

Value

The primary gene identifiers stored in the class.

Examples

```
# the primary gene identifiers held by the object
sc <- demo_single_cells(prepped = FALSE)
head(get_gene_names(sc), 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_gene_names_from_idx

Get the gene names based on the gene idx

Description

Get the gene names based on the gene idx

Usage

```
get_gene_names_from_idx(x, gene_idx, rust_based = TRUE)

## S3 method for class 'ScMap'
get_gene_names_from_idx(x, gene_idx, rust_based = TRUE)

## S7 method for class <bixverse::SingleCells>
get_gene_names_from_idx(x, gene_idx, rust_based = TRUE)

## S7 method for class <bixverse::SingleCellsSubset>
get_gene_names_from_idx(x, gene_idx, rust_based = TRUE)
```

Arguments

x	An object to get the gene names from.
gene_idx	Integer. The original gene indices for which to return the gene names.
rust_based	Boolean. Is it Rust-based, i.e., 0-index or R-based, i.e., 1-indexed.

Examples

```
# Rust indices translated back into gene identifiers
sc <- demo_single_cells(prepped = FALSE)
get_gene_names_from_idx(sc, gene_idx = 0:2, rust_based = TRUE)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_go_data_human *Wrapper function to load and process the gene ontology data.*

Description

This function loads in gene ontology data stored in the package and processes it into the format for `GeneOntologyElim()`. Wraps `load_go_human_data()` and `process_go_data()` into one.

Usage

```
get_go_data_human(filter_relationships = TRUE, .verbose = TRUE)
```

Arguments

filter_relationships	Boolean. Shall the ontology be filtered to only `"is_a"` and `"part_of"` relationships. Defaults to TRUE.
.verbose	Boolean. Controls verbosity of the function.

Value

A data.table

Examples

```
# human GO data ready for GeneOntologyElim()
go_dt <- get_go_data_human(.verbose = FALSE)
dim(go_dt)
```

get_grid_search_res *Get the grid search results*

Description

Getter function to extract the grid search results. If not found will return NULL.

Usage

```
get_grid_search_res(object)
```

Arguments

object	The class, see BulkCoExp().

Value

data.table with the grid search results (if found. Otherwise NULL.)

Examples

```
# the grid search table
syn <- generate_gene_module_data(n_samples = 24L, n_genes = 60L)
obj <- BulkCoExp(syn$data, syn$meta_data)
obj <- preprocess_bulk_coexp(obj, hvg = NULL, .verbose = FALSE)
obj <- dgrdl_grid_search(
  obj,
  neighbours_vec = c(3L, 5L),
  dict_size_vec = c(4L, 6L),
  seed_vec = 123L,
  .verbose = FALSE
)
get_grid_search_res(obj)
```

get_h *Get the H (cell activations) matrix*

Description

Get the H (cell activations) matrix

Usage

```
get_h(x)

## S3 method for class 'NmfResult'
get_h(x)

## S3 method for class 'StabilisedNmfResult'
get_h(x)

## S3 method for class 'ConsensusNmfResult'
get_h(x)
```

Arguments

x	An object holding NMF results.

Examples

```
# per cell activations of a five factor NMF
sc <- demo_single_cells()
res <- nmf_sc(sc, k = 5L, .verbose = FALSE)
dim(get_h(res))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_h5ad_dimensions *Helper function to get the dimensions and storage format*

Description

Distinguishes sparse (CSR/CSC) from dense storage. For dense X the orientation is inferred by comparing the matrix dims against the obs and var lengths (DENSE_ROW = cells x genes, DENSE_COL = genes x cells). Ties (no_obs == no_var) fall back to the AnnData convention (DENSE_ROW).

Usage

```
get_h5ad_dimensions(f_path)
```

Arguments

f_path	File path to the .h5ad file.

Value

A list with dims (named integer c(obs, var)) and type (one of "CSR", "CSC", "DENSE_ROW", "DENSE_COL").

Examples

```
# dimensions and storage layout without reading the counts
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
f_path <- tempfile(fileext = ".h5ad")
write_h5ad_sc(f_path, data$counts, data$obs, data$var, .verbose = FALSE)
get_h5ad_dimensions(f_path)

unlink(f_path)
```

---

get_hotspot_membership

Get the hotspot gene membership table

Description

Get the hotspot gene membership table

Usage

```
get_hotspot_membership(x)

## S3 method for class 'Hotspot'
get_hotspot_membership(x)
```

Arguments

x	The object from which to retrieve the hotspot gene membership

Examples

```
# the gene to module table hotspot clustered out
sc <- demo_single_cells()
hs <- hotspot_gene_cor_sc(sc, .verbose = FALSE)
hs <- generate_hotspot_membership(hs)
head(get_hotspot_membership(hs))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

get_hvg *Get the HVG*

Description

Returns the HVG indices. Pending class type this are 1-based (for R) or 0-based for Rust.

Usage

```
get_hvg(x)

## S7 method for class <bixverse::MetaCells>
get_hvg(x)

## S3 method for class 'ScMap'
get_hvg(x)

## S7 method for class <bixverse::SingleCells>
get_hvg(x)

## S7 method for class <bixverse::SingleCellsSubset>
get_hvg(x)
```

Arguments

x	An object to get HVG from.

Value

Indices of the stored HVG genes.

Examples

```
# stored 0-based, so map them back through the gene names
sc <- demo_single_cells()
get_gene_names_from_idx(sc, head(get_hvg(sc), 3))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

get_hvg_data_sc *Identify HVGs without mutating object state*

Description

Like find_hvg_sc() but does not mutate object. Returns a data.table with per-gene HVG statistics plus is_hvg/hvg_rank for the top hvg_no genes. Useful for computing HVGs on a subset of cells (e.g. a specific cell type) for downstream methods like NMF, without overwriting the HVGs stored on the object.

Usage

```
get_hvg_data_sc(
  object,
  cell_ids = NULL,
  hvg_no = 3000L,
  hvg_params = params_sc_hvg(),
  streaming = NULL,
  .verbose = TRUE
)
```

Arguments

object	SingleCells or MetaCells class.
cell_ids	Optional character. Cell ids (or meta cell ids) to restrict the HVG calculation to. If NULL, uses get_cells_to_keep() for SingleCells and all meta cells for MetaCells.
hvg_no	Integer. Number of top HVGs to flag. Defaults to 3000L.
hvg_params	List, see params_sc_hvg().
streaming	Optional Boolean. Stream the data. Ignored for MetaCells.
.verbose	Boolean or integer. Verbosity.

Value

data.table with gene_idx, gene_id, the HVG statistics returned by the Rust HVG function, an is_hvg boolean and an hvg_rank integer (NA for non-HVGs).

Examples

```
# HVG statistics without touching the object
sc <- demo_single_cells(prepped = FALSE)
dt <- get_hvg_data_sc(sc, hvg_no = 20L, .verbose = FALSE)
head(dt[(is_hvg), c("gene_id", "hvg_rank")], 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_ica_stability_res

Get the ICA component data (stability, convergence, redundancy)

Description

Getter function to extract the ICA component data in terms of stability, convergence and maximum absolute correlation between the component loadings (max_abs_loading_cor). If not found will return NULL.

Usage

```
get_ica_stability_res(object)
```

Arguments

object	The class, see BulkCoExp().

Value

data.table with the ICA parameter data (if found. Otherwise NULL.)

Examples

```
# the stability table behind the ncomp choice
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- ica_processing(obj, .verbose = FALSE)
obj <- ica_evaluate_comp(
  obj,
  ica_type = "logcosh",
  ncomp_params = params_ica_ncomp(custom_seq = seq(2L, 20L, by = 2L)),
  .verbose = FALSE
)
head(get_ica_stability_res(obj))
```

---

get_index_cells *Get the index cells*

Description

Get the index cells

Usage

```
get_index_cells(x)

## S3 method for class 'miloR'
get_index_cells(x)
```

Arguments

x	An object from which get the index cells.

Value

The indices of the cells in the neighbourhood.

Examples

```
# the cells the neighbourhoods were centred on
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 500L,
    n_genes = 50L,
    n_samples = 6L,
    sample_bias = "even"
  )
)
milo <- get_miloR_abundances_sc(
  sc,
  sample_id_col = "sample_id",
  miloR_params = params_sc_miloR(k_refine = 10L),
  .verbose = FALSE
)
head(get_index_cells(milo))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_inflection_point *Identify the inflection point for elbow-like data*

Description

This function will identify the index of the inflection point of an arbitrary series `x ~ y` via the biggest increase in the first derivative. Useful to identify key points in elbow plots.

Usage

```
get_inflection_point(x, y, span = 0.5)
```

Arguments

`x, y`	The x and y values.
`span`	The span parameter for the loess function.

Value

A list containing:

- inflection_idx - Index of the inflection point
- gradient_change - Absolute change in the first derivative

Examples

```
# elbow of a decaying curve that flattens out
x <- 1:20
y <- c(exp(-x[1:10] / 2), rep(0.005, 10))
get_inflection_point(x, y)$inflection_idx
```

get_influence *Get the ligand-target influence matrix*

Description

Get the ligand-target influence matrix

Usage

```
get_influence(x)

## S3 method for class 'LigandTargetInfluence'
get_influence(x)
```

Arguments

x	An object holding ligand-target influence results.

Examples

```
# the ligands x genes regulatory potential matrix
ppi <- data.table::data.table(
  from = c("L1", "SIG1", "L2", "SIG2"),
  to = c("SIG1", "TF1", "SIG2", "TF2"),
  weight = 1.0
)
grn <- data.table::data.table(
  from = rep(c("TF1", "TF2"), each = 3),
  to = c("G1", "G2", "G3", "G4", "G5", "G6"),
  weight = 1.0
)
inf <- generate_ligand_target_influence(
  ligand_seeds = list(L1 = "L1", L2 = "L2"),
  ppi_network = ppi,
  grn_network = grn,
  params = params_ligand_target(ltf_cutoff = 0)
)
dim(get_influence(inf))
```

get_kmeans_clusters *Get k-means cluster assignments from a fast cluster result*

Description

Get k-means cluster assignments from a fast cluster result

Usage

```
get_kmeans_clusters(x)

## S3 method for class 'SingleCellFastClusters'
get_kmeans_clusters(x)
```

Arguments

x	SingleCellFastClusters object.

Examples

```
# the centroid each cell was assigned to
sc <- demo_single_cells()
res <- fast_cluster_sc(
  sc,
  resolutions = 1.0,
  n_centroids = 30L,
  return_kmeans = TRUE,
  .verbose = FALSE
)
head(get_kmeans_clusters(res))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_knn_dist *Get the KNN distance*

Description

Getter for an integer matrix of samples x distances. Useful in combination with get_knn_mat().

Usage

```
get_knn_dist(x, ...)

## S7 method for class <bixverse::MetaCells>
get_knn_dist(x, ...)

## S3 method for class 'ScCache'
get_knn_dist(x, ...)

## S7 method for class <bixverse::SingleCells>
get_knn_dist(x, ...)

## S3 method for class 'SingleCellNearestNeighbour'
get_knn_dist(x, ...)
```

```
## S7 method for class <bixverse::SingleCellsSubset>
get_knn_dist(x, ...)
```

Arguments

x	An object to get the kNN distances from.
...	Other parameters.

Examples

```
# cells x neighbour distances
sc <- demo_single_cells()
dim(get_knn_dist(sc))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_knn_mat *Get the KNN matrix*

Description

Getter for an integer matrix of samples x neighbours.

Usage

```
get_knn_mat(x, ...)

## S7 method for class <bixverse::MetaCells>
get_knn_mat(x, ...)

## S3 method for class 'ScCache'
get_knn_mat(x, ...)

## S7 method for class <bixverse::SingleCells>
get_knn_mat(x, ...)

## S3 method for class 'SingleCellNearestNeighbour'
get_knn_mat(x, ...)

## S7 method for class <bixverse::SingleCellsSubset>
get_knn_mat(x, ...)
```

Arguments

x	An object to get the kNN matrix from.
...	Other parameters.

Examples

```
# cells x neighbours
sc <- demo_single_cells()
dim(get_knn_mat(sc))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_knn_obj *Get the KNN object*

Description

Returns the `SingleCellNearestNeighbour` from the object. This function is used for the single cell-related classes and methods.

Usage

```
get_knn_obj(x, ...)

## S7 method for class <bixverse::MetaCells>
get_knn_obj(x, ...)

## S3 method for class 'ScCache'
get_knn_obj(x, ...)

## S7 method for class <bixverse::SingleCells>
get_knn_obj(x, ...)

## S7 method for class <bixverse::SingleCellsSubset>
get_knn_obj(x, ...)
```

Arguments

`x`	An object to get the KNN class from.
`...`	Other parameters.

Value

The `SingleCellNearestNeighbour` object.

Examples

```
# the cached kNN, cells x neighbours
sc <- demo_single_cells()
dim(get_knn_mat(get_knn_obj(sc)))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`get_magic` *Get the MAGIC imputed layer*

Description

Returns the `ScMagic` layer written by `run_magic_sc()`. This function is used for the single cell-related classes and methods.

Usage

```
get_magic(x, ...)

## S3 method for class 'ScCache'
get_magic(x, ...)

## S7 method for class <bixverse::SingleCells>
get_magic(x, ...)

## S7 method for class <bixverse::SingleCellsSubset>
get_magic(x, ...)
```

Arguments

`x`	An object to get the imputed layer from.
`...`	Other parameters.

Value

The `ScMagic` object, or `NULL` when nothing was imputed.

Examples

```
# the imputed layer, only the genes MAGIC was asked for
sc <- demo_single_cells()
sc <- run_magic_sc(sc, features = get_gene_names(sc)[1:5], .verbose = FALSE)
dim(get_magic(sc)$data)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

get_marker_comparisons *Get the per-rival marker statistics*

Description

Returns the statistics that the summaries of `get_marker_summary()` are built from, i.e., one row per gene, reference group and rival. Useful to find out which rival a gene fails against.

Usage

```
get_marker_comparisons(x)

## S3 method for class 'ScSpecificMarkers'
get_marker_comparisons(x)
```

Arguments

x	`ScSpecificMarkers` object.

Value

A copy of the per comparison data.table.

Examples

```
# the per rival statistics the summaries are built from
sc <- demo_single_cells()
res <- find_specific_markers_sc(
  sc,
  column_of_interest = "cell_grp",
  .verbose = FALSE
)
head(get_marker_comparisons(res))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

get_marker_summary *Get the per-gene marker summaries across all rivals*

Description

Returns the summaries a marker is judged on: the AUROC of the reference against its rivals reduced to one row per gene and reference group. Rank on `median_auroc` for a marker that survives a single closely related rival, or on `min_auroc` when it has to beat every rival unambiguously.

Usage

```
get_marker_summary(x)

## S3 method for class 'ScSpecificMarkers'
get_marker_summary(x)
```

Arguments

x	ScSpecificMarkers object.

Value

A copy of the summary data.table.

Examples

```
# per gene summaries across every rival group
sc <- demo_single_cells()
res <- find_specific_markers_sc(
  sc,
  column_of_interest = "cell_grp",
  .verbose = FALSE
)
head(get_marker_summary(res))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_meta_cell_purity *Calculate meta cell purity without mutating object state*

Description

Like calc_meta_cell_purity() but does not mutate object. Returns a data.table with the purity per meta cell instead of stamping the columns onto the observation table. Useful for comparing several label columns, or for sweeping meta cell resolutions, without permanently annotating the object.

Usage

```
get_meta_cell_purity(
  object,
  original_cell_type,
  add_additional_info = c("none", "top_label", "top_two_labels"),
  add_entropy = FALSE
)
```

Arguments

`object`	`MetaCells` class.
`original_cell_type`	Character vector. The original cell type annotations of the object the meta cells came from. Either in the row order of its full (unfiltered) obs table, i.e. `get_sc_obs(x)$<column>`, or of the QC-passing cells only, i.e. `get_sc_obs(x, filtered = TRUE)$<column>`. Which one you passed is inferred from the length, so a vector matching neither is an error rather than a silently wrong purity.
`add_additional_info`	String. Which label information to add on top of the purity. One of `c("none", "top_label", "top_two_labels")`. Defaults to `"none"`, i.e. the purity only.
`add_entropy`	Boolean. Shall the normalised Shannon entropy of the label distribution be added as a diversity measure. Defaults to `FALSE`.

Details

Ties for the top (or second) label resolve to whichever label sorts first, as the labels are factorised internally. The entropy is the Shannon entropy of the label distribution within a meta cell, divided by `log(<number of distinct labels in original_cell_type>)`, so it sits in `[0, 1]` and stays comparable between meta cells. It is `0` if there is only a single label in the data.

Value

A data.table in observation table row order:

- meta_cell_idx - Index of the meta cell. Always returned.
- meta_cell_id - Identifier of the meta cell. Always returned.
- mc_purity - Fraction of the meta cell's cells that carry the most abundant label. Always returned.
- mc_top_label - Name of the most abundant label. Returned for `add_additional_info %in% c("top_label", "top_two_labels")`.
- mc_second_label - Name of the second most abundant label, `NA` for a pure meta cell. Returned for `add_additional_info = "top_two_labels"`.
- mc_second_frac - Fraction of the meta cell's cells carrying that second label, `0` for a pure meta cell. Returned for `add_additional_info = "top_two_labels"`.
- mc_entropy - Normalised Shannon entropy of the label distribution, see details. Returned for `add_entropy = TRUE`.

Examples

```
# the same numbers as a table, leaving the object untouched
sc <- demo_single_cells()
mc <- generate_bt_meta_cells_sc(
  sc,
```

```
  sc_meta_cell_params = params_sc_bt_metacells(target_no_metacells = 50L),
  .verbose = FALSE
)
purity <- get_meta_cell_purity(
  mc,
  original_cell_type = sc[["cell_grp"]]$cell_grp,
  add_additional_info = "top_label"
)
head(purity)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`get_metadata` *Return the metadata*

Description

Getter function to extract the metadata from the `BulkCoExp()` or `BulkDge()`.

Usage

```
get_metadata(object, ...)
```

Arguments

`object`	The underlying object, either `BulkCoExp()` or `BulkDge()`.
`...`	Additional arguments to parse to the functions.

Value

Returns the metadata stored in the class.

Examples

```
# sample metadata back out of a BulkDge
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
head(get_metadata(object))
```

get_miloR_abundances_sc *Generate an miloR abundance object for differential abundance testing*

Description

This function implements the miloR differential abundance testing on top of the kNN graph. The general idea of the approach is to use the kNN graph generated from the single cell data, generate representative neighbourhoods and calculate differential abundances within these neighbourhoods. For further details on the method, please refer to Dann, et al. This function will take a `SingleCells` class, run the neighbourhood detection, count the occurrences of a sample and return a `sc_miloR` class for subsequent differential abundance testing and further annotations.

Usage

```
get_miloR_abundances_sc(
  object,
  sample_id_col,
  embd_to_use = "pca",
  no_embd_to_use = NULL,
  miloR_params = params_sc_miloR(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`sample_id_col`	Character. The column in the obs table representing the sample identifier to count.
`embd_to_use`	Character. The embedding to use for the refinement procedure. Please use the same here as you used to generate the neighbours! Defaults to `"pca"`.
`no_embd_to_use`	Optional integer. If you only want to use a subset of the embedding.
`miloR_params`	A list, please see `params_sc_miloR()`. The list has the following parameters: • prop - Numeric. Proportion of cells to sample as neighbourhood indices. Must be in (0,1). • k_refine - Integer. Number of neighbours to use for refinement. • refinement_strategy - String. Strategy for refining sampled indices. One of `c("approximate", "bruteforce", "index")`. • index_type - String. Type of kNN index to use. One of `c("nndescent", "ivf", "hnsw", "annoy", "exhaustive")`.

- knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.

`seed` Integer. Seed for reproducibility

`.verbose` Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

A `miloR` class with the following elements:

- nhoods - Sparse matrix of cells x neighbourhoods.
- sample_counts - Matrix of neighbourhoods x samples with the cell counts per neighbourhood and sample.
- spatial_dist - Numeric. The kth nearest neighbour distance per neighbourhood index cell.
- nhood_overlap - The overlap between the neighbourhoods.
- params - List. The parameters used, plus the embedding and the index cells.

References

Dann, et al., Nat Biotechnol, 2022

Examples

```
# neighbourhood counts across six synthetic samples
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 500L,
    n_genes = 50L,
    n_samples = 6L,
    sample_bias = "even"
  )
)
milo <- get_miloR_abundances_sc(
  sc,
  sample_id_col = "sample_id",
  miloR_params = params_sc_miloR(k_refine = 10L),
  .verbose = FALSE
)
dim(milo$sample_counts)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

get_model_fit *Get the fitted model*

Description

[**Deprecated**] The neighbourhood test moved into Rust, so there is no `DGEGLM` to hand back any more. Everything the fit was consulted for now sits in the results table, see `get_differential_abundance_res()`.

Usage

```
get_model_fit(x)

## S3 method for class 'miloR'
get_model_fit(x)
```

Arguments

`x`	An object from which to get the fitted model from.

Value

`NULL`, with a deprecation warning.

get_modules *Get the module membership from a BulkModuleResult*

Description

Returns the data.table of gene to module_id assignments. The exact columns depend on the method that produced the result (CoReMo adds `sign` and `stability`; NMF/ICA/DGRDL add `loading`, `sign` and the thresholding score; Leiden adds only `module_id`).

`gene` is **not** a unique key for the matrix factorisation methods. ICA, NMF and DGRDL assign membership by keeping the tails of each component's loading distribution, so a gene loading strongly on three components appears in three rows, and a gene in no tail appears in none. That is the point of a factorisation. The partition-based methods (CoReMo, Leiden) do emit one row per gene. Do not assume uniqueness without checking `method`.

Usage

```
get_modules(object)
```

Arguments

`object`	A `BulkModuleResult`.

Value

data.table with at minimum `gene` and `module_id` columns. One row per (gene, module) pair.

Examples

```
# gene to module assignments from an NMF fit
syn <- synthetic_bulk_cor_matrix()
mat <- log1p(t(syn$counts))
meta <- data.table::data.table(sample_id = rownames(mat))
object <- BulkCoExp(raw_data = mat, meta_data = meta)
object <- preprocess_bulk_coexp(object, hvg = 500L, .verbose = FALSE)
object <- nmf_bulk(object, k = 3L, .verbose = FALSE)
res <- S7::prop(object, "final_results")
head(get_modules(res))
```

---

get_nmf_gene_loadings

Get the NMF gene loadings

Description

Getter function to extract the gene loadings matrix (features x k) from a bulk NMF fit stored in `BulkCoExp()`. If no NMF fit is present, returns `NULL` with a warning.

Usage

```
get_nmf_gene_loadings(object)
```

Arguments

`object`	The class, see `BulkCoExp()`.

Value

A features x k numeric matrix (if found) or `NULL`.

Examples

```
# gene loadings of a four-factor fit
syn <- generate_gene_module_data(n_samples = 24L, n_genes = 60L)
# NMF needs a non-negative matrix
mat <- syn$data - min(syn$data)
obj <- BulkCoExp(mat, syn$meta_data)
obj <- preprocess_bulk_coexp(
  obj, hvg = NULL, scaling = FALSE, .verbose = FALSE
)
obj <- nmf_bulk(obj, k = 4L, .verbose = FALSE)
dim(get_nmf_gene_loadings(obj))
```

get_nmf_modules *Get the NMF module membership data.table*

Description

Getter function to extract the gene-to-module data.table from a bulk NMF fit. Each row is one gene assigned to its top-loading module.

Usage

```
get_nmf_modules(object)
```

Arguments

object	The class, see BulkCoExp().

Value

A data.table with gene, module_id, loading, sign columns (if found) or NULL.

Examples

```
# gene to module assignments of a four-factor fit
syn <- generate_gene_module_data(n_samples = 24L, n_genes = 60L)
# NMF needs a non-negative matrix
mat <- syn$data - min(syn$data)
obj <- BulkCoExp(mat, syn$meta_data)
obj <- preprocess_bulk_coexp(
  obj, hvg = NULL, scaling = FALSE, .verbose = FALSE
)
obj <- nmf_bulk(obj, k = 4L, .verbose = FALSE)
head(get_nmf_modules(obj))
```

get_nmf_sample_activity *Get the NMF sample activity*

Description

Getter function to extract the sample activity matrix (samples x k) from a bulk NMF fit. If no NMF fit is present, returns NULL with a warning.

Usage

```
get_nmf_sample_activity(object)
```

Arguments

object	The class, see BulkCoExp().

Value

A samples x k numeric matrix (if found) or NULL.

Examples

```
# sample activity of a four-factor fit
syn <- generate_gene_module_data(n_samples = 24L, n_genes = 60L)
# NMF needs a non-negative matrix
mat <- syn$data - min(syn$data)
obj <- BulkCoExp(mat, syn$meta_data)
obj <- preprocess_bulk_coexp(
  obj, hvg = NULL, scaling = FALSE, .verbose = FALSE
)
obj <- nmf_bulk(obj, k = 4L, .verbose = FALSE)
dim(get_nmf_sample_activity(obj))
```

---

get_nmf_stability *Get the multi-run NMF diagnostics*

Description

Getter function for the diagnostics of a multi-run NMF fit. For a stabilised_nmf_bulk() fit that is the per-run losses, convergence flags and the best-run index. For a consensus_nmf_bulk() fit it is the cluster stability, the relative errors and the per-component clustering table. Returns NULL for a single-run fit.

Usage

```
get_nmf_stability(object)
```

Arguments

object	The class, see BulkCoExp().

Value

For a stabilised fit, a list with losses, converged and best_idx. For a consensus fit, a list with stability, rel_error, rel_run_errors, clusters, cluster_sizes, n_dropped and n_empty_clusters. NULL if neither was run.

Examples

```
# clustering diagnostics of a consensus fit
syn <- generate_gene_module_data(n_samples = 24L, n_genes = 60L)
# NMF needs a non-negative matrix
mat <- syn$data - min(syn$data)
obj <- BulkCoExp(mat, syn$meta_data)
obj <- preprocess_bulk_coexp(
  obj, hvg = NULL, scaling = FALSE, .verbose = FALSE
)
obj <- consensus_nmf_bulk(obj, k = 4L, n_runs = 10L, .verbose = FALSE)
get_nmf_stability(obj)$cluster_sizes
```

---

get_ontology_ancestry

Return ancestry terms from an ontology

Description

This function will return all ancestors and descendants based on a provided data.table with parent-child terms

Usage

```
get_ontology_ancestry(parent_child_dt)
```

Arguments

`parent_child_dt`
: data.table. The data.table with column parent and child.

Value

A list with

- ancestors A list with all ancestor terms.
- descendants A list with all descendant terms.

Examples

```
# ancestors and descendants of every term in a toy ontology
onto <- data.table::data.table(
  parent = c("a", "b", "b", "b", "c"),
  child = c("b", "c", "d", "e", "f")
)
ancestry <- get_ontology_ancestry(onto)
ancestry$ancestors[["f"]]
```

get_outputs *Return the outputs*

Description

Getter function to extract the outputs from the `BulkCoExp()` or `BulkDge()`.

Usage

```
get_outputs(object, ...)
```

Arguments

`object`	The underlying object, either `BulkCoExp()` or `BulkDge()`.
`...`	Additional arguments to parse to the functions.

Value

Returns the outputs stored in the class.

Examples

```
# everything the QC and normalisation steps stashed on the object
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- qc_bulk_dge(object, group_col = "case_control", .verbose = FALSE)
names(get_outputs(object))
```

get_params *Get the parameters that were used.*

Description

Extracts parameters from the `BixverseBaseClass` class (or child classes) and has options to return (pretty) JSONs. This generic also gets inherited by other classes and can be used to extract parameters. Also, can dispatch to specific methods for certain S3 classes.

Usage

```
get_params(object, to_json = FALSE, pretty_json = FALSE)

get_params.LdaResult(object, ...)

get_params.LdaKSweepResult(object, ...)

get_params.Hotspot(object, to_json = FALSE, pretty_json = FALSE)

get_params.miloR(object, to_json = FALSE, pretty_json = FALSE)

get_params.ScenicGrn(object, to_json = FALSE, pretty_json = FALSE)

get_params.NmfResult(object, to_json = FALSE, pretty_json = FALSE)

get_params.StabilisedNmfResult(object, to_json = FALSE, pretty_json = FALSE)

get_params.ConsensusNmfResult(object, to_json = FALSE, pretty_json = FALSE)

get_params.DialogueResult(object, ...)

get_params.ScNebula(object, to_json = FALSE, pretty_json = FALSE)
```

Arguments

`object`	A class within bixverse that inherits from `BixverseBaseClass()` or defined S3 classes.
`to_json`	Shall the params be returned as a JSON string.
`pretty_json`	Shall the params be returned as a pretty JSON string.
`...`	Unused, present so the S3 methods sharing this page match the generic.

Value

Depending on parameters either the R list or a (pretty) JSON string.

Examples

```
# parameters stored in a freshly created class
object <- SimilarityNetworkFusion(snf_params = params_snf(k = 3L))
names(get_params(object))
get_params(object, to_json = TRUE, pretty_json = TRUE)
```

`get_pca_factors` *Get the PCA factors*

Description

Returns the PCA factors (sample-based scores). This function is used for the single cell-related classes and methods.

Usage

```
get_pca_factors(x, ...)

## S7 method for class <bixverse::MetaCells>
get_pca_factors(x, ...)

## S3 method for class 'ScCache'
get_pca_factors(x, ...)

## S7 method for class <bixverse::SingleCells>
get_pca_factors(x, ...)

## S7 method for class <bixverse::SingleCellsSubset>
get_pca_factors(x, ...)
```

Arguments

`x`	An object to get PCA factors from.
`...`	Other parameters.

Value

The PCA factors from the object (if found).

Examples

```
# cells x PCs
sc <- demo_single_cells()
dim(get_pca_factors(sc))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`get_pca_loadings` *Get the PCA loadings*

Description

Returns the PCA loadings (feature-based scores). This function is used for the single cell-related classes and methods.

Usage

```
get_pca_loadings(x, ...)

## S7 method for class <bixverse::MetaCells>
get_pca_loadings(x, ...)

## S3 method for class 'ScCache'
get_pca_loadings(x, ...)

## S7 method for class <bixverse::SingleCells>
get_pca_loadings(x, ...)

## S7 method for class <bixverse::SingleCellsSubset>
get_pca_loadings(x, ...)
```

Arguments

`x`	An object to get PCA loadings from.
`...`	Other parameters.

Value

The PCA feature loadings from the object (if found).

Examples

```
# HVGs x PCs
sc <- demo_single_cells()
dim(get_pca_loadings(sc))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`get_pca_singular_val` *Get the PCA singular values*

Description

Returns the PCA singular values (can be useful to assess cumulative variance explained). This function is used for the single cell-related classes and methods.

Usage

```
get_pca_singular_val(x, ...)

## S7 method for class <bixverse::MetaCells>
get_pca_singular_val(x, ...)

## S3 method for class 'ScCache'
get_pca_singular_val(x, ...)

## S7 method for class <bixverse::SingleCells>
get_pca_singular_val(x, ...)

## S7 method for class <bixverse::SingleCellsSubset>
get_pca_singular_val(x, ...)
```

Arguments

`x`	An object to get PCA singular values from.
`...`	Other parameters.

Value

The PCA singular values from the object (if found).

Examples

```
# singular values, largest first
sc <- demo_single_cells()
head(get_pca_singular_val(sc), 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`get_pseudobulked_sc` *Generate pseudo-bulked matrices*

Description

This is a wrapper function to generate pseudo-bulked counts from a list of cell identifiers. You have the option to return sparse or dense matrices and if you wish to sum raw counts or calculate the mean of the normalised counts.

Usage

```
get_pseudobulked_sc(
  object,
  cell_list,
  return_format = c("dense", "sparse"),
  assay = c("raw", "norm"),
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`cell_list`	Named list.
`return_format`	String. One of `c("dense", "sparse")`.
`assay`	String. One of `c("raw", "norm")`.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

Pending on your setting in return_format a dense matrix or sparse CSR matrix with aggregated cells x genes.

Examples

```
# sum raw counts per planted cell type
sc <- demo_single_cells()
obs <- get_sc_obs(sc)
res <- get_pseudobulked_sc(
  sc,
  cell_list = split(obs$cell_id, obs$cell_grp),
  .verbose = FALSE
)
dim(res)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

get_rbh_res *Get the RBH results*

Description

Pulls out the RBH results if you ran `generate_rbh_graph()`

Usage

```
get_rbh_res(object)
```

Arguments

`object`	The underlying class `RbhGraph()`.

Value

The data.table with the RBH result if found, otherwise NULL.

Examples

```
# reciprocal best hits between modules of two data sets
set.seed(123)
modules <- data.table::data.table(
  origin = rep(c("set_a", "set_b"), each = 20),
  module = rep(c("m1", "m2", "m3", "m4"), each = 10),
  gene = unlist(replicate(4, sample(letters, 10), simplify = FALSE))
)
object <- RbhGraph(
  modules,
  rbh_type = "set",
  dataset_col = "origin",
  module_col = "module",
  value_col = "gene"
)
object <- generate_rbh_graph(object, minimum_similarity = 0)
head(get_rbh_res(object))
```

get_residual_fit *Get the fitted residual model*

Description

Returns the `ScResidualFit` written by `fit_residuals_sc()`. This function is used for the single cell-related classes and methods.

Warns and returns `NULL` when nothing was fitted, so it is safe to use as a presence probe. The functions that compute with the fit assert instead.

Usage

```
get_residual_fit(x, ...)

## S7 method for class <bixverse::MetaCells>
get_residual_fit(x, ...)

## S3 method for class 'ScCache'
get_residual_fit(x, ...)

## S7 method for class <bixverse::SingleCells>
get_residual_fit(x, ...)

## S7 method for class <bixverse::SingleCellsSubset>
get_residual_fit(x, ...)
```

Arguments

x	An object to get the fitted model from.
...	Other parameters.

Value

The `ScResidualFit` object, or `NULL` when nothing was fitted.

Examples

```
# the fitted model and the genes it covers
sc <- demo_single_cells()
sc <- fit_residuals_sc(sc, method = "analytic_pearson", .verbose = FALSE)
length(get_residual_fit(sc)$genes)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_resolution_res *Return the resolution results*

Description

Getter function to get the resolution results (if available).

Usage

```
get_resolution_res(object)
```

Arguments

object	The class, see `BulkCoExp()`.

Value

If resolution results were found, returns the data.table. Otherwise, throws a warning and returns NULL.

Examples

```
# cluster counts and modularity across the Leiden resolution sweep
syn <- synthetic_bulk_cor_matrix()
mat <- log1p(t(syn$counts))
meta <- data.table::data.table(sample_id = rownames(mat))
object <- BulkCoExp(raw_data = mat, meta_data = meta)
object <- preprocess_bulk_coexp(object, hvg = 200L, .verbose = FALSE)
object <- cor_module_processing(
  object,
  cor_method = "pearson",
  .verbose = FALSE
)
object <- cor_module_graph_check_res(
  object,
  parallel = FALSE,
  .verbose = FALSE
)
head(get_resolution_res(object))
```

---

get_results *Get the final results from the class*

Description

Get the final results from `BixverseBaseClass` class (or child classes).

Usage

```
get_results(object)

get_results.DialogueResult(object, ...)
```

Arguments

`object`	The underlying `BixverseBaseClass()` class. The class functionality is usually inherited by other S7 classes in `bixverse`.
`...`	Unused, present so the S3 methods sharing this page match the generic.

Value

Returns the final results if any have been stored in the class.

Examples

```
# communities found after a network diffusion
set.seed(42)
g <- igraph::sample_pa(15, directed = FALSE)
edges <- data.table::setDT(igraph::as_data_frame(g))[, `:=`(
  from = sprintf("node_%i", from),
  to = sprintf("node_%i", to)
)]
object <- NetworkDiffusions(edges, weighted = FALSE, directed = FALSE)
object <- diffuse_seed_nodes(object, c(node_1 = 1, node_3 = 1), "max")
object <- permute_seed_nodes(object, perm_iters = 100L, .verbose = FALSE)
object <- community_detection(
  object,
  community_params = params_community_detection(
    min_seed_nodes = 0L,
    min_nodes = 2L
  )
)
head(get_results(object))
```

---

get_sc_available_features

Returns the available features for single cell applications

Description

Returns a data.table with available features in the obs table and in the count matrices.

Usage

```
get_sc_available_features(object)
```

Arguments

`object`	`SingleCells`, `MetaCells` (or potentially other) class.

Value

A data.table with available features.

Examples

```
# what can be queried from the obs table and the counts
sc <- demo_single_cells(prepped = FALSE)
head(get_sc_available_features(sc), 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`get_sc_cache_status` *Status of everything held in a single cell object's caches*

Description

One row per cached artefact, saying whether it still agrees with the current cell set and with the artefacts it was derived from. Artefacts computed before provenance stamping existed report `stamped = FALSE` and are never flagged stale, because nothing is known about them either way.

Usage

```
get_sc_cache_status(object)
```

Arguments

`object`	`SingleCells`, `SingleCellsSubset`, `MetaCells` or `SingleCellsMultiModal` class.

Value

A `data.table` with the columns

- modality - The modality the artefact lives in.
- artefact - One of `pca`, `embedding`, `knn`, `snn`, `magic`.
- name - The embedding name, `NA` for the others.
- stamped - Whether the artefact carries a provenance stamp.
- stale - Whether it disagrees with the current state.
- reason - Why it is stale, `NA` otherwise.
- id - The artefact's stamp id.
- from - List column of the parent stamp ids.

Examples

```
# what the object holds and whether it still agrees with the cells
sc <- demo_single_cells()
get_sc_cache_status(sc)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

get_sc_counts *Getter the counts*

Description

Getter the counts

Usage

```
get_sc_counts(
  object,
  assay = c("raw", "norm"),
  return_format = c("cell", "gene"),
  cell_indices = NULL,
  gene_indices = NULL,
  use_cells_to_keep = TRUE,
  modality = c("rna", "adt"),
  .verbose = TRUE
)
```

Arguments

object	SingleCells, MetaCells, SingleCellsMultiModal class.
assay	String. Which slot to return. One of c("raw", "norm"). Defaults to "raw".
return_format	String. One of c("cell", "gene"). Return data in cell-centric compressed format (CSR) or gene-centric compressed format (CSC). Defaults to "cell". Not relevant for MetaCells.
cell_indices	Optional cell indices.
gene_indices	Optional gene indices.
use_cells_to_keep	Boolean. Shall cells to keep be found in the class, shall the counts be reduced to these. Not relevant for MetaCells.
modality	String. The modality to return. One of c("rna", "adt").
.verbose	Boolean. Controls verbosity of the function.

Value

The counts table

Examples

```
# raw counts for the first ten genes, cell-centric (CSR)
sc <- demo_single_cells(prepped = FALSE)
counts <- get_sc_counts(sc, gene_indices = 1:10, .verbose = FALSE)
dim(counts)
```

```
unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`get_sc_obs` *Getter the obs table*

Description

Getter the obs table

Usage

```
get_sc_obs(object, indices = NULL, cols = NULL, filtered = FALSE)
```

Arguments

`object`	`SingleCells`, `MetaCells`, `SingleCellsMultiModal` class.
`indices`	Optional integer vector. The integer positions of the cells to return.
`cols`	Optional string vector. The columns from the obs table to return.
`filtered`	Boolean. Whether to return all cells or filtered to `to_keep` cells. Not relevant for `MetaCells`.

Value

The obs table

Examples

```
# the obs table, restricted to a few columns
sc <- demo_single_cells(prepped = FALSE)
head(get_sc_obs(sc, cols = c("cell_id", "cell_grp", "lib_size")), 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`get_sc_var` *Getter the var table*

Description

Getter the var table

Usage

```
get_sc_var(object, indices = NULL, cols = NULL, modality = c("rna", "adt"))
```

Arguments

object	SingleCells, MetaCells, SingleCellsMultiModal class.
indices	Optional integer vector. The integer positions of the genes to return.
cols	Optional string vector. The columns from the var table to return.
modality	String. The modality to return. One of c("rna", "adt").

Value

The vars table

Examples

```
# the per gene statistics the HVG step wrote
sc <- demo_single_cells()
head(get_sc_var(sc, cols = c("gene_id", "mean", "var_std")), 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_scores *Get scores*

Description

Get scores

Usage

```
get_scores(x, ...)

## S3 method for class 'ScDblFinderRes'
get_scores(x, ..., score_type = c("weighted", "cxds_scores"))

## S3 method for class 'ScTypeResults'
get_scores(x, ...)

## S3 method for class 'ScTypeCellResults'
get_scores(x, ...)
```

Arguments

x	An object to get scores from.
...	Additional arguments passed to methods.
score_type	Either "weighted" or "cxds_scores".

Value

The score matrix or data.table held by the object, depending on the class. Methods exist for `ScTypeResults`, `ScTypeCellResults` and `ScDblFinderRes`.

A numeric matrix of cells x cell types.

A numeric vector with the winning score per cell.

Examples

```
# the ScType score matrix of cell types by cluster
sc <- demo_single_cells()
markers <- data.table::data.table(
  cell_type = rep(sprintf("cell_type_%i", 1:3), each = 10),
  gene_id = sprintf("gene_%02d", 1:30)
)
cell_markers <- prepare_cell_markers(obj = sc, marker_df = markers)
res <- calc_sc_type_scores(sc, cell_marker_list = cell_markers,
                           .verbose = FALSE)
dim(get_scores(res))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`get_sim_matrix` *Get the similarity matrix*

Description

Get the similarity matrix

Usage

```
get_sim_matrix(object, as_data_table = FALSE, .verbose = TRUE)
```

Arguments

`object`	`OntologySim class`. See `OntologySim()`.
`as_data_table`	Boolean. Shall the data be returned as a long data.table.
`.verbose`	Boolean. Controls verbosity of the function.

Value

Returns the semantic similarity data.table from the class

Examples

```
# the Wang similarity matrix back out of the class
onto <- data.table::data.table(
  parent = c("a", "b", "b", "b", "c"),
  child = c("b", "c", "d", "e", "f"),
  type = c("part_of", "part_of", "part_of", "is_a", "is_a")
)
onto_obj <- OntologySim(onto, .verbose = FALSE)
onto_obj <- calculate_wang_sim_onto(
  onto_obj,
  weights = c(part_of = 0.8, is_a = 0.6),
  .verbose = FALSE
)
round(get_sim_matrix(onto_obj, .verbose = FALSE), 3)
```

---

get_snf_adjcacency_mat

Get an individual affinity matrix

Description

Get an individual affinity matrix

Usage

```
get_snf_adjcacency_mat(object, name)
```

Arguments

object	The underlying class SimilarityNetworkFusion().
name	String. The name of the individual data modality affinity matrix to return.

Value

Returns adjcacency matrix if found.

Examples

```
# affinity matrix of a single modality
set.seed(42)
continuous <- matrix(rnorm(120), nrow = 12, ncol = 10)
rownames(continuous) <- sprintf("sample_%02i", 1:12)
colnames(continuous) <- sprintf("feature_%i", 1:10)
object <- SimilarityNetworkFusion(
  data = continuous,
  data_name = "continuous",
  snf_params = params_snf(k = 3L)
)
dim(get_snf_adjcacency_mat(object, name = "continuous"))
```

get_snf_final_mat *Get the final SNF matrix*

Description

Get the final SNF matrix

Usage

```
get_snf_final_mat(object)
```

Arguments

object	The underlying class SimilarityNetworkFusion().

Value

Returns the SNF adjacency/similarity matrix.

Examples

```
# fused similarity matrix across two modalities
set.seed(42)
continuous <- matrix(rnorm(120), nrow = 12, ncol = 10)
rownames(continuous) <- sprintf("sample_%02i", 1:12)
colnames(continuous) <- sprintf("feature_%i", 1:10)
clinical <- data.table::data.table(
  sample_id = rownames(continuous),
  sex = factor(sample(c("M", "F"), 12, replace = TRUE)),
  stage = factor(sample(c("I", "II", "III"), 12, replace = TRUE))
)
object <- SimilarityNetworkFusion(
  data = continuous,
  data_name = "continuous",
  snf_params = params_snf(k = 3L)
)
object <- add_snf_data_modality(object, clinical, data_name = "clinical")
object <- run_snf(object)
dim(get_snf_final_mat(object))
```

get_snf_params *Get the SNF params*

Description

Get the SNF params

Usage

```
get_snf_params(object)
```

Arguments

object	The underlying class SimilarityNetworkFusion().

Value

Returns the stored SNF params

Examples

```
# the SNF parameters stored in an empty class
object <- SimilarityNetworkFusion(snf_params = params_snf(k = 3L))
get_snf_params(object)$k
```

---

get_snn_graph *Get the sNN graph*

Description

Returns the shared nearest neighbour graph from the object. This function is used for the single cell-related classes and methods.

Usage

```
get_snn_graph(x, ...)

## S7 method for class <bixverse::MetaCells>
get_snn_graph(x, ...)

## S3 method for class 'ScCache'
get_snn_graph(x, ...)

## S7 method for class <bixverse::SingleCells>
get_snn_graph(x, ...)

## S7 method for class <bixverse::SingleCellsSubset>
get_snn_graph(x, ...)
```

Arguments

x	An object to get the sNN graph from.
...	Other parameters.

Value

The igraph that has the shared nearest neighbours.

Examples

```
# the sNN graph the clustering methods run on
sc <- demo_single_cells()
igraph::vcount(get_snn_graph(sc))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

get_stability *Get the consensus NMF stability diagnostics*

Description

Returns the mean silhouette of the consensus clusters, the relative reconstruction errors and the per-component clustering table. A stability near 1 means every restart found the same programmes; a low one means the factorisation is not reproducible at this `k`.

Usage

```
get_stability(x)

## S3 method for class 'ConsensusNmfResult'
get_stability(x)
```

Arguments

`x`	An object holding consensus NMF results.

Value

A list with `stability`, `rel_error`, `rel_run_errors`, `clusters`, `cluster_sizes`, `n_dropped` and `n_empty_clusters`.

Examples

```
# how much the restarts agreed on the consensus factors
sc <- demo_single_cells()
res <- consensus_nmf_sc(
  sc,
  k = 5L,
  n_runs = 5L,
  nmf_consensus_params = params_nmf_consensus(density_threshold = 2),
  .verbose = FALSE
)
get_stability(res)$stability

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`get_symphony_hvg_names` *Getter for the HVG gene names of a Symphony reference*

Description

Getter for the HVG gene names of a Symphony reference

Usage

```
get_symphony_hvg_names(object)
```

Arguments

`object`	`SymphonyReference` class.

Value

Character vector of HVG gene names in reference loading order.

Examples

```
# the HVG names in reference loading order
ref <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 500L,
    n_genes = 50L,
    n_batches = 2L
  )
)
symphony_ref <- build_symphony_ref(
  ref,
  batch_column = "batch_index",
  hvg = get_hvg(ref) + 1L,
  harmony_params = params_sc_harmony(k = 10L),
  no_pcs = 10L,
  label_columns = "cell_grp",
  .verbose = FALSE
)
head(get_symphony_hvg_names(symphony_ref))

unlink(ref@dir_data, recursive = TRUE, force = TRUE)
```

`get_symphony_labels` *Getter for the stored labels of a Symphony reference*

Description

Getter for the stored labels of a Symphony reference

Usage

```
get_symphony_labels(object)
```

Arguments

`object`	`SymphonyReference` class.

Value

A `data.table` of reference cell labels in `z_corr` row order, or `NULL` if no labels are stored.

Examples

```
# the cell labels snapshotted at build time
ref <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 500L,
    n_genes = 50L,
    n_batches = 2L
  )
)
symphony_ref <- build_symphony_ref(
  ref,
  batch_column = "batch_index",
  hvg = get_hvg(ref) + 1L,
  harmony_params = params_sc_harmony(k = 10L),
  no_pcs = 10L,
  label_columns = "cell_grp",
  .verbose = FALSE
)
head(get_symphony_labels(symphony_ref))

unlink(ref@dir_data, recursive = TRUE, force = TRUE)
```

`get_symphony_loadings` *Getter for the PCA loadings of a Symphony reference*

Description

Getter for the PCA loadings of a Symphony reference

Usage

```
get_symphony_loadings(object)
```

Arguments

`object`	`SymphonyReference` class.

Value

The PCA gene loadings matrix (n_hvgs x d).

Examples

```
# the PCA gene loadings the query gets projected through
ref <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 500L,
    n_genes = 50L,
    n_batches = 2L
  )
)
symphony_ref <- build_symphony_ref(
  ref,
  batch_column = "batch_index",
  hvg = get_hvg(ref) + 1L,
  harmony_params = params_sc_harmony(k = 10L),
  no_pcs = 10L,
  label_columns = "cell_grp",
  .verbose = FALSE
)
dim(get_symphony_loadings(symphony_ref))

unlink(ref@dir_data, recursive = TRUE, force = TRUE)
```

`get_symphony_z_corr` *Getter for the corrected embedding of a Symphony reference*

Description

Getter for the corrected embedding of a Symphony reference

Usage

```
get_symphony_z_corr(object)
```

Arguments

`object`	`SymphonyReference` class.

Value

The post-Harmony corrected embedding (N x d).

Examples

```
# the Harmony corrected reference embedding
ref <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 500L,
    n_genes = 50L,
    n_batches = 2L
  )
)
symphony_ref <- build_symphony_ref(
  ref,
  batch_column = "batch_index",
  hvg = get_hvg(ref) + 1L,
  harmony_params = params_sc_harmony(k = 10L),
  no_pcs = 10L,
  label_columns = "cell_grp",
  .verbose = FALSE
)
dim(get_symphony_z_corr(symphony_ref))

unlink(ref@dir_data, recursive = TRUE, force = TRUE)
```

`get_tf_to_gene` *Extract the TF to gene data from the ScenicGrn object*

Description

Extract the TF to gene data from the ScenicGrn object

Usage

```
get_tf_to_gene(x)

## S3 method for class 'ScenicGrn'
get_tf_to_gene(x)
```

Arguments

`x`	`ScenicGrn` object from which to extract the TF to gene data.table.

Value

data.table with TF to gene information

Examples

```
# the TF to gene links that survived the importance filter
sc <- demo_single_cells()
grn <- scenic_grn_sc(
  sc,
  tf_ids = sprintf("gene_%02d", 1:5),
  scenic_params = params_scenic(
    min_counts = 1L,
    learner_params = list(n_trees = 20L)
  ),
  .verbose = FALSE
)
grn <- identify_tf_to_genes(
  grn,
  method = "top_k",
  k_tfs = 3L,
  .verbose = FALSE
)
head(get_tf_to_gene(grn))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

get_top_terms *Get the highest-probability terms per topic*

Description

The interpretability surface of a topic model: for each topic, the terms carrying the most probability mass. Note that these are probabilities within a topic, so they are comparable down a topic but not across topics of different breadth.

Usage

```
get_top_terms(x, n = 20L)

## S3 method for class 'LdaResult'
get_top_terms(x, n = 20L)
```

Arguments

x	`LdaResult` object.
n	Integer. Number of terms to return per topic.

Value

A data.table with `topic`, `rank`, `term` and `probability`, sorted by topic and then rank.

Examples

```
# the five highest probability terms of each topic
set.seed(42L)
corpus <- matrix(rbinom(200L * 40L, 1L, 0.05), nrow = 200L, ncol = 40L)
corpus[1:100, 1:10] <- rbinom(1000L, 1L, 0.6)
corpus[101:200, 11:20] <- rbinom(1000L, 1L, 0.6)
colnames(corpus) <- sprintf("term_%02d", 1:40)
lda_res <- run_lda(corpus > 0, k = 2L, .verbose = FALSE)
get_top_terms(lda_res, n = 5L)
```

get_tpm_counts *Return the TPM-normalised counts*

Description

Getter function to extract the TPM-normalised counts from the `BulkDge()` class.

Usage

```
get_tpm_counts(object)
```

Arguments

object	BulkDge class.

Value

Returns the TPM-normalised counts. (If found.)

Examples

```
# TPM counts, which normalise_bulk_dge() only computes on request
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
gene_lengths <- stats::setNames(
  rep(2000, nrow(syn$counts)),
  rownames(syn$counts)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- qc_bulk_dge(object, group_col = "case_control", .verbose = FALSE)
object <- normalise_bulk_dge(
  object,
  group_col = "case_control",
  calc_tpm = TRUE,
  gene_lengths = gene_lengths,
  .verbose = FALSE
)
get_tpm_counts(object)[1:3, 1:3]
```

---

get_w *Get the W (gene loadings) matrix*

Description

Get the W (gene loadings) matrix

Usage

```
get_w(x)

## S3 method for class 'NmfResult'
get_w(x)

## S3 method for class 'StabilisedNmfResult'
get_w(x)

## S3 method for class 'ConsensusNmfResult'
get_w(x)
```

Arguments

x	An object holding NMF results.

Examples

```
# gene loadings of a five factor NMF
sc <- demo_single_cells()
res <- nmf_sc(sc, k = 5L, .verbose = FALSE)
dim(get_w(res))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

gse_go_elim_method *Run gene ontology enrichment with elimination method.*

Description

This method takes the GeneOntologyElim and a target gene set and performs an GSE enrichment leveraging ontological information. It starts at the lowest levels of the ontology and tests if there is significant enrichment for any GO terms. If the threshold of the p-value is below the elimination threshold, the genes from this term will be removed from all its ancestors. The function then proceeds to the next level of the ontology and repeats the process. The gene universe will be automatically set to every gene represented in the ontology.

Usage

```
gse_go_elim_method(
  object,
  target_genes,
  minimum_overlap = 3L,
  fdr_threshold = 0.05,
  elim_threshold = 0.05,
  min_genes = NULL
)
```

Arguments

object	The underlying class, see GeneOntologyElim().
target_genes	String. The target genes you wish to apply the GSEA over.
minimum_overlap	Integer. Threshold for the minimal overlap.
fdr_threshold	Float. Threshold for maximum fdr to include in the output.
elim_threshold	Float. Threshold from which p-value onwards the elimination on the ancestors shall be conducted.

`min_genes` Integer. Minimum number of genes that have to be included in the gene ontology term. If NULL, it will default to the number of minimum genes stored in `GeneOntologyElim`.

Value

data.table with enrichment results.

Examples

```
# human GO terms with at least 25 genes
go_obj <- GeneOntologyElim(
  get_go_data_human(.verbose = FALSE),
  min_genes = 25L
)
target_genes <- unique(unlist(S7::prop(go_obj, "go_to_genes")[1:5]))
head(gse_go_elim_method(go_obj, target_genes = target_genes), 3)
```

---

gse_go_elim_method_list

Run gene ontology enrichment with elimination method over a list.

Description

This method takes the GeneOntologyElim and a list of target gene sets and performs an GSE enrichment leveraging ontological information. It starts at the lowest levels of the ontology and tests if there is significant enrichment for any GO terms. If the threshold of the p-value is below the elimination threshold, the genes from this term will be removed from all its ancestors. The function then proceeds to the next level of the ontology and repeats the process. The class will leverage Rust threading to parallelise the process. The gene universe will be automatically set to every gene represented in the ontology.

Usage

```
gse_go_elim_method_list(
  object,
  target_gene_list,
  minimum_overlap = 3L,
  fdr_threshold = 0.05,
  elim_threshold = 0.05,
  min_genes = NULL
)
```

Arguments

`object`	The underlying class, see `GeneOntologyElim()`.
`target_gene_list`	List. The target genes list you wish to apply the gene set enrichment analysis over.
`minimum_overlap`	Integer. Threshold for the minimal overlap.
`fdr_threshold`	Float. Threshold for maximum fdr to include in the output.
`elim_threshold`	Float. Threshold from which p-value onwards the elimination on the ancestors shall be conducted.
`min_genes`	Integer. Minimum number of genes that have to be included in the gene ontology term. If NULL, it will default to the number of minimum genes stored in `GeneOntologyElim`.

Value

data.table with enrichment results.

Examples

```
# human GO terms with at least 25 genes
go_obj <- GeneOntologyElim(
  get_go_data_human(.verbose = FALSE),
  min_genes = 25L
)
target_genes <- unique(unlist(S7::prop(go_obj, "go_to_genes")[1:5]))
res <- gse_go_elim_method_list(
  go_obj,
  target_gene_list = list(
    set_a = target_genes,
    set_b = rev(target_genes)[1:50]
  )
)
head(res, 3)
```

`gse_hypergeometric` *Gene set enrichment (GSE) based on a hypergeometric test.*

Description

Takes a set of target genes, a list of gene sets and calculates a p-value (hypergeometric test) and odds ratio (OR) against all the gene sets. Also applies a multiple hypothesis correction (BH) to the p-values.

Usage

```
gse_hypergeometric(
  target_genes,
  gene_set_list,
  gene_universe = NULL,
  threshold = 0.05,
  minimum_overlap = 3L,
  .verbose = FALSE
)
```

Arguments

`target_genes`	Character vector. GeneID(s) of the target genes.
`gene_set_list`	Named list of character vectors. Names should represent the gene sets, pathways, and the elements the genes within the respective gene set.
`gene_universe`	Optional character vector. If you would like to specify specifically the gene universe. If set to NULL, the function will default to all represented genes in the `gene_set_list`.
`threshold`	Float between 0 and 1 to filter on the fdr. Default: 0.05. If 1 everything is returned.
`minimum_overlap`	Number of minimum overlap between the target genes and the respective gene set.
`.verbose`	Boolean. Controls verbosity of the function.

Value

data.table with enrichment results.

Examples

```
# hypergeometric test of a target set against a small universe
gene_universe <- sprintf("gene_%03i", 1:200)
gene_sets <- list(
  set_a = gene_universe[1:20],
  set_b = gene_universe[15:40],
  set_c = gene_universe[100:130]
)
target <- gene_universe[c(1:12, 150:158)]
gse_hypergeometric(target, gene_sets, gene_universe, threshold = 1)
```

`gse_hypergeometric_list` *Gene set enrichment (GSE) based on a hypergeometric test over a list.*

Description

Takes a set of list of target genes, a list of gene sets and calculates a p-value (hypergeometric test) and odds ratio (OR) against all the gene sets. Also applies a multiple hypothesis correction (BH) to the p-values.

Usage

```
gse_hypergeometric_list(
  target_genes_list,
  gene_set_list,
  gene_universe = NULL,
  threshold = 0.05,
  minimum_overlap = 3L,
  .verbose = FALSE
)
```

Arguments

`target_genes_list`	Named list of character vectors. Names should represent the identifiers of the target genes and the elements the genes.
`gene_set_list`	Named list of character vectors. Names should represent the gene sets, pathways, and the elements the genes within the respective gene set.
`gene_universe`	Optional character vector. If you would like to specify specifically the gene universe. If set to NULL, the function will default to all represented genes in the `gene_set_list`.
`threshold`	Float between 0 and 1 to filter on the fdr. Default: 0.05. If NULL everything is returned.
`minimum_overlap`	Number of minimum overlap between the target genes and the respective gene set.
`.verbose`	Boolean. Controls verbosity of the function.

Value

data.table with enrichment results.

Examples

```
# two target sets tested against the same gene sets in one call
gene_universe <- sprintf("gene_%03i", 1:200)
gene_sets <- list(
  set_a = gene_universe[1:20],
  set_b = gene_universe[100:130]
)
targets <- list(
  hit_a = gene_universe[c(1:12, 150:158)],
  hit_b = gene_universe[c(100:112, 5:8)]
)
gse_hypergeometric_list(targets, gene_sets, gene_universe, threshold = 1)
```

---

harmony_sc *Run Harmony*

Description

A version of Harmony by Korsunsky et al., implemented in Rust. Performs batch correction on PCA embeddings and stores the result as a `"harmony"` embedding in the object.

Usage

```
harmony_sc(
  object,
  batch_column,
  additional_batch_columns = NULL,
  modality = c("rna", "adt"),
  harmony_params = params_sc_harmony(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`batch_column`	String. Column name in the object containing the primary batch labels.
`additional_batch_columns`	Optional character vector. Additional batch columns to regress out. If `NULL`, only the primary batch column is used.
`modality`	String. One of `c("rna", "adt")`. You can only use `"adt"` on `SingleCellsMultiModal` class.
`harmony_params`	List. Output of `params_sc_harmony()`.
`seed`	Integer. For reproducibility.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

The object with a "harmony" embedding added. If no PCA embeddings are found, returns the object unchanged with a warning.

Examples

```
# Harmony correction of the PCA embedding
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 600L, n_genes = 50L, n_batches = 3L
  )
)
sc <- harmony_sc(sc, batch_column = "batch_index", .verbose = FALSE)
dim(get_embedding(sc, "harmony"))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

harmony_v2_sc *Run Harmony v2*

Description

A version of Harmony v2 by Patikas et al., 2026, implemented in Rust. Performs batch correction on PCA embeddings and stores the result as a "harmony_v2" embedding in the object.

Usage

```
harmony_v2_sc(
  object,
  batch_column,
  additional_batch_columns = NULL,
  modality = c("rna", "adt"),
  harmony_params = params_sc_harmony_v2(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

object	SingleCells or SingleCellsSubset class.
batch_column	String. Column name in the object containing the primary batch labels.
additional_batch_columns	Optional character vector. Additional batch columns to regress out. If NULL, only the primary batch column is used.
modality	String. One of c("rna", "adt"). You can only use "adt" on SingleCellsMultiModal class.

harmony_params
: List. Output of params_sc_harmony_v2().

seed
: Integer. For reproducibility.

.verbose
: Boolean or integer. Controls verbosity and returns run times. FALSE -> quiet, TRUE or 1L -> normal verbosity, 2L -> detailed verbosity.

Value

The object with a "harmony_v2" embedding added. If no PCA embeddings are found, returns the object unchanged with a warning.

Examples

```
# the reimplemented Harmony, writing its own embedding
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 600L, n_genes = 50L, n_batches = 3L
  )
)
sc <- harmony_v2_sc(
  sc,
  batch_column = "batch_index",
  .verbose = FALSE
)
dim(get_embedding(sc, "harmony_v2"))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

hedges_g_dge *Calculate the effect size*

Description

Calculate the effect size

Usage

```
hedges_g_dge(
  meta_data,
  main_contrast,
  normalised_counts,
  contrast_list = NULL,
  .verbose = TRUE
)
```

Arguments

meta_data	data.table. The meta information about the experiment in which the contrast info can be found.
main_contrast	String. Which column contains the main groups you want to calculate the Hedge's G effect for. Every permutation of the groups will be tested if contrast_list is NULL.
normalised_counts	Numeric Matrix. The normalised count matrix.
contrast_list	String vector or NULL. Optional string vector of contrast formatted as "contrast1-contrast2". Default NULL will create all contrasts automatically.
.verbose	Boolean. Controls the verbosity of the function.

Value

A data.table with the effect sizes and standard errors based on the Hedge's G effect size for the groups.

Examples

```
# Hedge's G on log CPM counts for the case vs control contrast
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
norm_counts <- rs_cpm(syn$counts, lib_size = NULL, log = TRUE,
  prior_count = 2)
res <- hedges_g_dge(
  meta_data = meta,
  main_contrast = "case_control",
  normalised_counts = norm_counts,
  .verbose = FALSE
)
head(res)
```

---

hotspot_autocor_sc *Calculate the local auto-correlation of a gene*

Description

This method implements the HotSpot approach (see DeTomaso, et al.) to calculate the auto-correlation of a given gene in the kNN graph based on the chosen embedding. This can be used to identify genes that have strong local correlations and vary across the kNN graph.

Usage

```
hotspot_autocor_sc(
  object,
  embd_to_use = "pca",
  use_knn = TRUE,
  hotspot_params = params_sc_hotspot(),
  no_embd_to_use = NULL,
  cells_to_take = NULL,
  genes_to_take = NULL,
  streaming = NULL,
  random_seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `MetaCells` class.
`embd_to_use`	String. The embedding to use. Defaults to `"pca"`.
`use_knn`	Boolean. Shall the internal kNN be used. If set to yes, you need to ensure consistency. If you provide `cells_to_take`, the function will regenerate the kNN graph with these cells.
`hotspot_params`	List with hotspot parameters, see `params_sc_hotspot()` with the following elements: • model - String. Which of the available models to use for the gene expression. Choices are one of `c("danb", "normal", "bernoulli")`. • normalise - Boolean. Shall the data be normalised. • weighted_graph - Boolean. Shall the Gaussian kernel be applied to the neighbour distances. If `FALSE`, every retained edge weighs one. • neighborhood_factor - Float. Kernel width for `weighted_graph`. • knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.
`no_embd_to_use`	Optional integer. Number of embedding dimensions to use. If `NULL` all will be used.
`cells_to_take`	Optional string vector. If you want to only use selected cells. If `NULL` will default to all cells_to_keep in the class.
`genes_to_take`	Optional string vector. If you wish to limit the search to a subset of genes. If `NULL` will default to all genes in the class.
`streaming`	Optional Boolean. Shall the data be streamed in. Useful for larger data sets where you wish to avoid loading in the whole data. If `NULL`, will automatically detect. Ignored for `MetaCells`, which are held in memory.
`random_seed`	Integer. Used for reproducibility.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Details

Should a gene not be found in sufficient cells, the gene will be automatically filtered out from the results. This can occur for example if you have filtered out the cells that contain a given gene. The underlying genes are still available, but the cells that might contain them are not included.

Whether the neighbour distances need squaring before the kernel sees them follows from the metric. With `use_knn = TRUE` it is taken from the metric stored on the cached kNN graph, otherwise from `ann_dist` in `hotspot_params`.

Value

A data.table with the auto-correlations on a per gene basis and various statistics.

Examples

```
# local auto-correlation of every gene on the cached kNN graph
sc <- demo_single_cells()
res <- hotspot_autocor_sc(sc, .verbose = FALSE)
head(res, 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`hotspot_gene_cor_sc` *Calculate the local pairwise gene-gene correlation*

Description

This method implements the HotSpot approach (see DeTomaso, et al.) to calculate the local gene-gene correlations and their Z-scores.

Usage

```
hotspot_gene_cor_sc(
  object,
  embd_to_use = "pca",
  use_knn = TRUE,
  hotspot_params = params_sc_hotspot(),
  no_embd_to_use = NULL,
  cells_to_take = NULL,
  genes_to_take = NULL,
  streaming = NULL,
  working_mem_gb = 4,
  random_seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `MetaCells` class.
`embd_to_use`	String. The embedding to use. Defaults to `"pca"`.
`use_knn`	Boolean. Shall the internal kNN be used. If set to yes, you need to ensure consistency. If you provide `cells_to_take`, the function will regenerate the kNN graph with these cells.
`hotspot_params`	List with hotspot parameters, see `params_sc_hotspot()` with the following elements: • model - String. Which of the available models to use for the gene expression. Choices are one of `c("danb", "normal", "bernoulli")`. • normalise - Boolean. Shall the data be normalised. • weighted_graph - Boolean. Shall the Gaussian kernel be applied to the neighbour distances. If `FALSE`, every retained edge weighs one. • neighborhood_factor - Float. Kernel width for `weighted_graph`. • knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.
`no_embd_to_use`	Optional integer. Number of embedding dimensions to use. If `NULL` all will be used.
`cells_to_take`	Optional string vector. If you want to only use selected cells. If `NULL` will default to all cells_to_keep in the class.
`genes_to_take`	Optional string vector. If you wish to limit the search to a subset of genes. If `NULL` will default to all genes in the class.
`streaming`	Optional Boolean. Shall the data be streamed in. Useful for larger data sets where you wish to avoid loading in the whole data. If `NULL`, will automatically detect. Ignored for `MetaCells`, which are held in memory.
`working_mem_gb`	Numeric. Approximate working memory (GB) the streaming pair path may use for resident gene panels. Ignored when `streaming` is `FALSE`. Larger values mean fewer disk re-reads. Note this excludes the two dense N_genes x N_genes output matrices, which scale with `genes_to_use`. Defaults to `4` (4 GB of memory allocated).
`random_seed`	Integer. Used for reproducibility.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Details

Should a gene not be found in sufficient cells, the pairs with this gene will be set to 0. Please ensure prior to running the function that you are only calculating gene-gene autocorrelations that occur in sufficient cells.

Whether the neighbour distances need squaring before the kernel sees them follows from the metric. With `use_knn = TRUE` it is taken from the metric stored on the cached kNN graph, otherwise from `ann_dist` in `hotspot_params`.

Value

A `sc_hotspot` class that can be used for subsequent analysis.

Examples

```
# local gene-gene correlations over a subset of the genes
sc <- demo_single_cells()
res <- hotspot_gene_cor_sc(
  sc,
  genes_to_take = get_gene_names(sc)[1:20],
  .verbose = FALSE
)
res

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

ica_evaluate_comp *Iterate over different ncomp parameters for ICA*

Description

This function allows to iterate over a vector of ncomp to identify which ncomp parameter to choose for your data set. The idea is to generate stability profiles over the different ncomps and identify a 'sweet spot' of good stability, low redundancy (maximum absolute correlation between the component loadings) and good convergence of the identified independent components

Usage

```
ica_evaluate_comp(
  object,
  ica_type = c("logcosh", "exp"),
  iter_params = params_ica_randomisation(),
  ncomp_params = params_ica_ncomp(),
  ica_params = params_ica_general(),
  random_seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	The class, see `BulkCoExp()`. You need to apply `ica_processing()` before running this function.
`ica_type`	String, element of `c("logcosh", "exp")`.
`iter_params`	List. This list controls the randomisation parameters for the ICA runs, see `params_ica_randomisation()` for estimating stability. Has the following elements:

- cross_validate - Boolean. Shall the data be split into different chunks on which ICA is run. This will slow down the function substantially, as every chunk needs to whitened again.
- random_init - Integer. How many random initialisations shall be used for the ICA runs.
- folds - If `cross_validate` is set to `TRUE` how many chunks shall be used. To note, you will run per ncomp random_init * fold ICA runs which can quickly increase.

`ncomp_params` List. Parameters for the ncomp to iterate through, see `params_ica_ncomp()`. In the standard setting, `c(2, 3, 4, 5)` will be tested and then in steps until max_no_comp will be tested, i.e., `c(2, 3, 4, 5, 10, 15, ..., max_no_comp - 5, max_no_comp)`.

- max_no_comp - Maximum number of ncomp to test.
- steps - Integer. In which steps to move from 5 onwards.
- custom_seq - An integer vector. If you wish to provide a custom version of no_comp to iterate through.

`ica_params` List. The ICA parameters, see `params_ica_general()` wrapper function. This function generates a list containing:

- maxit - Integer. Maximum number of iterations for ICA.
- alpha - Float. The alpha parameter for the logcosh version of ICA. Should be between 1 to 2.
- max_tol - Maximum tolerance of the algorithm.
- verbose - Controls verbosity of the function.

`random_seed` Integer. For reproducibility.

`.verbose` Boolean. Controls verbosity.

Value

`BulkCoExp` with the added information of stability of the components and other data to plot to choose the right `ncomp`.

Examples

```
# stability across a small grid of component counts
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- ica_processing(obj, .verbose = FALSE)
obj <- ica_evaluate_comp(
  obj,
  ica_type = "logcosh",
  ncomp_params = params_ica_ncomp(custom_seq = seq(2L, 20L, by = 2L)),
  .verbose = FALSE
)
head(get_ica_stability_res(obj))
```

`ica_optimal_ncomp` *Identify stability inflection point*

Description

This function can be used after having run `ica_evaluate_comp()`. It will calculate the inflection point, based on the first derivative of a loess function fitted `ncomp ~ combined_score` with the combined score being a product of the median stability, `1 - max_abs_loading_cor` and proportion of convergence and add these info to the object. Should the loess function raise a warning (e.g., singularity), the class will be returned as is and manual determination of optimal ncomp is warranted. Additionally, you have the option to plot the loess function for additional control over the span parameter (defaults to `TRUE`).

Usage

```
ica_optimal_ncomp(object, span = 0.2, show_plot = TRUE, .verbose = TRUE)
```

Arguments

`object`	The class, see `BulkCoExp()`. You need to apply `ica_evaluate_comp()` before running this function.
`span`	Float. A value between 0.1 and 1.0 for the loess function defining the span. Defaults to `0.2`.
`show_plot`	Boolean. Shall a plot be shown of the loess function fitted. Defaults to `TRUE`.
`.verbose`	Boolean. Controls the verbosity of the function.

Value

`BulkCoExp` with optimal ncomp based on the inflection point method.

Examples

```
# inflection point of the combined score
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- ica_processing(obj, .verbose = FALSE)
obj <- ica_evaluate_comp(
  obj,
  ica_type = "logcosh",
  ncomp_params = params_ica_ncomp(custom_seq = seq(2L, 20L, by = 2L)),
  .verbose = FALSE
)
obj <- ica_optimal_ncomp(
  obj, span = 0.4, show_plot = FALSE, .verbose = FALSE
)
obj@params$ica_stability_assessment$optimal_ncomp
```

ica_processing *Prepare class for ICA*

Description

This is the generic function for doing the necessary preprocessing for running independent component analysis.

Usage

```
ica_processing(object, fast_svd = TRUE, random_seed = 123L, .verbose = TRUE)
```

Arguments

object	The class, see BulkCoExp(). Ideally, you should run preprocess_bulk_coexp() before applying this function.
fast_svd	Boolean. Shall randomised SVD be used for the whitening. This is faster and usually causes little precision loss.
random_seed	Integer. Seed for the randomised SVD. Only relevant, if fast_svd = TRUE.
.verbose	Boolean. Controls verbosity of the function.

Value

BulkCoExp with the needed data for ICA in the properties of the class.

Examples

```
# whitening ahead of the ICA runs
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- ica_processing(obj, .verbose = FALSE)
dim(obj@processed_data$K)
```

ica_stabilised_results *Run stabilised ICA with a given number of components*

Description

This function runs stabilised ICA with the defined number of components.

Usage

```
ica_stabilised_results(
  object,
  no_comp = NULL,
  ica_type = c("logcosh", "exp"),
  iter_params = params_ica_randomisation(),
  ica_params = params_ica_general(),
  membership_params = params_module_membership(),
  random_seed = 42L,
  consistent_sign = TRUE,
  .verbose = TRUE
)
```

Arguments

`object` The class, see `BulkCoExp()`. You need to apply `ica_processing()` before running this function.

`no_comp` Optional integer. Number of components you wish to use for the ICA run. If you have run `ica_evaluate_comp()` the optimal number is identified via the elbow method and will be used if set to `NULL`. You can overwrite this however.

`ica_type` String, element of `c("logcosh", "exp")`.

`iter_params` List. This list controls the randomisation parameters for the ICA runs, see `params_ica_randomisation()` for estimating stability. Has the following elements:

- cross_validate - Boolean. Shall the data be split into different chunks on which ICA is run. This will slow down the function substantially, as every chunk needs to whitened again.
- random_init - Integer. How many random initialisations shall be used for the ICA runs.
- folds - If `cross_validate` is set to `TRUE` how many chunks shall be used. To note, you will run per ncomp random_init * fold ICA runs which can quickly increase.

`ica_params` List. The ICA parameters, see `params_ica_general()` wrapper function. This function generates a list containing:

- maxit - Integer. Maximum number of iterations for ICA.
- alpha - Float. The alpha parameter for the logcosh version of ICA. Should be between 1 to 2.
- max_tol - Maximum tolerance of the algorithm.
- verbose - Controls verbosity of the function.

`membership_params` List. Controls how the component loadings are turned into module membership, see `params_module_membership()`. Membership is not exclusive: a gene loading strongly on several components will appear in several modules, and a gene in no tail appears in none.

`random_seed` Integer. For reproducibility.

`consistent_sign`
Boolean. If set to `TRUE`, for each source the absolute maximum value will be positive, i.e., the sign will be inverted so that the absolute bigger tail is set to positive floats.

`.verbose` Boolean. Controls verbosity.

Value

`BulkCoExp` with the the source matrix S, mixing matrix A and other parameters added to the slots. The diagnostics carry `loading_condition_number` (condition number of S S') and `max_abs_loading_cor` (largest absolute correlation between two component loadings). High values mean near-duplicate components and an unstable sample activity A; reduce `no_comp`.

Examples

```
# stabilised five-component ICA and the resulting modules
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- ica_processing(obj, .verbose = FALSE)
obj <- ica_stabilised_results(
  obj, no_comp = 5L, ica_type = "logcosh", .verbose = FALSE
)
head(get_modules(get_results(obj)))
```

`identify_tf_to_genes` *Identify the TF to gene regulation*

Description

Generates the TF to gene associations from the importance matrix produced by the tree-based regression models. Two filtering strategies are available:

- **Threshold** (`method = "threshold"`): For each gene, computes mean + `n_sd` * SD of the importance scores across all TFs and retains only pairs exceeding that threshold. This adapts to the per-gene importance distribution and is less sensitive to differences between learners.
- **Top-k** (`method = "top_k"`): Selects the top `k_tfs` TFs per gene and/or the top `k_genes` genes per TF. Both margins can be combined (union). At least one of `k_tfs` or `k_genes` must be provided.

Both methods accept an optional `min_importance` floor.

Usage

```
identify_tf_to_genes(
  x,
  method = c("threshold", "top_k"),
  k_tfs = NULL,
  k_genes = NULL,
  n_sd = 2,
  min_importance = NULL,
  .verbose = TRUE
)

## S3 method for class 'ScenicGrn'
identify_tf_to_genes(
  x,
  method = c("threshold", "top_k"),
  k_tfs = NULL,
  k_genes = NULL,
  n_sd = 2,
  min_importance = NULL,
  .verbose = TRUE
)
```

Arguments

x	ScenicGrn object.
method	Character. Either "top_k" or "threshold".
k_tfs	Optional integer. Top TFs per gene (only used when method = "top_k").
k_genes	Optional integer. Top genes per TF (only used when method = "top_k").
n_sd	Numeric. Number of standard deviations above the per-gene mean to use as the threshold (only used when method = "threshold"). Default is 2.
min_importance	Optional numeric in [0, 1]. Absolute minimum importance score for inclusion.
.verbose	Boolean. Controls verbosity of the function.

Value

The ScenicGrn object with a tf_to_gene_results data.table added.

Examples

```
# keep the three strongest TFs per gene
sc <- demo_single_cells()
grn <- scenic_grn_sc(
  sc,
  tf_ids = sprintf("gene_%02d", 1:5),
  scenic_params = params_scenic(
    min_counts = 1L,
```

```
    learner_params = list(n_trees = 20L)
  ),
  .verbose = FALSE
)
grn <- identify_tf_to_genes(
  grn,
  method = "top_k",
  k_tfs = 3L,
  .verbose = FALSE
)
head(get_tf_to_gene(grn))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`install_agent_skill` *Install the bixverse agent skill*

Description

bixverse ships a skill that teaches coding agents how to *use* the package: the workflows, the `params_*()` convention, the ordering constraints and the traps. This copies it out of the installed package to where your agent looks for it.

The skill is versioned with the package, so re-run this after upgrading bixverse to keep the agent in sync with the code.

Usage

```
install_agent_skill(
  dest = NULL,
  agent = c("claude", "codex", "generic"),
  overwrite = FALSE
)
```

Arguments

`dest`	String or `NULL`. Directory to install into. `NULL` picks the default for `agent`: `"~/.claude/skills"` for `"claude"` and `"~/.codex/skills"` for `"codex"`. Required for `"generic"`. The skill lands in a `bixverse` subdirectory of this.
`agent`	String. One of `c("claude", "codex", "generic")`. Controls the name of the entry file and whether the frontmatter is kept.
`overwrite`	Boolean. Replace an existing installation. Defaults to `FALSE`, in which case an existing `bixverse` directory causes an error rather than a silent clobber.

Details

The reference files are plain markdown and work anywhere. Only the entry file differs between agents:

- `"claude"` writes `SKILL.md` with the YAML frontmatter Claude Code uses to decide when to load the skill. Discovery is automatic.
- `"codex"` and `"generic"` write `AGENTS.md` with the frontmatter stripped, since no other agent reads it. There is no auto-discovery, so point the agent at the directory yourself.

Value

The path the skill was written to, invisibly.

Examples

```
## Not run:
# Claude Code, into ~/.claude/skills, auto-discovered
install_agent_skill()

# Codex, into ~/.codex/skills as AGENTS.md
install_agent_skill(agent = "codex")

# anywhere else
install_agent_skill(dest = "~/my-agent/context", agent = "generic")

# refresh after a package upgrade
install_agent_skill(overwrite = TRUE)

## End(Not run)
```

---

knn_graph_label_propagation

kNN-based graph label propagation

Description

In case of a kNN graph with a subset of nodes having labels, this function can be used to propagate the labels through the graph. This function uses the label spreading version of the algorithm.

Usage

```
knn_graph_label_propagation(
  from,
  to,
  labels,
  weights = NULL,
  label_prop_params = params_label_propagation()
)
```

Arguments

from	Integer vector. Source node indices for each edge.
to	Integer vector. Target node indices for each edge. Must be the same length as from.
labels	Character vector. The labels with NA indicating unlabelled.
weights	Optional numeric vector. The weights between the nodes. Must be the same length as from. If not provided, all edges are treated as unweighted.
label_prop_params	List. Named list of label propagation parameters as generated by params_label_propagation(). If not provided, defaults will be used.

Value

A list with the following elements:

- assignment_probs - Matrix with the assignment probabilities.
- final_labels - Final labels in the graph.

Examples

```
# spread two labels through a five node kNN graph
from <- as.integer(c(1, 1, 2, 3, 4))
to <- as.integer(c(2, 3, 3, 4, 5))
labels <- c("A", NA, "B", NA, NA)
res <- knn_graph_label_propagation(from = from, to = to, labels = labels)
res$final_labels
```

---

lda_k_sweep *Sweep the topic count for LDA*

Description

Fits one model per requested topic count and scores each with three model selection metrics, then combines them into a single rescaled score. Use it to pick k, then pull the winning model out with get_best_model().

Usage

```
lda_k_sweep(
  x,
  k_range,
  lda_params = params_lda(),
  top_topics_coh = 5L,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

x	Documents x terms matrix. Either a dgCMatrix, a dgRMatrix, or a dense numeric or logical matrix. Column names are required, row names are generated if absent.
k_range	Integer vector. The topic counts to evaluate. Every entry must be at least 2.
lda_params	List, see params_lda().
top_topics_coh	Integer. Number of top-scoring topics averaged into the reported coherence.
seed	Integer. Seed for the variational initialisation.
.verbose	Boolean or integer. Verbosity.

Details

The three metrics disagree by design and are meant to be looked at together. Arun is a symmetric KL between the singular values of the topic-term matrix and the document length distribution (lower is better). Cao Juan is the mean pairwise cosine similarity between topics, so it punishes a k that has started splitting one signal in two (lower is better). Mimno is UMass coherence (higher is better), averaged over the top-scoring topics only, because coherence over all topics is dragged down by the ones that never specialised. combined_score is the min-max rescaled compromise, and best_k is its argmax.

One thing to know before reading best_k: any k below five is struck out of the selection entirely, because coherence saturates on small topic counts and would otherwise always win. That is upstream behaviour, inherited from pycisTopic. So best_k can never come back below five, however good the raw metrics of a smaller k look, and a sweep over 2:6 is really a choice between five and six. The excluded rows carry NA in combined_score, keep their metrics, and trigger a warning. If you suspect a handful of topics, read arun_2010 and cao_juan_2009 off the table yourself and pass the k you want to get_best_model().

Cost is one full fit per k. The corpus is built once and shared, and each fit is already parallel over documents, so the topic counts run sequentially rather than nesting thread pools.

Value

An LdaKSweepResult, which is a data.table with one row per k.

References

Arun, et al., PAKDD, 2010; Cao, et al., Neurocomputing, 2009; Mimno, et al., EMNLP, 2011

Examples

```
# small sweep above the coherence topic count floor
set.seed(42L)
corpus <- matrix(rbinom(200L * 40L, 1L, 0.05), nrow = 200L, ncol = 40L)
corpus[1:100, 1:10] <- rbinom(1000L, 1L, 0.6)
```

```
corpus[101:200, 11:20] <- rbinom(1000L, 1L, 0.6)
colnames(corpus) <- sprintf("term_%02d", 1:40)
sweep_res <- lda_k_sweep(corpus > 0, k_range = 5:7, .verbose = FALSE)
sweep_res
```

`leiden_clustering` *Leiden or Louvain community detection on an igraph*

Description

Modularity-based Leiden (Traag et al.) or Louvain (Blondel et al.) community detection, run in Rust. Mirrors `igraph::cluster_leiden(objective_function = "modularity")`, but the result depends on `seed` only (not on the global RNG or the number of threads). Communities are numbered by size, `1` being the largest.

Usage

```
leiden_clustering(
  graph,
  resolution = 1,
  algorithm = c("leiden", "louvain"),
  beta = 0.01,
  n_iterations = 2L,
  seed = 42L
)
```

Arguments

`graph`	igraph. Undirected. If it has an edge attribute `weight`, it is used, otherwise all edges get weight 1.
`resolution`	Numeric. Modularity resolution parameter. Higher values give more communities.
`algorithm`	String. One of `c("leiden", "louvain")`.
`beta`	Numeric. Randomness of the Leiden refinement merges. Ignored for Louvain.
`n_iterations`	Integer. Number of full iterations, each starting from the previous partition.
`seed`	Integer. Random seed.

Value

Integer vector with the community of each node (1-based, `1` is the largest community).

References

Traag, Waltman & van Eck, Sci Rep, 2019; Blondel, et al., J Stat Mech, 2008

Examples

```
g <- igraph::make_graph("Zachary")
table(leiden_clustering(g))
```

`ligand_activity_scores` *Compute ligand activity scores against gene sets*

Description

For each gene set, ranks ligands by how well their influence vector aligns with set membership across a background. Returns AUROC, AUPR, AUPR corrected against the random baseline, Pearson, and Spearman per ligand and per gene set.

The influence matrix is restricted to `background` columns before scoring, so AUROC / AUPR reflect ranking within the background (typical NicheNet practice: background = genes expressed in the receiver cells, gene set = the DEGs).

Usage

```
ligand_activity_scores(ligand_influence, gene_sets, background = NULL)
```

Arguments

`ligand_influence`	A `LigandTargetInfluence` object.
`gene_sets`	Either a character vector (one gene set) or a list of character vectors. Names of the list are propagated to the output.
`background`	Character vector or `NULL`. Genes against which gene sets are scored. Defaults to all genes in `ligand_influence`. Background members not present in the influence matrix are silently dropped.

