Package: randomwalk (via r-universe)

May 17, 2026

Type Package

Title Asynchronous Pixel Walking Simulation with Parallel Processing

Version 2.1.2

Description Implements parallel random walk simulations that create fractal graphs through asynchronous pixel walking on a grid. Features include parallel processing with crew workers (chunked mode recommended), comprehensive statistics tracking, and an optional Shiny interface for interactive visualization. Core simulation functions can be used programmatically without the GUI.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Depends R (>= 4.1.0)

Imports logger, ggplot2, RColorBrewer, viridisLite, rlang

Suggests nanonext, crew, mirai, munsell, dplyr, shiny, shinylive, shinytest2, targets, tarchetypes, visNetwork, testthat (>= 3.0.0), devtools, usethis, pkgdown, covr, bench, knitr, rmarkdown, quarto

Config/testthat/edition 3

VignetteBuilder knitr, quarto

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

Date/Publication 2026-05-17 17:01:51 UTC

RemoteUrl https://github.com/JohnGavin/randomwalk

RemoteRef HEAD

RemoteSha 72c793155a3bbdd752a4a32595c1f84a6b3c7502

Contents

broadcast_black_pixel *Broadcast Black Pixel Update*

Description

Serializes and broadcasts a black pixel update to all subscribers

Usage

```
broadcast_black_pixel(pub_socket, position, walker_id)
```

Arguments

pub_socket	Publisher socket
position	Vector c(x, y) of black pixel position
walker_id	ID of walker that created the black pixel

broadcast_update *Broadcast Grid Update to Workers*

Description

Sends a grid state update message to all subscribed workers via the nanonext publisher socket.

Usage

```
broadcast_update(socket, position, version)
```

Arguments

socket	A nanonext publisher socket created by `create_pub_socket()`.
position	Integer vector of length 2. Grid coordinates `c(row, col)` of the newly black pixel.
version	Integer. Current version number of the grid state. Workers use this to detect stale caches.

Details

Broadcasts a pixel update message containing:

- `type`: Always "pixel_update"
- `position`: Coordinates of new black pixel
- `version`: Grid state version number

Workers listening on the subscriber socket will receive this message and update their local caches accordingly.

This is a non-blocking operation - the function returns immediately after queuing the message for transmission.

Value

NULL (invisibly). Message is sent asynchronously.

See Also

create_pub_socket

Examples

```
## Not run:
# Broadcast update when walker terminates
broadcast_update(
  socket = pub_socket,
  position = c(10, 15),
  version = 42
)

## End(Not run)
```

---

check_termination *Check Walker Termination Conditions*

Description

Checks if the walker should terminate based on simulation rules.

Usage

```
check_termination(
  walker,
  grid,
  neighborhood = c("4-hood", "8-hood"),
  boundary = c("terminate", "wrap"),
  max_steps = 10000L
)
```

Arguments

walker	List. Walker object.
grid	Numeric matrix. The simulation grid.
neighborhood	Character. "4-hood" or "8-hood".
boundary	Character. "terminate" or "wrap".
max_steps	Integer. Maximum steps before forced termination. Default 10000.

Value

Modified walker object with updated active status.

`cleanup_async` *Clean Up Async Resources*

Description

Shuts down crew workers and closes nanonext sockets. Should always be called when async simulation completes or errors.

Usage

```
cleanup_async(controller, socket)
```

Arguments

`controller`	A crew controller object created by `create_controller()`. Can be NULL if controller was not successfully created.
`socket`	A nanonext publisher socket created by `create_pub_socket()`. Can be NULL if socket was not successfully created.

Details

Performs graceful shutdown:

1. Terminates all crew workers (sends shutdown signal)
2. Closes nanonext publisher socket
3. Logs cleanup status

This function is safe to call multiple times and handles NULL inputs gracefully (useful for error cleanup).

Always call this function in a `tryCatch()` finally block to ensure resources are cleaned up even if the simulation errors.

Value

NULL (invisibly). Side effect: workers stopped, sockets closed.

