# Package 'actimetrics'

October 29, 2026

**Type** Package

**Title** Create Metrics of Actigraphy and Activity Analysis

**Version** 0.10.0

**Description** Provides functions for calibrating, counting, and summarizing actigraphy and activity data into specific metrics and sleep measures. The metrics include activity counts, step counts, activity index, Monitor Independent Movement Summary Unit (MIMS), mean amplitude deviation (MAD), and provides wrappers for sleep estimation from activity counts using Tudor-Locke (2014) <doi:10.1139/apnm-2013-0173> and Sadeh (1994) <doi:10.1093/sleep/17.3.201>. Count-intensity thresholds include Montoye et al. (2020) <doi:10.1080/02640414.2020.1794244> and Sasaki et al. (2011) <doi:10.1016/j.jsams.2011.04.003>. Raw-acceleration Euclidean Norm Minus One (ENMO) thresholds include Hildebrand et al. (2014) <doi:10.1249/MSS.0000000000000289>.

**License** GPL-3

**Depends** R (>= 4.1.0)

**Suggests** testthat (>= 3.0.0), utils, covr, knitr, httr, rmarkdown, MIMSunit, data.table, reticulate, stepcount (>= 0.6.0), agcounts (>= 0.6.7), callr

**Encoding** UTF-8

**VignetteBuilder** knitr

**Imports** actibase (>= 0.8.0), actiread, actigraph.sleepr (>= 0.3.1), dplyr, lubridate, assertthat, purrr, readr, tibble, rlang, stats, tidyr, walking (>= 0.8.0), methods

**LazyData** true

**Config/roxygen2/version** 8.0.0

**URL** https://github.com/jhuwit/actimetrics, https://jhuwit.github.io/actimetrics/

**BugReports** https://github.com/jhuwit/actimetrics/issues

**Config/testthat/edition** 3

**NeedsCompilation** no

**Author** John Muschelli [aut, cre] (ORCID:
<https://orcid.org/0000-0001-6469-1750>)

**Maintainer** John Muschelli <muschellij2@gmail.com>

**Repository** CRAN

**Date/Publication** 2026-10-29 08:40:09 UTC

# Contents

acti_apply_tudor_locke . . . 3
acti_apply_tudor_locke_diary . . . 3
acti_calculate_ai . . . 4
acti_calculate_counts . . . 6
acti_calculate_forest . . . 8
acti_calculate_measures . . . 8
acti_calculate_sdt . . . 11
acti_calculate_stepcount . . . 12
acti_calculate_verisense . . . 13
acti_calculate_wear_raw . . . 13
acti_calculate_wear_raw_actipy . . . 14
acti_calculate_wear_raw_ggir . . . 16
acti_calibrate . . . 18
acti_count_data . . . 19
acti_process . . . 20
acti_process_study . . . 21
acti_sleep_diary . . . 22
acti_sleep_ensemble . . . 22
acti_sleep_fuse_labels . . . 23
acti_sleep_guider . . . 24
acti_sleep_guider_requirements . . . 25
acti_sleep_hdcza . . . 26
acti_sleep_hlrb . . . 27
acti_sleep_horangle . . . 27
acti_sleep_l5 . . . 28
acti_sleep_label . . . 29
acti_sleep_labels . . . 29
acti_sleep_notworn . . . 30
acti_sleep_setwindow . . . 31
acti_sleep_sib . . . 31
acti_threshold_counts . . . 32
acti_threshold_enmo . . . 33
acti_threshold_hildebrand . . . 34
acti_threshold_montoye . . . 35
acti_threshold_romanzini . . . 35
acti_threshold_sasaki . . . 36
mims_default_extrapolation . . . 37

mims_default_processing . . . . . 37
py_acti_calculate_forest . . . . . 38
py_acti_calculate_stepcount . . . . . 39

**Index** **41**

---

`acti_apply_tudor_locke` *Apply Tudor–Locke sleep period detection*

---

## Description

Delegates to `actigraph.sleepr::apply_tudor_locke()`. Input must be 60-second epoch data with activity counts and a `sleep` state. Data from `acti_apply_cole_kripke()` is accepted directly. Data labeled by `acti_apply_tudor_locke_diary()` is also accepted: its logical `sleep` column is converted to the `S/W` states expected downstream.

## Usage

```
acti_apply_tudor_locke(data, ..., sleep_col = "sleep")
```

## Arguments

| | |
|---|---|
| `data` | Epoch-level data containing activity counts and sleep state. |
| `...` | Arguments passed to `actigraph.sleepr::apply_tudor_locke()`. |
| `sleep_col` | Name of the sleep-state column in `data`, either logical (as from `acti_apply_tudor_locke_diary()`) or `S/W`. It is used as the `sleep` column expected by `actigraph.sleepr`. |

## Value

A data frame summarizing detected sleep periods.

---

`acti_apply_tudor_locke_diary` *Apply a sleep diary to epoch data for Tudor–Locke*

---

## Description

Adds a logical sleep column to epoch data from diary onset and wakeup times. The result can be passed to `acti_apply_tudor_locke()` when it also has activity counts and represents 60-second epochs. The wrapper converts the logical diary labels to the `S/W` states required downstream.