Value

A `data.table` with one row per (gene set, ligand) pair and columns `gene_set`, `ligand`, `auroc`, `aupr`, `aupr_corrected`, `pearson`, `spearman`.

Examples

```
# rank the ligands against the TF1 target set
ppi <- data.table::data.table(
  from = c("L1", "SIG1", "L2", "SIG2"),
  to = c("SIG1", "TF1", "SIG2", "TF2"),
  weight = 1.0
)
grn <- data.table::data.table(
  from = rep(c("TF1", "TF2"), each = 3),
  to = c("G1", "G2", "G3", "G4", "G5", "G6"),
  weight = 1.0
```

```
)
inf <- generate_ligand_target_influence(
  ligand_seeds = list(L1 = "L1", L2 = "L2"),
  ppi_network = ppi,
  grn_network = grn,
  params = params_ligand_target(ltf_cutoff = 0)
)
ligand_activity_scores(
  ligand_influence = inf,
  gene_sets = list(set_A = c("G1", "G2", "G3"))
)
```

`load_existing` *Load an existing SingleCells from disk*

Description

Helper function that can load the parameters to access the on-disk stored data into the class.

Usage

```
load_existing(object, .verbose = TRUE)
```

Arguments

`object`	`SingleCells`, `MetaCells` or `SingleCellsMultiModal` class.
`.verbose`	Boolean. Controls verbosity of the function.

Value

The object with added information on the data on disk.

Examples

```
# a fresh handle over a directory written earlier
sc <- demo_single_cells(prepped = FALSE)
dir <- sc@dir_data
sc <- load_existing(SingleCells(dir_data = dir), .verbose = FALSE)
dim(sc)

unlink(dir, recursive = TRUE, force = TRUE)
```

load_go_human_data *Get the Gene Ontology data human*

Description

This function loads in gene ontology data stored in the package. This is for humans only.

Usage

```
load_go_human_data()
```

Value

A list containing:

- go_info - data.table. The gene ontology identifier, name and namespace can be found in this one
- gene_ontology - data.table. The relationships between different gene ontology terms.
- go_to_genes - data.table. The gene ontology term to gene (ensembl id) relationships.

Examples

```
# the human gene ontology tables shipped with the package
go_data <- load_go_human_data()
names(go_data)
head(go_data$go_info)
```

load_h5ad *Load in h5ad to* SingleCells

Description

This function takes an h5ad file and loads the obs and var data into the DuckDB of the `SingleCells` class and the counts into a Rust-binarised format for rapid access. During the reading in of the counts, the log CPM transformation will occur automatically.

Usage

```
load_h5ad(
  object,
  h5_path,
  sc_qc_param = params_sc_min_quality(),
  raw_count_slot = c("auto", "X", "raw.X", "layers.counts"),
  cell_id_col = NULL,
  h5ad_streaming = TRUE,
  csc_mem_gb = NULL,
```

```
  streaming = deprecated(),
  batch_size = deprecated(),
  max_genes_in_memory = deprecated(),
  cell_batch_size = deprecated(),
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`h5_path`	File path to the h5ad object you wish to load in.
`sc_qc_param`	List. Output of `params_sc_min_quality()`. A list with the following elements: • min_unique_genes - Integer. Minimum number of genes to be detected in the cell to be included. • min_lib_size - Integer. Minimum library size in the cell to be included. • min_cells - Integer. Minimum number of cells a gene needs to be detected to be included. • target_size - Float. Target size to normalise to. Defaults to `1e5`.
`raw_count_slot`	Where raw counts live. `"auto"` detects per file via `detect_raw_count_slot()`; otherwise one of `"X"`, `"raw.X"`, `"layers.counts"`.
`cell_id_col`	Optional string. If a specific column in the h5ad obs data is representing the cell identifiers, you can specify it here.
`h5ad_streaming`	Boolean. Stream the h5ad counts into the cell-based binary in batches instead of materialising the filtered matrix first. Recommended for large files. Defaults to `TRUE`.
`csc_mem_gb`	Optional numeric. Memory in GB for the buffers of the cell-to-gene (CSR to CSC) conversion, at 10 bytes per non-zero. `NULL` (default) converts in one pass and holds the whole matrix. Set a cap for large data sets; every extra phase re-reads the cell file once.
`streaming`, `batch_size`, `max_genes_in_memory`, `cell_batch_size`	Replaced by `csc_mem_gb` and ignored. [**Deprecated**]
`.verbose`	Boolean. Controls the verbosity of the function.

Value

It will populate the files on disk and return the class with updated shape information.

Examples

```
# round trip through a sparse h5ad file
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
```

```
f_path <- tempfile(fileext = ".h5ad")
write_h5ad_sc(f_path, data$counts, data$obs, data$var, .verbose = FALSE)
dir_data <- tempfile("sc_h5ad")
dir.create(dir_data, recursive = TRUE)
sc <- load_h5ad(
  object = SingleCells(dir_data = dir_data),
  h5_path = f_path,
  sc_qc_param = params_sc_min_quality(
    min_unique_genes = 5L,
    min_lib_size = 25L,
    min_cells = 5L
  ),
  .verbose = FALSE
)
dim(sc)

unlink(c(f_path, dir_data), recursive = TRUE, force = TRUE)
```

`load_h5ad_norm` *Load in h5ad with normalised counts to* `SingleCells`

Description

This function takes an h5ad file where only normalised counts are available in the X slot and loads the obs and var data into the DuckDB of the `SingleCells` class and the counts into a Rust-binarised format for rapid access. Raw counts are reconstructed from the normalised values using the library sizes stored in a specified obs column.

The reconstruction assumes the normalisation was: `norm = log1p(x / lib_size * target_size)`

Usage

```
load_h5ad_norm(
  object,
  h5_path,
  obs_lib_size_col,
  target_size,
  sc_qc_param = params_sc_min_quality(),
  cell_id_col = NULL,
  csc_mem_gb = NULL,
  streaming = deprecated(),
  batch_size = deprecated(),
  max_genes_in_memory = deprecated(),
  cell_batch_size = deprecated(),
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`h5_path`	File path to the h5ad object you wish to load in.
`obs_lib_size_col`	String. Name of the obs column containing the total counts per cell or spot (e.g. `"nCount_RNA"`).
`target_size`	Numeric. The target size used in the original normalisation (e.g. `1e4`).
`sc_qc_param`	List. Output of `params_sc_min_quality()`.
`cell_id_col`	Optional string. If a specific column in the h5ad obs data represents the cell identifiers, you can specify it here.
`csc_mem_gb`	Optional numeric. Memory in GB for the buffers of the cell-to-gene (CSR to CSC) conversion, at 10 bytes per non-zero. `NULL` (default) converts in one pass and holds the whole matrix. Set a cap for large data sets; every extra phase re-reads the cell file once.
`streaming`, `batch_size`, `max_genes_in_memory`, `cell_batch_size`	Replaced by `csc_mem_gb` and ignored. [**Deprecated**]
`.verbose`	Boolean.

Value

It will populate the files on disk and return the class with updated shape information.

Examples

```
# an h5ad holding log1p(x / lib_size * 1e4), raw counts reconstructed on read
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
lib_size <- Matrix::rowSums(data$counts)
norm_counts <- data$counts
norm_counts@x <- log1p(
  norm_counts@x / rep(lib_size, diff(norm_counts@p)) * 1e4
)
obs <- data.table::copy(data$obs)[, total_counts := lib_size]

f_path <- tempfile(fileext = ".h5ad")
write_h5ad_sc(f_path, norm_counts, obs, data$var, .verbose = FALSE)
dir_data <- tempfile("sc_h5ad_norm")
dir.create(dir_data, recursive = TRUE)
sc <- load_h5ad_norm(
  object = SingleCells(dir_data = dir_data),
  h5_path = f_path,
  obs_lib_size_col = "total_counts",
  target_size = 1e4,
  sc_qc_param = params_sc_min_quality(
    min_unique_genes = 5L,
    min_lib_size = 25L,
    min_cells = 5L
```

```
  ),
  .verbose = FALSE
)
dim(sc)

unlink(c(f_path, dir_data), recursive = TRUE, force = TRUE)
```

`load_mtx` *Load in mtx/plain text files to* `SingleCells`

Description

This is a helper function to load in mtx files and corresponding plain text files. It will automatically filter out low quality cells and only keep high quality cells. Under the hood DucKDB and high performance Rust binary files are being used to store the counts.

Usage

```
load_mtx(
  object,
  sc_mtx_io_param = params_sc_mtx_io(),
  sc_qc_param = params_sc_min_quality(),
  mtx_streaming = TRUE,
  csc_mem_gb = NULL,
  streaming = deprecated(),
  batch_size = deprecated(),
  max_genes_in_memory = deprecated(),
  cell_batch_size = deprecated(),
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`sc_mtx_io_param`	List. Please generate this one via `params_sc_mtx_io()`.
`sc_qc_param`	List. Output of `params_sc_min_quality()`.
`mtx_streaming`	Boolean. Shall the .mtx file ingestion itself be streamed (via temp-file bucketing). Recommended for large mtx files. Defaults to `TRUE`.
`csc_mem_gb`	Optional numeric. Memory in GB for the buffers of the cell-to-gene (CSR to CSC) conversion, at 10 bytes per non-zero. `NULL` (default) converts in one pass and holds the whole matrix. Set a cap for large data sets; every extra phase re-reads the cell file once.
`streaming`, `batch_size`, `max_genes_in_memory`, `cell_batch_size`	Replaced by `csc_mem_gb` and ignored. [**Deprecated**]
`.verbose`	Boolean.

Value

The class with updated shape information.

Examples

```
# read back a CellRanger style .mtx trio
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
dir_src <- tempfile("cellranger")
dir.create(dir_src, recursive = TRUE)
write_cellranger_output(
  dir_src, data$counts, data$obs, data$var,
  rows = "cells", format_type = "csv", .verbose = FALSE
)

dir_data <- tempfile("sc_mtx")
dir.create(dir_data, recursive = TRUE)
sc <- load_mtx(
  object = SingleCells(dir_data = dir_data),
  sc_mtx_io_param = params_sc_mtx_io(
    path_mtx = file.path(dir_src, "matrix.mtx"),
    path_obs = file.path(dir_src, "barcodes.csv"),
    path_var = file.path(dir_src, "features.csv"),
    cells_as_rows = TRUE,
    has_hdr = TRUE
  ),
  sc_qc_param = params_sc_min_quality(
    min_unique_genes = 5L,
    min_lib_size = 25L,
    min_cells = 5L
  ),
  .verbose = FALSE
)
dim(sc)

unlink(c(dir_src, dir_data), recursive = TRUE, force = TRUE)
```

---

load_multi_h5ad *Load multiple h5ad files into a single* SingleCells

---

Description

Takes a pre-scan result from `prescan_h5ad_files()` and loads all files into a single experiment with global gene QC and sequential cell indexing.

Usage

```
load_multi_h5ad(
  object,
  prescan_result,
  sc_qc_param = params_sc_min_quality(),
  cell_id_col = NULL,
  csc_mem_gb = NULL,
  streaming = deprecated(),
  batch_size = deprecated(),
  max_genes_in_memory = deprecated(),
  cell_batch_size = deprecated(),
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`prescan_result`	Output of `prescan_h5ad_files()`.
`sc_qc_param`	List. Output of `params_sc_min_quality()`.
`cell_id_col`	Optional string. Column name for cell identifiers in obs.
`csc_mem_gb`	Optional numeric. Memory in GB for the buffers of the cell-to-gene (CSR to CSC) conversion, at 10 bytes per non-zero. `NULL` (default) converts in one pass and holds the whole matrix. Set a cap for large data sets; every extra phase re-reads the cell file once.
`streaming`, `batch_size`, `max_genes_in_memory`, `cell_batch_size`	Replaced by `csc_mem_gb` and ignored. [**Deprecated**]
`.verbose`	Boolean.

Value

The class with updated shape and populated DuckDB.

Examples

```
# two files into one experiment, cells tagged by exp_id
files <- c(a = tempfile(fileext = ".h5ad"), b = tempfile(fileext = ".h5ad"))
for (i in seq_along(files)) {
  data <- generate_single_cell_test_data(
    syn_data_params = params_sc_synthetic_data(
      n_cells = 200L,
      n_genes = 40L
    ),
    seed = i
  )
  write_h5ad_sc(files[i], data$counts, data$obs, data$var, .verbose = FALSE)
}
tasks <- prescan_h5ad_files(h5_paths = files, .verbose = FALSE)
```

```
dir_data <- tempfile("sc_multi_h5ad")
dir.create(dir_data, recursive = TRUE)
sc <- load_multi_h5ad(
  object = SingleCells(dir_data = dir_data),
  prescan_result = tasks,
  sc_qc_param = params_sc_min_quality(
    min_unique_genes = 5L,
    min_lib_size = 25L,
    min_cells = 5L
  ),
  .verbose = FALSE
)
table(sc[["exp_id"]])

unlink(c(files, dir_data), recursive = TRUE, force = TRUE)
```

`load_multi_mtx` *Load multiple mtx directories into a single* `SingleCells`

Description

Takes the result of `prescan_mtx_dirs()` and loads all inputs into a single experiment with global gene QC and sequential cell indexing. The feature space is the **intersection** of input gene IDs.

Usage

```
load_multi_mtx(
  object,
  prescan_result,
  sc_qc_param = params_sc_min_quality(),
  csc_mem_gb = NULL,
  streaming = deprecated(),
  batch_size = deprecated(),
  max_genes_in_memory = deprecated(),
  cell_batch_size = deprecated(),
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`prescan_result`	Output of `prescan_mtx_dirs()`.
`sc_qc_param`	List. Output of `params_sc_min_quality()`.
`csc_mem_gb`	Optional numeric. Memory in GB for the buffers of the cell-to-gene (CSR to CSC) conversion, at 10 bytes per non-zero. `NULL` (default) converts in one pass and holds the whole matrix. Set a cap for large data sets; every extra phase re-reads the cell file once.

`streaming`, `batch_size`, `max_genes_in_memory`, `cell_batch_size`
: Replaced by `csc_mem_gb` and ignored. [**Deprecated**]

`.verbose`
: Boolean.

Value

The class with updated shape and populated DuckDB.

Examples

```
# two CellRanger directories into one experiment
dirs <- c(tempfile("cr_a"), tempfile("cr_b"))
for (i in seq_along(dirs)) {
  dir.create(dirs[i], recursive = TRUE)
  data <- generate_single_cell_test_data(
    syn_data_params = params_sc_synthetic_data(
      n_cells = 200L,
      n_genes = 40L
    ),
    seed = i
  )
  write_cellranger_output(
    dirs[i], data$counts, data$obs, data$var,
    rows = "cells", format_type = "csv", .verbose = FALSE
  )
}
scan_res <- prescan_mtx_dirs(
  dirs = dirs,
  exp_ids = c("a", "b"),
  cells_as_rows = TRUE,
  has_hdr = TRUE,
  .verbose = FALSE
)

dir_data <- tempfile("sc_multi_mtx")
dir.create(dir_data, recursive = TRUE)
sc <- load_multi_mtx(
  object = SingleCells(dir_data = dir_data),
  prescan_result = scan_res,
  sc_qc_param = params_sc_min_quality(
    min_unique_genes = 5L,
    min_lib_size = 25L,
    min_cells = 5L
  ),
  .verbose = FALSE
)
dim(sc)

unlink(
  c(dirs, dir_data, scan_res$temp_files),
  recursive = TRUE,
  force = TRUE
)
```

`load_multi_tenx_h5` *Load multiple 10x CellRanger h5 files into a single* `SingleCells`

Description

Takes the result of `prescan_tenx_h5_files()` and loads all inputs into a single experiment with global gene QC and sequential cell indexing. The feature space is determined by the prescan (intersection or union of gene ids).

Usage

```
load_multi_tenx_h5(
  object,
  prescan_result,
  sc_qc_param = params_sc_min_quality(),
  csc_mem_gb = NULL,
  streaming = deprecated(),
  batch_size = deprecated(),
  max_genes_in_memory = deprecated(),
  cell_batch_size = deprecated(),
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`prescan_result`	Output of `prescan_tenx_h5_files()`.
`sc_qc_param`	List. Output of `params_sc_min_quality()`.
`csc_mem_gb`	Optional numeric. Memory in GB for the buffers of the cell-to-gene (CSR to CSC) conversion, at 10 bytes per non-zero. `NULL` (default) converts in one pass and holds the whole matrix. Set a cap for large data sets; every extra phase re-reads the cell file once.
`streaming`, `batch_size`, `max_genes_in_memory`, `cell_batch_size`	Replaced by `csc_mem_gb` and ignored. **[Deprecated]**
`.verbose`	Boolean.

Value

The class with updated shape and populated DuckDB.

Examples

```
# two 10x h5 files into one experiment
files <- c(a = tempfile(fileext = ".h5"), b = tempfile(fileext = ".h5"))
for (i in seq_along(files)) {
  data <- generate_single_cell_test_data(
```

```
    syn_data_params = params_sc_synthetic_data(
      n_cells = 200L,
      n_genes = 40L
    ),
    seed = i
  )
  write_tenx_h5_sc(
    f_path = files[i],
    counts = data$counts,
    barcodes = data$obs$cell_id,
    features = data.table::data.table(
      id = data$var$gene_id,
      name = data$var$ensembl_id,
      feature_type = "Gene Expression"
    )
  )
}
scan_res <- prescan_tenx_h5_files(h5_paths = files, .verbose = FALSE)

dir_data <- tempfile("sc_multi_tenx")
dir.create(dir_data, recursive = TRUE)
sc <- load_multi_tenx_h5(
  object = SingleCells(dir_data = dir_data),
  prescan_result = scan_res,
  sc_qc_param = params_sc_min_quality(
    min_unique_genes = 5L,
    min_lib_size = 25L,
    min_cells = 5L
  ),
  .verbose = FALSE
)
dim(sc)

unlink(c(files, dir_data), recursive = TRUE, force = TRUE)
```

`load_r_data` *Load in data directly from R objects.*

Description

This function loads in data directly from R objects. The counts matrix must be a `dgRMatrix` (rows = cells, columns = genes).

Usage

```
load_r_data(
  object,
  counts,
  obs,
  var,
```

```
  sc_qc_param = params_sc_min_quality(),
  csc_mem_gb = NULL,
  streaming = deprecated(),
  batch_size = deprecated(),
  max_genes_in_memory = deprecated(),
  cell_batch_size = deprecated(),
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`counts`	Sparse matrix. The cells represent the rows, the genes the columns. Needs to be a `"dgRMatrix"`.
`obs`	data.table. Cell metadata. Must have one row per cell in the same order as `counts`.
`var`	data.table. Feature metadata. Must have one row per gene in the same order as `counts`.
`sc_qc_param`	List. Output of `params_sc_min_quality()`. A list with the following elements: • min_unique_genes - Integer. Minimum number of genes to be detected in the cell to be included. • min_lib_size - Integer. Minimum library size in the cell to be included. • min_cells - Integer. Minimum number of cells a gene needs to be detected to be included. • target_size - Float. Target size to normalise to. Defaults to `1e5`.
`csc_mem_gb`	Optional numeric. Memory in GB for the buffers of the cell-to-gene (CSR to CSC) conversion, at 10 bytes per non-zero. `NULL` (default) converts in one pass and holds the whole matrix. Set a cap for large data sets; every extra phase re-reads the cell file once.
`streaming`, `batch_size`, `max_genes_in_memory`, `cell_batch_size`	Replaced by `csc_mem_gb` and ignored. [**Deprecated**]
`.verbose`	Boolean. Controls the verbosity of the function.

Value

It will populate the files on disk and return the class with updated shape information.

Examples

```
# straight from a dgRMatrix in memory onto disk
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
dir_data <- tempfile("sc_r_data")
dir.create(dir_data, recursive = TRUE)
```

```
sc <- load_r_data(
  object = SingleCells(dir_data = dir_data),
  counts = data$counts,
  obs = data$obs,
  var = data$var,
  sc_qc_param = params_sc_min_quality(
    min_unique_genes = 5L,
    min_lib_size = 25L,
    min_cells = 5L
  ),
  .verbose = FALSE
)
dim(sc)

unlink(dir_data, recursive = TRUE, force = TRUE)
```

`load_sce` *Load in data from a* `SingleCellExperiment`

Description

Brings a Bioconductor `SingleCellExperiment` into a `SingleCells` object. `colData` becomes the obs table, `rowData` becomes the var table, and the chosen assay goes through the same Rust quality control and normalisation every other loader uses.

The assay has to hold raw counts. Plenty of objects in the wild ship only `logcounts`, and a negative binomial cannot model those, so pick the right one rather than letting the default find whatever is there.

`reducedDims` and `altExps` are not carried over. Run the embedding on this side, and use `SingleCellsMultiModal()` for ADT.

Usage

```
load_sce(
  object,
  sce,
  assay_name = "counts",
  sc_qc_param = params_sc_min_quality(),
  csc_mem_gb = NULL,
  streaming = deprecated(),
  batch_size = deprecated(),
  max_genes_in_memory = deprecated(),
  cell_batch_size = deprecated(),
  .verbose = TRUE
)
```

Arguments

object	SingleCells class.
sce	SingleCellExperiment class you want to transform.
assay_name	String. Which assay holds the raw counts. Defaults to "counts".
sc_qc_param	List. Output of params_sc_min_quality(). A list with the following elements: • min_unique_genes - Integer. Minimum number of genes to be detected in the cell to be included. • min_lib_size - Integer. Minimum library size in the cell to be included. • min_cells - Integer. Minimum number of cells a gene needs to be detected to be included. • target_size - Float. Target size to normalise to. Defaults to 1e5.
csc_mem_gb	Optional numeric. Memory in GB for the buffers of the cell-to-gene (CSR to CSC) conversion, at 10 bytes per non-zero. NULL (default) converts in one pass and holds the whole matrix. Set a cap for large data sets; every extra phase re-reads the cell file once.
streaming, batch_size, max_genes_in_memory, cell_batch_size	Replaced by csc_mem_gb and ignored. [**Deprecated**]
.verbose	Boolean. Controls the verbosity of the function.

Value

It will populate the files on disk and return the class with updated shape information.

Examples

```
# colData becomes obs, rowData becomes var, the counts assay gets normalised
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
sce <- SingleCellExperiment::SingleCellExperiment(
  assays = list(counts = as(Matrix::t(data$counts), "CsparseMatrix")),
  colData = data.frame(data$obs, row.names = data$obs$cell_id),
  rowData = data.frame(data$var, row.names = data$var$gene_id)
)
dir_data <- tempfile("sc_sce")
dir.create(dir_data, recursive = TRUE)
sc <- load_sce(
  object = SingleCells(dir_data = dir_data),
  sce = sce,
  sc_qc_param = params_sc_min_quality(
    min_unique_genes = 5L,
    min_lib_size = 25L,
    min_cells = 5L
  ),
```

```
  .verbose = FALSE
)
dim(sc)

unlink(dir_data, recursive = TRUE, force = TRUE)
```

`load_seurat` *Load in Seurat to* `SingleCells`

Description

This function takes a Seurat object and generates a `SingleCells` class from it. The raw counts are extracted, written to the Rust binary format, and the metadata is loaded into the DuckDB.

Usage

```
load_seurat(
  object,
  seurat,
  sc_qc_param = params_sc_min_quality(),
  csc_mem_gb = NULL,
  streaming = deprecated(),
  batch_size = deprecated(),
  max_genes_in_memory = deprecated(),
  cell_batch_size = deprecated(),
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`seurat`	`Seurat` class you want to transform.
`sc_qc_param`	List. Output of `params_sc_min_quality()`. A list with the following elements: • min_unique_genes - Integer. Minimum number of genes to be detected in the cell to be included. • min_lib_size - Integer. Minimum library size in the cell to be included. • min_cells - Integer. Minimum number of cells a gene needs to be detected to be included. • target_size - Float. Target size to normalise to. Defaults to `1e5`.

`csc_mem_gb` Optional numeric. Memory in GB for the buffers of the cell-to-gene (CSR to CSC) conversion, at 10 bytes per non-zero. `NULL` (default) converts in one pass and holds the whole matrix. Set a cap for large data sets; every extra phase re-reads the cell file once.

`streaming`, `batch_size`, `max_genes_in_memory`, `cell_batch_size` Replaced by `csc_mem_gb` and ignored. **[Deprecated]**

`.verbose` Boolean. Controls the verbosity of the function.

Value

It will populate the files on disk and return the class with updated shape information.

Examples

```
# a Seurat object holds genes x cells, the loader transposes it
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
seurat_obj <- Seurat::CreateSeuratObject(
  counts = Matrix::t(data$counts),
  meta.data = data.frame(data$obs, row.names = data$obs$cell_id)
)
dir_data <- tempfile("sc_seurat")
dir.create(dir_data, recursive = TRUE)
sc <- load_seurat(
  object = SingleCells(dir_data = dir_data),
  seurat = seurat_obj,
  sc_qc_param = params_sc_min_quality(
    min_unique_genes = 5L,
    min_lib_size = 25L,
    min_cells = 5L
  ),
  .verbose = FALSE
)
dim(sc)

unlink(dir_data, recursive = TRUE, force = TRUE)
```

---

load_tenx_h5 *Load in a 10x CellRanger h5 file to* `SingleCells`

Description

Loads the gene-expression modality from a CellRanger v2/v3 h5 file. The counts go into the Rust-binarised format (with log normalisation applied on read) and the barcodes/features into the DuckDB. Non-gene modalities (e.g. Antibody Capture) are filtered out via `feature_type`.

Usage

```
load_tenx_h5(
  object,
  h5_path,
  sc_qc_param = params_sc_min_quality(),
  feature_type = "Gene Expression",
  csc_mem_gb = NULL,
  streaming = deprecated(),
  batch_size = deprecated(),
  max_genes_in_memory = deprecated(),
  cell_batch_size = deprecated(),
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`h5_path`	File path to the 10x h5 file.
`sc_qc_param`	List. Output of `params_sc_min_quality()`.
`feature_type`	String. Modality to keep. Defaults to `"Gene Expression"`. Ignored for v2 (single modality).
`csc_mem_gb`	Optional numeric. Memory in GB for the buffers of the cell-to-gene (CSR to CSC) conversion, at 10 bytes per non-zero. `NULL` (default) converts in one pass and holds the whole matrix. Set a cap for large data sets; every extra phase re-reads the cell file once.
`streaming`, `batch_size`, `max_genes_in_memory`, `cell_batch_size`	Replaced by `csc_mem_gb` and ignored. [**Deprecated**]
`.verbose`	Boolean.

Value

The class with updated shape information.

Examples

```
# read back a CellRanger v3 h5, gene expression only
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
f_path <- tempfile(fileext = ".h5")
write_tenx_h5_sc(
  f_path = f_path,
  counts = data$counts,
  barcodes = data$obs$cell_id,
  features = data.table::data.table(
    id = data$var$gene_id,
    name = data$var$ensembl_id,
    feature_type = "Gene Expression"
```

```
  )
)

dir_data <- tempfile("sc_tenx")
dir.create(dir_data, recursive = TRUE)
sc <- load_tenx_h5(
  object = SingleCells(dir_data = dir_data),
  h5_path = f_path,
  sc_qc_param = params_sc_min_quality(
    min_unique_genes = 5L,
    min_lib_size = 25L,
    min_cells = 5L
  ),
  .verbose = FALSE
)
dim(sc)

unlink(c(f_path, dir_data), recursive = TRUE, force = TRUE)
```

`map_symphony_query` *Map a SingleCells query onto a Symphony reference*

Description

Projects query cells through the reference's PCA loadings and applies the cached MoE batch correction. Gene matching is by name against the reference's stored HVG names; unmatched HVGs become zero columns. Results are attached to the query as embeddings.

Usage

```
map_symphony_query(
  reference,
  query,
  batch_column = NULL,
  additional_batch_columns = NULL,
  params = params_symphony_map(),
  .verbose = TRUE
)
```

Arguments

`reference`	`SymphonyReference`.
`query`	`SingleCells` query.
`batch_column`	Optional string. If `NULL`, no batch correction is applied (z_corr = z_pca).
`additional_batch_columns`	Optional character vector.
`params`	List. Output of `params_symphony_map()`.
`.verbose`	Boolean or integer.

Value

The `query` object with embeddings `"symphony"` (z_corr), `"symphony_pca"` (z_pca) and `"symphony_r"` (soft cluster assignments, transposed to N_q x K).

Examples

```
# project a second data set into the reference embedding
ref <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 500L,
    n_genes = 50L,
    n_batches = 2L
  )
)
symphony_ref <- build_symphony_ref(
  ref,
  batch_column = "batch_index",
  hvg = get_hvg(ref) + 1L,
  harmony_params = params_sc_harmony(k = 10L),
  no_pcs = 10L,
  .verbose = FALSE
)
query <- demo_single_cells(prepped = FALSE, seed = 7L)
query <- map_symphony_query(symphony_ref, query = query, .verbose = FALSE)
dim(get_embedding(query, "symphony"))

unlink(c(ref@dir_data, query@dir_data), recursive = TRUE, force = TRUE)
```

---

meld_sc — *Run MELD signal smoothing for differential abundance estimation*

Description

This function implements MELD for estimating sample-associated density on a cell manifold. The general idea is to smooth a binary sample indicator matrix over the kNN graph via spectral filtering, yielding per-cell likelihood estimates for each sample condition. For further details on the method, please refer to Burkhardt, et al. This function will take a `SingleCells` class and return the smoothed density estimates per cell and condition.

Usage

```
meld_sc(
  object,
  sample_id_col,
  embd_to_use = "pca",
  no_embd_to_use = NULL,
  meld_params = params_meld(),
  seed = 42L,
```

```
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`sample_id_col`	Character. The column in the obs table representing the sample identifier.
`embd_to_use`	Character. The embedding to use for kNN graph construction. Please use the same here as you used to generate the neighbours. Defaults to `"pca"`.
`no_embd_to_use`	Optional integer. If you only want to use a subset of the embedding dimensions.
`meld_params`	A list, please see `params_meld()`. The list has the following parameters: • beta - Numeric. Smoothing strength; larger values produce smoother densities. • offset - Numeric. Shift of the filter centre in the rescaled spectrum. Must be in `[0, 1]`. • order - Numeric. Filter falloff sharpness; larger values approach a square low-pass. • filter - Character. Filter family. One of `c("heat", "laplacian")`. • chebyshev_order - Integer. Number of Chebyshev coefficients. Must be >= 2. • lap_type - Character. Type of Laplacian. One of `c("combinatorial", "normalised")`. • normalise_indicators - Logical. If `TRUE`, indicator columns are divided by their column sum before filtering. • landmark - Logical. Whether to use landmark approximation. Useful for large data sets. • n_landmarks - Integer. Number of landmarks to use. • knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.
`seed`	Integer. Seed for reproducibility.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

A list with:

- raw_scores - The raw MELD scores
- norm_scores - Negative values were clamped to 0 and the rows L1 normalised. This yields probability-like values.

References

Burkhardt, et al. Nat. Biotechnol., 2021.

Examples

```
# per cell sample likelihoods over the kNN graph
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 500L,
    n_genes = 50L,
    n_samples = 6L,
    sample_bias = "even"
  )
)
res <- meld_sc(sc, sample_id_col = "sample_id", .verbose = FALSE)
head(res$norm_scores[, 1:3])

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

merge_meta_cells *Merge meta cell objects into one*

Description

Row-binds several `MetaCells()` objects into a single one. The typical use case is sample-pure meta cells: generate meta cells per patient (see `meta_cells_per_group()`), then merge them so that methods like SCENIC, AUCell or NMF can run across the full set.

The normalised counts are carried over as generated. They are normalised per meta cell, so row-binding leaves them valid. Caches (PCA, kNN, sNN, embeddings) are per source and are dropped; recompute them on the merged object.

Usage

```
merge_meta_cells(
  inputs,
  source_ids = NULL,
  feature_space = c("intersect", "union"),
  prefix_ids = TRUE,
  .verbose = TRUE
)
```

Arguments

`inputs`	List of `MetaCells` objects.
`source_ids`	Optional character vector of the same length as `inputs` with the source (e.g. patient) identifiers. Defaults to `names(inputs)` and falls back to `source_01`, `source_02`, ... Needs to be unique.
`feature_space`	String. One of `c("intersect", "union")`. Controls how differing gene spaces are resolved. With `"union"` genes missing from an input become structural zeros for its meta cells. Irrelevant when all inputs came from the same source object, as their gene spaces are then identical.

prefix_ids	Boolean. Prefix the meta cell identifiers with the source identifier. If FALSE, duplicated identifiers across inputs are an error.
.verbose	Boolean. Controls verbosity of the function.

Value

A single MetaCells object with all meta cells of the inputs and is_merged set to TRUE. The observation table gains a source_id column; original_cell_idx stays in the index space of its own source, which is why methods that resolve it against the source single cell data (calc_diffusion_coordinates(), calc_manifold_metrics()) refuse to run on the result. other_data holds the source identifiers under sources.

Examples

```
# meta cells generated per source, then pooled for downstream methods
sc_1 <- demo_single_cells(seed = 1L)
sc_2 <- demo_single_cells(seed = 2L)
params <- params_sc_bt_metacells(target_no_metacells = 25L)
merged <- merge_meta_cells(
  list(
    donor_1 = generate_bt_meta_cells_sc(sc_1, params, .verbose = FALSE),
    donor_2 = generate_bt_meta_cells_sc(sc_2, params, .verbose = FALSE)
  ),
  .verbose = FALSE
)
merged

unlink(c(sc_1@dir_data, sc_2@dir_data), recursive = TRUE, force = TRUE)
```

---

merge_sc_experiments *Merge multiple* SingleCells *experiments into one*

Description

Merges N existing SingleCells objects into a freshly constructed target object. The feature space of the result is the **intersection** of the input gene sets. Each input's cells_to_keep filter is honoured (i.e. only cells with to_keep = TRUE in the input's obs are carried over).

If renormalise = FALSE, the stored data_norm values are copied through unchanged. This is valid only when all inputs were normalised against the same target_size. If the gene intersection is much smaller than the individual input gene sets, the inherited data_norm becomes a lossy approximation (it was computed against the pre-intersection library size). In that case set renormalise = TRUE to recompute data_norm against the surviving raw counts using sc_qc_param$target_size.

Obs columns are intersected across inputs. The result obs gains an exp_id column. Inputs that already have an exp_id column are rejected. The sc_cache and sc_map of the target are populated fresh; any PCA, kNN, sNN or HVG state on the inputs is not carried over and must be re-run.

Usage

```
merge_sc_experiments(
  target,
  inputs,
  exp_ids,
  renormalise = FALSE,
  sc_qc_param = params_sc_min_quality(),
  csc_mem_gb = NULL,
  streaming = deprecated(),
  batch_size = deprecated(),
  max_genes_in_memory = deprecated(),
  cell_batch_size = deprecated(),
  .verbose = TRUE
)
```

Arguments

`target`	A freshly constructed `SingleCells` pointing at the output directory.
`inputs`	List of `SingleCells` objects to merge. Length >= 2.
`exp_ids`	Character vector of experiment identifiers, one per input. Must be unique.
`renormalise`	Boolean. Whether to recompute `data_norm` against `sc_qc_param$target_size`. Defaults to `FALSE`.
`sc_qc_param`	List. Output of `params_sc_min_quality()`. Only `target_size` is consulted here; no QC filtering is applied during merge.
`csc_mem_gb`	Optional numeric. Memory in GB for the buffers of the cell-to-gene (CSR to CSC) conversion, at 10 bytes per non-zero. `NULL` (default) converts in one pass and holds the whole matrix. Set a cap for large data sets; every extra phase re-reads the cell file once.
`streaming`, `batch_size`, `max_genes_in_memory`, `cell_batch_size`	Replaced by `csc_mem_gb` and ignored. **[Deprecated]**
`.verbose`	Boolean.

Value

The populated target `SingleCells`.

Examples

```
# two synthetic experiments merged over their shared gene space
sc_a <- demo_single_cells(prepped = FALSE, seed = 1L)
sc_b <- demo_single_cells(prepped = FALSE, seed = 2L)
merged_dir <- tempfile("bixverse_merged")
dir.create(merged_dir)

merged <- merge_sc_experiments(
  target = SingleCells(dir_data = merged_dir),
  inputs = list(sc_a, sc_b),
```

```
  exp_ids = c("exp_a", "exp_b"),
  .verbose = FALSE
)
table(unlist(merged[["exp_id"]]))

unlink(
  c(sc_a@dir_data, sc_b@dir_data, merged_dir),
  recursive = TRUE,
  force = TRUE
)
```

`merge_subset_obs` *Merge obs columns from subsets back into the parent object*

Description

Takes columns computed on one or more `SingleCellsSubset()` objects and writes them into the parent's obs table in the DuckDB, joining on `cell_idx`. The join is a left join, so every parent cell that is not part of any of the provided subsets ends up as `NA`.

Pass a single subset or a list of them, e.g. the output of `apply_pipeline_per_group()`. All subsets are written in a single join, and the function refuses to run if two subsets claim the same cell.

Usage

```
merge_subset_obs(
  object,
  subsets,
  cols = NULL,
  new_names = NULL,
  prefix_values = FALSE,
  overwrite = FALSE,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class. The parent the subsets were derived from.
`subsets`	A `SingleCellsSubset` or a list of them. Every element must originate from `object`; this is verified against the cell names.
`cols`	Optional character vector. Columns in the subset obs tables to merge. If `NULL`, every column that is present in all subsets but absent from the parent obs table is taken, which after a pipeline run is exactly the set of newly generated columns.
`new_names`	Optional character vector. Names to give the merged columns in the parent obs table. Same length as `cols`, which must be given explicitly if this is used.

`prefix_values`	Boolean. Prefix every merged value with the subset's group, i.e. `"<group>_<value>"`. Coerces the columns to character, so this only makes sense for discrete labels. Needed when merging a list of subsets into a shared column, since sub-cluster `1` of one group and sub-cluster `1` of another are otherwise indistinguishable.
`overwrite`	Boolean. Allow merged columns to replace existing columns of the same name in the parent obs table. Defaults to `FALSE`, in which case a name clash is an error.
`.verbose`	Boolean. Controls verbosity.

Value

The parent `SingleCells` object with the merged columns added to the obs table in the DuckDB.

See Also

`add_sc_new_obs()` for the equivalent on result objects that carry their own `cell_idx`, such as `fast_cluster_sc()`.

Examples

```
# write a column computed on the subset back onto the parent
sc <- demo_single_cells(prepped = FALSE)
subset_obj <- SingleCellsSubset(
  sc_object = sc,
  grouping_column = "cell_grp",
  group = "cell_type_1"
)
subset_obj[["sub_label"]] <- rep("a", dim(subset_obj)[1])
sc <- merge_subset_obs(sc, subset_obj, cols = "sub_label", .verbose = FALSE)
table(get_sc_obs(sc)$sub_label, useNA = "ifany")

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`meta_cells_per_group` *Generate source-pure meta cells and merge them*

Description

Splits `object` by `group_col`, optionally runs `pipeline` on each subset, generates meta cells per group and merges the results into a single `MetaCells()` object. This gives you meta cells that never mix cells from two patients/samples, while still being one object you can run SCENIC, AUCell or NMF over.

The meta cell generators need an embedding, so `pipeline` will normally be `step_hvg_sc() %>>% step_pca_sc() %>>% step_neighbours_sc()` unless every subset already carries one.

Usage

```
meta_cells_per_group(
  object,
  group_col,
  method = c("bootstrapped", "seacells", "supercells"),
  mc_params = list(),
  pipeline = NULL,
  groups = NULL,
  feature_space = c("intersect", "union"),
  .verbose = TRUE
)
```

Arguments

object	SingleCells.
group_col	String. Column in obs used to split.
method	String. One of c("bootstrapped", "seacells", "supercells"). Picks the meta cell generator.
mc_params	Named list. Arguments passed on to the generator, e.g. list(sc_meta_cell_params = params_sc_bt_metacells(), target_size = 1e5).
pipeline	Optional ScPipeline applied to each subset before the meta cells are generated.
groups	Optional character vector. Restrict to these group values; if NULL, all unique values of group_col are used.
feature_space	String. One of c("intersect", "union"). Passed to merge_meta_cells(). Irrelevant here as all groups share the gene space of the parent object.
.verbose	Boolean. Controls verbosity of the function.

Value

A merged MetaCells() object with a source_id column in its observation table.

Examples

```
# meta cells that never mix two cell groups
sc <- demo_single_cells(prepped = FALSE)
prep <- step_hvg_sc(hvg_no = 30L, .verbose = FALSE) %>>%
  step_pca_sc(no_pcs = 10L, .verbose = FALSE) %>>%
  step_neighbours_sc(.verbose = FALSE)
meta_cells_per_group(
  object = sc,
  group_col = "cell_grp",
  method = "bootstrapped",
  mc_params = list(
    sc_meta_cell_params = params_sc_bt_metacells(target_no_metacells = 10L),
    .verbose = FALSE
  ),
  pipeline = prep,
```

```
  .verbose = FALSE
)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`MetaCells` *bixverse meta cell class*

Description

This is the `bixverse`-based meta cell class. It is generated by running meta-cell generating functions on top of the main class. Due to the aggregation of the count matrices to a way smaller number of meta cells, the data is held in memory for this class specifically. Additional methods that make specific usage of meta cells (especially co-expression module detection ones will be made available).

Usage

```
MetaCells(
  meta_cell_data,
  var_data,
  meta_cell_method,
  obs_ids = NULL,
  cells_to_keep = NULL
)
```

Arguments

`meta_cell_data`	Named list. Output of meta-cell generation Rust functions. Will contain the aggregated raw and normalised counts, plus additional information on the origin of the meta cells.
`var_data`	data.table with the variable/feature informations.
`meta_cell_method`	String describing the origin of the metacell.
`obs_ids`	Optional character vector of length `n_metacells` with the meta cell identifiers. Defaults to `meta_cell_0001`, `meta_cell_0002`, ... Used by `merge_meta_cells()` so that the count matrices get their final row names at construction time rather than via a `rownames<-` that would duplicate them.
`cells_to_keep`	Optional integer vector. The source's `get_cells_to_keep()`, i.e. 0-indexed positions in its full obs table, in the row order its cached artefacts use. Recorded so that the meta cell memberships, which are positions in the *full* obs space, can be resolved against embeddings, kNN graphs and diffusion maps, whose rows only cover the QC-passing cells.

Value

Returns the `MetaCells` class for further operations.

Properties

obs_table The meta cell observation table.

var_table The meta cell variable table.

data List with the raw and normalised counts.

sc_cache Class with embeddings, kNN/sNN graphs, etc. Shared with `SingleCells()`.

original_assignment List with original assignment information.

dims Dimensions of the new meta cell matrices.

other_data Potential other data returned from the meta-cell generating methods.

is_merged Boolean. `TRUE` for objects returned by `merge_meta_cells()`. Methods that need to resolve `original_cell_idx` against the source single cell data use this to bail out early.

Examples

```
# the class is produced by the aggregation functions, not built by hand
sc <- demo_single_cells()
mc <- generate_bt_meta_cells_sc(
  sc,
  sc_meta_cell_params = params_sc_bt_metacells(target_no_metacells = 50L),
  .verbose = FALSE
)
mc

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

module_scores_sc *Calculate module activity scores*

Description

Implements the approach from Tirosh et al into bixverse, i.e., the AddModuleScore functionality from Seurat. For each module (gene set), computes the average expression of genes in the set minus the average expression of randomly selected control genes from the same expression bins. Genes are binned based on their average expression across cells to ensure controls are expression-matched.

Usage

```
module_scores_sc(
  object,
  gs_list,
  n_bins = 24L,
  n_ctrl = 100L,
  seed = 42L,
  streaming = NULL,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`gs_list`	Named list. The elements have the gene identifiers of the respective gene sets.
`n_bins`	Integer. Number of bins to use. Defaults to `24L`.
`n_ctrl`	Integer. Number of control genes to use per gene for each gene set. Defaults to `100L`.
`seed`	Integer. The random seed.
`streaming`	Optional Boolean. Shall the data be streamed in. Useful for larger data sets where you wish to avoid loading in the whole data. If `NULL`, will automatically detect.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

Returns a `ScMatrixRes` with the module scores.

References

Tirosh et al, Science (2016)

Examples

```
# score the three planted marker programmes per cell
sc <- demo_single_cells()
gs_list <- list(
  type_1 = sprintf("gene_%02d", 1:10),
  type_2 = sprintf("gene_%02d", 11:20),
  type_3 = sprintf("gene_%02d", 21:30)
)
res <- module_scores_sc(sc, gs_list = gs_list, n_ctrl = 5L, .verbose = FALSE)
dim(res)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`modules_from_loadings` *Derive sparse module membership from a loading matrix*

Description

Turns a `gene x k` loading matrix from a matrix factorisation (ICA, NMF, DGRDL) into a membership table by keeping the tails of each component's loading distribution.

Two thresholding rules, selected via `membership_params`:

- `"zscore"` - standardisation per component, keeping `abs(z) > cutoff`. No distributional assumption beyond rough symmetry.
- `"fdr"` - two-sided p-values against a Normal null fitted per component, Benjamini-Hochberg adjusted, keeping `padj < fdr`.

The standardisation itself is controlled by `membership_params$scaling`: `"robust"` centres and scales by the median and MAD, `"standard"` by the mean and standard deviation. The latter is stricter and keeps fewer genes on skewed loadings, which is common for NMF.

Usage

```
modules_from_loadings(loadings, membership_params = params_module_membership())
```

Arguments

`loadings`	Numeric matrix. `gene x k`, with row and column names.
`membership_params`	List. See `params_module_membership()`.

Value

A data.table with columns `gene`, `module_id`, `loading`, `sign` and the per-component score (`z` or `padj` depending on the method). One row per surviving (gene, component) pair, ordered by component then by descending absolute loading.

Examples

```
# turn NMF gene loadings into module membership
syn <- generate_gene_module_data(n_samples = 24L, n_genes = 60L)
# NMF needs a non-negative matrix
mat <- syn$data - min(syn$data)
obj <- BulkCoExp(mat, syn$meta_data)
obj <- preprocess_bulk_coexp(
  obj, hvg = NULL, scaling = FALSE, .verbose = FALSE
)
obj <- nmf_bulk(obj, k = 4L, .verbose = FALSE)
head(modules_from_loadings(get_nmf_gene_loadings(obj)))
```

---

nebula_mc *Run NEBULA on meta cells*

Description

Fits NEBULA's negative binomial gamma mixed model over aggregated counts. The arithmetic is `nebula_sc()` verbatim, only the counts come out of memory rather than the streamed store, and the fit is much cheaper because there are far fewer rows.

What changes is the interpretation, not the numerics. The cell-level overdispersion becomes the spread between aggregates within a subject rather than between cells, so it is smaller

and it absorbs whatever the aggregation smoothed away. The subject-level term keeps its meaning. Read the two as a variance decomposition over meta cells and do not compare them against a single cell run.

Usage

```
nebula_mc(
  object,
  subject_col,
  design,
  coef = NULL,
  contrast = NULL,
  genes_to_use = NULL,
  cells_to_use = NULL,
  offset = NULL,
  nebula_params = params_nebula(),
  .verbose = TRUE
)
```

Arguments

`object`	`MetaCells` class.
`subject_col`	String. The column in the obs table holding the subject (donor) identifier. This is what the random effect is over.
`design`	Formula. The experimental design, evaluated against the obs table, e.g. `~ condition` or `~ condition + age`. Include the intercept.
`coef`	Optional integer or character. Which coefficient of the design the Wald test reports, as a 1-based column position or a column name. Defaults to the last column.
`contrast`	Optional numeric vector. One weight per design column. Mutually exclusive with `coef`.
`genes_to_use`	Optional character vector. The genes to fit. Defaults to every gene in the object.
`cells_to_use`	Optional character vector. Meta cell identifiers (`meta_cell_id`) to fit. Defaults to every meta cell. Identifiers that cannot be matched are dropped with a warning.
`offset`	Optional numeric vector. Strictly positive scaling factor per meta cell, aligned to the meta cells that survive the design. Defaults to `NULL`, which uses the aggregated library sizes.
`nebula_params`	A list, see `params_nebula()`. See `nebula_sc()` for the individual elements.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

A `ScNebula` class, see `new_sc_nebula_res()`.

References

He, et al., Commun Biol, 2021

Examples

```
# differential expression over meta cells with a donor random effect
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 2000L,
    n_genes = 50L,
    n_samples = 6L,
    sample_bias = "even"
  )
)
mc <- generate_bt_meta_cells_sc(
  sc,
  sc_meta_cell_params = params_sc_bt_metacells(target_no_metacells = 100L),
  .verbose = FALSE
)
# the meta cell obs carries no donor labels, so vote them over
sample_id <- sc[["sample_id"]]$sample_id
mc[["sample_id"]] <- vapply(
  mc[[]]$original_cell_idx,
  function(idx) names(which.max(table(sample_id[idx]))),
  character(1)
)
mc[["condition"]] <- ifelse(
  mc[["sample_id"]]$sample_id %in% c("sample_1", "sample_2", "sample_3"),
  "ctr",
  "trt"
)
res <- nebula_mc(
  mc,
  subject_col = "sample_id",
  design = ~condition,
  .verbose = FALSE
)
head(res$results)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`nebula_sc` *Run NEBULA on single cells*

Description

Fits NEBULA's negative binomial gamma mixed model, which is the test to reach for when the cells are not independent because they came from a handful of donors. The variance is split into a subject-level random effect and a cell-level overdispersion, so the donor structure is modelled rather than ignored, and the nominal false discovery rate holds.

The design is a formula evaluated against the obs table, so anything in there can go into the model. Cells with a missing design or subject value are dropped with a warning.

Genes are streamed and fitted in batches, which is exact: NEBULA is gene-independent. A fit is milliseconds to seconds per gene though, so running it over the full gene axis is rarely what you want. Restrict `genes_to_use` to the highly variable genes or a candidate set.

Usage

```
nebula_sc(
  object,
  subject_col,
  design,
  coef = NULL,
  contrast = NULL,
  genes_to_use = NULL,
  cells_to_use = NULL,
  offset = NULL,
  nebula_params = params_nebula(),
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`subject_col`	String. The column in the obs table holding the subject (donor) identifier. This is what the random effect is over. Not the same thing as a sample or a batch, unless they happen to coincide.
`design`	Formula. The experimental design, evaluated against the obs table, e.g. `~ condition` or `~ condition + age`. Include the intercept.
`coef`	Optional integer or character. Which coefficient of the design the Wald test reports, as a 1-based column position or a column name. Defaults to the last column.
`contrast`	Optional numeric vector. One weight per design column. Mutually exclusive with `coef`.
`genes_to_use`	Optional character vector. The genes to fit. Defaults to every gene in the object, which is usually too many.
`cells_to_use`	Optional character vector. Names of the cells to fit, e.g. one cell type or one condition. Defaults to every cell that passed quality control. Cells that cannot be matched are dropped with a warning.
`offset`	Optional numeric vector. Strictly positive scaling factor per cell, aligned to the cells that survive the design. Defaults to `NULL`, which uses the library sizes.
`nebula_params`	A list, see `params_nebula()`. The list has the following parameters: • nebula_method - String. One of `c("ln", "hl")`. • min_sigma, max_sigma - Numeric. Bounds on the subject-level overdispersion.

- min_phi, max_phi - Numeric. Bounds on the cell-level overdispersion.
- cutoff_cell - Numeric. When to refit both overdispersions.
- kappa - Numeric. When to trust the stage-one subject overdispersion.
- cpc - Numeric. Minimum mean count per cell for a gene to be tested.
- mincp - Integer. Minimum number of cells expressing a gene.
- min_subjects - Integer. Minimum number of subjects whose own mean count per cell clears `cpc`. `0` switches the check off.
- reml - Boolean. Restricted maximum likelihood.
- eps - Numeric. Optimiser stopping tolerance.
- gene_batch_size - Integer. Genes read and fitted per batch.
- shrink_dispersion - Boolean. Empirical Bayes shrinkage of the cell-level overdispersions.

`.verbose` Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

A `ScNebula` class, see `new_sc_nebula_res()`, with

- results - data.table. One row per gene that survived NEBULA's expression filter, with the Wald test and both overdispersions.
- coefficients - Numeric matrix of genes x coefficients.
- se - Numeric matrix of genes x coefficients.
- params - List. The parameters the run used.

References

He, et al., Commun Biol, 2021

Examples

```
# mixed model DGE with the sample as the random effect
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 500L,
    n_genes = 50L,
    n_samples = 6L,
    sample_bias = "even"
  )
)
obs <- get_sc_obs(sc)
ctr <- sprintf("sample_%i", 1:3)
condition <- ifelse(obs$sample_id %in% ctr, "ctr", "trt")
sc <- set_sc_new_obs_col(sc, col_name = "condition", new_data = condition)
res <- nebula_sc(
  sc,
  subject_col = "sample_id",
```

```
  design = ~condition,
  .verbose = FALSE
)
head(res$results)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`NetworkDiffusions` *Network diffusion class*

Description

This class helps to do diffusion of seed nodes in a single or tied version over a network, measure the ability of these diffusion vectors to recall against a gold standard set of nodes and do community detection within the subset of the network that received the the most heat from the initial seed genes.

Usage

```
NetworkDiffusions(edge_data_frame, weighted, directed)
```

Arguments

`edge_data_frame`	data.table that contains the edge information. It is expected to have the columns 'from' and 'to'.
`weighted`	Boolean. Is the graph weighted. If set to TRUE, the `edge_data_frame` needs to have a weight column.
`directed`	Boolean. Shall the graph be stored as directed.

Value

Returns the `NetworkDiffusions` class for further operations.

Properties

graph igraph. The underlying graph.

diffusion_res Numeric vector. Contains contains the single or tied diffusion results.

params A (nested) list that will store all the parameters of the applied function.

final_results data.table. Contains final results.

Examples

```
# unweighted, undirected diffusion class from an edge table
set.seed(42)
g <- igraph::sample_pa(15, directed = FALSE)
edges <- data.table::setDT(igraph::as_data_frame(g))[, `:=`(
  from = sprintf("node_%i", from),
  to = sprintf("node_%i", to)
)]
object <- NetworkDiffusions(edges, weighted = FALSE, directed = FALSE)
class(object)[1]
```

---

new_adt_counts_clr *Generates a new* ADTCounts *class*

Description

This function generates a new `ADTCounts` class which uses CLR normalisation under the hood. You have the choice between Seurat-style CLR (no negative values) and normal CLR (allows negative values).

Usage

```
new_adt_counts_clr(
  raw_counts,
  cell_info,
  seurat_clr = FALSE,
  clean_clr_counts = TRUE,
  percentile = 0.01
)
```

Arguments

`raw_counts`	Numeric matrix. The raw ADT counts.
`cell_info`	Named integer vector. Output of `get_cell_info()`. Defines as elements the cell indices (R-based) and as names the barcodes.
`seurat_clr`	Boolean. Shall a Seurat-style CLR be applied.
`clean_clr_counts`	Boolean. Shall the per-protein 1st percentile be removed from the CLR normalised counts.
`percentile`	Numeric. The percentile to remove to reduce background effects.

Value

`ADTCounts` that contains the raw and normalised ADT counts.

Examples

```
# CLR normalisation of synthetic ADT counts
adt <- generate_single_cell_test_data_adt()
cell_info <- stats::setNames(
  seq_len(nrow(adt$counts)),
  rownames(adt$counts)
)
new_adt_counts_clr(adt$counts, cell_info = cell_info)
```

---

new_adt_counts_dsb *Generates a new* ADTCounts *class via DSB normalisation*

Description

This function generates a new ADTCounts class using DSB normalisation from Mulè et al., instead of CLR. When empty_drops is provided, the per-protein background is estimated from empty droplets. Without empty_drops, the function falls back to a 2-component k-means on the log-transformed cell counts ("ModelNegativeADTnorm").

Usage

```
new_adt_counts_dsb(
  raw_counts,
  cell_info,
  empty_drops = NULL,
  isotype_names = NULL,
  dsb_params = params_sc_dsb(),
  scale_factor = c("standardise", "mean_subtract"),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

raw_counts	Numeric matrix. Cells x proteins matrix of raw ADT counts with cell barcodes as row names and protein names as column names.
cell_info	Named integer vector. Output of get_cell_info(). Defines as elements the cell indices (R-based) and as names the barcodes.
empty_drops	Optional numeric matrix. Cells x proteins matrix of empty-droplet ADT counts. If provided, used to estimate per-protein ambient background.
isotype_names	Optional string vector. Column names in raw_counts identifying isotype control proteins. Required when dsb_params$use_isotype_controls = TRUE.
dsb_params	List. Output of params_sc_dsb() with DSB parameters.
scale_factor	String. One of c("standardise", "mean_subtract"). Only used when empty_drops is provided.

seed	Integer. Random seed for k-means initialisation.
.verbose	Boolean or integer. Controls verbosity and returns run times. FALSE -> quiet, TRUE or 1L -> normal verbosity, 2L -> detailed verbosity.

Value

ADTCounts containing the raw counts and the DSB-normalised counts.

References

Mulè et al., Nat Commun, 2022

Examples

```
# DSB without empty droplets, i.e. the k-means fallback, using the isotypes
adt <- generate_single_cell_test_data_adt()
cell_info <- stats::setNames(
  seq_len(nrow(adt$counts)),
  rownames(adt$counts)
)
new_adt_counts_dsb(
  adt$counts,
  cell_info = cell_info,
  isotype_names = adt$var$protein_id[adt$var$is_isotype],
  .verbose = FALSE
)
```

---

new_sc_knn *Helper function to generate kNN data with distances*

Description

Wrapper class that stores kNN data for subsequent usage in various other functions, methods and helpers.

Usage

```
new_sc_knn(knn_data, used_cells)
```

Arguments

knn_data	Named list with the following items: • indices - Integer matrix containing the indices of the nearest neighbours (0-indexed). • dist - Numerical matrix containing the distances to the nearest neighbours. • dist_metric - String. Distance metric used.
used_cells	Character vector. The cells used to generate the kNN graph with the distances.

Value

Generates the `SingleCellNearestNeighbour` class.

Examples

```
# rebuild the kNN wrapper from an existing graph
sc <- demo_single_cells()
knn <- generate_knn_sc(sc, .validate_index = FALSE, .verbose = FALSE)
new_sc_knn(
  knn_data = list(
    indices = get_knn_mat(knn),
    dist = get_knn_dist(knn),
    dist_metric = "euclidean"
  ),
  used_cells = get_cell_names(sc)
)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`nmf_bulk` *Run non-negative matrix factorisation on a BulkCoExp*

Description

Fits a single HALS-NMF `V ~ W H` to the bulk expression matrix with a fixed number of components `k`. `V` is samples x features (matches the layout of `raw_data` / `processed_data` on `BulkCoExp()`). The result is rearranged so that:

- `gene_loadings` (features x k) captures per-module gene contributions.
- `sample_activity` (samples x k) captures per-module sample activity.

Module membership is derived by keeping the upper tail of each component's gene loadings, see `params_module_membership()`. A gene can belong to several modules, and a gene in no tail belongs to none.

Usage

```
nmf_bulk(
  object,
  k,
  preprocessing = c("none", "sd", "sqrt_sd"),
  nmf_hals_params = params_nmf_hals(),
  membership_params = params_module_membership(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	The class, see `BulkCoExp()`. Ideally, you should run `preprocess_bulk_coexp()` before applying this function. The NMF requires non-negative input, so the pre-processing should either skip scaling or use a strictly non-negative representation.
`k`	Integer. Number of latent factors (modules).
`preprocessing`	String. One of `c("none", "sd", "sqrt_sd")`. Applied inside the Rust kernel to the input matrix before the HALS updates.
`nmf_hals_params`	List. Output of `params_nmf_hals()`: • max_iter - Integer. Maximum number of HALS iterations. • tol - Float. Convergence tolerance. • eps - Float. Numerical floor. • check_every - Integer. Convergence check interval. • nmf_init - String. One of `c("nndsvd", "svd", "random")`.
`membership_params`	List. Controls how the gene loadings are turned into module membership, see `params_module_membership()`. Membership is not exclusive: a gene loading strongly on several components appears in several modules, and a gene in no tail appears in none.
`seed`	Integer. Random seed for the NMF initialisation. Defaults to `42L`.
`.verbose`	Boolean or integer `0L`/`1L`/`2L`. Controls verbosity.

Value

The class with `final_results` populated (see description) and the fit parameters stored under `params$nmf_fit`, plus `params$detection_method = "nmf-based"`.

References

Cichocki & Phan, IEICE Trans., 2009.

Examples

```
# a single four-factor HALS-NMF fit
syn <- generate_gene_module_data(n_samples = 24L, n_genes = 60L)
# NMF needs a non-negative matrix
mat <- syn$data - min(syn$data)
obj <- BulkCoExp(mat, syn$meta_data)
obj <- preprocess_bulk_coexp(
  obj, hvg = NULL, scaling = FALSE, .verbose = FALSE
)
obj <- nmf_bulk(obj, k = 4L, .verbose = FALSE)
head(get_nmf_modules(obj))
```

`nmf_k_sweep_bulk` *Sweep k for consensus NMF on a BulkCoExp*

Description

Runs the consensus clustering step across a range of `k` and reports stability against reconstruction error, without keeping any factors. Pick the `k` where stability is still high and the error curve has not yet flattened out, then run `consensus_nmf_bulk()` there.

Usage

```
nmf_k_sweep_bulk(
  object,
  k_range,
  n_runs = 30L,
  preprocessing = c("none", "sd", "sqrt_sd"),
  nmf_hals_params = params_nmf_hals(),
  nmf_consensus_params = params_nmf_consensus(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	The class, see `BulkCoExp()`.
`k_range`	Integer vector. The ranks to evaluate. Every entry must be at least 2.
`n_runs`	Integer. Number of random restarts per `k`. Must be at least 2.
`preprocessing`	String. One of `c("none", "sd", "sqrt_sd")`.
`nmf_hals_params`	List. Output of `params_nmf_hals()`.
`nmf_consensus_params`	List. Output of `params_nmf_consensus()`.
`seed`	Integer. Base random seed.
`.verbose`	Boolean or integer `0L`/`1L`/`2L`. Controls verbosity.

Details

This is a diagnostic, so it leaves the object alone and hands you the result back directly. `plot()` on the returned object gives you the two curves.

Cost is `length(k_range) * n_runs` full NMF fits, so keep both modest on a first pass.

Value

An `NmfKSweepResult`, which is a data.table with one row per `k`.

References

Kotliar et al., eLife, 2019

Examples

```
# small sweep, keep both the grid and the restarts modest
syn <- generate_gene_module_data(n_samples = 24L, n_genes = 60L)
# NMF needs a non-negative matrix
mat <- syn$data - min(syn$data)
obj <- BulkCoExp(mat, syn$meta_data)
obj <- preprocess_bulk_coexp(
  obj, hvg = NULL, scaling = FALSE, .verbose = FALSE
)
sweep_res <- nmf_k_sweep_bulk(
  obj, k_range = 3:5, n_runs = 5L, .verbose = FALSE
)
sweep_res
```

---

nmf_k_sweep_sc *Sweep k for consensus NMF on single cell or meta cell data*

Description

Runs the consensus clustering step across a range of `k` and reports stability against reconstruction error, without keeping any factors. Pick the `k` where stability is still high and the error curve has not yet flattened out, then run `consensus_nmf_sc()` there.

Usage

```
nmf_k_sweep_sc(
  object,
  k_range,
  cell_ids = NULL,
  gene_ids = NULL,
  preprocessing = "none",
  use_second_layer = TRUE,
  nmf_hals_params = params_nmf_hals(),
  nmf_consensus_params = params_nmf_consensus(),
  n_runs = 30L,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `MetaCells` class.
`k_range`	Integer vector. The ranks to evaluate. Every entry must be at least 2.

`cell_ids` Optional character. Cell ids (or meta cell ids) to restrict the NMF to. If `NULL`, uses `get_cells_to_keep()` for `SingleCells` and all meta cells for `MetaCells`.

`gene_ids` Optional character. Gene ids to restrict the NMF to. If `NULL`, uses `get_hvg()` on the object.

`preprocessing` String. One of `c("none", "sd", "sqrt_sd")`.

`use_second_layer` Boolean. If `TRUE`, runs NMF on the normalised counts (recommended); if `FALSE`, on the raw counts.

`nmf_hals_params` List, see `params_nmf_hals()`. The `nmf_init` field is ignored, restarts always use random initialisation.

`nmf_consensus_params` List, see `params_nmf_consensus()`.

`n_runs` Integer. Number of random restarts. Must be at least 2.

`seed` Integer. Base random seed. Restart `i` uses `seed + i`, and the k-means step is seeded from it too.

`.verbose` Boolean or integer. Verbosity.

Details

This is a diagnostic, so it leaves the object alone and hands the result back directly. `plot()` on it gives you the two curves.

Cost is `length(k_range) * n_runs` full NMF fits. On the `SingleCells` path the counts are read from disk once and reused across every `k`, but the fits themselves are not free, so keep both modest on a first pass.

Value

An `NmfKSweepResult`, which is a data.table with one row per `k`.

References

Kotliar et al., eLife, 2019

Examples

```
# stability against reconstruction error across three ranks
sc <- demo_single_cells()
res <- nmf_k_sweep_sc(
  sc,
  k_range = 2:4,
  n_runs = 5L,
  nmf_consensus_params = params_nmf_consensus(density_threshold = 2),
  .verbose = FALSE
)
res

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`nmf_sc` *Run single-run NMF on single cell or meta cell data*

Description

Runs a single HALS NMF on a chosen subset of cells and genes. For `SingleCells`, the counts are streamed from disk via the Rust binary files; for `MetaCells`, the in-memory sparse counts are used.

Usage

```
nmf_sc(
  object,
  k,
  cell_ids = NULL,
  gene_ids = NULL,
  preprocessing = "none",
  use_second_layer = TRUE,
  nmf_hals_params = params_nmf_hals(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `MetaCells` class.
`k`	Integer. Number of latent factors to return.
`cell_ids`	Optional character. Cell ids (or meta cell ids) to restrict the NMF to. If `NULL`, uses `get_cells_to_keep()` for `SingleCells` and all meta cells for `MetaCells`.
`gene_ids`	Optional character. Gene ids to restrict the NMF to. If `NULL`, uses `get_hvg()` on the object.
`preprocessing`	String. One of `c("none", "sd", "sqrt_sd")`.
`use_second_layer`	Boolean. If `TRUE`, runs NMF on the normalised counts (recommended); if `FALSE`, on the raw counts.
`nmf_hals_params`	List, see `params_nmf_hals()`.
`seed`	Integer. Random seed for initialisation.
`.verbose`	Boolean or integer. Verbosity.

Value

An `NmfResult` object.

Examples

```
# three factors on the highly variable genes
sc <- demo_single_cells()
res <- nmf_sc(sc, k = 3L, .verbose = FALSE)
res

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

normalise_bulk_dge *Normalise the count data for DGE.*

Description

Calculates the normalisation factors and applies voom on the filtered counts from `qc_bulk_dge()`, both in Rust via the `edge-rs` crate. Can additionally calculate TPM and FPKM values for plotting purposes.

Usage

```
normalise_bulk_dge(
  object,
  group_col,
  norm_method = c("TMM", "TMMwsp", "RLE", "upperquartile", "none"),
  calc_tpm = FALSE,
  calc_fpkm = FALSE,
  gene_lengths = NULL,
  .verbose = TRUE
)
```

Arguments

`object`	The underlying class, see `BulkDge()`.
`group_col`	String. The column in the metadata that will contain the contrast groups. Needs to be part of the metadata stored in the class.
`norm_method`	String. One of `c("TMM", "TMMwsp", "RLE", "upperquartile", "none")`. Please refer to edgeR's `calcNormFactors()`.
`calc_tpm`	Boolean. Output TPM calculation (default = FALSE).
`calc_fpkm`	Boolean. Output FPKM calculation (default = FALSE).
`gene_lengths`	Optional named numeric. If you want to calculate TPM or FPKM you need to provide this one. The names need to be the same identifier as used in the counts.
`.verbose`	Boolean. Controls the verbosity of the function.

Value

Returns the class with the `processed_data` data slot populated and applied parameters added to the `params` slot.

Examples

```
# TMM library size normalisation followed by voom
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- qc_bulk_dge(object, group_col = "case_control", .verbose = FALSE)
object <- normalise_bulk_dge(
  object,
  group_col = "case_control",
  .verbose = FALSE
)
get_outputs(object)$normalised_counts[1:3, 1:3]
```

OntologySim *OntologySim class*

Description

This class is used to store any ontology and apply different methods to it. Currently implemented are different types of term similarities, i.e., based on semantic similarities and the Wang similarity.

Usage

```
OntologySim(parent_child_dt, .verbose = TRUE)
```

Arguments

`parent_child_dt`	A data.table that contains the ontological information in terms of parent child relationships. Need to contain the `c("parent", "child")` columns.
`.verbose`	Boolean. Controls the verbosity of the class

Value

Returns the class for subsequent usage.

Properties

edge_dt data.table. Contains the parent-child relationships. (For Wang similarity also the relationship type.)

outputs List. Contains various intermediary results used for some methods.

sim_mat List. Contains the potentially calculated similarity matrix in form of an R6 class. Getters to access the data are provided.

params A (nested) list that will store all the parameters of the applied function.

final_results Final results stored in the class.

Examples

```
# wrap a toy parent-child ontology into the class
onto <- data.table::data.table(
  parent = c("a", "b", "b", "b", "c"),
  child = c("b", "c", "d", "e", "f"),
  type = c("part_of", "part_of", "part_of", "is_a", "is_a")
)
OntologySim(onto, .verbose = FALSE)
```

---

ot_harmonic_score *Calculates a harmonic sum normalised between 0 to 1.*

Description

The function takes in a vector of scores between 0 and 1 and calculates a harmonic sum, based on the approach OpenTargets takes to do their gene - disease evidence scores, see: https://platform-docs.opentargets.org/associations.

Usage

```
ot_harmonic_score(x)
```

Arguments

x	Numeric vector. Needs to be between 0 and 1.

Value

Harmonic, normalised sum of the provided scores.

Examples

```
# harmonic sum over four evidence scores
ot_harmonic_score(c(1, 0.8, 0.5, 0.1))
```

---

params_binary_heatmap

Wrapper function for parameters for binary heatmap data

Description

Controls filtering, ordering and binning in `extract_binary_heatmap_data()`. Features are clustered on the Jaccard distance, samples on the Hamming distance within their group. Groups larger than `max_cluster_n` are ordered by barycentre instead, which skips the distance matrix. More samples than `max_cols` get binned within their group into fraction-on columns.

Usage

```
params_binary_heatmap(
  min_frac_on = 0.01,
  max_frac_on = 0.99,
  cluster_features = TRUE,
  cluster_samples = TRUE,
  max_cols = 2000L,
  max_cluster_n = 2000L
)
```

Arguments

`min_frac_on`	Numeric. Features on in a smaller fraction of samples are dropped. Defaults to `0.01`.
`max_frac_on`	Numeric. Features on in a larger fraction of samples are dropped. Defaults to `0.99`.
`cluster_features`	Boolean. Shall the features be clustered within their group. Defaults to `TRUE`.
`cluster_samples`	Boolean. Shall the samples be ordered within their group. Defaults to `TRUE`.
`max_cols`	Integer. Maximum number of columns before samples get binned. Defaults to `2000L`.
`max_cluster_n`	Integer. Groups with more samples than this are ordered by barycentre instead of hierarchical clustering. Defaults to `2000L`.

Value

A named list with the following elements:

- min_frac_on - Numeric. Features on in a smaller fraction of samples are dropped. Defaults to `0.01`.
- max_frac_on - Numeric. Features on in a larger fraction of samples are dropped. Defaults to `0.99`.
- cluster_features - Boolean. Shall the features be clustered within their group. Defaults to `TRUE`.
- cluster_samples - Boolean. Shall the samples be ordered within their group. Defaults to `TRUE`.
- max_cols - Integer. Maximum number of columns before samples get binned. Defaults to `2000L`.
- max_cluster_n - Integer. Groups with more samples than this are ordered by barycentre instead of hierarchical clustering. Defaults to `2000L`.

`params_blitzgsea` *Wrapper function to generate blitzGSEA parameters*

Description

Wrapper function to generate blitzGSEA parameters

Usage

```
params_blitzgsea(
  min_size = 5L,
  max_size = 500L,
  permutations = 2000L,
  anchors = 40L,
  symmetric = FALSE,
  centre = TRUE,
  ks_test = TRUE,
  seed = 42
)
```

Arguments

`min_size`	Integer. Minimum number of genes per gene set. Defaults to `5L`.
`max_size`	Integer. Maximum number of genes per gene set. Defaults to `500L`.
`permutations`	Integer. Random gene sets drawn per anchor size during calibration. Below `1000L` the two tails are pooled into a single gamma regardless of `symmetric`. Defaults to `2000L`.
`anchors`	Integer. Number of log-spaced anchor sizes requested. Sizes that collide after rounding are collapsed, so the realised grid is usually a little smaller. Defaults to `40L`.
`symmetric`	Boolean. Pool both tails into one gamma instead of fitting them separately. Defaults to `FALSE`.
`centre`	Boolean. Centre the signature on its mean before scoring. The enrichment score is not invariant to an offset, so the calibration and the scoring have to agree on this. Defaults to `TRUE`.
`ks_test`	Boolean. Run the Kolmogorov-Smirnov goodness-of-fit diagnostic at every anchor. Costs a sort per anchor. Defaults to `TRUE`.
`seed`	Numeric. Random seed for the calibration. Defaults to `42.0`.

Value

A named list with the following elements:

- min_size - Integer. Minimum number of genes per gene set. Defaults to `5L`.
- max_size - Integer. Maximum number of genes per gene set. Defaults to `500L`.

- permutations - Integer. Random gene sets drawn per anchor size during calibration. Below `1000L` the two tails are pooled into a single gamma regardless of `symmetric`. Defaults to `2000L`.
- anchors - Integer. Number of log-spaced anchor sizes requested. Sizes that collide after rounding are collapsed, so the realised grid is usually a little smaller. Defaults to `40L`.
- symmetric - Boolean. Pool both tails into one gamma instead of fitting them separately. Defaults to `FALSE`.
- centre - Boolean. Centre the signature on its mean before scoring. The enrichment score is not invariant to an offset, so the calibration and the scoring have to agree on this. Defaults to `TRUE`.
- ks_test - Boolean. Run the Kolmogorov-Smirnov goodness-of-fit diagnostic at every anchor. Costs a sort per anchor. Defaults to `TRUE`.
- seed - Numeric. Random seed for the calibration. Defaults to `42.0`.

References

Lachmann, et al., Bioinformatics, 2022

`params_boost` *Wrapper function for Boost parameters*

Description

Wrapper function for Boost parameters

Usage

```
params_boost(
  boost_rate = 0.25,
  replace = FALSE,
  resolution = 1,
  n_iters = 20L,
  p_thresh = 1e-07,
  voter_thresh = 0.9,
  fast_cluster = FALSE,
  normalisation = list(),
  hvg = list(),
  pca = list(),
  knn = list(k = 0L),
  fast_cluster_params = list()
)
```

Arguments

boost_rate	Numeric. Boosting rate for the algorithm. Must be between 0 and 1. Defaults to 0.25.
replace	Boolean. Whether to use replacement during boosting. Defaults to FALSE.
resolution	Numeric. Resolution parameter for graph-based clustering. Higher values lead to more clusters. Defaults to 1.0.
n_iters	Integer. Number of iterations to run the algorithm. Defaults to 20L.
p_thresh	Numeric. P-value threshold for significance testing. Defaults to 1e-07.
voter_thresh	Numeric. Voter threshold across iterations. Proportion of iterations a cell must be assigned to a cluster to be considered a member. Must be between 0 and 1. Defaults to 0.9.
fast_cluster	Boolean. Shall fast Louvain clustering be applied, i.e., k-means clustering and use the centroids for kNN graph generation and Louvain clustering with then backpropagating the membership based on centroid proximity. Defaults to FALSE.
normalisation	List. Optional overrides for normalisation parameters. See params_norm_doublets_defaults() for available parameters: log_transform, mean_center, normalise_variance, target_size. See params_norm_doublets_defaults() for the available elements. Defaults to list().
hvg	List. Optional overrides for highly variable gene selection parameters. See params_hvg_defaults() for available parameters: min_gene_var_pctl, hvg_method, loess_span, clip_max. See params_hvg_defaults() for the available elements. Defaults to list().
pca	List. Optional overrides for PCA parameters. See params_pca_defaults() for available parameters: no_pcs and svd_solver. See params_pca_defaults() for the available elements. Defaults to list().
knn	List. Optional overrides for kNN parameters. See params_knn_defaults() for available parameters: k, knn_method, ann_dist, search_budget, n_trees, delta, diversify_prob, ef_budget, extract_knn, m, ef_construction, ef_search, n_list and n_probe. Note: this function defaults to k = 0L (automatic neighbour detection). See params_knn_defaults() for the available elements. Defaults to list(k = 0L).
fast_cluster_params	List. Optional overrides for the fast clustering parameters. Only relevant if fast_cluster = TRUE. See params_fast_cluster_default() for available parameters: km_type, n_centroids, kmeans_iters and batch_size. See params_fast_cluster_default() for the available elements. Defaults to list().

Value

A named list with the following elements:

- The elements of params_norm_doublets_defaults(), overridden by normalisation, spliced in at this position.

- The elements of params_hvg_defaults(), overridden by hvg, spliced in at this position.
- The elements of params_pca_defaults(), overridden by pca, spliced in at this position.
- The elements of params_knn_defaults(), overridden by knn, spliced in at this position.
- The elements of params_fast_cluster_default(), overridden by fast_cluster_params, spliced in at this position.
- boost_rate - Numeric. Boosting rate for the algorithm. Must be between 0 and 1. Defaults to 0.25.
- replace - Boolean. Whether to use replacement during boosting. Defaults to FALSE.
- resolution - Numeric. Resolution parameter for graph-based clustering. Higher values lead to more clusters. Defaults to 1.0.
- fast_cluster - Boolean. Shall fast Louvain clustering be applied, i.e., k-means clustering and use the centroids for kNN graph generation and Louvain clustering with then backpropagating the membership based on centroid proximity. Defaults to FALSE.
- n_iters - Integer. Number of iterations to run the algorithm. Defaults to 20L.
- p_thresh - Numeric. P-value threshold for significance testing. Defaults to 1e-07.
- voter_thresh - Numeric. Voter threshold across iterations. Proportion of iterations a cell must be assigned to a cluster to be considered a member. Must be between 0 and 1. Defaults to 0.9.

`params_bpca` *Wrapper function for Bayesian PCA parameters*

Description

Parameters for PCA with missing values via Bayesian PCA, see run_bpca(). Defaults follow pcaMethods.

Usage

```
params_bpca(
  n_pcs = 2L,
  max_iter = 100L,
  tol = 1e-04,
  centre = TRUE,
  scale = FALSE
)
```

Arguments

n_pcs	Integer. Number of principal components. Defaults to 2L.
max_iter	Integer. Maximum number of variational steps. Defaults to 100L.
tol	Numeric. Change in log10(tau) over ten steps below which the fit stops. Defaults to 1e-04.
centre	Boolean. Shall the observed column means be subtracted first. Defaults to TRUE.
scale	Boolean. Shall the columns be divided by their observed standard deviation first (pcaMethods' "uv"). Defaults to FALSE.

Value

A named list with the following elements:

- n_pcs - Integer. Number of principal components. Defaults to 2L.
- max_iter - Integer. Maximum number of variational steps. Defaults to 100L.
- tol - Numeric. Change in log10(tau) over ten steps below which the fit stops. Defaults to 1e-04.
- centre - Boolean. Shall the observed column means be subtracted first. Defaults to TRUE.
- scale - Boolean. Shall the columns be divided by their observed standard deviation first (pcaMethods' "uv"). Defaults to FALSE.

References

Oba, et al., Bioinformatics, 2003; Stacklies, et al., Bioinformatics, 2007

---

params_bulk_sparsity *Wrapper function to generate bulk sparsification parameters*

Description

Parameters for simulate_dropouts(). Dropout falls out of the library size rather than an explicit per-gene dropout curve: a size factor s_j ~ LogNormal(0, capture_efficiency_sigma) is drawn per sample, giving a target library size of target_library_size * s_j, and each gene is binomially thinned towards that target.

Usage

```
params_bulk_sparsity(
  strategy = "seq_depth",
  target_library_size = 20000,
  capture_efficiency_sigma = 0.5,
  seed = 123L
)
```

Arguments

`strategy`	String. Which dropout strategy to apply. Currently only `"seq_depth"`. One of `"seq_depth"`. Defaults to `"seq_depth"`.
`target_library_size`	Numeric. Reference library size per sample. Defaults to `20000.0`.
`capture_efficiency_sigma`	Numeric. Standard deviation of the LogNormal size-factor distribution. Larger values spread the library sizes further apart. Defaults to `0.5`.
`seed`	Integer. Seed for reproducibility purposes. Defaults to `123L`.

Value

A named list with the following elements:

- strategy - String. Which dropout strategy to apply. Currently only `"seq_depth"`. One of `"seq_depth"`. Defaults to `"seq_depth"`.
- target_library_size - Numeric. Reference library size per sample. Defaults to `20000.0`.
- capture_efficiency_sigma - Numeric. Standard deviation of the LogNormal size-factor distribution. Larger values spread the library sizes further apart. Defaults to `0.5`.
- seed - Integer. Seed for reproducibility purposes. Defaults to `123L`.

References

Zappia, et al., Genome Biol, 2017

`params_cistarget` *Wrapper function to CisTarget parameters*

Description

`auc_threshold` and `max_rank` are two different cutoffs and are easy to confuse. The first truncates the recovery curve used to score motif enrichment, the second sets how deep into the ranking the background curve (mean + 2 SD across all motifs) is built, and therefore where the leading edge cuts. RcisTarget uses `maxRank = 5000` and `nMean = 100`; the defaults here follow it. `auc_threshold` follows pySCENIC at 5%, RcisTarget itself uses 3%.

Usage

```
params_cistarget(
  auc_threshold = 0.05,
  nes_threshold = 3,
  max_rank = 5000L,
  n_mean = 100L,
  rcc_method = c("approx", "icistarget"),
  high_conf_cats = c("directAnnotation", "inferredBy_Orthology"),
  low_conf_cats = c("inferredBy_MotifSimilarity",
    "inferredBy_MotifSimilarity_n_Orthology")
)
```

Arguments

`auc_threshold`	Numeric. Numeric between 0 and 1. Proportion of genes to use for AUC threshold calculation. Default is 0.05 (5% of genes). Defaults to `0.05`.
`nes_threshold`	Numeric. Normalised Enrichment Score threshold for significant motifs. Default is 3.0. Defaults to `3.0`.
`max_rank`	Integer. Depth of the recovery curves used to derive the background and the leading edge. Clamped to the number of genes in the ranking database. Default is 5000, the RcisTarget value. Defaults to `5000L`.
`n_mean`	Integer. Window for the rolling mean smoothing the background recovery curve. Only read when `rcc_method = "approx"`. Default is 100, the RcisTarget value. Defaults to `100L`.
`rcc_method`	String. Method for recovery curve calculation. Either "approx" (approximate, faster) or "icistarget" (exact, slower). One of `c("approx", "icistarget")`. Defaults to `"approx"`.
`high_conf_cats`	String vector. Annotation categories considered high confidence. Default includes direct annotations and orthology-based inferences. Defaults to `c("directAnnotation", "inferredBy_Orthology")`.
`low_conf_cats`	String vector. Annotation categories considered lower confidence. Default includes motif similarity-based inferences. Defaults to `c("inferredBy_MotifSimilarity", "inferredBy_MotifSimilarity_n_Orthology")`.

Value

A named list with the following elements:

- auc_threshold - Numeric. Numeric between 0 and 1. Proportion of genes to use for AUC threshold calculation. Default is 0.05 (5% of genes). Defaults to `0.05`.
- nes_threshold - Numeric. Normalised Enrichment Score threshold for significant motifs. Default is 3.0. Defaults to `3.0`.
- max_rank - Integer. Depth of the recovery curves used to derive the background and the leading edge. Clamped to the number of genes in the ranking database. Default is 5000, the RcisTarget value. Defaults to `5000L`.
- n_mean - Integer. Window for the rolling mean smoothing the background recovery curve. Only read when `rcc_method = "approx"`. Default is 100, the RcisTarget value. Defaults to `100L`.
- rcc_method - String. Method for recovery curve calculation. Either "approx" (approximate, faster) or "icistarget" (exact, slower). One of `c("approx", "icistarget")`. Defaults to `"approx"`.
- high_conf_cats - String vector. Annotation categories considered high confidence. Default includes direct annotations and orthology-based inferences. Defaults to `c("directAnnotation", "inferredBy_Orthology")`.
- low_conf_cats - String vector. Annotation categories considered lower confidence. Default includes motif similarity-based inferences. Defaults to `c("inferredBy_MotifSimilarity", "inferredBy_MotifSimilarity_n_Orthology")`.

`params_community_detection` *Wrapper function to generate community detection parameters*

Description

Wrapper function to generate community detection parameters

Usage

```
params_community_detection(
  max_nodes = 300L,
  min_nodes = 10L,
  min_seed_nodes = 2L,
  initial_res = 0.5,
  threshold_type = c("prop_based", "pval_based"),
  network_threshold = 0.5,
  pval_threshold = 0.1
)
```

Arguments

`max_nodes`	Integer. Maximum number of nodes in a given community. Defaults to `300L`.
`min_nodes`	Integer. Minimum number of nodes in a given community. Defaults to `10L`.
`min_seed_nodes`	Integer. Minimum number of seed nodes within a community. Defaults to `2L`.
`initial_res`	Numeric. Initial resolution parameter to start with. Defaults to `0.5`.
`threshold_type`	String. You can chose to include a certain proportion of the network with the highest diffusion scores, or use p-values based on permutations. One of `c("prop_based", "pval_based")`. Defaults to `"prop_based"`.
`network_threshold`	Numeric. The proportion of the network to include. Used if `threshold_type = "prop_based"`. Defaults to `0.5`.
`pval_threshold`	Numeric. The maximum p-value for nodes to be included. Used if `threshold_type = "pval_based"`. Defaults to `0.1`.