See Also

`create_controller`, `create_pub_socket`

Examples

```
## Not run:
# Typical usage pattern
controller <- NULL
socket <- NULL

tryCatch({
```

```
  controller <- create_controller(n_workers = 2)
  socket <- create_pub_socket(port = 5555)

  # ... run simulation ...

}, finally = {
  cleanup_async(controller, socket)
})

## End(Not run)
```

`close_sockets` *Close Sockets*

Description

Properly closes nanonext sockets

Usage

```
close_sockets(...)
```

Arguments

`...`	Socket objects to close

`count_black_pixels` *Count Black Pixels in Grid*

Description

Count Black Pixels in Grid

Usage

```
count_black_pixels(grid)
```

Arguments

`grid`	Numeric matrix. The simulation grid.

Value

Integer. Number of black pixels.

Examples

```
grid <- initialize_grid(10)
count_black_pixels(grid)  # 1 (only center)
```

---

create_controller *Create Async Controller (Auto-detect Backend)*

Description

Creates an async controller for parallel random walk simulation. Automatically selects the appropriate backend:

- WebR/WebAssembly: mirai (crew not available)
- Native R: crew (preferred for better abstractions)

Usage

```
create_controller(n_workers = 2, seconds_idle = 60)
```

Arguments

`n_workers`	Integer. Number of parallel workers to create (default: 2). Recommended: 2-4 for medium grids, 4-8 for large grids.
`seconds_idle`	Numeric. Seconds of idle time before worker shutdown (default: 60). Note: Only applies to crew backend (ignored for mirai).

Details

The controller manages a pool of R worker processes that execute walker step functions in parallel. Each worker maintains its own local cache of the grid state and subscribes to updates from the main process.

Backend Selection:

- WebR/WebAssembly detected by `is_webr()` → uses mirai
- Native R → uses crew

Both backends provide the same interface:

- `push()`: Submit task
- `pop()`: Retrieve completed task
- `terminate()`: Shutdown workers

Workers are automatically started when the controller is created and can be cleanly shut down using `cleanup_async()`.

Value

An async controller object with initialized workers.

- Crew backend: R6 crew controller object
- Mirai backend: List with crew-compatible interface

See Also

`create_pub_socket`, `cleanup_async`, `create_mirai_controller`

Examples

```
## Not run:
# Create controller (auto-detects environment)
controller <- create_controller(n_workers = 2)

# Use controller for parallel tasks
# ... simulation code ...

# Clean up when done
cleanup_async(controller, socket)

## End(Not run)
```

---

create_pub_socket *Create Nanonext Publisher Socket*

Description

Creates a nanonext publisher socket for broadcasting grid state updates to all worker processes.

Usage

```
create_pub_socket(port = 5555)
```

Arguments

port	Integer. TCP port for the publisher socket (default: 5555). Must be available and not blocked by firewall.

Details

The publisher socket uses TCP on localhost to broadcast grid updates. Workers subscribe to this socket to receive notifications when pixels are added to the black set (walker termination events).

Communication pattern:

- Main process publishes updates via this socket

- Worker processes subscribe (see `worker_init()`)
- Non-blocking: workers poll for updates between steps

The socket binds to `tcp://127.0.0.1:<port>` and allows multiple subscribers to connect.

Value

A nanonext socket object configured for publishing.

See Also

`broadcast_update`, `cleanup_async`

Examples

```
## Not run:
# Create publisher socket
pub_socket <- create_pub_socket(port = 5555)

# Broadcast an update
broadcast_update(pub_socket, position = c(10, 15), version = 42)

# Clean up
nanonext::close(pub_socket)

## End(Not run)
```

---

create_walker *Create a Walker*

Description

Creates a walker object for the random walk simulation.

Usage

```
create_walker(id, pos, grid_size, store_path = TRUE)
```

Arguments

`id`	Integer. Unique identifier for the walker.
`pos`	Integer vector of length 2 (row, col). Starting position.
`grid_size`	Integer. Size of the grid.
`store_path`	Logical. If TRUE, stores full path history. Default TRUE. Set to FALSE for performance when path visualization not needed.

Value

A list representing the walker with components:

id Walker identifier

pos Current position

steps Number of steps taken

active Logical indicating if walker is still active

termination_reason Character string if terminated, NULL otherwise

path List of all positions visited (NULL if store_path=FALSE)

store_path Logical indicating if path is being stored

Examples

```
walker <- create_walker(1, c(5, 5), 10)
walker_no_path <- create_walker(2, c(3, 3), 10, store_path = FALSE)
```

---

find_isolated_pixels *Find Isolated Pixels in Grid*

Description

Scans the grid to find black pixels that have no black neighbors, which violates the connectivity requirement for DLA simulations.