## Usage

```
acti_apply_tudor_locke_diary(
  data,
  sleep_diary,
  time_col = "time",
  diary_onset_col = "onset",
  diary_wakeup_col = "wakeup",
  sleep_col = "sleep"
)
```

## Arguments

| | |
|---|---|
| `data` | Epoch-level data containing a POSIXct timestamp column. |
| `sleep_diary` | A data frame with POSIXct sleep onset and wakeup columns. Each interval may span midnight; multiple days are supported. |
| `time_col` | Name of the POSIXct timestamp column in `data`. |
| `diary_onset_col` | Name of the diary onset column. |
| `diary_wakeup_col` | Name of the diary wakeup column. |
| `sleep_col` | Name of the logical sleep column to add or replace. |

## Value

`data` with a logical `sleep_col`, `TRUE` from onset (inclusive) to wakeup (exclusive).

---

# `acti_calculate_ai` *Calculate the Activity Index (AI)*

---

## Description

Computes the Activity Index of Bai et al. (2016) from raw tri-axial accelerometry. For each 1-second window $t$, with $\sigma_m^2(t)$ the variance of axis $m \in \{X, Y, Z\}$ over the samples in that second,

$$AI(t) = \sqrt{\frac{1}{3}\sum_m \sigma_m^2(t)}.$$

The per-second values are then summed within each `unit` (e.g. a minute).

## Usage

```
acti_calculate_ai(data, unit = "1 min", ensure_all_time = TRUE, verbose = TRUE)

acti_calculate_activity_index(
  data,
```

```
  unit = "1 min",
  ensure_all_time = TRUE,
  verbose = TRUE
)
```

## Arguments

| | |
|---|---|
| `data` | An object with columns `X`, `Y`, and `Z` and a time column, or an object of class `AccData` |
| `unit` | length of time to sum the per-second AI over, passed to `lubridate::floor_date()` |
| `ensure_all_time` | if `TRUE`, all times from the first to the last `unit` are in the output, even those with no input data |
| `verbose` | print diagnostic messages |

## Details

This is the unadjusted index (sometimes written $AI_0$): it does not subtract or normalise by the device's systematic noise variance $\bar{\sigma}_m^2$, which the original definition estimates from a stationary period. It differs from `AI_DEFINED` returned by `acti_calculate_mad()`, which takes the variances over the whole `unit` rather than summing 1-second values.

A 1-second variance needs at least two samples, so seconds with fewer than two samples are `NA` and are skipped when summing; a `unit` with no valid seconds is `NA`. Data sampled at under 2 Hz therefore give all-`NA` AI (with a warning).

## Value

A `data.frame` (or `data.table` if `data` is one) with columns `HEADER_TIME_STAMP` and `AI`.

## References

Bai J, Di C, Xiao L, et al. (2016). An Activity Index for Raw Accelerometry Data and Its Comparison with Other Activity Metrics. PLoS ONE 11(8): e0160644. doi:10.1371/journal.pone.0160644

## Examples

```
file = actiread::acti_example_gt3x()
res = actiread::acti_read_gt3x(file, verbose = FALSE)
ai = acti_calculate_ai(res[1:12000, ])
```

`acti_calculate_counts` *Process Count Data*

## Description

Process Count Data

Process Count Data

## Usage

```
acti_calculate_counts(
  data,
  epoch = 60L,
  resample = TRUE,
  lfe_select = FALSE,
  verbose = TRUE,
  fix_first_second = TRUE,
  round_digits = NULL
)

acti_calculate_wear(
  data,
  method = c("choi", "troiano"),
  use_magnitude = TRUE,
  ...
)

acti_calculate_nonwear(
  data,
  method = c("choi", "troiano"),
  use_magnitude = TRUE,
  ...
)

acti_apply_cole_kripke(data)

acti_apply_sadeh(data, ...)
```

## Arguments

| | |
|---|---|
| `data` | A `data.frame` from `acti_calculate_counts` that has columns `axis1-3` and `counts` |
| `epoch` | epoch length, either in seconds or a string such as `"1 min"` (see `actibase::acti_standardize_epoch()`) must be a whole number of seconds. Default is 60 seconds. See agcounts::calculate_counts |
| `resample` | (recommended) resample the data to 30Hz using actibase::acti_resample vs. using the resampling method from agcounts::calculate_counts. |

| | |
|---|---|
| lfe_select | Apply the Actigraph Low Frequency Extension filter. See agcounts::calculate_counts |
| verbose | print diagnostic messages. Either logical or integer, where higher values are higher levels of verbosity. |
| fix_first_second | Fix the first second bug in agcounts. Appends replicated data of the first record if the first record is not an "even" epoch (e.g. 60 second epoch and data does not start at a 00 second). |
| round_digits | If not NULL, the number of decimal places the X/Y/Z values are rounded to after resampling and before counting. Use 3 to match ActiLife, which rounds GT3X data to 3 decimals (half away from zero, so 0.0625 becomes 0.063, unlike base::round()). Useful if the data were read with more digits. Default NULL does not round. |
| method | Method for detecting non-wear, either "choi" or "troiano", corresponding to actigraph.sleepr::apply_choi() or actigraph.sleepr::apply_troiano() |
| use_magnitude | If TRUE, the magnitude of the vector (axis1, axis2, axis3) is used to measure activity; otherwise the axis1 value is used. |
| ... | additional arguments to pass to actigraph.sleepr function |

## Details

The sampling frequency is only needed when resample = FALSE; it is taken from the sample_rate attribute (or estimated from time).

acti_calculate_wear(), acti_apply_cole_kripke(), and acti_apply_sadeh() require 60-second epochs. Cole–Kripke, Sadeh, and acti_calculate_wear(use_magnitude = FALSE) score the vertical axis, so data must contain axis1; vector-magnitude-only counts are an error.