Value

A named list with the following elements:

- max_nodes - Integer. Maximum number of nodes in a given community. Defaults to `300L`.

- min_nodes - Integer. Minimum number of nodes in a given community. Defaults to `10L`.
- min_seed_nodes - Integer. Minimum number of seed nodes within a community. Defaults to `2L`.
- initial_res - Numeric. Initial resolution parameter to start with. Defaults to `0.5`.
- threshold_type - String. You can chose to include a certain proportion of the network with the highest diffusion scores, or use p-values based on permutations. One of `c("prop_based", "pval_based")`. Defaults to `"prop_based"`.
- network_threshold - Numeric. The proportion of the network to include. Used if `threshold_type = "prop_based"`. Defaults to `0.5`.
- pval_threshold - Numeric. The maximum p-value for nodes to be included. Used if `threshold_type = "pval_based"`. Defaults to `0.1`.

`params_cor_graph` *Wrapper function for graph generation*

Description

Wrapper function for graph generation

Usage

```
params_cor_graph(
  epsilon = 2,
  min_cor = 0.2,
  fdr_threshold = 0.05,
  verbose = TRUE
)
```

Arguments

`epsilon`	Numeric. Defines the epsilon parameter for the radial basis function. Defaults to `2.0`.
`min_cor`	Numeric. Minimum absolute correlation that needs to be observed in either data set. Only relevant for differential correlation-based graphs. Defaults to `0.2`.
`fdr_threshold`	Numeric. Maximum FDR for the differential correlation p-value. Defaults to `0.05`.
`verbose`	Boolean. Controls verbosity of the graph generation function. Defaults to `TRUE`.

Value

A named list with the following elements:

- epsilon - Numeric. Defines the epsilon parameter for the radial basis function. Defaults to `2.0`.
- min_cor - Numeric. Minimum absolute correlation that needs to be observed in either data set. Only relevant for differential correlation-based graphs. Defaults to `0.2`.
- fdr_threshold - Numeric. Maximum FDR for the differential correlation p-value. Defaults to `0.05`.
- verbose - Boolean. Controls verbosity of the graph generation function. Defaults to `TRUE`.

`params_coremo` *Wrapper function to generate CoReMo parameters*

Description

Wrapper function to generate CoReMo parameters

Usage

```
params_coremo(
  epsilon = 2,
  k_min = 2L,
  k_max = 150L,
  min_size = NULL,
  junk_module_threshold = 0.05,
  rbf_func = c("gaussian", "inverse_quadratic", "bump"),
  cor_method = c("spearman", "pearson")
)
```

Arguments

`epsilon`	Numeric. Epsilon parameter for the chosen RBF function, see `rbf_func`. The higher, the more aggressively low correlations will be shrunk. Defaults to `2.0`.
`k_min`	Integer. Minimum and maximum number of cuts to use for the hierarchical clustering. Defaults to `2L`.
`k_max`	Integer. Minimum and maximum number of cuts to use for the hierarchical clustering. Defaults to `150L`.
`min_size`	Integer or `NULL`. Minimum size of the clusters. Smaller clusters will be combined together. Defaults to `NULL`.
`junk_module_threshold`	Numeric. Threshold for the minimum correlation to be observed in a module. Defaults to `0.05`.

`rbf_func` String. Type of RBF you wish to apply to down-weigh weak correlations. One of `c("gaussian", "inverse_quadratic", "bump")`. Defaults to `"gaussian"`.

`cor_method` String. The type of correlation to use. One of `c("spearman", "pearson")`. Defaults to `"spearman"`.

Value

A named list with the following elements:

- epsilon - Numeric. Epsilon parameter for the chosen RBF function, see `rbf_func`. The higher, the more aggressively low correlations will be shrunk. Defaults to `2.0`.
- k_min - Integer. Minimum and maximum number of cuts to use for the hierarchical clustering. Defaults to `2L`.
- k_max - Integer. Minimum and maximum number of cuts to use for the hierarchical clustering. Defaults to `150L`.
- min_size - Integer or `NULL`. Minimum size of the clusters. Smaller clusters will be combined together. Defaults to `NULL`.
- junk_module_threshold - Numeric. Threshold for the minimum correlation to be observed in a module. Defaults to `0.05`.
- rbf_func - String. Type of RBF you wish to apply to down-weigh weak correlations. One of `c("gaussian", "inverse_quadratic", "bump")`. Defaults to `"gaussian"`.
- cor_method - String. The type of correlation to use. One of `c("spearman", "pearson")`. Defaults to `"spearman"`.

`params_dgrdl` *Wrapper function to generate DGRDL parameters*

Description

Wrapper function to generate DGRDL parameters

Usage

```
params_dgrdl(
  sparsity = 5L,
  dict_size = 5L,
  alpha = 1,
  beta = 1,
  max_iter = 20L,
  k_neighbours = 5L,
  admm_iter = 5L,
  rho = 1
)
```

Arguments

`sparsity`	Integer. Sparsity constraint (max non-zero coefficients per signal) Defaults to `5L`.
`dict_size`	Integer. Dictionary size Defaults to `5L`.
`alpha`	Numeric. Sample context regularisation weight. Defaults to `1.0`.
`beta`	Numeric. Feature effect regularisation weight. Defaults to `1.0`.
`max_iter`	Integer. Maximum number of iterations for the main algorithm. Defaults to `20L`.
`k_neighbours`	Integer. Number of neighbours in the KNN graph. Defaults to `5L`.
`admm_iter`	Integer. ADMM iterations for sparse coding. Defaults to `5L`.
`rho`	Numeric. ADMM step size. Defaults to `1.0`.

Value

A named list with the following elements:

- sparsity - Integer. Sparsity constraint (max non-zero coefficients per signal) Defaults to `5L`.
- dict_size - Integer. Dictionary size Defaults to `5L`.
- alpha - Numeric. Sample context regularisation weight. Defaults to `1.0`.
- beta - Numeric. Feature effect regularisation weight. Defaults to `1.0`.
- max_iter - Integer. Maximum number of iterations for the main algorithm. Defaults to `20L`.
- k_neighbours - Integer. Number of neighbours in the KNN graph. Defaults to `5L`.
- admm_iter - Integer. ADMM iterations for sparse coding. Defaults to `5L`.
- rho - Numeric. ADMM step size. Defaults to `1.0`.

`params_dialogue_hlm` *Wrapper function for the DIALOGUE mixed model parameters*

Description

Stage two of DIALOGUE: for every ordered pair of cell types and every candidate gene, a random-intercept mixed model over samples asking whether a cell's own programme score tracks the partner cell type's expression of that gene in the same sample.

Usage

```
params_dialogue_hlm(
  min_cells_per_sample = 2L,
  use_tme_qc = TRUE,
  use_cell_quality = TRUE,
  satterthwaite = TRUE
)
```

Arguments

`min_cells_per_sample`
: Integer. Minimum cells a sample must contribute, in *both* cell types of a pair, before it takes part in that pair's models. Defaults to `2L`.

`use_tme_qc`
: Boolean. Include the partner cell type's mean quality in that sample as a fixed effect. Upstream's `tme.qc`. Defaults to `TRUE`.

`use_cell_quality`
: Boolean. Include the responding cell's own quality as a fixed effect. Upstream's `cellQ`. Defaults to `TRUE`.

`satterthwaite`
: Boolean. Compute Satterthwaite denominator degrees of freedom, as `lmerTest` does. Defaults to `TRUE`.

Details

This stage dominates the runtime, and `satterthwaite` is the knob that decides how badly. Turning it off falls back to the residual count for the denominator degrees of freedom, which is far cheaper and barely differs once a cell type has thousands of cells.

`use_cell_quality` conditions on the cell's own quality covariate, which stage one has already regressed out of the scores by ordinary least squares. The default conditions on it twice, because upstream does.

Value

A named list with the following elements:

- min_cells_per_sample - Integer. Minimum cells a sample must contribute, in *both* cell types of a pair, before it takes part in that pair's models. Defaults to `2L`.
- use_tme_qc - Boolean. Include the partner cell type's mean quality in that sample as a fixed effect. Upstream's `tme.qc`. Defaults to `TRUE`.
- use_cell_quality - Boolean. Include the responding cell's own quality as a fixed effect. Upstream's `cellQ`. Defaults to `TRUE`.
- satterthwaite - Boolean. Compute Satterthwaite denominator degrees of freedom, as `lmerTest` does. Defaults to `TRUE`.

References

Jerby-Arnon & Regev, Nature Biotechnology, 2022

`params_dialogue_pmd` *Wrapper function for the DIALOGUE decomposition parameters*

Description

Stage one of DIALOGUE: the penalised matrix decomposition that turns the per-cell-type features into multicellular programmes, and the provisional gene signatures that come off it.

Usage

```
params_dialogue_pmd(
  k = 2L,
  n_permutations = 100L,
  extra_sparse = FALSE,
  abn_c = 15L,
  p_anova = 0.05,
  centre = TRUE,
  cap = 0.01,
  spatial = FALSE,
  n_genes = 200L,
  min_ci = 0.05,
  averaging = c("median", "mean"),
  mcp_assignment_p = 0.1,
  seed = 1234L
)
```

Arguments

`k`	Integer. Number of multicellular programmes to extract. Must be at least 1. Defaults to `2L`.
`n_permutations`	Integer. Permutations backing the empirical p-value per programme. Must be at least 2. Defaults to `100L`.
`extra_sparse`	Boolean. Tune the L1 bound by permutation instead of fixing it at `sqrt(p_1) / 2`. Costs ten more fits per permutation. Defaults to `FALSE`.
`abn_c`	Integer. Minimum cells a sample must contribute, within a cell type, before it counts towards the feature-level ANOVA. Defaults to `15L`.
`p_anova`	Numeric. BH-adjusted ANOVA cutoff for keeping a feature. Must be in `(0, 1]`. Defaults to `0.05`.
`centre`	Boolean. Centre and scale the sample-level feature matrix, then winsorise it. Defaults to `TRUE`.
`cap`	Numeric. Winsorising tail fraction applied to each column. Must be in `[0, 0.5)`. Defaults to `0.01`.
`spatial`	Boolean. Spatial data: skip the ANOVA feature filter entirely. Niches are small, so a feature need not vary across them to be real. Defaults to `FALSE`.
`n_genes`	Integer. Genes taken per programme per direction when building a signature. Defaults to `200L`.
`min_ci`	Numeric. Minimum absolute correlation for a gene to enter a signature. Must be in `[0, 1]`. Defaults to `0.05`.
`averaging`	String. How cell-level features are collapsed per sample. One of `c("median", "mean")`. Defaults to `"median"`.
`mcp_assignment_p`	Numeric. Empirical p below which a cell type pair counts as connected when deciding which cell types a programme spans. Must be in `(0, 1]`. Defaults to `0.1`.

`seed` Integer. Seed for the permutation null. Defaults to `1234L`.

Details

The defaults follow upstream's `DLG.get.param`. Two knobs are worth thinking about before anything else. `k` is how many programmes you are asking for, and there is no sweep to help you pick it. `n_permutations` sets the resolution of the empirical p-value: with the default of `100` the smallest p you can observe is `0.01`, so lower it for a quick look and leave it alone for anything you intend to believe.

`averaging` is exposed and honoured here. Upstream takes the same argument and then ignores it, hard-coding column medians, so `"median"` is what every published DIALOGUE run actually used.

Value

A named list with the following elements:

- k - Integer. Number of multicellular programmes to extract. Must be at least 1. Defaults to `2L`.
- n_permutations - Integer. Permutations backing the empirical p-value per programme. Must be at least 2. Defaults to `100L`.
- extra_sparse - Boolean. Tune the L1 bound by permutation instead of fixing it at `sqrt(p_1) / 2`. Costs ten more fits per permutation. Defaults to `FALSE`.
- abn_c - Integer. Minimum cells a sample must contribute, within a cell type, before it counts towards the feature-level ANOVA. Defaults to `15L`.
- p_anova - Numeric. BH-adjusted ANOVA cutoff for keeping a feature. Must be in `(0, 1]`. Defaults to `0.05`.
- centre - Boolean. Centre and scale the sample-level feature matrix, then winsorise it. Defaults to `TRUE`.
- cap - Numeric. Winsorising tail fraction applied to each column. Must be in `[0, 0.5)`. Defaults to `0.01`.
- spatial - Boolean. Spatial data: skip the ANOVA feature filter entirely. Niches are small, so a feature need not vary across them to be real. Defaults to `FALSE`.
- n_genes - Integer. Genes taken per programme per direction when building a signature. Defaults to `200L`.
- min_ci - Numeric. Minimum absolute correlation for a gene to enter a signature. Must be in `[0, 1]`. Defaults to `0.05`.
- averaging - String. How cell-level features are collapsed per sample. One of `c("median", "mean")`. Defaults to `"median"`.
- mcp_assignment_p - Numeric. Empirical p below which a cell type pair counts as connected when deciding which cell types a programme spans. Must be in `(0, 1]`. Defaults to `0.1`.
- seed - Integer. Seed for the permutation null. Defaults to `1234L`.

References

Jerby-Arnon & Regev, Nature Biotechnology, 2022

`params_dialogue_refine` *Wrapper function for the DIALOGUE refinement parameters*

Description

Stage three of DIALOGUE: the cross-partner meta-analysis that decides which genes survive, and the non-negative refit of the programme scores onto them.

Usage

```
params_dialogue_refine(
  support_p = 0.1,
  min_support_fraction = 0.333333333333333,
  min_stratum = 5L,
  early_stop_cor = 0.95,
  permissive_p = 0.001,
  strict_p = 0.05
)
```

Arguments

`support_p`	Numeric. Adjusted p below which one partner counts as supporting a gene. Must be in `(0, 1]`. Defaults to `0.1`.
`min_support_fraction`	Numeric. Minimum supporting fraction for a stratum to enter the staged fit. Must be in `[0, 1]`. Defaults to `0.3333333333333333`.
`min_stratum`	Integer. Minimum genes in a stratum before it is worth fitting. Defaults to `5L`.
`early_stop_cor`	Numeric. Correlation between the original score and the running fit at which the staged fit stops early. Must be in `(0, 1]`. Defaults to `0.95`.
`permissive_p`	Numeric. Fisher-combined p for the permissive gene list, where a gene is carried by partner support rather than by a positive coefficient. Must be in `(0, 1]`. Defaults to `0.001`.
`strict_p`	Numeric. Fisher-combined p for the strict gene list, which also demands that every partner supports the gene. Must be in `(0, 1]`. Defaults to `0.05`.

Details

Two gene lists come out. The permissive one asks only for a Fisher-combined p below `permissive_p`; the strict one is looser on the p-value but also demands that *every* partner supports the gene. They are not nested by threshold, they are nested by evidence, and the strict list is the one to quote.

Value

A named list with the following elements:

- support_p - Numeric. Adjusted p below which one partner counts as supporting a gene. Must be in `(0, 1]`. Defaults to `0.1`.
- min_support_fraction - Numeric. Minimum supporting fraction for a stratum to enter the staged fit. Must be in `[0, 1]`. Defaults to `0.3333333333333333`.
- min_stratum - Integer. Minimum genes in a stratum before it is worth fitting. Defaults to `5L`.
- early_stop_cor - Numeric. Correlation between the original score and the running fit at which the staged fit stops early. Must be in `(0, 1]`. Defaults to `0.95`.
- permissive_p - Numeric. Fisher-combined p for the permissive gene list, where a gene is carried by partner support rather than by a positive coefficient. Must be in `(0, 1]`. Defaults to `0.001`.
- strict_p - Numeric. Fisher-combined p for the strict gene list, which also demands that every partner supports the gene. Must be in `(0, 1]`. Defaults to `0.05`.

References

Jerby-Arnon & Regev, Nature Biotechnology, 2022

`params_edger_ql` *Wrapper function for parameters for the edgeR quasi-likelihood workflow*

Description

Parameters for the edgeR quasi-likelihood chain, implemented in Rust via the `edge-rs` crate and gated against edgeR 4.8.2. Defaults are edgeR's own. The `legacy` switch picks between two genuinely different pipelines. The current route estimates its own dispersion from the most abundant genes and skips `estimateDisp()`, which is where most of the runtime went and is edgeR 4's own recommendation. The legacy route shrinks the raw residual deviance, needs a dispersion handed to it, and is the only one where the Poisson bound bites.

Usage

```
params_edger_ql(
  norm_method = c("TMM", "TMMwsp", "RLE", "upperquartile", "none"),
  filter = TRUE,
  min_mean = 0,
  robust = FALSE,
  legacy = FALSE
)
```

Arguments

norm_method	String. Library size normalisation. "none" leaves every factor at one, which is what Milo's logMS amounts to. One of c("TMM", "TMMwsp", "RLE", "upperquartile", "none"). Defaults to "TMM".
filter	Boolean. Run filterByExpr() before fitting. Turn this off for anything that is not gene expression, e.g. Milo neighbourhood counts, where the heuristic means nothing. Defaults to TRUE.
min_mean	Numeric. Drop features whose mean count across samples is below this. Applied on top of filter. Defaults to 0.0.
robust	Boolean. Robust empirical Bayes squeezing, giving outlier features their own smaller prior degrees of freedom. Defaults to FALSE.
legacy	Boolean. Take edgeR's pre-4.0 quasi-likelihood pipeline. Defaults to FALSE.

Value

A named list with the following elements:

- norm_method - String. Library size normalisation. "none" leaves every factor at one, which is what Milo's logMS amounts to. One of c("TMM", "TMMwsp", "RLE", "upperquartile", "none"). Defaults to "TMM".
- filter - Boolean. Run filterByExpr() before fitting. Turn this off for anything that is not gene expression, e.g. Milo neighbourhood counts, where the heuristic means nothing. Defaults to TRUE.
- min_mean - Numeric. Drop features whose mean count across samples is below this. Applied on top of filter. Defaults to 0.0.
- robust - Boolean. Robust empirical Bayes squeezing, giving outlier features their own smaller prior degrees of freedom. Defaults to FALSE.
- legacy - Boolean. Take edgeR's pre-4.0 quasi-likelihood pipeline. Defaults to FALSE.

References

Chen, Lun and Smyth, F1000Research, 2016

---

params_fast_cluster_default

Helper function to generate default parameters for the fast clustering for the doublet detection methods

---

Description

Helper function to generate default parameters for the fast clustering for the doublet detection methods

Usage

```
params_fast_cluster_default()
```

Value

A named list with the following elements:

- km_type - String. The type of k-means clustering. One of `c("minibatch", "standard")`. Defaults to `"minibatch"`.
- n_centroids - Integer or `NULL`. The number of centroids to use. `NULL` uses `sqrt(N_cells) * 4` centroids. Defaults to `NULL`.
- kmeans_iters - Integer. Number of maximum k-means iterations. Defaults to `100L`.
- batch_size - Integer. Max batch size, capped at `N_cells / 2` depending on the data set. Defaults to `4098L`.

`params_graph_resolution` *Wrapper function to generate resolution parameters for Leiden or Louvain clustering.*

Description

Wrapper function to generate resolution parameters for Leiden or Louvain clustering.

Usage

```
params_graph_resolution(min_res = 0.1, max_res = 10, number_res = 15L)
```

Arguments

`min_res`	Numeric. Minimum resolution to test. Defaults to `0.1`.
`max_res`	Numeric. Maximum resolution to test. Defaults to `10.0`.
`number_res`	Integer. Number of resolutions to test between the `max_res` and `min_res`. Defaults to `15L`.

Value

A named list with the following elements:

- min_res - Numeric. Minimum resolution to test. Defaults to `0.1`.
- max_res - Numeric. Maximum resolution to test. Defaults to `10.0`.
- number_res - Integer. Number of resolutions to test between the `max_res` and `min_res`. Defaults to `15L`.

`params_gsea` *Wrapper function to generate GSEA parameters*

Description

Wrapper function to generate GSEA parameters

Usage

```
params_gsea(
  min_size = 5L,
  max_size = 500L,
  gsea_param = 1,
  sample_size = 101L,
  eps = 1e-50
)
```

Arguments

`min_size`	Integer. Minimum number of genes per gene set. Defaults to `5L`.
`max_size`	Integer. Maximum number of genes per gene set. Defaults to `500L`.
`gsea_param`	Numeric. GSEA parameter. Defaults to `1.0`.
`sample_size`	Integer. Number of samples to iterate through for the multi-level implementation of fgsea. Defaults to `101L`.
`eps`	Numeric. Boundary for calculating the p-value. Used for the multi- level implementation of fgsea. Defaults to `1e-50`.

Value

A named list with the following elements:

- min_size - Integer. Minimum number of genes per gene set. Defaults to `5L`.
- max_size - Integer. Maximum number of genes per gene set. Defaults to `500L`.
- gsea_param - Numeric. GSEA parameter. Defaults to `1.0`.
- sample_size - Integer. Number of samples to iterate through for the multi-level implementation of fgsea. Defaults to `101L`.
- eps - Numeric. Boundary for calculating the p-value. Used for the multi- level implementation of fgsea. Defaults to `1e-50`.

`params_gsva` *Wrapper function to generate GSVA parameters*

Description

Wrapper function to generate GSVA parameters

Usage

```
params_gsva(
  tau = 1,
  min_size = 5L,
  max_size = 500L,
  max_diff = TRUE,
  abs_rank = FALSE
)
```

Arguments

`tau`	Numeric. Tau parameter, usual recommendation is to use `1.0` here. Larger values emphasise the tails more. Defaults to `1.0`.
`min_size`	Integer. Minimum number of genes per gene set. Defaults to `5L`.
`max_size`	Integer. Maximum number of genes per gene set. Defaults to `500L`.
`max_diff`	Boolean. Scoring mode for GSVA, if `TRUE` = difference; if `FALSE` = larger absolute value. Defaults to `TRUE`.
`abs_rank`	Boolean. If `TRUE` = pos - neg, `FALSE` = pos + neg. Defaults to `FALSE`.

Value

A named list with the following elements:

- tau - Numeric. Tau parameter, usual recommendation is to use `1.0` here. Larger values emphasise the tails more. Defaults to `1.0`.
- min_size - Integer. Minimum number of genes per gene set. Defaults to `5L`.
- max_size - Integer. Maximum number of genes per gene set. Defaults to `500L`.
- max_diff - Boolean. Scoring mode for GSVA, if `TRUE` = difference; if `FALSE` = larger absolute value. Defaults to `TRUE`.
- abs_rank - Boolean. If `TRUE` = pos - neg, `FALSE` = pos + neg. Defaults to `FALSE`.

params_hvg_defaults *Helper function to generate HVG defaults*

Description

Helper function to generate HVG defaults

Usage

```
params_hvg_defaults()
```

Value

A named list with the following elements:

- min_gene_var_pctl - Numeric. Which percentile of the highly variable genes to include. Defaults to `0.7`.
- hvg_method - String. Which method to use to identify HVG. `"scran"` runs with the default trend parameters, see `params_hvg_scran_defaults()`. One of `c("vst", "meanvarbin", "dispersion", "scran")`. Defaults to `"vst"`.
- loess_span - Numeric. In case of `"vst"` the span of the loess function. Defaults to `0.3`.
- clip_max - Numeric or `NULL`. The maximum clipping value (optional). Defaults to `NULL`.
- n_bins - Integer. The number of bins to use for the `"meanvarbin"` and `"dispersion"` HVG detection. Defaults to `20L`.
- binning_strategy - String. Which binning strategy to use for `"meanvarbin"` and `"dispersion"`. One of `c("equal_width", "equal_frequency")`. Defaults to `"equal_width"`.

params_hvg_scran_defaults *Helper function to generate the scran HVG trend defaults*

Description

Trend parameters for `method = "scran"` in `params_sc_hvg()`. They mirror scrapper's `fitVarianceTrend()` defaults before scrapper 1.8, which switched `use.min.width` to `TRUE`. The lowess span comes from `loess_span` in `params_sc_hvg()`.

Usage

```
params_hvg_scran_defaults()
```

Value

A named list with the following elements:

- mean_filter - Boolean. Shall genes below `min_mean` be left out of the trend fit. Defaults to `TRUE`.
- min_mean - Numeric. Minimum mean log-expression for a gene to enter the trend fit. Defaults to `0.1`.
- transform - Boolean. Shall the variances be fourth-root transformed before the fit. Defaults to `TRUE`.
- use_min_width - Boolean. Shall the lowess window be defined by `min_width` and `min_window_count` instead of the span. Defaults to `FALSE`.
- min_width - Numeric. Minimum window width, only used with `use_min_width = TRUE`. Defaults to `1.0`.
- min_window_count - Integer. Minimum number of genes per window, only used with `use_min_width = TRUE`. Defaults to `200L`.

`params_ica_general` *Wrapper function for standard ICA parameters*

Description

Wrapper function for standard ICA parameters

Usage

```
params_ica_general(maxit = 200L, alpha = 1, max_tol = 1e-04, verbose = FALSE)
```

Arguments

`maxit`	Integer. Maximum number of iterations for ICA. Defaults to `200L`.
`alpha`	Numeric. The alpha parameter for the logcosh version of ICA. Should be between 1 to 2. Defaults to `1.0`.
`max_tol`	Numeric. Should be `0 < max_tol < 1`. Maximum tolerance of the algorithm. Defaults to `1e-04`.
`verbose`	Boolean. Controls verbosity of the function. Defaults to `FALSE`.

Value

A named list with the following elements:

- maxit - Integer. Maximum number of iterations for ICA. Defaults to `200L`.
- alpha - Numeric. The alpha parameter for the logcosh version of ICA. Should be between 1 to 2. Defaults to `1.0`.
- max_tol - Numeric. Should be `0 < max_tol < 1`. Maximum tolerance of the algorithm. Defaults to `1e-04`.
- verbose - Boolean. Controls verbosity of the function. Defaults to `FALSE`.

params_ica_ncomp *Wrapper function for ICA ncomp iterations*

Description

Wrapper function to provide parameters through which ncomps to iterate through.

Usage

```
params_ica_ncomp(max_no_comp = 75L, steps = 5L, custom_seq = NULL)
```

Arguments

max_no_comp	Integer. Maximum number of ncomp to test. Defaults to 75L.
steps	Integer. In which steps to move from 5 onwards. Defaults to 5L.
custom_seq	Integer vector or NULL. If you wish to provide a custom version of no_comp to iterate through. If NULL, you will iterate through c(2, 3, 4, 5, 5 + step, ... max_no_co Defaults to NULL.

Value

A named list with the following elements:

- max_no_comp - Integer. Maximum number of ncomp to test. Defaults to 75L.
- steps - Integer. In which steps to move from 5 onwards. Defaults to 5L.
- custom_seq - Integer vector or NULL. If you wish to provide a custom version of no_comp to iterate through. If NULL, you will iterate through c(2, 3, 4, 5, 5 + step, ... max_no_comp Defaults to NULL.

params_ica_randomisation *Wrapper function for ICA randomisation*

Description

Wrapper function for ICA randomisation

Usage

```
params_ica_randomisation(
  cross_validate = FALSE,
  random_init = 50L,
  folds = 10L
)
```

Arguments

`cross_validate`	Boolean. Do you want to apply a cross-validation type approach and split the data into `folds` folds to assess within data stability of the component. Defaults to `FALSE`.
`random_init`	Integer. Number of random initialisations to use. Defaults to `50L`.
`folds`	Integer. Number of folds to use if `cross_validate` is set to `TRUE`. To note, you will be running `random_init * folds` ICA runs. Defaults to `10L`.

Value

A named list with the following elements:

- cross_validate - Boolean. Do you want to apply a cross-validation type approach and split the data into `folds` folds to assess within data stability of the component. Defaults to `FALSE`.
- random_init - Integer. Number of random initialisations to use. Defaults to `50L`.
- folds - Integer. Number of folds to use if `cross_validate` is set to `TRUE`. To note, you will be running `random_init * folds` ICA runs. Defaults to `10L`.

`params_kmeans_defaults` *K-mean parameter defaults.*

Description

Helper function to generate defaults for the k-mean clustering were more control is needed.

Usage

```
params_kmeans_defaults()
```

Value

A named list with the following elements:

- k_means_iter - Integer. The number of iterations to use for the clustering. Defaults to `30L`.
- k_means_init - String. The initialisation. One of `c("parallel", "random")`. Defaults to `"parallel"`.
- gemm - Boolean or `NULL`. Controls which CPU implementation is used by the method. GEMM is faster with large dimensionality. Defaults to `FALSE`.
- hamerly - Boolean or `NULL`. Shall a faster exact method be used leveraging the triangle inequality. Faster on large data sets with large numbers of centroids. Defaults to `TRUE`.

`params_knn_defaults` *Helper function to generate kNN defaults*

Description

This function generates various sensible default parameters for all of the different approximate nearest neighbours that are available within this package.

Usage

```
params_knn_defaults()
```

Value

A named list with the following elements:

- k - Integer. Number of neighbours. Defaults to `15L`.
- knn_method - String. Which method to use for the approximate nearest neighbour search. One of `c("kmknn", "hnsw", "annoy", "nndescent", "ivf", "exhaustive")`. Defaults to `"kmknn"`.
- ann_dist - String. Which distance metric to use for the approximate nearest neighbour search. One of `c("euclidean", "cosine")`. Defaults to `"euclidean"`.
- n_trees - Integer. Annoy param: number of trees to generate for Annoy. Defaults to `50L`.
- search_budget - Integer or `NULL`. Annoy param: optional search budget per tree for Annoy. If not provided, it will default to `n_tree * k * 20L`. Defaults to `NULL`.
- delta - Numeric. NNDescent param: early termination criterium for NNDescent. Defaults to `0.001`.
- diversify_prob - Numeric. NNDescent param: diversification probability for the NNDescent index. This will diversify the index at the end and identify potentially better edges. Defaults to `0.0`.
- ef_budget - Integer or `NULL`. NNDescent param: optional query budget parameter. Can accelerate querying, but at the cost of Recall. Defaults to `NULL`.
- extract_knn - Boolean. NNDescent param: hand back the graph the descent already built instead of beam searching it. Skips the query pass entirely, so it is much faster, at the cost of some recall. Rows the descent never filled come back padded with duplicate edges. Ignored by every other method. Defaults to `FALSE`.
- m - Integer. HNSW param: number of connections between layers for HNSW. Defaults to `16L`.
- ef_construction - Integer. HNSW param: size of dynamic candidate list during construction. Defaults to `200L`.
- ef_search - Integer. HNSW param: size of candidate list (higher = better recall, slower). Defaults to `100L`.

- n_list - Integer or `NULL`. IVF param: number of clusters/centroids to generate. `NULL` generates `sqrt(n)` lists. Defaults to `NULL`.
- n_probe - Integer or `NULL`. IVF param: number of clusters/centroids to query. `NULL` queries `sqrt(n_lists)` clusters. Defaults to `NULL`.

`params_label_propagation`
Wrapper function to generate label propagation parameters

Description

Wrapper function to generate label propagation parameters

Usage

```
params_label_propagation(
  alpha = 0.9,
  iter = 100L,
  tolerance = 1e-06,
  symmetrise = TRUE,
  symmetry_strategy = c("average", "avg", "min", "max"),
  max_hops = NULL
)
```

Arguments

`alpha`	Numeric. Controls the spreading strength. Higher values anchor labelled nodes more strongly to their original label. Defaults to `0.9`.
`iter`	Integer. Maximum number of iterations to run. Defaults to `100L`.
`tolerance`	Numeric. Convergence threshold. Stops early if the maximum change across all nodes falls below this value. Defaults to `1e-06`.
`symmetrise`	Boolean. Whether to symmetrise the graph. Defaults to `TRUE`.
`symmetry_strategy`	String. Strategy to resolve weight conflicts when symmetrising. Only relevant when `symmetrise = TRUE` and edge weights are provided. One of `c("average", "avg", "min", "max")`. Defaults to `"average"`.
`max_hops`	Integer or `NULL`. If provided, restricts label spreading to nodes within this many hops of any labelled node. Nodes beyond this limit remain all-zeroes. Defaults to `NULL`.

Value

A named list with the following elements:

- alpha - Numeric. Controls the spreading strength. Higher values anchor labelled nodes more strongly to their original label. Defaults to `0.9`.

- iter - Integer. Maximum number of iterations to run. Defaults to `100L`.
- tolerance - Numeric. Convergence threshold. Stops early if the maximum change across all nodes falls below this value. Defaults to `1e-06`.
- symmetrise - Boolean. Whether to symmetrise the graph. Defaults to `TRUE`.
- symmetry_strategy - String. Strategy to resolve weight conflicts when symmetrising. Only relevant when `symmetrise = TRUE` and edge weights are provided. One of `c("average", "avg", "min", "max")`. Defaults to `"average"`.
- max_hops - Integer or `NULL`. If provided, restricts label spreading to nodes within this many hops of any labelled node. Nodes beyond this limit remain all-zeroes. Defaults to `NULL`.

`params_lda` *Wrapper function for the LDA parameters*

Description

Solver options for the variational Bayes latent Dirichlet allocation, see `run_lda()`.

Usage

```
params_lda(
  alpha = 50,
  alpha_by_topic = TRUE,
  eta = 0.1,
  eta_by_topic = FALSE,
  max_iter = 150L,
  tol = 0.001,
  inner_max_iter = 100L,
  inner_tol = 0.001,
  check_every = 10L,
  learning = c("batch", "online"),
  batch_size = 1024L,
  n_epochs = 10L
)
```

Arguments

`alpha`	Numeric. Dirichlet prior on the document-topic distributions. Defaults to `50.0`.
`alpha_by_topic`	Boolean. Shall `alpha` be divided by the topic count. Defaults to `TRUE`.
`eta`	Numeric. Dirichlet prior on the topic-term distributions. Defaults to `0.1`.
`eta_by_topic`	Boolean. Shall `eta` be divided by the topic count. Defaults to `FALSE`.
`max_iter`	Integer. Maximum outer iterations. Ignored by the online variant, which counts epochs instead. Defaults to `150L`.

`tol` Numeric. Relative change in the bound below which the solver stops. Defaults to `0.001`.

`inner_max_iter` Integer. Maximum fixed-point iterations of the per-document E-step. Defaults to `100L`.

`inner_tol` Numeric. Relative L1 change in the variational parameters below which the per-document E-step stops. Defaults to `0.001`.

`check_every` Integer. Iterations between bound evaluations. Defaults to `10L`.

`learning` String. Batch or online variational inference. One of `c("batch", "online")`. Defaults to `"batch"`.

`batch_size` Integer. Documents per mini-batch. Online only. Defaults to `1024L`.

`n_epochs` Integer. Passes over the corpus. Online only. Defaults to `10L`.

Details

The defaults follow pycisTopic, so the knobs mean the same thing on both sides. `alpha_by_topic = TRUE` turns `alpha` into the Griffiths and Steyvers `50 / k` heuristic that cisTopic defaults to; set it to `FALSE` if you want `alpha` taken literally.

`learning = "batch"` sweeps every document once per iteration and is monotone in the bound. `"online"` takes decaying steps from shuffled mini-batches, which reaches a usable fit in far fewer passes on a large corpus at the cost of that guarantee. `batch_size` and `n_epochs` are only read by the online variant.

Value

A named list with the following elements:

- alpha - Numeric. Dirichlet prior on the document-topic distributions. Defaults to `50.0`.
- alpha_by_topic - Boolean. Shall `alpha` be divided by the topic count. Defaults to `TRUE`.
- eta - Numeric. Dirichlet prior on the topic-term distributions. Defaults to `0.1`.
- eta_by_topic - Boolean. Shall `eta` be divided by the topic count. Defaults to `FALSE`.
- max_iter - Integer. Maximum outer iterations. Ignored by the online variant, which counts epochs instead. Defaults to `150L`.
- tol - Numeric. Relative change in the bound below which the solver stops. Defaults to `0.001`.
- inner_max_iter - Integer. Maximum fixed-point iterations of the per-document E-step. Defaults to `100L`.
- inner_tol - Numeric. Relative L1 change in the variational parameters below which the per-document E-step stops. Defaults to `0.001`.
- check_every - Integer. Iterations between bound evaluations. Defaults to `10L`.
- learning - String. Batch or online variational inference. One of `c("batch", "online")`. Defaults to `"batch"`.
- batch_size - Integer. Documents per mini-batch. Online only. Defaults to `1024L`.
- n_epochs - Integer. Passes over the corpus. Online only. Defaults to `10L`.

References

Hoffman, Blei and Bach, NIPS, 2010; Bravo Gonzalez-Blas, et al., Nat Methods, 2019

`params_ligand_target` *Parameters for ligand to target influence computation*

Description

Parameters for ligand to target influence computation

Usage

```
params_ligand_target(
  lr_sig_hub = 0,
  gr_hub = 0,
  ltf_cutoff = 0.99,
  damping_factor = 0.5,
  tol = 1e-12,
  max_iter = 1000L,
  topology_correction = FALSE,
  secondary_targets = FALSE
)
```

Arguments

`lr_sig_hub`	Numeric. Numeric in `[0, 1]`. Hub correction strength for the ligand-receptor / signalling layer. 0 disables correction. Defaults to `0.0`.
`gr_hub`	Numeric. Numeric in `[0, 1]`. Hub correction strength for the gene regulatory layer. 0 disables correction. Defaults to `0.0`.
`ltf_cutoff`	Numeric. Numeric in `[0, 1]`. Quantile cutoff applied to the intermediate ligand-to-TF matrix. Defaults to `0.99`.
`damping_factor`	Numeric. Numeric in `[0, 1]`. PageRank-style damping factor. Defaults to `0.5`.
`tol`	Numeric. Numeric > 0. Convergence tolerance (L1 change) for the propagation step. With `ltf_cutoff > 0` the output is only stable once the propagation has converged well below the score gaps around the quantile, so keep this tight. Defaults to `1e-12`.
`max_iter`	Integer. Integer $>= 1$. Maximum iterations for the propagation step. Defaults to `1000L`.
`topology_correction`	Boolean. Apply topology correction. Defaults to `FALSE`.
`secondary_targets`	Boolean. Run a second round through targets. Defaults to `FALSE`.

Value

A named list with the following elements:

- lr_sig_hub - Numeric. Numeric in `[0, 1]`. Hub correction strength for the ligand-receptor / signalling layer. 0 disables correction. Defaults to `0.0`.
- gr_hub - Numeric. Numeric in `[0, 1]`. Hub correction strength for the gene regulatory layer. 0 disables correction. Defaults to `0.0`.
- ltf_cutoff - Numeric. Numeric in `[0, 1]`. Quantile cutoff applied to the intermediate ligand-to-TF matrix. Defaults to `0.99`.
- damping_factor - Numeric. Numeric in `[0, 1]`. PageRank-style damping factor. Defaults to `0.5`.
- tol - Numeric. Numeric > 0. Convergence tolerance (L1 change) for the propagation step. With `ltf_cutoff > 0` the output is only stable once the propagation has converged well below the score gaps around the quantile, so keep this tight. Defaults to `1e-12`.
- max_iter - Integer. Integer $>= 1$. Maximum iterations for the propagation step. Defaults to `1000L`.
- topology_correction - Boolean. Apply topology correction. Defaults to `FALSE`.
- secondary_targets - Boolean. Run a second round through targets. Defaults to `FALSE`.

`params_limma_voom` *Wrapper function for parameters for the limma-voom workflow*

Description

Parameters for the limma linear model chain, implemented in Rust via the `edge-rs` crate and gated against limma 3.66.0. Defaults are limma's own, except for `filter`: inside `BulkDge()` the genes were already filtered by `qc_bulk_dge()`. `route = "voom"` is `voomLmFit()`: precision weights from the mean-variance trend, then weighted least squares. `route = "trend"` is limma-trend: log-CPM straight into `lmFit()`, with the trend absorbed by `eBayes(trend = TRUE)`. The empirical Bayes trend follows the route.

Usage

```
params_limma_voom(
  route = c("voom", "trend"),
  norm_method = c("TMM", "TMMwsp", "RLE", "upperquartile", "none"),
  filter = FALSE,
  min_mean = 0,
  robust = FALSE,
  prior_count = NULL,
  adaptive_span = TRUE,
  span = 0.5,
  proportion = 0.01
)
```

Arguments

route	String. Whether to run the voom or the limma-trend route. One of c("voom", "trend"). Defaults to "voom".
norm_method	String. Library size normalisation. One of c("TMM", "TMMwsp", "RLE", "upperquartile", "none"). Defaults to "TMM".
filter	Boolean. Run filterByExpr() before fitting. Defaults to FALSE.
min_mean	Numeric. Drop genes whose mean count across samples is below this. Applied on top of filter. Defaults to 0.0.
robust	Boolean. Robust empirical Bayes, eBayes(robust = TRUE). Defaults to FALSE.
prior_count	Numeric or NULL. Count added before the log. NULL takes the route's own default, 0.5 for voom and 2 for trend. Defaults to NULL.
adaptive_span	Boolean. Derive the lowess span from the number of genes, as limma does since 3.56. Only used by voom. Defaults to TRUE.
span	Numeric. Lowess span for the voom trend, only read if adaptive_span = FALSE. Defaults to 0.5.
proportion	Numeric. Assumed proportion of differentially expressed genes, only used for the B-statistic. Defaults to 0.01.

Value

A named list with the following elements:

- route - String. Whether to run the voom or the limma-trend route. One of c("voom", "trend"). Defaults to "voom".
- norm_method - String. Library size normalisation. One of c("TMM", "TMMwsp", "RLE", "upperquartile", "none"). Defaults to "TMM".
- filter - Boolean. Run filterByExpr() before fitting. Defaults to FALSE.
- min_mean - Numeric. Drop genes whose mean count across samples is below this. Applied on top of filter. Defaults to 0.0.
- robust - Boolean. Robust empirical Bayes, eBayes(robust = TRUE). Defaults to FALSE.
- prior_count - Numeric or NULL. Count added before the log. NULL takes the route's own default, 0.5 for voom and 2 for trend. Defaults to NULL.
- adaptive_span - Boolean. Derive the lowess span from the number of genes, as limma does since 3.56. Only used by voom. Defaults to TRUE.
- span - Numeric. Lowess span for the voom trend, only read if adaptive_span = FALSE. Defaults to 0.5.
- proportion - Numeric. Assumed proportion of differentially expressed genes, only used for the B-statistic. Defaults to 0.01.

References

Law, et al., Genome Biol, 2014; Smyth, Stat Appl Genet Mol Biol, 2004

`params_meld` *Constructor for MELD parameters*

Description

Constructor for MELD parameters

Usage

```
params_meld(
  beta = 60,
  offset = 0,
  order = 1,
  filter = c("heat", "laplacian"),
  chebyshev_order = 50L,
  lap_type = c("combinatorial", "normalised"),
  normalise_indicators = TRUE,
  knn = list()
)
```

Arguments

`beta`	Numeric. Smoothing strength; larger values produce smoother densities. Must be strictly positive. Defaults to `60.0`.
`offset`	Numeric. Shift of the filter centre in the rescaled spectrum. Must be in `[0, 1]`. Defaults to `0.0`.
`order`	Numeric. Filter falloff sharpness; larger values approach a square low-pass. Must be strictly positive. Defaults to `1.0`.
`filter`	String. Filter family to use. One of `c("heat", "laplacian")`. Defaults to `"heat"`.
`chebyshev_order`	Integer. Number of Chebyshev coefficients (polynomial terms). Must be >= 2. Defaults to `50L`.
`lap_type`	String. Type of Laplacian to use for spectral filtering. One of `c("combinatorial", "normalised")`. Defaults to `"combinatorial"`.
`normalise_indicators`	Boolean. If `TRUE`, each column of the indicator matrix is divided by its column sum before filtering, making cross-condition densities comparable regardless of cells-per-condition. Defaults to `TRUE`.
`knn`	List. Optional overrides for kNN parameters. See `params_knn_defaults()` for available parameters: `k`, `knn_method`, `ann_dist`, `search_budget`, `n_trees`, `delta`, `diversify_prob`, `ef_budget`, `m`, `ef_construction`, `ef_search`, `n_list` and `n_probe`. See `params_knn_defaults()` for the available elements. Defaults to `list()`.

Value

A named list with the following elements:

- The elements of `params_knn_defaults()`, overridden by `knn`, spliced in at this position.
- beta - Numeric. Smoothing strength; larger values produce smoother densities. Must be strictly positive. Defaults to `60.0`.
- offset - Numeric. Shift of the filter centre in the rescaled spectrum. Must be in `[0, 1]`. Defaults to `0.0`.
- order - Numeric. Filter falloff sharpness; larger values approach a square low-pass. Must be strictly positive. Defaults to `1.0`.
- filter - String. Filter family to use. One of `c("heat", "laplacian")`. Defaults to `"heat"`.
- chebyshev_order - Integer. Number of Chebyshev coefficients (polynomial terms). Must be >= 2. Defaults to `50L`.
- lap_type - String. Type of Laplacian to use for spectral filtering. One of `c("combinatorial", "normalised")`. Defaults to `"combinatorial"`.
- normalise_indicators - Boolean. If `TRUE`, each column of the indicator matrix is divided by its column sum before filtering, making cross-condition densities comparable regardless of cells-per-condition. Defaults to `TRUE`.

---

params_module_membership

Wrapper function to generate module membership parameters

Description

Controls how a `gene x k` loading matrix from ICA, NMF or DGRDL is turned into module membership. Genes are kept where they sit in the tail of a component's loading distribution, which means membership is not exclusive: a gene loading strongly on three components belongs to three modules. Genes in no tail belong to nothing, which is the background category an argmax assignment cannot give you.

Usage

```
params_module_membership(
  method = c("zscore", "fdr"),
  cutoff = 3,
  fdr = 0.05,
  tails = c("auto", "upper", "both"),
  scaling = c("robust", "standard")
)
```

Arguments

`method`	String. `"zscore"` standardises each component and keeps `abs(z) > cutoff`. `"fdr"` converts to two-sided p-values against a Normal null fitted the same way, Benjamini-Hochberg adjusts, and keeps `padj < fdr`. One of `c("zscore", "fdr")`. Defaults to `"zscore"`.
`cutoff`	Numeric. Absolute z threshold for `method = "zscore"`. Defaults to `3.0`.
`fdr`	Numeric. Adjusted p-value threshold for `method = "fdr"`. Defaults to `0.05`.
`tails`	String. `"auto"` uses an upper-tail-only test when every loading is non-negative (the NMF case) and a two-sided one otherwise. `"upper"` and `"both"` force the choice. One of `c("auto", "upper", "both")`. Defaults to `"auto"`.
`scaling`	String. `"robust"` centres and scales each component by its median and MAD. `"standard"` uses the mean and standard deviation instead, which is stricter and less forgiving of skewed loadings (e.g. NMF). One of `c("robust", "standard")`. Defaults to `"robust"`.

Value

A named list with the following elements:

- method - String. `"zscore"` standardises each component and keeps `abs(z) > cutoff`. `"fdr"` converts to two-sided p-values against a Normal null fitted the same way, Benjamini-Hochberg adjusts, and keeps `padj < fdr`. One of `c("zscore", "fdr")`. Defaults to `"zscore"`.
- cutoff - Numeric. Absolute z threshold for `method = "zscore"`. Defaults to `3.0`.
- fdr - Numeric. Adjusted p-value threshold for `method = "fdr"`. Defaults to `0.05`.
- tails - String. `"auto"` uses an upper-tail-only test when every loading is non-negative (the NMF case) and a two-sided one otherwise. `"upper"` and `"both"` force the choice. One of `c("auto", "upper", "both")`. Defaults to `"auto"`.
- scaling - String. `"robust"` centres and scales each component by its median and MAD. `"standard"` uses the mean and standard deviation instead, which is stricter and less forgiving of skewed loadings (e.g. NMF). One of `c("robust", "standard")`. Defaults to `"robust"`.

References

Biton, et al., Cell Rep, 2014

`params_nebula` *Wrapper function for parameters for NEBULA*

Description

Parameters for the NEBULA negative binomial gamma mixed model, implemented in Rust via the `edge-rs` crate and ported from the `nebula` package's own C++. Defaults are the R package's own. NEBULA splits the variance into a subject-level random effect and a cell-level overdispersion. Run it on meta cells and the cell-level term becomes the spread between aggregates within a subject rather than between cells, so it is smaller and absorbs whatever the aggregation smoothed away. The subject-level term keeps its meaning either way.

Usage

```
params_nebula(
  nebula_method = c("ln", "hl"),
  min_sigma = 1e-04,
  min_phi = 1e-04,
  max_sigma = 10,
  max_phi = 1000,
  cutoff_cell = 20,
  kappa = 800,
  cpc = 0.005,
  mincp = 5L,
  min_subjects = 0L,
  reml = FALSE,
  eps = 1e-06,
  gene_batch_size = 1000L,
  shrink_dispersion = TRUE
)
```

Arguments

`nebula_method`	String. Which variant to run. NEBULA downgrades `"ln"` to `"hl"` below 30 cells per subject, as the R package does. One of `c("ln", "hl")`. Defaults to `"ln"`.
`min_sigma`	Numeric. Lower bound on the subject-level overdispersion. Defaults to `1e-04`.
`min_phi`	Numeric. Lower bound on the cell-level overdispersion. Defaults to `1e-04`.
`max_sigma`	Numeric. Upper bound on the subject-level overdispersion. Defaults to `10.0`.
`max_phi`	Numeric. Upper bound on the cell-level overdispersion. Defaults to `1000.0`.
`cutoff_cell`	Numeric. Refit both overdispersions when the product of the cells per subject and the estimated `phi` falls below this. Defaults to `20.0`.

`kappa` Numeric. Threshold on NEBULA's `kappa_obs` above which the subject-level overdispersion from stage one is trusted as is. Defaults to `800.0`.

`cpc` Numeric. Drop a gene whose mean count per cell is at most this. Defaults to `0.005`.

`mincp` Integer. Drop a gene expressed in fewer than this many cells. Defaults to `5L`.

`min_subjects` Integer. Drop a gene that fewer than this many subjects express, a subject expressing it when its own mean count per cell is above `cpc`. `cpc` and `mincp` pool every cell, so one subject can carry a gene through on its own. `0` switches the check off, as in the `nebula` package. Defaults to `0L`.

`reml` Boolean. Estimate the overdispersions by restricted maximum likelihood. The R package only honours this for `NBLMM`, which the Rust port does not implement, so this arm has not been validated against an R reference. Leave it off unless you know why you want it. Defaults to `FALSE`.

`eps` Numeric. Absolute stopping tolerance for the optimiser. Defaults to `1e-06`.

`gene_batch_size`
Integer. Genes read and fitted per batch. Bounds how much of the store is resident at once and changes nothing about the answer, since NEBULA is gene-independent. Defaults to `1000L`.

`shrink_dispersion`
Boolean. Shrink the cell-level overdispersions towards an empirical Bayes prior once the sweep is done. Defaults to `TRUE`.

Value

A named list with the following elements:

- nebula_method - String. Which variant to run. NEBULA downgrades `"ln"` to `"hl"` below 30 cells per subject, as the R package does. One of `c("ln", "hl")`. Defaults to `"ln"`.
- min_sigma - Numeric. Lower bound on the subject-level overdispersion. Defaults to `1e-04`.
- min_phi - Numeric. Lower bound on the cell-level overdispersion. Defaults to `1e-04`.
- max_sigma - Numeric. Upper bound on the subject-level overdispersion. Defaults to `10.0`.
- max_phi - Numeric. Upper bound on the cell-level overdispersion. Defaults to `1000.0`.
- cutoff_cell - Numeric. Refit both overdispersions when the product of the cells per subject and the estimated `phi` falls below this. Defaults to `20.0`.
- kappa - Numeric. Threshold on NEBULA's `kappa_obs` above which the subject-level overdispersion from stage one is trusted as is. Defaults to `800.0`.
- cpc - Numeric. Drop a gene whose mean count per cell is at most this. Defaults to `0.005`.
- mincp - Integer. Drop a gene expressed in fewer than this many cells. Defaults to `5L`.

- min_subjects - Integer. Drop a gene that fewer than this many subjects express, a subject expressing it when its own mean count per cell is above `cpc`. `cpc` and `mincp` pool every cell, so one subject can carry a gene through on its own. `0` switches the check off, as in the `nebula` package. Defaults to `0L`.
- reml - Boolean. Estimate the overdispersions by restricted maximum likelihood. The R package only honours this for `NBLMM`, which the Rust port does not implement, so this arm has not been validated against an R reference. Leave it off unless you know why you want it. Defaults to `FALSE`.
- eps - Numeric. Absolute stopping tolerance for the optimiser. Defaults to `1e-06`.
- gene_batch_size - Integer. Genes read and fitted per batch. Bounds how much of the store is resident at once and changes nothing about the answer, since NEBULA is gene-independent. Defaults to `1000L`.
- shrink_dispersion - Boolean. Shrink the cell-level overdispersions towards an empirical Bayes prior once the sweep is done. Defaults to `TRUE`.

References

He, et al., Commun Biol, 2021

`params_nmf_consensus` *Wrapper function for consensus NMF parameters*

Description

Controls the clustering step of consensus NMF: the components of every restart are pooled, outliers are dropped by local density, and the survivors are k-means clustered into `k` groups whose median becomes the consensus factor.

Usage

```
params_nmf_consensus(
  consensus_target = c("h", "w"),
  n_neighbours = 0L,
  density_threshold = 0.5,
  kmeans_iters = 100L,
  kmeans_n_init = 3L
)
```

Arguments

`consensus_target`
: String. `"h"` clusters the gene programmes (the spectra), which is what cNMF does and what you almost always want. `"w"` clusters in sample space instead, which on single cell data means cell space: it pools a dense `(k * n_runs) x n_cells` matrix and runs an exhaustive cosine search over it, so it gets expensive fast. One of `c("h", "w")`. Defaults to `"h"`.

`n_neighbours` Integer. Neighbours used for the local density estimate. `0L` picks `ceiling(0.3 * n_runs)` for you. Defaults to `0L`.

`density_threshold` Numeric. Components whose mean cosine distance to their neighbours exceeds this are dropped as unstable. Cosine distance cannot exceed 2, so any value `>= 2` disables the filter entirely. Defaults to `0.5`.

`kmeans_iters` Integer. Maximum k-means iterations. Defaults to `100L`.

`kmeans_n_init` Integer. Number of k-means restarts. Defaults to `3L`.

Value

A named list with the following elements:

- consensus_target - String. `"h"` clusters the gene programmes (the spectra), which is what cNMF does and what you almost always want. `"w"` clusters in sample space instead, which on single cell data means cell space: it pools a dense `(k * n_runs) x n_cells` matrix and runs an exhaustive cosine search over it, so it gets expensive fast. One of `c("h", "w")`. Defaults to `"h"`.
- n_neighbours - Integer. Neighbours used for the local density estimate. `0L` picks `ceiling(0.3 * n_runs)` for you. Defaults to `0L`.
- density_threshold - Numeric. Components whose mean cosine distance to their neighbours exceeds this are dropped as unstable. Cosine distance cannot exceed 2, so any value `>= 2` disables the filter entirely. Defaults to `0.5`.
- kmeans_iters - Integer. Maximum k-means iterations. Defaults to `100L`.
- kmeans_n_init - Integer. Number of k-means restarts. Defaults to `3L`.

References

Kotliar et al., eLife, 2019

`params_nmf_hals` *Wrapper function for NMF (HALS) parameters*

Description

Wrapper function for NMF (HALS) parameters

Usage

```
params_nmf_hals(
  max_iter = 250L,
  tol = 1e-04,
  eps = 1e-10,
  check_every = 10L,
  nmf_init = c("nndsvd", "svd", "random")
)
```

Arguments

`max_iter`	Integer. Maximum number of HALS iterations. Defaults to `250L`.
`tol`	Numeric. Convergence tolerance on the relative change in reconstruction loss. Defaults to `1e-04`.
`eps`	Numeric. Numerical floor for non-negativity / division safety. Defaults to `1e-10`.
`check_every`	Integer. Convergence check interval in iterations. Defaults to `10L`.
`nmf_init`	String. `"nndsvd"` and `"svd"` both map to deterministic NNDSVD initialisation; `"random"` uses random non-negative draws. For stabilised (multi-run) NMF this field is ignored and random init is always used. One of `c("nndsvd", "svd", "random")`. Defaults to `"nndsvd"`.

Value

A named list with the following elements:

- max_iter - Integer. Maximum number of HALS iterations. Defaults to `250L`.
- tol - Numeric. Convergence tolerance on the relative change in reconstruction loss. Defaults to `1e-04`.
- eps - Numeric. Numerical floor for non-negativity / division safety. Defaults to `1e-10`.
- check_every - Integer. Convergence check interval in iterations. Defaults to `10L`.
- nmf_init - String. `"nndsvd"` and `"svd"` both map to deterministic NNDSVD initialisation; `"random"` uses random non-negative draws. For stabilised (multi-run) NMF this field is ignored and random init is always used. One of `c("nndsvd", "svd", "random")`. Defaults to `"nndsvd"`.

`params_norm_doublets_defaults`

Helper function to generate normalisation defaults for doublet detection.

Description

Helper function to generate normalisation defaults for doublet detection.

Usage

```
params_norm_doublets_defaults()
```

Value

A named list with the following elements:

- log_transform - Boolean. Shall the counts be log-normalised. Defaults to `TRUE`.
- mean_center - Boolean. Shall mean centring be applied. Defaults to `FALSE`.
- normalise_variance - Boolean. Shall the variance be normalised. Defaults to `FALSE`.
- target_size - Numeric. Target library size. Defaults to `1000000.0`.

params_pca_defaults *Helper function to generate default parameters for PCA*

Description

Helper function to generate default parameters for PCA

Usage

```
params_pca_defaults()
```

Value

A named list with the following elements:

- no_pcs - Integer. Number of PCs to consider. Defaults to `30L`.
- svd_solver - String. Which solver to use. `"randomised"` (default) is a randomised SVD, approximate in the trailing components. `"covariance"` builds the gene x gene cross-product and eigendecomposes it. `"exact"` is Lanczos on the sparse path and a full SVD on the dense one. See `params_sc_pca()` for the trade-offs. One of `c("randomised", "covariance", "exact")`. Defaults to `"randomised"`.

params_ppca *Wrapper function for probabilistic PCA parameters*

Description

Parameters for PCA with missing values via probabilistic PCA, see `run_ppca()`. Defaults follow pcaMethods.

Usage

```
params_ppca(
  n_pcs = 2L,
  max_iter = 1000L,
  tol = 1e-05,
  seed = 42L,
  centre = TRUE,
  scale = FALSE
)
```

Arguments

n_pcs	Integer. Number of principal components. Defaults to 2L.
max_iter	Integer. Maximum number of EM iterations. Defaults to 1000L.
tol	Numeric. Relative change in the objective below which EM stops. Defaults to 1e-05.
seed	Integer. Seed for the random initial loadings. Defaults to 42L.
centre	Boolean. Shall the observed column means be subtracted first. Defaults to TRUE.
scale	Boolean. Shall the columns be divided by their observed standard deviation first (pcaMethods' "uv"). Defaults to FALSE.

Value

A named list with the following elements:

- n_pcs - Integer. Number of principal components. Defaults to 2L.
- max_iter - Integer. Maximum number of EM iterations. Defaults to 1000L.
- tol - Numeric. Relative change in the objective below which EM stops. Defaults to 1e-05.
- seed - Integer. Seed for the random initial loadings. Defaults to 42L.
- centre - Boolean. Shall the observed column means be subtracted first. Defaults to TRUE.
- scale - Boolean. Shall the columns be divided by their observed standard deviation first (pcaMethods' "uv"). Defaults to FALSE.

References

Tipping and Bishop, J R Stat Soc B, 1999; Stacklies, et al., Bioinformatics, 2007

---

params_sc_apr *Wrapper function for analytic Pearson residual parameters*

Description

The closed-form alternative to scTransform: one shared dispersion instead of a fitted model per gene. Much cheaper, and on most data sets it ranks genes about as well.

Usage

```
params_sc_apr(theta = 100, min_cells = 5L, clip_min = NULL, clip_max = NULL)
```

Arguments

`theta`	Numeric. The shared negative binomial dispersion. `Inf` gives the Poisson limit. Defaults to `100.0`.
`min_cells`	Integer. Minimum number of cells a gene must be detected in to be retained. `0L` keeps everything. Defaults to `5L`.
`clip_min`	Numeric or `NULL`. Lower residual clipping bound. `NULL` uses `-sqrt(n_cells)`. Must be given together with `clip_max`. Defaults to `NULL`.
`clip_max`	Numeric or `NULL`. Upper residual clipping bound. `NULL` uses `sqrt(n_cells)`. Must be given together with `clip_min`. Defaults to `NULL`.

Value

A named list with the following elements:

- theta - Numeric. The shared negative binomial dispersion. `Inf` gives the Poisson limit. Defaults to `100.0`.
- min_cells - Integer. Minimum number of cells a gene must be detected in to be retained. `0L` keeps everything. Defaults to `5L`.
- clip_min - Numeric or `NULL`. Lower residual clipping bound. `NULL` uses `-sqrt(n_cells)`. Must be given together with `clip_max`. Defaults to `NULL`.
- clip_max - Numeric or `NULL`. Upper residual clipping bound. `NULL` uses `sqrt(n_cells)`. Must be given together with `clip_min`. Defaults to `NULL`.

References

Lause, Berens and Kobak, Genome Biology, 2021.

`params_sc_aucell` *Wrapper function for parameters for AUCell*

Description

The three statistics consume the same within-cell ranking, but weight it very differently. `"wilcox"` is a pure function of the gene set's rank sum, so a gene at rank 2 and one at rank 200 count for almost the same thing. `"recovery"` and `"ap"` are top-heavy. For SCENIC use `"recovery"`, the default. `"wilcox"` is a bixverse addition and its flatter score does not separate into on/off populations, so it binarises badly. Ranking ties are averaged (midranks) rather than broken at random the way AUCell does it, which is why there is no need for SCENIC's trick of setting the cutoff to the 1st percentile of genes detected per cell. Undetected genes all collapse onto one rank well outside any sensible `max_rank`. Note the recovery AUC is normalised by `max_rank * length(gene_set)`, matching pySCENIC and AUCell's `normAUC = FALSE`. Modern AUCell divides by the attainable maximum instead, so absolute values differ by a per-gene-set constant. Cell ordering within a regulon is unaffected.

Usage

```
params_sc_aucell(
  auc_type = c("recovery", "wilcox", "ap"),
  max_rank = NULL,
  standardise = FALSE
)
```

Arguments

auc_type	String. Which statistic to calculate. "wilcox" is the AUC derived from the Mann-Whitney U statistic over the full ranking, with the null at 0.5 for any gene set size. "recovery" is the recovery-curve AUC under a rank cutoff, i.e. the actual AUCell statistic of Aibar, et al. "ap" is average precision, the most top-heavy of the three, but its null tracks the gene set prevalence so raw values are not comparable across gene sets of different size unless standardise is on. One of c("recovery", "wilcox", "ap"). Defaults to "recovery".
max_rank	Numeric or NULL. Rank cutoff for "recovery", counted from the top of the within-cell ranking. If NULL, resolves to the top 5% of the gene universe, following Aibar, et al. Ignored by the other two statistics. Defaults to NULL.
standardise	Boolean. Shall each gene set's scores be z-scored across the cells. This is what makes "ap" comparable across gene sets of different size. Defaults to FALSE.

Value

A named list with the following elements:

- auc_type - String. Which statistic to calculate. "wilcox" is the AUC derived from the Mann-Whitney U statistic over the full ranking, with the null at 0.5 for any gene set size. "recovery" is the recovery-curve AUC under a rank cutoff, i.e. the actual AUCell statistic of Aibar, et al. "ap" is average precision, the most top-heavy of the three, but its null tracks the gene set prevalence so raw values are not comparable across gene sets of different size unless standardise is on. One of c("recovery", "wilcox", "ap"). Defaults to "recovery".
- max_rank - Numeric or NULL. Rank cutoff for "recovery", counted from the top of the within-cell ranking. If NULL, resolves to the top 5% of the gene universe, following Aibar, et al. Ignored by the other two statistics. Defaults to NULL.
- standardise - Boolean. Shall each gene set's scores be z-scored across the cells. This is what makes "ap" comparable across gene sets of different size. Defaults to FALSE.

References

Aibar, et al., Nat Methods, 2017

`params_sc_bbknn` *Wrapper function for the BBKNN parameters*

Description

Wrapper function for the BBKNN parameters

Usage

```
params_sc_bbknn(
  neighbours_within_batch = 3L,
  set_op_mix_ratio = 1,
  local_connectivity = 1,
  trim = NULL,
  knn = list()
)
```

Arguments

`neighbours_within_batch`	Integer. Number of neighbours to consider per batch. Defaults to `3L`.
`set_op_mix_ratio`	Numeric. Mixing ratio between union (1.0) and intersection (0.0). Defaults to `1.0`.
`local_connectivity`	Numeric. UMAP connectivity computation parameter, how many nearest neighbours of each cell are assumed to be fully connected. Defaults to `1.0`.
`trim`	Integer or `NULL`. Trim the neighbours of each cell to these many top connectivities. May help with population independence and improve the tidiness of clustering. If `NULL`, it defaults to `10 * neighbours_within_batch`. Defaults to `NULL`.
`knn`	List. Optional overrides for kNN parameters. See `params_knn_defaults()` for available parameters: `k`, `knn_method`, `ann_dist`, `search_budget`, `n_trees`, `delta`, `diversify_prob`, `ef_budget`, `extract_knn`, `m`, `ef_construction`, `ef_search`, `n_list` and `n_probe`. See `params_knn_defaults()` for the available elements. Defaults to `list()`.

Value

A named list with the following elements:

- neighbours_within_batch - Integer. Number of neighbours to consider per batch. Defaults to `3L`.
- set_op_mix_ratio - Numeric. Mixing ratio between union (1.0) and intersection (0.0). Defaults to `1.0`.
- local_connectivity - Numeric. UMAP connectivity computation parameter, how many nearest neighbours of each cell are assumed to be fully connected. Defaults to `1.0`.

- trim - Integer or `NULL`. Trim the neighbours of each cell to these many top connectivities. May help with population independence and improve the tidiness of clustering. If `NULL`, it defaults to `10 * neighbours_within_batch`. Defaults to `NULL`.
- The elements of `params_knn_defaults()`, overridden by `knn`, spliced in at this position.

`params_sc_bonsai` *Wrapper function for Bonsai parameters*

Description

Parameters for `bonsai_sc()`. Sanity turns the raw counts into posterior log fold changes with error bars, Bonsai builds a tree over the cells from those, and the tree is laid out in 2D for plotting. The Sanity defaults come from sanity-sc-rs, the search defaults from bonsai-rs.

Usage

```
params_sc_bonsai(
  variance_rule = c("marginalise", "posterior_mean", "max_posterior", "fixed"),
  fixed_variance = NULL,
  variance_bins = 161L,
  variance_min = 0.001,
  variance_max = 50,
  min_signal_to_noise = 1,
  start = c("linkage", "greedy_merge"),
  spr_search = c("approximate", "exact"),
  nni_search = c("approximate", "exact"),
  seed = 42L,
  reroot = TRUE,
  layout = c("equal_angle", "equal_daylight", "dendrogram"),
  hyperbolic = FALSE
)
```

Arguments

`variance_rule`	String. How Sanity treats each gene's variance in log fold change. `"marginalise"` integrates over the variance grid, `"posterior_mean"` and `"max_posterior"` collapse it to one value first (cheaper, less accurate error bars), and `"fixed"` uses `fixed_variance` for every gene. One of `c("marginalise", "posterior_mean", "max_posterior", "fixed")`. Defaults to `"marginalise"`.
`fixed_variance`	Numeric or `NULL`. The variance for `variance_rule = "fixed"`. Ignored otherwise. Defaults to `NULL`.
`variance_bins`	Integer. Bins in the log-spaced variance grid. Defaults to `161L`.
`variance_min`	Numeric. Smallest variance on the grid. Defaults to `0.001`.
`variance_max`	Numeric. Largest variance on the grid. Defaults to `50.0`.

`min_signal_to_noise`
: Numeric. Genes whose signal-to-noise ratio falls below this are left out of the tree. Defaults to `1.0`.

`start`
: String. Initial topology. `"linkage"` is a Ward linkage over a neighbour graph: faster and a better tree on real data. `"greedy_merge"` is the paper's star merge, kept for comparisons with the published method. One of `c("linkage", "greedy_merge")`. Defaults to `"linkage"`.

`spr_search`
: String. Subtree pruning and regrafting. `"approximate"` stays within a few nats of `"exact"` on bonsai-rs's benchmarks at up to twice the speed. One of `c("approximate", "exact")`. Defaults to `"approximate"`.

`nni_search`
: String. Nearest-neighbour interchange. `"approximate"` found the same tree as `"exact"` on every bonsai-rs benchmark, and faster. One of `c("approximate", "exact")`. Defaults to `"approximate"`.

`seed`
: Integer. Seed for the linkage, SPR and NNI searches. Defaults to `42L`.

`reroot`
: Boolean. Reroot the finished tree for display. Changes the drawing, not the likelihood. Defaults to `TRUE`.

`layout`
: String. 2D layout. `"equal_daylight"` refines equal angle but falls back to it above 2,048 nodes, so it only matters for small trees. Can be changed later with `relayout_bonsai()`. One of `c("equal_angle", "equal_daylight", "dendrogram")`. Defaults to `"equal_angle"`.

`hyperbolic`
: Boolean. Project the layout onto the hyperbolic disk. Defaults to `FALSE`.

Value

A named list with the following elements:

- variance_rule - String. How Sanity treats each gene's variance in log fold change. `"marginalise"` integrates over the variance grid, `"posterior_mean"` and `"max_posterior"` collapse it to one value first (cheaper, less accurate error bars), and `"fixed"` uses `fixed_variance` for every gene. One of `c("marginalise", "posterior_mean", "max_posterior", "fixed")`. Defaults to `"marginalise"`.
- fixed_variance - Numeric or `NULL`. The variance for `variance_rule = "fixed"`. Ignored otherwise. Defaults to `NULL`.
- variance_bins - Integer. Bins in the log-spaced variance grid. Defaults to `161L`.
- variance_min - Numeric. Smallest variance on the grid. Defaults to `0.001`.
- variance_max - Numeric. Largest variance on the grid. Defaults to `50.0`.
- min_signal_to_noise - Numeric. Genes whose signal-to-noise ratio falls below this are left out of the tree. Defaults to `1.0`.
- start - String. Initial topology. `"linkage"` is a Ward linkage over a neighbour graph: faster and a better tree on real data. `"greedy_merge"` is the paper's star merge, kept for comparisons with the published method. One of `c("linkage", "greedy_merge")`. Defaults to `"linkage"`.
- spr_search - String. Subtree pruning and regrafting. `"approximate"` stays within a few nats of `"exact"` on bonsai-rs's benchmarks at up to twice the speed. One of `c("approximate", "exact")`. Defaults to `"approximate"`.

- nni_search - String. Nearest-neighbour interchange. `"approximate"` found the same tree as `"exact"` on every bonsai-rs benchmark, and faster. One of `c("approximate", "exact")`. Defaults to `"approximate"`.
- seed - Integer. Seed for the linkage, SPR and NNI searches. Defaults to `42L`.
- reroot - Boolean. Reroot the finished tree for display. Changes the drawing, not the likelihood. Defaults to `TRUE`.
- layout - String. 2D layout. `"equal_daylight"` refines equal angle but falls back to it above 2,048 nodes, so it only matters for small trees. Can be changed later with `relayout_bonsai()`. One of `c("equal_angle", "equal_daylight", "dendrogram")`. Defaults to `"equal_angle"`.
- hyperbolic - Boolean. Project the layout onto the hyperbolic disk. Defaults to `FALSE`.

References

de Groot, et al., Nat. Biotechnol., 2026; Breda, et al., Nat. Biotechnol., 2021.

`params_sc_branch_selection` *Wrapper function for the branch cell selection parameters*

Description

Parameters controlling which cells `run_gene_trends_sc()` assigns to each branch. The threshold on a fate's probability is an expanding quantile over the pseudotime-sorted cells, made monotone with a cumulative maximum, so a fate's bar can only rise as differentiation proceeds. The defaults are the reference ones.

Usage

```
params_sc_branch_selection(q = 0.01, eps = 0.01, resolution = 500L)
```

Arguments

`q`	Numeric. Upper-tail quantile of the fate probability used as the threshold. Defaults to `0.01`.
`eps`	Numeric. Slack subtracted from the threshold before the comparison. Defaults to `0.01`.
`resolution`	Integer. Number of pseudotime buckets, capped at the cell count. Defaults to `500L`.

Value

A named list with the following elements:

- q - Numeric. Upper-tail quantile of the fate probability used as the threshold. Defaults to `0.01`.
- eps - Numeric. Slack subtracted from the threshold before the comparison. Defaults to `0.01`.
- resolution - Integer. Number of pseudotime buckets, capped at the cell count. Defaults to `500L`.

References

Setty, et al., Nat. Biotechnol., 2019.

`params_sc_bt_metacells` *Wrapper function for parameters for bootstrapped meta cell generation*

Description

This function generates parameters for the bootstrapped meta cell generation based on hdWGCNA, see Morabito, et al., Cell Rep. Methods, 2023.

Usage

```
params_sc_bt_metacells(
  max_shared = 15L,
  target_no_metacells = 1000L,
  max_iter = 5000L,
  knn = list()
)
```

Arguments

`max_shared`	Integer. Maximum number of allowed shared neighbours for the meta cell to be considered. Defaults to `15L`.
`target_no_metacells`	Integer. Target number of meta-cells to generate. Defaults to `1000L`.
`max_iter`	Integer. Maximum number of iterations for the algorithm. Defaults to `5000L`.
`knn`	List. Optional overrides for kNN parameters. See `params_knn_defaults()` for available parameters: `k`, `knn_method`, `ann_dist`, `search_budget`, `n_trees`, `delta`, `diversify_prob`, `ef_budget`, `extract_knn`, `m`, `ef_construction`, `ef_search`, `n_list` and `n_probe`. See `params_knn_defaults()` for the available elements. Defaults to `list()`.

Value

A named list with the following elements:

- max_shared - Integer. Maximum number of allowed shared neighbours for the meta cell to be considered. Defaults to `15L`.
- target_no_metacells - Integer. Target number of meta-cells to generate. Defaults to `1000L`.
- max_iter - Integer. Maximum number of iterations for the algorithm. Defaults to `5000L`.
- The elements of `params_knn_defaults()`, overridden by `knn`, spliced in at this position.

`params_sc_cellphonedb` *Parameters for the CellPhoneDB analysis*

Description

Defaults match CellPhoneDB v5: 1000 permutations and a 10% expression threshold.

Usage

```
params_sc_cellphonedb(
  n_perm = 1000L,
  threshold = 0.1,
  seed = 42L,
  perm_batch = NULL
)
```

Arguments

`n_perm`	Integer. Number of cluster label permutations for the p-values. Defaults to `1000L`.
`threshold`	Numeric. Both partners need a fraction of expressing cells strictly above this value in their cluster. Defaults to `0.1`.
`seed`	Integer. Seed for the permutations. Defaults to `42L`.
`perm_batch`	Integer or `NULL`. Permutations that share one pass over the gene data. `NULL` uses the Rust default (16). Defaults to `NULL`.

Value

A named list with the following elements:

- n_perm - Integer. Number of cluster label permutations for the p-values. Defaults to `1000L`.

- threshold - Numeric. Both partners need a fraction of expressing cells strictly above this value in their cluster. Defaults to `0.1`.
- seed - Integer. Seed for the permutations. Defaults to `42L`.
- perm_batch - Integer or `NULL`. Permutations that share one pass over the gene data. `NULL` uses the Rust default (16). Defaults to `NULL`.

`params_sc_cellsweep` *Default parameters for CellSweep denoising*

Description

Mirrors the CellSweep reference implementation's defaults. The pseudocounts (`celltype_lambda`, `ambient_lambda`, `bulk_lambda`) are given on the scale you see here and divided by the gene count internally. Two of these are worth knowing about before you touch anything else. `freeze_ambient_profile = TRUE` keeps the ambient profile at its empty-droplet estimate, which is the recommended path and the only one where `alpha_cap`, the repulsion terms and cell-type reassignment are live. And `freeze_empties` only accepts `TRUE`: the reference gives empty droplets a cell-type component they have no label for, which indexes past the end of the profile matrix and wraps onto the last cell type.

Usage

```
params_sc_cellsweep(
  freeze_empties = TRUE,
  freeze_ambient_profile = TRUE,
  init_alpha = 0.9,
  init_beta = 0.1,
  alpha_cap = 0.9,
  repulsion_strength = 1e-04,
  max_frac_gene_repulsion = 0.2,
  celltype_lambda = 50,
  ambient_lambda = 50,
  bulk_lambda = 10,
  eps = 1e-12,
  log_eps = as.numeric("1e-300"),
  max_iter = 2000L,
  del0_ll_tol = 0.001,
  min_ll_tol = 1e-06,
  tol_p = 1e-04,
  tol_f = 1e-04,
  norm_from_rounded = FALSE,
  seed = 42L
)
```

Arguments

`freeze_empties`
: Boolean. Keep the contamination fraction of empty droplets pinned at 1. Only `TRUE` is supported, see the description. Defaults to `TRUE`.

`freeze_ambient_profile`
: Boolean. Keep the ambient profile at its empty-droplet estimate rather than re-estimating it as a mixture over the cell-type profiles. Defaults to `TRUE`.

`init_alpha`
: Numeric. Starting ambient fraction for every real barcode. With `freeze_ambient_profile = TRUE` the final result barely depends on it, so it sits at `alpha_cap`. Defaults to `0.9`.

`init_beta`
: Numeric. Starting bulk contamination fraction. Set below `init_alpha` on purpose: bulk and ambient are not fully separable, so this biases unassignable contamination towards ambient. Defaults to `0.1`.

`alpha_cap`
: Numeric. Ceiling on the per-cell ambient fraction before the log-likelihood converges. Barcodes wanting to exceed it are excluded from the cell-type profile update and allowed to switch cell type. Defaults to `0.9`.

`repulsion_strength`
: Numeric. Strength of the repulsion pushing cell-type profiles away from the ambient profile. Scales with cluster mass, so it is inert on small data and only bites at realistic cell counts. Defaults to `1e-04`.

`max_frac_gene_repulsion`
: Numeric. Ceiling on the fraction of any single profile entry that repulsion may remove. Defaults to `0.2`.

`celltype_lambda`
: Numeric. Pseudocount smoothing the cell-type profile update. Higher values give smoother profiles. Defaults to `50.0`.

`ambient_lambda`
: Numeric. Pseudocount smoothing the ambient profile estimate. Defaults to `50.0`.

`bulk_lambda`
: Numeric. Pseudocount smoothing the bulk profile estimate. Defaults to `10.0`.

`eps`
: Numeric. Floor on denominators. Defaults to `1e-12`.

`log_eps`
: Numeric. Floor on the argument of `log`. Defaults to `1e-300`.

`max_iter`
: Integer. Hard cap on EM iterations. Defaults to `2000L`.

`del0_ll_tol`
: Numeric. Log-likelihood change, as a fraction of the first EM step's change, below which stage one ends and parameter convergence starts being checked. Defaults to `0.001`.

`min_ll_tol`
: Numeric. Floor on the adaptive tolerance, relative to the current log-likelihood. Stops `del0_ll_tol` chasing floating point noise. Defaults to `1e-06`.

`tol_p`
: Numeric. Convergence threshold on the maximum row-wise L1 change in the cell-type profiles. Defaults to `1e-04`.

`tol_f`
: Numeric. Convergence threshold on the change in the total contamination fraction. Defaults to `1e-04`.

`norm_from_rounded`
: Boolean. Derive the normalised layer from the integerised counts rather than the denoised floats. Consistent across the two layers at the cost of the sub-integer signal, which is where CellSweep is most informative. Defaults to `FALSE`.

`seed`
: Integer. Seed for the stochastic rounding of the denoised counts. Defaults to `42L`.

Value

A named list with the following elements:

- freeze_empties - Boolean. Keep the contamination fraction of empty droplets pinned at 1. Only `TRUE` is supported, see the description. Defaults to `TRUE`.
- freeze_ambient_profile - Boolean. Keep the ambient profile at its empty-droplet estimate rather than re-estimating it as a mixture over the cell-type profiles. Defaults to `TRUE`.
- init_alpha - Numeric. Starting ambient fraction for every real barcode. With `freeze_ambient_profile = TRUE` the final result barely depends on it, so it sits at `alpha_cap`. Defaults to `0.9`.
- init_beta - Numeric. Starting bulk contamination fraction. Set below `init_alpha` on purpose: bulk and ambient are not fully separable, so this biases unassignable contamination towards ambient. Defaults to `0.1`.
- alpha_cap - Numeric. Ceiling on the per-cell ambient fraction before the log-likelihood converges. Barcodes wanting to exceed it are excluded from the cell-type profile update and allowed to switch cell type. Defaults to `0.9`.
- repulsion_strength - Numeric. Strength of the repulsion pushing cell-type profiles away from the ambient profile. Scales with cluster mass, so it is inert on small data and only bites at realistic cell counts. Defaults to `1e-04`.
- max_frac_gene_repulsion - Numeric. Ceiling on the fraction of any single profile entry that repulsion may remove. Defaults to `0.2`.
- celltype_lambda - Numeric. Pseudocount smoothing the cell-type profile update. Higher values give smoother profiles. Defaults to `50.0`.
- ambient_lambda - Numeric. Pseudocount smoothing the ambient profile estimate. Defaults to `50.0`.
- bulk_lambda - Numeric. Pseudocount smoothing the bulk profile estimate. Defaults to `10.0`.
- eps - Numeric. Floor on denominators. Defaults to `1e-12`.
- log_eps - Numeric. Floor on the argument of `log`. Defaults to `1e-300`.
- max_iter - Integer. Hard cap on EM iterations. Defaults to `2000L`.
- del0_ll_tol - Numeric. Log-likelihood change, as a fraction of the first EM step's change, below which stage one ends and parameter convergence starts being checked. Defaults to `0.001`.
- min_ll_tol - Numeric. Floor on the adaptive tolerance, relative to the current log-likelihood. Stops `del0_ll_tol` chasing floating point noise. Defaults to `1e-06`.

- tol_p - Numeric. Convergence threshold on the maximum row-wise L1 change in the cell-type profiles. Defaults to `1e-04`.
- tol_f - Numeric. Convergence threshold on the change in the total contamination fraction. Defaults to `1e-04`.
- norm_from_rounded - Boolean. Derive the normalised layer from the integerised counts rather than the denoised floats. Consistent across the two layers at the cost of the sub-integer signal, which is where CellSweep is most informative. Defaults to `FALSE`.
- seed - Integer. Seed for the stochastic rounding of the denoised counts. Defaults to `42L`.

`params_sc_dsb` *Default parameters for DSB ADT normalisation*

Description

Default parameters for DSB ADT normalisation

Usage

```
params_sc_dsb(
  denoise_counts = TRUE,
  use_isotype_controls = TRUE,
  pseudocount = 10,
  quantile_low = NULL,
  quantile_high = NULL
)
```

Arguments

`denoise_counts`	Boolean. Run Step II (cell-to-cell technical noise removal). Defaults to `TRUE`.
`use_isotype_controls`	Boolean. Include isotype controls in the noise matrix in Step II. Requires `isotype_indices` to be passed at call time. Defaults to `TRUE`.
`pseudocount`	Numeric. Pseudocount added before the log transform. The DSB paper recommends `10` with empty droplets and `1` without. Defaults to `10.0`.
`quantile_low`	Numeric or `NULL`. Optional numeric in `[0, 1)`. Lower quantile for per-protein output clipping. If `NULL` (and `quantile_high` is also `NULL`), no clipping is applied. Defaults to `NULL`.
`quantile_high`	Numeric or `NULL`. Optional numeric in `(0, 1]`. Upper quantile for per-protein output clipping. If `NULL` (and `quantile_low` is also `NULL`), no clipping is applied. Defaults to `NULL`.

Value

A named list with the following elements:

- denoise_counts - Boolean. Run Step II (cell-to-cell technical noise removal). Defaults to `TRUE`.
- use_isotype_controls - Boolean. Include isotype controls in the noise matrix in Step II. Requires `isotype_indices` to be passed at call time. Defaults to `TRUE`.
- pseudocount - Numeric. Pseudocount added before the log transform. The DSB paper recommends `10` with empty droplets and `1` without. Defaults to `10.0`.
- quantile_low - Numeric or `NULL`. Optional numeric in `[0, 1)`. Lower quantile for per-protein output clipping. If `NULL` (and `quantile_high` is also `NULL`), no clipping is applied. Defaults to `NULL`.
- quantile_high - Numeric or `NULL`. Optional numeric in `(0, 1]`. Upper quantile for per-protein output clipping. If `NULL` (and `quantile_low` is also `NULL`), no clipping is applied. Defaults to `NULL`.

[0, 1). Lower quantile for per-protein output clipping. If NULL(andquantile_highis alsoNULL), n
1]: R:0,%201)%60.%20Lower%0A%20quantile%20for%20per-protein%20output%20clipping.%20If%20%60NULL%
%20Numeric%20or%20%60NULL%60.%20Optional%20numeric%20in%20%60(0,%201

`params_sc_empty_droplets`

Parameters for identifying empty droplets

Description

CellSweep trains its ambient profile on the empty droplets, so it needs them called before it runs. Four ways to do that, in descending order of how much you should trust them. `"supplied"` takes an existing logical column from the obs table. This is the common case: if you have CellRanger's filtered barcode list you already know which barcodes are empty, and nothing here will beat that. `"umi_cutoff"` calls everything below an absolute library size empty. `"expected_cells"` turns a cell count into that cutoff via the sorted library sizes. `"knee"` finds the cutoff from the curvature of the rank / log-count curve; it is experimental in the reference too, and smoothing puts the curvature minimum a few ranks ahead of the actual cliff, so it sweeps up the real barcodes nearest the transition.