Usage

```
find_isolated_pixels(grid, neighborhood = "4-hood")
```

Arguments

grid	Numeric matrix representing the grid state (0 = white, 1 = black)
neighborhood	Character string specifying neighborhood type: "4-hood" (orthogonal) or "8-hood" (includes diagonals)

Details

This function is useful for:

- Debugging WebR/Shiny reactive timing issues
- Validating grid state after async simulations
- Testing grid connectivity requirements

A pixel is considered isolated if:

- It is black (value = 1)
- None of its neighbors (in the specified neighborhood) are black

Note: The center pixel is never considered isolated (it's the seed).

Value

List of isolated pixel positions (each element is a 2-element numeric vector with row and column indices). Returns empty list if no isolated pixels found.

See Also

validate_termination_position, validate_no_isolated_pixels

Examples

```
grid <- initialize_grid(10)
grid[3, 3] <- 1  # Add isolated pixel far from center
isolated <- find_isolated_pixels(grid, "4-hood")
length(isolated)  # Should be 1
```

---

format_statistics *Format Simulation Statistics for Display*

Description

Formats simulation statistics into a readable character vector.

Usage

```
format_statistics(stats)
```

Arguments

stats	List. Statistics from run_simulation().

Value

Character vector with formatted statistics.

generate_walker_positions *Generate Random Starting Positions for Walkers*

Description

Creates random starting positions for walkers, avoiding the center pixel and any black pixels.

Usage

```
generate_walker_positions(n_walkers, grid, avoid_black = TRUE)
```

Arguments

n_walkers	Integer. Number of walkers to create.
grid	Numeric matrix. The simulation grid.
avoid_black	Logical. If TRUE, avoids placing walkers on black pixels.

Value

A list of integer vectors, each of length 2 (row, col).

Examples

```
grid <- initialize_grid(10)
positions <- generate_walker_positions(5, grid)
```

get_black_percentage *Get Percentage of Black Pixels*

Description

Get Percentage of Black Pixels

Usage

```
get_black_percentage(grid)
```

Arguments

grid	Numeric matrix. The simulation grid.

Value

Numeric. Percentage of black pixels (0-100).

Examples

```
grid <- initialize_grid(10)
get_black_percentage(grid)  # 1% for 10x10 grid
```

---

get_neighbors *Get Neighboring Positions*

Description

Returns all valid neighbor positions for a given position.

Usage

```
get_neighbors(pos, neighborhood = c("4-hood", "8-hood"))
```

Arguments

pos	Integer vector of length 2 (row, col).
neighborhood	Character. Either "4-hood" (NSEW) or "8-hood" (includes diagonals).

Value

A list of integer vectors, each of length 2.

Examples

```
get_neighbors(c(5, 5), "4-hood")
get_neighbors(c(5, 5), "8-hood")
```

---

get_neighbors_bounded *Get Neighbor Positions (Bounded)*

Description

Returns list of neighbor positions based on neighborhood type, filtering out positions that are out of bounds.

Usage

```
get_neighbors_bounded(position, grid_size, neighborhood)
```

Arguments

position	Vector c(x, y) current position
grid_size	Integer grid dimensions
neighborhood	Character "4-hood" or "8-hood"

Value

List of neighbor positions (within bounds only)

---

get_pixel *Get Pixel Value from Grid*

Description

Retrieves the value at a given position, handling boundary conditions.

Usage

```
get_pixel(grid, pos, boundary = "terminate")
```

Arguments

grid	Numeric matrix. The simulation grid.
pos	Integer vector of length 2 (row, col).
boundary	Character. Either "terminate" or "wrap". Default is "terminate".

Value

Integer. The pixel value (0 or 1), or NA if out of bounds with "terminate" boundary.

Examples

```
grid <- initialize_grid(10)
get_pixel(grid, c(5, 5))  # 1 (center is black)
get_pixel(grid, c(0, 5))  # NA (out of bounds with terminate)
get_pixel(grid, c(0, 5), boundary = "wrap")  # Value from wrapped position
```

has_black_neighbor *Check if Walker Has Black Neighbor*

Description

Checks if any of the walker's neighbors are black pixels.

Usage

```
has_black_neighbor(
  walker,
  grid,
  neighborhood = c("4-hood", "8-hood"),
  boundary = c("terminate", "wrap")
)
```

Arguments

walker	List. Walker object.
grid	Numeric matrix. The simulation grid.
neighborhood	Character. "4-hood" or "8-hood".
boundary	Character. Boundary condition.

Value

Logical. TRUE if walker has at least one black neighbor.

init_publisher_socket *Initialize Publisher Socket*

Description

Creates a nanonext publisher socket for broadcasting black pixel updates

Usage

```
init_publisher_socket(port = 5555)
```

Arguments

port	Port number for socket (default 5555)

Value

Publisher socket object

init_subscriber_socket *Initialize Subscriber Socket*

Description

Creates a nanonext subscriber socket for receiving black pixel updates

Usage

```
init_subscriber_socket(host = "localhost", port = 5555)
```

Arguments

host	Host address (default "localhost")
port	Port number (default 5555)

Value

Subscriber socket object

initialize_grid *Initialize a Grid*

Description

Creates an n x n grid for the random walk simulation. By default, the center pixel is set to black (1), and all other pixels are white (0).

Usage

```
initialize_grid(n, center_black = TRUE)
```

Arguments

n	Integer. The size of the grid (n x n). Must be >= 3.
center_black	Logical. If TRUE, initializes the center pixel as black. Default is TRUE.

Value

A numeric matrix of size n x n with 0 (white) and 1 (black) values.

Examples

```
grid <- initialize_grid(10)
grid[5, 5]  # Center pixel should be 1 (black)
```

is_within_bounds *Check if a Position is Within Grid Bounds*

Description

Check if a Position is Within Grid Bounds

Usage

```
is_within_bounds(pos, n)
```

Arguments

pos	Integer vector of length 2 (row, col).
n	Integer. Grid size.

Value

Logical. TRUE if position is within bounds, FALSE otherwise.

Examples

```
is_within_bounds(c(5, 5), 10)  # TRUE
is_within_bounds(c(0, 5), 10)  # FALSE
is_within_bounds(c(11, 5), 10) # FALSE
```

plot_grid *Plot Final Grid State*

Description

Visualizes the final state of the simulation grid, showing black pixels that formed during the random walk simulation. Returns a ggplot2 object for display via targets pipeline.

Usage

```
plot_grid(
  result,
  main = NULL,
  col_palette = c("white", "black"),
  show_legend = FALSE
)

## S3 method for class 'randomwalk_result'
plot(x, ...)
```

Arguments

result	A simulation result object returned by run_simulation
main	Character string for the plot title. If NULL (default), generates an informative title with statistics: black pixels, collisions, walkers, time.
col_palette	A vector of two colors for white (0) and black (1) pixels. Default is c("white", "black")
show_legend	Logical. If FALSE, hides the legend. Default is FALSE to save space.
x	A simulation result object (same as result)
...	Additional arguments passed to plot_grid

Value

A ggplot2 object that can be displayed or saved

Examples

```
## Not run:
result <- run_simulation(grid_size = 20, n_walkers = 8)
p <- plot_grid(result)
print(p)  # Display the plot with auto-generated title

# Custom title
p <- plot_grid(result, main = "My Custom Title")

# With legend
p <- plot_grid(result, show_legend = TRUE)

## End(Not run)
```

---

plot_grid_enhanced *Enhanced Grid Plot with Arrival Time Coloring*

Description

Visualizes the final state of the simulation grid with pixels colored by their arrival time (termination order). Earlier arrivals are shown in darker colors, later arrivals in lighter colors.

Usage

```
plot_grid_enhanced(
  result,
  main = NULL,
  quantiles = 5,
  color_scheme = "viridis"
)
```

Arguments

result	A simulation result object from run_simulation()
main	Custom title. If NULL, auto-generates with pixel statistics
quantiles	Number of quantiles for color grouping (default 5)
color_scheme	Color palette name: "viridis", "plasma", "blues", "heat"

Value

A ggplot2 object

---

plot_simulation *Plot Simulation Results*

Description

Creates a combined visualization showing both the final grid state and walker paths in a side-by-side layout.

Usage

```
plot_simulation(result, ...)
```

Arguments

result	A simulation result object returned by run_simulation
...	Additional arguments passed to plot_grid and plot_walker_paths

Value

Invisibly returns the previous graphical parameters (from par()).