Usage

```
params_sc_empty_droplets(
  method = c("supplied", "umi_cutoff", "expected_cells", "knee"),
  is_empty_column = NULL,
  umi_cutoff = NULL,
  expected_cells = NULL
)
```

Arguments

`method`	String. How the empty droplets are called, see the description. One of `c("supplied", "umi_cutoff", "expected_cells", "knee")`. Defaults to `"supplied"`.
`is_empty_column`	String or `NULL`. Name of the logical obs column holding the mask. Required for `method = "supplied"`, ignored otherwise. Defaults to `NULL`.
`umi_cutoff`	Integer or `NULL`. Barcodes with a library size strictly below this are empty. Required for `method = "umi_cutoff"`, ignored otherwise. Defaults to `NULL`.
`expected_cells`	Integer or `NULL`. Number of real cells expected in the run. Required for `method = "expected_cells"`, ignored otherwise. Defaults to `NULL`.

Value

A named list with the following elements:

- method - String. How the empty droplets are called, see the description. One of `c("supplied", "umi_cutoff", "expected_cells", "knee")`. Defaults to `"supplied"`.
- is_empty_column - String or `NULL`. Name of the logical obs column holding the mask. Required for `method = "supplied"`, ignored otherwise. Defaults to `NULL`.
- umi_cutoff - Integer or `NULL`. Barcodes with a library size strictly below this are empty. Required for `method = "umi_cutoff"`, ignored otherwise. Defaults to `NULL`.
- expected_cells - Integer or `NULL`. Number of real cells expected in the run. Required for `method = "expected_cells"`, ignored otherwise. Defaults to `NULL`.

---

params_sc_fast_cluster

Fast single cell clustering parameters

Description

Fast single cell clustering parameters

Usage

```
params_sc_fast_cluster(
  kmeans_iters = 100L,
  batch_size = 4096L,
  drift_threshold = 1e-04,
  lr_alpha = 1,
  full_snn = FALSE,
  pruning = NULL,
  snn_similarity = c("jaccard", "rank"),
  louvain_iters = 10L,
```

```
  cell_mediated = FALSE,
  n_anchors = 5L,
  knn = list(k = 5L)
)
```

Arguments

kmeans_iters	Integer. Number of iterations for k-means clustering. Defaults to 100L.
batch_size	Integer. Batch size for mini batch k-means clustering. Defaults to 4096L.
drift_threshold	Numeric. The drift for the mini batch k-means clustering. If the centroid drift is below this, the mini batch k-means terminates. Defaults to 1e-04.
lr_alpha	Numeric. Learning rate alpha parameter for mini batch k-means. Defaults to 1.0.
full_snn	Boolean. Shall the full shared nearest neighbour graph be generated that generates edges between all cells instead of between only neighbours. Defaults to FALSE.
pruning	Numeric or NULL. Weights below this threshold will be set to 0 in the generation of the sNN graph. A node that loses every edge keeps its strongest kNN edge. If not provided, defaults to 1 / ceil(k * 0.8). Defaults to NULL.
snn_similarity	String. The Jaccard similarity calculates the Jaccard between the neighbours, whereas the rank method calculates edge weights based on the ranking of shared neighbours. For the rank method, the weight is determined by finding the shared neighbour with the lowest combined rank across both cells, where lower-ranked (closer) shared neighbours result in higher edge weights Both methods produce weights normalised to the range [0, 1]. One of c("jaccard", "rank"). Defaults to "jaccard".
louvain_iters	Integer. Number of iterations for Louvain clustering. Defaults to 10L.
cell_mediated	Boolean. Shall the cell-mediated centroid graph from Secuer be built instead of the kNN/sNN graph on the centroids. Centroids are then linked by the cells that sit near both, so two centroids that are close in space but have no cells between them stay apart. If TRUE, snn, full_snn, pruning and snn_similarity are ignored. See Wei, et al., PLoS Comput Biol, 2022. Defaults to FALSE.
n_anchors	Integer. Number of nearest centroids each cell links to in the cell-mediated graph. Only used if cell_mediated = TRUE. Defaults to 5L.
knn	List. Optional overrides for kNN parameters. See params_knn_defaults() for available parameters: k, knn_method, ann_dist, search_budget, n_trees, delta, diversify_prob, ef_budget, extract_knn, m, ef_construction, ef_search, n_list and n_probe. Sets the default k = 5L. See params_knn_defaults() for the available elements. Defaults to list(k = 5L).

Value

A named list with the following elements:

- kmeans_iters - Integer. Number of iterations for k-means clustering. Defaults to `100L`.
- batch_size - Integer. Batch size for mini batch k-means clustering. Defaults to `4096L`.
- drift_threshold - Numeric. The drift for the mini batch k-means clustering. If the centroid drift is below this, the mini batch k-means terminates. Defaults to `1e-04`.
- lr_alpha - Numeric. Learning rate alpha parameter for mini batch k-means. Defaults to `1.0`.
- louvain_iters - Integer. Number of iterations for Louvain clustering. Defaults to `10L`.
- full_snn - Boolean. Shall the full shared nearest neighbour graph be generated that generates edges between all cells instead of between only neighbours. Defaults to `FALSE`.
- pruning - Numeric or `NULL`. Weights below this threshold will be set to 0 in the generation of the sNN graph. A node that loses every edge keeps its strongest kNN edge. If not provided, defaults to `1 / ceil(k * 0.8)`. Defaults to `NULL`.
- snn_similarity - String. The Jaccard similarity calculates the Jaccard between the neighbours, whereas the rank method calculates edge weights based on the ranking of shared neighbours. For the rank method, the weight is determined by finding the shared neighbour with the lowest combined rank across both cells, where lower-ranked (closer) shared neighbours result in higher edge weights Both methods produce weights normalised to the range `[0, 1]`. One of `c("jaccard", "rank")`. Defaults to `"jaccard"`.
- cell_mediated - Boolean. Shall the cell-mediated centroid graph from Secuer be built instead of the kNN/sNN graph on the centroids. Centroids are then linked by the cells that sit near both, so two centroids that are close in space but have no cells between them stay apart. If `TRUE`, `snn`, `full_snn`, `pruning` and `snn_similarity` are ignored. See Wei, et al., PLoS Comput Biol, 2022. Defaults to `FALSE`.
- n_anchors - Integer. Number of nearest centroids each cell links to in the cell-mediated graph. Only used if `cell_mediated = TRUE`. Defaults to `5L`.
- The elements of `params_knn_defaults()`, overridden by `knn`, spliced in at this position.

`params_sc_fastmnn` *Wrapper function for the fastMNN parameters*

Description

Wrapper function for the fastMNN parameters

Usage

```
params_sc_fastmnn(
  ndist = 3,
  cos_norm = TRUE,
  no_pcs = 30L,
  sparse_svd = TRUE,
  knn = list(k = 20L),
  pca = params_sc_pca()
)
```

Arguments

ndist	Numeric. Number of median distances for the tricube kernel bandwidth. Defaults to 3.0.
cos_norm	Boolean. Apply cosine normalisation before computing distances. Defaults to TRUE.
no_pcs	Integer. Number of PCs to use for MNN calculations. Defaults to 30L.
sparse_svd	Boolean. Shall the sparse SVD be used. Defaults to TRUE.
knn	List. Optional overrides for kNN parameters. See params_knn_defaults() for available parameters: k, knn_method, ann_dist, search_budget, n_trees, delta, diversify_prob, ef_budget, extract_knn, m, ef_construction, ef_search, n_list and n_probe. See params_knn_defaults() for the available elements. Defaults to list(k = 20L).
pca	List. Parameters to feed through to the optional recalculation of the PCA, see params_sc_pca(). See params_sc_pca() for the available elements. Defaults to params_sc_pca().

Value

A named list with the following elements:

- ndist - Numeric. Number of median distances for the tricube kernel bandwidth. Defaults to 3.0.
- cos_norm - Boolean. Apply cosine normalisation before computing distances. Defaults to TRUE.
- no_pcs - Integer. Number of PCs to use for MNN calculations. Defaults to 30L.
- sparse_svd - Boolean. Shall the sparse SVD be used. Defaults to TRUE.
- The elements of params_knn_defaults(), overridden by knn, spliced in at this position.
- The elements of params_sc_pca(), overridden by pca, spliced in at this position.

---

params_sc_gene_trends

Wrapper function for gene trend parameters

Description

Parameters controlling the landmark Gaussian process that run_gene_trends_sc() fits per branch. The kernel is a Matern-5/2 one and the prediction grid doubles as the landmark set. The defaults come from the reference and are prior-dominated. Palantir's pseudotime is min-max scaled to [0, 1], so a length_scale of 1.0 spans the entire domain and a sigma of 1.0 sits at roughly the signal scale of log-normalised expression. The posterior will flatten genuine transient structure and resolve almost any gene into a smooth monotone or single-peaked curve. That is a presentation choice, not inference. Shorten length_scale before believing a bump.

Usage

```
params_sc_gene_trends(
  resolution = 500L,
  weighting = c("hard_mask", "fate_probability"),
  length_scale = 1,
  sigma = 1,
  jitter = 1e-06,
  max_jitter_retries = 3L,
  chunk_size = 2048L
)
```

Arguments

`resolution`	Integer. Grid points per branch. Kept at the default even when a branch holds fewer cells, as the reference does. Defaults to `500L`.
`weighting`	String. With `"hard_mask"` every selected cell enters its branch's fit with equal weight, which is what the reference does. With `"fate_probability"` every cell enters every fit weighted by its fate probability, which is more defensible: a cell at 0.6 is not a member. One of `c("hard_mask", "fate_probability")`. Defaults to `"hard_mask"`.
`length_scale`	Numeric. Matern-5/2 length scale. Defaults to `1.0`.
`sigma`	Numeric. Noise standard deviation. Defaults to `1.0`.
`jitter`	Numeric. Added to the landmark covariance diagonal before the Cholesky. Defaults to `1e-06`.
`max_jitter_retries`	Integer. Times the jitter is raised and the Cholesky retried before giving up. Defaults to `3L`.
`chunk_size`	Integer. Training points held at once when accumulating the cross-covariance. Defaults to `2048L`.

Value

A named list with the following elements:

- resolution - Integer. Grid points per branch. Kept at the default even when a branch holds fewer cells, as the reference does. Defaults to `500L`.
- weighting - String. With `"hard_mask"` every selected cell enters its branch's fit with equal weight, which is what the reference does. With `"fate_probability"` every cell enters every fit weighted by its fate probability, which is more defensible: a cell at 0.6 is not a member. One of `c("hard_mask", "fate_probability")`. Defaults to `"hard_mask"`.
- length_scale - Numeric. Matern-5/2 length scale. Defaults to `1.0`.
- sigma - Numeric. Noise standard deviation. Defaults to `1.0`.
- jitter - Numeric. Added to the landmark covariance diagonal before the Cholesky. Defaults to `1e-06`.

- max_jitter_retries - Integer. Times the jitter is raised and the Cholesky retried before giving up. Defaults to `3L`.
- chunk_size - Integer. Training points held at once when accumulating the cross-covariance. Defaults to `2048L`.

References

Setty, et al., Nat. Biotechnol., 2019.

`params_sc_harmony` *Default parameters for Harmony batch correction*

Description

Default parameters for Harmony batch correction

Usage

```
params_sc_harmony(
  k = NULL,
  sigma = 0.1,
  theta = 2,
  lambda = 1,
  block_size = 0.2,
  max_iter_kmeans = 20L,
  max_iter_harmony = 10L,
  epsilon_kmeans = 0.001,
  epsilon_harmony = 0.01,
  window_size = 3L,
  kmeans = list()
)
```

Arguments

`k`	Integer or `NULL`. Number of clusters for k-means clustering. If not provided, it will be automatically determined as `min(round(N / 30), 100)`. Defaults to `NULL`.
`sigma`	Numeric vector. Per-cluster diversity weights. Either a single value (broadcast to all clusters) or a vector of length k. Defaults to `0.1`.
`theta`	Numeric vector. Per-variable diversity penalties. Either a single value (broadcast to all variables) or a vector of length equal to the number of batch variables. Defaults to `2.0`.
`lambda`	Numeric vector. Ridge regression penalty for the linear model. Typically a single value that is broadcast to every batch column of the design matrix; the intercept is not penalised. Defaults to `1.0`.

`block_size` Numeric. Fraction of cells to update per block during optimisation (0.0-1.0). Lower values reduce memory usage but increase computation time. Defaults to `0.2`.

`max_iter_kmeans` Integer. Maximum number of k-means iterations per Harmony round. Defaults to `20L`.

`max_iter_harmony` Integer. Maximum number of Harmony outer iterations. Defaults to `10L`.

`epsilon_kmeans` Numeric. Convergence threshold for k-means clustering. Stops when the relative change in cluster assignments falls below this value. Defaults to `0.001`.

`epsilon_harmony` Numeric. Convergence threshold for Harmony. Stops when the relative change in the objective function falls below this value. Defaults to `0.01`.

`window_size` Integer. Number of previous iterations to consider when checking convergence. Defaults to `3L`.

`kmeans` List. Optional overrides for the k-means clustering algorithm Possible parameters are `"k_means_iter"` (10 here, as in R harmony), `"k_means_init"`, `"gemm"` and `"hamerly"`, see `params_kmeans_defaults()`. See `params_kmeans_defaults()` for the available elements. Defaults to `list()`.

Value

A named list with the following elements:

- k - Integer or `NULL`. Number of clusters for k-means clustering. If not provided, it will be automatically determined as `min(round(N / 30), 100)`. Defaults to `NULL`.
- sigma - Numeric vector. Per-cluster diversity weights. Either a single value (broadcast to all clusters) or a vector of length k. Defaults to `0.1`.
- theta - Numeric vector. Per-variable diversity penalties. Either a single value (broadcast to all variables) or a vector of length equal to the number of batch variables. Defaults to `2.0`.
- lambda - Numeric vector. Ridge regression penalty for the linear model. Typically a single value that is broadcast to every batch column of the design matrix; the intercept is not penalised. Defaults to `1.0`.
- block_size - Numeric. Fraction of cells to update per block during optimisation (0.0-1.0). Lower values reduce memory usage but increase computation time. Defaults to `0.2`.
- max_iter_kmeans - Integer. Maximum number of k-means iterations per Harmony round. Defaults to `20L`.
- max_iter_harmony - Integer. Maximum number of Harmony outer iterations. Defaults to `10L`.
- epsilon_kmeans - Numeric. Convergence threshold for k-means clustering. Stops when the relative change in cluster assignments falls below this value. Defaults to `0.001`.

- epsilon_harmony - Numeric. Convergence threshold for Harmony. Stops when the relative change in the objective function falls below this value. Defaults to `0.01`.
- window_size - Integer. Number of previous iterations to consider when checking convergence. Defaults to `3L`.
- The elements of `params_kmeans_defaults()`, overridden by `kmeans`, spliced in at this position.

`params_sc_harmony_v2` *Default parameters for Harmony v2 batch correction*

Description

Default parameters for Harmony v2 batch correction

Usage

```
params_sc_harmony_v2(
  k = NULL,
  sigma = 0.1,
  theta = 2,
  lambda = 1,
  block_size = 0.2,
  max_iter_kmeans = 4L,
  max_iter_harmony = 10L,
  epsilon_kmeans = 0.001,
  epsilon_harmony = 0.01,
  window_size = 3L,
  alpha = 0.2,
  tau = 0,
  batch_proportion_cutoff = 1e-05,
  use_dynamic_lambda = TRUE,
  kmeans = list()
)
```

Arguments

`k`	Integer or `NULL`. Number of clusters for k-means clustering. If not provided, it will be automatically determined as `min(round(N / 30), 100)`. Defaults to `NULL`.
`sigma`	Numeric vector. Per-cluster diversity weights. Either a single value (broadcast to all clusters) or a vector of length k. Defaults to `0.1`.
`theta`	Numeric vector. Per-variable diversity penalties. Either a single value (broadcast to all variables) or a vector of length equal to the number of batch variables. Defaults to `2.0`.

`lambda` Numeric vector. Ridge regression penalty for the linear model. Typically a single value that is broadcast to every batch column of the design matrix; the intercept is not penalised. Ignored when `use_dynamic_lambda = TRUE`. Defaults to `1.0`.

`block_size` Numeric. Fraction of cells to update per block during optimisation (0.0-1.0). Lower values reduce memory usage but increase computation time. Defaults to `0.2`.

`max_iter_kmeans` Integer. Maximum number of k-means iterations per Harmony round. Defaults to `4L`.

`max_iter_harmony` Integer. Maximum number of Harmony outer iterations. Defaults to `10L`.

`epsilon_kmeans` Numeric. Convergence threshold for k-means clustering. Stops when the relative change in cluster assignments falls below this value. Defaults to `0.001`.

`epsilon_harmony` Numeric. Convergence threshold for Harmony. Stops when the relative change in the objective function falls below this value. Defaults to `0.01`.

`window_size` Integer. Number of previous iterations to consider when checking convergence. Defaults to `3L`.

`alpha` Numeric. Scaling factor for dynamic lambda estimation. Must be in (0, 1). Only relevant when `use_dynamic_lambda = TRUE`. Defaults to `0.2`.

`tau` Numeric. Scaling factor for theta based on batch size. A value of 0 disables batch-size scaling of theta. Defaults to `0.0`.

`batch_proportion_cutoff` Numeric. Cutoff for pruning batches with small proportions during ridge regression. Defaults to `1e-05`.

`use_dynamic_lambda` Boolean. If `TRUE`, lambda is estimated per cluster as `alpha` times the expected counts, as in R harmony v2, instead of using the fixed `lambda` value. Defaults to `TRUE`.

`kmeans` List. Optional overrides for the k-means clustering algorithm Possible parameters are `"k_means_iter"` (10 here, as in R harmony), `"k_means_init"`, `"gemm"` and `"hamerly"`, see `params_kmeans_defaults()`. See `params_kmeans_defaults()` for the available elements. Defaults to `list()`.

Value

A named list with the following elements:

- k - Integer or `NULL`. Number of clusters for k-means clustering. If not provided, it will be automatically determined as `min(round(N / 30), 100)`. Defaults to `NULL`.
- sigma - Numeric vector. Per-cluster diversity weights. Either a single value (broadcast to all clusters) or a vector of length k. Defaults to `0.1`.
- theta - Numeric vector. Per-variable diversity penalties. Either a single value (broadcast to all variables) or a vector of length equal to the number of batch variables. Defaults to `2.0`.

- lambda - Numeric vector. Ridge regression penalty for the linear model. Typically a single value that is broadcast to every batch column of the design matrix; the intercept is not penalised. Ignored when `use_dynamic_lambda = TRUE`. Defaults to `1.0`.
- block_size - Numeric. Fraction of cells to update per block during optimisation (0.0-1.0). Lower values reduce memory usage but increase computation time. Defaults to `0.2`.
- max_iter_kmeans - Integer. Maximum number of k-means iterations per Harmony round. Defaults to `4L`.
- max_iter_harmony - Integer. Maximum number of Harmony outer iterations. Defaults to `10L`.
- epsilon_kmeans - Numeric. Convergence threshold for k-means clustering. Stops when the relative change in cluster assignments falls below this value. Defaults to `0.001`.
- epsilon_harmony - Numeric. Convergence threshold for Harmony. Stops when the relative change in the objective function falls below this value. Defaults to `0.01`.
- window_size - Integer. Number of previous iterations to consider when checking convergence. Defaults to `3L`.
- alpha - Numeric. Scaling factor for dynamic lambda estimation. Must be in (0, 1). Only relevant when `use_dynamic_lambda = TRUE`. Defaults to `0.2`.
- tau - Numeric. Scaling factor for theta based on batch size. A value of 0 disables batch-size scaling of theta. Defaults to `0.0`.
- batch_proportion_cutoff - Numeric. Cutoff for pruning batches with small proportions during ridge regression. Defaults to `1e-05`.
- use_dynamic_lambda - Boolean. If `TRUE`, lambda is estimated per cluster as `alpha` times the expected counts, as in R harmony v2, instead of using the fixed `lambda` value. Defaults to `TRUE`.
- The elements of `params_kmeans_defaults()`, overridden by `kmeans`, spliced in at this position.

`params_sc_hotspot` *Wrapper function for parameters for HotSpot*

Description

`weighted_graph` controls how the kNN distances become edge weights. The default of `FALSE` follows the reference implementation: the distances only decide who is a neighbour and every retained edge weighs one. Set it to `TRUE` for the Gaussian kernel, whose width is the `ceil(k / neighborhood_factor)`-th neighbour distance.

Usage

```
params_sc_hotspot(
  model = c("danb", "normal", "bernoulli"),
  normalise = TRUE,
  weighted_graph = FALSE,
```

```
  neighborhood_factor = 3,
  knn = list()
)
```

Arguments

`model`	String. Model to use for modelling the GEX. One of `c("danb", "normal", "bernoulli")`. Defaults to `"danb"`.
`normalise`	Boolean. Shall the data be normalised. Defaults to `TRUE`.
`weighted_graph`	Boolean. Shall the Gaussian kernel be applied to the neighbour distances. Defaults to `FALSE`.
`neighborhood_factor`	Numeric. Kernel width is the `ceil(k / neighborhood_factor)`-th neighbour distance. Only read when `weighted_graph = TRUE`. Defaults to `3.0`.
`knn`	List. Optional overrides for kNN parameters. See `params_knn_defaults()` for available parameters: `k`, `knn_method`, `ann_dist`, `search_budget`, `n_trees`, `delta`, `diversify_prob`, `ef_budget`, `extract_knn`, `m`, `ef_construction`, `ef_search`, `n_list` and `n_probe`. See `params_knn_defaults()` for the available elements. Defaults to `list()`.

Value

A named list with the following elements:

- model - String. Model to use for modelling the GEX. One of `c("danb", "normal", "bernoulli")`. Defaults to `"danb"`.
- normalise - Boolean. Shall the data be normalised. Defaults to `TRUE`.
- weighted_graph - Boolean. Shall the Gaussian kernel be applied to the neighbour distances. Defaults to `FALSE`.
- neighborhood_factor - Numeric. Kernel width is the `ceil(k / neighborhood_factor)`-th neighbour distance. Only read when `weighted_graph = TRUE`. Defaults to `3.0`.
- The elements of `params_knn_defaults()`, overridden by `knn`, spliced in at this position.

References

DeTomaso and Yosef, Cell Systems, 2021

params_sc_hvg *Wrapper function for HVG detection parameters.*

Description

Wrapper function for HVG detection parameters.

Usage

```
params_sc_hvg(
  method = c("vst", "meanvarbin", "dispersion", "scran", "residual"),
  loess_span = 0.3,
  num_bin = 20L,
  bin_method = c("equal_width", "equal_freq"),
  scran = list()
)
```

Arguments

`method`	String. `"scran"` fits a weighted lowess trend to the variance of the log-expression against its mean and ranks genes by the residual, as in scran's `modelGeneVar()`. `"residual"` ranks genes by the residual variance of a model fitted with `fit_residuals_sc()`, and needs that fit on the object first. It also treats `hvg_no` as a per-group count and returns the union across groups, so a grouped fit can select more than `hvg_no` genes. One of `c("vst", "meanvarbin", "dispersion", "scran", "residual")`. Defaults to `"vst"`.
`loess_span`	Numeric. The span parameter for the loess function that is used to standardise the variance for `method = "vst"`, and the lowess span of the trend for `method = "scran"`. Defaults to `0.3`.
`num_bin`	Integer. Not yet implemented. Defaults to `20L`.
`bin_method`	String. The binning method. One of `c("equal_width", "equal_freq")`. Defaults to `"equal_width"`.
`scran`	List. Optional overrides for the `method = "scran"` trend. See `params_hvg_scran_defaults()` for available parameters: `mean_filter`, `min_mean`, `transform`, `use_min_width`, `min_width` and `min_window_count`. See `params_hvg_scran_defaults()` for the available elements. Defaults to `list()`.

Value

A named list with the following elements:

- method - String. `"scran"` fits a weighted lowess trend to the variance of the log-expression against its mean and ranks genes by the residual, as in scran's `modelGeneVar()`. `"residual"` ranks genes by the residual variance of a model fitted with `fit_residuals_sc()`, and needs that fit on the object first. It also treats `hvg_no` as a per-group count and

returns the union across groups, so a grouped fit can select more than `hvg_no` genes. One of `c("vst", "meanvarbin", "dispersion", "scran", "residual")`. Defaults to `"vst"`.

- loess_span - Numeric. The span parameter for the loess function that is used to standardise the variance for `method = "vst"`, and the lowess span of the trend for `method = "scran"`. Defaults to `0.3`.
- num_bin - Integer. Not yet implemented. Defaults to `20L`.
- bin_method - String. The binning method. One of `c("equal_width", "equal_freq")`. Defaults to `"equal_width"`.
- The elements of `params_hvg_scran_defaults()`, overridden by `scran`, spliced in at this position.

`params_sc_knn` *Parameters for single cell kNN searches*

Description

Parameters for single cell kNN searches

Usage

```
params_sc_knn(
  k = 15L,
  knn_method = c("kmknn", "hnsw", "annoy", "nndescent", "ivf", "exhaustive"),
  ann_dist = c("euclidean", "cosine"),
  n_trees = 50L,
  search_budget = NULL,
  delta = 0.001,
  diversify_prob = 0,
  ef_budget = NULL,
  extract_knn = FALSE,
  m = 16L,
  ef_construction = 200L,
  ef_search = 100L,
  n_list = NULL,
  n_probe = NULL
)
```

Arguments

`k`	Integer. Number of neighbours. Defaults to `15L`.
`knn_method`	String. Which method to use for the approximate nearest neighbour search. One of `c("kmknn", "hnsw", "annoy", "nndescent", "ivf", "exhaustive")`. Defaults to `"kmknn"`.
`ann_dist`	String. Distance metric to use. One of `c("euclidean", "cosine")`. Defaults to `"euclidean"`.

`n_trees` Integer. Annoy param: number of trees. Defaults to `50L`.

`search_budget` Integer or `NULL`. Annoy param: optional search budget per tree. If `NULL`, defaults to `n_trees * k * 20L` internally. Defaults to `NULL`.

`delta` Numeric. NNDescent param: early termination criterion. Defaults to `0.001`.

`diversify_prob` Numeric. NNDescent param: diversification probability applied at the end of index construction. Defaults to `0.0`.

`ef_budget` Integer or `NULL`. NNDescent param: optional query budget. Higher values improve recall at the cost of speed. Defaults to `NULL`.

`extract_knn` Boolean. NNDescent param: hand back the graph the descent already built instead of beam searching it. Skips the query pass entirely, so it is much faster, at the cost of some recall. Rows the descent never filled come back padded with duplicate edges. Ignored by every other method. Defaults to `FALSE`.

`m` Integer. HNSW param: number of connections between layers. Defaults to `16L`.

`ef_construction` Integer. HNSW param: size of the dynamic candidate list during construction. Defaults to `200L`.

`ef_search` Integer. HNSW param: size of the candidate list at query time. Higher values improve recall at the cost of speed. Defaults to `100L`.

`n_list` Integer or `NULL`. IVF param: number of clusters to generate. If `NULL`, defaults to `sqrt(n)` internally. Defaults to `NULL`.

`n_probe` Integer or `NULL`. IVF param: number of clusters to query. If `NULL`, defaults to `sqrt(n_list)` internally. Defaults to `NULL`.

Value

A named list with the following elements:

- k - Integer. Number of neighbours. Defaults to `15L`.
- knn_method - String. Which method to use for the approximate nearest neighbour search. One of `c("kmknn", "hnsw", "annoy", "nndescent", "ivf", "exhaustive")`. Defaults to `"kmknn"`.
- ann_dist - String. Distance metric to use. One of `c("euclidean", "cosine")`. Defaults to `"euclidean"`.
- n_trees - Integer. Annoy param: number of trees. Defaults to `50L`.
- search_budget - Integer or `NULL`. Annoy param: optional search budget per tree. If `NULL`, defaults to `n_trees * k * 20L` internally. Defaults to `NULL`.
- delta - Numeric. NNDescent param: early termination criterion. Defaults to `0.001`.
- diversify_prob - Numeric. NNDescent param: diversification probability applied at the end of index construction. Defaults to `0.0`.
- ef_budget - Integer or `NULL`. NNDescent param: optional query budget. Higher values improve recall at the cost of speed. Defaults to `NULL`.

- extract_knn - Boolean. NNDescent param: hand back the graph the descent already built instead of beam searching it. Skips the query pass entirely, so it is much faster, at the cost of some recall. Rows the descent never filled come back padded with duplicate edges. Ignored by every other method. Defaults to `FALSE`.
- m - Integer. HNSW param: number of connections between layers. Defaults to `16L`.
- ef_construction - Integer. HNSW param: size of the dynamic candidate list during construction. Defaults to `200L`.
- ef_search - Integer. HNSW param: size of the candidate list at query time. Higher values improve recall at the cost of speed. Defaults to `100L`.
- n_list - Integer or `NULL`. IVF param: number of clusters to generate. If `NULL`, defaults to `sqrt(n)` internally. Defaults to `NULL`.
- n_probe - Integer or `NULL`. IVF param: number of clusters to query. If `NULL`, defaults to `sqrt(n_list)` internally. Defaults to `NULL`.

`params_sc_magic` *Wrapper function for MAGIC imputation parameters*

Description

Parameters controlling the MAGIC imputation run by `run_magic_sc()`. The defaults mirror the reference implementation.

Usage

```
params_sc_magic(
  n_steps = 3L,
  clip_threshold = 0.01,
  gene_batch_size = 1000L,
  layer = c("norm", "raw"),
  allow_large = FALSE
)
```

Arguments

`n_steps`	Integer. Diffusion steps applied to the counts. Zero is legal and hands back the un-imputed values, which is a cheap way to compare the two. Defaults to `3L`.
`clip_threshold`	Numeric. Imputed values below this are zeroed after the last step. Defaults to `0.01`.
`gene_batch_size`	Integer. Genes streamed off the binary store per block. Bounds the scratch memory and is clamped to the number of requested genes. Defaults to `1000L`.

`layer` String. Which stored layer to impute. The operator preserves per-cell mass, so imputed values sit on the scale of whatever went in: imputing raw counts and imputing log-normalised counts are different operations rather than the same one rescaled. One of `c("norm", "raw")`. Defaults to `"norm"`.

`allow_large` Boolean. Skip the output size guard. The dense output is capped at 1e9 elements, i.e. 4 GB of `f32`. Defaults to `FALSE`.

Value

A named list with the following elements:

- n_steps - Integer. Diffusion steps applied to the counts. Zero is legal and hands back the un-imputed values, which is a cheap way to compare the two. Defaults to `3L`.
- clip_threshold - Numeric. Imputed values below this are zeroed after the last step. Defaults to `0.01`.
- gene_batch_size - Integer. Genes streamed off the binary store per block. Bounds the scratch memory and is clamped to the number of requested genes. Defaults to `1000L`.
- layer - String. Which stored layer to impute. The operator preserves per-cell mass, so imputed values sit on the scale of whatever went in: imputing raw counts and imputing log-normalised counts are different operations rather than the same one rescaled. One of `c("norm", "raw")`. Defaults to `"norm"`.
- allow_large - Boolean. Skip the output size guard. The dense output is capped at 1e9 elements, i.e. 4 GB of `f32`. Defaults to `FALSE`.

References

van Dijk, et al., Cell, 2018.

`params_sc_miloR` *Wrapper function for parameters for MiloR*

Description

Wrapper function for parameters for MiloR

Usage

```
params_sc_miloR(
  prop = 0.2,
  k_refine = 20L,
  refinement_strategy = c("index", "approximate", "bruteforce"),
  index_type = c("nndescent", "ivf", "hnsw", "annoy", "exhaustive"),
  knn = list()
)
```

Arguments

prop	Numeric. Proportion of cells to sample as neighbourhood indices. Defaults to 0.2. Must be in (0,1).
k_refine	Integer. Number of neighbours to use for refinement. Defaults to 20L.
refinement_strategy	String. Strategy for refining sampled indices. One of c("approximate", "bruteforce", "index"). Defaults to "index".
index_type	String. Type of kNN index to use. One of c("nndescent", "ivf", "hnsw", "annoy", "exhaustive"). Defaults to "nndescent". "exhaustive" scans every cell, so it returns the true nearest neighbour rather than an approximation, at a cost that grows with the number of cells.
knn	List. Optional overrides for kNN parameters. See params_knn_defaults() for available parameters: k, knn_method, ann_dist, search_budget, n_trees, delta, diversify_prob, ef_budget, extract_knn, m, ef_construction, ef_search, n_list and n_probe.

Value

A list with the MiloR parameters.

---

params_sc_min_quality

Wrapper function to generate QC metric params for single cell

---

Description

Wrapper function to generate QC metric params for single cell

Usage

```
params_sc_min_quality(
  min_unique_genes = 100L,
  min_lib_size = 250L,
  min_cells = 10L,
  target_size = 10000
)
```

Arguments

min_unique_genes	Integer. Minimum number of unique genes per cell/spot to be included. Defaults to 100L.
min_lib_size	Integer. Minimum library size per cell/spot to be included. Defaults to 250L.
min_cells	Integer. Minimum number of cells a gene has to be expressed to be included. Defaults to 10L.
target_size	Numeric. The target size for the normalisation. Defaults to 10000.0.

Value

A named list with the following elements:

- min_unique_genes - Integer. Minimum number of unique genes per cell/spot to be included. Defaults to `100L`.
- min_lib_size - Integer. Minimum library size per cell/spot to be included. Defaults to `250L`.
- min_cells - Integer. Minimum number of cells a gene has to be expressed to be included. Defaults to `10L`.
- target_size - Numeric. The target size for the normalisation. Defaults to `10000.0`.

`params_sc_mtx_io` *Wrapper function to provide data for mtx-based loading*

Description

Wrapper function to provide data for mtx-based loading

Usage

```
params_sc_mtx_io(path_mtx, path_obs, path_var, cells_as_rows, has_hdr)
```

Arguments

`path_mtx`	Any. Path to the .mtx file Required.
`path_obs`	Any. Path to the file containing cell/barcode info. Required.
`path_var`	Any. Path to the file containing gene/variable info. Required.
`cells_as_rows`	Boolean. Do cells represent the rows or columns. Required.
`has_hdr`	Boolean. Do the plain text files have headers. Required.

Value

A named list with the following elements:

- path_mtx - Any. Path to the .mtx file Required.
- path_obs - Any. Path to the file containing cell/barcode info. Required.
- path_var - Any. Path to the file containing gene/variable info. Required.
- cells_as_rows - Boolean. Do cells represent the rows or columns. Required.
- has_hdr - Boolean. Do the plain text files have headers. Required.

`params_sc_neighbours` *Wrapper function for parameters for neighbour identification in single cell*

Description

Wrapper function for parameters for neighbour identification in single cell

Usage

```
params_sc_neighbours(
  full_snn = TRUE,
  pruning = 0.0833333333333333,
  snn_similarity = c("jaccard", "rank"),
  knn = list()
)
```

Arguments

`full_snn`	Boolean. Shall the full shared nearest neighbour graph be generated that generates edges between all cells instead of between only neighbours. Defaults to `TRUE`.
`pruning`	Numeric. Weights below this threshold will be set to 0 in the generation of the sNN graph. Seurat uses for example `1/15` with `k = 20`. As the default k is set to 15, we set it to `1/12`. Track this against `k` rather than leaving it: the threshold is a share of the neighbourhood, so the same value prunes far harder at a larger `k`. A cell that loses every edge to pruning keeps its strongest kNN edge, so it does not drop out as a one-cell cluster. Over-pruning still fails quietly otherwise: small groups of cells that only share neighbours with each other split off as tiny communities. Defaults to `0.0833333333333333`.
`snn_similarity`	String. The Jaccard similarity calculates the Jaccard between the neighbours, whereas the rank method calculates edge weights based on the ranking of shared neighbours. For the rank method, the weight is determined by finding the shared neighbour with the lowest combined rank across both cells, where lower-ranked (closer) shared neighbours result in higher edge weights Both methods produce weights normalised to the range `[0, 1]`. One of `c("jaccard", "rank")`. Defaults to `"jaccard"`.
`knn`	List. Optional overrides for kNN parameters. See `params_knn_defaults()` for available parameters: `k`, `knn_method`, `ann_dist`, `search_budget`, `n_trees`, `delta`, `diversify_prob`, `ef_budget`, `extract_knn`, `m`, `ef_construction`, `ef_search`, `n_list` and `n_probe`. See `params_knn_defaults()` for the available elements. Defaults to `list()`.

Value

A named list with the following elements:

- full_snn - Boolean. Shall the full shared nearest neighbour graph be generated that generates edges between all cells instead of between only neighbours. Defaults to `TRUE`.
- pruning - Numeric. Weights below this threshold will be set to 0 in the generation of the sNN graph. Seurat uses for example `1/15` with `k = 20`. As the default k is set to 15, we set it to `1/12`. Track this against `k` rather than leaving it: the threshold is a share of the neighbourhood, so the same value prunes far harder at a larger `k`. A cell that loses every edge to pruning keeps its strongest kNN edge, so it does not drop out as a one-cell cluster. Over-pruning still fails quietly otherwise: small groups of cells that only share neighbours with each other split off as tiny communities. Defaults to `0.08333333333333333`.
- snn_similarity - String. The Jaccard similarity calculates the Jaccard between the neighbours, whereas the rank method calculates edge weights based on the ranking of shared neighbours. For the rank method, the weight is determined by finding the shared neighbour with the lowest combined rank across both cells, where lower-ranked (closer) shared neighbours result in higher edge weights Both methods produce weights normalised to the range `[0, 1]`. One of `c("jaccard", "rank")`. Defaults to `"jaccard"`.
- The elements of `params_knn_defaults()`, overridden by `knn`, spliced in at this position.

---

params_sc_palantir *Wrapper function for Palantir parameters*

Description

Parameters controlling the Palantir trajectory inference. The kNN graph you hand to `run_palantir_sc()` feeds the diffusion kernel. The geodesics are measured over a second kNN graph that Palantir builds internally on the multiscale space, and `knn` is what controls that one. It is a different knob from `k` in the kNN parameter block, which sizes the backend index. Palantir overrides `k` and `ann_dist` for its internal search, so only `knn_method` and the backend tuning parameters have an effect.

Usage

```
params_sc_palantir(
  n_dcs = 10L,
  n_eigs = NULL,
  knn = 30L,
  num_waypoints = 1200L,
  scale_components = TRUE,
  use_early_cell_as_start = TRUE,
  max_iterations = 25L,
  branch_prob_threshold = 0.01,
```

```
  lanczos_basis_size = NULL,
  lanczos_max_restarts = 16L,
  lanczos_tol = 1e-08,
  knn_params = list()
)
```

Arguments

n_dcs	Integer. Diffusion components to extract before the multiscale scaling. Defaults to 10L.
n_eigs	Integer or NULL. Eigenvectors to retain, not components: the trivial leading eigenvector is counted here and then dropped, so 3L leaves two multiscale components. If NULL, the count is picked from the largest eigengap, as the reference does. Defaults to NULL.
knn	Integer. Neighbours for the geodesic graph over the multiscale space, in the reference's self-inclusive convention. Defaults to 30L.
num_waypoints	Integer. Target waypoint count for the max-min sampler. Defaults to 1200L.
scale_components	Boolean. Min-max scale each multiscale component to [0, 1] before any distance is taken. Defaults to TRUE.
use_early_cell_as_start	Boolean. Use the provided early cell directly rather than snapping it to the nearest diffusion-map boundary cell. Defaults to TRUE.
max_iterations	Integer. Iteration cap for the pseudotime refinement. Defaults to 25L.
branch_prob_threshold	Numeric. Fate probabilities below this are zeroed. Defaults to 0.01.
lanczos_basis_size	Integer or NULL. Krylov basis vectors held at once during the diffusion-map eigendecomposition. If NULL, derived from the requested component count. Defaults to NULL.
lanczos_max_restarts	Integer. Maximum restart cycles for the Lanczos solver. Defaults to 16L.
lanczos_tol	Numeric. Relative residual tolerance for the Lanczos solver. Defaults to 1e-08.
knn_params	List. Optional overrides for the kNN parameters of the internal multiscale search. See params_knn_defaults() for available parameters: k, knn_method, ann_dist, search_budget, n_trees, delta, diversify_prob, ef_budget, m, ef_construction, ef_search, n_list and n_probe. See params_knn_defaults() for the available elements. Defaults to list().

Value

A named list with the following elements:

- n_dcs - Integer. Diffusion components to extract before the multiscale scaling. Defaults to 10L.

- n_eigs - Integer or NULL. Eigenvectors to retain, not components: the trivial leading eigenvector is counted here and then dropped, so 3L leaves two multiscale components. If NULL, the count is picked from the largest eigengap, as the reference does. Defaults to NULL.
- knn - Integer. Neighbours for the geodesic graph over the multiscale space, in the reference's self-inclusive convention. Defaults to 30L.
- num_waypoints - Integer. Target waypoint count for the max-min sampler. Defaults to 1200L.
- scale_components - Boolean. Min-max scale each multiscale component to [0, 1] before any distance is taken. Defaults to TRUE.
- use_early_cell_as_start - Boolean. Use the provided early cell directly rather than snapping it to the nearest diffusion-map boundary cell. Defaults to TRUE.
- max_iterations - Integer. Iteration cap for the pseudotime refinement. Defaults to 25L.
- branch_prob_threshold - Numeric. Fate probabilities below this are zeroed. Defaults to 0.01.
- lanczos_basis_size - Integer or NULL. Krylov basis vectors held at once during the diffusion-map eigendecomposition. If NULL, derived from the requested component count. Defaults to NULL.
- lanczos_max_restarts - Integer. Maximum restart cycles for the Lanczos solver. Defaults to 16L.
- lanczos_tol - Numeric. Relative residual tolerance for the Lanczos solver. Defaults to 1e-08.
- The elements of params_knn_defaults(), overridden by knn_params, spliced in at this position.

References

Setty, et al., Nat. Biotechnol., 2019.

---

params_sc_pca *Wrapper for PCA specifically designed for single cells*

Description

Wrapper for PCA specifically designed for single cells

Usage

```
params_sc_pca(
  mean_center = TRUE,
  normalise_variance = TRUE,
  randomised = NULL,
  clr = FALSE,
  size_factor = 10000,
  svd_solver = c("covariance", "randomised", "exact")
)
```

Arguments

`mean_center`	Boolean. Shall the data be mean centred Defaults to `TRUE`.
`normalise_variance`	Boolean. Shall the data have normalised variance Defaults to `TRUE`.
`randomised`	Boolean or `NULL`. Deprecated, use `svd_solver`. `TRUE` maps to `"randomised"`, `FALSE` to `"exact"`. Defaults to `NULL`.
`clr`	Boolean. Shall the CLR-type `PFlogPF` be applied, see Booeshaghi, et al. Defaults to `FALSE`.
`size_factor`	Numeric. The used size factor during I/O. It needs to be the same as during I/O to have correct results when using the `PFlogPF` transformation. Defaults to `10000.0`.
`svd_solver`	String. Which solver to use. `"covariance"` builds the gene x gene cross-product and eigendecomposes it: exact, and the fastest option for a few thousand HVGs, but its cost grows with the square of the gene number in memory and the cube in time. `"randomised"` is a randomised SVD, approximate in the trailing components. `"exact"` is a full SVD on the dense path and Lanczos on the sparse one. One of `c("covariance", "randomised", "exact")`. Defaults to `"covariance"`.

Value

A named list with the following elements:

- mean_center - Boolean. Shall the data be mean centred Defaults to `TRUE`.
- normalise_variance - Boolean. Shall the data have normalised variance Defaults to `TRUE`.
- randomised - Boolean or `NULL`. Deprecated, use `svd_solver`. `TRUE` maps to `"randomised"`, `FALSE` to `"exact"`. Defaults to `NULL`.
- clr - Boolean. Shall the CLR-type `PFlogPF` be applied, see Booeshaghi, et al. Defaults to `FALSE`.
- size_factor - Numeric. The used size factor during I/O. It needs to be the same as during I/O to have correct results when using the `PFlogPF` transformation. Defaults to `10000.0`.
- svd_solver - String. Which solver to use. `"covariance"` builds the gene x gene cross-product and eigendecomposes it: exact, and the fastest option for a few thousand HVGs, but its cost grows with the square of the gene number in memory and the cube in time. `"randomised"` is a randomised SVD, approximate in the trailing components. `"exact"` is a full SVD on the dense path and Lanczos on the sparse one. One of `c("covariance", "randomised", "exact")`. Defaults to `"covariance"`.

`params_sc_sctransform` *Wrapper function for scTransform (v2) parameters*

Description

Defaults are sctransform's own with `vst.flavor = "v2"` applied. Only the step-1 fit scales with `n_genes` and `n_cells`: those bound the subsample the negative binomial models are fitted on, and every later pass streams gene by gene. Raising them costs fitting time, not memory.

Usage

```
params_sc_sctransform(
  n_genes = 2000L,
  n_cells = 2000L,
  min_cells = 5L,
  bw_adjust = 3,
  gmean_eps = 1,
  outlier_th = 10,
  poisson_diff_theta = 0.001,
  clip_min = NULL,
  clip_max = NULL
)
```

Arguments

`n_genes`	Integer. Genes in the step-1 subsample. Defaults to `2000L`.
`n_cells`	Integer. Cells in the step-1 subsample. Defaults to `2000L`.
`min_cells`	Integer. Minimum number of cells a gene must be detected in to be modelled. Defaults to `5L`.
`bw_adjust`	Numeric. Bandwidth multiplier for the kernel regression that regularises the parameters. Defaults to `3.0`.
`gmean_eps`	Numeric. Offset in the geometric mean. Defaults to `1.0`.
`outlier_th`	Numeric. Threshold, in median absolute deviations, past which a step-1 fit is treated as an outlier. Defaults to `10.0`.
`poisson_diff_theta`	Numeric. Below this, the fitted dispersion is taken as the Poisson limit. Defaults to `0.001`.
`clip_min`	Numeric or `NULL`. Lower residual clipping bound. `NULL` uses `-sqrt(n_cells)`. Must be given together with `clip_max`. Defaults to `NULL`.
`clip_max`	Numeric or `NULL`. Upper residual clipping bound. `NULL` uses `sqrt(n_cells)`. Must be given together with `clip_min`. Defaults to `NULL`.

Value

A named list with the following elements:

- n_genes - Integer. Genes in the step-1 subsample. Defaults to `2000L`.
- n_cells - Integer. Cells in the step-1 subsample. Defaults to `2000L`.
- min_cells - Integer. Minimum number of cells a gene must be detected in to be modelled. Defaults to `5L`.
- bw_adjust - Numeric. Bandwidth multiplier for the kernel regression that regularises the parameters. Defaults to `3.0`.
- gmean_eps - Numeric. Offset in the geometric mean. Defaults to `1.0`.
- outlier_th - Numeric. Threshold, in median absolute deviations, past which a step-1 fit is treated as an outlier. Defaults to `10.0`.
- poisson_diff_theta - Numeric. Below this, the fitted dispersion is taken as the Poisson limit. Defaults to `0.001`.
- clip_min - Numeric or `NULL`. Lower residual clipping bound. `NULL` uses `-sqrt(n_cells)`. Must be given together with `clip_max`. Defaults to `NULL`.
- clip_max - Numeric or `NULL`. Upper residual clipping bound. `NULL` uses `sqrt(n_cells)`. Must be given together with `clip_min`. Defaults to `NULL`.

References

Choudhary and Satija, Genome Biology, 2022.

`params_sc_seacells` *Wrapper function for the SEACells parameters*

Description

Wrapper function for the SEACells parameters

Usage

```
params_sc_seacells(
  n_sea_cells,
  max_fw_iters = 50L,
  convergence_epsilon = 0.001,
  max_iter = 100L,
  min_iter = 10L,
  greedy_threshold = 20000L,
  graph_building = "union",
  pruning = TRUE,
  pruning_threshold = 1e-07,
  n_landmarks = NULL,
  knn = list()
)
```

Arguments

`n_sea_cells`	Integer. Number of SEA cells to detect. Required.
`max_fw_iters`	Integer. Maximum iterations for the Franke-Wolfe algorithm. Defaults to `50L`.
`convergence_epsilon`	Numeric. Convergence threshold. Algorithm stops when RSS change < epsilon * RSS(0). Defaults to `0.001`.
`max_iter`	Integer. Maximum iterations to run SEACells for. Defaults to `100L`.
`min_iter`	Integer. Minimum iterations to run SEACells for. Defaults to `10L`.
`greedy_threshold`	Integer. Maximum number of cells before defaulting to rapid random selection of archetypes. Defaults to `20000L`.
`graph_building`	String. Graph building method. Defaults to `"union"`.
`pruning`	Boolean. Shall tiny values be pruned during Franke-Wolfe updates. Defaults to `TRUE`.
`pruning_threshold`	Numeric. If `pruning = TRUE` values below which threshold shall be pruned. Defaults to `1e-07`.
`n_landmarks`	Integer or `NULL`. If provided, it will use the Nystroem extension during the archetype finding. Useful for larger data sets. Defaults to `NULL`.
`knn`	List. Optional overrides for kNN parameters. See `params_knn_defaults()` for available parameters: `k`, `knn_method`, `ann_dist`, `search_budget`, `n_trees`, `delta`, `diversify_prob`, `ef_budget`, `extract_knn`, `m`, `ef_construction`, `ef_search`, `n_list` and `n_probe`. See `params_knn_defaults()` for the available elements. Defaults to `list()`.

Value

A named list with the following elements:

- n_sea_cells - Integer. Number of SEA cells to detect. Required.
- max_fw_iters - Integer. Maximum iterations for the Franke-Wolfe algorithm. Defaults to `50L`.
- convergence_epsilon - Numeric. Convergence threshold. Algorithm stops when RSS change < epsilon * RSS(0). Defaults to `0.001`.
- max_iter - Integer. Maximum iterations to run SEACells for. Defaults to `100L`.
- min_iter - Integer. Minimum iterations to run SEACells for. Defaults to `10L`.
- greedy_threshold - Integer. Maximum number of cells before defaulting to rapid random selection of archetypes. Defaults to `20000L`.
- graph_building - String. Graph building method. Defaults to `"union"`.
- pruning - Boolean. Shall tiny values be pruned during Franke-Wolfe updates. Defaults to `TRUE`.

- pruning_threshold - Numeric. If `pruning = TRUE` values below which threshold shall be pruned. Defaults to `1e-07`.
- n_landmarks - Integer or `NULL`. If provided, it will use the Nystroem extension during the archetype finding. Useful for larger data sets. Defaults to `NULL`.
- The elements of `params_knn_defaults()`, overridden by `knn`, spliced in at this position.

`params_sc_seurat_cca` *Wrapper function for the Seurat CCA parameters*

Description

Wrapper function for the Seurat CCA parameters

Usage

```
params_sc_seurat_cca(
  num_cc = 30L,
  dims = 30L,
  k_anchor = 5L,
  k_filter = 200L,
  k_score = 30L,
  k_weight = 100L,
  n_top_features = 200L,
  l2_norm = TRUE,
  sd = 1,
  knn = list(),
  pca = params_sc_pca()
)
```

Arguments

`num_cc`	Integer. Number of canonical correlation dimensions to compute for the anchor space. The effective rank used is `max(num_cc, dims)`. Defaults to `30L`.
`dims`	Integer. Number of dimensions used for the anchor kNN queries and the size of the returned embedding. Defaults to `30L`.
`k_anchor`	Integer. Neighbourhood size for the mutual nearest neighbour anchor search. Defaults to `5L`.
`k_filter`	Integer. Neighbourhood size for the gene-space anchor filter. Defaults to `200L`.
`k_score`	Integer. Neighbourhood size for the shared-neighbour anchor scoring. Defaults to `30L`.
`k_weight`	Integer. Neighbourhood size for the kernel weights applied during the correction. Defaults to `100L`.

`n_top_features`
: Integer. Number of top-loading genes used for the gene-space anchor filter. Defaults to `200L`.

`l2_norm`
: Boolean. Shall the canonical correlation embedding be L2-normalised per cell. Defaults to `TRUE`.

`sd`
: Numeric. Bandwidth divisor of the Gaussian kernel used for the anchor weights. Defaults to `1.0`.

`knn`
: List. Optional overrides for kNN parameters. See `params_knn_defaults()` for available parameters: `k`, `knn_method`, `ann_dist`, `search_budget`, `n_trees`, `delta`, `diversify_prob`, `ef_budget`, `extract_knn`, `m`, `ef_construction`, `ef_search`, `n_list` and `n_probe`. Note that `k` is unused here, the neighbourhood sizes come from `k_anchor`, `k_filter`, `k_score` and `k_weight`. See `params_knn_defaults()` for the available elements. Defaults to `list()`.

`pca`
: List. Parameters to feed through to the optional recalculation of the PCA, see `params_sc_pca()`. See `params_sc_pca()` for the available elements. Defaults to `params_sc_pca()`.

Value

A named list with the following elements:

- num_cc - Integer. Number of canonical correlation dimensions to compute for the anchor space. The effective rank used is `max(num_cc, dims)`. Defaults to `30L`.
- dims - Integer. Number of dimensions used for the anchor kNN queries and the size of the returned embedding. Defaults to `30L`.
- k_anchor - Integer. Neighbourhood size for the mutual nearest neighbour anchor search. Defaults to `5L`.
- k_filter - Integer. Neighbourhood size for the gene-space anchor filter. Defaults to `200L`.
- k_score - Integer. Neighbourhood size for the shared-neighbour anchor scoring. Defaults to `30L`.
- k_weight - Integer. Neighbourhood size for the kernel weights applied during the correction. Defaults to `100L`.
- n_top_features - Integer. Number of top-loading genes used for the gene-space anchor filter. Defaults to `200L`.
- l2_norm - Boolean. Shall the canonical correlation embedding be L2-normalised per cell. Defaults to `TRUE`.
- sd - Numeric. Bandwidth divisor of the Gaussian kernel used for the anchor weights. Defaults to `1.0`.
- The elements of `params_knn_defaults()`, overridden by `knn`, spliced in at this position.
- The elements of `params_sc_pca()`, overridden by `pca`, spliced in at this position.

References

Stuart, et al., Cell, 2019

`params_sc_seurat_rpca` *Wrapper function for the Seurat rPCA parameters*

Description

Wrapper function for the Seurat rPCA parameters

Usage

```
params_sc_seurat_rpca(
  dims = 30L,
  k_anchor = 5L,
  k_score = 30L,
  k_weight = 100L,
  l2_norm = TRUE,
  sd = 1,
  knn = list(),
  pca = params_sc_pca()
)
```

Arguments

`dims`	Integer. Number of dimensions used for the per-batch PCA projections, the anchor kNN queries and the size of the returned embedding. Defaults to `30L`.
`k_anchor`	Integer. Neighbourhood size for the mutual nearest neighbour anchor search. Defaults to `5L`.
`k_score`	Integer. Neighbourhood size for the shared-neighbour anchor scoring. Defaults to `30L`.
`k_weight`	Integer. Neighbourhood size for the kernel weights applied during the correction. Defaults to `100L`.
`l2_norm`	Boolean. Shall the projected embeddings be L2-normalised per cell. Defaults to `TRUE`.
`sd`	Numeric. Bandwidth divisor of the Gaussian kernel used for the anchor weights. Defaults to `1.0`.
`knn`	List. Optional overrides for kNN parameters. See `params_knn_defaults()` for available parameters: `k`, `knn_method`, `ann_dist`, `search_budget`, `n_trees`, `delta`, `diversify_prob`, `ef_budget`, `extract_knn`, `m`, `ef_construction`, `ef_search`, `n_list` and `n_probe`. Note that `k` is unused here, the neighbourhood sizes come from `k_anchor`, `k_score` and `k_weight`. See `params_knn_defaults()` for the available elements. Defaults to `list()`.
`pca`	List. Parameters to feed through to the optional recalculation of the PCA, see `params_sc_pca()`. See `params_sc_pca()` for the available elements. Defaults to `params_sc_pca()`.

Value

A named list with the following elements:

- dims - Integer. Number of dimensions used for the per-batch PCA projections, the anchor kNN queries and the size of the returned embedding. Defaults to `30L`.
- k_anchor - Integer. Neighbourhood size for the mutual nearest neighbour anchor search. Defaults to `5L`.
- k_score - Integer. Neighbourhood size for the shared-neighbour anchor scoring. Defaults to `30L`.
- k_weight - Integer. Neighbourhood size for the kernel weights applied during the correction. Defaults to `100L`.
- l2_norm - Boolean. Shall the projected embeddings be L2-normalised per cell. Defaults to `TRUE`.
- sd - Numeric. Bandwidth divisor of the Gaussian kernel used for the anchor weights. Defaults to `1.0`.
- The elements of `params_knn_defaults()`, overridden by `knn`, spliced in at this position.
- The elements of `params_sc_pca()`, overridden by `pca`, spliced in at this position.

References

Stuart, et al., Cell, 2019

`params_sc_supercell` *Wrapper function for parameters for SuperCell generation*

Description

Wrapper function for parameters for SuperCell generation

Usage

```
params_sc_supercell(
  walk_length = 3L,
  graining_factor = 20,
  use_kernel = TRUE,
  k_ith = NULL,
  max_support = NULL,
  knn = list()
)
```

Arguments

walk_length	Integer. Walk length for the Walktrap algorithm. Defaults to 3L.
graining_factor	Numeric. Graining level of data (proportion of number of single cells in the initial dataset to the number of metacells in the final dataset). (One meta cell per 20 cells.) Defaults to 20.0.
use_kernel	Boolean. Shall a kernel function akin to MAGIC be applied akin to the approach in SuperCell2, see Hérault, et al., bioRxiv, 2026 and van Dijk, et al., Cell, 2018. Defaults to TRUE.
k_ith	Integer or NULL. The k-ith neighbour to use for the kernel. Defaults to NULL.
max_support	Integer or NULL. Caps each cell's walk-probability vector to its top entries by mass, bounding memory at ~max_support * n_cells on large data. Makes the result an approximation. NULL (default) keeps the walks exact. Defaults to NULL.
knn	List. Optional overrides for kNN parameters. See params_knn_defaults() for available parameters: k, knn_method, ann_dist, search_budget, n_trees, delta, diversify_prob, ef_budget, extract_knn, m, ef_construction, ef_search, n_list and n_probe. See params_knn_defaults() for the available elements. Defaults to list().

Value

A named list with the following elements:

- walk_length - Integer. Walk length for the Walktrap algorithm. Defaults to 3L.
- graining_factor - Numeric. Graining level of data (proportion of number of single cells in the initial dataset to the number of metacells in the final dataset). (One meta cell per 20 cells.) Defaults to 20.0.
- use_kernel - Boolean. Shall a kernel function akin to MAGIC be applied akin to the approach in SuperCell2, see Hérault, et al., bioRxiv, 2026 and van Dijk, et al., Cell, 2018. Defaults to TRUE.
- k_ith - Integer or NULL. The k-ith neighbour to use for the kernel. Defaults to NULL.
- max_support - Integer or NULL. Caps each cell's walk-probability vector to its top entries by mass, bounding memory at ~max_support * n_cells on large data. Makes the result an approximation. NULL (default) keeps the walks exact. Defaults to NULL.
- The elements of params_knn_defaults(), overridden by knn, spliced in at this position.

`params_sc_synthetic_cellsweep` *Default parameters for generation of synthetic CellSweep data*

Description

Shapes the fixture with a planted ambient profile that `generate_cellsweep_test_data()` builds. The defaults give 600 real barcodes over 3 cell types plus 2000 empty droplets, on 200 genes.

Usage

```
params_sc_synthetic_cellsweep(
  n_real = 600L,
  n_empty = 2000L,
  n_genes = 200L,
  n_celltypes = 3L,
  n_markers = 20L,
  marker_weight = 25,
  ambient_dominance = 0.6,
  alpha_mean = 0.3,
  alpha_sd = 0.12,
  real_lib_size = 3000L,
  empty_lib_size = 120L
)
```

Arguments

`n_real`	Integer. Number of real barcodes. Cell types are assigned round-robin over them. Defaults to `600L`.
`n_empty`	Integer. Number of empty droplets. The ambient profile is estimated off these, so at least 30 and preferably a lot more. Defaults to `2000L`.
`n_genes`	Integer. Number of genes. Defaults to `200L`.
`n_celltypes`	Integer. Number of cell types. Defaults to `3L`.
`n_markers`	Integer. Width of each cell type's marker block. The blocks are contiguous and disjoint, so `n_markers * n_celltypes` has to fit into `n_genes`. Defaults to `20L`.
`marker_weight`	Numeric. Enrichment of a marker gene over background in its own cell type's profile. Must exceed 1. Defaults to `25.0`.
`ambient_dominance`	Numeric. Fraction of the soup coming from the first cell type. The remainder is flat background. Defaults to `0.6`.
`alpha_mean`	Numeric. Mean planted ambient fraction across real barcodes. Defaults to `0.3`.
`alpha_sd`	Numeric. Spread of the planted ambient fraction. Defaults to `0.12`.

`real_lib_size` Integer. Expected library size of a real barcode. Defaults to `3000L`.

`empty_lib_size`
Integer. Expected library size of an empty droplet. Defaults to `120L`.

Details

The soup is the first cell type plus flat background rather than a mixture of every cell type profile. A soup sitting in the span of the cell type profiles makes the contamination fraction unidentifiable, and the fixture would then be testing the repulsion term rather than the model.

Value

A named list with the following elements:

- n_real - Integer. Number of real barcodes. Cell types are assigned round-robin over them. Defaults to `600L`.
- n_empty - Integer. Number of empty droplets. The ambient profile is estimated off these, so at least 30 and preferably a lot more. Defaults to `2000L`.
- n_genes - Integer. Number of genes. Defaults to `200L`.
- n_celltypes - Integer. Number of cell types. Defaults to `3L`.
- n_markers - Integer. Width of each cell type's marker block. The blocks are contiguous and disjoint, so `n_markers * n_celltypes` has to fit into `n_genes`. Defaults to `20L`.
- marker_weight - Numeric. Enrichment of a marker gene over background in its own cell type's profile. Must exceed 1. Defaults to `25.0`.
- ambient_dominance - Numeric. Fraction of the soup coming from the first cell type. The remainder is flat background. Defaults to `0.6`.
- alpha_mean - Numeric. Mean planted ambient fraction across real barcodes. Defaults to `0.3`.
- alpha_sd - Numeric. Spread of the planted ambient fraction. Defaults to `0.12`.
- real_lib_size - Integer. Expected library size of a real barcode. Defaults to `3000L`.
- empty_lib_size - Integer. Expected library size of an empty droplet. Defaults to `120L`.

`params_sc_synthetic_data`

Default parameters for generation of synthetic single cell data (RNA)

Description

For the generation of synthetic single cell data mostly for testing or showcasing purposes. The default configurations generates 1000 cells x 100 genes with genes 1:10 being cell markers for cell type 1, genes 11:20 for cell type 2 and genes 21:30 for cell type.

Usage

```
params_sc_synthetic_data(
  n_cells = 1000L,
  n_genes = 100L,
  n_batches = 1L,
  marker_genes = list(cell_type_1 = list(marker_genes = 0:9), cell_type_2 =
    list(marker_genes = 10:19), cell_type_3 = list(marker_genes = 20:29)),
  batch_effect_strength = c("strong", "medium", "weak"),
  n_samples = NULL,
  sample_bias = NULL
)
```

Arguments

`n_cells`	Integer. Number of cells. Defaults to `1000L`.
`n_genes`	Integer. Number of genes. Defaults to `100L`.
`n_batches`	Integer. Number of batches. Defaults to `1L`.
`marker_genes`	Any. A nested list that indicates which gene indices are markers for which cell. Defaults to `list(cell_type_1 = list(marker_genes = 0:9), cell_type_2 = list(marker_genes = 10:19), cell_type_3 = list(marker_genes = 20:29))`.
`batch_effect_strength`	String. The strength of the batch effect to add. One of `c("strong", "medium", "weak")`. Defaults to `"strong"`.
`n_samples`	Integer or `NULL`. Shall sample membership be added to the synthetic data. If you want sample information you need to provide `n_samples` and `sample_bias`. Defaults to `NULL`.
`sample_bias`	Any. One of `c("even", "slightly_uneven", "very_uneven")` Defaults to `NULL`.

Value

A named list with the following elements:

- n_cells - Integer. Number of cells. Defaults to `1000L`.
- n_genes - Integer. Number of genes. Defaults to `100L`.
- marker_genes - Any. A nested list that indicates which gene indices are markers for which cell. Defaults to `list(cell_type_1 = list(marker_genes = 0:9), cell_type_2 = list(marker_genes = 10:19), cell_type_3 = list(marker_genes = 20:29))`.
- n_batches - Integer. Number of batches. Defaults to `1L`.
- batch_effect_strength - String. The strength of the batch effect to add. One of `c("strong", "medium", "weak")`. Defaults to `"strong"`.
- n_samples - Integer or `NULL`. Shall sample membership be added to the synthetic data. If you want sample information you need to provide `n_samples` and `sample_bias`. Defaults to `NULL`.
- sample_bias - Any. One of `c("even", "slightly_uneven", "very_uneven")` Defaults to `NULL`.

`params_sc_synthetic_data_adt` *Default parameters for generation of synthetic single cell data (ADT)*

Description

For the generation of synthetic single cell data mostly for testing or showcasing purposes. In this case, ADT counts to test multi-modal integration. The default configurations generates 1000 cells x 15 proteins with probes 1:3 being cell markers for cell type 1, genes 4:6 for cell type 2 and genes 7:9 for cell type. Columns 13:15 represents isotype controls.

Usage

```
params_sc_synthetic_data_adt(
  n_cells = 1000L,
  n_proteins = 15L,
  n_batches = 1L,
  marker_genes = list(cell_type_1 = list(marker_genes = 0:2), cell_type_2 =
    list(marker_genes = 3:5), cell_type_3 = list(marker_genes = 6:8)),
  isotype_controls = 12:14,
  batch_effect_strength = c("strong", "medium", "weak")
)
```

Arguments

`n_cells`	Integer. Number of cells. Defaults to `1000L`.
`n_proteins`	Integer. Number of proteins Defaults to `15L`.
`n_batches`	Integer. Number of batches. Defaults to `1L`.
`marker_genes`	Any. A nested list that indicates which gene indices are markers for which cell. Defaults to `list(cell_type_1 = list(marker_genes = 0:2), cell_type_2 = list(marker_genes = 3:5), cell_type_3 = list(marker_genes = 6:8))`.
`isotype_controls`	Integer vector. The columns that defines the isotype controls. (0-indexed!) Defaults to `12:14`.
`batch_effect_strength`	String. The strength of the batch effect to add. One of `c("strong", "medium", "weak")`. Defaults to `"strong"`.

Value

A named list with the following elements:

- n_cells - Integer. Number of cells. Defaults to `1000L`.
- n_proteins - Integer. Number of proteins Defaults to `15L`.

- marker_genes - Any. A nested list that indicates which gene indices are markers for which cell. Defaults to `list(cell_type_1 = list(marker_genes = 0:2), cell_type_2 = list(marker_genes = 3:5), cell_type_3 = list(marker_genes = 6:8))`.
- n_batches - Integer. Number of batches. Defaults to `1L`.
- isotype_controls - Integer vector. The columns that defines the isotype controls. (0-indexed!) Defaults to `12:14`.
- batch_effect_strength - String. The strength of the batch effect to add. One of `c("strong", "medium", "weak")`. Defaults to `"strong"`.

`params_sc_synthetic_dialogue`
Default parameters for generation of synthetic DIALOGUE data

Description

Shapes the fixture with a planted multicellular programme that `generate_dialogue_test_data()` builds. The defaults give 14 samples x 25 cells x 3 cell types = 1050 cells over 400 genes.

Usage

```
params_sc_synthetic_dialogue(
  n_samples = 14L,
  cells_per_sample = 25L,
  n_cell_types = 3L,
  n_features = 8L,
  n_sample_features = 5L,
  n_genes = 400L,
  n_planted = 8L
)
```

Arguments

`n_samples`	Integer. Samples the experiment spans. DIALOGUE needs at least 5. Defaults to `14L`.
`cells_per_sample`	Integer. Cells per sample per cell type. Defaults to `25L`.
`n_cell_types`	Integer. Number of cell types. Must be at least 2. Defaults to `3L`.
`n_features`	Integer. Feature columns per cell type. Must be at least 2. Defaults to `8L`.
`n_sample_features`	Integer. Feature columns carrying a per-sample component. The first of those is the shared programme, the rest are cell-type-specific nuisance; anything past this count is pure noise and exists so the ANOVA filter has something to reject. Defaults to `5L`.
`n_genes`	Integer. Number of genes. Defaults to `400L`.
`n_planted`	Integer. Planted genes per cell type. The blocks are contiguous, so `n_planted * n_cell_types` has to fit into `n_genes`. Defaults to `8L`.

Details

`n_genes` defaults higher here than on the Rust side, which plants the same structure into 90. R runs the counts through the normal ingestion path, and on a small panel the planted genes are a large enough share of the library that log-normalisation divides the signal back out: at 90 genes the library size tracks the programme at an r of 0.75 and background genes pick up a spurious correlation of their own. At 400 the planted block is a few percent of the library and the contrast survives. The Rust tests feed the planted layer straight in, so they never meet this.

Value

A named list with the following elements:

- n_samples - Integer. Samples the experiment spans. DIALOGUE needs at least 5. Defaults to `14L`.
- cells_per_sample - Integer. Cells per sample per cell type. Defaults to `25L`.
- n_cell_types - Integer. Number of cell types. Must be at least 2. Defaults to `3L`.
- n_features - Integer. Feature columns per cell type. Must be at least 2. Defaults to `8L`.
- n_sample_features - Integer. Feature columns carrying a per-sample component. The first of those is the shared programme, the rest are cell-type-specific nuisance; anything past this count is pure noise and exists so the ANOVA filter has something to reject. Defaults to `5L`.
- n_genes - Integer. Number of genes. Defaults to `400L`.
- n_planted - Integer. Planted genes per cell type. The blocks are contiguous, so `n_planted * n_cell_types` has to fit into `n_genes`. Defaults to `8L`.

References

Jerby-Arnon & Regev, Nature Biotechnology, 2022

`params_sc_vision` *Wrapper function for parameters for VISION with auto-correlation*

Description

Wrapper function for parameters for VISION with auto-correlation

Usage

```
params_sc_vision(n_perm = 500L, n_cluster = 5L, knn = list())
```

Arguments

n_perm	Integer. Number of random gene sets to generate per cluster. Defaults to 500L.
n_cluster	Integer. Number of clusters for the random gene set clustering generation. Defaults to 5L.
knn	List. Optional overrides for kNN parameters. See params_knn_defaults() for available parameters: k, knn_method, ann_dist, search_budget, n_trees, delta, diversify_prob, ef_budget, extract_knn, m, ef_construction, ef_search, n_list and n_probe. See params_knn_defaults() for the available elements. Defaults to list().

Value

A named list with the following elements:

- n_perm - Integer. Number of random gene sets to generate per cluster. Defaults to 500L.
- n_cluster - Integer. Number of clusters for the random gene set clustering generation. Defaults to 5L.
- The elements of params_knn_defaults(), overridden by knn, spliced in at this position.

---

params_sc_wnn *Wrapper function for WNN parameters*

Description

Wrapper function for WNN parameters

Usage

```
params_sc_wnn(
  k_nn = 20L,
  knn_range = 200L,
  sigma_method = c("snn_farthest", "sigma_idx"),
  sigma_idx = 19L,
  snn_type = c("full_connection", "limited"),
  s_nn = 20L,
  sd_scale = 1,
  kernel_power = 1,
  cross_const = 1e-04,
  sigma_floor = 1e-08,
  knn = list()
)
```

Arguments

k_nn	Integer. Final number of multimodal neighbours per cell. Defaults to 20L.
knn_range	Integer. Candidate pool size per modality. Each cell's kNN input must contain at least this many neighbours. Defaults to 200L.
sigma_method	String. Bandwidth method. One of c("snn_farthest", "sigma_idx"). Defaults to "snn_farthest".
sigma_idx	Integer. "sigma_idx" only: 0-based kNN index for bandwidth.e. k_nn - 1). Defaults to 19L.
snn_type	String. sNN type. The limited version only considers edges that exist in the kNN. One of c("full_connection", "limited"). Defaults to "full_connection".
s_nn	Integer. "snn_farthest" only: kNN size used to build the SNN graph. Defaults to 20L.
sd_scale	Numeric. Multiplier on sigma. Defaults to 1.0.
kernel_power	Numeric. Kernel exponent power. Defaults to 1.0.
cross_const	Numeric. Cross-modality kernel stabiliser. Defaults to 1e-04.
sigma_floor	Numeric. Minimum sigma value (avoids division by zero). Defaults to 1e-08.
knn	List. Optional overrides for kNN parameters. See params_knn_defaults() for available parameters: k, knn_method, ann_dist, search_budget, n_trees, delta, diversify_prob, ef_budget, extract_knn, m, ef_construction, ef_search, n_list and n_probe. See params_knn_defaults() for the available elements. Defaults to list().

Value

A named list with the following elements:

- k_nn - Integer. Final number of multimodal neighbours per cell. Defaults to 20L.
- knn_range - Integer. Candidate pool size per modality. Each cell's kNN input must contain at least this many neighbours. Defaults to 200L.
- sigma_method - String. Bandwidth method. One of c("snn_farthest", "sigma_idx"). Defaults to "snn_farthest".
- sigma_idx - Integer. "sigma_idx" only: 0-based kNN index for bandwidth.e. k_nn - 1). Defaults to 19L.
- snn_type - String. sNN type. The limited version only considers edges that exist in the kNN. One of c("full_connection", "limited"). Defaults to "full_connection".
- s_nn - Integer. "snn_farthest" only: kNN size used to build the SNN graph. Defaults to 20L.
- sd_scale - Numeric. Multiplier on sigma. Defaults to 1.0.
- kernel_power - Numeric. Kernel exponent power. Defaults to 1.0.
- cross_const - Numeric. Cross-modality kernel stabiliser. Defaults to 1e-04.

- sigma_floor - Numeric. Minimum sigma value (avoids division by zero). Defaults to `1e-08`.
- The elements of `params_knn_defaults()`, overridden by `knn`, spliced in at this position.

`params_scdblfinder` *Wrapper function for scDblFinder doublet detection parameters*

Description

Constructor for the scDblFinder parameters. This method combines cluster-aware doublet simulation with a gradient-boosted classifier trained on engineered features.

Usage

```
params_scdblfinder(
  n_genes = 1352L,
  doublet_ratio = 1,
  heterotypic_bias = 1,
  cluster_resolution = 1,
  cluster_iters = 10L,
  fast_cluster = FALSE,
  n_iterations = 3L,
  gbm_n_trees = 200L,
  max_depth = 4L,
  learning_rate = 0.3,
  min_samples_leaf = 20L,
  subsample_rate = 0.75,
  cv_folds = 5L,
  cv_early_stop = 2L,
  se_fraction = 1,
  include_pcs = 19L,
  expected_doublet_rate = NULL,
  cxds_genes = NULL,
  manual_threshold = NULL,
  normalisation = list(mean_center = TRUE),
  pca = list(),
  knn = list(k = 0L),
  fast_cluster_params = list()
)
```

Arguments

`n_genes`	Integer. Number of top-expressed genes to use as features. Defaults to `1352L`.
`doublet_ratio`	Numeric. Ratio of simulated doublets to observed cells. Defaults to `1.0`.

`heterotypic_bias`
: Numeric. Fraction of simulated pairs forced to come from different clusters (0-1). Defaults to `1.0`.

`cluster_resolution`
: Numeric. Resolution for the initial Louvain clustering. Defaults to `1.0`.

`cluster_iters`
: Integer. Number of Louvain iterations per clustering step. Defaults to `10L`.

`fast_cluster`
: Boolean. Shall fast Louvain clustering be applied, i.e., k-means clustering and use the centroids for kNN graph generation and Louvain clustering with then backpropagating the membership based on centroid proximity. Defaults to `FALSE`.

`n_iterations`
: Integer. Number of refinement iterations. Typically 2-3. Defaults to `3L`.

`gbm_n_trees`
: Integer. Maximum number of boosting rounds for the GBM classifier. Defaults to `200L`.

`max_depth`
: Integer. Maximum tree depth. Shallow trees (3-5) work best. Defaults to `4L`.

`learning_rate`
: Numeric. Shrinkage applied to each tree. Defaults to `0.3`.

`min_samples_leaf`
: Integer. Minimum training samples per leaf. Defaults to `20L`.

`subsample_rate`
: Numeric. Fraction of samples used per tree. Defaults to `0.75`.

`cv_folds`
: Integer. Number of cross-validation folds for boosting round selection. Defaults to `5L`.

`cv_early_stop`
: Integer. Early stopping patience per CV fold. Defaults to `2L`.

`se_fraction`
: Numeric. Multiplier on the standard error for the SE rule used in round selection. Defaults to `1.0`.

`include_pcs`
: Any. Number of leading principal components to include as classifier features. Defaults to `19L`.

`expected_doublet_rate`
: Numeric or `NULL`. Expected doublet rate as a percentage. If not provided, will be calculated internally. Defaults to `NULL`.

`cxds_genes`
: Integer or `NULL`. Number of CXDS genes to consider. If not provided, defaults to `500L`. Defaults to `NULL`.

`manual_threshold`
: Numeric or `NULL`. Manual score threshold. If `NULL` (default), expected-rate thresholding is used. Defaults to `NULL`.

`normalisation`
: List. Optional overrides for normalisation parameters. See `params_norm_doublets_defaults()`. See `params_norm_doublets_defaults()` for the available elements. Defaults to `list(mean_center = TRUE)`.

`pca`
: List. Optional overrides for PCA parameters. See `params_pca_defaults()`. See `params_pca_defaults()` for the available elements. Defaults to `list()`.

`knn`
: List. Optional overrides for kNN parameters. See `params_knn_defaults()`. NNDescent works better for the larger k-values often used here. See `params_knn_defaults()` for the available elements. Defaults to `list(k = 0L)`.

`fast_cluster_params`
: List. Optional overrides for the fast clustering parameters. Only relevant if `fast_cluster = TRUE`. See `params_fast_cluster_default()` for available parameters: `km_type`, `n_centroids`, `kmeans_iters` and `batch_size`. See `params_fast_cluster_default()` for the available elements. Defaults to `list()`.

Value

A named list with the following elements:

- The elements of `params_norm_doublets_defaults()`, overridden by `normalisation`, spliced in at this position.
- The elements of `params_pca_defaults()`, overridden by `pca`, spliced in at this position.
- The elements of `params_knn_defaults()`, overridden by `knn`, spliced in at this position.
- The elements of `params_fast_cluster_default()`, overridden by `fast_cluster_params`, spliced in at this position.
- n_genes - Integer. Number of top-expressed genes to use as features. Defaults to `1352L`.
- doublet_ratio - Numeric. Ratio of simulated doublets to observed cells. Defaults to `1.0`.
- heterotypic_bias - Numeric. Fraction of simulated pairs forced to come from different clusters (0-1). Defaults to `1.0`.
- cluster_resolution - Numeric. Resolution for the initial Louvain clustering. Defaults to `1.0`.
- cluster_iters - Integer. Number of Louvain iterations per clustering step. Defaults to `10L`.
- fast_cluster - Boolean. Shall fast Louvain clustering be applied, i.e., k-means clustering and use the centroids for kNN graph generation and Louvain clustering with then backpropagating the membership based on centroid proximity. Defaults to `FALSE`.
- n_iterations - Integer. Number of refinement iterations. Typically 2-3. Defaults to `3L`.
- gbm_n_trees - Integer. Maximum number of boosting rounds for the GBM classifier. Defaults to `200L`.
- max_depth - Integer. Maximum tree depth. Shallow trees (3-5) work best. Defaults to `4L`.
- learning_rate - Numeric. Shrinkage applied to each tree. Defaults to `0.3`.
- min_samples_leaf - Integer. Minimum training samples per leaf. Defaults to `20L`.
- subsample_rate - Numeric. Fraction of samples used per tree. Defaults to `0.75`.
- cv_folds - Integer. Number of cross-validation folds for boosting round selection. Defaults to `5L`.
- cv_early_stop - Integer. Early stopping patience per CV fold. Defaults to `2L`.

- se_fraction - Numeric. Multiplier on the standard error for the SE rule used in round selection. Defaults to `1.0`.
- include_pcs - Any. Number of leading principal components to include as classifier features. Defaults to `19L`.
- expected_doublet_rate - Numeric or `NULL`. Expected doublet rate as a percentage. If not provided, will be calculated internally. Defaults to `NULL`.
- manual_threshold - Numeric or `NULL`. Manual score threshold. If `NULL` (default), expected-rate thresholding is used. Defaults to `NULL`.
- cxds_genes - Integer or `NULL`. Number of CXDS genes to consider. If not provided, defaults to `500L`. Defaults to `NULL`.

`params_scenic` *Constructor for SCENIC parameters*

Description

Constructor for SCENIC parameters

Usage

```
params_scenic(
  min_counts = 50L,
  min_cells = 0.03,
  learner_type = c("randomforest", "extratrees", "grnboost2"),
  gene_batch_strategy = c("correlated", "random"),
  gene_batch_size = NULL,
  n_pcs = 50L,
  n_subsample = 100000L,
  learner_params = list()
)
```

Arguments

`min_counts`	Integer. Minimum total counts a gene needs to be included in the analysis. Defaults to `50L`.
`min_cells`	Numeric. Minimum proportion of cells (between 0 and 1) that must express a gene for it to be considered. Defaults to `0.03`.
`learner_type`	String. Regression learner to use. One of `c("randomforest", "extratrees", "grnboost2")`. Defaults to `"randomforest"`.
`gene_batch_strategy`	String. Strategy for grouping target genes into batches. Only used for `"randomforest"` and `"extratrees"` learners; ignored for `"grnboost2"`. One of `c("correlated", "random")`. Defaults to `"correlated"`.

`gene_batch_size`
: Integer or `NULL`. Number of genes per batch. If `NULL` (default), the batch size is determined automatically. Ignored for `"grnboost2"`. Defaults to `NULL`.

`n_pcs`
: Integer. Number of PCs to use for the correlated gene batch strategy. Defaults to `50L`.

`n_subsample`
: Integer. Cell subsampling threshold for the correlated gene batch strategy. If the number of cells meets or exceeds this value, `n_subsample` cells are randomly selected prior to running randomised SVD. Defaults to `100000L`.

`learner_params`
: List. Optional overrides for the regression learner parameters. For `"randomforest"`, see `params_scenic_random_forest_defaults()`. For `"extratrees"`, see `params_scenic_extra_trees_defaults()`. For `"grnboost2"`, see `params_scenic_gradient_b`
Defaults to `list()`.

Value

A named list with the following elements:

- min_counts - Integer. Minimum total counts a gene needs to be included in the analysis. Defaults to `50L`.
- min_cells - Numeric. Minimum proportion of cells (between 0 and 1) that must express a gene for it to be considered. Defaults to `0.03`.
- learner_type - String. Regression learner to use. One of `c("randomforest", "extratrees", "grnboost2")`. Defaults to `"randomforest"`.
- gene_batch_strategy - String. Strategy for grouping target genes into batches. Only used for `"randomforest"` and `"extratrees"` learners; ignored for `"grnboost2"`. One of `c("correlated", "random")`. Defaults to `"correlated"`.
- gene_batch_size - Integer or `NULL`. Number of genes per batch. If `NULL` (default), the batch size is determined automatically. Ignored for `"grnboost2"`. Defaults to `NULL`.
- n_pcs - Integer. Number of PCs to use for the correlated gene batch strategy. Defaults to `50L`.
- n_subsample - Integer. Cell subsampling threshold for the correlated gene batch strategy. If the number of cells meets or exceeds this value, `n_subsample` cells are randomly selected prior to running randomised SVD. Defaults to `100000L`.
- The elements of the base list, overridden by `learner_params`, spliced in at this position.

`params_scenic_binarise` *Wrapper function for parameters for the SCENIC binarisation*

Description

Each regulon gets its own threshold. A two-component Gaussian mixture is fitted and compared against a single Gaussian by BIC; if the mixture wins, the threshold is the kernel density minimum between the two component means, otherwise it falls back to `mean + 2 * sd`. This follows pySCENIC. AUCell fits six candidates and then lets the density trough override all of them whenever one exists, so the two land in much the same place. Turn `bw_adjust` up if shallow wobbles in the density are being picked up as troughs. AUCell effectively runs at `2`.

Usage

```
params_scenic_binarise(bw_adjust = 1, n_grid = 512L, n_bins = 512L)
```

Arguments

`bw_adjust`	Numeric. Multiplier on the Silverman bandwidth of the kernel density estimate. Higher values smooth more. Defaults to `1.0`.
`n_grid`	Integer. Number of points at which the density is evaluated between the two component means. Defaults to `512L`.
`n_bins`	Integer. Number of histogram bins used to approximate the density. Defaults to `512L`.

Value

A named list with the following elements:

- bw_adjust - Numeric. Multiplier on the Silverman bandwidth of the kernel density estimate. Higher values smooth more. Defaults to `1.0`.
- n_grid - Integer. Number of points at which the density is evaluated between the two component means. Defaults to `512L`.
- n_bins - Integer. Number of histogram bins used to approximate the density. Defaults to `512L`.

References

Aibar, et al., Nat Methods, 2017

`params_scenic_extra_trees_defaults` *Default parameters for the SCENIC ExtraTrees regression learner*

Description

Default parameters for the SCENIC ExtraTrees regression learner

Usage

```
params_scenic_extra_trees_defaults()
```

Value

A named list with the following elements:

- n_trees - Any. TODO Defaults to `500L`.
- min_samples_leaf - Any. TODO Defaults to `50L`.
- n_features_split - Any. TODO Defaults to `0L`.
- n_thresholds - Any. TODO Defaults to `1L`.
- max_depth - Any. TODO Defaults to `8L`.
- subsample_frac - Any. TODO Defaults to `NULL`.

---

params_scenic_gradient_boosting_defaults

Default parameters for the SCENIC GradientBoosting (GRNBoost2) regression learner

---

Description

Default parameters for the SCENIC GradientBoosting (GRNBoost2) regression learner

Usage

```
params_scenic_gradient_boosting_defaults()
```

Value

A named list with the following elements:

- n_trees_max - Any. TODO Defaults to `1000L`.
- learning_rate - Any. TODO Defaults to `0.01`.
- max_depth - Any. TODO Defaults to `3L`.
- min_samples_leaf - Any. TODO Defaults to `50L`.
- early_stop_window - Any. TODO Defaults to `25L`.
- subsample_rate - Any. TODO Defaults to `0.9`.
- n_features_split - Any. TODO Defaults to `0L`.

params_scenic_random_forest_defaults

Default parameters for the SCENIC RandomForest regression learner

Description

Default parameters for the SCENIC RandomForest regression learner

Usage

```
params_scenic_random_forest_defaults()
```

Value

A named list with the following elements:

- n_trees - Any. TODO Defaults to `250L`.
- min_samples_leaf - Any. TODO Defaults to `50L`.
- n_features_split - Any. TODO Defaults to `0L`.
- subsample_rate - Any. TODO Defaults to `0.632`.
- bootstrap - Any. TODO Defaults to `FALSE`.
- max_depth - Any. TODO Defaults to `8L`.
- subsample_frac - Any. TODO Defaults to `NULL`.

params_scrublet

Wrapper function for Scrublet doublet detection parameters

Description

Constructor for the various Scrublet parameters. In this case, the default for the kNN graph generation was set to `"hnsw"` as this algorithm showed the best performance in different empirical benchmarks.

Usage

```
params_scrublet(
  sim_doublet_ratio = 1.5,
  expected_doublet_rate = 0.1,
  stdev_doublet_rate = 0.02,
  n_bins_histogram = 100L,
  manual_threshold = NULL,
  normalisation = list(),
  hvg = list(),
  pca = list(),
  knn = list(k = 0L)
)
```

Arguments

`sim_doublet_ratio`
: Numeric. Number of doublets to simulate relative to the number of observed cells. For example, 2.0 simulates twice as many doublets as there are cells. Defaults to `1.5`.

`expected_doublet_rate`
: Numeric. Expected doublet rate for the experiment, typically 0.05-0.10 depending on cell loading. Must be between 0 and 1. Defaults to `0.1`.

`stdev_doublet_rate`
: Numeric. Uncertainty in the expected doublet rate. Defaults to `0.02`.

`n_bins_histogram`
: Integer. Number of bins for histogram-based automatic threshold detection. Typically 50-100. Defaults to `100L`.

`manual_threshold`
: Numeric or `NULL`. Manual doublet score threshold. If `NULL` (default), threshold is automatically detected from simulated doublet score distribution. Defaults to `NULL`.

`normalisation`
: List. Optional overrides for normalisation parameters. See `params_norm_doublets_defaults()` for available parameters: `log_transform`, `mean_center`, `normalise_variance`, `target_size`. See `params_norm_doublets_defaults()` for the available elements. Defaults to `list()`.

`hvg`
: List. Optional overrides for highly variable gene selection parameters. See `params_hvg_defaults()` for available parameters: `min_gene_var_pctl`, `hvg_method`, `loess_span`, `clip_max`. See `params_hvg_defaults()` for the available elements. Defaults to `list()`.

`pca`
: List. Optional overrides for PCA parameters. See `params_pca_defaults()` for available parameters: `no_pcs` and `svd_solver`. See `params_pca_defaults()` for the available elements. Defaults to `list()`.

`knn`
: List. Optional overrides for kNN parameters. See `params_knn_defaults()` for available parameters: `k`, `knn_method`, `ann_dist`, `search_budget`, `n_trees`, `delta`, `diversify_prob`, `ef_budget`, `extract_knn`, `m`, `ef_construction`, `ef_search`, `n_list` and `n_probe`. Note: this function defaults to `k = 0L` (automatic neighbour detection). See `params_knn_defaults()` for the available elements. Defaults to `list(k = 0L)`.

Value

A named list with the following elements:

- The elements of `params_norm_doublets_defaults()`, overridden by `normalisation`, spliced in at this position.
- The elements of `params_hvg_defaults()`, overridden by `hvg`, spliced in at this position.
- The elements of `params_pca_defaults()`, overridden by `pca`, spliced in at this position.
- The elements of `params_knn_defaults()`, overridden by `knn`, spliced in at this position.

- sim_doublet_ratio - Numeric. Number of doublets to simulate relative to the number of observed cells. For example, 2.0 simulates twice as many doublets as there are cells. Defaults to `1.5`.
- expected_doublet_rate - Numeric. Expected doublet rate for the experiment, typically 0.05-0.10 depending on cell loading. Must be between 0 and 1. Defaults to `0.1`.
- stdev_doublet_rate - Numeric. Uncertainty in the expected doublet rate. Defaults to `0.02`.
- n_bins_histogram - Integer. Number of bins for histogram-based automatic threshold detection. Typically 50-100. Defaults to `100L`.
- manual_threshold - Numeric or `NULL`. Manual doublet score threshold. If `NULL` (default), threshold is automatically detected from simulated doublet score distribution. Defaults to `NULL`.

`params_sctype_cells` *Parameters for the per-cell ScType assignment*

Description

Controls the per-cell path of `assign_sc_type()`: how the raw ScType scores are rescaled, how hard the scores get smoothed over the sNN graph, and where the cut-offs for an Unknown call and for a mixed cluster sit.

Usage

```
params_sctype_cells(
  alpha = 0.5,
  iterations = 2L,
  tolerance = 1e-04,
  calibration = c("none", "column_z"),
  score_floor = 0.25,
  purity_threshold = 0.9
)
```

Arguments

`alpha`	Numeric. Numeric in `[0, 1]`. Self-retention during smoothing. Each iteration computes `alpha * original + (1 - alpha) * neighbour_average`. Defaults to `0.5`.
`iterations`	Integer. Integer >= 0. Number of smoothing iterations. `0` disables smoothing. Defaults to `2L`.
`tolerance`	Numeric. Numeric > 0. Convergence tolerance for the smoothing. Defaults to `1e-04`.

`calibration` String. `"column_z"` standardises each cell type's score column across cells, which removes the bias towards cell types whose marker sets happen to produce larger scores. One of `c("none", "column_z")`. Defaults to `"none"`.

`score_floor` Numeric. Numeric >= 0. Minimum score for a cell to get a call instead of `NA`. Defaults to `0.25`.

`purity_threshold` Numeric. Numeric in `[0, 1]`. Cluster purity above which the hybrid assignment keeps the cluster-level call. Defaults to `0.9`.

Value

A named list with the following elements:

- alpha - Numeric. Numeric in `[0, 1]`. Self-retention during smoothing. Each iteration computes `alpha * original + (1 - alpha) * neighbour_average`. Defaults to `0.5`.
- iterations - Integer. Integer >= 0. Number of smoothing iterations. `0` disables smoothing. Defaults to `2L`.
- tolerance - Numeric. Numeric > 0. Convergence tolerance for the smoothing. Defaults to `1e-04`.
- calibration - String. `"column_z"` standardises each cell type's score column across cells, which removes the bias towards cell types whose marker sets happen to produce larger scores. One of `c("none", "column_z")`. Defaults to `"none"`.
- score_floor - Numeric. Numeric >= 0. Minimum score for a cell to get a call instead of `NA`. Defaults to `0.25`.
- purity_threshold - Numeric. Numeric in `[0, 1]`. Cluster purity above which the hybrid assignment keeps the cluster-level call. Defaults to `0.9`.

References

Zhou et al., NIPS, 2004.

---

params_snf *Wrapper function to generate SNF parameters*

Description

Wrapper function to generate SNF parameters

Usage

```
params_snf(
  k = 20L,
  t = 20L,
  mu = 0.5,
  alpha = 1,
  normalise = TRUE,
  distance_metric = c("euclidean", "manhattan", "canberra", "cosine")
)
```

Arguments

k	Integer. Number of neighbours to consider. Defaults to 20L.
t	Integer. Number of iterations for the SNF algorithm. Defaults to 20L.
mu	Numeric. Normalisation factor for the Gaussian kernel width. Defaults to 0.5.
alpha	Numeric. Normalisation parameter controlling the fusion strength. Defaults to 1.0.
normalise	Boolean. Shall continuous values be Z-scored. Defaults to TRUE.
distance_metric	String. Which distance metric to use for the continuous calculations. In case of pure categorical, Hamming will be used, for mixed data types Gower distance is used. One of c("euclidean", "manhattan", "canberra", "cosine"). Defaults to "euclidean".

Value

A named list with the following elements:

- k - Integer. Number of neighbours to consider. Defaults to 20L.
- t - Integer. Number of iterations for the SNF algorithm. Defaults to 20L.
- mu - Numeric. Normalisation factor for the Gaussian kernel width. Defaults to 0.5.
- alpha - Numeric. Normalisation parameter controlling the fusion strength. Defaults to 1.0.
- distance_metric - String. Which distance metric to use for the continuous calculations. In case of pure categorical, Hamming will be used, for mixed data types Gower distance is used. One of c("euclidean", "manhattan", "canberra", "cosine"). Defaults to "euclidean".
- normalise - Boolean. Shall continuous values be Z-scored. Defaults to TRUE.

---

params_ssgsea *Wrapper function to generate ssGSEA parameters*

Description

Wrapper function to generate ssGSEA parameters

Usage

```
params_ssgsea(alpha = 0.25, min_size = 5L, max_size = 500L, normalise = TRUE)
```

Arguments

alpha	Numeric. The exponent defining the weight of the tail in the random walk performed by ssGSEA. Defaults to 0.25.
min_size	Integer. Minimum number of genes per gene set. Defaults to 5L.
max_size	Integer. Maximum number of genes per gene set. Defaults to 500L.
normalise	Boolean. Shall the scores be normalised. Defaults to TRUE.

Value

A named list with the following elements:

- alpha - Numeric. The exponent defining the weight of the tail in the random walk performed by ssGSEA. Defaults to `0.25`.
- min_size - Integer. Minimum number of genes per gene set. Defaults to `5L`.
- max_size - Integer. Maximum number of genes per gene set. Defaults to `500L`.
- normalise - Boolean. Shall the scores be normalised. Defaults to `TRUE`.

`params_symphony_map` *Default parameters for Symphony query mapping*

Description

Default parameters for Symphony query mapping

Usage

```
params_symphony_map(sigma = 0.1, lambda = 1)
```

Arguments

`sigma`	Numeric. Soft-clustering fuzziness for query -> reference centroid assignment. Symphony R default is 0.1. Defaults to `0.1`.
`lambda`	Numeric. Ridge penalty on batch coefficients. Symphony R hardcodes 1.0. Defaults to `1.0`.

Value

A named list with the following elements:

- sigma - Numeric. Soft-clustering fuzziness for query -> reference centroid assignment. Symphony R default is 0.1. Defaults to `0.1`.
- lambda - Numeric. Ridge penalty on batch coefficients. Symphony R hardcodes 1.0. Defaults to `1.0`.

`params_synthetic_bulk_rnaseq` *Wrapper function to generate synthetic bulk RNAseq parameters*

Description

Parameters for `synthetic_bulk_cor_matrix()`. Counts come from a negative binomial with a mean-dispersion trend; co-expression modules are planted by putting each module's genes on a shared latent factor. The `generator` picks how loadings and factors are drawn, which is what makes a given dataset a fair or unfair benchmark for a given method:

- `"hub_modular"` - LogNormal loadings on a Normal factor. Some genes end up far more connected than others, so this is the WGCNA-style default.
- `"modular"` - Beta(5, 2) loadings on a Normal factor. Homogeneous within-module correlation and no hubs.
- `"non_negative_factor"` - LogNormal loadings on a Gamma factor. The activity matrix is non-negative by construction, so NMF has a ground truth it can actually reach.
- `"non_gaussian_factor"` - LogNormal loadings on a Laplace factor. Non-Gaussian sources satisfy ICA identifiability.

Usage

```
params_synthetic_bulk_rnaseq(
  num_samples = 100L,
  num_genes = 1000L,
  module_sizes = c(100L, 100L, 100L),
  generator = c("hub_modular", "modular", "non_negative_factor", "non_gaussian_factor"),
  seed = 123L,
  mean_exp_gamma_shape = 5,
  mean_exp_gamma_scale = 10,
  disp_intercept = 0.2,
  disp_slope = 0.3,
  noise_std = 0.1,
  factor_std = 0.5,
  factor_shape = 2,
  factor_scale = 0.3,
  loading_mu = 0,
  loading_sigma = 0.7,
  hub_percentile = 0.1
)
```

Arguments

`num_samples`	Integer. Number of samples (columns) to simulate. Defaults to `100L`.
`num_genes`	Integer. Number of genes (rows) to simulate. Defaults to `1000L`.

`module_sizes` Integer vector. Sizes of the co-expression modules. The sum must be smaller or equal to `num_genes`. Genes are assigned in contiguous blocks from the first gene onwards, any remainder is background. Use `integer(0)` for no modules. Must be an integer vector, see the note below. Defaults to `c(100L, 100L, 100L)`.

`generator` String. Which topology and distribution family to plant., see the description. One of `c("hub_modular", "modular", "non_negative_factor", "non_gaussian_factor")`. Defaults to `"hub_modular"`.

`seed` Integer. Seed for reproducibility purposes. Defaults to `123L`.

`mean_exp_gamma_shape`
Numeric. Shape and scale of the Gamma the per-gene mean expression is drawn from. Defaults to `5.0`.

`mean_exp_gamma_scale`
Numeric. Shape and scale of the Gamma the per-gene mean expression is drawn from. Defaults to `10.0`.

`disp_intercept`
Numeric. Intercept and slope of the negative binomial dispersion trend `disp = 1 / (a + b * mean)`. This is what gives you heteroskedasticity: lowly expressed genes show higher variance. Defaults to `0.2`.

`disp_slope` Numeric. Intercept and slope of the negative binomial dispersion trend `disp = 1 / (a + b * mean)`. This is what gives you heteroskedasticity: lowly expressed genes show higher variance. Defaults to `0.3`.

`noise_std` Numeric. Per-gene per-sample noise standard deviation on the latent log-signal. Smaller values track the module factor more tightly and give stronger within-module correlation. Defaults to `0.1`.

`factor_std` Numeric. Standard deviation of the Normal factor. Only used by `"hub_modular"` and `"modular"`; the other two generators draw their factor from `factor_shape`/`factor_scale` instead. Defaults to `0.5`.

`factor_shape` Numeric. Shape and scale of the Gamma factor for `"non_negative_factor"`. `factor_scale` doubles as the Laplace scale for `"non_gaussian_factor"`. Defaults to `2.0`.

`factor_scale` Numeric. Shape and scale of the Gamma factor for `"non_negative_factor"`. `factor_scale` doubles as the Laplace scale for `"non_gaussian_factor"`. Defaults to `0.3`.

`loading_mu` Numeric. Location and scale of the LogNormal the loadings are drawn from. Unused by `"modular"`, which draws Beta(5, 2). Defaults to `0.0`.

`loading_sigma` Numeric. Location and scale of the LogNormal the loadings are drawn from. Unused by `"modular"`, which draws Beta(5, 2). Defaults to `0.7`.

`hub_percentile`
Numeric. Top fraction of module genes flagged as hubs by loading rank. Must be in `(0, 1]`. Defaults to `0.1`.

Details

`noise_std` and `factor_std` default to `0.1` and `0.5`, not to the `bixverse-rs` values of `0.3` and `0.3`. At the crate defaults the `"modular"` generator plants modules too weakly to

detect at 1000 genes by 100 samples: the within-module minus cross-module mean absolute Spearman gap comes out around `0.06`, against `0.17` to `0.23` for the other three. The values here put all four generators in the `0.30` to `0.39` band, so a comparison across generators reflects the method rather than the signal strength it happened to be handed. Pass the crate values explicitly if you want a harder problem.

Value

A named list with the following elements:

- num_samples - Integer. Number of samples (columns) to simulate. Defaults to `100L`.
- num_genes - Integer. Number of genes (rows) to simulate. Defaults to `1000L`.
- module_sizes - Integer vector. Sizes of the co-expression modules. The sum must be smaller or equal to `num_genes`. Genes are assigned in contiguous blocks from the first gene onwards, any remainder is background. Use `integer(0)` for no modules. Must be an integer vector, see the note below. Defaults to `c(100L, 100L, 100L)`.
- generator - String. Which topology and distribution family to plant., see the description. One of `c("hub_modular", "modular", "non_negative_factor", "non_gaussian_factor")`. Defaults to `"hub_modular"`.
- seed - Integer. Seed for reproducibility purposes. Defaults to `123L`.
- mean_exp_gamma_shape - Numeric. Shape and scale of the Gamma the per-gene mean expression is drawn from. Defaults to `5.0`.
- mean_exp_gamma_scale - Numeric. Shape and scale of the Gamma the per-gene mean expression is drawn from. Defaults to `10.0`.
- disp_intercept - Numeric. Intercept and slope of the negative binomial dispersion trend `disp = 1 / (a + b * mean)`. This is what gives you heteroskedasticity: lowly expressed genes show higher variance. Defaults to `0.2`.
- disp_slope - Numeric. Intercept and slope of the negative binomial dispersion trend `disp = 1 / (a + b * mean)`. This is what gives you heteroskedasticity: lowly expressed genes show higher variance. Defaults to `0.3`.
- noise_std - Numeric. Per-gene per-sample noise standard deviation on the latent log-signal. Smaller values track the module factor more tightly and give stronger within-module correlation. Defaults to `0.1`.
- factor_std - Numeric. Standard deviation of the Normal factor. Only used by `"hub_modular"` and `"modular"`; the other two generators draw their factor from `factor_shape`/`factor_scale` instead. Defaults to `0.5`.
- factor_shape - Numeric. Shape and scale of the Gamma factor for `"non_negative_factor"`. `factor_scale` doubles as the Laplace scale for `"non_gaussian_factor"`. Defaults to `2.0`.
- factor_scale - Numeric. Shape and scale of the Gamma factor for `"non_negative_factor"`. `factor_scale` doubles as the Laplace scale for `"non_gaussian_factor"`. Defaults to `0.3`.
- loading_mu - Numeric. Location and scale of the LogNormal the loadings are drawn from. Unused by `"modular"`, which draws Beta(5, 2). Defaults to `0.0`.

- loading_sigma - Numeric. Location and scale of the LogNormal the loadings are drawn from. Unused by `"modular"`, which draws Beta(5, 2). Defaults to `0.7`.
- hub_percentile - Numeric. Top fraction of module genes flagged as hubs by loading rank. Must be in `(0, 1]`. Defaults to `0.1`.

References

Zhang & Horvath, Stat Appl Genet Mol Biol, 2005

`per_cell_qc_outlier` *Use MAD outlier detection on per-cell QC metrics*

Description

Use MAD outlier detection on per-cell QC metrics

Usage

```
per_cell_qc_outlier(
  metric,
  threshold = 3,
  direction = c("twosided", "below", "above")
)
```

Arguments

`metric`	Numerical vector. The QC metric to check for.
`threshold`	Numeric. How many MADs in either direction to consider for outlier detection.
`direction`	String. One of `c("twosided", "below", "above")`. Which directionality to consider

Value

A list with:

- outlier - Boolean vector indicating which cell is an outlier
- metrics - The applied thresholds.

Examples

```
# one badly undersequenced cell, flagged on the lower tail only
set.seed(42L)
lib_size <- c(rnorm(99, 1000, 100), 50)
res <- per_cell_qc_outlier(lib_size, direction = "below")
sum(res$outlier)
res$metrics
```

`permute_seed_nodes` *Generate permuation scores for the diffusion*

Description

This function generate node-degree adjusted permutations of a given diffusion score and adds Z-scores to the object. The function will automatically determine if the original diffusion was a single or tied diffusion and construct permutations accordingly.

Usage

```
permute_seed_nodes(
  object,
  perm_iters = 1000L,
  random_seed = 10101L,
  .verbose = TRUE
)
```

Arguments

`object`	`NetworkDiffusions` object. The underlying class `NetworkDiffusions()`.
`perm_iters`	Integer. Number of permutations to test for. Defaults to `1000L`.
`random_seed`	Integer. Random seed for determinism.
`.verbose`	Boolean. Controls verbosity.

Value

The class with added diffusion score based on a single set of seed genes. Additionally, the seed genes are stored in the class.

Examples

```
# 100 node-degree adjusted permutations of a single diffusion
set.seed(42)
g <- igraph::sample_pa(15, directed = FALSE)
edges <- data.table::setDT(igraph::as_data_frame(g))[, `:=`(
  from = sprintf("node_%i", from),
  to = sprintf("node_%i", to)
)]
object <- NetworkDiffusions(edges, weighted = FALSE, directed = FALSE)
object <- diffuse_seed_nodes(object, c(node_1 = 1, node_3 = 1), "max")
object <- permute_seed_nodes(object, perm_iters = 100L, .verbose = FALSE)
head(get_diffusion_perms(object))
```

phate_sc *Run PHATE on a SingleCells/MetaCells object*

Description

Wrapper around `manifoldsR::phate()` for the `SingleCells` and `MetaCells` classes. PHATE (Potential of Heat-diffusion for Affinity-based Trajectory Embedding) produces a low-dimensional embedding that preserves both local and global structure by operating on a diffusion process over the data manifold. Unlike UMAP or t-SNE, PHATE is explicitly designed to reveal continuous progressions and branching structure, making it the preferred choice for data with developmental or trajectory-like organisation.

When `use_knn = TRUE` (the default), the kNN graph already stored on the object is reused; otherwise neighbours are computed from the chosen embedding. The algorithm then constructs a diffusion operator, raises it to a power (the diffusion time `t`, see `manifoldsR::params_phate()`) that denoises the manifold, and computes potential distances that are finally embedded via metric MDS.

Because PHATE inherently smooths over the kNN graph, it pairs naturally with `MetaCells`: the combination yields a particularly clean view of continuous biological processes on denoised data.

Usage

```
phate_sc(
  object,
  use_knn = TRUE,
  embd_to_use = "pca",
  slot_name = "phate",
  no_embd_to_use = NULL,
  modality = c("rna", "adt", "wnn"),
  n_dim = 2L,
  k = 5L,
  knn_method = c("kmknn", "hnsw", "annoy", "nndescent", "balltree", "ivf", "exhaustive"),
  nn_params = manifoldsR::params_nn(),
  phate_params = manifoldsR::params_phate(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells`, `MetaCells` class.
`use_knn`	Boolean. Use the kNN graph found in the object. Defaults to `TRUE`. If not available, will default to the embedding.
`embd_to_use`	String. The embedding to use for PHATE. Must be available in the object.
`slot_name`	String. The name of this embedding within the object. Defaults to `"phate"`.

`no_embd_to_use`
: Optional integer. Number of embedding dimensions to use. If `NULL` all will be used.

`modality`
: String. On which modality to run PHATE. One of `c("rna", "adt", "wnn")`. The two latter options are only available for multi-modal versions with the added data.

`n_dim`
: Integer. Number of PHATE dimensions. Currently only `2L` is supported. Defaults to `2L`.

`k`
: Integer. Number of nearest neighbours for graph construction. Defaults to `5L`.

`knn_method`
: String. Approximate nearest neighbour algorithm. One of `c("kmknn", "hnsw", "annoy", "nndescent", "balltree", "ivf", "exhaustive")`. Defaults to `"kmknn"`. Only used when neighbours are computed from the embedding.

`nn_params`
: Named list. See `manifoldsR::params_nn()`.

`phate_params`
: Named list. See `manifoldsR::params_phate()`.

`seed`
: Integer. For reproducibility.

`.verbose`
: Boolean. Controls verbosity.

Value

The object with a `"phate"` embedding added.

Examples

```
# PHATE embedding off the cached kNN graph
sc <- demo_single_cells()
sc <- phate_sc(sc, .verbose = FALSE)
dim(get_embedding(sc, "phate"))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

plot.LdaKSweepResult *Plot the LDA topic count sweep*

Description

The three selection metrics and the combined score against `k`. Look for the `k` where the combined score peaks, then sanity check that Cao Juan has not already started climbing, which is the signal that topics are duplicating.

Usage

```
## S3 method for class 'LdaKSweepResult'
plot(x, ...)
```

Arguments

x	`LdaKSweepResult` object.
...	Ignored.

Value

A `ggplot2` object with one panel per metric.

Examples

```
# the three selection metrics and the combined score against k
set.seed(42L)
corpus <- matrix(rbinom(200L * 40L, 1L, 0.05), nrow = 200L, ncol = 40L)
corpus[1:100, 1:10] <- rbinom(1000L, 1L, 0.6)
corpus[101:200, 11:20] <- rbinom(1000L, 1L, 0.6)
colnames(corpus) <- sprintf("term_%02d", 1:40)
sweep_res <- lda_k_sweep(corpus > 0, k_range = 5:7, .verbose = FALSE)
plot(sweep_res)
```

`plot.NmfKSweepResult` *Plot a consensus NMF k sweep*

Description

Stability and relative reconstruction error against `k`, the standard cNMF diagnostic. Pick the `k` where stability is still high and the error curve has not yet flattened out. Values of `k` whose consensus step failed have no stability point.

Usage

```
## S3 method for class 'NmfKSweepResult'
plot(x, ...)
```

Arguments

x	`NmfKSweepResult` object.
...	Additional params. Currently unused.

Value

A `ggplot2` object with the two curves side by side.

References

Kotliar et al., eLife, 2019

Examples

```
# stability against reconstruction error across k
sc <- demo_single_cells()
res <- nmf_k_sweep_sc(
  sc,
  k_range = 2:4,
  n_runs = 3L,
  nmf_consensus_params = params_nmf_consensus(density_threshold = 2),
  .verbose = FALSE
)
plot(res)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

plot_boxplot_normalization

Helper plot function for boxplot of normalised data

Description

Helper plot function for boxplot of normalised data

Usage

```
plot_boxplot_normalization(samples, norm_counts, group_col)
```

Arguments

`samples`	data.table with sample information with perc_detected_genes and a column specifying the cohort.
`norm_counts`	Numeric matrix. The normalised log2 expression, genes x samples, with the samples in the same order as the rows of `samples`.
`group_col`	String. The grouping column.

Value

ggplot object, i.e., box plot with expression per sample.

Examples

```
# normalised expression per sample, coloured by cohort
syn <- synthetic_bulk_cor_matrix()
samples <- data.table::data.table(
  sample_id = colnames(syn$counts),
  cohort = rep(c("case", "control"), each = 50)
)
norm_counts <- rs_cpm(syn$counts, lib_size = NULL, log = TRUE,
  prior_count = 0.5)
plot_boxplot_normalization(samples, norm_counts, group_col = "cohort")
```

plot_epsilon_res *Plot the epsilon vs. power law goodness of fit result*

Description

Plots the epsilon results (if they can be found in the class). The x-axis reflects the different epsilon parameters for the radial basis function, and the y-axis the R2 value that the resulting networks follows a power law distribution (i.e., scale free topology).

Usage

```
plot_epsilon_res(object)
```

Arguments

object	The class, see BulkCoExp().

Value

If epsilon results were found, returns the ggplot. Otherwise, throws a warning and returns NULL.

Examples

```
# epsilon versus scale-free fit
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- cor_module_processing(obj, cor_method = "spearman", .verbose = FALSE)
obj <- cor_module_check_epsilon(
  obj, rbf_func = "gaussian", .verbose = FALSE
)
plot_epsilon_res(obj)
```

plot_hvgs *Plot the highly variable genes*

Description

Plots the median-absolute deviation of the genes and applied thresholds. Expects that preprocess_bulk_coexp() was run and will throw an error otherwise.

Usage

```
plot_hvgs(object, bins = 50L)
```

Arguments

object	The underlying class, see BulkCoExp().
bins	Integer. Number of bins to plot.

Examples

```
# MAD distribution with the selected genes highlighted
syn <- synthetic_bulk_cor_matrix()
mat <- log1p(t(syn$counts))
meta <- data.table::data.table(sample_id = rownames(mat))
object <- BulkCoExp(raw_data = mat, meta_data = meta)
object <- preprocess_bulk_coexp(object, hvg = 200L, .verbose = FALSE)
plot_hvgs(object)
```

---

plot_ica_ncomp_params

Plot various parameters with no comp

Description

Helper function to plot various parameters with the no of components. You have:

- Median stability - The median stability of the clusters at this no of components given random initialisations.
- % Converged - The percentage of ICA runs that converged at this no of components.
- IC redundancy - `1 - max_abs_loading_cor`, one minus the largest absolute correlation between two component loadings at this level.
- Combined score - The product of the three other scores.

If found, the function will also add the optimal number of components based on `ica_optimal_ncomp()` (if a loess function could be fitted).

Usage

```
plot_ica_ncomp_params(object)
```

Arguments

object	The class, see BulkCoExp(). You need to apply ica_evaluate_comp() before running this function.

Value

The plot with no comp ~ vs. various parameters.

Examples

```
# stability, convergence and orthogonality against ncomp
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- ica_processing(obj, .verbose = FALSE)
obj <- ica_evaluate_comp(
  obj,
  ica_type = "logcosh",
  ncomp_params = params_ica_ncomp(custom_seq = seq(2L, 20L, by = 2L)),
  .verbose = FALSE
)
plot_ica_ncomp_params(obj)
```

---

plot_ica_stability_individual

Plot the stability of the ICA components

Description

Helper function to plot the individual stability profiles over the tested ncomps.

Usage

```
plot_ica_stability_individual(object)
```

Arguments

object	The class, see BulkCoExp(). You need to apply ica_evaluate_comp() before running this function.

Value

A ggplot with the per-component stability profiles over the tested number of components.

Examples

```
# per-component stability profiles across the tested ncomps
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- ica_processing(obj, .verbose = FALSE)
obj <- ica_evaluate_comp(
  obj,
  ica_type = "logcosh",
  ncomp_params = params_ica_ncomp(custom_seq = seq(2L, 20L, by = 2L)),
  .verbose = FALSE
)
plot_ica_stability_individual(obj)
```

plot_optimal_cuts *Plot the k cuts vs median R2*

Description

Plots the optimal k vs. median of median R2 graph to identify the optimal number of cuts.

Usage

```
plot_optimal_cuts(object)
```

Arguments

object	The class, see BulkCoExp().

Value

If optimal cuts results were found, returns the ggplot. Otherwise, throws a warning and returns NULL.

Examples

```
# k cuts versus median R2, with the chosen cut marked
mat <- t(synthetic_signal_matrix()$mat)
obj <- BulkCoExp(mat, data.table::data.table(sample_id = rownames(mat)))
obj <- preprocess_bulk_coexp(obj, hvg = 0.3, .verbose = FALSE)
obj <- cor_module_processing(obj, cor_method = "spearman", .verbose = FALSE)
obj <- cor_module_coremo_clustering(obj, .verbose = FALSE)
plot_optimal_cuts(obj)
```

plot_pca *Helper plot function for pca with contrasts*

Description

Helper plot function for pca with contrasts

Usage

```
plot_pca(pca_dt, grps)
```

Arguments

pca_dt	data.table. data.table with PCA and contrast information.
grps	String. Name of the column in pca_dt holding the groups.

Value

ggplot object for the pca

Examples

```
# PC1 against PC2, coloured by the named grouping column
syn <- synthetic_bulk_cor_matrix()
pcs <- stats::prcomp(t(log1p(syn$counts)))$x[, 1:2]
pca_dt <- data.table::data.table(
  PC_1 = pcs[, 1],
  PC_2 = pcs[, 2],
  cohort = rep(c("case", "control"), each = 50)
)
plot_pca(pca_dt, grps = "cohort")
```

---

plot_pca_res *Plot the PCA data*

Description

Plot the PCA data

Usage

```
plot_pca_res(
  object,
  cols_to_plot = c("contrast_info", "sample_source"),
  pcs_to_plot = c("PC_1", "PC_2"),
  ...
)
```

Arguments

`object`	The underlying class, see `BulkDge()`.
`cols_to_plot`	String vector. The columns within the meta-data to plot. Defaults to `c('contrast_info', 'sample_source')`
`pcs_to_plot`	String vector of length 2. Will default to `c("PC_1", "PC_2")`.
`...`	additional parameters

Value

A plot if the PCA information was found. `NULL` if no PCA was found.

Examples

```
# PCA faceted over the metadata columns you care about
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- qc_bulk_dge(object, group_col = "case_control", .verbose = FALSE)
object <- normalise_bulk_dge(
  object,
  group_col = "case_control",
  .verbose = FALSE
)
object <- calculate_pca_bulk_dge(object, no_hvg_genes = 500L)
plot_pca_res(object, cols_to_plot = "case_control")
```

---

plot_preprocessing_genes

Helper plot function of distribution of genes by samples

Description

Boxplot of the number of detected genes per sample, split by cohort. Used in the bulk DGE pre-processing report to spot samples with poor library complexity.

Usage

```
plot_preprocessing_genes(samples, group_col)
```

Arguments

samples	data.table with sample information with nb_detected_genes and a column specifying the cohort.
group_col	String specifying the column with cohort information

Value

ggplot object, i.e. boxplot with number of genes by cohort

Examples

```
# detected genes per sample, split by cohort
syn <- synthetic_bulk_cor_matrix()
samples <- data.table::data.table(
  cohort = rep(c("case", "control"), each = 50),
  nb_detected_genes = colSums(syn$counts > 0)
)
plot_preprocessing_genes(samples, group_col = "cohort")
```

`plot_preprocessing_outliers` *Helper plot function for identification of outliers*

Description

Beeswarm plot of the percentage of detected genes per sample, with the outlier cutoffs drawn in. Used in the bulk DGE pre-processing report.

Usage

```
plot_preprocessing_outliers(samples, group_col, min_perc, max_perc)
```

Arguments

`samples`	data.table with sample information with perc_detected_genes and a column specifying the cohort.
`group_col`	String specifying the column with cohort information
`min_perc`	Numeric. Lower cutoff to identify outliers.
`max_perc`	Numeric. Upper cutoff to identify outliers

Value

ggplot object, i.e., beeswarm plot with outlier indication

Examples

```
# percentage of detected genes with the outlier cutoffs drawn in
syn <- synthetic_bulk_cor_matrix()
samples <- data.table::data.table(
  cohort = rep(c("case", "control"), each = 50),
  perc_detected_genes = colMeans(syn$counts > 0) * 100
)
plot_preprocessing_outliers(
  samples,
  group_col = "cohort",
  min_perc = 60,
  max_perc = 95
)
```

plot_rbf_impact *Helper function to plot distance to affinity relationship*

Description

This function plots the distance to affinity relationship after applying a given radial basis function (RBF) with a given epsilon value.

Usage

```
plot_rbf_impact(rbf_type, epsilon)
```

Arguments

rbf_type	String. The RBF type. One of `c("gaussian", "bump", "inverse_quadratic")`
epsilon	Numerical. The epsilon parameter.

Value

A plot depicting the distance to affinity relationship after applying the RBF function.

Examples

```
# distance to affinity under a Gaussian RBF with epsilon 2
plot_rbf_impact(rbf_type = "gaussian", epsilon = 2)
```

plot_resolution_res *Plot the resolution results.*

Description

Plots the resolution results (if they can be found in the class). The x-axis reflects the different resolutions and the y axis the modularity observed with that resolution.

Usage

```
plot_resolution_res(object, print_head = TRUE, ...)
```

Arguments

object	The class, either `RbhGraph` or `BulkCoExp`.
print_head	Boolean. Print the Top5 resolution parameters and their meta data. Only applicable for `BulkCoExp` objects.
...	Additional arguments to parse to the functions.

Value

Plots the result, if the results were found in the class. Otherwise, throws a warning and returns NULL.

Examples

```
# modularity across the resolutions tested on an RBH graph
set.seed(123)
modules <- data.table::data.table(
  origin = rep(c("set_a", "set_b"), each = 20),
  module = rep(c("m1", "m2", "m3", "m4"), each = 10),
  gene = unlist(replicate(4, sample(letters, 10), simplify = FALSE))
)
object <- RbhGraph(
  modules,
  rbh_type = "set",
  dataset_col = "origin",
  module_col = "module",
  value_col = "gene"
)
object <- generate_rbh_graph(object, minimum_similarity = 0)
object <- find_rbh_communities(object, parallel = FALSE, .verbose = FALSE)
plot_resolution_res(object)
```

---

plot_voom_normalization

Helper plot function for Voom normalisation

Description

Helper plot function for Voom normalisation

Usage

```
plot_voom_normalization(norm_counts)
```

Arguments

`norm_counts`	Numeric matrix. The voom log2-CPM values, genes x samples.

Value

ggplot object, i.e., voom normalisation plot.

Examples

```
# mean variance trend after voom
syn <- synthetic_bulk_cor_matrix()
grp <- rep(c("case", "control"), each = 50)
norm_counts <- rs_cpm(syn$counts, lib_size = NULL, log = TRUE,
  prior_count = 0.5)
plot_voom_normalization(norm_counts)
```

---

pre_process_sim_onto *Pre-process data for subsequent ontology similarity*

Description

This function calculates needed information for semantic similiary calculations

Usage

```
pre_process_sim_onto(object, .verbose = TRUE)
```

Arguments

object	OntologySim class. See OntologySim().
.verbose	Boolean. Controls verbosity of the function.

Value

The class with added pre-processed data for semantic similarities to the properties.

Examples

```
# ancestors, descendants and information content added to the class
onto <- data.table::data.table(
  parent = c("a", "b", "b", "b", "c"),
  child = c("b", "c", "d", "e", "f"),
  type = c("part_of", "part_of", "part_of", "is_a", "is_a")
)
onto_obj <- OntologySim(onto, .verbose = FALSE)
onto_obj <- pre_process_sim_onto(onto_obj, .verbose = FALSE)
names(S7::prop(onto_obj, "outputs"))
```

prepare_cell_markers *Helper function to prepare cell markers*

Description

This function is a helper to generate a list of cell type to cell marker annotations that can be fed into for example [REF].

Usage

```
prepare_cell_markers(obj, marker_df)
```

Arguments

obj	A single cell class, i.e. one of SingleCells or SingleCellsMultiModal.
marker_df	A data.table with the columns "cell_type" and "gene_id". You need to ensure that the gene id type matches the obj gene id.

Value

A list of cell type to marker associations ready for subsequent usage. Genes not found in the object will be automatically removed.

Examples

```
# a marker table turned into the indexed list the scorers want
sc <- demo_single_cells(prepped = FALSE)
markers <- data.table::data.table(
  cell_type = c("type_a", "type_a", "type_b"),
  gene_id = get_gene_names(sc)[1:3]
)
names(prepare_cell_markers(sc, markers))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

preprocess_bulk_coexp *Process the raw data*

Description

Function to do general pre-processing on top of the BulkCoExp(). Options to do scaling, HVG selection, etc.

Usage

```
preprocess_bulk_coexp(
  object,
  hvg = NULL,
  mad_threshold = NULL,
  scaling = FALSE,
  scaling_type = c("normal", "robust"),
  .verbose = TRUE
)
```

Arguments

`object`	The underlying class, see `BulkCoExp()`.
`hvg`	Integer or float. If an integer, the top `hvg` genes will be included; if float, the float has to be between 0 and 1, representing the percentage of genes to include.
`mad_threshold`	Float. Instead of of selecting number or proportion of genes, you can also provide a mad_threshold.
`scaling`	Boolean. Shall the data be scaled.
`scaling_type`	String. You have the option to use normal scaling or robust scaling.
`.verbose`	Boolean. Controls the verbosity of the function.

Value

Returns the class with the `processed_data` data slot populated and applied parameters added to the `params` slot.

Examples

```
# keep the 200 most variable genes for module detection
syn <- synthetic_bulk_cor_matrix()
mat <- log1p(t(syn$counts))
meta <- data.table::data.table(sample_id = rownames(mat))
object <- BulkCoExp(raw_data = mat, meta_data = meta)
object <- preprocess_bulk_coexp(object, hvg = 200L, .verbose = FALSE)
object
```

`preprocess_bulk_dge` *QC on the bulk dge data (DEPRECATED!)*

Description

This is a deprecated method and will raise an error. Please use `qc_bulk_dge()` and `normalise_bulk_dge()` instead.

Usage

```
preprocess_bulk_dge(
  object,
  group_col,
  norm_method = c("TMM", "TMMwsp", "RLE", "upperquartile", "none"),
  outlier_threshold = 2,
  min_prop = 0.2,
  .verbose = TRUE
)
```

Arguments

`object`	The underlying class, see `BulkDge()`.
`group_col`	String. The column in the metadata that will contain the contrast groups. Needs to be part of the metadata stored in the class.
`norm_method`	String. One of `c("TMM", "TMMwsp", "RLE", "upperquartile", "none")`. Please refer to edgeR's `calcNormFactors()`.
`outlier_threshold`	Float. Number of standard deviations in terms of percentage genes detected you allow before removing a sample. Defaults to `2`.
`min_prop`	Float. Minimum proportion of samples in which the gene has to be identified in.
`.verbose`	Boolean. Controls the verbosity of the function.

Value

Throws an error, as it is deprecated.

`prescan_h5ad_files` *Pre-scan multiple h5ad files for multi-sample loading*

Description

Pre-scan multiple h5ad files for multi-sample loading

Usage

```
prescan_h5ad_files(
  h5_paths,
  gene_universe = c("intersection", "union"),
  var_index = "_index",
  raw_count_slot = c("auto", "X", "raw.X", "layers.counts"),
  .verbose = TRUE
)
```

Arguments

h5_paths	Character vector of file paths to h5ad files. If names are provided, these will be used as experimental identifiers.
gene_universe	One of "intersection" or "union".
var_index	String. The name within the h5ad var part in which the variable names are stored. Defaults to "_index".
raw_count_slot	Where raw counts live. "auto" detects per file via detect_raw_count_slot(); otherwise one of "X", "raw.X", "layers.counts".
.verbose	Boolean. Controls verbosity of the function.

Value

A list with:

- universe - Character vector of gene names in the universe
- file_tasks - List of per-file task structures, each containing: exp_id, h5_path, cs_type, no_cells, no_genes, gene_local_to_universe (integer vector, NA for genes not in universe, 0-indexed into universe)

Examples

```
# build the gene universe across two files before a multi-file load
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
files <- c(a = tempfile(fileext = ".h5ad"), b = tempfile(fileext = ".h5ad"))
for (f in files) {
  write_h5ad_sc(f, data$counts, data$obs, data$var, .verbose = FALSE)
}
tasks <- prescan_h5ad_files(h5_paths = files, .verbose = FALSE)
tasks$universe_size

unlink(files)
```

---

prescan_mtx_dirs *Prescan multiple mtx directories for a multi-load*

Description

Walks each input directory, reads the features file to build the **intersection** of gene IDs across inputs (matched by the first column, typically Ensembl gene IDs), decompresses any .mtx.gz files into a temporary directory, and builds the file tasks expected by load_multi_mtx().

Each input directory must contain the standard 10x trio: a .mtx (or .mtx.gz) file, a barcodes file, and a features/genes file. File names are matched by extension; if a directory contains multiple matching files an error is raised.

Usage

```
prescan_mtx_dirs(
  dirs,
  exp_ids,
  cells_as_rows = FALSE,
  has_hdr = FALSE,
  .verbose = TRUE
)
```

Arguments

`dirs`	Character vector of input directories. Length >= 2.
`exp_ids`	Character vector of experiment identifiers, one per directory. Must be unique.
`cells_as_rows`	Boolean. Applied uniformly to all inputs. Defaults to `FALSE` (10x convention: genes are rows).
`has_hdr`	Boolean. Whether the barcodes/features files have a header row. Applied uniformly. Defaults to `FALSE` (10x convention).
`.verbose`	Boolean. Controls verbosity of the function.

Value

A list with:

- universe - Character vector of gene IDs in the intersection, in the order they will appear in the final var table.
- universe_size - Length of the universe.
- file_tasks - List of per-input task lists for Rust and DuckDB.
- temp_files - Character vector of temp files created during decompression; the caller should `unlink()` these after use.

Examples

```
# two CellRanger style directories reduced to their shared gene universe
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
dirs <- c(tempfile("cr_a"), tempfile("cr_b"))
for (d in dirs) {
  dir.create(d, recursive = TRUE)
  write_cellranger_output(
    d, data$counts, data$obs, data$var,
    rows = "cells", format_type = "csv", .verbose = FALSE
  )
}
scan_res <- prescan_mtx_dirs(
  dirs = dirs,
  exp_ids = c("a", "b"),
```

```
  cells_as_rows = TRUE,
  has_hdr = TRUE,
  .verbose = FALSE
)
scan_res$universe_size

unlink(c(dirs, scan_res$temp_files), recursive = TRUE, force = TRUE)
```

`prescan_tenx_h5_files` *Pre-scan multiple 10x CellRanger h5 files for multi-sample loading*

Description

Walks each input file, reads the per-file feature ids, restricts to the chosen `feature_type` (V3 only; ignored for V2), builds the intersection or union universe of gene ids, and returns the file tasks expected by `load_multi_tenx_h5()`.

V2 and V3 files can be mixed in a single batch. Gene matching is done on the feature `id` (ensembl-style) since gene symbols collide.

Usage

```
prescan_tenx_h5_files(
  h5_paths,
  feature_type = "Gene Expression",
  gene_universe = c("intersection", "union"),
  .verbose = TRUE
)
```

Arguments

`h5_paths`	Character vector of file paths to 10x h5 files. If names are provided, these will be used as experimental identifiers.
`feature_type`	String. Modality to keep across all files (V3 only; ignored for V2). Defaults to `"Gene Expression"`.
`gene_universe`	One of `"intersection"` or `"union"`.
`.verbose`	Boolean. Controls verbosity.

Value

A list with:

- universe - Character vector of gene ids in the universe.
- universe_size - Length of the universe.
- file_tasks - Named list of per-file task structures, each containing `exp_id`, `h5_path`, `version`, `no_cells`, `no_genes`, `feature_type` and `gene_local_to_universe` (integer vector, `NA` for features outside the universe / non-target modality, 0-indexed).

Examples

```
# gene universe across two 10x h5 files
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
features <- data.table::data.table(
  id = data$var$gene_id,
  name = data$var$ensembl_id,
  feature_type = "Gene Expression"
)
files <- c(a = tempfile(fileext = ".h5"), b = tempfile(fileext = ".h5"))
for (f in files) {
  write_tenx_h5_sc(f, data$counts, data$obs$cell_id, features)
}
scan_res <- prescan_tenx_h5_files(h5_paths = files, .verbose = FALSE)
scan_res$universe_size

unlink(files)
```

---

prioritise_interactions

Prioritise sender-ligand-receiver-receptor interactions

Description

For every (sender, ligand, receiver, receptor) tuple drawn from the LR network, compute a weighted prioritisation score combining:

- ligand DE in the sender (`de_ligand`)
- receptor DE in the receiver (`de_receptor`)
- ligand activity in the receiver (`activity_scaled`)
- ligand expression specificity across senders (`exprs_ligand`)
- receptor expression specificity across receivers (`exprs_receptor`)
- ligand condition specificity (`ligand_condition_specificity`)
- receptor condition specificity (`receptor_condition_specificity`)

Components with weight zero are skipped (their joins are not performed).

DE tables are supplied by the user. `find_markers_sc` and `find_all_markers_sc` give a quick Wilcox; pseudo-bulk + DESeq2 / edgeR is generally preferable for statistical inference.

Usage

```
prioritise_interactions(
  celltype_de,
  expression_info,
  ligand_activities,
```

```
  lr_network,
  senders_oi,
  receivers_oi,
  condition_de = NULL,
  weights = NULL,
  scenario = c("case_control", "one_condition")
)
```

Arguments

`celltype_de`	A `data.table` with columns `cluster_id`, `gene`, `lfc`, `pval`. One row per (cluster, gene). Must cover both senders and receivers.
`expression_info`	A `data.table` with columns `cluster_id`, `gene`, `avg_expr`. As returned by `compute_expression_info_sc`.
`ligand_activities`	A `data.table` with at least the columns `ligand` and `aupr_corrected`. One row per ligand. Subset to a single gene set before calling.
`lr_network`	A `data.table` with columns `ligand`, `receptor`.
`senders_oi`	Character vector of sender cluster IDs.
`receivers_oi`	Character vector of receiver cluster IDs.
`condition_de`	Optional `data.table` with columns `gene`, `lfc`, `pval` from a condition contrast (e.g. case vs control). Required when `scenario == "case_control"` and the corresponding weights are non-zero.
`weights`	Optional named numeric vector. If `NULL`, defaults are chosen by `scenario`. Names: `de_ligand`, `de_receptor`, `activity_scaled`, `exprs_ligand`, `exprs_receptor`, `ligand_condition_specificity`, `receptor_condition_specificity`.
`scenario`	`"case_control"` (all weights 1) or `"one_condition"` (condition-specificity weights 0). Ignored if `weights` is supplied.

Value

A `data.table` with one row per surviving (sender, ligand, receiver, receptor) tuple, sorted by descending `prioritisation_score`, with a `prioritisation_rank` column.

Examples

```
# one sender, one receiver, with the signal planted on L1 -> R1
lr_network <- data.table::data.table(
  ligand = c("L1", "L2"),
  receptor = c("R1", "R2")
)
celltype_de <- data.table::CJ(
  cluster_id = c("sender", "receiver"),
  gene = c("L1", "L2", "R1", "R2")
)
celltype_de[, lfc := c(3, 0.1, 0.1, 0.1, 0.1, 0.1, 2.5, 0.1)]
celltype_de[, pval := c(1e-10, 0.5, 0.5, 0.5, 0.5, 0.5, 1e-9, 0.5)]
```

```
expression_info <- celltype_de[, .(cluster_id, gene, avg_expr = 0.05)]
expression_info[cluster_id == "sender" & gene == "L1", avg_expr := 5]
expression_info[cluster_id == "receiver" & gene == "R1", avg_expr := 5]
res <- prioritise_interactions(
  celltype_de = celltype_de,
  expression_info = expression_info,
  ligand_activities = data.table::data.table(
    ligand = c("L1", "L2"),
    aupr_corrected = c(0.75, -0.125)
  ),
  lr_network = lr_network,
  senders_oi = "sender",
  receivers_oi = "receiver",
  scenario = "one_condition"
)
head(res[, .(sender, ligand, receiver, receptor, prioritisation_score)])
```

`process_go_data` *Process Gene Ontology data into the right format*

Description

Helper function that takes in different files containing gene ontology data and puts them together for various gene set enrichment methods using the ontological information

Usage

```
process_go_data(go_info, go_genes, go_relationships)
```

Arguments

`go_info`	data.table. Contains `go_id`, `go_name` and `namespace`.
`go_genes`	data.table. Contains `go_id` and corresponding `ensembl_id`.
`go_relationships`	data.table. Contains `parent`, `child` and `relationship`

Value

data.table ready for usage in `GeneOntologyElim()`.

Examples

```
# assemble the packaged human GO data by hand
go_data <- load_go_human_data()
relationships <- data.table::setnames(
  data.table::copy(go_data$gene_ontology),
  old = c("from", "to"),
  new = c("parent", "child")
```

```
)
go_dt <- process_go_data(
  go_info = go_data$go_info,
  go_genes = go_data$go_to_genes,
  go_relationships = relationships[relationship %in% c("is_a", "part_of")]
)
dim(go_dt)
```

`pseudobulk_dge_sc` *Run the edgeR quasi-likelihood workflow on pseudo-bulked single cells*

Description

Sums the raw counts per sample and treats the result as a bulk experiment. That is the whole method, and it is the one that holds its nominal false discovery rate when the cells within a sample are not independent, which they never are.

This is a convenience wrapper: it pseudo-bulks with `get_pseudobulked_sc()` and hands the aggregate to `run_edger_ql()`. Reach for the two separately if you want to inspect or reuse the aggregated matrix.

Usage

```
pseudobulk_dge_sc(
  object,
  cell_list,
  design,
  coef = NULL,
  contrast = NULL,
  edger_params = params_edger_ql(),
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class.
`cell_list`	Named list of character vectors. The cell identifiers per pseudo-bulk sample. The names become the sample identifiers, and the rows of `design` must follow that order.
`design`	Numeric matrix. The design matrix of samples x coefficients, including the intercept. Rows aligned to `cell_list`.
`coef`	Optional integer or character. See `run_edger_ql()`.
`contrast`	Optional numeric vector or matrix. See `run_edger_ql()`.
`edger_params`	A list, see `params_edger_ql()`.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

A data.table, see `run_edger_ql()`. The `feature_id` column holds the gene identifiers.

References

Squair, et al., Nat Commun, 2021; Chen, Lun and Smyth, F1000Research, 2016

Examples

```
# six pseudo-bulk samples out of a demo single cell object
sc <- demo_single_cells(prepped = FALSE)
cells <- get_sc_obs(sc)$cell_id
cell_list <- split(
  cells,
  rep(sprintf("sample_%i", 1:6), length.out = length(cells))
)
design <- stats::model.matrix(~ rep(c("ctrl", "case"), each = 3))
res <- pseudobulk_dge_sc(sc, cell_list, design, .verbose = FALSE)
head(res)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

qc_bulk_dge *QC on the bulk dge data*

Description

This function will do QC on the bulk data and remove outlier samples that show substantially lower expression of genes compared to the rest of the data and remove lowly expressed genes.

Usage

```
qc_bulk_dge(
  object,
  group_col,
  outlier_threshold = 2,
  min_prop = 0.2,
  min_count = 10,
  .verbose = TRUE
)
```

Arguments

`object`	The underlying class, see `BulkDge()`.
`group_col`	String. The column in the metadata that will contain the contrast groups. Needs to be part of the metadata stored in the class.

`outlier_threshold`
: Float. Number of standard deviations in terms of percentage genes detected you allow before removing a sample. Defaults to `2`.

`min_prop`
: Float. Minimum proportion of samples in which the gene has to be identified in.

`min_count`
: Float. Minimum number of counts (cpm) to be detected in min_prop of the samples (in cohorts defined by groups_coll)

`.verbose`
: Boolean. Controls the verbosity of the function.

Value

Returns the class with the `processed_data` data slot populated and applied parameters added to the `params` slot.

Examples

```
# drop outlier samples and lowly expressed genes
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- qc_bulk_dge(object, group_col = "case_control", .verbose = FALSE)
dim(get_outputs(object)$raw_counts_filtered)
```

`RbhGraph` *Reciprocal best hit graph*

Description

This class can be used to generate reciprocal best hit graphs between gene modules from different origins.

Usage

```
RbhGraph(
  module_results,
  rbh_type = c("cor", "set"),
  dataset_col = NULL,
  module_col = NULL,
  value_col = NULL
)
```

Arguments

`module_results`	data.table or list. If data.table (you set the class to `"set"`, i.e., set similarities will be used) it cointains all of the gene modules for which you wish to generate the RBH graph and you need to have the three column names. If list (you set the class to ‘"cor"“, i.e., correlations between for example gene loadings will be used), the list contains all of the matrices (must have col and row names).
`rbh_type`	String. One of `c("cor", "set")`.
`dataset_col`	The column (name) which indicates from which data set/ method the gene module was derived. Only needed if you want to use set similarities.
`module_col`	The column (name) which stores the names of the modules. Only needed if you want to use set similarities.
`value_col`	The column (name) which stores the genes that are part of the modules. Only needed if you want to use set similarities.

Value

Returns the `RbhGraph` class for further operations.

Properties

module_data Nested list of the modules generated by different methods.

rbh_graph igraph. The reciprocal best hit graph.

rbh_edge_df data.table. Contains the RBH graph data as an edge df.

params A (nested) list that will store all the parameters of the applied function.

final_results data.table. Contains final results.

Examples

```
# set similarity based RBH class over four gene modules
set.seed(123)
modules <- data.table::data.table(
  origin = rep(c("set_a", "set_b"), each = 20),
  module = rep(c("m1", "m2", "m3", "m4"), each = 10),
  gene = unlist(replicate(4, sample(letters, 10), simplify = FALSE))
)
object <- RbhGraph(
  modules,
  rbh_type = "set",
  dataset_col = "origin",
  module_col = "module",
  value_col = "gene"
)
class(object)[1]
```

read_h5ad_metadata *Read obs and var tables and metadata from an h5ad file*

Description

Useful for exploring the data stored in an h5ad file.

Usage

```
read_h5ad_metadata(f_path)
```

Arguments

f_path	File path to the .h5ad file.

Value

A list with:

- obs - data.table of cell-level metadata
- var - data.table of gene-level metadata
- dims - named integer vector c(obs, var)
- type - "CSR" or "CSC"

Examples

```
# obs and var tables straight out of the file
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
f_path <- tempfile(fileext = ".h5ad")
write_h5ad_sc(f_path, data$counts, data$obs, data$var, .verbose = FALSE)
meta <- read_h5ad_metadata(f_path)
head(meta$var, 3)

unlink(f_path)
```

read_h5ad_x_summary *Read summary statistics from the X slot of an h5ad file*

Description

Read summary statistics from the X slot of an h5ad file

Usage

```
read_h5ad_x_summary(f_path, n_sample = 10000L)
```

Arguments

`f_path`	File path to the `.h5ad` file.
`n_sample`	Number of non-zero values to sample for the preview. NULL reads all.

Value

A list with:

- stats - named vector: min, max, mean, median, and fraction of values that are whole numbers
- is_integer_valued - logical; TRUE if >99% of sampled non-zero values are whole numbers
- type - "CSR" or "CSC"
- dims - named integer vector c(obs, var)
- sample - numeric vector of sampled non-zero values

Examples

```
# check whether /X holds raw counts before loading it
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
f_path <- tempfile(fileext = ".h5ad")
write_h5ad_sc(f_path, data$counts, data$obs, data$var, .verbose = FALSE)
summary_x <- read_h5ad_x_summary(f_path)
summary_x$stats
summary_x$is_integer_valued

unlink(f_path)
```

`read_motif_annotation_file`

Read in the motif annotation file

Description

This function loads in the motif2tf information that you can get from `https://resources.aertslab.org/cistarg`
The function will generate a data.table that can be subsequently used.

Usage

```
read_motif_annotation_file(annot_file)
```

Arguments

`annot_file`	String. Path to the motif2tf file that you downloaded.

Value

data.table with the motif to transcription factor information.

Examples

```
## Not run:
# motif to transcription factor table from the downloaded reference
paths <- download_cistarget_hg38()
annot <- read_motif_annotation_file(paths$motif_annotations)
head(annot)

## End(Not run)
```

---

read_motif_ranking *Read in the motif rankings and transform them into a matrix*

Description

This function loads in the .feather files with the motif to target gene rankings. These can be found here: `https://resources.aertslab.org/cistarget/databases/`

Usage

```
read_motif_ranking(ranking_file)
```

Arguments

`ranking_file`	String. The file path to the .feather file

Value

An integer matrix that has been transposed for easier use in the underlying Rust code.

Examples

```
## Not run:
# transposed motif rankings from the downloaded feather file
paths <- download_cistarget_hg38()
rankings <- read_motif_ranking(paths$rankings)
dim(rankings)

## End(Not run)
```

`read_multi_tenx_h5_adt` *Read in 10x h5 ADT data from multiple files*

Description

Multi-file counterpart to `read_tenx_h5_adt()`. Reads the same modality from each input, stacks the cells (rows), and prefixes each barcode with its `exp_id` so the result matches the cell_id convention used by `load_multi_tenx_h5()`. The feature space is either the intersection or union of features across inputs; missing features in the union case are filled with zero.

Usage

```
read_multi_tenx_h5_adt(
  h5_paths,
  feature_type = "Antibody Capture",
  gene_universe = c("intersection", "union")
)
```

Arguments

`h5_paths`	Character vector of file paths to 10x h5 files. If names are provided, these will be used as `exp_id`s; otherwise the file basename is used.
`feature_type`	String. The feature type to return. Defaults to `"Antibody Capture"`.
`gene_universe`	One of `"intersection"` or `"union"`.

Value

A dense matrix of cells x features with `exp_id_barcode` rownames and feature names as colnames.

Examples

```
# stack the ADT layer of two files, barcodes prefixed by exp_id
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
adt <- generate_single_cell_test_data_adt(
  params_sc_synthetic_data_adt(n_cells = 200L)
)
features <- data.table::data.table(
  id = c(data$var$gene_id, colnames(adt$counts)),
  name = c(data$var$ensembl_id, colnames(adt$counts)),
  feature_type = rep(
    c("Gene Expression", "Antibody Capture"),
    c(ncol(data$counts), ncol(adt$counts))
  )
```

```
)
counts <- cbind(data$counts, as(adt$counts, "RsparseMatrix"))
files <- c(a = tempfile(fileext = ".h5"), b = tempfile(fileext = ".h5"))
for (f in files) {
  write_tenx_h5_sc(f, counts, data$obs$cell_id, features)
}
adt_counts <- read_multi_tenx_h5_adt(files)
dim(adt_counts)
head(rownames(adt_counts), 2)

unlink(files)
```

`read_tenx_h5_adt` *Read in 10x h5 ADT data*

Description

Helper function to load in ADT counts from h5. Leverages Rust under the hood for faster filtering and reading.

Usage

```
read_tenx_h5_adt(f_path, feature_type = "Antibody Capture")
```

Arguments

`f_path`	String. File to the h5 file from which to read the ADT counts.
`feature_type`	String. The feature type to return. Defaults here to `"Antibody Capture"`

Value

A dense matrix of cells x features

Examples

```
# pull the antibody capture layer out of a multi-modal 10x file
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
adt <- generate_single_cell_test_data_adt(
  params_sc_synthetic_data_adt(n_cells = 200L)
)
f_path <- tempfile(fileext = ".h5")
write_tenx_h5_sc(
  f_path = f_path,
  counts = cbind(data$counts, as(adt$counts, "RsparseMatrix")),
  barcodes = data$obs$cell_id,
  features = data.table::data.table(
    id = c(data$var$gene_id, colnames(adt$counts)),
    name = c(data$var$ensembl_id, colnames(adt$counts)),
```

```
    feature_type = rep(
      c("Gene Expression", "Antibody Capture"),
      c(ncol(data$counts), ncol(adt$counts))
    )
  )
)
adt_counts <- read_tenx_h5_adt(f_path)
dim(adt_counts)

unlink(f_path)
```

`read_tenx_h5_metadata`
Read barcode and feature tables and metadata from a 10x h5 file

---

Description

Useful for exploring the data stored in a 10x CellRanger v2/v3 h5 file, including the breakdown of feature types in a multi-modal file.

Usage

```
read_tenx_h5_metadata(f_path)
```

Arguments

`f_path`	File path to the 10x `.h5` file.

Value

A list with:

- obs - data.table of barcodes
- var - data.table of features (id, name, and feature_type for v3)
- dims - named integer vector c(obs, var)
- version - "v2" or "v3"
- feature_types - named integer vector of feature_type counts (v3 only, otherwise NULL)

Examples

```
# barcodes, features and the feature type breakdown
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
f_path <- tempfile(fileext = ".h5")
write_tenx_h5_sc(
  f_path = f_path,
```

```
  counts = data$counts,
  barcodes = data$obs$cell_id,
  features = data.table::data.table(
    id = data$var$gene_id,
    name = data$var$ensembl_id,
    feature_type = "Gene Expression"
  )
)
meta <- read_tenx_h5_metadata(f_path)
meta$dims
meta$feature_types

unlink(f_path)
```

`relayout_bonsai` *Recompute the layout of a Bonsai tree*

Description

Lays the finished tree out again without searching. The tree is renumbered on the Rust side, so the inferred ancestors can come back with different node numbers; the leaves, and so the cells, keep theirs.

Usage

```
relayout_bonsai(
  x,
  layout = c("equal_angle", "equal_daylight", "dendrogram"),
  hyperbolic = FALSE
)
```

Arguments

`x`	A `BonsaiTree` object.
`layout`	String. One of `c("equal_angle", "equal_daylight", "dendrogram")`.
`hyperbolic`	Boolean. Project the layout onto the hyperbolic disk.

Value

The `BonsaiTree` with the new coordinates.

Examples

```
# the same tree as a dendrogram
sc <- demo_single_cells(
  prepped = FALSE,
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 50L)
)
tree <- bonsai_sc(sc, .verbose = FALSE)
```

```
tree <- relayout_bonsai(tree, layout = "dendrogram")
plot(tree)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`remove_adt_isotypes` *Return the ADT feature names removing the isotypes*

Description

Return the ADT feature names removing the isotypes

Usage

```
remove_adt_isotypes(feature_names, pattern = "isotype")
```

Arguments

`feature_names`	Character. ADT feature names (matrix colnames or from `get_adt_names()`).
`pattern`	String. Case-insensitive regex. Defaults to `"isotype"`.

Value

Character vector of ADT features, but anything with `"isotype"`.

Examples

```
# the same features with the isotype controls dropped
features <- c("CD3", "CD19", "IgG1_isotype", "IgG2a_isotype")
remove_adt_isotypes(features)
```

`remove_magic` *Remove the MAGIC imputed layer*

Description

Remove the MAGIC imputed layer

Usage

```
remove_magic(x, ...)

## S3 method for class 'ScCache'
remove_magic(x, ...)

## S7 method for class <bixverse::SingleCells>
remove_magic(x, ...)

## S7 method for class <bixverse::SingleCellsSubset>
remove_magic(x, ...)
```

Arguments

x	An object from which to remove the imputed layer.
...	Other parameters.

Value

The object with the imputed layer dropped.

Examples

```
# drop the imputed layer again
sc <- demo_single_cells()
sc <- run_magic_sc(sc, features = get_gene_names(sc)[1:5], .verbose = FALSE)
sc <- remove_magic(sc)
is.null(get_magic(sc))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

remove_residual_fit *Remove the fitted residual model*

Description

Remove the fitted residual model

Usage

```
remove_residual_fit(x, ...)

## S7 method for class <bixverse::MetaCells>
remove_residual_fit(x, ...)

## S3 method for class 'ScCache'
remove_residual_fit(x, ...)

## S7 method for class <bixverse::SingleCells>
remove_residual_fit(x, ...)

## S7 method for class <bixverse::SingleCellsSubset>
remove_residual_fit(x, ...)
```

Arguments

x	An object from which to remove the fitted model.
...	Other parameters.

Value

The object with the fitted model dropped.

Examples

```
# drop the fitted model again
sc <- demo_single_cells()
sc <- fit_residuals_sc(sc, method = "analytic_pearson", .verbose = FALSE)
sc <- remove_residual_fit(sc)
is.null(get_residual_fit(sc))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

remove_samples *Remove samples from object*

Description

This function allows to remove certain samples from the object

Usage

```
remove_samples(object, samples_to_remove, ...)
```

Arguments

object	The underlying object, either BulkCoExp() or BulkDge().
samples_to_remove	Character vector. The sample identifiers to remove.
...	Additional arguments to parse to the functions.

Value

Returns the object with the samples removed. This will regenerated the object from the start and remove any data in it.

Examples

```
# drop the first two samples and rebuild the object
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(sample_id = colnames(syn$counts))
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- remove_samples(object, c("sample_1", "sample_2"))
dim(S7::prop(object, "raw_counts"))
```

rescue_cells *Rescue MAD-flagged cells that fall within safe bounds*

Description

For each metric in `rescue_thresholds`, cells whose value is within `[lower, upper]` are unflagged from MAD. Hard-threshold flags are not cleared; a warning is issued when overlaps occur. Calling `rescue_cells` overwrites any prior rescue set for the metrics named.

Usage

```
rescue_cells(x, rescue_thresholds)
```

Arguments

`x`	A `CellQc` object.
`rescue_thresholds`	Named list of numeric vectors with `lower` and/or `upper`.

Value

Updated `CellQc`.

Examples

```
# un-flag MAD outliers that still sit above a sane library size
set.seed(42L)
metrics <- list(
  lib_size = c(rnorm(99, 1000, 100), 50),
  pct_mt = runif(100, 0, 20)
)
qc <- run_cell_qc(
  metrics,
  cells_to_keep = 0:99,
  directions = c(lib_size = "below", pct_mt = "above")
)
rescue_cells(qc, list(lib_size = c(lower = 500)))
```

reset_cells_to_keep *Reset the cells to keep*

Description

Restores every cell found in the binary count file and wipes the cache, taking the object back to a pristine state. Filtering only ever flips a `to_keep` flag in the DuckDB, so nothing was deleted and nothing is lost by resetting.

Wiping the cache is not optional: a PCA or a kNN computed on a filtered subset does not describe the full cell set, and keeping it would recreate exactly the mismatch this guards against. With `force = FALSE` you are asked to confirm before that happens.

Usage

```
reset_cells_to_keep(object, force = FALSE)

## S7 method for class <bixverse::MetaCells>
reset_cells_to_keep(object, force = FALSE)

## S7 method for class <bixverse::SingleCells>
reset_cells_to_keep(object, force = FALSE)

## S7 method for class <bixverse::SingleCellsMultiModal>
reset_cells_to_keep(object, force = FALSE)

## S7 method for class <bixverse::SingleCellsSubset>
reset_cells_to_keep(object, force = FALSE)
```

Arguments

`object`	`SingleCells` or `SingleCellsMultiModal` class.
`force`	Boolean. Skip the confirmation prompt. Defaults to `FALSE`, in which case an interactive session asks before wiping the cache and a non-interactive one errors, because there is no one there to ask.

Value

The object with every cell restored and an empty cache. Unchanged if the confirmation was declined.

Examples

```
# a filter taken back off, cache wiped with it
sc <- demo_single_cells(prepped = FALSE)
sc <- set_cells_to_keep(sc, get_cell_names(sc)[1:100])
sc <- reset_cells_to_keep(sc, force = TRUE)
length(get_cells_to_keep(sc))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`robust_scale` *Robust scaler.*

Description

Robust scaling, i.e., removes the median and scales data based on the interquartile range (IQR). Useful if outliers are expected. NAs will be ignored.

Usage

```
robust_scale(x)
```

Arguments

x	Numeric vector.

Value

x, robustly scaled.

Examples

```
# median-centred, IQR-scaled vector
set.seed(123)
head(robust_scale(rnorm(10)))
```

---

rs_2d_loess *Rust implementation of a Loess function*

Description

[**Experimental**] Fits a Loess regression of `y` on `x`. Points where either value is non-finite are dropped from the fit.

Usage

```
rs_2d_loess(x, y, span, degree)
```

Arguments

x	Numeric. The x values to fit.
y	Numeric. The y values to fit.
span	Numeric. The span parameter. Needs to be in `(0, 1]`.
degree	Integer. Either 1 (linear) or 2 (quadratic). Other values will cause an error.

Value

A list with the following items

- predicted - The predicted values, `0` for dropped points.
- residuals - The residuals for every data point, `0` for dropped points.
- valid_idx - 1-based indices of the points included in the fit.

`rs_batch_silhouette_width`

Calculate batch silhouette width from an embedding

Description

[**Experimental**] Computes the average silhouette width on batch labels using pairwise distances in the embedding space. Values near 0 indicate good batch mixing, values near 1 indicate batch separation.

Usage

```
rs_batch_silhouette_width(embedding, batch_vector, max_cells, verbose, seed)
```

Arguments

`embedding`	Numeric matrix. The embedding to assess (e.g. PCA or corrected embedding). Rows are cells, columns are dimensions.
`batch_vector`	Integer vector. The batch per cell. The codes need not be 0-based or contiguous.
`max_cells`	Integer or NULL. If not NULL, subsample to this many cells for performance. If NULL, all cells are used.
`verbose`	Boolean. Controls verbosity of the function.
`seed`	Integer. Seed for subsampling reproducibility.

Value

A list with the following items

- per_cell - Per-cell silhouette scores
- mean_asw - Mean silhouette width
- median_asw - Median silhouette width

`rs_blitzgsea_calibrate`

Calibrate the blitzGSEA gamma null for a signature

Description

[**Experimental**]

Draws random gene sets across a log-spaced grid of anchor sizes, fits gamma tails to the resulting enrichment scores and smooths the fitted parameters across sizes. Nothing about any gene set library enters here, so one calibration serves every library scored against that signature.

Usage

```
rs_blitzgsea_calibrate(stats, blitz_params)
```

Arguments

`stats`	Numeric vector. The gene level statistic. Needs to be sorted in descending nature.
`blitz_params`	List. The blitzGSEA parameters, see `params_blitzgsea()`. Recognised elements are `permutations`, `anchors`, `symmetric`, `centre`, `ks_test` and `seed`; anything else is ignored and any missing element takes its default.

Value

List with the following elements

- anchor_sizes Numeric vector. The anchor set sizes, ascending.
- shape_pos Numeric vector. Smoothed positive-tail gamma shape.
- scale_pos Numeric vector. Smoothed positive-tail gamma scale.
- shape_neg Numeric vector. Smoothed negative-tail gamma shape.
- scale_neg Numeric vector. Smoothed negative-tail gamma scale.
- pos_ratio Numeric vector. Smoothed fraction of positive null scores at each anchor.
- ks_pos Float. Mean goodness-of-fit p-value for the positive tail.
- ks_neg Float. Mean goodness-of-fit p-value for the negative tail.
- centred Boolean. Whether the signature was centred.

`rs_blitzgsea_score` *Score gene sets against a calibrated blitzGSEA null*

Description

[Experimental]

Each gene set costs one enrichment score plus one gamma tail evaluation. The gene sets are expected to have been filtered to the desired size bounds and intersected with the signature already.

Usage

```
rs_blitzgsea_score(stats, pathways, null_model, blitz_params)
```

Arguments

`stats`	Numeric vector. The gene level statistic. Needs to be sorted in descending nature and be the same signature the null was calibrated on.
`pathways`	List. One integer vector of index positions per gene set, indexed to R's 1-indexing. Order and duplicates do not matter.
`null_model`	List. The calibrated null from `rs_blitzgsea_calibrate()`.
`blitz_params`	List. The blitzGSEA parameters, see `params_blitzgsea()`. Only `centre` is read here and it has to match what the calibration used.

Value

List with the following elements

- es Numeric vector. Enrichment scores for the gene sets.
- nes Numeric vector. Normalised enrichment scores.
- pvals Numeric vector. Two-sided p-values from the gamma approximation.
- sidak Numeric vector. Sidak-adjusted p-values.
- fdr Numeric vector. Benjamini-Hochberg adjusted p-values.
- size Integer vector. Gene set size after intersection.
- leading_edge List of integer vectors with the leading edge index positions, indexed to R's 1-indexing.

`rs_calc_norm_factors` *Calculate normalisation factors*

Description

[Experimental] edgeR's `calcNormFactors`, via the `edge-rs` crate.

Usage

```
rs_calc_norm_factors(counts, lib_size, norm_method)
```

Arguments

`counts`	Integer or double matrix. Raw counts of genes x samples.
`lib_size`	Numeric vector or NULL. Library size per sample. Pass the pre-filter column sums after filtering genes, as edgeR keeps them. NULL uses the column sums of `counts`.
`norm_method`	String. One of `c("TMM", "TMMwsp", "RLE", "upperquartile", "none")`, case-insensitive.

Value

Numeric vector of normalisation factors, one per sample.

References

Robinson and Oshlack, Genome Biol, 2010

`rs_cell_type_asw` *Calculate cell type silhouette width from an embedding*

Description

[**Experimental**] Average silhouette width on cell type labels, rescaled to `[0, 1]` via `(s + 1) / 2`. Higher values mean cell types stay separated.

Usage

```
rs_cell_type_asw(embedding, labels, max_cells, verbose, seed)
```

Arguments

`embedding`	Numeric matrix. Cells x dimensions.
`labels`	Integer vector. The cell type per cell. The codes need not be 0-based or contiguous.
`max_cells`	Integer or NULL. If not NULL, subsample to this many cells for performance. If NULL, all cells are used.
`verbose`	Boolean. Controls verbosity of the function.
`seed`	Integer. Seed for subsampling reproducibility.

Value

A list with the following items

- per_cell - Per-cell rescaled silhouette scores
- mean_asw - Mean rescaled silhouette width
- median_asw - Median rescaled silhouette width

References

Luecken, et al., Nat Methods, 2022

`rs_cistarget` *Run CisTarget motif enrichment analysis*

Description

[**Experimental**] Core Rust function for motif enrichment analysis using recovery curves. Gene sets are processed in parallel; only motifs with an NES at or above `nes_threshold` are returned.

Usage

```
rs_cistarget(
  rankings,
  gs_list,
  auc_threshold,
  nes_threshold,
  max_rank,
  method,
  n_mean,
  verbose
)
```

Arguments

`rankings`	Integer matrix with motif rankings for genes (genes in rows, motifs in columns). Lower ranks indicate higher regulatory potential.
`gs_list`	List of integer vectors. Each element contains 1-based indices of genes in the gene set (matching row indices in rankings).
`auc_threshold`	Integer. Absolute number of top-ranked genes to use for the AUC calculation (e.g., for 5% of 10000 genes, use 500).
`nes_threshold`	Numeric. Normalised Enrichment Score threshold for filtering significant motifs.
`max_rank`	Integer. Maximum rank to consider for the recovery curves (at most `nrow(rankings)`).
`method`	String. Recovery curve calculation method, one of `c("approx", "icistarget")`. Anything else falls back to `"approx"`.
`n_mean`	Integer. Window size for the smoothing in the approximate method.
`verbose`	Boolean. Report progress per decile of gene sets.

Value

List of lists, one per gene set, each containing

- motif_idx - 1-based column index of the motif in `rankings`.
- nes - Normalised enrichment score.

- auc - Area under the recovery curve.
- rank_at_max - Rank at which the leading edge is reached.
- n_enriched - Number of genes in the leading edge.
- leading_edge - List of 1-based row indices of the leading edge genes, one element per motif.

`rs_compare_knn` *Helper to compare kNN graphs*

Description

[**Experimental**] Compare two kNN graphs and return the distance ratios and overlaps of k-nearest neighbours between them.

Usage

```
rs_compare_knn(knn_data_a, knn_data_b)
```

Arguments

`knn_data_a`	Named list. This contains the kNN data (including distances) of the first kNN graph. This one will be treated as the ground truth
`knn_data_b`	Named list. This contains the kNN data (including distances) of the second kNN graph.

Value

A list with the following elements:

- all_matches - Integer vector. Number of shared neighbours per sample.
- all_ratios - Numerical vector. Ratio of the summed distances (b / a) per sample. Samples whose summed distance in a is ~0 are skipped, so this can be shorter than `all_matches`.
- final_recall - The mean recall across all samples, with a as the ground truth.
- final_ratio - The mean distance ratio across the retained samples.

`rs_constrained_page_rank` *Calculate a constrained page-rank score*

Description

[**Experimental**] This function can be used to get constrained personalised page-rank scores akin to Ruiz, et al. You can provide optionally `sink_nodes` (node types that will force a reset) and/or `sink_edges` (edge types that will force a reset).

Usage

```
rs_constrained_page_rank(
  node_names,
  node_types,
  from,
  to,
  weights,
  edge_type,
  personalised,
  sink_nodes,
  sink_edges
)
```

Arguments

`node_names`	String vector. Name of the graph nodes.
`node_types`	String vector. The node types.
`from`	String vector. The names of the `from` edges from the edge list.
`to`	String vector. The names of the `to` edges from the edge list.
`weights`	Numerical vector. The edge weights from the edge list.
`edge_type`	String vector. The edge types.
`personalised`	Numerical vector. The reset values. They must sum to 1 and be of same length of `node_names`!
`sink_nodes`	Optional string vector. Should these node types be seen as sinks, i.e., the reset occurs when this node is reached.
`sink_edges`	Optional string vector. Shall an automatic reset occur when this edge type is traversed.

Value

The personalised constrained page rank values.

References

Ruiz, et al., Nat Commun, 2021

`rs_constrained_page_rank_list`
Calculate a constrained page-rank score over a list.

Description

[**Experimental**] This function can be used to get constrained personalised page-rank scores akin to Ruiz, et al. You can provide optionally `sink_nodes` (node types that will force a reset) and/or `sink_edges` (edge types that will force a reset). This version can take in a list of personalisation vectors and returns a list as result.

Usage

```
rs_constrained_page_rank_list(
  personalisation_list,
  node_names,
  node_types,
  from,
  to,
  weights,
  edge_type,
  sink_nodes,
  sink_edges
)
```

Arguments

`personalisation_list`	List. The list with the personalisation vectors. The sum must equal to 1, otherwise the function panics!
`node_names`	String vector. Name of the graph nodes.
`node_types`	String vector. The node types.
`from`	String vector. The names of the `from` edges from the edge list.
`to`	String vector. The names of the `to` edges from the edge list.
`weights`	Numerical vector. The edge weights from the edge list.
`edge_type`	String vector. The edge types.
`sink_nodes`	Optional string vector. Should these node types be seen as sinks, i.e., the reset occurs when this node is reached.
`sink_edges`	Optional string vector. Shall an automatic reset occur when this edge type is traversed.

Value

A list of the personalised (constrained) page rank values.

References

Ruiz, et al., Nat Commun, 2021

`rs_contrastive_pca` *Calculate the contrastive PCA*

Description

[**Experimental**] This function calculates the contrastive PCA given a target covariance matrix and the background covariance matrix you wish to subtract. The alpha parameter controls how much of the background covariance you wish to remove. You have the options to return the feature loadings and you can specify the number of cPCAs to return.

Usage

```
rs_contrastive_pca(
  target_covar,
  background_covar,
  target_mat,
  alpha,
  n_pcs,
  return_loadings
)
```

Arguments

`target_covar`	The co-variance matrix of the target data set.
`background_covar`	The co-variance matrix of the background data set.
`target_mat`	The original values of the target matrix. Rows = samples, columns = features.
`alpha`	How much of the background co-variance should be removed.
`n_pcs`	How many contrastive PCs to return
`return_loadings`	Shall the loadings be returned from the contrastive PCA

Value

A list containing:

- factors - The factors of the contrastive PCA, i.e. `target_mat` multiplied by the loadings. Samples x `n_pcs`.
- loadings - The loadings (top eigenvectors) of the contrastive PCA. Features x `n_pcs`. Will be `NULL` if `return_loadings = FALSE`.

`rs_cor` *Calculate the column wise correlations.*

Description

[**Experimental**] Calculates the pairwise correlations of the columns.

Usage

```
rs_cor(x, spearman)
```

Arguments

`x`	R matrix with doubles.
`spearman`	Shall the Spearman correlation be calculated instead of Pearson.

Value

The correlation matrix.

`rs_cor_upper_triangle` *Calculate the column wise correlations and returns the upper triangle*

Description

[**Experimental**] Calculates the correlation matrix of the columns, but returns the upper triangle only as a flat vector.

Usage

```
rs_cor_upper_triangle(x, spearman, shift)
```

Arguments

`x`	R matrix with doubles.
`spearman`	Shall the Spearman correlation be calculated instead of Pearson.
`shift`	Boolean. If `TRUE`, the diagonal is excluded, otherwise it is included.

Value

The upper triangle of the correlation matrix iterating through the rows, with or without the diagonal depending on `shift`.

rs_cor2 — *Calculate the column wise correlations.*

Description

[**Experimental**] Calculates the correlation between the columns of two matrices. The number of rows need to be the same!

Usage

```
rs_cor2(x, y, spearman)
```

Arguments

x	R matrix with doubles.
y	R matrix with doubles.
spearman	Shall the Spearman correlation be calculated instead of Pearson.

Value

The correlation matrix of dimension `ncol(x)` by `ncol(y)`.

rs_cos — *Calculate the column wise cosine similarities*

Description

[**Experimental**] Calculates the pairwise cosine similarity matrix of the columns.

Usage

```
rs_cos(x)
```

Arguments

x	R matrix with doubles.

Value

The cosine similarity matrix.

rs_count_zeroes *Helper to get zero stats from a given matrix*

Description

[**Experimental**] Calculates in a single matrix pass the total number of zeroes, the row zeroes and column zeroes.

Usage

```
rs_count_zeroes(x)
```

Arguments

x	Numeric matrix. The matrix for which to calculate the total zeroes, column and row zeroes.

Value

A list with:

- total_zeroes - Total zeroes in the matrix.
- row_zeroes - Vector with number of zeroes per row.
- col_zeroes - Vector with number of zeroes per column.

rs_cov2cor *Calculates the correlation matrix from the co-variance matrix*

Description

[**Experimental**] Calculates the correlation matrix from a co-variance matrix.

Usage

```
rs_cov2cor(x)
```

Arguments

x	R matrix with doubles that is the co-variance matrix

Value

The correlation matrix.

rs_covariance *Calculate the column-wise co-variance.*

Description

[**Experimental**] Calculates the pairwise co-variance of the columns.

Usage

```
rs_covariance(x)
```

Arguments

x	R matrix with doubles.

Value

The co-variance matrix.

rs_cpm *Counts per million*

Description

[**Experimental**] edgeR's `cpm` on a plain count matrix, via the `edge-rs` crate.

Usage

```
rs_cpm(counts, lib_size, log, prior_count)
```

Arguments

counts	Integer or double matrix. Raw counts of genes x samples.
lib_size	Numeric vector or NULL. Library size per sample, e.g. `lib.size * norm.factors`. NULL uses the column sums.
log	Boolean. Return log2-CPM.
prior_count	Numeric. Prior count added before the log. Ignored if `log = FALSE`.

Value

Numeric matrix of (log2-)CPM values, genes x samples.

`rs_dense_to_upper_triangle` *Generate a vector-based representation of the upper triangle of a matrix*

Description

[**Experimental**] This function generates a vector from the upper triangle of a given symmetric matrix, iterating through the rows. You have the option to remove the diagonal with setting `shift = TRUE`.

Usage

```
rs_dense_to_upper_triangle(x, shift)
```

Arguments

`x`	Numeric matrix. The symmetric matrix to flatten.
`shift`	Boolean. If `TRUE`, the diagonal is excluded, otherwise it is included.

Value

Numeric vector with the upper triangle values.

`rs_differential_cor` *Calculate the column wise differential correlation between two sets of data.*

Description

[**Experimental**] This function calculates the differential correlation based on the Fisher method. For speed purposes, the function will only calculate the differential correlation on the upper triangle of the two correlation matrices.

Usage

```
rs_differential_cor(x_a, x_b, spearman)
```

Arguments

`x_a`	Numeric matrix a, samples x features.
`x_b`	Numeric matrix b, samples x features. Needs the same number of columns as `x_a`.
`spearman`	Boolean. Shall the Spearman correlation be calculated instead of Pearson.

Value

A list containing, one entry per upper-triangle feature pair:

- r_a - The correlation coefficients in the upper triangle of matrix a.
- r_b - The correlation coefficients in the upper triangle of matrix b.
- z_score - The z-scores of the difference in correlation coefficients.
- p_val - The z-scores transformed to two-sided p-values.