Examples

```
## Not run:
result <- run_simulation(grid_size = 20, n_walkers = 8)
plot_simulation(result)

## End(Not run)
```

`plot_walker_paths` *Plot Walker Paths*

Description

Visualizes the paths taken by all walkers during the simulation, showing their starting positions, trajectories, and termination points.

Usage

```
plot_walker_paths(
  result,
  main = "Walker Paths and Final Positions",
  colors = NULL,
  add_grid = TRUE,
  grid_col = "lightgray",
  lwd = 1.5,
  cex_start = 1.5,
  cex_end = 2,
  legend = TRUE,
  legend_pos = "topright"
)
```

Arguments

`result`	A simulation result object returned by `run_simulation`
`main`	Character string for the plot title. Default is "Walker Paths and Final Positions"
`colors`	Optional vector of colors for walker paths. If NULL (default), uses rainbow colors
`add_grid`	Logical indicating whether to add grid lines. Default is TRUE
`grid_col`	Color for grid lines. Default is "lightgray"
`lwd`	Line width for paths. Default is 1.5
`cex_start`	Size of starting position markers. Default is 1.5
`cex_end`	Size of ending position markers. Default is 2
`legend`	Logical indicating whether to add a legend. Default is TRUE
`legend_pos`	Position of legend. Default is "topright"

Details

Walker paths are shown in different colors. Starting positions are marked with circles (pch 19). Ending positions use different markers:

- Square (pch 15): Terminated due to black pixel (black_neighbor or touched_black)
- Triangle (pch 17): Terminated at boundary (hit_boundary)
- X (pch 4): Terminated at max steps (max_steps)

Value

Invisibly returns NULL. Called for side effect of creating a plot.

Examples

```
## Not run:
result <- run_simulation(grid_size = 20, n_walkers = 8)
plot_walker_paths(result)

## End(Not run)
```

---

print_simulation_result *Print Simulation Result*

Description

Print Simulation Result

Usage

```
print_simulation_result(result)
```

Arguments

result	List. Result from run_simulation().

---

run_dashboard *Run Shiny Dashboard App*

Description

Launches the interactive Shiny dashboard for random walk simulations. This is a convenience wrapper that combines the input and output modules into a complete application.

Usage

```
run_dashboard(...)
```

Arguments

...	Additional arguments passed to shiny::shinyApp().

Value

A Shiny app object.

Examples

```
## Not run:
# Launch the dashboard
run_dashboard()

## End(Not run)
```

`run_simulation` *Run a Random Walk Simulation*

Description

Executes a complete random walk simulation with the specified parameters. This is the main entry point for running simulations programmatically.

Usage

```
run_simulation(
  grid_size = 10,
  n_walkers = 5,
  neighborhood = c("4-hood", "8-hood"),
  boundary = c("terminate", "wrap"),
  workers = 0,
  sync_mode = c("static", "dynamic", "mirai_dynamic", "chunked"),
  max_steps = 10000L,
  verbose = FALSE,
  quiet = FALSE,
  validate_strict = FALSE,
  validate_percent = 5,
  log_interval = 50
)
```

Arguments

`grid_size`	Integer. Size of the grid (n x n). Default 10.
`n_walkers`	Integer. Number of simultaneous walkers. Default 5. Must be between 1 and 60% of grid size.
`neighborhood`	Character. Either "4-hood" or "8-hood". Default "4-hood".
`boundary`	Character. Either "terminate" or "wrap". Default "terminate".
`workers`	Integer. Number of parallel workers (0 = synchronous). Default 0. For async mode, use 2-4 workers for medium grids, 4-8 for large grids. Requires crew and nanonext packages.

`sync_mode` Character. Grid synchronization mode for async simulations. Options: "static" (default, workers receive frozen grid snapshot), "dynamic" (DEPRECATED - nanonext sockets fail in crew, falls back to static), "mirai_dynamic" (DEPRECATED - same socket issues as dynamic), or "chunked" (RECOMMENDED - processes walkers in batches of 10, updating grid between batches for near-real-time collision detection - ~3x more black pixels). Only applies when workers > 0.

`max_steps` Integer. Maximum steps per walker before forced termination. Default 10000.

`verbose` Logical. If TRUE, enables detailed logging. Default FALSE.

`quiet` Logical. If TRUE, suppresses INFO-level logs (shows only WARN/ERROR). Useful for tests to reduce console output. Default FALSE.