`rs_dist` *Calculate the pairwise column distance in a matrix*

Description

[**Experimental**] This function allows to calculate pairwise between all columns the specified distance metric.

Usage

```
rs_dist(x, distance_type)
```

Arguments

`x`	Numerical matrix. The matrix for which to calculate the pairwise column distances.
`distance_type`	String. One of `c("euclidean", "manhattan", "canberra", "cosine", "correlation")`. `"correlation"` is `1 - Pearson r`. Unknown strings raise an error.

Value

The calculated distance matrix.

`rs_edger_ql` *Run the edgeR quasi-likelihood chain on a count matrix*

Description

[**Experimental**] Runs optional `filterByExpr` -> `calcNormFactors` -> `glmQLFit` -> `glmQLFTest`, implemented in Rust via the `edge-rs` crate and gated against edgeR 4.8.2. `legacy = TRUE` adds `estimateDisp` before the fit. The tested axis does not have to be genes: Milo's neighbourhood counts are tested with the same call, with `filter = FALSE`.

Usage

```
rs_edger_ql(counts, design, edger_params)
```

Arguments

`counts`	Numeric matrix. Raw counts of features x samples. Must not be normalised or log-transformed.
`design`	Numeric matrix. The design matrix of samples x coefficients, including the intercept. Needs at least two columns, since the null model has to retain one.
`edger_params`	Named list. The edgeR parameters, see `params_edger_ql()`, plus either `coef` (0-indexed(!) design columns to drop from the null model) or `contrast` (column-major weights with `n_contrasts` columns).

Value

A list with the following elements, all but `features_to_keep` with one entry per kept feature

- features_to_keep - Boolean. Which features survived the filters. Spans the full feature axis of `counts`.
- log_fc - Log2 fold changes of the tested coefficient or contrast.
- log_cpm - Average log2 counts per million.
- f_stat - The quasi-likelihood F statistic.
- p_values - Raw p-values.
- fdr - Benjamini-Hochberg adjusted p-values.

References

Chen, Lun and Smyth, F1000Research, 2016

`rs_fast_auc` *Fast AUC calculation*

Description

[**Experimental**] This function calculates rapidly AUCs based on an approximation: the fraction of `iters` random (positive, negative) pairs, drawn with replacement, where the positive score is larger.

Usage

```
rs_fast_auc(pos_scores, neg_scores, iters, seed)
```

Arguments

`pos_scores`	The scores of your hits.
`neg_scores`	The scores of your non-hits.
`iters`	Number of iterations to run the function for. Recommended size: 10000L.
`seed`	Seed.

Value

The approximate AUC.

`rs_fast_ica` *Run the Rust implementation of fast ICA.*

Description

[**Experimental**] This function serves as a wrapper over the fast ICA implementations in Rust. It assumes a whitened matrix and also an initialised w_init.

Usage

```
rs_fast_ica(whiten, w_init, ica_type, ica_params)
```

Arguments

`whiten`	Numerical matrix. The whitened matrix.
`w_init`	Numerical matrix. The initial unmixing matrix. ncols need to be equal to nrows of whiten.
`ica_type`	String. One of `c("logcosh", "exp")`. Any other string defaults to `"logcosh"`.
`ica_params`	A list containing: • maxit - Integer. Maximum number of iterations for ICA. • alpha - Float. The alpha parameter for the logcosh version of ICA. Should be between 1 to 2. • max_tol - Maximum tolerance of the algorithm • verbose - Verbosity of the function, i.e., shall individual iters be shown. If the list is empty or the expected elements are not found, default values are used.

Value

A list with the following items:

- mixing - The mixing matrix for subsequent usage.
- converged - Boolean. Did the best tolerance reached fall below `max_tol`.

rs_fdr_adjustment *Calculate a BH-based FDR*

Description

[**Experimental**] Rust implementation that will be faster if you have a terrifying amount of p-values to adjust.

Usage

```
rs_fdr_adjustment(pvals)
```

Arguments

pvals	Numeric vector. The p-values you wish to adjust.

Value

The Benjamini-Hochberg adjusted p-values.

rs_filter_by_expr *Filter lowly expressed genes*

Description

[**Experimental**] edgeR's `filterByExpr`, via the `edge-rs` crate.

Usage

```
rs_filter_by_expr(
  counts,
  group,
  lib_size,
  min_count,
  min_total_count,
  min_prop
)
```

Arguments

counts	Integer or double matrix. Raw counts of genes x samples.
group	Integer vector or NULL. 1-based group code per sample (e.g. `as.integer(factor(x))`). If NULL, all samples form one group.
lib_size	Numeric vector or NULL. Library size per sample. NULL uses the column sums.

`min_count` Numeric. Minimum count in the median-sized library.

`min_total_count`
Numeric. Minimum total count across all samples.

`min_prop` Numeric. Fraction of the smallest group size beyond edgeR's `large.n` that still has to express the gene.

Value

Boolean vector, one per gene. `TRUE` if the gene is kept.

References

Chen, Lun and Smyth, F1000Research, 2016

`rs_gene_trends` *Fit Palantir gene trends over pseudotime*

Description

[**Experimental**] Selects the cells belonging to each branch from the fate probabilities, then fits a landmark Gaussian process with a Matern-5/2 kernel per branch. This is the Mellon-based estimator of the reference, not the legacy GAM.

Whatever expression matrix is handed over is what gets fitted. Nothing here imputes; that decision belongs to the caller.

The defaults are prior-dominated. Palantir's pseudotime is min-max scaled to `[0, 1]`, so a `length_scale` of `1.0` spans the whole domain and a `sigma` of `1.0` sits at roughly the signal scale of log-normalised expression. That resolves almost any gene into a smooth monotone or single-peaked curve. Shorten `length_scale` before believing a bump.

Usage

```
rs_gene_trends(
  expression,
  pseudotime,
  branch_probs,
  branch_params,
  gene_trend_params
)
```

Arguments

`expression` Numerical matrix of cells x genes. All cells, in the same row order as `pseudotime`, not just the branch members.

`pseudotime` Numerical vector. Pseudotime per cell.

`branch_probs` Numerical matrix of cells x fates with the fate probabilities. Rows need not sum to one.

`branch_params` List. Parameter list, see `params_sc_branch_selection()`.

`gene_trend_params`
List. Parameter list, see `params_sc_gene_trends()`.

Value

A list with

- trends - List of numerical matrices, one per branch, each of resolution x genes.
- grids - List of numerical vectors with the pseudotime grid per branch, running from the branch minimum to its maximum.
- branch_cells - List of integer vectors with the cell indices (0-indexed!) selected for each branch.
- n_cells - Integer vector with the cell count per branch.
- jitter_used - Numerical vector with the jitter each branch's Cholesky needed.

References

Setty, et al., Nat. Biotechnol., 2019.

`rs_generate_ligand_target_influence`

Generate the ligand to target influence matrices

Description

[Experimental] Helper function to generate the ligand to target influence matrix for the NicheNet like approach.

Usage

```
rs_generate_ligand_target_influence(
  ligand_seeds,
  ppi_network,
  grn_network,
  n_nodes,
  params
)
```

Arguments

`ligand_seeds` List of integer vectors. The 0-indexed seed node(s) per ligand or ligand combination.

`ppi_network` Named list. Contains the PPI network with the ligand to receptor to signalling to TFs. Must contain `from` and `to` (0-indexed node indices) and `weight`.

`grn_network`	Named list. Contains the gene regulatory network with the TF to target gene network. Must contain `from` and `to` (0-indexed node indices) and `weight`.
`n_nodes`	Integer. Number of total nodes.
`params`	Named list. The ligand-target diffusion parameters.

Value

A dense matrix of ligands x `n_nodes`, rows in `ligand_seeds` order, with the ligand to target influence scores.

`rs_geom_elim_fgsea_simple` *Run fgsea simple method for gene ontology with elimination method*

Description

[**Experimental**]

Usage

```
rs_geom_elim_fgsea_simple(
  stats,
  levels,
  go_obj,
  gsea_params,
  elim_threshold,
  iters,
  seed
)
```

Arguments

`stats`	Named numerical vector. Needs to be sorted. The gene level statistics.
`levels`	A character vector representing the levels to iterate through. The order will be the one the iterations are happening in.
`go_obj`	The `GeneOntologyElim` S7 class. See `GeneOntologyElim()`.
`gsea_params`	List. The GSEA parameters, see `params_gsea()` wrapper function. This function generates a list containing: • min_size - Integer. Minimum size for the gene sets. • max_size - Integer. Maximum size for the gene sets. • gsea_param - Float. The GSEA parameter. Defaults to `1.0`. • sample_size - Integer. Number of samples to iterate through for the multi-level implementation of fgsea.

- eps - Float. Boundary for calculating the p-value. Used for the multi-level implementation of fgsea.

`elim_threshold` p-value below which the elimination procedure shall be applied to the ancestors.

`iters` Integer. Number of random permutations for the fgsea simple method to use

`seed` Integer. For reproducibility purposes.

Value

List with the following elements

- go_id The name of the tested gene ontology identifier.
- es The enrichment scores for the pathway
- nes The normalised enrichment scores for the pathway. Can be `NA`.
- size The pathway sizes (after elimination!).
- pvals The p-values for this pathway based on permutation testing
- n_more_extreme Number of times the enrichment score was bigger or smaller than the permutation (pending sign).
- le_zero Number of times the permutation was less than zero.
- ge_zero Number of times the permutation was greater than zero.
- leading_edge A list of the 1-based index positions (in `stats`) of the leading edge genes for this given GO term.

`rs_gower_dist` *Calculates the Gower distance for a given matrix*

Description

[**Experimental**] Calculates the pairwise Gower distance between the rows. Continuous columns contribute the range-normalised absolute difference, categorical columns a simple mismatch.

Usage

```
rs_gower_dist(x, is_cat)
```

Arguments

`x` Numerical matrix. Converted matrix of continuous and categorical variables as numerical values. Rows = samples, columns = features.

`is_cat` Logical vector of length `ncol(x)`. Which of the columns represent categorical values.

Value

The Gower distance matrix between the rows, values in `[0, 1]`.

`rs_graph_connectivity` *Calculate graph connectivity per cell type*

Description

[**Experimental**] For each cell type, the fraction of its cells in the largest connected component of the kNN graph restricted to that cell type. 1 means every cell type forms one connected piece. Edge direction is ignored.

Usage

```
rs_graph_connectivity(knn_mat, labels)
```

Arguments

`knn_mat`	Integer matrix. The rows represent the cells and the columns the neighbour indices (0-indexed!).
`labels`	Integer vector. The cell type per cell. The codes need not be 0-based or contiguous.

Value

A list with the following items

- per_label - Connectivity per cell type, in order of first appearance in `labels`.
- mean - Mean connectivity.
- median - Median connectivity.

References

Luecken, et al., Nat Methods, 2022

`rs_gse_geom_elim` *Run hypergeometric enrichment over the gene ontology*

Description

[**Experimental**] This function implements a Rust version of the gene ontology enrichment with elimination: the starting point are the leaves of the ontology and hypergeometric tests will first be conducted there. Should the hypergeometric test p-value be below a certain threshold, the genes of that gene ontology term will be removed from all ancestors.

Usage

```
rs_gse_geom_elim(
  target_genes,
  levels,
  go_obj,
  gene_universe_length,
  min_genes,
  elim_threshold,
  min_overlap,
  fdr_threshold
)
```

Arguments

`target_genes`	A character vector representing the target gene set.
`levels`	A character vector representing the levels to iterate through. The order will be the one the iterations are happening in.
`go_obj`	The `GeneOntologyElim` S7 class. See `GeneOntologyElim()`.
`gene_universe_length`	The length of the gene universe.
`min_genes`	number of minimum genes for the gene ontology term to be tested.
`elim_threshold`	p-value below which the elimination procedure shall be applied to the ancestors.
`min_overlap`	Optional minimum overlap threshold.
`fdr_threshold`	Optional fdr threshold.

Value

A list containing (for the terms passing the thresholds):

- go_ids - The gene ontology identifier.
- pvals - The calculated p-values.
- fdr - The calculated FDRs.

- odds_ratios - The calculated odds ratios.
- hits - The size of the overlap.
- gene_set_lengths - The length of the gene sets.

`rs_gse_geom_elim_list` *Run hypergeometric enrichment a list of target genes over the gene ontology*

Description

[**Experimental**] This function implements a Rust version of the gene ontology enrichment with elimination: the starting point are the leaves of the ontology and hypergeometric tests will first be conducted there. Should the hypergeometric test p-value be below a certain threshold, the genes of that gene ontology term will be removed from all ancestors. This function is designed to leverage Rust-based threading for parallel processing of a list of target genes.

Usage

```
rs_gse_geom_elim_list(
  target_genes_list,
  levels,
  go_obj,
  gene_universe_length,
  min_genes,
  elim_threshold,
  min_overlap,
  fdr_threshold
)
```

Arguments

`target_genes_list`	A list of target genes against which to run the method.
`levels`	A character vector representing the levels to iterate through. The order will be the one the iterations are happening in.
`go_obj`	The `GeneOntologyElim` S7 class. See `GeneOntologyElim()`.
`gene_universe_length`	The length of the gene universe.
`min_genes`	number of minimum genes for the gene ontology term to be tested.
`elim_threshold`	p-value below which the elimination procedure shall be applied to the ancestors.
`min_overlap`	Optional minimum overlap threshold.
`fdr_threshold`	Optional fdr threshold.

Value

A list containing (results of all target sets concatenated):

- go_ids - The gene ontology identifier.
- pvals - The calculated p-values.
- fdr - The calculated FDRs.
- odds_ratios - The calculated odds ratios.
- hits - The size of the overlap.
- gene_set_lengths - The length of the gene sets.
- no_test - The number of tests for that target set that passed the thresholds.

`rs_gsva` *Rust version of the GSVA algorithm*

Description

[**Experimental**] Rust-based implementation of the popular GSVA algorithm. Has further performance optimisations compared to the original implementation.

Usage

```
rs_gsva(exp, gs_list, tau, kernel, max_diff, abs_rank, timings)
```

Arguments

`exp`	Numerical matrix. The expression matrix with rows = genes, and columns = samples
`gs_list`	List. A list containing the 0-based indices of the pathway genes. See `rs_prepare_gsva_gs()`.
`tau`	Float. Tau parameter, usual recommendation is to use `1.0` here. Larger values emphasise the tails more.
`kernel`	String. One of `c("gaussian", "poisson", "none")`. The kernel function to use. Unknown strings default to `"gaussian"`.
`max_diff`	Boolean. Scoring mode: `TRUE` = difference, `FALSE` = larger absolute value
`abs_rank`	Boolean. If `TRUE` = pos-neg, `FALSE` = pos+neg
`timings`	Boolean. Prints timings from the algorithm.

Value

Returns a matrix of gene set ES scores x samples.

`rs_h5ad_data` *Load in h5ad data via Rust*

Description

[**Experimental**] Loads in h5ad data within Rust and automatically converts the data into CSR with cells x genes.

Usage

```
rs_h5ad_data(f_path, cs_type, nrows, ncols, cell_quality, slot, verbose)
```

Arguments

`f_path`	File path. The path to the h5ad file.
`cs_type`	String. One of `c("csr", "csc")`. How the data is stored in the file. Other values raise an error.
`nrows`	Integer. Number of rows in the file.
`ncols`	Integer. Number of columns in the file.
`cell_quality`	List. Specifying the cell quality. Please refer to `params_sc_min_quality()`.
`slot`	String. In which slot the raw data can be found. One of `c("X", "raw", "layers.counts")`. Unknown strings default to `"X"`.
`verbose`	Boolean. Controls verbosity of the function

Value

A list with the CSR data (cells x genes) of the cells and genes passing `cell_quality`:

- data - The counts of the sparse matrix.
- indices - The 0-based gene indices of the sparse matrix.
- indptr - The index pointers of the sparse matrix.
- no_genes - No of genes in the sparse matrix (i.e., ncol).
- no_cells - No of cells in the sparse matrix (i.e., nrow).

rs_hamming_dist *Calculates the Hamming distance between categorical columns*

Description

[**Experimental**] Calculates the pairwise Hamming distance between the columns, i.e. the fraction of rows in which two columns differ.

Usage

```
rs_hamming_dist(x)
```

Arguments

x	Integer matrix. The integers represent the factor data.

Value

The Hamming distance matrix between the columns, values in `[0, 1]`.

rs_harmony *Harmony batch correction in Rust*

Description

[**Experimental**] This function implements the Harmony algorithm from Korsunsky et al., 2019.

Usage

```
rs_harmony(pca, harmony_params, batch_labels, seed, verbose)
```

Arguments

pca	Numerical matrix, i.e., the PCA matrix you want to correct.
harmony_params	List. The parameters for the Harmony algorithm.
batch_labels	List. Each element needs to be a 0-indexed integer vector, one per batch variable you wish to regress out.
seed	Integer. Seed for reproducibility purposes.
verbose	Integer. `0L` - quiet; `1L` - normal verbosity; `2L` - detailed verbosity.

Value

Numerical matrix, cells x dimensions, with the batch-corrected Harmony embedding.

rs_harmony_v2 *Harmony batch correction in Rust (version 2)*

Description

[**Experimental**] This function implements the version 2 Harmony algorithm from Patikas, et al., 2026.

Usage

```
rs_harmony_v2(pca, harmony_params, batch_labels, seed, verbose)
```

Arguments

`pca`	Numerical matrix, i.e., the PCA matrix you want to correct.
`harmony_params`	List. The parameters for the Harmony (v2) algorithm.
`batch_labels`	List. Each element needs to be a 0-indexed integer vector, one per batch variable you wish to regress out.
`seed`	Integer. Seed for reproducibility purposes.
`verbose`	Integer. `0L` - quiet; `1L` - normal verbosity; `2L` - detailed verbosity.

Value

Numerical matrix, cells x dimensions, with the batch-corrected Harmony (v2) embedding.

rs_hedges_g *Calculate the Hedge's G effect*

Description

[**Experimental**] Calculates the Hedge's G effect for two sets of matrices. The function assumes that rows = samples and columns = features.

Usage

```
rs_hedges_g(mat_a, mat_b, small_sample_correction)
```

Arguments

`mat_a`	The matrix of samples and features in grp A for which to calculate the Hedge's G effect.
`mat_b`	The matrix of samples and features in grp B for which to calculate the Hedge's G effect.
`small_sample_correction`	Shall the small sample correction be applied.

Value

A list with the following items:

- effect_sizes - Hedge's G effect size per feature.
- standard_errors - Standard error of the effect size per feature.

`rs_hypergeom_test` *Run a single hypergeometric test.*

Description

[**Experimental**] Given a set of target genes, this is a Rust implementation of an hypergeometric test testing for overenrichment of the target genes in the gene sets.

Usage

```
rs_hypergeom_test(
  target_genes,
  gene_sets,
  gene_universe,
  min_overlap,
  fdr_threshold
)
```

Arguments

`target_genes`	String vector. Represents the target gene set.
`gene_sets`	List. Contains the strings that represent the gene sets to test against.
`gene_universe`	String vector. The features representing the gene universe from which the target genes and gene sets are sampled from.
`min_overlap`	Optional integer. Shall a filter be applied on the minimum of overlapping genes.
`fdr_threshold`	Optional float. Shall a filter be applied for the maximum tolerated FDR.

Value

A list containing (only for the gene sets passing the filters):

- pvals - The p-values from the hypergeometric test
- odds_ratios - The calculated odds ratios
- hits - The size of the overlap
- gene_set_lengths - The length of the gene sets.
- fdr - The FDR calculated across all gene sets (before filtering).
- to_keep - 1-based indices of the gene sets that passed the (optional) thresholds.

`rs_hypergeom_test_list`

Run a hypergeometric test over a list of target genes

Description

[**Experimental**] Given a list of target gene sets, this function will test for each of the individual target gene sets the hypergeometric enrichment against the specified gene sets.

Usage

```
rs_hypergeom_test_list(
  target_genes_list,
  gene_sets,
  gene_universe,
  min_overlap,
  fdr_threshold
)
```

Arguments

`target_genes_list`	A list of character vectors, each representing a target gene set.
`gene_sets`	A list of strings that represent the gene sets to test against.
`gene_universe`	A character vector representing the gene universe from which the target genes and gene sets are sampled from.
`min_overlap`	Optional integer. Shall a filter be applied on the minimum of overlapping genes.
`fdr_threshold`	Optional float. Shall a filter be applied for the maximum tolerated FDR.

Value

A list containing (results of all target sets concatenated, only for the tests passing the filters):

- pvals - The p-values from the hypergeometric test.
- fdr - The FDRs for each target set calculated across all gene sets.
- odds_ratios - The calculated odds ratios
- hits - The size of the overlap between the target gene set and individual gene sets.
- gene_set_lengths - The length of the gene sets.
- to_keep - 1-based indices (into `gene_sets`) of the tests that passed.
- tests_passed - How many tests passed the filter criteria for that target set.

`rs_ica_iters` *Run ICA over a given no_comp with random initialisations of w_init*

Description

[**Experimental**] This function implements a stabilised ICA like algorithm in Rust. Briefly, it generates random w_init matrices (total number being no_random_init) and runs ICA given the x_processed and k data over these. The function returns combined S from the different runs and a boolean vector indicating if this specific run converged.

Usage

```
rs_ica_iters(x1, k, no_comp, no_random_init, ica_type, random_seed, ica_params)
```

Arguments

`x1`	Numerical matrix. The processed matrix (but not yet whitened!)
`k`	Numerical matrix. The whitening matrix. Needs at least `no_comp` rows, only the first `no_comp` are used.
`no_comp`	Integer. Number of independent components to return.
`no_random_init`	Integer. Number of random initialisations to test.
`ica_type`	String. One of `c("logcosh", "exp")`.
`random_seed`	Integer. Seed for randomisations. Run `i` uses `random_seed + i`.
`ica_params`	A list containing: • maxit - Integer. Maximum number of iterations for ICA. • alpha - Float. The alpha parameter for the logcosh version of ICA. Should be between 1 to 2. • max_tol - Maximum tolerance of the algorithm • verbose - Verbosity of the function, i.e., shall individual iters be shown. If the list is empty or the expected elements are not found, default values are used.

Value

A list containing:

- s_combined - The combined matrices for S. Dimensions are nrows = features; and ncols = ncomp * no_random_init.
- converged - Boolean vector indicating if the respective run reached convergence. Length = no_random_init

`rs_ica_iters_cv` *Run ICA with cross-validation and random initialisation*

Description

[**Experimental**] This function will split the data into `no_folds` and apply ICA with `no_random_init` random initialisations over each fold.

Usage

```
rs_ica_iters_cv(
  x,
  no_comp,
  no_folds,
  no_random_init,
  ica_type,
  random_seed,
  ica_params
)
```

Arguments

`x`	Numeric matrix. The processed data (no whitening function has been applied yet.)
`no_comp`	Integer. Number of components to test for.
`no_folds`	Integer. Number of folds to use for the cross-validation.
`no_random_init`	Integer. Number of random initialisations per fold.
`ica_type`	String. One of `c("logcosh", "exp")`.
`random_seed`	Integer. For reproducibility.
`ica_params`	A list containing: • maxit - Integer. Maximum number of iterations for ICA. • alpha - Float. The alpha parameter for the logcosh version of ICA. Should be between 1 to 2. • max_tol - Maximum tolerance of the algorithm • verbose - Verbosity of the function, i.e., shall individual iters be shown. If the list is empty or the expected elements are not found, default values are used.

Value

A list containing:

- s_combined - The combined matrices for S. Dimensions are nrows = features; and ncols = no_comp * no_random_init * no_folds.
- converged - Boolean vector indicating if the respective run reached convergence. Length = no_random_init * no_folds

---

`rs_jaccard_row_integers` *Calculate rapidly Jaccard similarities between rows*

Description

[**Experimental**] Helper function to quickly calculate the Jaccard similarity between matching rows of the two matrices. Each row is treated as a set of integers (duplicates removed).

Usage

```
rs_jaccard_row_integers(data_1, data_2)
```

Arguments

`data_1`	Integer matrix. The first matrix to compare.
`data_2`	Integer matrix. The second matrix to compare. Needs the same number of rows as `data_1`.

Value

The Jaccard similarity averaged over the rows.

`rs_kbet` *Calculate kBET type scores*

Description

[**Experimental**] The function takes in a kNN matrix and a batch vector indicating which cell belongs to which batch. For the neighbourhood of each cell, a chi-square test checks whether the batch proportions differ from the overall batch proportions (with Yates' correction for two batches). Good mixing means few cells with significant differences; bad mixing means many.

Usage

```
rs_kbet(knn_mat, batch_vector, verbose)
```

Arguments

`knn_mat`	Integer matrix. The rows represent the cells and the columns the neighbour indices (0-indexed!).
`batch_vector`	Integer vector. The batch per cell. The codes need not be 0-based or contiguous.
`verbose`	Boolean. Controls verbosity of the function.

Value

A list with the following items

- pval - Per-cell p-values from the chi-square test.
- chi_square_stats - Per-cell chi-square statistics.
- mean_chi_square - The mean chi-square value.
- median_chi_square - The median chi-square value.

`rs_knn_label_propagation`

kNN label propagation

Description

[**Experimental**] The function is a helper function to do kNN label propagation. This can be useful for semi-supervised tasks. It implements the label spreading method.

Usage

```
rs_knn_label_propagation(
  from,
  to,
  one_hot_encoding,
  label_mask,
  weights,
  label_prop_params
)
```

Arguments

`from`	Integer vector. 1-based source node indices for each edge.
`to`	Integer vector. 1-based target node indices for each edge. Must be the same length as `from`.
`one_hot_encoding`	Integer matrix. Each row represents a sample, the columns the one-hot encodings. Everything 0 denotes the unlabelled data.

`label_mask` Boolean vector. Which of the samples do not have a label. Needs to be same length as `nrow(one_hot_encoding)`.

`weights` Optional numeric vector. Edge weights for each pair in `from`/`to`. Must have the same length as `from`. If `NULL`, all edges are treated as unweighted.

`label_prop_params`
List. Named list of parameters with the following optional fields (defaults in parentheses):

- `alpha` numeric, spreading strength (0.9)
- `iter` integer, max iterations (100)
- `tolerance` numeric, convergence threshold (1e-6)
- `symmetrise` logical, symmetrise the graph (FALSE)
- `symmetry_strategy` character, one of `"average"`, `"min"`, `"max"` (`"average"`). Only used for weighted graphs.
- `max_hops` integer, restrict spreading radius (unrestricted)

Value

The matrix with the probabilities of being of a certain class. Same shape as `one_hot_encoding`.

`rs_knn_mat_to_edge_list`

Flatten kNN matrix to edge list

Description

[**Experimental**] Helper function to leverage Rust to transform a kNN matrix into a flat edge list.

Usage

```
rs_knn_mat_to_edge_list(knn_mat, one_index)
```

Arguments

`knn_mat` Integer matrix. Rows represent the samples and the columns the 0-based indices of the k-nearest neighbours.

`one_index` Boolean. Shall 1-based indices be returned.

Value

A flat vector representing the edge list, alternating from and to, i.e. `c(from_1, to_1, from_2, to_2, ...)`.

`rs_leiden_clustering` *Rust implementation of Leiden and Louvain clustering*

Description

[**Experimental**] Modularity-based Leiden (or Louvain, with `refine = FALSE`) on an undirected weighted graph. Mirrors `igraph::cluster_leiden(objective_function = "modularity")`. The result depends on the seed only, not on the number of threads.

Usage

```
rs_leiden_clustering(from, to, weights, n_nodes, leiden_params)
```

Arguments

`from`	Integer vector. 1-based source node of each edge.
`to`	Integer vector. 1-based target node of each edge. Same length as `from`.
`weights`	Numeric vector. Edge weights, same length as `from`.
`n_nodes`	Integer. Number of nodes in the graph.
`leiden_params`	List. Named list with the optional fields (defaults in parentheses): • `resolution` numeric, modularity resolution (1.0) • `beta` numeric, randomness of the refinement merges (0.01) • `n_iterations` integer, full Leiden iterations (2) • `refine` logical, `TRUE` for Leiden, `FALSE` for Louvain (TRUE) • `seed` integer, random seed (42)

Value

Integer vector with the 1-based community of each node; `1` is the largest community.

`rs_ligand_activity_scores`
Calculate the NicheNet ligand activity scores

Description

[**Experimental**]

Usage

```
rs_ligand_activity_scores(ligand_influence, in_gene_sets)
```

Arguments

`ligand_influence`	A ligand x background genes matrix that measures the ligand to target gene influence.
`in_gene_sets`	List of logical vectors, one per gene set, each of length `ncol(ligand_influence)`. Genes of interest are `TRUE`, the background genes `FALSE`.

Value

A list with one element per gene set, each a list of per-ligand vectors (`NaN` where the metric is undefined):

- `auroc` - The Area Under the Receiver Operating Characteristic.
- `aupr` - The Area Under the Precision-Recall curve.
- `aupr_corrected` - The corrected AUPR.
- `pearson` - The Pearson correlations.
- `spearman` - The Spearman correlations.

`rs_limma_voom` *Run the limma linear model chain on a count matrix*

Description

[**Experimental**] Runs optional `filterByExpr` -> `calcNormFactors` -> `voomLmFit` (or limma-trend) -> `contrasts.fit` -> `eBayes` -> `topTable` for one coefficient or contrast, implemented in Rust via the `edge-rs` crate and gated against limma 3.66.0.

Usage

```
rs_limma_voom(counts, design, lib_size, limma_params)
```

Arguments

`counts`	Integer or double matrix. Raw counts of genes x samples. Must not be normalised or log-transformed.
`design`	Numeric matrix. The design matrix of samples x coefficients, including the intercept. Must be full rank.
`lib_size`	Numeric vector or NULL. Library size per sample. NULL uses the column sums of `counts`. Pass the column sums from before gene filtering to match edgeR, which keeps those on a subset `DGEList`.
`limma_params`	Named list. The limma parameters, see `params_limma_voom()`, plus either `coef` (a single 0-indexed(!) design column) or `contrast` (column-major weights with `n_contrasts` columns).

Value

A list with the following elements, all but `features_to_keep` with one entry per kept gene, in input order

- features_to_keep - Boolean. Which genes survived the filters. Spans the full gene axis of `counts`.
- log_fc - Log2 fold changes of the tested coefficient or contrast.
- ci_lower - Lower end of the 95% confidence interval on `log_fc`.
- ci_upper - Upper end of the 95% confidence interval on `log_fc`.
- ave_expr - Average log2 counts per million.
- t_stat - Moderated t statistic.
- p_values - Raw p-values.
- fdr - Benjamini-Hochberg adjusted p-values.
- b_stat - Log-odds of differential expression.

References

Law, et al., Genome Biol, 2014; Smyth, Stat Appl Genet Mol Biol, 2004

`rs_lisi` *Calculate LISI scores on any label*

Description

[**Experimental**] Computes the Local Inverse Simpson's Index on the kNN graph: the effective number of labels in each cell's neighbourhood. On batch labels this is iLISI (higher is better mixing), on cell type labels cLISI (lower is better separation). Both come back rescaled to `[0, 1]`, higher is better, as in scIB.

Usage

```
rs_lisi(knn_mat, knn_dist, labels, perplexity, verbose)
```

Arguments

`knn_mat`	Integer matrix. The rows represent the cells and the columns the neighbour indices (0-indexed!).
`knn_dist`	Numeric matrix or NULL. The kNN distances, same shape as `knn_mat`. If provided, neighbours are weighted with a perplexity-calibrated Gaussian kernel as in Korsunsky et al.; if NULL, neighbours are weighted uniformly.
`labels`	Integer vector. The label (batch or cell type) per cell. The codes need not be 0-based or contiguous.
`perplexity`	Numeric or NULL. Perplexity for the weighted version. NULL defaults to 30; values above k are clamped to k. Ignored if `knn_dist` is NULL.
`verbose`	Boolean. Controls verbosity of the function.

Value

A list with the following items

- per_cell - Per-cell LISI scores
- mean_lisi - Mean LISI
- median_lisi - Median LISI
- n_labels - Number of distinct labels
- ilisi_norm - Median LISI rescaled as iLISI, `(median - 1) / (n - 1)`
- clisi_norm - Median LISI rescaled as cLISI, `(n - median) / (n - 1)`

References

Korsunsky, et al., Nat Methods, 2019; Luecken, et al., Nat Methods, 2022

`rs_mad_outlier` *Calculate MAD outlier detection in Rust.*

Description

[Experimental]

Usage

```
rs_mad_outlier(x, threshold, direction)
```

Arguments

`x`	Numerical vector to test.
`threshold`	Numeric. Number of (unscaled) MADs from the median in either direction that is acceptable.
`direction`	String. One of `c("below", "above", "twosided")`. Shall the outlier direction be done for values below the threshold, above the threshold or in both directions. Unknown strings default to twosided tests.

Details

Should you provide an empty vector, the function will return an empty boolean and a threshold of 0.

Value

A list with the following items:

- outlier - Boolean vector if element is an outlier
- threshold - Applied margin, i.e. `threshold * MAD`.

rs_mc_aucell *Calculate AUCell in Rust (for meta cells)*

Description

[**Experimental**] The function will take in a list of gene set indices (0-indexed!) and calculate an AUCell type statistic. Three options here: the recovery-curve AUC of Aibar, et al. (the actual AUCell statistic), an AUC derived from the Mann-Whitney statistic, or average precision. This version works on MetaCell counts which are stored in memory directly. Genes are ranked within each meta cell on the second layer, an `f32` cast of the supplied counts.

Usage

```
rs_mc_aucell(sparse_data, gs_list, aucell_params, verbose)
```

Arguments

`sparse_data`	A named list that needs to have `data`, `indptr`, `indices`, `nrow`, `ncol` and `cs_type`. Shape is (metacells, genes).
`gs_list`	List. List with the gene set indices (0-indexed!) of the genes of interest.
`aucell_params`	List. The AUCell parameters, see `params_sc_aucell()`.
`verbose`	Integer. `0L` - quiet; `1L` - normal verbosity; `2L` - detailed verbosity.

Value

A matrix of meta cells x gene sets with the values representing the AUC.

rs_mc_dialogue *Run DIALOGUE over meta cells*

Description

[**Experimental**] The meta cell entry point into DIALOGUE, see `rs_dialogue_sc()` for what the method does. This is a shim rather than a second implementation: everything DIALOGUE asks of the expression matrix is per-gene, so the in-memory matrix is wrapped as a gene-major reader and the same core runs.

Only the normalised layer is ever read, so `sparse_data` has to carry the normalised counts. The `data` layer is cast to integers on the way in and then goes unused.

Meta cells are already aggregates, so the sample a meta cell belongs to has to be unambiguous: build them within samples, not across them. The random intercept in stage two is over samples, and a meta cell straddling two of them has no well-defined level.

Usage

```
rs_mc_dialogue(
  sparse_data,
  cell_type_indices,
  features,
  sample_ids,
  cell_quality,
  gene_indices,
  dialogue_params,
  verbose
)
```

Arguments

`sparse_data`	A named list that needs to have `data`, `indptr`, `indices`, `cs_type`, `nrow` and `ncol`, holding the *normalised* meta cell counts with shape (metacells, genes).
`cell_type_indices`	List of integer vectors. 0-indexed(!) positions of the meta cells belonging to each cell type. At least two cell types are needed.
`features`	List of numeric matrices, one per cell type, shaped `n_metacells_in_type x k_i` with rows aligned to `cell_type_indices`. Needs at least two columns per cell type.
`sample_ids`	Integer vector. 0-indexed(!) sample code per meta cell, over all meta cells rather than per cell type.
`cell_quality`	Numeric vector. Quality covariate per meta cell, indexed the same way as `sample_ids`.
`gene_indices`	Integer vector. 0-indexed(!) positions of the genes to consider when building signatures.
`dialogue_params`	Named list. Contains the DIALOGUE parameters across all three stages, see `params_dialogue_pmd()`, `params_dialogue_hlm()` and `params_dialogue_refine()`. The three blocks share one flat list.
`verbose`	Integer. `0L` - quiet; `1L` - normal verbosity; `2L` - detailed verbosity.

Value

A list, identical in shape to `rs_dialogue_sc()`.

References

Jerby-Arnon & Regev, Nature Biotechnology, 2022

`rs_mc_hotspot_autocor` *Calculate gene spatial auto-correlations (for meta cells)*

Description

[**Experimental**] This function implements the HotSpot auto-correlation functionality and will return to what extent a given gene shows auto-correlation in the kNN-graph over the meta cells. For details see DeTomaso, et al. This version works on MetaCell counts which are stored in memory directly. There is no streaming variant: streaming bounds disk re-reads, which is not a problem an in-memory matrix has.

Usage

```
rs_mc_hotspot_autocor(
  sparse_data,
  embd,
  knn_data,
  hotspot_params,
  cells_to_keep,
  genes_to_use,
  verbose,
  seed
)
```

Arguments

`sparse_data`	A named list that needs to have `data`, `indptr`, `indices`, `nrow`, `ncol` and `cs_type`. Shape is (metacells, genes) and the data are the raw counts.
`embd`	Numerical matrix. The embedding matrix from which to generate the kNN graph. Needs one row per entry of `cells_to_keep`.
`knn_data`	Optional list. This contains pre-computed kNN data (`indices`, `dist`, `k`) and the `dist_metric` it was built with. The user has to ensure consistency! If provided, this will be used rather than a graph built from the parameter list.
`hotspot_params`	List. The HotSpot parameter list. The kNN parameters are only read when no `knn_data` is provided.
`cells_to_keep`	Integer vector. 0-index vector indicating which meta cells to include in the analysis. Ensure that this is of same order/length as the embedding matrix.
`genes_to_use`	Integer vector. 0-index vector indicating which genes to include.
`verbose`	Integer. `0L` - quiet; `1L` - normal verbosity; `2L` - detailed verbosity.
`seed`	Integer. Random seed for reproducibility.

Value

A list with the following elements.

- gene_idx - 0-based integer indicating the gene index.
- gaerys_c - Geary's C calculation for the autocorrelation coefficient.
- z_score - Z-score of the auto-correlation.
- pval - P-value derived from the Z-score.
- fdr - False discovery rate based on the p-value.

References

DeTomaso, et al., Cell Systems, 2021

---

rs_mc_hotspot_gene_cor

Calculate gene to gene spatial correlations (for meta cells)

Description

[**Experimental**] This function implements the HotSpot gene <> gene local correlation functionality from HotSpot, see DeTomaso, et al. This version works on MetaCell counts which are stored in memory directly.

Three dense metacells x genes blocks are live at once, so keep `genes_to_use` to the panel actually of interest rather than the whole transcriptome.

Usage

```
rs_mc_hotspot_gene_cor(
  sparse_data,
  embd,
  knn_data,
  hotspot_params,
  cells_to_keep,
  genes_to_use,
  verbose,
  seed
)
```

Arguments

`sparse_data`	A named list that needs to have `data`, `indptr`, `indices`, `nrow`, `ncol` and `cs_type`. Shape is (metacells, genes) and the data are the raw counts.
`embd`	Numerical matrix. The embedding matrix from which to generate the kNN graph. Needs one row per entry of `cells_to_keep`.

knn_data	Optional list. This contains pre-computed kNN data (indices, dist, k) and the dist_metric it was built with. The user has to ensure consistency! If provided, this will be used rather than a graph built from the parameter list.
hotspot_params	List. The HotSpot parameter list. The kNN parameters are only read when no knn_data is provided; normalise is unused on this path.
cells_to_keep	Integer vector. 0-index vector indicating which meta cells to include in the analysis. Ensure that this is of same order/length as the embedding matrix.
genes_to_use	Integer vector. 0-index vector indicating which genes to include.
verbose	Integer. 0L - quiet; 1L - normal verbosity; 2L - detailed verbosity.
seed	Integer. Random seed for reproducibility.

Value

A list with the following elements.

- cor - The genes x genes local correlation matrix, in the order of genes_to_use.
- z - The Z-scores of these local correlations, same shape.

References

DeTomaso, et al., Cell Systems, 2021

---

rs_mc_hvg *Meta cells highly variable genes*

Description

[**Experimental**] Calculates highly variable genes for MetaCells or more generally speaking sparse data. This is happening in-memory compared to the (usually much) larger single cell data sets. "meanvarbin" and "dispersion" compute the same statistics; they differ only in how the R side selects from them.

Usage

```
rs_mc_hvg(
  sparse_data,
  hvg_method,
  loess_span,
  binning,
  n_bins,
  clip_max,
  scran_params,
  verbose
)
```

Arguments

sparse_data	A named list that needs to have data, indptr, indices, nrow, ncol and cs_type. Shape is (metacells, genes). Pass raw counts for "vst" and log1p-normalised counts otherwise.
hvg_method	String. Which HVG detection method to use. Options are c("vst", "meanvarbin", "dispersion", "scran").
loess_span	Numeric. The span parameter for the loess function ("vst") or the lowess trend ("scran").
binning	String. The binning strategy for the meanvarbin and dispersion methods. One of c("equal_width", "equal_freq").
n_bins	Integer. Number of bins for the meanvarbin and dispersion methods.
clip_max	Optional clipping number. Defaults to sqrt(no_cells) if not provided (only used for "vst").
scran_params	List. Trend parameters for "scran", see params_hvg_scran_defaults(). Missing elements fall back to the scrapper defaults. The span is taken from loess_span.
verbose	Integer. 0L - quiet; 1L - normal verbosity; 2L - detailed verbosity.

Value

A list with the HVG statistics. If hvg_method == "vst":

- mean - The average expression of the gene.
- var - The variance of the gene.
- var_exp - The expected variance of the gene.
- var_std - The standardised variance of the gene.

If hvg_method == "scran", all on the log2 scale:

- scran_mean - The mean log-expression of the gene.
- scran_var - The variance of the log-expression of the gene.
- scran_fitted - The variance the trend expects at the gene's mean.
- scran_residual - scran_var - scran_fitted. Genes are ranked on this.

For "meanvarbin" and "dispersion":

- mean - The average expression of the gene.
- dispersion - The dispersion of the gene.
- dispersion_scaled - The scaled dispersion per bin per gene.
- bin - The bin of the gene.

`rs_mc_pca` *PCA on MetaCells (sparse data)*

Description

[**Experimental**] Calculates PCA for MetaCells or more generally speaking sparse data. This is happening in-memory compared to the (usually much) larger single cell data sets. The matrix is densified, optionally CLR transformed and scaled according to `pca_params` before the SVD.

Usage

```
rs_mc_pca(sparse_data, no_pcs, pca_params, clr_offsets, seed, verbose)
```

Arguments

`sparse_data`	A named list that needs to have `data`, `indptr`, `indices`, `nrow`, `ncol` and `cs_type`. Shape is (metacells, genes), holding the normalised counts of the genes to use.
`no_pcs`	Integer. Number of PCs to return.
`pca_params`	Named list. Contains the parameters to use for this PCA run, see `params_sc_pca()`.
`clr_offsets`	Optional numeric. One offset per meta cell for the `PFlogPF` normalisation from Booeshaghi, et al., computed against the full gene panel. Required if `pca_params$clr` is `TRUE`, ignored otherwise.
`seed`	Integer. Random seed for the randomised SVD.
`verbose`	Integer. `0L` - quiet; `1L` - normal verbosity; `2L` - detailed verbosity.

Value

A list with the following items

- scores - The samples projected on the PCA space (solved via sparse SVD).
- loadings - The loadings of the features for the PCA (solved via sparse SVD).
- singular_values - The singular values for the PCA (solved via sparse SVD).

References

Booeshaghi, et al., bioRxive, 2026.

rs_mc_scenic *SCENIC on MetaCells*

Description

[**Experimental**] Assumes that the sparse data is pre-filtered for the cells and genes you wish to include: every column is a target gene. The regressors read the second layer, which here is an `f32` cast of the supplied counts.

Usage

```
rs_mc_scenic(sparse_data, tf_indices, scenic_params, seed, verbose)
```

Arguments

`sparse_data`	A named list that needs to have `data`, `indptr`, `indices`, `nrow`, `ncol` and `cs_type`. Shape is (metacells, genes).
`tf_indices`	Integer vector. 0-indexed(!) column positions of the transcription factors within `sparse_data`.
`scenic_params`	Named list. Contains all of the parameters needed for SCENIC, see `params_scenic()`.
`seed`	Integer. Controls reproducibility of the function.
`verbose`	Integer. `0L` - quiet; `1L` - normal verbosity; `2L` - detailed verbosity.

Value

A genes x TFs importance matrix, rows in column order of `sparse_data`, columns in the order of `tf_indices`.

rs_mc_vision *Calculate VISION pathway scores in Rust (for meta cells)*

Description

[**Experimental**] The function will take in a list of gene sets that contains lists of `"pos"` and `"neg"` gene indices (0-indexed). You don't have to provide the `"neg"`, but it can be useful to classify the delta of two stats (EMT, Th1; Th2) etc. This version works on MetaCell counts which are stored in memory directly.

Usage

```
rs_mc_vision(sparse_data, gs_list, verbose)
```

Arguments

`sparse_data`	A named list that needs to have `data`, `indptr`, `indices`, `nrow`, `ncol` and `cs_type`. Shape is (metacells, genes) and the data need to be the **normalised** counts.
`gs_list`	Nested list. Each sublist contains the (0-indexed!) positive and negative gene indices of that specific gene set.
`verbose`	Integer. `0L` - quiet; `1L` - normal verbosity; `2L` - detailed verbosity.

Value

A matrix of meta cells x vision scores per gene set.

References

DeTomaso, et al., Nat. Commun., 2019

`rs_mc_vision_with_autocorrelation`

Calculate VISION pathway scores with auto-correlation (for meta cells)

Description

[**Experimental**] The function will take in a list of gene sets that contains lists of `"pos"` and `"neg"` gene indices (0-indexed). You don't have to provide the `"neg"`, but it can be useful to classify the delta of two stats (EMT, Th1; Th2) etc. Additionally, it will take a random gene list and calculate an auto-correlation score based on Geary's C to identify pathways that show significant patterns on the kNN graph generated on the provided embedding. This version works on MetaCell counts which are stored in memory directly.

Usage

```
rs_mc_vision_with_autocorrelation(
  sparse_data,
  embd,
  knn_data,
  gs_list,
  random_gs_list,
  vision_params,
  cluster_membership,
  verbose,
  seed
)
```

Arguments

`sparse_data`	A named list that needs to have `data`, `indptr`, `indices`, `nrow`, `ncol` and `cs_type`. Shape is (metacells, genes) and the data need to be the **normalised** counts.
`embd`	Numerical matrix. The embedding matrix to use to generate the kNN graph. Needs to be of the same order/length as the meta cells in `sparse_data`.
`knn_data`	Optional list. This contains pre-computed kNN data (`indices`, `dist`, `k`) and the `dist_metric` it was built with. The user has to ensure consistency! If provided, this will be used rather than a graph built from the parameter list.
`gs_list`	Nested list. Each sublist contains the (0-indexed!) positive and negative gene indices of that specific gene set.
`random_gs_list`	Double-nested list. The outer list represents the clusters of gene sets and the inner list represents the permutations within that cluster.
`vision_params`	List. The kNN parameters, only read when no `knn_data` is provided.
`cluster_membership`	Integer vector. 1-indexed(!) position in `random_gs_list` of the permuted cluster each gene set belongs to.
`verbose`	Integer. `0L` - quiet; `1L` - normal verbosity; `2L` - detailed verbosity.
`seed`	Integer. Random seed for reproducibility.

Value

A list with the following items:

- autocor_res - List with `auto_cor` (1 - Geary's C), `p_val` and `fdr`, one entry per gene set.
- vision_mat - A matrix of meta cells x vision scores per gene set.

References

DeTomaso, et al., Nat. Commun., 2019

`rs_mitch_calc` *Calculate mitch enrichment leveraging Rust under the hood*

Description

[**Experimental**] Ranks the contrasts column-wise and runs the mitch MANOVA/ANOVA tests per pathway.

Usage

```
rs_mitch_calc(x, pathway_list, min_size)
```

Arguments

x	Numerical matrix. Each column represents one of the contrasts you wish to test for and the rows represent the gene statistics per contrast. Needs row names (the genes).
pathway_list	Named list. Each element represents one of the pathways to test for.
min_size	Integer. Minimum size of the gene set to be tested for.

Value

A list with the following elements:

- pathway_names - The name of the pathway.
- pathway_sizes - The size of the pathway.
- manova_pval - The p-value of the MANOVA test.
- manova_fdr - The Benjamini-Hochberg adjusted `manova_pval`.
- anova_pvals - The p-values of the ANOVA test on top of the MANOVA results. Total length = `ncol(x)` * number of pathways, pathway-major.
- scores - The scores for each pathway set, contrast. Same length and layout as `anova_pvals`.
- s_dist - Calculated distances from the hypotenuse.
- sd - SDs of the scores.

`rs_mutual_info` *Calculates the mutual information matrix*

Description

[**Experimental**] Calculates the pairwise mutual information across all columns in the data.

Usage

```
rs_mutual_info(x, n_bins, strategy, normalise)
```

Arguments

x	R matrix with doubles for which to calculate the mutual information
n_bins	Optional integer. Number of bins to use. If `NULL` is provided the function will default to `sqrt(nrow(x))`.
strategy	String. Binning strategy. One of `c("equal_width", "equal_freq")`. Unknown strings default to `"equal_width"`.
normalise	Boolean. Shall the normalised mutual information be calculated via joint entropy.

Value

The symmetric mutual information matrix. The diagonal holds the column entropies, or `0` if `normalise = TRUE`.

`rs_nebula_mc` *Fit the NEBULA negative binomial gamma mixed model over meta cells*

Description

[Experimental] The arithmetic is the single cell one verbatim, only the counts come from memory rather than the streamed store. What changes is the interpretation: the cell-level overdispersion becomes the spread between aggregates within a subject, not between cells, so it is smaller and absorbs whatever the aggregation smoothed away. The subject-level term keeps its meaning. Do not compare the two against a single cell run.

Usage

```
rs_nebula_mc(
  sparse_data,
  metacells_to_keep,
  gene_indices,
  subject_ids,
  design,
  offset,
  nebula_params,
  verbose
)
```

Arguments

`sparse_data`	A named list that needs to have `data`, `indptr`, `indices`, `nrow`, `ncol` and `cs_type`. Shape (metacells, genes), holding the raw counts.
`metacells_to_keep`	Integer vector. 0-indexed(!) positions of the meta cells to analyse, in any order. Must not hold duplicates.
`gene_indices`	Integer vector. 0-indexed(!) positions of the genes to fit.
`subject_ids`	Integer vector. 0-indexed(!) subject label per meta cell. One entry per row of `sparse_data`, not per element of `metacells_to_keep`.
`design`	Numeric matrix. Predictors of meta cells x coefficients, rows aligned to `metacells_to_keep` and including an intercept.
`offset`	Optional numeric vector. Strictly positive scaling factor per selected meta cell. `NULL` uses the aggregated library sizes.
`nebula_params`	Named list. The NEBULA parameters, see `params_nebula()`, plus either `coef` (a 0-indexed(!) coefficient) or `contrast` (one weight per coefficient).
`verbose`	Integer. `0L` - quiet; `1L` - normal verbosity; `2L` - detailed verbosity.

Value

A list with the same elements `rs_nebula_sc()` returns.

References

He, et al., Commun Biol, 2021

`rs_onto_semantic_sim` *Calculate the semantic similarity in an ontology*

Description

[**Experimental**] This function calculates the specified semantic similarity between all unique pairs of `terms` (upper triangle, no diagonal) and returns them in long format.

Usage

```
rs_onto_semantic_sim(terms, sim_type, ancestor_list, ic_list)
```

Arguments

`terms`	Vector of strings. The terms in the ontology you wish to screen.
`sim_type`	String. Must be one of `c("resnik", "lin", "combined")`.
`ancestor_list`	R list with names being the term and the elements in the list the names of the ancestors.
`ic_list`	R list with the names being the term and the elements the information content of this given term. Needs to be a single float!

Value

A list with:

- term1 - name of the first term.
- term2 - name of the second term.
- sims - similarity between the two terms.

`rs_onto_semantic_sim_mat`
Calculate the semantic similarity in an ontology

Description

[**Experimental**] This function calculates the specified semantic similarity between all terms in `ic_list` (only calculating the upper triangle) and returns it either as the flat upper triangle or as the full matrix.

Usage

```
rs_onto_semantic_sim_mat(sim_type, ancestor_list, ic_list, flat_matrix)
```

Arguments

`sim_type`	String. Must be one of `c("resnik", "lin", "combined")`.
`ancestor_list`	R list with names being the term and the elements in the list the names of the ancestors.
`ic_list`	R list with the names being the term and the elements the information content of this given term. Needs to be a single float!
`flat_matrix`	Boolean. Shall only the upper triangle (row-wise, diagonal excluded) be returned.

Value

A list with:

- sim_mat - the semantic similarity matrix (flat or as matrix). The diagonal of the full matrix is `1`.
- names - the row and column names for the calculated matrix, i.e. the names of `ic_list` sorted alphabetically.

`rs_onto_sim_wang` *Calculate the Wang similarity for specific terms*

Description

[**Experimental**] This function calculates the Wang similarities between all unique pairs of a given set of terms. Terms not found in the ontology get `NaN`.

Usage

```
rs_onto_sim_wang(terms, parents, children, w)
```

Arguments

`terms`	String vector. The terms you wish to calculate the similarities for.
`parents`	String vector. The names of the parents.
`children`	String vector. The names of the children. The length of `parents` needs to be equal to `children`.
`w`	Numerics. The weights between the parents and children. Need to be values between 0 and 1.

Value

A list with:

- term1 - name of the first term.
- term2 - name of the second term.
- sims - similarity between the two terms.

rs_onto_sim_wang_mat *Calculate the Wang similarity matrix for an ontology*

Description

[**Experimental**] This function calculates the Wang similarity matrix for a given ontology.

Usage

```
rs_onto_sim_wang_mat(parents, children, w, flat_matrix)
```

Arguments

`parents`	String vector. The names of the parents.
`children`	String vector. The names of the children. The length of `parents` needs to be equal to `children`.
`w`	Numerics. The weights between the parents and children. Need to be values between 0 and 1.
`flat_matrix`	Boolean. Shall only the upper triangle (row-wise, diagonal excluded) be returned.

Value

A list with:

- sim_mat - the Wang similarity matrix (flat or as matrix.)
- names - the row and column names for the calculated matrix.

rs_ot_harmonic_sum *Calculate the OT harmonic sum*

Description

[**Experimental**] Sorts `x` in decreasing order and sums `x[i] / i^2`, normalised by the same sum over a vector of ones of equal length.

Usage

```
rs_ot_harmonic_sum(x)
```

Arguments

`x`	The numeric vector (should be between 0 and 1) for which to calculate the harmonic sum

Value

The normalised harmonic sum according to the OT calculation.

`rs_page_rank` *Rust version of calculating the personalised page rank*

Description

[**Experimental**] Personalised page rank with at most 1000 iterations.

Usage

```
rs_page_rank(
  node_names,
  from,
  to,
  weights,
  personalised,
  undirected,
  damping_factor,
  tol
)
```

Arguments

`node_names`	String vector. Name of the graph nodes.
`from`	String vector. The names of the `from` edges from the edge list.
`to`	String vector. The names of the `to` edges from the edge list.
`weights`	Optional weight vector. If NULL, defaults to 1.0 as weight for all edges.
`personalised`	Numerical vector. The reset values. They must sum to 1 and be of same length of `node_names`!
`undirected`	Boolean. Is this an undirected graph.
`damping_factor`	Numeric in `[0, 1]`. Probability of continuing the walk.
`tol`	Numeric > 0. Convergence threshold on the L1 change between iterations.

Value

The personalised page rank values.

`rs_page_rank_parallel` *Calculate massively parallelised personalised page rank scores*

Description

[**Experimental**] Helper function to calculate in parallel on the same network the personalised page rank as fast as possible. Can be used for permutation type approaches.

Usage

```
rs_page_rank_parallel(
  node_names,
  from,
  to,
  weights,
  diffusion_scores,
  undirected,
  damping_factor,
  tol
)
```

Arguments

`node_names`	String vector. Name of the graph nodes.
`from`	String vector. The names of the `from` edges from the edge list.
`to`	String vector. The names of the `to` edges from the edge list.
`weights`	Optional weight vector. If NULL, defaults to 1.0 as weight for all edges.
`diffusion_scores`	List. The personalised vectors for the page rank reset values. Each element must sum to 1 and be of same length of `node_names`!
`undirected`	Boolean. Is this an undirected graph.
`damping_factor`	Numeric in `[0, 1]`. Probability of continuing the walk.
`tol`	Numeric > 0. Convergence threshold on the L1 change between iterations.

Value

A matrix of the scores with each row representing an element in the `diffusion_scores` list (in order), and each column representing the value of the personalised page rank diffusion for this node.

rs_pcr — *Principal component regression on batch*

Description

[**Experimental**] Regresses each embedding dimension on the batch labels (one-way ANOVA) and weights the per-dimension R-squared by the variance each dimension carries. Compare the value on the uncorrected PCA with the one on the corrected embedding, `(pre - post) / pre`, rather than reading it on its own.

Usage

```
rs_pcr(embedding, batch_vector)
```

Arguments

`embedding`	Numeric matrix. Cells x dimensions, ideally a PCA.
`batch_vector`	Integer vector. The batch per cell. The codes need not be 0-based or contiguous.

Value

A list with the following items

- var_explained - Variance per embedding dimension.
- r_squared - R-squared of batch per embedding dimension.
- pcr - Variance-weighted R-squared of batch.

References

Büttner, et al., Nat Methods, 2019; Luecken, et al., Nat Methods, 2022

rs_phyper — *Calculate the hypergeometric test in Rust*

Description

[**Experimental**]

Usage

```
rs_phyper(q, m, n, k)
```

Arguments

`q`	Number of white balls drawn out of urn.
`m`	Number of white balls in the urn.
`n`	Number of black balls in the urn.
`k`	The number of balls drawn out of the urn.

Value

P-value, i.e. `P(X > q)` (equivalent to `lower.tail = FALSE`).

`rs_pointwise_mutual_info`

Calculates the point wise mutual information

Description

[**Experimental**] Calculates the pointwise mutual information (can be also normalised) across all columns of the data. This can be used to identify (dis)similar samples based on Boolean characteristics.

Usage

```
rs_pointwise_mutual_info(x, normalise)
```

Arguments

`x`	Logical matrix. The columns represent features and the rows represent samples
`normalise`	Shall the normalised pointwise mutual information be returned.

Value

The (normalised) pointwise mutual information matrix between the columns. Pairs without co-occurrence are `-Inf`.

`rs_prcomp` *Rust implementation of prcomp*

Description

[**Experimental**] Runs the singular value decomposition over the matrix x. Assumes that samples = rows, and columns = features.

Usage

```
rs_prcomp(x, scale, top_pcs)
```

Arguments

`x`	Numeric matrix. Rows = samples, columns = features.
`scale`	Boolean. Shall the columns be variance normalised. (Mean centring will automatically occur.)
`top_pcs`	Optional integer. Only return the top PCs (under the hood all of them will be calculated). `NULL` returns all.

Value

A list with:

- scores - The product of x (centred and potentially scaled) with v.
- v - v matrix of the SVD.
- s - Standard deviations of the PCs, i.e. singular values divided by `sqrt(nrow(x) - 1)`.
- scaled - Boolean. Was the matrix scaled.

`rs_prepare_whitening` *Prepare the data for whitening*

Description

[**Experimental**] Prepares the data for subsequent usage in ICA. Incorrect use can cause kernel crashes. Wrapper around the Rust functions with type checks are provided in the package.

Usage

```
rs_prepare_whitening(x, fast_svd, seed, rank, oversampling, n_power_iter)
```

Arguments

`x`	Numeric matrix to whiten. The columns are centred and the whitening happens over the columns.
`fast_svd`	Boolean. Shall a randomised SVD be used. This is way faster on larger data sets.
`seed`	Integer. Only relevant with fast_svd is set to `TRUE`.
`rank`	Integer. How many ranks to use for the fast SVD approximation. If you supply `NULL`, it will default to `10L`. Only relevant with fast_svd is set to `TRUE`.
`oversampling`	Integer. Oversampling parameter to make the approximation more precise. If you supply `NULL`, it will default to `10L`. Only relevant with fast_svd is set to `TRUE`.
`n_power_iter`	Integer. Number of power iterations for the randomised SVD. If you supply `NULL`, it will default to `2L`. Only relevant with fast_svd is set to `TRUE`.

Value

A list containing:

- x - The column-centred input, transposed.
- k - The whitening matrix K. With `fast_svd = TRUE` it can carry more than `rank` rows; the caller trims it.

`rs_random_svd` *Run randomised SVD over a matrix*

Description

[**Experimental**] Runs a randomised singular value decomposition over a matrix. This implementation is faster than the full SVD on large data sets, with slight loss in precision.

Usage

```
rs_random_svd(x, scale, rank, seed, oversampling, n_power_iter)
```

Arguments

`x`	Numeric matrix. Rows = samples, columns = features.
`scale`	Boolean. Shall the columns be variance normalised. (Mean centring will automatically occur.)
`rank`	Integer. The rank to use.
`seed`	Integer. Random seed for reproducibility.
`oversampling`	Optional integer. Defaults to `10L` if `NULL`.
`n_power_iter`	Optional integer. Number of power iterations (each with a QR decomposition). Defaults to `2L` if `NULL`.

Value

A list with:

- scores - u matrix of the SVD multiplied by the singular values.
- v - v matrix of the SVD.
- s - Singular values of the SVD.
- scaled - Boolean. Was the matrix scaled.

`rs_range_norm` *Apply a range normalisation on a vector.*

Description

[**Experimental**] Applies a range normalisation on an R vector.

Usage

```
rs_range_norm(x, max_val, min_val)
```

Arguments

`x`	Numerical vector. The data to normalise.
`max_val`	Numeric. The upper bound value to normalise into. If set to 1, the function will be equal to a min-max normalisation.
`min_val`	Numeric. The lower bound value to normalise into. If set to 0, the function will equal a min-max normalisation.

Value

Normalised values. A constant vector returns `max_val` for every element.

`rs_rank_matrix_col` *Gene rank matrix*

Description

[**Experimental**] Ranks the columns of a matrix with the tie method.

Usage

```
rs_rank_matrix_col(exp)
```

Arguments

`exp`	Numerical matrix. The expression matrix (rows = genes, columns = samples).

Value

Returns a matrix of ranks with the same shape as `exp`.

`rs_rank_matrix_col_stable` *Stable-gene rank matrix*

Description

[**Experimental**] Unit-normalised column ranks computed against a set of stable genes. Use with the `stable = TRUE` setting of the singscore functions.

Usage

```
rs_rank_matrix_col_stable(exp, stable_gene_indices)
```

Arguments

`exp`	Numerical matrix. The expression matrix (rows = genes, columns = samples).
`stable_gene_indices`	Integer vector of stable genes. One-indexed.

Value

Returns a matrix of normalised ranks with the same shape as `exp`.

`rs_rbf_function` *Apply a Radial Basis Function*

Description

[**Experimental**] Applies a radial basis function (RBF) to a given distance vector. Has the option to apply a Gaussian, Bump or Inverse Quadratic RBF.

Usage

```
rs_rbf_function(x, epsilon, rbf_type)
```

Arguments

`x`	Numeric vector. The distances you wish to apply the RBF onto.
`epsilon`	Float. Epsilon parameter for the RBF.
`rbf_type`	String. Needs to be from `c("gaussian", "bump", "inverse_quadratic")`. Other values raise an error.

Value

The affinities after the Kernel was applied.

`rs_rbf_function_mat` *Apply a Radial Basis Function (to a matrix)*

Description

[**Experimental**] Applies a radial basis function (RBF) to a given distance matrix. Has the option to apply a Gaussian, Bump or Inverse Quadratic RBF.

Usage

```
rs_rbf_function_mat(x, epsilon, rbf_type)
```

Arguments

`x`	Numeric Matrix. The distances you wish to apply the RBF onto.
`epsilon`	Float. Epsilon parameter for the RBF.
`rbf_type`	String. Needs to be from `c("gaussian", "bump", "inverse_quadratic")`. Other values raise an error.

Value

The affinities after the Kernel was applied.

`rs_rbh_cor` *Generate reciprocal best hits based on correlations*

Description

[**Experimental**] This function takes list of (named) matrices which represent for example matrix factorisation results you wish to identify (k-th) reciprocal best hits (RBH) for. The rows need to represent the features and the columns the parts you wish to calculate the RBH for.

Usage

```
rs_rbh_cor(module_matrices, k_best, spearman, min_similarity)
```

Arguments

`module_matrices`
: A named list of matrices with row and column names. Rows represent features and columns the modules you wish to calculate the correlations for. Only features shared between two matrices are used.

`k_best`
: Integer. Number of best neighbours to consider. If set to `1L`, this behaves as the traditional reciprocal best hit. If you set this to `3L` you consider edges if the modules is in the top 3 best modules by similarity for each other.

`spearman`
: Boolean. Shall Spearman correlation be used.

`min_similarity`
: Numeric. Only hits with an absolute correlation strictly above this are returned.

Value

A list containing:

- origin - The name of the origin of the gene modules.
- target - The name of the target of the gene modules.
- comparisons - Integer vector indicating how many RBH hits were identified in this comparison
- origin_modules - Names of the gene modules from the origin.
- target_modules - Names of the gene modules from the target.
- similarity - The absolute correlations between the two respective gene modules.

`rs_rbh_sets` *Generate reciprocal best hits based on set similarities*

Description

[**Experimental**] This function takes a nested list that contains gene modules/sets derived from various methods and identifies (k-th) reciprocal best hits between gene modules/sets across every pair of origins.

Usage

```
rs_rbh_sets(module_list, k_best, overlap_coefficient, min_similarity)
```

Arguments

`module_list`	A nested named list. The outer list should contain the origin of the gene modules, the inner list the names of the gene modules and the respective genes in them.
`k_best`	Integer. Number of best neighbours to consider. If set to `1L`, this behaves as the traditional reciprocal best hit. If you set this to `3L` you consider edges if the modules is in the top 3 best modules by similarity for each other.
`overlap_coefficient`	Boolean. Shall the overlap coefficient instead of the Jaccard similarity be used.
`min_similarity`	Numeric. Only hits with a similarity strictly above this are returned.

Value

A list containing:

- origin - The name of the origin of the gene modules.
- target - The name of the target of the gene modules.
- comparisons - Integer vector indicating how many RBH hits were identified in this comparison
- origin_modules - Names of the gene modules from the origin.
- target_modules - Names of the gene modules from the target.
- similarity - The similarities between the two respective gene modules.

`rs_remove_batch_effect` *Remove batch effects from a log-expression matrix*

Description

[**Experimental**] limma's `removeBatchEffect`, via the `edge-rs` crate. Batch gets sum-to-zero contrasts, is fitted jointly with the design of interest and only the batch part is subtracted. Meant for plotting and unsupervised work; for testing put batch into the design.

Usage

```
rs_remove_batch_effect(x, batch, design)
```

Arguments

`x`	Numeric matrix. Log-expression values of genes x samples.
`batch`	Integer vector. Batch label per sample, e.g. `as.integer(factor(x))`. Needs at least two distinct labels.
`design`	Numeric matrix or NULL. The design of interest, samples x coefficients, whose effects are protected. NULL is an intercept only.

Value

Numeric matrix of corrected values, genes x samples.

References

Smyth, Stat Appl Genet Mol Biol, 2004

---

rs_sc_knn *Generates the kNN graph*

Description

[**Experimental**] This function is a wrapper over the Rust-based generation of the approximate nearest neighbours.

Usage

```
rs_sc_knn(embd, knn_params, validate_index, verbose, seed)
```

Arguments

`embd`	Numerical matrix. The embedding matrix to use to generate the kNN graph.
`knn_params`	List. The kNN parameters defined by `params_sc_neighbours()`.
`validate_index`	Boolean. If you want to validate the index via an exhaustive search in a subset of cells.
`verbose`	Integer. `0L` - quiet; `1L` - normal verbosity; `2L` - detailed verbosity.
`seed`	Integer. Seed for reproducibility purposes.

Value

An integer matrix of N x k with the neighbour indices (0-indexed!), N being the number of cells and k the number of neighbours. Rows the search left short are padded by repeating their last neighbour.

rs_sc_knn_w_dist — *Generates the kNN graph with additional distances*

Description

[**Experimental**] This function is a wrapper over the Rust-based generation of the approximate nearest neighbours.

Usage

```
rs_sc_knn_w_dist(embd, knn_params, validate_index, verbose, seed)
```

Arguments

`embd`	Numerical matrix. The embedding matrix to use to generate the kNN graph.
`knn_params`	List. The kNN parameters defined by `params_sc_neighbours()`.
`validate_index`	Boolean. If you want to validate the index via an exhaustive search in a subset of cells.
`verbose`	Integer. `0L` - quiet; `1L` - normal verbosity; `2L` - detailed verbosity.
`seed`	Integer. Seed for reproducibility purposes.

Value

A list with:

- indices - An integer matrix (cells x k) representing the indices (0-indexed!) of the approximate nearest neighbours.
- dist - A numerical matrix (cells x k) representing the distances to the nearest neighbours.
- dist_metric - String representing the used distance metric.

Rows the search left short are padded by repeating their last neighbour and distance.

rs_sc_snn — *Generates the sNN graph for igraph*

Description

[**Experimental**] This function takes a kNN matrix and generates the inputs for an SNN graph based on it.

Usage

```
rs_sc_snn(knn_mat, snn_method, limited_graph, pruning, verbose)
```

Arguments

knn_mat	Integer matrix. Rows represent cells and the columns represent the neighbours (0-indexed!).
snn_method	String. Which method to use to calculate the similarity. Choice of c("jaccard", "rank"); anything else errors.
limited_graph	Boolean. Shall the sNNs only be calculated between direct neighbours in the graph, or between all possible combinations.
pruning	Float. Below which similarity value to prune the weight to 0. A cell that loses every edge keeps its strongest kNN edge.
verbose	Integer. 0L - quiet; 1L - normal verbosity; 2L - detailed verbosity.

Value

A list with the following items:

- edges - Integer vector with the flattened sNN edge pairs (1-indexed!), ready for igraph::add_edges().
- weights - sNN weights of the pairs above.

---

rs_set_similarity *Set similarities*

Description

[**Experimental**] This function calculates the Jaccard similarity or overlap coefficient between two string vectors.

Usage

```
rs_set_similarity(s_1, s_2, overlap_coefficient)
```

Arguments

s_1	Character vector. The first set.
s_2	Character vector. The second set.
overlap_coefficient	Boolean. Use the overlap coefficient instead of the Jaccard similarity.

Value

The Jaccard similarity or overlap coefficient.

`rs_set_similarity_list`

Set similarities over one list

Description

[**Experimental**] This function calculates the set similarity via Jaccard or overlap coefficient across all permutations of one list.

Usage

```
rs_set_similarity_list(list, overlap_coefficient)
```

Arguments

`list`	A named R list.
`overlap_coefficient`	Boolean. Use the overlap coefficient instead of the Jaccard similarity.

Value

A list with the following items, one entry per unordered pair:

- from - Name of element i
- to - Name of element j
- sim - Similarity between the two elements

`rs_set_similarity_list2`

Set similarities over two list

Description

[**Experimental**] This function calculates the Jaccard similarity or overlap coefficient between all elements of two lists.

Usage

```
rs_set_similarity_list2(s_1_list, s_2_list, overlap_coefficient)
```

Arguments

`s_1_list`	R list. The first list of string elements you want to compare against.
`s_2_list`	R list. The second list of string elements you want to compare against.
`overlap_coefficient`	Boolean. Use the overlap coefficient instead of the Jaccard similarity.

Value

A matrix of the similarities between the elements. The rows represent `s_1_list` and the columns `s_2_list`.

`rs_singscore_multi` *Rust version of singscore for many gene sets*

Description

[**Experimental**] Rust-based implementation of singscore over many gene sets with optional paired down sets.

Usage

```
rs_singscore_multi(
  ranks,
  up_list,
  down_list,
  center_score,
  known_direction,
  stable
)
```

Arguments

`ranks`	Numerical matrix. The ranked expression matrix.
`up_list`	List. Up gene sets as 0-based indices. See `rs_prepare_gsva_gs()`.
`down_list`	List or NULL. Paired down gene sets as 0-based indices, same length and ordering as `up_list`.
`center_score`	Boolean. Centre scores around 0. Disabled internally when `known_direction = FALSE`.
`known_direction`	Boolean. Whether the up-set direction is known.
`stable`	Boolean. If `TRUE`, use stable-gene score bounds.

Value

A named list with

- `scores` - Numerical matrix with the scores
- `dispersions` - Numerical matrix with the dispersions

Both matrices are of shape gene sets x samples.

`rs_singscore_permutation_test` *Rust version of the singscore permutation test*

Description

[**Experimental**] For `n_permutations` iterations, draws random gene indices of the same total size as the real gene set(s), scores them with the same options as the real call, and builds a per-sample null distribution. Returns empirical one-tailed p-values: `max(1 / n_permutations, mean(null > observed))`.

Usage

```
rs_singscore_permutation_test(
  ranks,
  up_set,
  down_set,
  center_score,
  known_direction,
  stable,
  n_permutations,
  seed
)
```

Arguments

`ranks`	Numerical matrix. The ranked expression matrix.
`up_set`	Integer vector. One-indexed (shifted internally).
`down_set`	Integer vector or NULL. One-indexed (shifted internally).
`center_score`, `known_direction`, `stable`	Booleans. Should match the values used for the real `rs_singscore_single()` call.
`n_permutations`	Integer. Number of random draws (B).
`seed`	Integer. RNG seed.

Value

A named list with `observed_scores` (length n_samples), `null_distribution` (B x n_samples matrix), and `p_values` (length n_samples).

`rs_singscore_single` *Rust version of singscore for a single gene set*

Description

[**Experimental**] Rust-based implementation of singscore for a single up-regulated gene set with an optional paired down-regulated set.

Usage

```
rs_singscore_single(
  ranks,
  up_set,
  down_set,
  center_score,
  known_direction,
  stable
)
```

Arguments

`ranks`	Numerical matrix. The ranked expression matrix (rows = genes, columns = samples). Produce with column-wise ranks (standard) or with `rs_rank_matrix_col_stable()` (stable).
`up_set`	Integer vector. One-indexed gene indices of the up set.
`down_set`	Integer vector or NULL. One-indexed gene indices of the optional down set.
`center_score`	Boolean. Centre scores around 0. Disabled internally when `known_direction = FALSE`.
`known_direction`	Boolean. Whether the up-set direction is known. Becomes irrelevant when `down_set` is also provided.
`stable`	Boolean. If `TRUE`, use stable-gene score bounds.

Value

A named list with `total_score`, `total_dispersion`, `up_score`, `up_dispersion`, `down_score` and `down_dispersion`. The last four are `NULL` unless `down_set` is provided.

rs_snf *Similarity network fusion*

Description

[**Experimental**] This function iteratively fuses the affinity matrices together.

Usage

```
rs_snf(aff_mat_list, k, t, alpha)
```

Arguments

aff_mat_list	A list of numerical matrices. The affinity matrices to fuse together.
k	Integer. Number of neighbours to consider.
t	Integer. Number of iterations for the algorithm.
alpha	Float. Normalisation parameter controlling the fusion strength.

Value

The final affinity matrix after the fusion.

rs_sparse_dict_dgrdl *Generate a sparse dictionary with DGRDL*

Description

[**Experimental**] This is the Rust implementation of dual graph regularised dictionary learning in the implementation of Pan, et al., Cell Systems, 2022.

Usage

```
rs_sparse_dict_dgrdl(x, dgrdl_params, seed, verbose)
```

Arguments

x	Numerical matrix. Rows = samples, columns = features.
dgrdl_params	A list with the parameters for the algorithm. Missing items fall back to defaults. Expects the following items. • sparsity - Integer. Sparsity constraint (max non-zero coefficients per signal). • dict_size - Integer. Size of the dictionary. • alpha - Float. Sample context regularisation weight. The higher the stronger the regularisation.

- beta - Float. Feature context regularisation weight. The higher the stronger the regularisation.
- max_iter - Integer. Maximum iteration for the algorithm.
- k_neighbours - Integer. Number of k neighbours for the sample and feature Laplacian matrix for the regularisation.
- admm_iter - Integer. Number of iterations for using alternating direction method of multipliers (ADMM).
- rho - Float. ADMM step size.

`seed` Integer. Seed for the initialisation of the algorithm.

`verbose` Boolean. Controls the verbosity of the function and reports timing of individual steps.

Value

A list with the following elements:

- dictionary - The dictionary of samples x dict_size.
- coefficients - The feature loadings of size dict_size x features.
- feature_laplacian - The kNN graph Laplacian of the features as a CSR list with `data`, `indptr`, `indices`, `nrow`, `ncol` and `cs_type`.
- sample_laplacian - The kNN graph Laplacian of the samples, same format.

`rs_sparse_dict_dgrdl_grid_search`

Generate a sparse dictionary with DGRDL

Description

[**Experimental**] This is the Rust implementation of dual graph regularised dictionary learning in the implementation of Pan, et al., Cell Systems, 2022. This helper function is designed to run a grid search over the data.

Usage

```
rs_sparse_dict_dgrdl_grid_search(
  x,
  dgrdl_params,
  seeds,
  dict_sizes,
  k_neighbours_vec,
  verbose
)
```

Arguments

`x`	Numerical matrix. Rows = samples, columns = features.
`dgrdl_params`	A list with the parameters for the algorithm. Missing items fall back to defaults. Expects the following items. • sparsity - Integer. Sparsity constraint (max non-zero coefficients per signal). • dict_size - Integer. Size of the dictionary. Ignored here, `dict_sizes` is used instead. • alpha - Float. Sample context regularisation weight. The higher the stronger the regularisation. • beta - Float. Feature context regularisation weight. The higher the stronger the regularisation. • max_iter - Integer. Maximum iteration for the algorithm. • k_neighbours - Integer. Number of k neighbours for the sample and feature Laplacian matrix. Ignored here, `k_neighbours_vec` is used instead. • admm_iter - Integer. Number of iterations for using alternating direction method of multipliers (ADMM). • rho - Float. ADMM step size.
`seeds`	Integer vector. The random seeds to include in the grid search.
`dict_sizes`	Integer vector. The dictionary sizes to test in the grid search.
`k_neighbours_vec`	Integer vector. The number of neighbours for the kNN graph generation to test in the grid search.
`verbose`	Boolean. Controls verbosity of the function.

Value

A list with the following elements, one entry per tested combination:

- seed - The tested seeds.
- dict_size - The tested dictionary sizes.
- k_neighbours - The tested numbers of neighbours.
- reconstruction_errs - The reconstruction errors (squared Frobenius norm) for these hyperparameters.
- feature_laplacian_objective - The objective values of the feature Laplacian term for these hyperparameters.
- sample_laplacian_objective - The objective values of the sample Laplacian term for these hyperparameters.

`rs_spatial_fdr` *Weighted Benjamini-Hochberg over overlapping neighbourhoods*

Description

[**Experimental**] Milo's spatial FDR. Neighbourhoods overlap, so the tests are not independent and a plain BH is anti-conservative. Each p-value is weighted by the reciprocal of its connectivity and the step-up runs on those weights. Non-finite p-values are carried through untouched and take no part in the adjustment.

Usage

```
rs_spatial_fdr(p_values, connectivity)
```

Arguments

`p_values`	Numeric vector. One raw p-value per tested neighbourhood.
`connectivity`	Numeric vector. The matching connectivity per neighbourhood, either the k-th neighbour distances or the neighbourhood overlaps. A zero connectivity gets a weight of one, as in the upstream.

Value

The adjusted p-values, in the input order.

References

Dann, et al., Nat Biotechnol, 2022

`rs_spectral_clustering`
Rust implementation of spectral clustering

Description

[**Experimental**] This version can take in data as is and will calculate the distance matrix internally and convert it into similarities with a Gaussian RBF.

Usage

```
rs_spectral_clustering(
  data,
  distance_type,
  epsilon,
  k_neighbours,
  n_clusters,
```

```
  max_iters,
  seed
)
```

Arguments

`data`	Numerical matrix. The data to cluster. Rows = samples, columns = features.
`distance_type`	String. One of `c("euclidean", "manhattan", "canberra", "cosine", "correlation")`. Unknown strings raise an error.
`epsilon`	Numerical. The epsilon parameter for the Gaussian Radial Basis function
`k_neighbours`	Integer. Number of neighbours to consider in the kNN graph generation
`n_clusters`	Integer. Number of clusters to identify
`max_iters`	Integer. Number of iterations for k-means clustering
`seed`	Integer. Seed for reproducibility

Value

A vector with the 1-based cluster membership of the samples.

`rs_spectral_clustering_sim`

Rust implementation of spectral clustering

Description

[**Experimental**] Spectral clustering on a pre-calculated similarity matrix.

Usage

```
rs_spectral_clustering_sim(
  similarities,
  k_neighbours,
  n_clusters,
  max_iters,
  seed
)
```

Arguments

`similarities`	Numerical matrix representing the similarities. Needs to be symmetric!
`k_neighbours`	Integer. Number of neighbours to consider in the kNN graph generation
`n_clusters`	Integer. Number of clusters to identify
`max_iters`	Integer. Number of iterations for k-means clustering
`seed`	Integer. Seed for reproducibility

Value

A vector with the 1-based cluster membership of the samples.

`rs_ssgsea` *Rust version of the ssGSEA algorithm*

Description

[**Experimental**] Rust-based implementation of the popular single sample GSEA algorithm. Has further performance optimisations compared to the original implementation.

Usage

```
rs_ssgsea(exp, gs_list, alpha, normalise, timings)
```

Arguments

`exp`	Numerical matrix. The expression matrix with rows = genes, and columns = samples
`gs_list`	List. A list containing the 0-based indices of the pathway genes. See `rs_prepare_gsva_gs()`.
`alpha`	Float. The alpha parameter to adjust the weights.
`normalise`	Boolean. Shall the scores be normalised.
`timings`	Boolean. Prints timings from the algorithm.

Value

Returns a matrix of gene set ES scores x samples.

`rs_tied_diffusion_parallel`
Calculate massively parallelised tied diffusion scores

Description

[**Experimental**] Helper function to calculate in parallel on the same network the tied diffusions as fast as possible. Can be used for permutation.

Usage

```
rs_tied_diffusion_parallel(
  node_names,
  from,
  to,
  weights,
  diffusion_scores_1,
  diffusion_scores_2,
  summarisation_fun,
  undirected,
  damping_factor,
  tol
)
```

Arguments

`node_names`	String vector. Name of the graph nodes.
`from`	String vector. The names of the `from` edges from the edge list.
`to`	String vector. The names of the `to` edges from the edge list.
`weights`	Optional weight vector. If NULL, defaults to 1.0 as weight for all edges.
`diffusion_scores_1`	List. The first set of personalised vectors for the page rank reset values. Each element must sum to 1 and be of same length of `node_names`!
`diffusion_scores_2`	List. The second set of personalised vectors for the page rank reset values. Each element must sum to 1 and be of same length of `node_names`!
`summarisation_fun`	String. One of `c("min", "max", "mean")`. Which type of summarisation function to use to calculate the tied diffusion. Other values cause a panic.
`undirected`	Boolean. Is this an undirected graph.
`damping_factor`	Numeric in `[0, 1]`. Probability of continuing the walk.
`tol`	Numeric > 0. Convergence threshold on the L1 change between iterations.

Value

A matrix of the scores with each row representing a tied diffusion of the `diffusion_scores_1` and `diffusion_scores_2` lists (in order), and each column representing the value of the tied diffusion for this node.

`rs_tom` *Calculates the TOM over an affinity matrix*

Description

[**Experimental**] Calculates the topological overlap measure for a given affinity matrix x. Has the option to calculate the signed and unsigned version.

Usage

```
rs_tom(x, tom_type, signed)
```

Arguments

`x`	Numerical matrix. Affinity matrix.
`tom_type`	String. One of `c("v1", "v2")` - pending on choice, a different normalisation method will be used.
`signed`	Boolean. Shall the signed TOM be calculated, i.e. the connectivity be taken over absolute affinities. If set to `FALSE`, values should be >= 0.

Value

Returns the TOM matrix.

`rs_upper_triangle_to_dense`
Reconstruct a matrix from a flattened upper triangle vector

Description

[**Experimental**] This function takes a flattened vector of the upper triangle from a symmetric matrix (think correlation matrix) and reconstructs the full dense matrix for you.

Usage

```
rs_upper_triangle_to_dense(data, shift, n)
```

Arguments

`data`	Numeric vector. The vector of for example correlation coefficients that you want to use to go back to a dense matrix.
`shift`	Boolean. If `TRUE`, `data` excludes the diagonal and the diagonal is set to `1`. If `FALSE`, `data` includes the diagonal.
`n`	Integer. Original dimension (i.e., ncol/nrow) of the matrix to be reconstructed.

Value

The dense R matrix.

`rs_upper_triangle_to_sparse`

Generate sparse data from an upper triangle

Description

[**Experimental**] This function takes the values from an upper triangle matrix, the shift and the nrows/ncols and returns the full symmetric matrix as a compressed sparse list.

Usage

```
rs_upper_triangle_to_sparse(value, shift, n, cs_type)
```

Arguments

`value`	Numeric vector. The upper triangle values.
`shift`	Boolean. Was the matrix shifted up (`FALSE` = diagonal included; `TRUE` = diagonal not included).
`n`	Integer. The number of columns/rows in the symmetric matrix.
`cs_type`	String. One of `c("csr", "csc")`. Which type of list to return. Other values raise an error.

Value

A list containing:

- data - Numeric vector with the non-zero values.
- indptr - Integer vector with the index pointers.
- indices - Integer vector with the 0-based indices.
- nrow - Number of rows.
- ncol - Number of columns.
- cs_type - `"csr"` or `"csc"`.

`rs_voom_normalise` *Voom-transform a count matrix*

Description

[**Experimental**] limma's `voom` on counts that are already filtered: log2-CPM against the supplied (effective) library sizes, the mean-variance trend and the precision weights. No filtering and no normalisation happen in here; pass `lib.size * norm.factors` as `lib_size` to get voom on a normalised DGEList.