`validate_strict` Logical. If TRUE, validation errors stop simulation. If FALSE, they only log warnings. Default FALSE. Tests should use TRUE.

`validate_percent` Numeric. Validate grid every X% of walkers complete. Default 5 (validates every 5% = 20 times total). Set to 0 to disable periodic validation (only validates at end). Minimum interval is 1 walker.

`log_interval` Integer. Log progress every N completed walkers. Default 50. Set to 0 to disable progress logging (only shows start/end). Lower values (e.g., 5) increase verbosity, higher values (e.g., 100) reduce output.

Value

A list with components:

grid Final grid state

walkers List of final walker states

statistics Simulation statistics

parameters Input parameters

Examples

```
## Not run:
result <- run_simulation(grid_size = 20, n_walkers = 8)
plot_grid(result$grid)
print(result$statistics)

## End(Not run)
```

set_pixel_black *Set Pixel Value in Grid*

Description

Sets the value at a given position to black (1).

Usage

```
set_pixel_black(grid, pos, boundary = "terminate")
```

Arguments

grid	Numeric matrix. The simulation grid.
pos	Integer vector of length 2 (row, col).
boundary	Character. Either "terminate" or "wrap". Default is "terminate".

Value

Modified grid matrix.

Examples

```
grid <- initialize_grid(10)
grid <- set_pixel_black(grid, c(3, 3))
grid[3, 3]  # 1
```

sim_input_server *Simulation Input Module Server*

Description

Server logic for handling simulation input parameters and validation.

Usage

```
sim_input_server(id)
```

Arguments

id	Character string. Module namespace ID.

Value

A reactive list containing:

params Reactive list of validated simulation parameters

run_trigger Reactive trigger for simulation execution

---

sim_input_ui *Simulation Input Module UI*

Description

Creates the UI for simulation input parameters.

Usage

```
sim_input_ui(id)
```

Arguments

id	Character string. Module namespace ID.

Value

A shiny tagList containing the input UI elements.

---

sim_output_server *Simulation Output Module Server*

Description

Server logic for displaying simulation results.

Usage

```
sim_output_server(id, sim_result)
```

Arguments

id	Character string. Module namespace ID.
sim_result	Reactive expression containing simulation result from run_simulation().

Value

NULL (called for side effects).

sim_output_ui *Simulation Output Module UI*

Description

Creates the UI for displaying simulation results across multiple tabs.

Usage

```
sim_output_ui(id)
```

Arguments

id	Character string. Module namespace ID.

Value

A shiny tagList containing the output UI elements.

simulate_walker_dynamic *Simulate Walker with Dynamic Grid Broadcasting*

Description

Simulates a single walker with real-time grid synchronization via nanonext pub/sub pattern. Workers receive broadcasts of new black pixels, enabling realistic walker interactions.

Usage

```
simulate_walker_dynamic(
  walker_id,
  initial_grid,
  pub_socket = NULL,
  sub_socket,
  grid_size,
  neighborhood = "4-hood",
  boundary = "terminate",
  max_steps = 10000L
)
```

Arguments

walker_id	Integer. Unique walker identifier.
initial_grid	Matrix. Initial grid state.
pub_socket	Publisher socket for broadcasting updates (optional). If NULL, worker returns results without broadcasting (main process broadcasts).
sub_socket	Subscriber socket for receiving updates.
grid_size	Integer. Grid dimensions (n x n).
neighborhood	Character. "4-hood" or "8-hood".
boundary	Character. "terminate" or "wrap".
max_steps	Integer. Maximum steps before forced termination.

Value

List with walker results:

walker_id Walker ID

status Termination reason

steps Steps taken

position Final position

path List of positions visited

black_pixel_created Logical. TRUE if created black pixel

---

step_walker *Move Walker One Step*

Description

Moves the walker one random step in the neighborhood.

Usage

```
step_walker(
  walker,
  neighborhood = c("4-hood", "8-hood"),
  boundary = c("terminate", "wrap")
)
```

Arguments

walker	List. Walker object.
neighborhood	Character. "4-hood" or "8-hood".
boundary	Character. "terminate" or "wrap".

Value

Modified walker object.

touches_black *Check if Walker Touches a Black Pixel*

Description

Checks if the walker's current position is on a black pixel.