Usage

```
rs_voom_normalise(counts, design, lib_size, span, adaptive_span)
```

Arguments

`counts`	Integer or double matrix. Raw counts of genes x samples.
`design`	Numeric matrix. The design matrix of samples x coefficients. Must be full rank.
`lib_size`	Numeric vector. The effective library size per sample.
`span`	Numeric. Lowess span, only used if `adaptive_span = FALSE`.
`adaptive_span`	Boolean. Derive the span from the number of genes, as limma does since 3.56.

Value

A list with the following elements

- e - Numeric matrix. The log2-CPM values, genes x samples. limma's `E`.
- weights - Numeric matrix. The precision weights, genes x samples.
- trend_x - The mean-variance trend abscissae (average log2 count).
- trend_y - The mean-variance trend ordinates (sqrt standard deviation).
- amean - Average log2-CPM per gene.

References

Law, et al., Genome Biol, 2014

`rs_wnn` *Run the weighted nearest neighbour algorithm*

Description

[**Experimental**] This provides a Rust-based implementation of the WNN algorithm from Hao, et al. Both embeddings are L2-normalised per cell, a kNN graph with `knn_range` neighbours is built per modality, and the per-cell modality weights are then used to fuse both into one kNN graph.

Usage

```
rs_wnn(modality_emb_one, modality_emb_two, wnn_params, seed, verbose)
```

Arguments

`modality_emb_one`	Numerical matrix of the first modality, cells x dimensions. For example the PCA (or other embeddings) from the transcriptomics.
`modality_emb_two`	Numerical matrix of the second modality, same cells in the same row order. For example the PCA (or other embeddings) from the ADT counts.
`wnn_params`	Named list. The weighted nearest neighbour parameters.
`seed`	Integer. For reproducibility purposes.
`verbose`	Integer. `0L` - quiet; `1L` - normal verbosity; `2L` - detailed verbosity.

Value

A list with

- indices - Integer matrix of cells x neighbours with the indices (0-indexed!) of the weighted nearest neighbours.
- dist - Numerical matrix with the distances to these neighbours.
- dist_metric - String. Always `"kernelised pseudo-distance"`.
- modality_one_weights - Numerical vector with the per-cell weights of the first modality.
- modality_two_weights - Numerical vector with the per-cell weights of the second modality.

References

Hao et al., Cell, 2021

`run_bpca` *Bayesian PCA on a matrix with missing values*

Description

[**Experimental**] Port of `bpca()` from pcaMethods. Variational Bayes with an ARD prior per component, so components that the data do not support shrink towards zero rather than fitting noise. Missing entries are imputed from the fit.

Usage

```
run_bpca(x, bpca_params = params_bpca(), .verbose = TRUE)
```

Arguments

`x`	Numeric matrix. Rows = samples, columns = features. `NA` (or `NaN`) marks a missing value. No row or column may be entirely missing and observed values must be finite.
`bpca_params`	List, see `params_bpca()`. `n_pcs` must not exceed `min(dim(x))`.
`.verbose`	Boolean or integer. Verbosity.

Details

Deterministic, the start comes from an SVD. BPCA loadings are not orthonormal, and `r2_cum` is computed on the observed entries only, as in pcaMethods. Signs of the components are arbitrary.

Value

A list with:

- scores - Samples x `n_pcs`.
- loadings - Features x `n_pcs`, not orthonormal.
- r2_cum - Cumulative R^2 per component on the observed entries.
- centre - Column centres that were subtracted.
- scale - Column scales that were divided out.
- completed - `x` with the missing entries imputed. Observed entries are untouched.
- noise_var - Residual variance, `1 / tau`.
- n_iter - Variational steps run.
- converged - Whether `tol` was reached before `max_iter`.

References

Oba, et al., Bioinformatics, 2003; Stacklies, et al., Bioinformatics, 2007

Examples

```
set.seed(42L)
x <- matrix(rnorm(50L * 10L), nrow = 50L, ncol = 10L)
x[sample(length(x), 50L)] <- NA
bpca_res <- run_bpca(x, .verbose = FALSE)
head(bpca_res$scores)
```

---

run_cell_qc *Run outlier detection on per-cell QC metrics*

Description

Helper function to run initial quality control on cells.

Usage

```
run_cell_qc(
  metrics,
  cells_to_keep,
  directions = NULL,
  threshold = 3,
  groups = NULL,
  hard_thresholds = NULL,
  mad = TRUE
)
```

Arguments

`metrics`	Named list of numeric vectors.
`cells_to_keep`	Integer. 0-indexed cell positions.
`directions`	Named character vector, one of `"twosided"`, `"below"`, `"above"`. Defaults to `"twosided"`.
`threshold`	Numeric. MADs for MAD outlier detection.
`groups`	Optional grouping vector.
`hard_thresholds`	Optional named list of numeric vectors with `lower` and/or `upper` bounds, e.g. `list(MT = c(upper = 15))`. Applied independent of groups.
`mad`	Logical. If `FALSE`, skip MAD entirely; `hard_thresholds` must then be supplied.

Value

A `CellQc` object.

Examples

```
# MAD outlier detection over two metrics at once
set.seed(42L)
metrics <- list(
  lib_size = c(rnorm(99, 1000, 100), 50),
  pct_mt = runif(100, 0, 20)
)
run_cell_qc(
  metrics,
  cells_to_keep = 0:99,
  directions = c(lib_size = "below", pct_mt = "above")
)
```

`run_cell_qc_fixed` *Fixed-threshold cell QC*

Description

Thin wrapper around `run_cell_qc` with MAD disabled.

Usage

```
run_cell_qc_fixed(metrics, cells_to_keep, hard_thresholds, groups = NULL)
```

Arguments

`metrics`	Named list of numeric vectors.
`cells_to_keep`	Integer. 0-indexed cell positions.
`hard_thresholds`	Required. See `run_cell_qc`.
`groups`	Optional grouping vector.

Examples

```
# a hard upper bound, no MAD anywhere
set.seed(42L)
qc <- run_cell_qc_fixed(
  metrics = list(pct_mt = runif(100, 0, 30)),
  cells_to_keep = 0:99,
  hard_thresholds = list(pct_mt = c(upper = 15))
)
sum(qc$combined)
```

`run_cistarget` *Main function to run CisTarget*

Description

The `bixverse` implementation of the RCisTarget workflow, one of the algorithms used in SCENIC, see Aibar, et al. You will need motif to target gene rankings, see `read_motif_ranking()` and the motif to TF annotations, see `read_motif_annotation_file()`.

Usage

```
run_cistarget(
  gs_list,
  rankings,
  annot_data,
  cis_target_params = params_cistarget(),
  .verbose = TRUE
)
```

Arguments

`gs_list`	Named list of character vectors. Each element is a gene set containing gene identifiers that must match row names in `rankings`.
`rankings`	Integer matrix. Motif rankings for genes. Row names are gene identifiers, column names are motif identifiers. Lower values indicate higher regulatory potential.
`annot_data`	data.table. Motif annotation database mapping motifs to transcription factors. Must contain columns: `motif`, `TF`, and `annotationSource`.
`cis_target_params`	List. Output of `params_cistarget()`: • auc_threshold - Numeric. Proportion of genes to use for AUC threshold calculation. Default 0.05 means top 5 percent of genes. • nes_threshold - Numeric. Normalised Enrichment Score threshold for determining significant motifs. Default is 3.0. • rcc_method - Character. Recovery curve calculation method: "approx" (approximate, faster) or "icistarget" (exact, slower). • high_conf_cats - Character vector. Annotation categories considered high confidence (e.g., "directAnnotation", "inferredBy_Orthology"). • low_conf_cats - Character vector. Annotation categories considered lower confidence (e.g., "inferredBy_MotifSimilarity").
`.verbose`	Boolean. Controls verbosity of the function.

Value

data.table with enriched motifs and corresponding statistics and high & low confidence TFs for each gene set.

References

Aibar, et al., Nat Methods, 2017

Examples

```
# motif enrichment against a tiny synthetic ranking database
rankings <- matrix(
  c(1L, 5L, 4L, 2L, 2L, 2L, 1L, 5L, 4L, 3L, 2L, 1L, 3L, 1L, 5L, 4L,
    5L, 4L, 3L, 3L),
  nrow = 5,
  byrow = TRUE,
  dimnames = list(sprintf("gene_%i", 1:5), sprintf("motif_%i", 1:4))
)
annot <- data.table::data.table(
  motif = sprintf("motif_%i", 1:4),
  TF = sprintf("TF%i", 1:4),
  annotationSource = factor(c(
    "directAnnotation",
    "inferredBy_Orthology",
    "inferredBy_MotifSimilarity",
    "inferredBy_MotifSimilarity_n_Orthology"
  ))
)
res <- run_cistarget(
  gs_list = list(set_a = c("gene_1", "gene_2", "gene_3")),
  rankings = rankings,
  annot_data = annot,
  cis_target_params = params_cistarget(
    auc_threshold = 1,
    nes_threshold = 0.2
  ),
  .verbose = FALSE
)
res[, c("gs_name", "motif", "nes")]
```

`run_edger_ql` *Run the edgeR quasi-likelihood workflow*

Description

Runs `filterByExpr()` -> `calcNormFactors()` -> `glmQLFit()` -> `glmQLFTest()` in Rust via the `edge-rs` crate, gated against edgeR 4.8.2.

The tested axis does not have to be genes. Anything with a counts matrix of features x samples goes through here, which is why the Milo neighbourhood test calls the same function with `filter = FALSE`.

By default this skips `estimateDisp()` and lets the fit estimate its own dispersion from the most abundant features. That is edgeR 4's own recommendation and where most of the runtime went. Set `legacy = TRUE` in `params_edger_ql()` for the pre-4.0 pipeline, which does run `estimateDisp()`.

Usage

```
run_edger_ql(
  counts,
  design,
  coef = NULL,
  contrast = NULL,
  edger_params = params_edger_ql()
)
```

Arguments

`counts`	Numeric matrix. Raw counts of features x samples, with rownames. Must not be normalised or log-transformed.
`design`	Numeric matrix. The design matrix of samples x coefficients, including the intercept. Usually the output of `stats::model.matrix()`. Needs at least two columns, since the null model has to retain one.
`coef`	Optional integer or character. Which coefficient(s) of `design` to drop from the null model, given as 1-based column positions or column names. Defaults to the last column, as edgeR does.
`contrast`	Optional numeric vector or matrix. Weights over the coefficients, one entry (or row) per column of `design`. Mutually exclusive with `coef`.
`edger_params`	A list, see `params_edger_ql()`. The list has the following parameters: • norm_method - String. Library size normalisation. One of `c("TMM", "TMMwsp", "RLE", "upperquartile", "none")`. • filter - Boolean. Run `filterByExpr()` before fitting. • min_mean - Numeric. Minimum mean count across samples. • robust - Boolean. Robust empirical Bayes squeezing. • legacy - Boolean. Take edgeR's pre-4.0 pipeline.

Value

A data.table with one row per feature that survived the filters:

- feature_id - The rowname of `counts`.
- log_fc - Log2 fold change of the tested coefficient or contrast.
- log_cpm - Average log2 counts per million.
- f_stat - The quasi-likelihood F statistic.
- p_value - Raw p-value.
- fdr - Benjamini-Hochberg adjusted p-value.

References

Chen, Lun and Smyth, F1000Research, 2016

Examples

```
# quasi-likelihood F test on the second design column
syn <- synthetic_bulk_cor_matrix()
grp <- factor(rep(c("ctrl", "case"), each = 50), levels = c("ctrl", "case"))
design <- stats::model.matrix(~grp)
res <- run_edger_ql(counts = syn$counts, design = design)
head(res)
```

---

run_gene_trends_sc *Fit gene trends over Palantir pseudotime*

Description

Assigns cells to branches from their fate probabilities, then fits a landmark Gaussian process with a Matern-5/2 kernel per branch, which is the Mellon-based estimator of the reference rather than the legacy GAM. Out comes one smooth curve per gene and branch, on a common pseudotime grid.

Whatever expression is handed over is what gets fitted. With `use_magic = FALSE` that is the normalised counts; with `TRUE` it is the imputed layer `run_magic_sc()` wrote. Imputing first smooths twice, which is a defensible presentation choice and not usually a necessary one.

Usage

```
run_gene_trends_sc(
  object,
  palantir_res,
  features = NULL,
  use_magic = FALSE,
  branch_params = params_sc_branch_selection(),
  gene_trend_params = params_sc_gene_trends(),
  .verbose = TRUE
)
```

Arguments

`object`	One of `SingleCells`, `SingleCellsSubset`, `MetaCells` or `SingleCellsMultiModal`.
`palantir_res`	`PalantirRes` class. The output of `run_palantir_sc()`, run over the same cells.
`features`	Optional character vector. The genes to fit. If `NULL`, every gene in the imputed layer is used, which needs `use_magic = TRUE`.
`use_magic`	Boolean. Fit the MAGIC imputed layer rather than the normalised counts. Needs `run_magic_sc()` to have been run.
`branch_params`	List. See `params_sc_branch_selection()`.
`gene_trend_params`	List. See `params_sc_gene_trends()`.

`.verbose` Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

A `GeneTrendsRes` S3 object with:

- trends - Long data.table with `branch`, `pseudotime`, `gene` and `expression`, ready to facet.
- branch_cells - Named list of the cell names selected for each branch.
- branches - Character vector with the branch names, taken from the column names of the fate probability matrix.
- params - List with the parameters the run used.
- run_info - List with `n_cells` and `jitter_used` per branch and the grid `resolution`.

On the defaults

Palantir's pseudotime is min-max scaled to `[0, 1]`, so the reference's `length_scale` of `1.0` spans the entire domain and its `sigma` of `1.0` sits at roughly the signal scale of log-normalised expression. The posterior is prior-dominated: it flattens genuine transient structure and resolves almost any gene into a smooth monotone or single-peaked curve. Shorten `length_scale` in `params_sc_gene_trends()` before believing a bump.

References

Setty, et al., Nat. Biotechnol., 2019.

Examples

```
# GP trends for five genes along the Palantir branches
sc <- demo_single_cells()
palantir <- run_palantir_sc(
  sc,
  early_cell = get_knn_obj(sc)$used_cells[1],
  palantir_params = params_sc_palantir(
    knn = 15L,
    num_waypoints = 100L,
    n_eigs = 3L,
    use_early_cell_as_start = TRUE,
    knn_params = list(knn_method = "exhaustive")
  ),
  .verbose = FALSE
)
res <- run_gene_trends_sc(
  sc,
  palantir_res = palantir,
  features = get_gene_names(sc)[1:5],
  gene_trend_params = params_sc_gene_trends(resolution = 50L),
  .verbose = FALSE
)
head(res$trends)
```

```
unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`run_lda` *Fit a latent Dirichlet allocation model*

Description

Topic model over a documents x terms count matrix, fitted by variational Bayes. The intended input is a binary matrix: this is the model behind cisTopic, where binarised cells x regions scATAC becomes cells x topics for clustering and topics x regions for region set discovery. Nothing here is ATAC-specific, so binarised regulon activity from `binarise_regulon_activity()` works the same way, as does any count matrix.

Usage

```
run_lda(x, k, lda_params = params_lda(), seed = 42L, .verbose = TRUE)
```

Arguments

`x`	Documents x terms matrix. Either a `dgCMatrix`, a `dgRMatrix`, or a dense numeric or logical matrix. Column names are required, row names are generated if absent.
`k`	Integer. Number of topics.
`lda_params`	List, see `params_lda()`.
`seed`	Integer. Seed for the variational initialisation.
`.verbose`	Boolean or integer. Verbosity.

Details

Binarising first is what makes a topic model the right tool. LDA on raw single cell counts is dominated by library size, which is precisely the effect cisTopic sidesteps.

Use `lda_k_sweep()` if you do not already know `k`.

Value

An `LdaResult` object.

References

Hoffman, Blei and Bach, NIPS, 2010; Bravo Gonzalez-Blas, et al., Nat Methods, 2019

Examples

```
# two planted term blocks recovered as two topics
set.seed(42L)
corpus <- matrix(rbinom(200L * 40L, 1L, 0.05), nrow = 200L, ncol = 40L)
corpus[1:100, 1:10] <- rbinom(1000L, 1L, 0.6)
corpus[101:200, 11:20] <- rbinom(1000L, 1L, 0.6)
colnames(corpus) <- sprintf("term_%02d", 1:40)
lda_res <- run_lda(corpus > 0, k = 2L, .verbose = FALSE)
lda_res
```

---

run_limma_voom *Wrapper for a Limma Voom analysis*

Description

Runs the limma-voom workflow (`calcNormFactors()` -> `voomLmFit()` -> `contrasts.fit()` -> `eBayes()` -> `topTable()`) in Rust via the `edge-rs` crate, gated against limma `3.66.0`. The design is `~ 0 + main_contrast + co_variates` and every requested contrast is tested separately. limma and edgeR are not needed.

Usage

```
run_limma_voom(
  meta_data,
  main_contrast,
  counts,
  contrast_list = NULL,
  co_variates = NULL,
  limma_params = params_limma_voom(),
  lib_size = NULL,
  .verbose = TRUE
)
```

Arguments

`meta_data`	data.table. The meta information about the experiment in which the contrast info (and potential co-variates) can be found. Rows need to be in the same order as the columns of `counts`.
`main_contrast`	String. Which column contains the main groups you want to test differential gene expression with the Limma-Voom workflow for.
`counts`	Numeric matrix. Raw counts of genes x samples, with gene identifiers as row names.
`contrast_list`	String vector or NULL. Optional string vector of contrast formatted as `"contrast1-contrast2"`. Default NULL will create all contrasts automatically.
`co_variates`	String or NULL. Optional co-variates you wish to consider during model fitting.

limma_params	List. The limma parameters, see params_limma_voom().
lib_size	Optional numeric vector. Library size per sample, in the column order of counts. NULL uses the column sums of counts. Pass the column sums from before gene filtering to match edgeR, which keeps those on a subset DGEList.
.verbose	Boolean. Controls verbosity of the function.

Value

A data.table with the columns of limma's topTable(confint = TRUE) (gene_id, logFC, CI.L, CI.R, AveExpr, t, P.Value, adj.P.Val, B) plus contrast, sorted by p-value within each contrast.

References

Law, et al., Genome Biol, 2014

Examples

```
# voom fit and topTable results for the single case vs control contrast
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
res <- run_limma_voom(
  meta_data = meta,
  main_contrast = "case_control",
  counts = syn$counts,
  .verbose = FALSE
)
head(res)
```

---

run_magic_sc *Impute a subset of genes with MAGIC*

Description

MAGIC smooths counts over the kNN graph: a row-stochastic diffusion operator is built from the graph stored on the object and applied n_steps times, so every cell becomes a weighted average of its neighbourhood. It makes sparse marker genes readable on an embedding and gives gene trends something less speckled to fit. For details, please refer to van Dijk, et al.

Deliberately restricted to a gene subset. The output is dense and the binary store is never touched, so pick the genes you want to plot or fit and impute those. Genome-wide imputation is not offered.

Each call replaces the whole layer. Read it back with get_magic(), or hand it to any of the extractors with layer = "magic".

Usage

```
run_magic_sc(
  object,
  features,
  modality = c("rna", "adt", "wnn"),
  magic_params = params_sc_magic(),
  .verbose = TRUE
)
```

Arguments

`object`	One of `SingleCells`, `SingleCellsSubset` or `SingleCellsMultiModal`. `MetaCells` are already aggregated and are not supported.
`features`	Character vector. The genes to impute. Anything not found in the object is dropped with a warning.
`modality`	String. One of `c("rna", "adt", "wnn")`. Which kNN graph does the smoothing. Anything but `"rna"` requires a `SingleCellsMultiModal` object. The counts themselves always come from the RNA store, and the layer is always written to the RNA cache.
`magic_params`	List. See `params_sc_magic()`.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

The object with an `ScMagic` layer in its cache.

What this does to your results

Neighbourhoods overlap, so smoothing over them makes nearby cells resemble each other in every gene at once. Gene-gene correlation after imputation is inflated, badly, and the inflation is a property of the graph rather than of the biology. Do not feed imputed values into Hotspot, SCENIC, differential correlation or CoReMo: those methods measure exactly the quantity MAGIC manufactures. Visualisation and trend fitting are what it is for.

The operator preserves per-cell mass, so imputed values sit on the scale of whatever went in. Imputing raw counts and imputing log-normalised counts are different operations rather than the same one rescaled, see the `layer` element of `params_sc_magic()`.

References

van Dijk, et al., Cell, 2018.

Examples

```
# diffusion imputation of five genes for plotting
sc <- demo_single_cells()
sc <- run_magic_sc(
  sc,
```

```
  features = get_gene_names(sc)[1:5],
  .verbose = FALSE
)
dim(get_magic(sc)$data)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`run_paga_sc` *Run PAGA graph abstraction*

Description

PAGA abstracts the single cell kNN graph into a cluster-level graph, where the edge weight between two clusters measures how much more connected they are than expected under a random null model. It also returns the maximum spanning forest of that graph, which is the backbone typically plotted. For details, please refer to Wolf, et al.

Only the kNN indices are used, no distances and no expression data, so the method is safe on any of the available graphs, including the WNN one. Run `find_neighbours_sc()` and a clustering first.

Usage

```
run_paga_sc(
  object,
  cluster_col,
  modality = c("rna", "adt", "wnn"),
  .verbose = TRUE
)
```

Arguments

`object`	One of `SingleCells`, `SingleCellsSubset`, `MetaCells` or `SingleCellsMultiModal`.
`cluster_col`	String. The obs column holding the cluster assignments. Empty factor levels are retained.
`modality`	String. One of `c("rna", "adt", "wnn")`. Which kNN graph to run over. Anything but `"rna"` requires a `SingleCellsMultiModal` object.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

A `PagaRes` S3 object with:

- connectivities - Symmetric sparse matrix of clusters x clusters with a zero diagonal. Values lie in `(0, 1]`.
- connectivities_tree - Maximum spanning forest of `connectivities`, carrying the original connectivity values on the retained edges.
- sizes - data.table with `cluster` and `n_cells`.
- params - List with `cluster_col` and `modality`.

References

Wolf, et al., Genome Biol., 2019.

Examples

```
# cluster level graph abstraction of the Leiden clusters
sc <- demo_single_cells()
sc <- find_clusters_sc(sc, res = 1.0)
res <- run_paga_sc(sc, cluster_col = "leiden_clustering", .verbose = FALSE)
res$sizes

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

run_palantir_sc *Run Palantir trajectory inference*

Description

Palantir models differentiation as a Markov chain over a diffusion map of the cells. It returns a pseudotime per cell, the probability of each cell reaching each terminal state, and the differentiation entropy derived from those probabilities. For details, please refer to Setty, et al.

The kNN graph stored on the object feeds the diffusion kernel, so run `find_neighbours_sc()` first. The geodesics themselves are measured over a second kNN graph that Palantir builds internally on the multiscale space, controlled by the `knn` element of `params_sc_palantir()`.

For `SingleCellsMultiModal` the `"wnn"` graph works, but its distances are kernel-derived and not a metric, so the diffusion map on top is a heuristic on a heuristic. You will get a warning saying as much.

Usage

```
run_palantir_sc(
  object,
  early_cell,
  terminal_states = NULL,
  modality = c("rna", "adt", "wnn"),
  palantir_params = params_sc_palantir(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	One of `SingleCells`, `SingleCellsSubset`, `MetaCells` or `SingleCellsMultiModal`.
`early_cell`	String. Name of the cell to start the trajectory from. Must be one of the cells the kNN graph was built over.

terminal_states
: Optional character vector. Names of the terminal state cells. If NULL, they are detected from the waypoint Markov chain. Name the vector (e.g. c(Ery = "Run4_2005...")) and those labels become the column names of branch_probs, which saves relabelling the fate columns by hand downstream.

modality
: String. One of c("rna", "adt", "wnn"). Which kNN graph to run over. Anything but "rna" requires a SingleCellsMultiModal object.

palantir_params
: List. See params_sc_palantir().

seed
: Integer. For reproducibility.

.verbose
: Boolean or integer. Controls verbosity and returns run times. FALSE -> quiet, TRUE or 1L -> normal verbosity, 2L -> detailed verbosity.

Value

A PalantirRes S3 object with:

- pseudotime - data.table with cell_id, pseudotime (min-max scaled to [0, 1]; the start cell is not pinned to 0, and a start cell far from 0 means the refinement disagreed with the anchor) and entropy.
- branch_probs - Numeric matrix of cells x terminal states with the fate probabilities. Rows need not sum to one, as sub-threshold values are zeroed without renormalisation.
- terminal_states - Character vector with the terminal state cell names, carrying the labels of a named terminal_states argument as its own names. Sets the column order of branch_probs.
- waypoints - Character vector with the waypoint cell names. The first element is the start cell.
- start_cell - String. The start cell that was actually used.
- multiscale - Numeric matrix of cells x components with the multiscale diffusion components.
- run_info - List with iterations, converged, eigen_converged, eigen_residual, repair_edges, stranded_waypoints, n_waypoints and modality. A eigen_converged of FALSE means the diffusion eigensolve ran out of restarts and the embedding is underresolved.

References

Setty, et al., Nat. Biotechnol., 2019.

Examples

```
# pseudotime from an arbitrary start cell on the demo kNN graph
sc <- demo_single_cells()
res <- run_palantir_sc(
  sc,
  early_cell = get_knn_obj(sc)$used_cells[1],
```

```
  palantir_params = params_sc_palantir(
    knn = 15L,
    num_waypoints = 100L,
    n_eigs = 3L,
    use_early_cell_as_start = TRUE,
    knn_params = list(knn_method = "exhaustive")
  ),
  .verbose = FALSE
)
head(res$pseudotime)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`run_ppca` *Probabilistic PCA on a matrix with missing values*

Description

[**Experimental**] Port of `ppca()` from pcaMethods. Fits the principal subspace by EM on the observed entries only, then imputes the missing entries from it. Use it to get a PCA or a completed matrix out of data with sporadic `NA`s, for example proteomics or metabolomics intensities.

Usage

```
run_ppca(x, ppca_params = params_ppca(), .verbose = TRUE)
```

Arguments

`x`	Numeric matrix. Rows = samples, columns = features. `NA` (or `NaN`) marks a missing value. No row or column may be entirely missing and observed values must be finite.
`ppca_params`	List, see `params_ppca()`. `n_pcs` must not exceed `min(dim(x))`.
`.verbose`	Boolean or integer. Verbosity.

Details

The initial loadings are drawn from `ppca_params$seed` with the Rust RNG, so results agree with pcaMethods at convergence, not iterate by iterate. PPCA loadings are orthonormal. Signs of the components are arbitrary.

Value

A list with:

- scores - Samples x `n_pcs`.
- loadings - Features x `n_pcs`, orthonormal.
- r2_cum - Cumulative R^2 per component on the completed matrix.

- centre - Column centres that were subtracted.
- scale - Column scales that were divided out.
- completed - `x` with the missing entries imputed. Observed entries are untouched.
- noise_var - Residual variance outside the subspace.
- n_iter - EM iterations run.
- converged - Whether `tol` was reached before `max_iter`.

References

Roweis, NIPS, 1998; Tipping and Bishop, J R Stat Soc B, 1999; Stacklies, et al., Bioinformatics, 2007

Examples

```
set.seed(42L)
x <- matrix(rnorm(50L * 10L), nrow = 50L, ncol = 10L)
x[sample(length(x), 50L)] <- NA
ppca_res <- run_ppca(x, .verbose = FALSE)
head(ppca_res$scores)
```

---

run_snf *Run the SNF algorithm*

Description

This function will run the SNF algorithm on top of the adjacency matrices found in the object. You can also optionally specify which adjacency matrices to use via the `to_include` parameter.

Usage

```
run_snf(object, to_include = NULL, params = NULL)
```

Arguments

`object`	The underlying class, see `SimilarityNetworkFusion()`.
`to_include`	Optional string, if you wish to only use a subset of the generated adjacency matrices. If `NULL` all matrices will be used for the fusion process.
`params`	Optional List. If you wish to overwite the already set up parameters for SNF, see `params_snf()`. If `NULL`, the settings from within the object will be used. If not NULL, the new parameters will be used for this modality specifically and only for this modality!

Value

The class with added adjacency matrix based on the SNF algorithm.

Examples

```
# fuse a continuous and a categorical modality
set.seed(42)
continuous <- matrix(rnorm(120), nrow = 12, ncol = 10)
rownames(continuous) <- sprintf("sample_%02i", 1:12)
colnames(continuous) <- sprintf("feature_%i", 1:10)
clinical <- data.table::data.table(
  sample_id = rownames(continuous),
  sex = factor(sample(c("M", "F"), 12, replace = TRUE)),
  stage = factor(sample(c("I", "II", "III"), 12, replace = TRUE))
)
object <- SimilarityNetworkFusion(
  data = continuous,
  data_name = "continuous",
  snf_params = params_snf(k = 3L)
)
object <- add_snf_data_modality(object, clinical, data_name = "clinical")
object <- run_snf(object)
dim(get_snf_final_mat(object))
```

---

save_h5ad *Save a* SingleCells *object to h5ad*

Description

Writes the counts, the DuckDB obs/var tables and whatever is cached in memory (PCA, embeddings, sNN graph) to a spec-compliant h5ad file, so that the experiment can be handed to ScanPy or shared. The counts are streamed cell batch by cell batch and never fully materialised in R.

Only the cells that are currently kept are written, i.e. the export matches what `object[]` and `object[[]]` return. Counts are stored as `float32`, which is the ScanPy convention.

A `SingleCellsMultiModal` object inherits this method and exports its RNA modality; the ADT layer is not written.

Usage

```
save_h5ad(
  object,
  h5_path,
  assay = c("raw", "norm"),
  chunk_size = 10000L,
  overwrite = TRUE,
  .verbose = TRUE
)
```

Arguments

object	SingleCells class.
h5_path	File path to write the h5ad file to.
assay	String. One of c("raw", "norm"). Which count assay to place in X.
chunk_size	Integer. Number of cells per streaming batch. Defaults to 10000L.
overwrite	Boolean. Shall an existing file be overwritten. Defaults to TRUE.
.verbose	Boolean. Controls the verbosity of the function.

Value

Returns the path to the written file, invisibly.

---

save_sc_exp_to_disk *Save memory-bound data to disk*

Description

Helper function that stores the memory-bound data to disk for checkpointing or when you close the session for quick recovery of prior work. You have the option to save as ".rds" or ".qs2" (you need to have the package "qs2" installed for this option!).

Usage

```
save_sc_exp_to_disk(object, type = c("qs2", "rds"))
```

Arguments

object	SingleCells, MetaCells or SingleCellsMultiModal class.
type	String. One of c("qs2", "rds"). Defines which binary format to use. Will default to "qs2" for speed.

Value

NULL, invisibly. Called for the side effect of writing the in-memory maps and caches next to the counts. It does not return the object, so do not assign the result.

Examples

```
# checkpoint the in-memory map and cache next to the counts
sc <- demo_single_cells()
save_sc_exp_to_disk(sc, type = "rds")
list.files(sc@dir_data)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

sc_knn_to_nearest_neighbours

Convert SingleCellNearestNeighbour to manifoldsR NearestNeighbours

Description

Converts the bixverse single cell kNN class into the manifoldsR `NearestNeighbours` class for use in UMAP, tSNE, etc. Note that the bixverse class stores 0-indexed indices whereas manifoldsR expects 1-indexed, so the conversion handles that.

Usage

```
sc_knn_to_nearest_neighbours(x)
```

Arguments

`x`	`SingleCellNearestNeighbour` class.

Value

A `NearestNeighbours` class compatible with manifoldsR.

Examples

```
# hand the single cell kNN graph over to manifoldsR
sc <- demo_single_cells()
knn <- generate_knn_sc(sc, .validate_index = FALSE, .verbose = FALSE)
sc_knn_to_nearest_neighbours(knn)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

sc_pipeline

Construct an empty single cell pipeline

Description

Linear container of `ScSteps`. Append steps with `%>>%` and execute with `apply_pipeline()`. Pipelines are inert until applied; steps can be inspected via `pipeline$steps`.

Usage

```
sc_pipeline()
```

Value

An empty `ScPipeline` object.

Examples

```
# an empty container, filled with `%>>%`
sc_pipeline() %>>%
  step_hvg_sc(hvg_no = 30L) %>>%
  step_pca_sc(no_pcs = 10L)
```

---

scdblfinder_sc *Run scDblFinder doublet detection on a SingleCells object*

Description

Cluster-aware doublet detection using engineered features and a gradient-boosted classifier. See Germain et al., F1000Research, 2022.

Usage

```
scdblfinder_sc(
  object,
  scdblfinder_params = params_scdblfinder(),
  return_features = FALSE,
  cells_to_use = NULL,
  group_by = NULL,
  streaming = NULL,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`scdblfinder_params`	List. Parameters from `params_scdblfinder()`.
`return_features`	Boolean. Shall the features used to train the classifier be returned.
`cells_to_use`	Optional string. Names of the cells to use for the run of the boosted doublet detection. Useful when you wish to run doublet detection on individual batches within your data. The object returned will be specifically using these cells.
`group_by`	Optional grouping variable. Useful if you want to run the method on a per-sample basis.
`streaming`	Optional Boolean. Shall the data be streamed in. Useful for larger data sets where you wish to avoid loading in the whole data. If `NULL`, will automatically detect.
`seed`	Integer. Seed for reproducibility.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

An S3 object of class `ScDblFinderRes` containing:

predicted_doublets Logical vector of doublet calls.

doublet_score Numeric vector of classifier probabilities.

cxds_scores Numeric vector of the cxds scores.

weighted Numeric vector of the weighted scores.

threshold The threshold used for calling.

cluster_labels Integer vector of final cluster assignments.

detected_doublet_rate Fraction of cells called as doublets.

with `cell_indices` stored as an attribute.

Examples

```
# cluster aware doublet calls from the gradient boosted classifier
sc <- demo_single_cells(prepped = FALSE)
scdblfinder_sc(
  sc,
  scdblfinder_params = params_scdblfinder(
    pca = list(no_pcs = 10L),
    n_genes = 25L,
    cxds_genes = 25L
  ),
  .verbose = FALSE
)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

scenic_gene_filter_sc

Filter genes for SCENIC GRN inference

Description

Filters genes by minimum total counts and minimum expressed-cell fraction using the SCENIC inclusion criteria. Returns a character vector of gene identifiers passing both filters.

Usage

```
scenic_gene_filter_sc(
  object,
  scenic_params = params_scenic(),
  cells_to_take = NULL,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells`, `MetaCells` (or potentially other) class.
`scenic_params`	List. SCENIC parameters, see `params_scenic()`. Only `min_counts` and `min_cells` are used by this function.
`cells_to_take`	Optional string vector. Cell identifiers to restrict to. If `NULL`, defaults to all filtered cells in the class.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

A character vector of gene identifiers passing the SCENIC inclusion criteria.

Examples

```
# genes clearing the SCENIC count and prevalence thresholds
sc <- demo_single_cells()
genes <- scenic_gene_filter_sc(
  sc,
  scenic_params = params_scenic(min_counts = 100L, min_cells = 0.05),
  .verbose = FALSE
)
length(genes)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`scenic_grn_sc` *Run SCENIC GRN inference*

Description

Runs SCENIC GRN inference on the provided genes using the specified transcription factors as predictors. Returns a `ScenicGrn` object containing the TF-gene importance matrix for further processing.

Usage

```
scenic_grn_sc(
  object,
  tf_ids,
  scenic_params = params_scenic(),
  genes_to_take = NULL,
  cells_to_take = NULL,
  streaming = NULL,
  random_seed = 42L,
  .verbose = TRUE
)
```

Arguments

object	SingleCells, MetaCells (or potentially other) class.
tf_ids	Character vector. Transcription factor gene identifiers to use as predictors. Must be a subset of gene identifiers present in the object.
scenic_params	List. SCENIC parameters, see params_scenic().
genes_to_take	Optional character vector. Target gene identifiers. If NULL, genes are selected automatically via scenic_gene_filter_sc() using the min_counts and min_cells thresholds in scenic_params.
cells_to_take	Optional string vector. Cell identifiers to restrict to. If NULL, defaults to all filtered cells in the class.
streaming	Optional Boolean. Shall the data be streamed in. Useful for larger data sets where you wish to avoid loading in the whole data. If NULL, will automatically detect. Ignored when applied to MetaCells.
random_seed	Integer. Used for reproducibility. Defaults to 42L.
.verbose	Boolean or integer. Controls verbosity and returns run times. FALSE -> quiet, TRUE or 1L -> normal verbosity, 2L -> detailed verbosity.

Details

TF identifiers that are not found in the object's gene list are silently dropped with a warning indicating how many were removed. TF indices are intersected with the target gene indices so that TFs not passing the gene filter are excluded from the predictor set but remain as potential targets if present in genes_to_take. You have the option to generate the TF-gene importance values with three distinct methods. For the random_forest and the extratrees version, a batching strategy is applied in the default settings. Correlated genes are identified and clustered together via k-means clustering on the feature loadings of the PCA. These are then divided into batches of gene_batch_size and the regression learners are leveraging multi-target regression to fit all genes in the batch in one go. This massively accelerates the algorithm and the importance values per gene-TF pair are calculated then individually. Due to the batching by similar gene, the signal dilution is limited. If you wish to run the traditional approach, you can set gene_batch_size to 1L or use the grnboost2 learner that can only fit one gene at a given time.

Value

A ScenicGrn object.

Examples

```
# TF to gene importances from a small random forest
sc <- demo_single_cells()
res <- scenic_grn_sc(
  sc,
  tf_ids = get_gene_names(sc)[1:5],
  scenic_params = params_scenic(
    min_counts = 100L,
    learner_params = list(n_trees = 20L)
```

```
  ),
  .verbose = FALSE
)
res

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`score_clusters` *Score clusters based on ScType*

Description

Score clusters based on ScType

Usage

```
score_clusters(x, cluster_labels)

## S3 method for class 'ScTypeResults'
score_clusters(x, cluster_labels)
```

Arguments

`x`	`ScTypeResults` object.
`cluster_labels`	Integer vector. Cluster assignment, of length of the scored cells.

Value

A `data.table` with cluster_id, cell_type, scores and n_cells.

Examples

```
# aggregate the per cell ScType scores onto a clustering
sc <- demo_single_cells()
markers <- data.table::data.table(
  cell_type = rep(sprintf("cell_type_%i", 1:3), each = 10),
  gene_id = sprintf("gene_%02d", 1:30)
)
cell_markers <- prepare_cell_markers(obj = sc, marker_df = markers)
scores <- calc_sc_type_scores(
  sc,
  cell_marker_list = cell_markers,
  .verbose = FALSE
)
clusters <- fast_cluster_sc(
  sc,
  resolutions = 1.0,
  n_centroids = 30L,
  .verbose = FALSE
```

```
)
head(score_clusters(scores, cluster_labels = get_data(clusters)$res_1))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`scrublet_sc` *Doublet detection with Scrublet*

Description

This function implements the doublet detection from Scrublet, see Wolock, et al. Briefly, arteficial doublets are being generated from the data via random combination of initial cells. Highly variable genes (HVG) are being identified and the observed cells are being projected on a PCA space. Subsequently, the simulated doublets are being projected on the same PCA space given the same HVGs; kNN graphs are being generated and a kNN classifier is used to assign a probability that a given cell in the original data is a doublet. For more details, please check the publication.

Usage

```
scrublet_sc(
  object,
  scrublet_params = params_scrublet(),
  seed = 42L,
  streaming = NULL,
  cells_to_use = NULL,
  group_by = NULL,
  return_combined_pca = FALSE,
  return_pairs = FALSE,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`scrublet_params`	A list with the final scrublet parameters, see `params_scrublet()` for full details.
`seed`	Integer. Random seed.
`streaming`	Optional Boolean. Shall the data be streamed in. Useful for larger data sets where you wish to avoid loading in the whole data. If `NULL`, will automatically detect.
`cells_to_use`	Optional string. Names of the cells to use for the generation of the Scrublet. Useful when you wish to run doublet detection on individual batches within your data. The object returned will be specifically using these cells.

`group_by` Optional grouping variable. Useful if you want to run the method on a per-sample basis.

`return_combined_pca` Boolean. Shall the PCA of the observed cells and simulated doublets be returned.

`return_pairs` Boolean. Shall the pairs be returned.

`.verbose` Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

A `scrublet_res` class that has with the following items:

- predicted_doublets - Boolean vector indicating which observed cells predicted as doublets (TRUE = doublet, FALSE = singlet).
- doublet_scores_obs - Numerical vector with the likelihood of being a doublet for the observed cells.
- doublet_scores_sim - Numerical vector with the likelihood of being a doublet for the simulated cells.
- doublet_errors_obs - Numerical vector with the standard errors of the scores for the observed cells.
- z_scores - Z-scores for the observed cells. Represents: `score - threshold / error`.
- threshold - Used threshold.
- detected_doublet_rate - Fraction of cells that are called as doublet.
- detectable_doublet_fraction - Fraction of simulated doublets with scores above the threshold.
- overall_doublet_rate - Estimated overall doublet rate. Should roughly match the expected doublet rate.
- pca - Optional PCA embeddings across the original cells and simulated doublets.
- pair_1 - Optional index of the parent cell 1 of the simulated doublets.
- pair_2 - Optional index of the parent cell 2 of the simulated doublets.

References

Wollock, et al., Cell Syst, 2020

Examples

```
# simulated doublet scoring on 500 synthetic cells
sc <- demo_single_cells(prepped = FALSE)
scrublet_sc(
  sc,
  scrublet_params = params_scrublet(
    pca = list(no_pcs = 10L),
    hvg = list(min_gene_var_pctl = 0.0),
    n_bins = 20L
```

```
  ),
  .verbose = FALSE
)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`sct_corrected_counts_sc` *Write scTransform-corrected counts to a new store*

Description

Reverses the residual transform with every latent variable, the library size included, held at its median. The depth structure goes, the per-sample intercept stays.

The result is a new store on disk, not a layer on `object`. Its gene axis is the model's, so it is narrower than the source and the indices do not line up, which is why the observation and variable tables are rebuilt rather than copied.

Usage

```
sct_corrected_counts_sc(
  object,
  dir_out = NULL,
  build_cell_store = TRUE,
  overwrite = FALSE,
  gene_batch_size = NULL,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `SingleCellsSubset` class with a scTransform fit attached. The analytic Pearson model has no corrected-count equivalent.
`dir_out`	String or `NULL`. Directory to write to. `NULL` uses `sct_corrected` inside the object's own data directory.
`build_cell_store`	Boolean. Also write the `counts_cells.bin` companion and the database, giving back a `SingleCells` rather than a path. Costs a second pass over the data.
`overwrite`	Boolean. Overwrite an existing store in `dir_out`.
`gene_batch_size`	Integer or `NULL`. Genes held in memory per batch.
`.verbose`	Boolean or Integer. Controls verbosity.

Value

With `build_cell_store = TRUE` a new `SingleCells` over the corrected counts, otherwise the path of the gene-major file, invisibly.

Examples

```
# corrected counts as a fresh object
sc <- demo_single_cells(prepped = FALSE)
sc <- fit_residuals_sc(sc, .verbose = FALSE)
corrected <- sct_corrected_counts_sc(sc, .verbose = FALSE)
dim(corrected)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

set_bonsai_embedding *Store the leaf coordinates of a Bonsai tree as an embedding*

Description

Writes the 2D position of every leaf, cell or metacell, into the object's embeddings, so the usual embedding plots can use it. The tree has to have been built over the object's current cells (or all of its metacells), in the same order.

Usage

```
set_bonsai_embedding(object, tree, name = "bonsai")
```

Arguments

`object`	`SingleCells` or `MetaCells` class.
`tree`	A `BonsaiTree`, from `bonsai_sc()` on this object.
`name`	String. Name of the embedding.

Value

The object with the embedding added.

Examples

```
# Bonsai leaf coordinates next to the other embeddings
sc <- demo_single_cells(
  prepped = FALSE,
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 50L)
)
tree <- bonsai_sc(sc, .verbose = FALSE)
sc <- set_bonsai_embedding(sc, tree)
get_available_embeddings(sc)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`set_magic` *Set/add the MAGIC imputed layer*

Description

Set/add the MAGIC imputed layer

Usage

```
set_magic(x, magic, ...)

## S3 method for class 'ScCache'
set_magic(x, magic, ...)

## S7 method for class <bixverse::SingleCells>
set_magic(x, magic, ...)

## S7 method for class <bixverse::SingleCellsSubset>
set_magic(x, magic, ...)
```

Arguments

`x`	An object to add the imputed layer to.
`magic`	`ScMagic` class with the imputed counts.
`...`	Other parameters.

Value

The object with the imputed layer attached.

Examples

```
# the imputed layer taken out and put back
sc <- demo_single_cells()
sc <- run_magic_sc(sc, features = get_gene_names(sc)[1:5], .verbose = FALSE)
magic <- get_magic(sc)
sc <- set_magic(remove_magic(sc), magic)
dim(get_magic(sc)$data)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`set_residual_fit` *Set/add the fitted residual model*

Description

Set/add the fitted residual model

Usage

```
set_residual_fit(x, residual_fit, ...)

## S7 method for class <bixverse::MetaCells>
set_residual_fit(x, residual_fit, ...)

## S3 method for class 'ScCache'
set_residual_fit(x, residual_fit, ...)

## S7 method for class <bixverse::SingleCells>
set_residual_fit(x, residual_fit, ...)

## S7 method for class <bixverse::SingleCellsSubset>
set_residual_fit(x, residual_fit, ...)
```

Arguments

`x`	An object to add the fitted model to.
`residual_fit`	`ScResidualFit` class, as returned by `fit_residuals_sc()`.
`...`	Other parameters.

Value

The object with the fitted model attached.

Examples

```
# the fitted model taken out and put back
sc <- demo_single_cells()
sc <- fit_residuals_sc(sc, method = "analytic_pearson", .verbose = FALSE)
fit <- get_residual_fit(sc)
sc <- set_residual_fit(remove_residual_fit(sc), fit)
get_residual_fit(sc)$method

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

set_sc_new_obs_col *Add a new column to the obs table*

Description

Add a new column to the obs table

Usage

```
set_sc_new_obs_col(object, col_name, new_data)
```

Arguments

object	SingleCells class.
col_name	String. The name of the column to add.
new_data	Atomic vector. The data to add to the column. Needs to be of same length as get_cells_to_keep() and have the same order!

Value

The class with updated obs table in the DuckDB

Examples

```
# one new column, in the order of the cells that passed quality control
sc <- demo_single_cells(prepped = FALSE)
sc <- set_sc_new_obs_col(
  sc,
  col_name = "arm",
  new_data = rep(c("ctrl", "treated"), length.out = dim(sc)[1])
)
table(get_sc_obs(sc)$arm)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

set_sc_new_obs_col_multiple *Add multiple new columns to the obs table*

Description

Add multiple new columns to the obs table

Usage

```
set_sc_new_obs_col_multiple(object, new_data)
```

Arguments

`object`	`SingleCells` class.
`new_data`	Named list. The names will be the column names and the elements will be added to the obs table. Needs to be of same length as `get_cells_to_keep()` and have the same order!

Value

The class with updated obs table in the DuckDB

Examples

```
# several columns in one go
sc <- demo_single_cells(prepped = FALSE)
n_cells <- dim(sc)[1]
sc <- set_sc_new_obs_col_multiple(
  sc,
  new_data = list(
    arm = rep(c("ctrl", "treated"), length.out = n_cells),
    donor = rep(sprintf("donor_%i", 1:4), length.out = n_cells)
  )
)
head(get_sc_obs(sc)[, c("arm", "donor")], 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`set_sc_new_var_cols` *Add a new column to the var table*

Description

Add a new column to the var table

Usage

```
set_sc_new_var_cols(object, data_list)
```

Arguments

`object`	`SingleCells` class.
`data_list`	Named list with the data to add.

Value

The class with updated var table in the DuckDB

Examples

```
# flag a couple of genes in the var table
sc <- demo_single_cells(prepped = FALSE)
sc <- set_sc_new_var_cols(
  sc,
  data_list = list(
    is_marker = get_gene_names(sc) %in% c("gene_01", "gene_02")
  )
)
head(get_sc_var(sc), 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

setnames_sc *Rename columns in the obs or var table*

Description

Renames the columns in the obs or var table of single cell-related classes.

Usage

```
setnames_sc(object, table = c("obs", "var"), old, new)
```

Arguments

`object`	`SingleCells`, `MetaCells` (or potentially other) class.
`table`	String. One of `c("obs", "var")`. In which of the tables to rename the columns.
`old`	Character vector. The old column names.
`new`	Character vector. The new column names.

Value

Invisible self

Examples

```
# rename a column in the obs table
sc <- demo_single_cells(prepped = FALSE)
sc <- setnames_sc(sc, table = "obs", old = "cell_grp", new = "cell_type")
head(get_sc_obs(sc)$cell_type, 3)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`seurat_cca_sc` *Run Seurat CCA integration*

Description

This function implements the canonical correlation analysis (CCA) anchor integration from Stuart, et al. For each pair of batches a shared CCA embedding is computed, mutual nearest neighbours in that embedding become anchors, these are filtered in gene space, scored by shared neighbours and finally used to apply a kernel-weighted correction on the union PCA embedding. Batches are merged in the order of their pairwise anchor counts.

This port deviates from Seurat in two places. It skips the per-gene `ScaleData` step and works from per-cell standardised log-normalised HVG expression, and it never materialises the `N1 x N2` cross-product (the canonical correlations come from a matrix-free randomised SVD). The correction runs on the embedding, not on full log-expression. In practice the anchor structure comes out close to identical at a fraction of the memory.

Usage

```
seurat_cca_sc(
  object,
  batch_column,
  batch_hvg_genes,
  cca_params = params_sc_seurat_cca(),
  use_precomputed_pca = FALSE,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`batch_column`	String. The column with the batch information in the obs data of the class.
`batch_hvg_genes`	Integer vector. These are the highly variable genes, identified by a batch-aware method. Please refer to `find_hvg_batch_aware_sc()` for more details. These genes have to be 0-indexed!
`cca_params`	A list, please see `params_sc_seurat_cca()`. The list has the following parameters: • num_cc - Integer. Number of canonical correlation dimensions. The effective rank used is `max(num_cc, dims)`. • dims - Integer. Dimensions used for the anchor kNN queries and size of the returned embedding. • k_anchor - Integer. Neighbourhood size for the anchor search. • k_filter - Integer. Neighbourhood size for the gene-space filter.

- k_score - Integer. Neighbourhood size for the anchor scoring.
- k_weight - Integer. Neighbourhood size for the kernel weights.
- n_top_features - Integer. Top-loading genes for the gene-space filter.
- l2_norm - Logical. L2-normalise the CCA embedding per cell.
- sd - Numeric. Bandwidth divisor of the Gaussian kernel.
- knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.
- pca - List of PCA parameters, see `params_sc_pca()` for available parameters and their defaults.

`use_precomputed_pca`
: Boolean. Should the PCA in the object be used if found. If you decide to do this, make sure that you have run the PCA on the batch-aware HVG ideally. Note that CCA still needs the PCA loadings for the gene-space filter, so this saves less work than it does for fastMNN.

`seed`
: Integer. Random seed.

`.verbose`
: Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

The object with the added `"cca"` embedding.

References

Stuart, et al., Cell, 2019

Examples

```
# CCA anchor integration over batch aware highly variable genes
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 600L, n_genes = 50L, n_batches = 3L
  )
)
hvg <- find_hvg_batch_aware_sc(
  sc, hvg_no = 20L, batch_column = "batch_index", .verbose = FALSE
)
sc <- seurat_cca_sc(
  sc,
  batch_column = "batch_index",
  batch_hvg_genes = hvg$hvg_gene_idx,
  cca_params = params_sc_seurat_cca(num_cc = 10L, dims = 10L),
  .verbose = FALSE
)
dim(get_embedding(sc, "cca"))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`seurat_rpca_sc` *Run Seurat rPCA integration*

Description

This function implements the reciprocal PCA (rPCA) anchor integration from Stuart, et al. It runs the same anchor pipeline as `seurat_cca_sc()` but builds a cheaper per-pair anchor space: each batch keeps its own PCA basis and the other batch's HVG expression is projected into it. Cross-batch mutual nearest neighbours are then found in these projected bases.

rPCA is faster than CCA and corrects less aggressively, which makes it the safer choice when batches share most of their cell types. As in Seurat, no gene-space anchor filter is applied, that step is CCA-only.

Usage

```
seurat_rpca_sc(
  object,
  batch_column,
  batch_hvg_genes,
  rpca_params = params_sc_seurat_rpca(),
  use_precomputed_pca = FALSE,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`batch_column`	String. The column with the batch information in the obs data of the class.
`batch_hvg_genes`	Integer vector. These are the highly variable genes, identified by a batch-aware method. Please refer to `find_hvg_batch_aware_sc()` for more details. These genes have to be 0-indexed!
`rpca_params`	A list, please see `params_sc_seurat_rpca()`. The list has the following parameters: • dims - Integer. Dimensions used for the per-batch projections, the anchor kNN queries and the size of the returned embedding. • k_anchor - Integer. Neighbourhood size for the anchor search. • k_score - Integer. Neighbourhood size for the anchor scoring. • k_weight - Integer. Neighbourhood size for the kernel weights. • l2_norm - Logical. L2-normalise the projected embeddings per cell. • sd - Numeric. Bandwidth divisor of the Gaussian kernel.

- knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.
- pca - List of PCA parameters, see `params_sc_pca()` for available parameters and their defaults.

`use_precomputed_pca`
: Boolean. Should the PCA in the object be used if found. This only applies to the union PCA that gets corrected, the per-batch PCAs are always recomputed because rPCA needs them.

`seed`
: Integer. Random seed.

`.verbose`
: Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

The object with the added `"rpca"` embedding.

References

Stuart, et al., Cell, 2019

Examples

```
# reciprocal PCA anchor integration, the cheaper sibling of CCA
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 600L, n_genes = 50L, n_batches = 3L
  )
)
hvg <- find_hvg_batch_aware_sc(
  sc, hvg_no = 20L, batch_column = "batch_index", .verbose = FALSE
)
sc <- seurat_rpca_sc(
  sc,
  batch_column = "batch_index",
  batch_hvg_genes = hvg$hvg_gene_idx,
  rpca_params = params_sc_seurat_rpca(dims = 10L),
  .verbose = FALSE
)
dim(get_embedding(sc, "rpca"))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`SimilarityNetworkFusion`
Similarity network fusion

Description

This class is designed to generate and store the adjacency matrices for subsequent usage in similarity network fusion. There are methods to add different types of adjacency matrices (pending the type of data, it can be continuous, categorical and/or mixed). Subsequently, there are methods to generate the final network.

Usage

```
SimilarityNetworkFusion(
  data = NULL,
  data_name = NULL,
  snf_params = params_snf()
)
```

Arguments

`data`	Optional data to already transform into adjacency data. Any data supplied will be assumed to be samples x features. The provided data taype can be a data.table (for categorical and/or mixed types) or a matrix (for continous types). If you provide a data.table, the function will assume the first column are the sample identifiers. Please ensure that setting.
`data_name`	Optional string. Name of the data modality.
`snf_params`	List. The SNF parameters, see `params_snf()`. The list contains the following elements: • k - Integer. Number of neighbours to consider. • t - Integer. Number of iterations for the SNF algorithm. • mu - Float. Normalisation factor for the Gaussian kernel width. • alpha - Float. Normalisation parameter controlling the fusion strength. • normalise - Boolean. Shall continuous values be Z-scored. • distance_metric - String. One of `c("euclidean", "manhattan", "canberra", "cosine")`. Which distance metric to use for the continuous calculations. In case of pure categorical, Hamming will be used, for mixed data types Gower distance is used. The parameters will be internally stored for subsequent usage in other functions.

Value

Returns the `SimilarityNetworkFusion` class for further operations.

Properties

adj_matrices A list containing the processed adjacency matrices.

snf_adj Matrix. The final adjacency of the fused network.

params A (nested) list that will store all the parameters of the applied function.

final_results data.table. Contains final results.

Examples

```
# continuous modality turned into an affinity matrix
set.seed(42)
continuous <- matrix(rnorm(120), nrow = 12, ncol = 10)
rownames(continuous) <- sprintf("sample_%02i", 1:12)
colnames(continuous) <- sprintf("feature_%i", 1:10)
object <- SimilarityNetworkFusion(
  data = continuous,
  data_name = "continuous",
  snf_params = params_snf(k = 3L)
)
dim(get_snf_adjcacency_mat(object, "continuous"))
```

---

simplify_hypergeom_res

Simplify gene set results via ontologies

Description

This function provides an interface to simplify overenrichment results based on ontological information of the pathway origin (typical use case is to simplify gene ontology results). To do so, the function will calculate the Wang similarity and keep within a set of highly similar terms the one with the lowest fdr. Should there be terms with the same fdr, the function will keep the most specific term within the ontology.

Usage

```
simplify_hypergeom_res(res, parent_child_dt, weights, min_sim = 0.7)
```

Arguments

`res`	data.table. The enrichment results. Needs to have the columns `c("gene_set_name", "fdr")`.
`parent_child_dt`	data.table. The data.table with column parent and child. You also need to have a type column for the Wang similarity to provide the weights for the relationships.
`weights`	Named numeric. The relationship of type to weight for this specific edge. For example `c("part_of" = 0.8, "is_a" = 0.6)`.
`min_sim`	Float between 0 and 1. The minimum similarity that the terms need to have.

Value

data.table with enrichment results.

Examples

```
# collapse redundant terms of a toy ontology by Wang similarity
onto <- data.table::data.table(
  parent = c("a", "b", "b", "c"),
  child = c("b", "c", "d", "e"),
  type = c("is_a", "is_a", "part_of", "is_a")
)
res <- data.table::data.table(
  gene_set_name = c("b", "c", "d", "e"),
  fdr = c(0.001, 0.001, 0.4, 0.02)
)
simplify_hypergeom_res(
  res,
  onto,
  weights = c("is_a" = 0.8, "part_of" = 0.6),
  min_sim = 0.6
)
```

---

simulate_dropouts *Simulate sequencing-depth dropouts on synthetic bulk data*

Description

This function induces Splatter-style sequencing-depth sparsity on the data. Per sample a size factor `s_j ~ LogNormal(0, capture_efficiency_sigma)` is drawn, giving a target library size of `target_library_size * s_j`. Each gene is then binomially thinned to approach that target, so dropout falls out of the library size rather than an explicit per-gene dropout curve. Retention probability is capped at 1, meaning samples already below their target are left alone rather than upsampled.

Usage

```
simulate_dropouts(object, sparsity_params = params_bulk_sparsity())
```

Arguments

`object`	The `synthetic_bulk_data` class.
`sparsity_params`	List. The sparsification parameters, see `params_bulk_sparsity()`.

Value

`synthetic_bulk_data` with added sparse data.

References

Zappia, et al., Genome Biol, 2017

Examples

```
# thin the counts down to a shallower library size
syn <- synthetic_bulk_cor_matrix()
syn <- simulate_dropouts(syn, params_bulk_sparsity())
mean(syn$counts == 0)
mean(syn$sparse_counts == 0)
```

SingleCellCountData *Single cell count data handler*

Description

[**Experimental**] A class for handling single cell count data stored on disk in two complementary binary representations: a CSR-like layout (`f_path_cells`) for fast cell-wise access and a CSC-like layout (`f_path_genes`) for fast gene-wise access. Both raw counts and log-normalised counts are stored side by side. Provides methods for ingesting data from R, `h5ad`, `mtx` and 10x CellRanger h5 sources (including multi-file workflows), converting between layouts, retrieving slices of the matrix, merging existing binary objects and writing CellSweep-denoised counts.

Arguments

`f_path_cells`	(`character`) Path to the `.bin` file for the cell-based (CSR-like) representation.
`f_path_genes`	(`character`) Path to the `.bin` file for the gene-based (CSC-like) representation.
`n_cells`	(`integer`) Number of cells represented in the data.
`n_genes`	(`integer`) Number of genes represented in the data.

Value

A new instance of the `SingleCellCountData` class.

Methods

Method `new`: Create a new instance of the class

Arguments:

`f_path_cells` (`character`)
: Path to the `.bin` file for the cell-based representation.

`f_path_genes` (`character`)
: Path to the `.bin` file for the gene-based representation.

returns: A new `SingleCellCountData` instance with `n_cells` and `n_genes` initialised to zero.

Method `get_shape`**:** Get the shape of the matrix

returns: An integer vector `c(n_cells, n_genes)`.

Method `set_from_file`**:** Populate `n_cells` and `n_genes` from the cells binary file

description: Reads the header of the file at `f_path_cells` and updates the `n_cells` and `n_genes` fields accordingly. Useful when reconnecting to an existing object on disk.

returns: Invisible `NULL`.

Method `r_data_to_file`**:** Write a CSR matrix from R to the cells binary file

Arguments:

`r_data` (`list`)
: A named list convertible into `CompressedSparseData2`. Must contain `"indptr"`, `"indices"` (0-indexed), `"data"`, `"nrow"`, `"ncol"` and `"cs_type"` (`"csr"` or `"csc"`).

`qc_params` (`list`)
: Quality control parameters parseable into `MinCellQuality`.

`verbose` (`logical`)
: Controls verbosity of the function.

description: Ingest a sparse matrix passed in from R, apply per-cell QC, and write the result to `f_path_cells`.

returns: A list with `cell_indices` and `gene_indices` (0-indexed, surviving QC), `lib_size` and `nnz`.

Method `h5ad_to_file`**:** Write an h5ad file to the cells binary file

Arguments:

`cs_type` (`character`)
: Storage layout of the h5ad data. One of `"CSC"` or `"CSR"`.

`h5_path` (`character`)
: Path to the h5ad file.

`no_cells` (`integer`)
: Number of cells in the h5ad file.

`no_genes` (`integer`)
: Number of genes in the h5ad file.

`qc_params` (`list`)
: Quality control parameters parseable into `MinCellQuality`.

`slot` (`character`)
: Where to find the raw counts. One of `"X"`, `"raw.X"` (or `"raw"`, for CellXGene data) or `"layers.counts"`. Unmatched values fall back to `"X"`.

`verbose` (`logical`)
: Controls verbosity of the function.

returns: A list with `cell_indices` and `gene_indices` (0-indexed, surviving QC), `lib_size` and `nnz`.

Method `norm_h5ad_to_file`**:** Write an h5ad file with normalised counts to the cells binary file

Arguments:

`cs_type` (`character`)
: Storage layout of the h5 data. One of `"CSC"` or `"CSR"`.

`h5_path` (`character`)
: Path to the h5 file.

`no_cells` (`integer`)
: Number of cells in the h5 file.

`no_genes` (`integer`)
: Number of genes in the h5 file.

`obs_lib_size_col` (`character`)
: Name of the `obs` column containing total counts per cell (e.g. `"nCount_RNA"`).

`target_size` (`numeric`)
: Target size used in the original normalisation (e.g. `1e4`).

`qc_params` (`list`)
: Quality control parameters parseable into `MinCellQuality`.

`verbose` (`logical`)
: Controls verbosity of the function.

description: For data sets where only normalised counts are available in `X`. Reads library sizes from a specified `obs` column to reconstruct raw counts before writing.

returns: A list with `cell_indices` and `gene_indices` (0-indexed, surviving QC), `lib_size` and `nnz`.

Method `h5ad_to_file_streaming`**:** Write an h5ad file to disk using streaming

Arguments:

`cs_type` (`character`)
: Storage layout of the h5 data. One of `"CSC"` or `"CSR"`.

`h5_path` (`character`)
: Path to the h5 file.

`no_cells` (`integer`)
: Number of cells in the h5 file.

`no_genes` (`integer`)
: Number of genes in the h5 file.

`qc_params` (`list`)
: Quality control parameters parseable into `MinCellQuality`.

`slot` (`character`)
: Where to find the raw counts. One of `"X"`, `"raw.X"` (or `"raw"`, for CellXGene data) or `"layers.counts"`. Unmatched values fall back to `"X"`.

`verbose` (`logical`)
: Controls verbosity of the function.

description: Slower but lighter on memory than `h5ad_to_file`; streams the input where possible.

returns: A list with `cell_indices` and `gene_indices` (0-indexed, surviving QC), `lib_size` and `nnz`.

Method `multi_h5ad_to_file`**:** Load multiple h5ad files into a single binary

Arguments:

`file_tasks` (`list`)
: A list of lists, each produced by the R prescan function. Each inner list must contain `exp_id`, `h5_path`, `cs_type`, `no_cells`, `no_genes` and `gene_local_to_universe` (0-indexed integer vector, `NA` for unmapped genes), plus an optional `raw_slot`.

`universe_size` (`integer`)
: Total number of genes in the universe.

`qc_params` (`list`)
: Quality control parameters (`min_unique_genes`, `min_lib_size`, `min_cells`, `target_size`).

`verbose` (`logical`)
: Controls verbosity.

returns: A list with `global_gene_indices`, `total_cells`, `total_genes` and `per_file` (a list of lists with `exp_id`, `cell_indices` (file-local, 0-indexed), `lib_size`, `nnz`).

Method `mtx_to_file`**:** Write an mtx file to the cells binary file

Arguments:

`mtx_path` (`character`)
: Path to the mtx file.

`qc_params` (`list`)
: Quality control parameters parseable into `MinCellQuality`.

`cells_as_rows` (`logical`)
: `TRUE` if cells are rows in the mtx file, `FALSE` if cells are columns.

`verbose` (`logical`)
: Controls verbosity of the function.

returns: A list with `cell_indices` and `gene_indices` (0-indexed, surviving QC), `lib_size` and `nnz`.

Method `mtx_to_file_streaming`**:** Write an mtx file to the cells binary file using streaming

Arguments:

`mtx_path` (`character`)
: Path to the mtx file.

`qc_params` (`list`)
: Quality control parameters parseable into `MinCellQuality`.

`cells_as_rows` (`logical`)
: `TRUE` if cells are rows in the mtx file, `FALSE` if cells are columns.

`verbose` (`logical`)
: Controls verbosity of the function.

returns: A list with `cell_indices` and `gene_indices` (0-indexed, surviving QC), `lib_size` and `nnz`.

Method `multi_mtx_to_file`**:** Load multiple mtx files into a single binary

Arguments:

`file_tasks` (`list`)
: A list of lists, each containing `exp_id`, `mtx_path`, `cells_as_rows` and `gene_local_to_universe` (integer vector, `NA` for unmapped genes).

`universe_size` (`integer`)
: Number of genes in the intersection universe.

`qc_params` (`list`)
: Quality control parameters parseable into `MinCellQuality`.

`verbose` (`logical`)
: Controls verbosity.

returns: A list with `global_gene_indices`, `total_cells`, `total_genes` and `per_file` (a list of lists with `exp_id`, `cell_indices` (file-local, 0-indexed), `lib_size`, `nnz`).

Method `tenx_h5_to_file_streaming`**:** Write a 10x CellRanger h5 file to the cells binary file using streaming

Arguments:

`h5_path` (`character`)
: Path to the 10x h5 file.

`version` (`character`)
: One of `"auto"`, `"v2"` or `"v3"`. `"auto"`, and any unmatched value, detects the layout from the file.

`no_cells` (`integer`)
: Number of cells (columns) in the file.

`no_genes` (`integer`)
: Number of features (rows), including all modalities.

`qc_params` (`list`)
: Quality control parameters parseable into `MinCellQuality`.

`feature_type` (`character` or `NULL`)
: Target modality for v3. Defaults to `"Gene Expression"` when `NULL`.

`verbose` (`logical`)
: Controls verbosity of the function.

description: Ingests the gene-expression modality from a CellRanger v2/v3 h5 file. Other modalities (e.g. Antibody Capture) are filtered out via `feature_type`.

returns: A list with `cell_indices` and `gene_indices` (0-indexed, surviving QC), `lib_size` and `nnz`.

Method `multi_tenx_h5_to_file`**:** Load multiple 10x CellRanger h5 files into a single binary

Arguments:

`file_tasks` (`list`)
: A list of lists, each produced by the R prescan function. Each inner list must contain `exp_id`, `h5_path`, `version` (`"v2"` or `"v3"`), `no_cells`, `no_genes`, `gene_local_to_universe` (integer vector, `NA` for unmapped / non-gene features) and `feature_type` (optional string, defaults to `"Gene Expression"`).

`universe_size` (`integer`)
: Total number of genes in the universe.

`qc_params` (`list`)
: Quality control parameters (`min_unique_genes`, `min_lib_size`, `min_cells`, `target_size`).

`verbose` (`logical`)
: Controls verbosity.

returns: A list with `global_gene_indices`, `total_cells`, `total_genes` and `per_file` (a list of lists with `exp_id`, `cell_indices` (file-local, 0-indexed), `lib_size`, `nnz`).

Method `return_full_mat`**:** Return the full count matrix

Arguments:

`assay` (`character`)
: One of `"raw"` or `"norm"`. Selects whether raw counts or log-normalised counts are returned.

`cell_based` (`logical`)
: If `TRUE`, the data is returned in CSR layout (cells as rows). If `FALSE`, the data is returned in CSC layout (genes as columns).

`verbose` (`logical`)
: Controls verbosity of the function.

returns: A list with `indptr`, `indices` (0-indexed), `data`, `no_cells` and `no_genes`, parseable into a sparse matrix in R. `data` is integer for `"raw"` and double for `"norm"`.

Method `get_cells_by_indices`**:** Return cells by index positions

Arguments:

`indices` (`integer`)
: The cell indices to return (1-indexed).

`assay` (`character`)
: One of `"raw"` or `"norm"`.

description: Leverages the CSR-stored data for fast cell retrieval.

returns: A list with `indptr`, `indices` (0-indexed gene positions), `data`, `no_cells` (number of returned cells) and `no_genes`, parseable into a CSR matrix in R.

Method `generate_gene_based_data`**:** Generate gene-based data from the cells binary file

Arguments:

`max_mem_gb` (`numeric` or `NULL`)
: Memory for the conversion buffers in GB, at 10 bytes per non-zero. `NULL` converts in a single phase.

`verbose` (`logical`)
: Controls verbosity of the function.

description: Reads the `.bin` file at `f_path_cells` and writes a gene-friendly (CSC) representation to `f_path_genes`. A parallel counting-sort transpose: one pass counts the non-zeros per gene, then genes are converted in phases that fit `max_mem_gb`. Each phase re-reads the cell file, so a tighter budget trades time for memory.

returns: Invisible `NULL`.

Method `archive`**:** Archive the cell-based binary for cold storage

Arguments:

`f_path_archive` (`character`)
: Path of the archive to write.

`level` (`integer`)
: zstd compression level, 1 to 22.

`verbose` (`logical`)
: Controls verbosity of the function.

description: Writes a zstd-compressed archive of `f_path_cells`. The gene-based file is not archived; `restore_archive()` rebuilds it. Normalised values are only stored for cells where they cannot be recomputed from the raw counts.

returns: A list with `n_cells`, `nnz`, `n_norm_stored` and `archive_bytes`.

Method `restore_archive`**:** Restore both binaries from an archive

Arguments:

`f_path_archive` (`character`)
: Path to the archive.

`max_mem_gb` (`numeric` or `NULL`)
: Memory for the gene file conversion buffers in GB. `NULL` converts in a single phase.

`verbose` (`logical`)
: Controls verbosity of the function.

description: Rebuilds `f_path_cells` from the archive, then generates `f_path_genes` from it.

returns: Invisible `NULL`.

Method `get_genes_by_indices`**:** Return genes by index positions

Arguments:

`indices` (`integer`)
: The gene indices to return (1-indexed).

`assay` (`character`)
: One of `"raw"` or `"norm"`.

description: Leverages the CSC-stored data for fast gene retrieval.

returns: A list with `indptr`, `indices` (0-indexed cell positions), `data`, `no_cells` and `no_genes` (number of returned genes), parseable into a CSC matrix in R.

Method `get_nnz_genes`**:** Get the number of cells expressing each gene

Arguments:

`gene_indices` (`integer` or `NULL`)
: Optional 1-indexed gene indices. If `NULL`, results are returned for all genes.

returns: An integer vector of NNZ counts for the requested genes.

Method `merge_sc_files`**:** Merge multiple existing bin files into the cells binary file

Arguments:

`merge_tasks` (`list`)
: A list of lists. Each inner list must contain `exp_id`, `bin_cells_path`, `cells_to_keep` (0-indexed integer vector) and `gene_local_to_universe` (0-indexed integer vector, `NA` for genes absent from the universe).

`universe_size` (`integer`)
: Number of genes in the intersection universe.

`renormalise` (`logical`)
: If `TRUE`, recompute `data_norm` against `target_size` using each cell's surviving raw counts. If `FALSE`, pass `data_norm` through untouched; the caller must guarantee all inputs were normalised against the same `target_size`.

`target_size` (`numeric`)
: Target library size for renormalisation. Ignored when `renormalise = FALSE`.

`verbose` (`logical`)
: Controls verbosity.

returns: A list with `total_cells`, `total_genes` and `per_file` (a list of lists with `exp_id`, `lib_size`, `nnz`).

Method `cellsweep`**:** Run CellSweep and write the denoised barcodes into the cells binary

One independent EM fit per sample, since the ambient profile is a property of a single emulsion. Only the real barcodes are written: the empty droplets exist to train the ambient profile, and barcodes that are neither empty nor annotated are not part of the model.

Writes the cell-based file only; regenerate the gene-based companion afterwards, as for a merge.

Arguments:

`f_path_source` (`character`)
: Path to the raw `counts_cells.bin`, which must still contain the empty droplets.

`samples` (`list`)
: A list of lists. Each inner list must contain `sample_id`, `real_cells` and `empty_cells` (0-indexed integer vectors of store indices), `celltype_idx` (0-indexed integer vector, one entry per `real_cells` entry) and `n_celltypes`.

`cellsweep_params` (`list`)
: The CellSweep model parameters. Missing entries fall back to the reference implementation's defaults.

`target_size` (`numeric`)
: Library size the normalised layer is scaled to.

`verbose` (`integer`)
: `0` silent, `1` per-sample progress, `2` per-EM-iteration.

returns: A list with `cell_order` (0-indexed source indices in output order), `lib_size`, `nnz` and `fits` (one list per sample with `sample_id`, `alpha`, `z_hat` (1-indexed), `beta`, `ambient`, `celltype_profiles`, `n_celltypes`, `log_likelihood`, `n_iter` and `converged`).

`SingleCells` *bixverse SingleCells class*

Description

This is the `bixverse`-based SingleCells class. Under the hood it uses a DuckDB for obs and vars storing, and a Rust-based binarised file format to store the raw and normalised counts. In both cases, the idea is not to hold any data that is not needed at a given point of time in memory, but leverage speedy on-disk computations and streaming engines powered by Rust and DuckDB to run the analysis. This version is specifically designed for single cell RNAseq. If you want to use multi-modal data, please refer to `SingleCellsMultiModal()` - this class can store multiple layers of 'omics.

Usage

```
SingleCells(dir_data)
```

Arguments

`dir_data`	String. This is the directory in which the experimental files will be

Value

Returns the `SingleCells` class for further operations.

Properties

db__connection This contains an R6 class with DuckDB pointers and wrappers to interact with the table-like data for this experiment.

count__connection This contains an R6-like environment that points to Rust functions that can work on the counts more specifically.

dir__data Path to the directory in which the data will be saved on disk.

sc__cache Class with cached data. Contains less memory-heavy objects such as embeddings, kNN information or sNN graphs.

sc__map Class containing various mapping information such as HVG indices, cells to keep, etc.

dims Dimensions of the original data.

Examples

```
# demo_single_cells() wraps the construction and the ingestion
sc <- demo_single_cells(prepped = FALSE)
sc

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`SingleCellsMultiModal`
bixverse SingleCells (multi modal) class

Description

This is the `bixverse`-based SingleCells class for multiple modalities. Under the hood it uses a DuckDB for obs and vars storing, and a Rust-based binarised file format to store the raw and normalised counts for single cell RNAseq. In both cases, the idea is not to hold any data that is not needed at a given point of time in memory, but leverage speedy on-disk computations and streaming engines powered by Rust and DuckDB to run the analysis.

Usage

```
SingleCellsMultiModal(dir_data)
```

Arguments

`dir_data`	String. This is the directory in which the experimental files will be

Value

Returns the `SingleCellsMultiModal` class for further operations.

Properties

db__connection This contains an R6 class with DuckDB pointers and wrappers to interact with the table-like data for this experiment.

count__connection This contains an R6-like environment that points to Rust functions that can work on the RNAseq counts more specifically.

adt__counts ...

peak__connection Future feature: to store ATAC Seq counts in the future.

dir__data Path to the directory in which the data will be saved on disk.

sc__cache Class with cached data. Contains less memory-heavy objects such as embeddings, kNN information or sNN graphs for the single cell RNAseq.

adt__cache Class with cached data. Contains less memory-heavy objects such as embeddings, kNN information or sNN graphs for the single cell Antibody-Derived Tags.

atac__cache Class with cached data. Contains less memory-heavy objects such as embeddings, kNN information or sNN graphs for the single cell chromatin accessability.

sc__map Class containing various mapping information such as HVG indices, cells to keep, etc.

other__data List that contains additional data and results, such as for example the WNN graph.

dims Dimensions of the original data.

Examples

```
# an empty multi-modal handle with the RNA modality ingested into it
rna <- generate_single_cell_test_data()
dir <- tempfile("bixverse_mm")
dir.create(dir)
object <- load_r_data(
  SingleCellsMultiModal(dir_data = dir),
  counts = rna$counts,
  obs = rna$obs,
  var = rna$var,
  sc_qc_param = params_sc_min_quality(min_unique_genes = 5L),
  .verbose = FALSE
)
object

unlink(dir, recursive = TRUE, force = TRUE)
```

SingleCellsSubset *bixverse single cell subset class*

Description

Subset view onto a SingleCells() object, restricted to cells belonging to a single level of a grouping variable. The Rust count connection is shared with the parent (no data copy). obs_table and var_table are held in memory; sc_map is rebuilt to point only at the subset cells but stays in the original index space so Rust calls remain valid without further translation.

Usage

```
SingleCellsSubset(sc_object, grouping_column, group)
```

Arguments

sc_object	A SingleCells() object to subset.
grouping_column	String. Column in the obs table that defines the grouping.
group	String. Level of grouping_column to retain.

Value

A SingleCellsSubset object.

Properties

count_connection Shared Rust pointer to the on-disk counts.

dir_data Directory holding the binary count files.

obs_table Subset obs (rows for the chosen group only). `cell_idx` keeps the original 1-indexed position in the parent.

var_table Variable/feature table (unchanged from parent).

grouping_column Column in obs used to define the subset.

group Value of `grouping_column` represented by this subset.

sc_cache Fresh `ScCache` for subset-specific PCA, kNN, sNN, embeddings.

sc_map `ScMap` restricted to the subset cells. `cell_mapping` stays 1-indexed and `cells_to_keep_idx` stays 0-indexed, both in the original parent index space.

subset_to_original Integer vector. 1-indexed original cell positions, in subset row order. `subset_to_original[i]` is the parent position of subset row `i`.

dims `c(n_cells_subset, n_genes)`.

Examples

```
# a view onto one cell type, sharing the parent's counts on disk
sc <- demo_single_cells(prepped = FALSE)
subset_obj <- SingleCellsSubset(
  sc_object = sc,
  grouping_column = "cell_grp",
  group = "cell_type_1"
)
dim(subset_obj)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`stabilised_nmf_bulk` *Run stabilised (multi-restart) NMF on a BulkCoExp*

Description

Runs `n_runs` HALS-NMF fits with random initialisations and returns the run with the lowest final reconstruction loss as the primary result, along with the per-run losses and convergence flags. Useful when the objective is known to have multiple local minima. Uses `f64` precision.

Usage

```
stabilised_nmf_bulk(
  object,
  k,
  n_runs = 30L,
  preprocessing = c("none", "sd", "sqrt_sd"),
```

```
  nmf_hals_params = params_nmf_hals(),
  membership_params = params_module_membership(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	The class, see `BulkCoExp()`.
`k`	Integer. Number of latent factors (modules) per run.
`n_runs`	Integer. Number of random restarts.
`preprocessing`	String. One of `c("none", "sd", "sqrt_sd")`.
`nmf_hals_params`	List. Output of `params_nmf_hals()`. Note that `nmf_init` is ignored for the stabilised variant, which always uses random initialisation seeded by `seed + i`.
`membership_params`	List. Controls how the gene loadings are turned into module membership, see `params_module_membership()`.
`seed`	Integer. Base random seed.
`.verbose`	Boolean or integer `0L`/`1L`/`2L`. Controls verbosity.

Value

The class with `final_results` populated from the best run (lowest final loss) plus stability diagnostics (`losses`, `converged`, `best_idx`, `w_all_runs`, `h_per_run`).

Examples

```
# ten random restarts, best run kept
syn <- generate_gene_module_data(n_samples = 24L, n_genes = 60L)
# NMF needs a non-negative matrix
mat <- syn$data - min(syn$data)
obj <- BulkCoExp(mat, syn$meta_data)
obj <- preprocess_bulk_coexp(
  obj, hvg = NULL, scaling = FALSE, .verbose = FALSE
)
obj <- stabilised_nmf_bulk(obj, k = 4L, n_runs = 10L, .verbose = FALSE)
get_nmf_stability(obj)$losses
```

`stabilised_nmf_sc` *Run stabilised (multi-run) NMF on single cell or meta cell data*

Description

Runs `n_runs` HALS NMF with random initialisations seeded by `seed + i`. The `nmf_init` field in `nmf_hals_params` is ignored; random init is always used.

Usage

```
stabilised_nmf_sc(
  object,
  k,
  cell_ids = NULL,
  gene_ids = NULL,
  preprocessing = "none",
  use_second_layer = TRUE,
  nmf_hals_params = params_nmf_hals(),
  n_runs = 30L,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` or `MetaCells` class.
`k`	Integer. Number of latent factors to return.
`cell_ids`	Optional character. Cell ids (or meta cell ids) to restrict the NMF to. If `NULL`, uses `get_cells_to_keep()` for `SingleCells` and all meta cells for `MetaCells`.
`gene_ids`	Optional character. Gene ids to restrict the NMF to. If `NULL`, uses `get_hvg()` on the object.
`preprocessing`	String. One of `c("none", "sd", "sqrt_sd")`.
`use_second_layer`	Boolean. If `TRUE`, runs NMF on the normalised counts (recommended); if `FALSE`, on the raw counts.
`nmf_hals_params`	List, see `params_nmf_hals()`.
`n_runs`	Integer. Number of random restarts.
`seed`	Integer. Random seed for initialisation.
`.verbose`	Boolean or integer. Verbosity.

Value

A `StabilisedNmfResult` object.

Examples

```
# five random restarts, the best one reported
sc <- demo_single_cells()
res <- stabilised_nmf_sc(sc, k = 3L, n_runs = 5L, .verbose = FALSE)
res

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`step_bbknn_sc` *Pipeline step: BBKNN batch correction*

Description

Wraps `bbknn_sc()` as an `ScStep`.

Usage

```
step_bbknn_sc(
  batch_column,
  no_neighbours_to_keep = 5L,
  embd_to_use = "pca",
  no_embd_to_use = NULL,
  bbknn_params = params_sc_bbknn(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`batch_column`	String. The column with the batch information in the obs data of the class.
`no_neighbours_to_keep`	Integer. Maximum number of neighbours to keep from the BBKNN algorithm. Due to generating neighbours for each batch, there might be a large number of generated neighbours. This will only keep the top `no_neighbours_to_keep` neighbours. Defaults to `5L`.
`embd_to_use`	String. The embedding to use. Atm, the only option is `"pca"`.
`no_embd_to_use`	Optional integer. Number of embedding dimensions to use. If `NULL` all will be used.
`bbknn_params`	A list, please see `params_sc_bbknn()`. The list has the following parameters: • neighbours_within_batch - Integer. Number of neighbours to consider per batch. • set_op_mix_ratio - Numeric. Mixing ratio between union (1.0) and intersection (0.0). • local_connectivity - Numeric. UMAP connectivity computation parameter, how many nearest neighbours of each cell are assumed to be fully connected. • trim - Optional integer. Trim the neighbours of each cell to these many top connectivities. May help with population independence and improve the tidiness of clustering. If `NULL`, it defaults to `10 * neighbours_within_batch`.

- knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.

`seed` Integer. Random seed.

`.verbose` Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

An `ScStep`.

Examples

```
# BBKNN replaces the graph outright, so nothing follows it
step_pca_sc(no_pcs = 10L) %>>%
  step_bbknn_sc(batch_column = "batch_index")
```

---

step_clusters_sc *Pipeline step: graph-based clustering*

Description

Wraps `find_clusters_sc()` as an `ScStep`.

Usage

```
step_clusters_sc(
  cluster_algorithm = c("leiden", "louvain"),
  res = 1,
  name = "leiden_clustering",
  modality = c("rna", "adt", "wnn"),
  seed = 42L
)
```

Arguments

`cluster_algorithm` String. One of `c("leiden", "louvain")`.

`res` Numeric. The modularity resolution parameter, see `leiden_clustering()`.

`name` String. The name to add to the obs table in the DuckDB.

`modality` String. On which modality to run the UMAP. One of `c("rna", "adt", "wnn")`. The two latter options are only available for multi-modal versions with the added data.

`seed` Integer. For reproducibility.

Value

An `ScStep`.

Examples

```
# Leiden over the graph the neighbours step wrote
step_neighbours_sc() %>>% step_clusters_sc(res = 0.5)
```

`step_fast_mnn_sc` *Pipeline step: fastMNN batch correction*

Description

Wraps `fast_mnn_sc()` as an `ScStep`.