Usage

```
touches_black(walker, grid, boundary = c("terminate", "wrap"))
```

Arguments

walker	List. Walker object.
grid	Numeric matrix. The simulation grid.
boundary	Character. Boundary condition ("terminate" or "wrap").

Value

Logical. TRUE if walker is on a black pixel.

update_grid_from_broadcasts *Update Local Grid from Broadcasts*

Description

Non-blocking check for pending black pixel updates and applies them to local grid

Usage

```
update_grid_from_broadcasts(sub_socket, local_grid)
```

Arguments

sub_socket	Subscriber socket
local_grid	Current local grid state

Value

Updated local grid

validate_no_isolated_pixels *Validate Grid State for Isolated Pixels*

Description

Checks that no black pixel is completely isolated (has no black neighbors). An isolated pixel indicates a bug in the simulation logic. Also validates that the grid has at least one black pixel, as the number of black pixels should increase monotonically from the initial center pixel.

Usage

```
validate_no_isolated_pixels(
  grid,
  neighborhood = "4-hood",
  boundary = "terminate",
  strict = FALSE,
  last_black_positions = NULL,
  walkers = NULL,
  step_count = NULL
)
```

Arguments

grid	Numeric matrix representing the grid.
neighborhood	Character, "4-hood" or "8-hood" for neighbor checking. Default is "4-hood".
boundary	Character. "terminate" or "wrap". Default "terminate".
strict	Logical, if TRUE throws error on isolation, if FALSE logs warning. Default FALSE.
last_black_positions	Matrix of previously validated black pixel positions (optional). Used for optimization - only checks NEW pixels since last validation. Default NULL (checks all pixels).
walkers	List of walker objects (optional). Used for detailed debugging output when isolation detected. Default NULL.
step_count	Integer simulation step count (optional). Used for detailed debugging output when isolation detected. Default NULL.

Value

Logical, TRUE if valid (no isolated pixels), FALSE otherwise.

Examples

```
grid <- initialize_grid(10)
validate_no_isolated_pixels(grid)  # TRUE for single center pixel initially

# Create invalid grid with isolated pixel
bad_grid <- initialize_grid(10, center_black = FALSE)
bad_grid[3, 3] <- 1
validate_no_isolated_pixels(bad_grid)  # FALSE - isolated pixel
```

`validate_termination_position` *Validate Termination Position*

Description

Checks if a position is valid for termination (has at least one black neighbor or is the initial center pixel). Used in async mode to prevent isolated pixels when workers operate on stale grid snapshots.

Usage

```
validate_termination_position(pos, grid, neighborhood = "4-hood")
```

Arguments

`pos`	Integer vector of length 2 (row, col). Position to validate.
`grid`	Numeric matrix. Current grid state.
`neighborhood`	Character. "4-hood" or "8-hood". Default "4-hood".

Details

A termination position is valid if:

1. The position itself is already black (touching black), OR
2. The position has at least one black neighbor

This prevents the creation of isolated black pixels in async mode where workers may have stale grid snapshots.

Value

Logical. TRUE if position is valid for termination, FALSE otherwise.

See Also

`validate_no_isolated_pixels`

Examples

```
grid <- initialize_grid(10)
# Center (5,5) is black, so (5,6) has a black neighbor
validate_termination_position(c(5, 6), grid, "4-hood")  # TRUE
# Position (1,1) is far from center - no black neighbors
validate_termination_position(c(1, 1), grid, "4-hood")  # FALSE
```

---

worker_check_updates *Check for Grid Updates (Worker)*

Description

Non-blocking check for grid state updates from the publisher socket. Updates the worker's local cache if new pixel updates are available.

Usage

```
worker_check_updates(worker_state)
```

Arguments

worker_state	List. Worker state from `worker_init()`.

Details

Polls the subscriber socket for new messages without blocking. If updates are available, processes all queued messages and updates the local black_pixels cache and version number.

This function should be called periodically between walker steps to keep the cache reasonably fresh. In Phase 1, it's called before each walker step.

Update message format (from broadcaster): `list(type = "pixel_update", position = c(row, col), version = int)`

Value

Modified worker_state with updated cache (if updates received).

See Also

worker_init, broadcast_update

Examples

```
## Not run:
# Check for updates before stepping walker
worker_state <- worker_check_updates(worker_state)

## End(Not run)
```

`worker_init` *Initialize Worker with Subscriber Socket*

Description

Sets up a worker process to receive grid state updates via nanonext. Called once per worker at startup.