Usage

```
step_fast_mnn_sc(
  batch_column,
  batch_hvg_genes,
  fastmnn_params = params_sc_fastmnn(),
  use_precomputed_pca = FALSE,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`batch_column` String. The column with the batch information in the obs data of the class.

`batch_hvg_genes`
Integer vector. These are the highly variable genes, identified by a batch-aware method. Please refer to `find_hvg_batch_aware_sc()` for more details. These genes have to be 0-indexed!

`fastmnn_params`
A list, please see `params_sc_fastmnn()`. The list has the following parameters:

- sigma - Numeric. Bandwidth of the Gaussian smoothing kernel (as proportion of space radius).
- cos_norm - Logical. Apply cosine normalisation before computing distances.
- var_adj - Logical. Apply variance adjustment to avoid kissing effects.
- no_pcs - Integer. Number of PCs to use for MNN calculations.
- knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.
- pca - List of PCA parameters, see `params_sc_pca()` for available parameters and their defaults.

`use_precomputed_pca`
Boolean. Should the PCA in the object be used if found. If you decide to do this, make sure that you have run the PCA on the batch-aware HVG ideally.

seed	Integer. Random seed.
.verbose	Boolean or integer. Controls verbosity and returns run times. FALSE -> quiet, TRUE or 1L -> normal verbosity, 2L -> detailed verbosity.

Value

An ScStep.

Examples

```
# fastMNN wants the batch aware genes handed to it up front
step_fast_mnn_sc(
  batch_column = "batch_index",
  batch_hvg_genes = 0:29L
) %>>%
  step_neighbours_sc(embd_to_use = "mnn")
```

---

step_harmony_sc *Pipeline step: Harmony batch correction*

Description

Wraps harmony_sc() as an ScStep.

Usage

```
step_harmony_sc(
  batch_column,
  additional_batch_columns = NULL,
  modality = c("rna", "adt"),
  harmony_params = params_sc_harmony(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

batch_column	String. Column name in the object containing the primary batch labels.
additional_batch_columns	Optional character vector. Additional batch columns to regress out. If NULL, only the primary batch column is used.
modality	String. One of c("rna", "adt"). You can only use "adt" on SingleCellsMultiModal class.
harmony_params	List. Output of params_sc_harmony().
seed	Integer. For reproducibility.
.verbose	Boolean or integer. Controls verbosity and returns run times. FALSE -> quiet, TRUE or 1L -> normal verbosity, 2L -> detailed verbosity.

Value

An ScStep.

Examples

```
# correct the PCA, then build the graph on the corrected embedding
step_harmony_sc(batch_column = "batch_index") %>>%
  step_neighbours_sc(embd_to_use = "harmony")
```

---

step_harmony_v2_sc *Pipeline step: Harmony v2 batch correction*

Description

Wraps harmony_v2_sc() as an ScStep.

Usage

```
step_harmony_v2_sc(
  batch_column,
  additional_batch_columns = NULL,
  modality = c("rna", "adt"),
  harmony_params = params_sc_harmony_v2(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

batch_column	String. Column name in the object containing the primary batch labels.
additional_batch_columns	Optional character vector. Additional batch columns to regress out. If NULL, only the primary batch column is used.
modality	String. One of c("rna", "adt"). You can only use "adt" on SingleCellsMultiModal class.
harmony_params	List. Output of params_sc_harmony_v2().
seed	Integer. For reproducibility.
.verbose	Boolean or integer. Controls verbosity and returns run times. FALSE -> quiet, TRUE or 1L -> normal verbosity, 2L -> detailed verbosity.

Value

An ScStep.

Examples

```
# the v2 implementation writes its own embedding name
step_harmony_v2_sc(batch_column = "batch_index") %>>%
  step_neighbours_sc(embd_to_use = "harmony_v2")
```

---

step_hvg_sc *Pipeline step: identify highly variable genes*

Description

Wraps `find_hvg_sc()` as an `ScStep`.

Usage

```
step_hvg_sc(
  hvg_no = 2000L,
  hvg_params = params_sc_hvg(),
  streaming = NULL,
  .verbose = TRUE
)
```

Arguments

`hvg_no`	Integer. Number of highly variable genes to include. Defaults to `2000L`.
`hvg_params`	List, see `params_sc_hvg()`. This list contains • method - Which method to use. One of `c("vst", "meanvarbin", "dispersion", "scran", "residual")` • loess_span - The span for the loess function to standardise the variance (`"vst"`), or of the lowess trend (`"scran"`) • num_bin - Integer. Not yet implemented. • bin_method - String. One of `c("equal_width", "equal_freq")`. Not implemented yet. • mean_filter, min_mean, transform, use_min_width, min_width, min_window_count - The `"scran"` trend parameters, see `params_hvg_scran_defaults()`
`streaming`	Optional Boolean. Shall the data be streamed in. Useful for larger data sets where you wish to avoid loading in the whole data. If `NULL`, will automatically detect. Not used for `MetaCells`.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

An `ScStep`.

Examples

```
# a step is inert until the pipeline is applied
step_hvg_sc(hvg_no = 30L)
```

`step_metacells_sc` *Pipeline step: generate meta cells*

Description

Wraps the meta cell generators as an `ScStep`. Unlike the other steps this one changes the class of the object: it takes a `SingleCells` or `SingleCellsSubset` and returns a `MetaCells()`. Steps that follow it need `MetaCells` methods, which `validate_pipeline()` checks up front.

Combined with `apply_pipeline_per_group()` this gives you per-group pre-processing (HVG, PCA, batch correction within a patient, kNN) followed by source-pure meta cells, which you then hand to `merge_meta_cells()`.

Usage

```
step_metacells_sc(method = c("bootstrapped", "seacells", "supercells"), ...)
```

Arguments

`method`	String. One of `c("bootstrapped", "seacells", "supercells")`.
`...`	Arguments passed on to the generator, e.g. `sc_meta_cell_params`, `target_size` or `.verbose`.

Value

An `ScStep`.

Examples

```
# per group pre-processing that ends on source-pure meta cells
sc <- demo_single_cells(prepped = FALSE)
pipeline <- step_hvg_sc(hvg_no = 30L, .verbose = FALSE) %>>%
  step_pca_sc(no_pcs = 10L, .verbose = FALSE) %>>%
  step_neighbours_sc(.verbose = FALSE) %>>%
  step_metacells_sc(
    "bootstrapped",
    sc_meta_cell_params = params_sc_bt_metacells(target_no_metacells = 10L),
    .verbose = FALSE
  )

per_group <- apply_pipeline_per_group(pipeline, sc, group_col = "cell_grp")
merge_meta_cells(per_group, .verbose = FALSE)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`step_neighbours_sc` *Pipeline step: nearest neighbours*

Description

Wraps `find_neighbours_sc()` as an `ScStep`.

Usage

```
step_neighbours_sc(
  embd_to_use = "pca",
  no_embd_to_use = NULL,
  modality = c("rna", "adt"),
  neighbours_params = params_sc_neighbours(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`embd_to_use`	String. The embedding to use. Whichever you chose, it needs to be part of the object.
`no_embd_to_use`	Optional integer. Number of embedding dimensions to use. If `NULL` all will be used.
`modality`	String. One of `c("rna", "adt")`. You can only use `"adt"` on `SingleCellsMultiModal` class.
`neighbours_params`	List. Output of `params_sc_neighbours()`. A list with the following items: • full_snn - Boolean. Shall the full shared nearest neighbour graph be generated that generates edges between all cells instead of between only neighbours. • pruning - Numeric. Weights below this threshold will be set to 0 in the generation of the sNN graph. • snn_similarity - String. One of `c("rank", "jaccard")`. Defines how the weight from the SNN graph is calculated. For details, please see `params_sc_neighbours()`. • knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.
`seed`	Integer. For reproducibility.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

An `ScStep`.

Examples

```
# neighbours on the PCA embedding
step_pca_sc(no_pcs = 10L) %>>% step_neighbours_sc()
```

---

step_pca_sc *Pipeline step: PCA*

Description

Wraps `calculate_pca_sc()` as an `ScStep`.

Usage

```
step_pca_sc(
  no_pcs = 30L,
  pca_params = params_sc_pca(),
  sparse_svd = NULL,
  hvg = NULL,
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`no_pcs`	Integer. Number of PCs to calculate.
`pca_params`	Named list. Controls the parameters to be used for the PCA calculation which is single cell-specific, see `params_sc_pca()`
`sparse_svd`	Optional boolean. Solve on the sparse data with implicit centring and scaling instead of materialising the dense scaled matrix. `NULL` picks the sparse path, which is faster and lighter on memory for every solver, or the dense one for `residuals = TRUE`. `FALSE` uses the dense path, but only below 500,000 cells. Not used for `MetaCells`.
`hvg`	Optional integer. If you want to provide your own HVG genes. Otherwise, the function will default to what is found in `get_hvg()`. Please provide 1-indexed genes here! If you provide these, the internal HVG will be overwritten.
`seed`	Integer. Controls reproducibility. Only relevant for `svd_solver = "randomised"` and the Lanczos start vector.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

An `ScStep`.

Examples

```
# PCA restricted to whatever the HVG step selected
step_hvg_sc(hvg_no = 30L) %>>% step_pca_sc(no_pcs = 10L)
```

---

stream_h5ad *Stream in h5ad to* SingleCells *(alias)*

Description

Convenience alias for `load_h5ad(h5ad_streaming = TRUE)`. Kept for backwards compatibility. Prefer calling `load_h5ad()` directly.

Usage

```
stream_h5ad(
  object,
  h5_path,
  sc_qc_param = params_sc_min_quality(),
  raw_count_slot = c("auto", "X", "raw.X", "layers.counts"),
  csc_mem_gb = NULL,
  streaming = deprecated(),
  batch_size = deprecated(),
  max_genes_in_memory = deprecated(),
  cell_batch_size = deprecated(),
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`h5_path`	File path to the h5ad object.
`sc_qc_param`	List. Output of `params_sc_min_quality()`.
`raw_count_slot`	Where raw counts live. `"auto"` detects per file via `detect_raw_count_slot()`; otherwise one of `"X"`, `"raw.X"`, `"layers.counts"`.
`csc_mem_gb`	Optional numeric. Memory in GB for the buffers of the cell-to-gene (CSR to CSC) conversion, at 10 bytes per non-zero. `NULL` (default) converts in one pass and holds the whole matrix. Set a cap for large data sets; every extra phase re-reads the cell file once.
`streaming`, `batch_size`, `max_genes_in_memory`, `cell_batch_size`	Replaced by `csc_mem_gb` and ignored. **[Deprecated]**
`.verbose`	Boolean.

Value

The class with updated shape information.

Examples

```
# same as load_h5ad(h5ad_streaming = TRUE)
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
f_path <- tempfile(fileext = ".h5ad")
write_h5ad_sc(f_path, data$counts, data$obs, data$var, .verbose = FALSE)
dir_data <- tempfile("sc_stream")
dir.create(dir_data, recursive = TRUE)
sc <- stream_h5ad(
  object = SingleCells(dir_data = dir_data),
  h5_path = f_path,
  sc_qc_param = params_sc_min_quality(
    min_unique_genes = 5L,
    min_lib_size = 25L,
    min_cells = 5L
  ),
  .verbose = FALSE
)
dim(sc)

unlink(c(f_path, dir_data), recursive = TRUE, force = TRUE)
```

SymphonyReference *bixverse SymphonyReference class*

Description

Holds a Symphony reference: PCA loadings, per-HVG scaling stats, soft cluster centroids and the cached compression terms (Nr, C) needed to map queries without the reference cells. The Harmony-corrected reference embedding (`z_corr`) is retained for downstream label transfer. Reference cell labels (e.g. cell type annotations) can be snapshotted at build time via `label_columns` in `build_symphony_ref()` or attached post-hoc via `add_symphony_labels()`. For details on the method, refer to Kang et al.

Usage

```
SymphonyReference(
  hvg_gene_names,
  gene_means,
  gene_sds,
  loadings,
  z_corr,
  centroids,
  nr,
  c_cache,
  no_pcs,
  harmony_backend,
```

```
  batch_vars,
  slim = FALSE,
  z_orig = NULL,
  r = NULL,
  labels = NULL
)
```

Arguments

hvg_gene_names	Character vector of HVG gene names in reference loading order.
gene_means	Per-HVG mean of the normalised reference data.
gene_sds	Per-HVG standard deviation of the normalised reference data.
loadings	PCA gene loadings matrix (n_hvgs x d).
z_corr	Post-Harmony corrected embedding (N x d).
centroids	Cosine-normalised reference centroids (K x d).
nr	Reference cluster sizes; row-sums of `r` (length K).
c_cache	Cached `R * Z_corr` compression term (K x d).
no_pcs	Number of principal components.
harmony_backend	Which Harmony variant was used (`"v1"` or `"v2"`).
batch_vars	Names of the batch variables used during reference construction.
slim	Logical; if `TRUE`, `z_orig` and `r` are dropped. Default `FALSE`.
z_orig	Pre-Harmony PCA scores (N x d). `NULL` in slim references.
r	Soft cluster assignments (K x N). `NULL` in slim references.
labels	Optional `data.table` of reference cell labels with `nrow(z_corr)` rows, one column per label. `NULL` if no labels stored.

Value

Returns the `SymphonyReference` class for further operations.

References

Kang et al., Nat. Commun., 2021

Examples

```
# a reference built off a two batch synthetic data set
ref <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 500L,
    n_genes = 50L,
    n_batches = 2L
  )
)
```

```
symphony_ref <- build_symphony_ref(
  ref,
  batch_column = "batch_index",
  hvg = get_hvg(ref) + 1L,
  harmony_params = params_sc_harmony(k = 10L),
  no_pcs = 10L,
  label_columns = "cell_grp",
  .verbose = FALSE
)
symphony_ref

unlink(ref@dir_data, recursive = TRUE, force = TRUE)
```

`synthetic_bulk_cor_matrix`
Generates synthetic bulk RNAseq data

Description

Function generates synthetic bulkRNAseq data with heteroskedasticity (lowly expressed genes show higher variance) and optional co-expression modules for testing module detection methods. The ground truth comes back with the counts, so a recovered module or eigengene can be scored against what was actually simulated. Which generator to pick and what each parameter does is documented in `params_synthetic_bulk_rnaseq()`.

Usage

```
synthetic_bulk_cor_matrix(synthetic_params = params_synthetic_bulk_rnaseq())
```

Arguments

`synthetic_params`
: List. The synthetic data parameters, see `params_synthetic_bulk_rnaseq()`.

Value

A `synthetic_bulk_data` class containing:

- counts - The count matrix. Rows are genes, columns are samples.
- sparse_counts - A slot for sparse counts that can be added later, see `simulate_dropouts()`.
- module_data - Per-gene ground truth: the gene identifier, its `membership` (`0` for background), its `loading` on the module factor and whether it is a hub gene.
- module_factors - The latent factors the modules were built on. Rows are modules, columns are samples. This is what a module eigengene or an ICA/NMF component is trying to recover.

The parameters used are stored on the `synthetic_params` attribute.

Examples

```
# three co-expression modules on a 1000 x 100 count matrix
syn <- synthetic_bulk_cor_matrix(params_synthetic_bulk_rnaseq())
dim(syn$counts)
head(syn$module_data)
dim(syn$module_factors)
```

`synthetic_c_pca_data` *Generates synthetic data for contrastive PCA exploration.*

Description

Generates three elements: target matrix, background matrix and labels for the target matrix.

Usage

```
synthetic_c_pca_data(seed = 10101L)
```

Arguments

`seed`	Integer. Initial random seed for generation of the synthetic data. Default: 10101L.

Value

A `cpca_synthetic_data` class with the following elements:

- target - The target matrix.
- background - The background matrix.
- target_labels - The target labels

Examples

```
# target and background matrices for a contrastive PCA run
cpca_data <- synthetic_c_pca_data()
dim(cpca_data$target)
dim(cpca_data$background)
table(cpca_data$target_labels)
```

`synthetic_signal_matrix` *Generates a simple synthetic, pseudo gene expression matrix*

Description

This is a function to create a synthetic, pseudo gene expression matrix with a pre-defined number of total genes, groups and number of genes differentially expressed in the groups. The function will create in a standard setting a 1000 x 90 matrix, and add optionally a small group into the mixture that is slightly more difficult to detect

Usage

```
synthetic_signal_matrix(
  no_grps = 3,
  per_group = 30,
  total_genes = 1000L,
  no_genes_up = 100L,
  add_small_group = TRUE,
  size_small_grp = 5L,
  seed = 10101L
)
```

Arguments

`no_grps`	Integer. Number of initial larger groups. Default: 3L.
`per_group`	Integer. Number of samples per larger groups. Default: 30L.
`total_genes`	Integer. Number of total genes in the matrix. Default: 1000L.
`no_genes_up`	Integer. Number of genes per group that are differentially expressed. Default: 100L.
`add_small_group`	Boolean. Add a smaller group that overlaps with group 1? Default: TRUE.
`size_small_grp`	Integer. Size of the smaller group.
`seed`	Integer. Initial random seed for generation of the synthetic data. Default: 10101L.

Value

A `synthetic_matrix_simple` class containing:

- mat - The random matrix
- diff - List of differentially expressed genes per group
- group - Look-up vector for sample to group mapping

Examples

```
# default 1000 x 90 matrix with three groups and a small fourth one
synthetic_gex <- synthetic_signal_matrix()
dim(synthetic_gex$mat)
table(synthetic_gex$group)
lengths(synthetic_gex$diff)
```

---

test_nhoods *Test neighbourhoods for differential abundance*

Description

Performs differential abundance testing on single-cell neighbourhoods with edgeR's quasi-likelihood negative binomial framework, implemented in Rust via the `edge-rs` crate. A generalised linear model is fitted to the neighbourhood counts, one coefficient or contrast is tested, and the spatial FDR correction accounts for the fact that neighbourhoods overlap and their tests are therefore not independent.

`filterByExpr()` is off here and cannot be turned on. It is a gene expression heuristic and means nothing for a neighbourhood. Use `min_mean` if you want to drop sparsely populated neighbourhoods.

Usage

```
test_nhoods(
  x,
  design,
  design_df,
  coef = NULL,
  contrast = NULL,
  norm_method = c("TMM", "TMMwsp", "RLE", "upperquartile", "logMS"),
  min_mean = 0,
  robust = TRUE,
  legacy = TRUE,
  fdr_weighting = c("k-distance", "graph-overlap", "none")
)

## S3 method for class 'miloR'
test_nhoods(
  x,
  design,
  design_df,
  coef = NULL,
  contrast = NULL,
  norm_method = c("TMM", "TMMwsp", "RLE", "upperquartile", "logMS"),
  min_mean = 0,
  robust = TRUE,
  legacy = TRUE,
```

```
  fdr_weighting = c("k-distance", "graph-overlap", "none")
)
```

Arguments

x	miloR object for which to run the differential abundance analysis.
design	Formula for the experimental design, e.g. ~ grps.
design_df	data.frame. The metadata used to build the model matrix. Its rownames need to cover the sample names of the neighbourhood counts.
coef	Optional integer or character. Which coefficient(s) of the design to drop from the null model, given as 1-based column positions or column names. Defaults to the last column, as edgeR does.
contrast	Optional numeric vector or matrix. Weights over the design columns. Mutually exclusive with coef.
norm_method	String. Library size normalisation. One of c("TMM", "TMMwsp", "RLE", "upperquartile", "logMS"). Defaults to "TMM". "logMS" is Milo's own name for leaving every factor at one.
min_mean	Numeric. Minimum mean count across samples. Neighbourhoods below it are dropped. Defaults to 0 (no filtering).
robust	Logical. Robust estimation of the quasi-likelihood dispersion. Defaults to TRUE.
legacy	Logical. Take edgeR's pre-4.0 quasi-likelihood pipeline, which runs estimateDisp() and applies the Poisson bound. Defaults to TRUE, so this keeps matching what Milo itself does.
fdr_weighting	String. Spatial FDR weighting scheme. One of c("k-distance", "graph-overlap", "none"). "k-distance" weights by the distance to the k-th nearest neighbour, "graph-overlap" by the number of cells shared with other neighbourhoods. Defaults to "k-distance".

Value

The miloR object with the differential abundance results added.

References

Dann, et al., Nat Biotechnol, 2022; Chen, Lun and Smyth, F1000Research, 2016

Examples

```
# differential abundance of neighbourhoods across two sample groups
sc <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 500L,
    n_genes = 50L,
    n_samples = 6L,
    sample_bias = "even"
  )
)
```

```
milo <- get_miloR_abundances_sc(
  sc,
  sample_id_col = "sample_id",
  miloR_params = params_sc_miloR(k_refine = 10L),
  .verbose = FALSE
)
design_df <- data.frame(
  grp = rep(c("a", "b"), each = 3),
  row.names = sprintf("sample_%i", 1:6)
)
milo <- test_nhoods(milo, design = ~grp, design_df = design_df)
head(get_differential_abundance_res(milo))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`tf_to_genes_correlations`

Generate TF to gene correlations

Description

This function will calculate the correlations between the identified TF to gene pairs. You need to have run `identify_tf_to_genes()`!

Following SCENIC, the correlation is turned into a three-level sign at `rho_threshold`: `+1` for activating links, `-1` for repressing ones and `0` for everything in the band between. `mode` then decides which of those you keep. The default keeps the activating links only, which is what SCENIC does with `onlyPositiveCorr = TRUE`.

Usage

```
tf_to_genes_correlations(
  x,
  object,
  rho_threshold = 0.03,
  mode = c("activating", "repressing", "both"),
  remove_self = TRUE,
  spearman = TRUE,
  cor_filter = NULL,
  .verbose = TRUE
)

## S3 method for class 'ScenicGrn'
tf_to_genes_correlations(
  x,
  object,
  rho_threshold = 0.03,
  mode = c("activating", "repressing", "both"),
  remove_self = TRUE,
```

```
  spearman = TRUE,
  cor_filter = NULL,
  .verbose = TRUE
)
```

Arguments

x	`ScenicGrn` object for which to generate the TF to gene associations.
object	`SingleCells` or `MetaCells` object that was used to generate the original GRNs.
rho_threshold	Float. Absolute correlation above which a TF to gene link counts as activating or repressing. Defaults to `0.03`, the SCENIC value.
mode	String. Which links to keep. One of `c("activating", "repressing", "both")`. Defaults to `"activating"`.
remove_self	Boolean. Shall self loops (where TF controls its own expression) be removed. Defaults to `TRUE`. Note that `build_regulons()` adds the TF back to its own regulon, which is also what SCENIC does.
spearman	Boolean. Shall Spearman correlation be used. Defaults to `TRUE`.
cor_filter	Deprecated. Use `rho_threshold` and `mode` instead. If given, it is treated as a one-sided lower bound on the correlation and `rho_threshold` and `mode` are ignored.
.verbose	Boolean. Controls verbosity of the function.

Value

Adds a `pairwise_cor` and a `cor_sign` column to the TF to gene results.

References

Aibar, et al., Nat Methods, 2017

Examples

```
# sign the TF to gene links and keep the activating ones
sc <- demo_single_cells()
grn <- scenic_grn_sc(
  sc,
  tf_ids = sprintf("gene_%02d", 1:5),
  scenic_params = params_scenic(
    min_counts = 1L,
    learner_params = list(n_trees = 20L)
  ),
  .verbose = FALSE
)
grn <- identify_tf_to_genes(
  grn,
  method = "top_k",
  k_tfs = 3L,
  .verbose = FALSE
```

```
)
grn <- tf_to_genes_correlations(grn, object = sc, .verbose = FALSE)
head(get_tf_to_gene(grn))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`tf_to_genes_motif_enrichment` *Run the SCENIC motif enrichment*

Description

This function will run the motif enrichment based on RCisTarget (or in this case the internal implementation via `run_cistarget()`). You need to provide the expected rankings and TF annotations (for details, please see `run_cistarget()`). Briefly, this function will run CisTarget and add the results to the `ScenicGrn` object and add additionally a column `"in_motif"`, for a given TF to gene set to say if it was part of the motifs associated with this TF (or not). You have the option to limit this to only the the high confidence TFs (default), or also include the low confidence TFs (i.e., links from TF to motif that are less certain).

Usage

```
tf_to_genes_motif_enrichment(
  x,
  motif_rankings,
  annot_data,
  cis_target_params = params_cistarget(),
  gene_id_to_symbol = NULL,
  only_high_conf_tf = TRUE,
  .verbose = TRUE
)

## S3 method for class 'ScenicGrn'
tf_to_genes_motif_enrichment(
  x,
  motif_rankings,
  annot_data,
  cis_target_params = params_cistarget(),
  gene_id_to_symbol = NULL,
  only_high_conf_tf = TRUE,
  .verbose = TRUE
)
```

Arguments

`x`	`ScenicGrn` object for which to generate the TF to gene associations.

`motif_rankings`
: Integer matrix. Motif rankings for genes. Row names are gene identifiers, column names are motif identifiers. Lower values indicate higher regulatory potential.

`annot_data`
: data.table. Motif annotation database mapping motifs to transcription factors. Must contain columns: `motif`, `TF`, and `annotationSource`.

`cis_target_params`
: List. Output of `params_cistarget()`:
  - auc_threshold - Numeric. Proportion of genes to use for AUC threshold calculation. Default 0.05 means top 5 percent of genes.
  - nes_threshold - Numeric. Normalised Enrichment Score threshold for determining significant motifs. Default is 3.0.
  - max_rank - Integer. Depth of the recovery curves used for the background and the leading edge. Default is 5000.
  - n_mean - Integer. Rolling mean window for the approximate background curve. Default is 100.
  - rcc_method - Character. Recovery curve calculation method: "approx" (approximate, faster) or "icistarget" (exact, slower).
  - high_conf_cats - Character vector. Annotation categories considered high confidence (e.g., "directAnnotation", "inferredBy_Orthology").
  - low_conf_cats - Character vector. Annotation categories considered lower confidence (e.g., "inferredBy_MotifSimilarity").

`gene_id_to_symbol`
: Named character vector. Mapping from gene identifiers used internally (e.g., Ensembl IDs) to the identifiers used in the motif rankings (e.g., HGNC symbols). Names are internal IDs, values are ranking IDs. If `NULL` (default), no mapping is applied and the internal IDs are assumed to match the ranking rownames.

`only_high_conf_tf`
: Boolean. Shall only the high confidence TF to motif association be used. Defaults to `TRUE`.

`.verbose`
: Boolean. Controls verbosity of the function.

Value

Adds a data.table with the first tf to gene results to the class.

Examples

```
# CisTarget against a synthetic ranking database
sc <- demo_single_cells()
tfs <- sprintf("gene_%02d", 1:5)
grn <- scenic_grn_sc(
  sc,
  tf_ids = tfs,
  scenic_params = params_scenic(
    min_counts = 1L,
    learner_params = list(n_trees = 20L)
```

```
  ),
  .verbose = FALSE
)
grn <- identify_tf_to_genes(
  grn,
  method = "top_k",
  k_tfs = 3L,
  .verbose = FALSE
)
genes <- get_gene_names(sc)
targets <- split(get_tf_to_gene(grn)$gene, get_tf_to_gene(grn)$tf)
rankings <- matrix(
  vapply(
    tfs,
    function(tf) {
      as.integer(rank(!(genes %in% targets[[tf]]), ties.method = "first"))
    },
    integer(length(genes))
  ),
  nrow = length(genes),
  dimnames = list(genes, sprintf("motif_%s", tfs))
)
annot <- data.table::data.table(
  motif = sprintf("motif_%s", tfs),
  TF = tfs,
  annotationSource = factor(
    c(rep("directAnnotation", 4), "inferredBy_MotifSimilarity")
  )
)
grn <- tf_to_genes_motif_enrichment(
  grn,
  motif_rankings = rankings,
  annot_data = annot,
  cis_target_params = params_cistarget(
    auc_threshold = 1,
    nes_threshold = 1,
    high_conf_cats = "directAnnotation",
    low_conf_cats = "inferredBy_MotifSimilarity"
  ),
  .verbose = FALSE
)
head(get_tf_to_gene(grn))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`tied_diffusion` *Diffuse seed genes in a tied manner over a network*

Description

This function takes two sets of diffusion vector and leverages tied diffusion to identify an intersection of influential nodes. If the network is undirected, the method will run two

personalised page rank diffusions based on the diffusion vectors and generate the score aggregation

Usage

```
tied_diffusion(
  object,
  diffusion_vector_1,
  diffusion_vector_2,
  summarisation = c("max", "mean", "harmonic_sum"),
  score_aggregation = c("min", "max", "mean"),
  .verbose = FALSE
)
```

Arguments

`object`	`NetworkDiffusions` object. The underlying class `NetworkDiffusions()`.
`diffusion_vector_1`	Named numeric. The first named vector with values to use for the reset parameter in the personalised page-rank diffusion. Names should represent node names of the graph.
`diffusion_vector_2`	Named numeric. The second named vector with values to use for the reset parameter in the personalised page-rank diffusion. Names should represent node names of the graph.
`summarisation`	String. If there are duplicated names in the `diffusion_vector` how to summarise these.
`score_aggregation`	String. How to summarise the tied scores.
`.verbose`	Boolean. Controls verbosity of the function.

Value

The class with added diffusion score based on a two sets of seed genes. Additionally, the seed genes are stored in the class.

Examples

```
# tied diffusion between two sets of seed nodes
set.seed(42)
g <- igraph::sample_pa(15, directed = FALSE)
edges <- data.table::setDT(igraph::as_data_frame(g))[, `:=`(
  from = sprintf("node_%i", from),
  to = sprintf("node_%i", to)
)]
object <- NetworkDiffusions(edges, weighted = FALSE, directed = FALSE)
object <- tied_diffusion(
  object,
  diffusion_vector_1 = c(node_1 = 1, node_3 = 1),
  diffusion_vector_2 = c(node_2 = 1, node_6 = 1),
```

```
  summarisation = "max",
  score_aggregation = "min"
)
head(get_diffusion_vector(object))
```

`to_snake_case` *Helper function to transform strings to snake_case*

Description

Helper function to transform strings to snake_case

Usage

```
to_snake_case(x, ignore_na = FALSE)
```

Arguments

`x`	String. The string (vector) you wish to transform to snake_case.
`ignore_na`	Boolean. Shall the function just ignore `NA` values and return `NA` at this position. Defaults to `FALSE`.

Value

Returns the string in snake_case format.

Examples

```
# normalise mixed naming conventions
to_snake_case(c("Gene Name", "someRNAValue", "Foo-Bar"))
```

`top_genes_perc_sc` *Calculate the proportions of reads for the Top N genes*

Description

This is a helper function that calculates proportions of reads to the Top N genes by expression in a given cell. High values here can indicate low complexity, quality cells. The values will be automatically added to the obs table.

Usage

```
top_genes_perc_sc(
  object,
  top_n_vals = c(25L, 50L, 100L),
  streaming = NULL,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells` class.
`top_n_vals`	Integer. The Top N thresholds to test.
`streaming`	Optional Boolean. Shall the data be streamed in. Useful for larger data sets where you wish to avoid loading in the whole data. If `NULL`, will automatically detect.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

It will add the columns based on the names in the `gene_set_list` to the obs table.

Examples

```
# share of a cell's reads taken by its top 5 and top 10 genes
sc <- demo_single_cells(prepped = FALSE)
sc <- top_genes_perc_sc(sc, top_n_vals = c(5L, 10L), .verbose = FALSE)
head(unlist(sc[["top_5_genes_percentage"]]))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

transfer_labels_symphony

Transfer labels from a Symphony reference to a query via kNN majority vote

---

Description

kNN majority vote on the Harmony-corrected embeddings. The reference's `z_corr` is the searchable index; the query's `"symphony"` embedding is what gets queried. Reference labels come from the reference's stored `labels` slot — populate it via the `label_columns` argument to `build_symphony_ref()` or post-hoc via `add_symphony_labels()`.

Distances on `z_corr` are typically Euclidean — set `knn_params$ann_dist` to `"euclidean"` unless you have a reason to use cosine.

Usage

```
transfer_labels_symphony(
  reference,
  query,
  label_column,
  knn_params = params_sc_knn(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`reference`	`SymphonyReference` (must have `z_corr` and stored labels).
`query`	`SingleCells` with a `"symphony"` embedding attached.
`label_column`	String. Name of a column in the reference's stored labels.
`knn_params`	List. Output of `params_sc_knn()`.
`seed`	Integer.
`.verbose`	Boolean or integer.

Value

A data.table with columns `predicted_<label_column>` and `confidence_<label_column>`, in `get_cells_to_keep(query)` order.

Examples

```
# kNN label transfer from the reference onto a mapped query
ref <- demo_single_cells(
  syn_data_params = params_sc_synthetic_data(
    n_cells = 500L,
    n_genes = 50L,
    n_batches = 2L
  )
)
symphony_ref <- build_symphony_ref(
  ref,
  batch_column = "batch_index",
  hvg = get_hvg(ref) + 1L,
  harmony_params = params_sc_harmony(k = 10L),
  no_pcs = 10L,
  label_columns = "cell_grp",
  .verbose = FALSE
)
query <- demo_single_cells(prepped = FALSE, seed = 7L)
query <- map_symphony_query(symphony_ref, query = query, .verbose = FALSE)
labels <- transfer_labels_symphony(
  symphony_ref,
  query = query,
  label_column = "cell_grp",
  .verbose = FALSE
)
head(labels)

unlink(c(ref@dir_data, query@dir_data), recursive = TRUE, force = TRUE)
```

tsne_sc *Run t-SNE on a SingleCells/MetaCells object*

Description

Wrapper around manifoldsR::tsne() for the SingleCells and MetaCells classes. t-SNE produces a low-dimensional embedding that emphasises local neighbourhood structure. Distances between well-separated clusters should not be over-interpreted quantitatively, but the common claim that t-SNE discards global structure while UMAP preserves it is largely an artefact of default initialisations rather than a property of the loss functions themselves.

When use_knn = TRUE, the kNN graph already stored on the object is reused. With use_knn = FALSE (the default), neighbours are computed from the chosen embedding.

Three approximation strategies are available via approx_type: "bh" (Barnes-Hut) is the classical O(n log n) approximation and works well across a wide range of dataset sizes; "fft" (interpolation-based, as in FIt-SNE) scales better to very large datasets; "fft_3k" is the three-kernel variant of the latter, with one forward and three inverse FFTs per epoch instead of four each. The FFT options are only available on Unix systems. perplexity controls the bandwidth of the Gaussian kernel used to compute affinities within the neighbour set (typical values 5-50). When a pre-computed kNN is supplied via use_knn = TRUE, perplexity no longer drives neighbour retrieval but still shapes the affinity distribution over the retrieved neighbours; values too close to the kNN size will produce poor results. With tSNE in particular the rule of thumb is to set k to 3 * perplexity. When k <= perplexity the algorithm does not behave properly anymore, thus, will throw an error.

Usage

```
tsne_sc(
  object,
  use_knn = FALSE,
  embd_to_use = "pca",
  slot_name = "tsne",
  no_embd_to_use = NULL,
  modality = c("rna", "adt", "wnn"),
  n_dim = 2L,
  perplexity = 10,
  approx_type = c("bh", "fft", "fft_3k"),
  knn_method = c("kmknn", "hnsw", "annoy", "nndescent", "balltree", "ivf", "exhaustive"),
  nn_params = manifoldsR::params_nn(),
  tsne_params = manifoldsR::params_tsne(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

object SingleCells, MetaCells class.

use_knn — Boolean. Use the kNN graph found in the object. Defaults to FALSE. If not available, will default to the embedding.

embd_to_use — String. The embedding to use for t-SNE. Must be available in the object.

slot_name — String. The name of this embedding within the object. Defaults to "tsne".

no_embd_to_use — Optional integer. Number of embedding dimensions to use. If NULL all will be used.

modality — String. On which modality to run the t-SNE. One of c("rna", "adt", "wnn"). The two latter options are only available for multi-modal versions with the added data.

n_dim — Integer. Number of t-SNE dimensions. Currently only 2L is supported. Defaults to 2L.

perplexity — Numeric. Perplexity parameter. Typical values between 5 and 50. Defaults to 10.0.

approx_type — String. Approximation method. One of "bh" (Barnes-Hut), "fft" or "fft_3k". Defaults to "bh".

knn_method — String. Approximate nearest neighbour algorithm. One of c("kmknn", "hnsw", "annoy", "nndescent", "balltree", "ivf", "exhaustive"). Defaults to "kmknn". Only used when neighbours are computed from the embedding.

nn_params — Named list. See manifoldsR::params_nn().

tsne_params — Named list. See manifoldsR::params_tsne().

seed — Integer. For reproducibility.

.verbose — Boolean. Controls verbosity.

Value

The object with a "tsne" embedding added.

Examples

```
# Barnes-Hut t-SNE on the PCA factors
sc <- demo_single_cells()
sc <- tsne_sc(sc, .verbose = FALSE)
dim(get_embedding(sc, "tsne"))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`umap_sc` *Run UMAP on a SingleCells/MetaCells object*

Description

Wrapper around `manifoldsR::umap()` for the `SingleCells` and `MetaCells` classes. UMAP produces a low-dimensional embedding that emphasises local neighbourhood structure while being computationally efficient via its negative-sampling-based optimisation. It is the de facto default for visualising single-cell data, though claims that it preserves global structure substantially better than t-SNE are not well supported; with matched initialisation (e.g. PCA or Laplacian Eigenmaps), the two methods behave similarly on global geometry, and both should be interpreted primarily as views of local structure.

When `use_knn = TRUE` (the default), the kNN graph already stored on the object (via `find_neighbours_sc()`) is reused, which avoids recomputing nearest neighbours and keeps the UMAP consistent with any downstream sNN-based clustering. If no kNN is present, neighbours are computed from the chosen embedding on the fly.

Key parameters to tune: `k` controls the balance between local and global structure (larger values produce more global layouts), while `min_dist` and `spread` together control how tightly points are packed in the embedding. For `MetaCells`, smaller `k` values are often appropriate given the reduced number of points.

Usage

```
umap_sc(
  object,
  use_knn = TRUE,
  embd_to_use = "pca",
  slot_name = "umap",
  no_embd_to_use = NULL,
  modality = c("rna", "adt", "wnn"),
  n_dim = 2L,
  k = 15L,
  min_dist = 0.5,
  spread = 1,
  knn_method = c("kmknn", "hnsw", "annoy", "nndescent", "balltree", "ivf", "exhaustive"),
  nn_params = manifoldsR::params_nn(),
  umap_params = manifoldsR::params_umap(),
  seed = 42L,
  .verbose = TRUE
)
```

Arguments

`object`	`SingleCells`, `MetaCells` class.
`use_knn`	Boolean. Use the kNN graph found in the object. Defaults to `TRUE`. If not available, will default to the embedding.

`embd_to_use`	String. The embedding to use for UMAP. Must be available in the object.
`slot_name`	String. The name of this embedding within the object. Defaults to `"umap"`.
`no_embd_to_use`	Optional integer. Number of embedding dimensions to use. If `NULL` all will be used.
`modality`	String. On which modality to run the UMAP. One of `c("rna", "adt", "wnn")`. The two latter options are only available for multi-modal versions with the added data.
`n_dim`	Integer. Number of UMAP dimensions. Defaults to `2L`.
`k`	Integer. Number of nearest neighbours. Defaults to `15L`.
`min_dist`	Numeric. Minimum distance between embedded points. Defaults to `0.5`.
`spread`	Numeric. Effective scale of embedded points. Defaults to `1.0`.
`knn_method`	String. Approximate nearest neighbour algorithm. One of `c("kmknn", "hnsw", "annoy", "nndescent", "balltree", "ivf", "exhaustive")`. Defaults to `"kmknn"`. Only used when neighbours are computed from the embedding.
`nn_params`	Named list. See `manifoldsR::params_nn()`.
`umap_params`	Named list. See `manifoldsR::params_umap()`.
`seed`	Integer. For reproducibility.
`.verbose`	Boolean. Controls verbosity.

Value

The object with a `"umap"` embedding added.

Examples

```
# UMAP off the cached kNN graph
sc <- demo_single_cells()
sc <- umap_sc(sc, .verbose = FALSE)
dim(get_embedding(sc, "umap"))

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

update_metadata_values

Replace values in a metadata column

Description

This function will update the values in a given metadata column based on what you are providing in terms of replacement.

Usage

```
update_metadata_values(object, column, replacement, ...)
```

Arguments

`object`	The underlying object, either `BulkCoExp()` or `BulkDge()`.
`column`	Character vector. The columns for which to replace the values.
`replacement`	Named character vector. The values with which to replace the data.
`...`	Additional arguments to parse to the functions.

Value

Returns the object with the respective metadata updated.

Examples

```
# relabel the levels of a metadata column
syn <- synthetic_bulk_cor_matrix()
meta <- data.table::data.table(
  sample_id = colnames(syn$counts),
  case_control = rep(c("case", "control"), each = 50)
)
object <- BulkDge(raw_counts = syn$counts, meta_data = meta)
object <- update_metadata_values(
  object,
  column = "case_control",
  replacement = c(case = "disease", control = "healthy")
)
unique(get_metadata(object)$case_control)
```

`upper_triangle_to_sparse` *Transform an upper triangle-stored matrix to a sparse one*

Description

Transform an upper triangle-stored matrix to a sparse one

Usage

```
upper_triangle_to_sparse(upper_triangle_vals, shift, n, type = c("csc", "csr"))
```

Arguments

`upper_triangle_vals`	Numerical vector. The values of the upper triangle stored in a row major format.
`shift`	Boolean. Did you exclude the diagonal.
`n`	Integer. Number of columns and rows of the symmetric matrix.
`type`	String. One of `c("csc", "csr")`. Which type of of sparse matrix to return.

Value

The sparse matrix.

Examples

```
# 3 x 3 symmetric matrix from its off-diagonal upper triangle
mat <- upper_triangle_to_sparse(c(0.5, 0.2, 0.8), shift = TRUE, n = 3L)
dim(mat)
```

`upper_triangular_sym_mat` *Class for symmetric correlation matrices*

Description

The class allows to store the upper triangular matrix of a symmetric matrix (think correlation matrix, distance matrix, etc.) in an memory-efficient form and return a data.table or dense (or sparse) R matrix if need be.

Methods

Public methods:

- `upper_triangular_sym_mat$new()`
- `upper_triangular_sym_mat$print()`
- `upper_triangular_sym_mat$get_data_table()`
- `upper_triangular_sym_mat$get_sym_matrix()`
- `upper_triangular_sym_mat$get_sparse_matrix()`
- `upper_triangular_sym_mat$get_data()`
- `upper_triangular_sym_mat$clone()`

`upper_triangular_sym_mat$new()`: Initialises the R6 class.

Usage:

```
upper_triangular_sym_mat$new(values, features, shift)
```

Arguments:

`values` Numerical vector. The correlation coefficients of the upper triangular correlation matrix stored as a row-major vector

`features` String vector. The features of the correlation matrix.

`shift` Bollean Was a shift applied during the calculation of the upper triangular matrix. If `FALSE`, the diagonal was included, if `TRUE`, the diagonal was removed.

Returns: Returns the initialised class.

`upper_triangular_sym_mat$print()`: Print the class

Usage:

```
upper_triangular_sym_mat$print()
```

Returns: Returns the initialised class

`upper_triangular_sym_mat$get_data_table()`: Returns the data in form of a data.table.

Usage:

```
upper_triangular_sym_mat$get_data_table(factor = FALSE, .verbose = TRUE)
```

Arguments:

`factor` Boolean. Shall the string columns be transformed into factors. Reduces size of the object; however, takes longer to generate.

`.verbose` Boolean. Controls verbosity.

Returns: A data.table with three columns:

- feature_a: The name of the first feature in the correlation matrix.
- feature_b: The name of the second feature in the correlation matrix.
- cor: The correlation coefficients between these two features.

`upper_triangular_sym_mat$get_sym_matrix()`: Return the full correlation matrix.

Usage:

```
upper_triangular_sym_mat$get_sym_matrix(.verbose = TRUE)
```

Arguments:

`.verbose` Boolean. Controls verbosity.

Returns: Returns the correlation matrix as a dense R matrix.

`upper_triangular_sym_mat$get_sparse_matrix()`: Return a sparse version of the correlation matrix

Usage:

```
upper_triangular_sym_mat$get_sparse_matrix(.verbose = TRUE)
```

Arguments:

`.verbose` Boolean. Controls verbosity

Returns: The sparse matrix.

`upper_triangular_sym_mat$get_data()`: Return the correlation data and shift

Usage:

```
upper_triangular_sym_mat$get_data()
```

Returns: A list with

- cor_data - Numeric vector. The values.
- features - String. The feature names.
- n_features - Integer. Number of initial features.
- shift - Integer. The applied shift.

`upper_triangular_sym_mat$clone()`: The objects of this class are cloneable with this method.

Usage:

```
upper_triangular_sym_mat$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# store the off-diagonal of a correlation matrix and get it back
set.seed(42)
cor_mat <- cor(matrix(rnorm(80), nrow = 20, ncol = 4))
object <- upper_triangular_sym_mat$new(
  values = cor_mat[lower.tri(cor_mat)],
  features = sprintf("gene_%i", 1:4),
  shift = TRUE
)
object$get_data_table(.verbose = FALSE)
dim(object$get_sym_matrix(.verbose = FALSE))
```

---

validate_pipeline *Check that a pipeline can run on a given class*

Description

Walks the pipeline and tracks which class each step would receive. Steps declare the classes their generic has methods for, and most steps hand back what they were given; `step_metacells_sc()` does not, it turns a `SingleCells`/`SingleCellsSubset` into a `MetaCells`. Without this you would only find out about a mismatch when S7 fails to dispatch, i.e. after the expensive steps already ran.

Usage

```
validate_pipeline(pipeline, class)
```

Arguments

`pipeline`	`ScPipeline`.
`class`	String. Class of the object the pipeline would start on. One of `c("SingleCells", "SingleCellsSubset", "MetaCells")`.

Value

Invisibly, the class the pipeline would return.

Examples

```
# the meta cell step changes what the next step would receive
p <- step_hvg_sc() %>>% step_metacells_sc("bootstrapped")
print(validate_pipeline(p, "SingleCells"))
```

---

vision_sc *Calculate VISION scores*

Description

Calculates an VISION-type scores for pathways based on DeTomaso, et al. Compared to other score types, you can also calculate delta-type scores between positive and negative gene indices, think epithelial vs mesenchymal gene signature, etc.

Usage

```
vision_sc(object, gs_list, streaming = NULL, .verbose = TRUE)
```

Arguments

`object`	`SingleCells`, `MetaCells` (or potentially other) class.
`gs_list`	Named nested list. Every element must itself be a list with at least a `"pos"` element holding the gene identifiers, and optionally a `"neg"` one. A bare character vector is not accepted. The gene identifiers need to be part of the variables of the object.
`streaming`	Optional Boolean. Shall the data be streamed in. Useful for larger data sets where you wish to avoid loading in the whole data. If `NULL`, will automatically detect. Ignored when applied to `MetaCells`.
`.verbose`	Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

The VISION scores in form of a matrix that is cells x gene sets or as `ScMatrixRes` pending the input.

References

DeTomaso, et al., Nat. Commun., 2019

Examples

```
# a signed signature alongside a plain one
sc <- demo_single_cells()
gs_list <- list(
  programme_a = list(
    pos = get_gene_names(sc)[1:10],
    neg = get_gene_names(sc)[11:20]
  ),
  programme_b = list(pos = get_gene_names(sc)[21:30])
)
res <- vision_sc(sc, gs_list = gs_list, .verbose = FALSE)
dim(res)

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

---

vision_w_autocor_sc *Calculate VISION scores (with auto-correlation scores)*

Description

Calculates VISION-type scores for pathways based on DeTomaso, et al. Compared to other score types, you can also calculate delta-type scores between positive and negative gene indices, think epithelial vs mesenchymal gene signature, etc. Additionally, this function also calculates the auto- correlation values, answering the question if a given signature shows non- random enrichment on the kNN graph. The kNN graph (and distance measures) will be generated on-the-fly based on the embedding you wish to use.

Usage

```
vision_w_autocor_sc(
  object,
  gs_list,
  embd_to_use,
  no_embd_to_use = NULL,
  use_knn = TRUE,
  vision_params = params_sc_vision(),
  streaming = NULL,
  random_seed = 42L,
  .verbose = TRUE
)
```

Arguments

object	`SingleCells`, `MetaCells` (or potentially other) class.
gs_list	Named nested list. Every element must itself be a list with at least a `"pos"` element holding the gene identifiers, and optionally a `"neg"` one. A bare character vector is not accepted. The gene identifiers need to be part of the variables of the object.

`embd_to_use` String. The embedding to use. Whichever you chose, it needs to be part of the object.

`no_embd_to_use`
Optional integer. Number of embedding dimensions to use. If `NULL` all will be used.

`use_knn` Boolean. Shall the internal kNN be used. If set to yes, you need to ensure consistency.

`vision_params` List with vision parameters, see `params_sc_vision()` with the following elements:

- n_perm - Integer. Number of random permutations
- n_cluster - Integer. Number of random clusters to generate to associate each set with.
- knn - List of kNN parameters. See `params_knn_defaults()` for available parameters and their defaults.

`streaming` Optional Boolean. Shall the data be streamed in. Useful for larger data sets where you wish to avoid loading in the whole data. If `NULL`, will automatically detect. Ignored when applied to `MetaCells`.

`random_seed` Integer. The random seed.

`.verbose` Boolean or integer. Controls verbosity and returns run times. `FALSE` -> quiet, `TRUE` or `1L` -> normal verbosity, `2L` -> detailed verbosity.

Value

A list with the following elements:

- vision_matrix - Matrix of cells x signatures with the VISION pathway scores as values.
- auto_cor_dt - data.table with the auto-correlation results per gene set, i.e., `auto_cor` (1 - Gaery's C), `p_val` and `fdr`.

References

DeTomaso, et al., Nat. Commun., 2019

Examples

```
# scores plus whether they sit non-randomly on the kNN graph
sc <- demo_single_cells()
gs_list <- list(
  programme_a = list(pos = get_gene_names(sc)[1:10]),
  programme_b = list(pos = get_gene_names(sc)[21:30])
)
res <- vision_w_autocor_sc(
  sc,
  gs_list = gs_list,
  embd_to_use = "pca",
  vision_params = params_sc_vision(n_perm = 50L, n_cluster = 3L),
  .verbose = FALSE
)
```

```
res$auto_cor_dt

unlink(sc@dir_data, recursive = TRUE, force = TRUE)
```

`write_cellranger_output` *Helper function to write data to a cell ranger like output*

Description

This is a helper to write synthetic data to cell ranger like output, i.e., an .mtx file, an barcodes.csv (or .tsv) and a features.csv (or .tsv).

Usage

```
write_cellranger_output(
  f_path,
  counts,
  obs,
  var,
  format_type = c("csv", "tsv"),
  rows = c("cells", "genes"),
  overwrite = TRUE,
  .verbose = TRUE
)
```

Arguments

`f_path`	String. The filepath to which to save the data
`counts`	Sparse matrix. Needs to be of class `dgRMatrix` or `dgCMatrix`.
`obs`	data.table. The observations. Needs to have `nrow(obs) == nrow(counts)`.
`var`	data.table. The variable data. Needs to have `ncol(var) == ncol(counts)`.
`format_type`	String. One of `c("csv", "tsv")`. Shall the data be saved in TSV or CSV.
`rows`	String. One of `c("cells", "genes")`. Shall the rows represent cells or genes in the .mtx file.
`overwrite`	Boolean. Shall any found h5ad file be overwritten.
`.verbose`	Boolean. Controls verbosity of the function.

Value

Returns invisible

Examples

```
# the 10x trio: an .mtx plus barcode and feature tables
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
dir_out <- tempfile("cellranger")
dir.create(dir_out, recursive = TRUE)
write_cellranger_output(
  f_path = dir_out,
  counts = data$counts,
  obs = data$obs,
  var = data$var,
  rows = "cells",
  format_type = "csv",
  .verbose = FALSE
)
list.files(dir_out)

unlink(dir_out, recursive = TRUE, force = TRUE)
```

---

write_h5ad_sc *Helper function to write data to h5ad format*

Description

This is a helper to write synthetic data to h5ad file. This version will write the data into the common compressed sparse data format.

Usage

```
write_h5ad_sc(f_path, counts, obs, var, overwrite = TRUE, .verbose = TRUE)
```

Arguments

f_path	String. The filepath to which to save the data
counts	Sparse matrix. Needs to be of class dgRMatrix or dgCMatrix.
obs	data.table. The observations. Needs to have nrow(obs) == nrow(counts).
var	data.table. The variable data. Needs to have ncol(var) == ncol(counts).
overwrite	Boolean. Shall any found h5ad file be overwritten.
.verbose	Boolean. Controls verbosity of the function.

Value

Returns invisible

Examples

```
# round trip synthetic counts through a sparse h5ad
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
f_path <- tempfile(fileext = ".h5ad")
write_h5ad_sc(f_path, data$counts, data$obs, data$var, .verbose = FALSE)
get_h5ad_dimensions(f_path)$dims

unlink(f_path)
```

---

write_h5ad_sc_dense *Helper function to write data to a dense h5ad file*

Description

Helper to write synthetic data to h5ad with `/X` stored as a dense 2D dataset (cells x genes). Useful for exercising the dense ingestion path.

Usage

```
write_h5ad_sc_dense(
  f_path,
  counts,
  obs,
  var,
  overwrite = TRUE,
  .verbose = TRUE
)
```

Arguments

`f_path`	String. The filepath to which to save the data
`counts`	Matrix or sparse matrix; sparse input is densified.
`obs`	data.table. The observations. Needs `nrow(obs) == nrow(counts)`.
`var`	data.table. The variable data. Needs `nrow(var) == ncol(counts)`.
`overwrite`	Boolean. Shall any found h5ad file be overwritten.
`.verbose`	Boolean. Controls verbosity of the function.

Value

Returns invisible

Examples

```
# the same data with a dense /X, which reads back as DENSE_ROW
data <- generate_single_cell_test_data(
  syn_data_params = params_sc_synthetic_data(n_cells = 200L, n_genes = 40L)
)
f_path <- tempfile(fileext = ".h5ad")
write_h5ad_sc_dense(
  f_path, data$counts, data$obs, data$var, .verbose = FALSE
)
get_h5ad_dimensions(f_path)$type

unlink(f_path)
```

Index

∗ **datasets**
    cell_cycle_genes, 97
$.SingleCellCountData
        (SingleCellCountData), 638
%>>%, 15, *616*

add_adt_counts_sc, 15
add_adt_counts_sc(), *129*
add_new_metadata, 16
add_nhoods_info, 17
add_sc_new_obs, 18
add_sc_new_obs(), *337*
add_snf_data_modality, 19
add_symphony_labels, 20
add_symphony_labels(), *40*, *662*, *676*
AnnDataParser, 21
apply_pipeline, 23
apply_pipeline(), *616*
apply_pipeline_per_group, 24
apply_pipeline_per_group(), *336*, *658*
archive_sc_exp, 25
as_ligand_target_influence, 26
assert_sc_state, 26
assign_sc_type, 27
assign_sc_type(), *464*
aucell_sc, 29
aucell_sc(), *34*, *37*

batch_correction_bulk_dge, 30
batch_correction_bulk_dge(), *225*
bbknn_sc, 31
bbknn_sc(), *652*
binarise_regulon_activity, 33
binarise_regulon_activity(), *147*, *605*
BixverseBaseClass(), *262*, *269*
blitzgsea_calibrate, 34
blitzgsea_calibrate(), *45*, *46*
bonsai_sc, 35
bonsai_sc(), *406*, *625*
build_regulons, 37
build_regulons(), *670*
build_symphony_ref, 39
build_symphony_ref(), *20*, *662*, *676*
bulk_dge_from_h5ad, 41
BulkCoExp, 42
BulkCoExp(), *43*, *44*, *106*, *112–119*, *121–123*, *129*, *131*, *134*, *182*, *211*, *212*, *220*, *226*, *231*, *238*, *243*, *253*, *257–259*, *261*, *268*, *299*, *301–303*, *351–353*, *477–480*, *487*, *488*, *509*, *650*, *682*
BulkDge, 43
BulkDge(), *30*, *41*, *66*, *71*, *72*, *82*, *102*, *182*, *222–224*, *233*, *253*, *261*, *285*, *357*, *391*, *481*, *489*, *497*, *509*, *682*

c_pca_plot_alphas, 44
calc_blitzgsea, 45
calc_diffusion_coordinates, 47
calc_diffusion_coordinates(), *55*, *56*, *59*, *334*
calc_fgsea, 48
calc_fgsea_simple, 49
calc_gsea_traditional, 51
calc_gsva, 52
calc_knn_metrics, 54
calc_manifold_metrics, 55
calc_manifold_metrics(), *56*, *58*, *59*, *334*
calc_meta_cell_compactness, 56
calc_meta_cell_purity, 57
calc_meta_cell_purity(), *251*
calc_meta_cell_separation, 58
calc_mitch, 59
calc_sc_type_scores, 60
calc_sc_type_scores(), *28*
calc_singscore, 61
calc_singscore_multi, 62
calc_singscore_rank, 64
calc_singscore_rank(), *61*, *63*

calc_ssgsea, 65
calculate_all_dges, 66
calculate_batch_asw_sc, 67
calculate_batch_asw_sc(), *68*
calculate_cell_type_asw_sc, 68
calculate_critical_value, 69
calculate_critical_value(), *167*
calculate_dge_hedges, 70
calculate_dge_hedges(), *30*, *66*
calculate_dge_limma, 71
calculate_dge_limma(), *66*
calculate_diffusion_auc, 73
calculate_effect_size, 74
calculate_graph_connectivity_sc, 75
calculate_information_content, 76
calculate_information_content(), *87*, *88*
calculate_integration_metrics_sc, 77
calculate_kbet_sc, 78
calculate_lisi_sc, 80
calculate_lisi_sc(), *67*
calculate_pca_adt_sc, 81
calculate_pca_bulk_dge, 82
calculate_pca_bulk_dge(), *225*
calculate_pca_sc, 83
calculate_pca_sc(), *133*, *181*, *660*
calculate_pcr_sc, 85
calculate_rpkm, 86
calculate_semantic_sim, 86
calculate_semantic_sim_mat, 87
calculate_semantic_sim_onto, 88
calculate_sparsity_stats, 89
calculate_tom, 90
calculate_tom(), *123*
calculate_tom_from_exp, 91
calculate_tpm, 93
calculate_wang_sim, 94
calculate_wang_sim_mat, 95
calculate_wang_sim_onto, 95
call_doublets_manual, 96
cell_cycle_genes, 97
cellphonedb_sc, 98
cellsweep_sc, 99
cellsweep_sc(), *142*, *190*
change_gene_identifier, 101
check_sc_state, 102
checkmate::makeAssertCollection(), *27*
community_detection, 103
compute_expression_info_sc, 104
consensus_nmf_bulk, 106
consensus_nmf_bulk(), *259*, *353*
consensus_nmf_sc, 107
consensus_nmf_sc(), *354*
constrained_page_rank, 109
constrained_page_rank_ls, 110
contrastive_pca, 112
contrastive_pca_processing, 113
contrastive_pca_processing(), *44*
cor_module_check_epsilon, 114
cor_module_check_epsilon(), *115*
cor_module_coremo_clustering, 115
cor_module_coremo_cor_sign, 116
cor_module_coremo_eigengene, 117
cor_module_coremo_stability, 118
cor_module_graph_check_res, 119
cor_module_graph_check_res(), *121*
cor_module_graph_final_modules, 120
cor_module_processing, 122
cor_module_processing(), *114*, *123*
cor_module_tom, 123

demo_single_cells, 124
densmap_sc, 125
densne_sc, 127
detect_adt_isotypes, 128
detect_raw_count_slot(), *314*, *490*, *661*
dgrdl_grid_search, 129
dgrdl_grid_search(), *130*
dgrdl_result, 130
dialogue_sc, 132
dialogue_sc(), *140*, *191*
diffcor_module_processing, 134
diffuse_seed_nodes, 135
diffuse_seed_nodes(), *229*
doublet_detection_boost_sc, 136
download_cd34_data, 137
download_cistarget_hg38, 138
download_demuxlet_pbmc, 139
download_dialogue_uc, 140
download_kang_pbmc, 141
download_marrow_cd34, 142
download_pbmc3k, 144
download_pbmc8k, 145
download_pbmc_1k_5p, 142
download_pbmc_batches, 143
download_pbmc_totalseq_data, 144

download_thymus_ageing, 146
drop_cols_sc, 147

extract_binary_heatmap_data, 147
extract_binary_heatmap_data(), *359*
extract_dot_plot_data, 149
extract_dot_plot_data(), *154*
extract_embedding_data, 150
extract_embedding_data(), *152, 153*, *156, 157*
extract_feature_pair, 151
extract_feature_plot_data, 152
extract_gene_expression, 153
extract_gene_expression(), *151–155*
extract_gene_violin_data, 154
extract_milo_plot_data, 155
extract_paga_plot_data, 156

f1_score_confusion_mat, 158
fast_cluster_sc, 159
fast_cluster_sc(), *337*
fast_ica_rust, 160
fast_ica_rust_helper, 161
fast_mnn_sc, 163
fast_mnn_sc(), *654*
fgsea_go_elim, 165
fgsea_simple_go_elim, 166
filter_similarities, 167
find_all_markers_sc, 168
find_all_markers_sc(), *99*
find_clusters_sc, 170
find_clusters_sc(), *653*
find_hvg_batch_aware_sc, 171
find_hvg_batch_aware_sc(), *163*, *631*, *633*, *654*
find_hvg_sc, 172
find_hvg_sc(), *181*, *241*, *657*
find_markers_sc, 174
find_neighbours_sc, 175
find_neighbours_sc(), *184*, *204*, *609, 610*, *659*, *680*
find_rbh_communities, 177
find_specific_markers_sc, 178
find_threshold_otsu, 180
fit_residuals_sc, 181
fit_residuals_sc(), *84*, *267*, *427*, *627*
fix_meta_data_column, 182
flag_cells, 183
forceatlas2_sc, 184

gene_set_proportions_sc, 186
GeneOntologyElim, 187
GeneOntologyElim(), *165*, *167*, *237*, *287*, *289*, *495*, *533*, *536, 537*
generate_bt_meta_cells_sc, 188
generate_cellsweep_test_data, 189
generate_cellsweep_test_data(), *447*
generate_dialogue_test_data, 190
generate_dialogue_test_data(), *451*
generate_gene_module_data, 192
generate_hotspot_membership, 193
generate_knn_sc, 193
generate_ligand_target_influence, 195
generate_personalisation_vec, 196
generate_rbh_graph, 197
generate_rbh_graph(), *267*
generate_sc_knn, 198
generate_seacells_sc, 199
generate_single_cell_test_data, 201
generate_single_cell_test_data(), *124*
generate_supercells_sc, 202
generate_wnn_graph_sc, 204
get_adt_feature_info, 206
get_adt_names, 207
get_adt_names(), *507*
get_adt_sample_info, 208
get_available_embeddings, 208
get_best_model, 209
get_best_model(), *308, 309*
get_best_run, 210
get_c_pca_factors, 211
get_c_pca_loadings, 212
get_cell_indices, 212
get_cell_info, 213
get_cell_info(), *348, 349*
get_cell_names, 214
get_cell_names(), *174*
get_cell_ranger_params, 215
get_cellphonedb_db, 216
get_cellphonedb_db(), *98*
get_cells_to_keep, 217
get_cells_to_keep(), *108*, *242*, *339*, *355, 356*, *628, 629*, *651*
get_centroids_sc, 217
get_cistarget_res, 218
get_cor_graph, 220
get_data, 221
get_data(), *18*

get_dge_effect_sizes, 222
get_dge_limma_voom, 223
get_dge_list, 224
get_dge_list(), *43*
get_dge_qc_plot, 224
get_diagnostics, 225
get_diffcor_graph, 226
get_differential_abundance_res, 227
get_differential_abundance_res(), *256*
get_diffusion_perms, 228
get_diffusion_vector, 229
get_embedding, 230
get_embedding(), *150*, *153*, *156*, *157*
get_epsilon_res, 231
get_factors, 231
get_feature_mat, 232
get_fpkm_counts, 233
get_gene_indices, 234
get_gene_lengths, 235
get_gene_lengths,bixverse::BulkCoExp-method
(get_gene_lengths), 235
get_gene_names, 235
get_gene_names_from_idx, 236
get_go_data_human, 237
get_go_data_human(), *187*
get_grid_search_res, 238
get_h, 238
get_h5ad_dimensions, 239
get_hotspot_membership, 240
get_hvg, 241
get_hvg(), *36*, *84*, *108*, *132*, *355*, *356*, *651*, *660*
get_hvg_data_sc, 241
get_ica_stability_res, 242
get_index_cells, 243
get_inflection_point, 244
get_influence, 245
get_kmeans_clusters, 245
get_knn_dist, 246
get_knn_mat, 247
get_knn_mat(), *246*
get_knn_obj, 248
get_magic, 249
get_magic(), *607*
get_marker_comparisons, 250
get_marker_summary, 250
get_marker_summary(), *250*
get_meta_cell_purity, 251
get_metadata, 253
get_miloR_abundances_sc, 254
get_miloR_abundances_sc(), *146*, *155*
get_model_fit, 256
get_modules, 256
get_nmf_gene_loadings, 257
get_nmf_modules, 258
get_nmf_sample_activity, 258
get_nmf_stability, 259
get_ontology_ancestry, 260
get_ontology_ancestry(), *76*, *87*, *88*
get_outputs, 261
get_params, 261
get_pca_factors, 263
get_pca_loadings, 264
get_pca_singular_val, 265
get_pseudobulked_sc, 266
get_pseudobulked_sc(), *496*
get_rbh_res, 267
get_residual_fit, 267
get_resolution_res, 268
get_results, 269
get_sc_available_features, 270
get_sc_cache_status, 271
get_sc_cache_status(), *102*
get_sc_counts, 272
get_sc_obs, 273
get_sc_var, 273
get_scores, 274
get_sim_matrix, 275
get_snf_adjcacency_mat, 276
get_snf_final_mat, 277
get_snf_params, 277
get_snn_graph, 278
get_stability, 279
get_symphony_hvg_names, 280
get_symphony_labels, 281
get_symphony_loadings, 282
get_symphony_z_corr, 283
get_tf_to_gene, 284
get_top_terms, 285
get_tpm_counts, 285
get_w, 286
gse_go_elim_method, 287
gse_go_elim_method_list, 288
gse_hypergeometric, 289
gse_hypergeometric_list, 291

harmony_sc, 292

harmony_sc(), *655*
harmony_v2_sc, 293
harmony_v2_sc(), *656*
hedges_g_dge, 294
hotspot_autocor_sc, 295
hotspot_gene_cor_sc, 297

ica_evaluate_comp, 299
ica_evaluate_comp(), *301*, *303*, *478*, *479*
ica_optimal_ncomp, 301
ica_optimal_ncomp(), *478*
ica_processing, 302
ica_processing(), *299*, *303*
ica_stabilised_results, 302
identify_tf_to_genes, 304
identify_tf_to_genes(), *669*
install_agent_skill, 306

knn_graph_label_propagation, 307

lda_k_sweep, 308
lda_k_sweep(), *210*, *605*
leiden_clustering, 310
leiden_clustering(), *170*, *653*
ligand_activity_scores, 311
ligand_activity_scores(), *26*
load_existing, 312
load_existing(), *25*
load_go_human_data, 313
load_go_human_data(), *237*
load_h5ad, 313
load_h5ad(), *140*, *661*
load_h5ad_norm, 315
load_mtx, 317
load_mtx(), *215*
load_multi_h5ad, 318
load_multi_mtx, 320
load_multi_mtx(), *490*
load_multi_tenx_h5, 322
load_multi_tenx_h5(), *492*, *503*
load_r_data, 323
load_sce, 325
load_sce(), *141*, *146*
load_seurat, 327
load_tenx_h5, 328

manifoldsR::densmap(), *125*
manifoldsR::densne(), *127*
manifoldsR::forceatlas2(), *184*
manifoldsR::forceatlas2_from_graph(), *184*
manifoldsR::params_densmap(), *125*, *126*
manifoldsR::params_densne(), *127*, *128*
manifoldsR::params_fa2(), *185*
manifoldsR::params_nn(), *126*, *128*, *185*, *474*, *679*, *681*
manifoldsR::params_phate(), *473*, *474*
manifoldsR::params_tsne(), *128*, *679*
manifoldsR::params_umap(), *126*, *681*
manifoldsR::phate(), *473*
manifoldsR::tsne(), *678*
manifoldsR::umap(), *680*
map_symphony_query, 330
meld_sc, 331
meld_sc(), *146*
merge_meta_cells, 333
merge_meta_cells(), *24*, *338–340*, *658*
merge_sc_experiments, 334
merge_subset_obs, 336
meta_cells_per_group, 337
meta_cells_per_group(), *333*
MetaCells, 339
MetaCells(), *189*, *201*, *203*, *333*, *337*, *338*, *658*
module_scores_sc, 340
module_scores_sc(), *97*
modules_from_loadings, 341

nebula_mc, 342
nebula_sc, 344
nebula_sc(), *141*, *342*, *343*
NetworkDiffusions, 347
NetworkDiffusions(), *73*, *103*, *135*, *228*, *229*, *472*, *674*
new_adt_counts_clr, 348
new_adt_counts_clr(), *16*
new_adt_counts_dsb, 349
new_adt_counts_dsb(), *16*
new_sc_knn, 350
nmf_bulk, 351
nmf_k_sweep_bulk, 353
nmf_k_sweep_bulk(), *107*
nmf_k_sweep_sc, 354
nmf_k_sweep_sc(), *108*
nmf_sc, 356
normalise_bulk_dge, 357
normalise_bulk_dge(), *224*, *225*, *488*

OntologySim, 358
OntologySim(), *70*, *89*, *96*, *168*, *275*, *486*
ot_harmonic_score, 359

params_binary_heatmap, 359
params_binary_heatmap(), *148*
params_blitzgsea, 361
params_blitzgsea(), *35*, *45*, *514*, *515*
params_boost, 362
params_boost(), *136*
params_bpca, 364
params_bpca(), *597*
params_bulk_sparsity, 365
params_bulk_sparsity(), *637*
params_cistarget, 366
params_cistarget(), *600*, *672*
params_community_detection, 368
params_community_detection(), *103*
params_cor_graph, 369
params_cor_graph(), *119*, *121*
params_coremo, 370
params_coremo(), *115*
params_dgrdl, 371
params_dgrdl(), *129*, *131*
params_dialogue_hlm, 372
params_dialogue_hlm(), *132*, *554*
params_dialogue_pmd, 373
params_dialogue_pmd(), *132*, *133*, *554*
params_dialogue_refine, 376
params_dialogue_refine(), *132*, *554*
params_edger_ql, 377
params_edger_ql(), *496*, *528*, *601*, *602*
params_fast_cluster_default, 378
params_fast_cluster_default(), *363*, *364*, *457*
params_graph_resolution, 379
params_graph_resolution(), *119*, *177*
params_gsea, 380
params_gsea(), *48*, *50*, *51*, *165*, *167*, *533*
params_gsva, 381
params_gsva(), *53*
params_hvg_defaults, 382
params_hvg_defaults(), *363*, *364*, *463*
params_hvg_scran_defaults, 382
params_hvg_scran_defaults(), *171*, *173*, *382*, *427*, *428*, *558*, *657*
params_ica_general, 383
params_ica_general(), *161*, *162*, *300*, *303*
params_ica_ncomp, 384
params_ica_ncomp(), *300*
params_ica_randomisation, 384
params_ica_randomisation(), *299*, *303*
params_kmeans_defaults, 385
params_kmeans_defaults(), *422–425*
params_knn_defaults, 386
params_knn_defaults(), *32*, *164*, *176*, *188*, *195*, *199*, *200*, *203*, *255*, *296*, *298*, *332*, *363*, *364*, *393*, *394*, *405*, *406*, *409*, *410*, *417–419*, *426*, *432*, *434–437*, *441–446*, *453–457*, *463*, *632*, *634*, *653*, *654*, *659*, *688*
params_label_propagation, 387
params_label_propagation(), *308*
params_lda, 388
params_lda(), *309*, *605*
params_ligand_target, 390
params_ligand_target(), *196*
params_limma_voom, 391
params_limma_voom(), *72*, *550*, *607*
params_meld, 393
params_meld(), *332*
params_module_membership, 394
params_module_membership(), *106*, *131*, *303*, *342*, *351*, *352*, *650*
params_nebula, 396
params_nebula(), *343*, *345*, *564*
params_nmf_consensus, 398
params_nmf_consensus(), *106*, *108*, *109*, *353*, *355*
params_nmf_hals, 399
params_nmf_hals(), *106*, *108*, *352*, *353*, *355*, *356*, *650*, *651*
params_norm_doublets_defaults, 400
params_norm_doublets_defaults(), *363*, *456*, *457*, *463*
params_pca_defaults, 401
params_pca_defaults(), *363*, *364*, *456*, *457*, *463*
params_ppca, 401
params_ppca(), *612*
params_sc_apr, 402
params_sc_apr(), *182*
params_sc_aucell, 403
params_sc_aucell(), *29*, *33*, *553*
params_sc_bbknn, 405
params_sc_bbknn(), *32*, *652*

params_sc_bonsai, 406
params_sc_bonsai(), *36*
params_sc_branch_selection, 408
params_sc_branch_selection(), *532*, *603*
params_sc_bt_metacells, 409
params_sc_bt_metacells(), *188*
params_sc_cellphonedb, 410
params_sc_cellphonedb(), *99*
params_sc_cellsweep, 411
params_sc_cellsweep(), *100*
params_sc_dsb, 414
params_sc_dsb(), *349*
params_sc_empty_droplets, 415
params_sc_empty_droplets(), *100*
params_sc_fast_cluster, 416
params_sc_fast_cluster(), *159*
params_sc_fastmnn, 418
params_sc_fastmnn(), *163*, *654*
params_sc_gene_trends, 419
params_sc_gene_trends(), *532*, *603*, *604*
params_sc_harmony, 421
params_sc_harmony(), *40*, *292*, *655*
params_sc_harmony_v2, 423
params_sc_harmony_v2(), *40*, *294*, *656*
params_sc_hotspot, 425
params_sc_hotspot(), *296*, *298*
params_sc_hvg, 427
params_sc_hvg(), *171*, *173*, *181*, *242*, *382*, *657*
params_sc_knn, 428
params_sc_knn(), *677*
params_sc_magic, 430
params_sc_magic(), *608*
params_sc_miloR, 431
params_sc_miloR(), *254*
params_sc_min_quality, 432
params_sc_min_quality(), *100*, *190*, *314*, *316*, *317*, *319*, *320*, *322*, *324*, *326*, *327*, *329*, *335*, *539*, *661*
params_sc_mtx_io, 433
params_sc_mtx_io(), *215*, *317*
params_sc_neighbours, 434
params_sc_neighbours(), *176*, *194*, *195*, *199*, *579*, *580*, *659*
params_sc_palantir, 435
params_sc_palantir(), *610*, *611*
params_sc_pca, 437
params_sc_pca(), *40*, *83*, *84*, *164*, *401*, *419*, *443–445*, *559*, *632*, *634*, *654*, *660*
params_sc_sctransform, 439
params_sc_sctransform(), *181*, *182*
params_sc_seacells, 440
params_sc_seacells(), *200*
params_sc_seurat_cca, 442
params_sc_seurat_cca(), *631*
params_sc_seurat_rpca, 444
params_sc_seurat_rpca(), *633*
params_sc_supercell, 445
params_sc_supercell(), *203*
params_sc_synthetic_cellsweep, 447
params_sc_synthetic_cellsweep(), *190*
params_sc_synthetic_data, 448
params_sc_synthetic_data(), *124*, *202*
params_sc_synthetic_data_adt, 450
params_sc_synthetic_dialogue, 451
params_sc_synthetic_dialogue(), *191*
params_sc_vision, 452
params_sc_vision(), *688*
params_sc_wnn, 453
params_sc_wnn(), *204*, *205*
params_scdblfinder, 455
params_scdblfinder(), *617*
params_scenic, 458
params_scenic(), *560*, *619*, *620*
params_scenic_binarise, 459
params_scenic_binarise(), *34*
params_scenic_extra_trees_defaults, 460
params_scenic_extra_trees_defaults(), *459*
params_scenic_gradient_boosting_defaults, 461
params_scenic_gradient_boosting_defaults(), *459*
params_scenic_random_forest_defaults, 462
params_scenic_random_forest_defaults(), *459*
params_scrublet, 462
params_scrublet(), *622*
params_sctype_cells, 464
params_sctype_cells(), *28*
params_snf, 465
params_snf(), *19*, *613*, *635*

params_ssgsea, 466
params_ssgsea(), *65*
params_symphony_map, 467
params_symphony_map(), *330*
params_synthetic_bulk_rnaseq, 468
params_synthetic_bulk_rnaseq(), *664*
per_cell_qc_outlier, 471
permute_seed_nodes, 472
permute_seed_nodes(), *228*
phate_sc, 473
plot.LdaKSweepResult, 474
plot.NmfKSweepResult, 475
plot.ScrubletRes(), *96*
plot_boxplot_normalization, 476
plot_epsilon_res, 477
plot_hvgs, 477
plot_ica_ncomp_params, 478
plot_ica_stability_individual, 479
plot_optimal_cuts, 480
plot_pca, 480
plot_pca_res, 481
plot_preprocessing_genes, 482
plot_preprocessing_outliers, 483
plot_rbf_impact, 484
plot_resolution_res, 484
plot_voom_normalization, 485
pre_process_sim_onto, 486
prepare_cell_markers, 487
prepare_cell_markers(), *60*
preprocess_bulk_coexp, 487
preprocess_bulk_coexp(), *106*, *122*, *129*, *131*, *134*, *302*, *352*, *477*
preprocess_bulk_dge, 488
prescan_h5ad_files, 489
prescan_h5ad_files(), *318*, *319*
prescan_mtx_dirs, 490
prescan_mtx_dirs(), *320*
prescan_tenx_h5_files, 492
prescan_tenx_h5_files(), *322*
prioritise_interactions, 493
process_go_data, 495
process_go_data(), *237*
pseudobulk_dge_sc, 496

qc_bulk_dge, 497
qc_bulk_dge(), *224*, *225*, *357*, *391*, *488*

RbhGraph, 498
RbhGraph(), *177*, *197*, *267*
read_h5ad_metadata, 500
read_h5ad_x_summary, 500
read_motif_annotation_file, 501
read_motif_annotation_file(), *600*
read_motif_ranking, 502
read_motif_ranking(), *600*
read_multi_tenx_h5_adt, 503
read_tenx_h5_adt, 504
read_tenx_h5_adt(), *503*
read_tenx_h5_metadata, 505
relayout_bonsai, 506
relayout_bonsai(), *407*, *408*
remove_adt_isotypes, 507
remove_magic, 507
remove_residual_fit, 508
remove_samples, 509
rescue_cells, 510
reset_cells_to_keep, 510
robust_scale, 511
rs_2d_loess, 512
rs_batch_silhouette_width, 513
rs_blitzgsea_calibrate, 513
rs_blitzgsea_calibrate(), *515*
rs_blitzgsea_score, 514
rs_calc_norm_factors, 515
rs_cell_type_asw, 516
rs_cistarget, 517
rs_compare_knn, 518
rs_constrained_page_rank, 519
rs_constrained_page_rank_list, 520
rs_contrastive_pca, 521
rs_cor, 522
rs_cor2, 523
rs_cor_upper_triangle, 522
rs_cos, 523
rs_count_zeroes, 524
rs_cov2cor, 524
rs_covariance, 525
rs_cpm, 525
rs_dense_to_upper_triangle, 526
rs_dialogue_sc(), *553*, *554*
rs_differential_cor, 526
rs_dist, 527
rs_edger_ql, 527
rs_fast_auc, 528
rs_fast_ica, 529
rs_fdr_adjustment, 530
rs_filter_by_expr, 530

rs_gene_trends, 531
rs_generate_ligand_target_influence, 532
rs_geom_elim_fgsea_simple, 533
rs_gower_dist, 534
rs_graph_connectivity, 535
rs_gse_geom_elim, 536
rs_gse_geom_elim_list, 537
rs_gsva, 538
rs_h5ad_data, 539
rs_hamming_dist, 540
rs_harmony, 540
rs_harmony_v2, 541
rs_hedges_g, 541
rs_hypergeom_test, 542
rs_hypergeom_test_list, 543
rs_ica_iters, 544
rs_ica_iters_cv, 545
rs_jaccard_row_integers, 546
rs_kbet, 546
rs_knn_label_propagation, 547
rs_knn_mat_to_edge_list, 548
rs_leiden_clustering, 549
rs_ligand_activity_scores, 549
rs_limma_voom, 550
rs_lisi, 551
rs_mad_outlier, 552
rs_mc_aucell, 553
rs_mc_dialogue, 553
rs_mc_hotspot_autocor, 555
rs_mc_hotspot_gene_cor, 556
rs_mc_hvg, 557
rs_mc_pca, 559
rs_mc_scenic, 560
rs_mc_vision, 560
rs_mc_vision_with_autocorrelation, 561
rs_mitch_calc, 562
rs_mutual_info, 563
rs_nebula_mc, 564
rs_nebula_sc(), *564*
rs_onto_semantic_sim, 565
rs_onto_semantic_sim_mat, 565
rs_onto_sim_wang, 566
rs_onto_sim_wang_mat, 567
rs_ot_harmonic_sum, 567
rs_page_rank, 568
rs_page_rank_parallel, 569
rs_pcr, 570
rs_phyper, 570
rs_pointwise_mutual_info, 571
rs_prcomp, 572
rs_prepare_gsva_gs(), *538*, *583*, *591*
rs_prepare_whitening, 572
rs_prepare_whitening(), *162*
rs_random_svd, 573
rs_range_norm, 574
rs_rank_matrix_col, 574
rs_rank_matrix_col_stable, 575
rs_rank_matrix_col_stable(), *585*
rs_rbf_function, 575
rs_rbf_function_mat, 576
rs_rbh_cor, 576
rs_rbh_sets, 577
rs_remove_batch_effect, 578
rs_sc_knn, 579
rs_sc_knn_w_dist, 580
rs_sc_snn, 580
rs_set_similarity, 581
rs_set_similarity_list, 582
rs_set_similarity_list2, 582
rs_singscore_multi, 583
rs_singscore_permutation_test, 584
rs_singscore_single, 585
rs_singscore_single(), *584*
rs_snf, 586
rs_sparse_dict_dgrdl, 586
rs_sparse_dict_dgrdl_grid_search, 587
rs_spatial_fdr, 589
rs_spectral_clustering, 589
rs_spectral_clustering_sim, 590
rs_ssgsea, 591
rs_tied_diffusion_parallel, 591
rs_tom, 593
rs_upper_triangle_to_dense, 593
rs_upper_triangle_to_sparse, 594
rs_voom_normalise, 595
rs_wnn, 596
run_bpca, 597
run_bpca(), *364*
run_cell_qc, 598
run_cell_qc_fixed, 599
run_cistarget, 600
run_cistarget(), *671*
run_edger_ql, 601
run_edger_ql(), *496*, *497*

run_gene_trends_sc, 603
run_gene_trends_sc(), *408*, *419*
run_lda, 605
run_lda(), *388*
run_limma_voom, 606
run_magic_sc, 607
run_magic_sc(), *153*, *154*, *249*, *430*, *603*
run_paga_sc, 609
run_paga_sc(), *156*, *157*
run_palantir_sc, 610
run_palantir_sc(), *435*, *603*
run_ppca, 612
run_ppca(), *401*
run_snf, 613

save_h5ad, 614
save_sc_exp_to_disk, 615
save_sc_exp_to_disk(), *25*
sc_knn_to_nearest_neighbours, 616
sc_pipeline, 616
scdblfinder_sc, 617
scenic_gene_filter_sc, 618
scenic_gene_filter_sc(), *620*
scenic_grn_sc, 619
score_clusters, 621
score_clusters(), *27*
scrublet_sc, 622
sct_corrected_counts_sc, 624
sct_corrected_counts_sc(), *181*
set_bonsai_embedding, 625
set_bonsai_embedding(), *36*
set_cells_to_keep(), *214*
set_magic, 626
set_residual_fit, 627
set_sc_new_obs_col, 628
set_sc_new_obs_col_multiple, 628
set_sc_new_var_cols, 629
setnames_sc, 630
seurat_cca_sc, 631
seurat_cca_sc(), *633*
seurat_rpca_sc, 633
SimilarityNetworkFusion, 634
SimilarityNetworkFusion(), *19*, *276–278*, *613*
simplify_hypergeom_res, 636
simulate_dropouts, 637
simulate_dropouts(), *89*, *365*, *664*
SingleCellCountData, 638
SingleCells, 645
SingleCells(), *340*, *648*
SingleCellsMultiModal, 647
SingleCellsMultiModal(), *325*, *646*
SingleCellsSubset, 648
SingleCellsSubset(), *133*, *336*
stabilised_nmf_bulk, 649
stabilised_nmf_bulk(), *106*, *259*
stabilised_nmf_sc, 650
stats::model.matrix(), *602*
step_bbknn_sc, 652
step_clusters_sc, 653
step_fast_mnn_sc, 654
step_harmony_sc, 655
step_harmony_v2_sc, 656
step_hvg_sc, 657
step_metacells_sc, 658
step_metacells_sc(), *24*, *685*
step_neighbours_sc, 659
step_pca_sc, 660
stream_h5ad, 661
SymphonyReference, *40*, 662
synthetic_bulk_cor_matrix, 664
synthetic_bulk_cor_matrix(), *468*
synthetic_c_pca_data, 665
synthetic_signal_matrix, 666

test_nhoods, 667
test_nhoods(), *156*
tf_to_genes_correlations, 669
tf_to_genes_correlations(), *38*
tf_to_genes_motif_enrichment, 671
tf_to_genes_motif_enrichment(), *37*, *38*
tied_diffusion, 673
tied_diffusion(), *229*
to_snake_case, 675
top_genes_perc_sc, 675
transfer_labels_symphony, 676
transfer_labels_symphony(), *20*, *40*
tsne_sc, 678
tsne_sc(), *127*

umap_sc, 680
umap_sc(), *125*
update_metadata_values, 681
upper_triangle_to_sparse, 682
upper_triangular_sym_mat, 683

validate_pipeline, 685

validate_pipeline(), *23*, *658*
vision_sc, 686
vision_w_autocor_sc, 687

write_cellranger_output, 689
write_cellranger_output(), *215*
write_h5ad_sc, 690
write_h5ad_sc_dense, 691