Usage

```
worker_init(pub_address)
```

Arguments

`pub_address`	Character. Publisher socket address (e.g., "tcp://127.0.0.1:5555").

Details

Creates a subscriber socket that listens for pixel update broadcasts from the main process. Also initializes a local cache to store the current set of black pixels and grid state version.

The cache is used to avoid querying the main process for grid state on every walker step. Workers check for updates periodically and refresh the cache when the version number changes.

Value

A list containing:

socket nanonext subscriber socket

cache List with black_pixels and version

See Also

`worker_check_updates`, `worker_step_walker`

Examples

```
## Not run:
# Worker initialization (executed in crew worker process)
worker_state <- worker_init("tcp://127.0.0.1:5555")

## End(Not run)
```

worker_run_walker *Run Walker Until Termination (Worker Task)*

Description

Executes a complete walker simulation until termination. This is the function pushed to crew workers as a task.

Usage

```
worker_run_walker(
  walker,
  grid_state,
  pub_address = NULL,
  neighborhood = "4-hood",
  boundary = "terminate",
  max_steps = 10000L
)
```

Arguments

walker	List. Walker object.
grid_state	List. Grid state (grid matrix, black_pixels, grid_size).
pub_address	Character. Publisher socket address.
neighborhood	Character. "4-hood" or "8-hood".
boundary	Character. "terminate" or "wrap".
max_steps	Integer. Maximum steps limit.

Details

This function represents a complete worker task:

1. Initialize worker (create subscriber socket, cache)
2. Step walker repeatedly until termination
3. Clean up worker resources
4. Return final walker state

The worker maintains a local cache of black pixels and subscribes to updates from the main process. This allows independent execution while staying reasonably synchronized with the global grid state.

Value

Terminated walker object with complete path.

See Also

worker_step_walker, worker_init

Examples

```
## Not run:
# This function is called by crew workers, not directly by users
# Controller pushes this task:
controller$push(
  command = worker_run_walker(walker, grid_state, pub_address, ...),
  data = list(walker = walker, grid_state = grid_state, ...)
)

## End(Not run)
```

`worker_step_walker` *Execute Walker Step (Worker)*

Description

Executes one step of a walker using cached grid state. This is the main work function executed by crew workers.

Usage

```
worker_step_walker(
  walker,
  grid_state,
  worker_state,
  neighborhood = "4-hood",
  boundary = "terminate",
  max_steps = 10000L
)
```

Arguments

`walker`	List. Walker object (from `create_walker()`).
`grid_state`	List. Contains grid matrix and black_pixels set.
`worker_state`	List. Worker state from `worker_init()`.
`neighborhood`	Character. "4-hood" or "8-hood" (default: "4-hood").
`boundary`	Character. "terminate" or "wrap" (default: "terminate").
`max_steps`	Integer. Maximum steps before forced termination (default: 10000).

Details

Performs one iteration of the random walk:

1. Check for grid updates (refresh cache)
2. Move walker one step using `step_walker()`

3. Check termination using cached black pixels

The worker uses its local cache of black pixels to check termination conditions. This avoids querying the main process on every step, significantly reducing synchronization overhead.

Cache staleness is acceptable because:

- Random walks are stochastic (small delays don't affect statistical properties)
- Updates are broadcast immediately when walkers terminate
- Worker checks for updates before each step

Value

Modified walker object after one step.

See Also

step_walker, check_termination

Examples

```
## Not run:
# Step a walker in a worker process
walker <- worker_step_walker(
  walker = walker,
  grid_state = grid_state,
  worker_state = worker_state,
  neighborhood = "4-hood",
  boundary = "terminate"
)

## End(Not run)
```

---

wrap_position *Wrap Position Around Grid Boundaries (Torus Topology)*

Description

Wrap Position Around Grid Boundaries (Torus Topology)

Usage

```
wrap_position(pos, n)
```

Arguments

pos	Integer vector of length 2 (row, col).
n	Integer. Grid size.

Value

Integer vector of length 2 with wrapped coordinates.

Examples

```
wrap_position(c(0, 5), 10)   # c(10, 5)
wrap_position(c(11, 5), 10)  # c(1, 5)
wrap_position(c(5, 0), 10)   # c(5, 10)
```

Index