## Value

A data.frame of transformed data

A data.frame of transformed data with columns axis1-3, counts, and counts_log10.

A data.frame of transformed data

## Note

This calls the downstream wear-processing helpers used by actigraph.sleepr

## Examples

```
path = actiread::acti_example_gt3x()
ac = actiread::acti_read_gt3x(path)
out = acti_calculate_counts(ac)

data = actimetrics::acti_count_data
wear = actimetrics::acti_calculate_wear(data)
tro_wear = actimetrics::acti_calculate_wear(data, method = "troiano")
ck = actimetrics::acti_apply_cole_kripke(data)
```

```
tl = actimetrics::acti_apply_tudor_locke(ck)
sadeh = actimetrics::acti_apply_sadeh(ck)
```

---

`acti_calculate_forest` *Calculate Step Counts via Oak/Forest*

---

## Description

Calculate Step Counts via Oak/Forest

## Usage

```
acti_calculate_forest(data, ...)
```

## Arguments

| | |
|---|---|
| `data` | A `data.frame`, `AccData` object, or GT3X file with `X`, `Y`, `Z`, and `time` |
| `...` | Additional arguments passed to `walking::find_walking()` |

## Value

A tibble with minute-level `time`, `steps` columns.

## Examples

```
Sys.setenv("SSQ_PARALLEL" = 0)
if (reticulate::py_module_available("forest")) {
  data = actiread::acti_read_gt3x(actiread::acti_example_gt3x())
  steps = acti_calculate_forest(data, sample_rate = 100)
}
```

---

`acti_calculate_measures`
*Calculate Summary Measures from Raw Accelerometer Data*

---

## Description

Calculate Summary Measures from Raw Accelerometer Data

## Usage

```
acti_calculate_measures(
  data,
  unit = "1 min",
  fix_zeros = TRUE,
  dynamic_range = NULL,
  calculate_mims = TRUE,
  calculate_ac = TRUE,
  flag_data = TRUE,
  flags = NULL,
  ensure_all_time = TRUE,
  verbose = TRUE,
  sample_rate = NULL,
  ...
)

acti_calculate_flags(data, unit = "1 min", ensure_all_time = TRUE)

acti_calculate_n_idle(data, unit = "1 min", ensure_all_time = TRUE)

acti_calculate_enmo(...)

acti_calculate_ai_defined(...)

acti_calculate_mad(
  data,
  unit = "1 min",
  ensure_all_time = TRUE,
  verbose = TRUE
)

acti_calculate_auc(
  data,
  unit = "1 min",
  sample_rate = NULL,
  allow_truncation = FALSE,
  ensure_all_time = TRUE,
  verbose = TRUE
)

acti_calculate_fast_mims(
  data,
  unit = "1 min",
  dynamic_range = NULL,
  sample_rate = NULL,
  allow_truncation = TRUE,
  ensure_all_time = TRUE,
  verbose = TRUE,
```

```
  ...
)

acti_calculate_mims(
  data,
  unit = "1 min",
  dynamic_range = c(-6, 6),
  ensure_all_time = TRUE,
  ...
)
```

## Arguments

| | |
|---|---|
| data | An object with columns X, Y, and Z or an object of class AccData |
| unit | length of time to calculate measures over. a character string specifying a time unit or a multiple of a unit to be rounded to. Valid base units are second, minute, hour, day, week, month, bimonth, quarter, season, halfyear, and year. Arbitrary unique English abbreviations as in the lubridate::period() constructor are allowed. |
| fix_zeros | Should actibase::acti_fill_zeros() be run before calculating the measures? |
| dynamic_range | Dynamic range of the device, in gravity units |
| calculate_mims | Should MIMS units be calculated? |
| calculate_ac | Should Activity Counts from the agcounts package be calculated? |
| flag_data | Should the downstream overlay flag_qc() be run? It will be executed after fix_zeros before any measure calculation |
| flags | the flags to calculate, passed to the downstream overlay flag_qc() |
| ensure_all_time | if TRUE, then all times from the first to last times will be in the output, even if data during that time was not in the input |
| verbose | print diagnostic messages |
| sample_rate | sample rate of data, if not specified in header of object |
| ... | additional arguments to pass to MIMSunit::mims_unit() |
| allow_truncation | truncate small values to 0 and mark epochs with impossibly large values as invalid (-0.01 for all axes and the total), following MIMSunit |

## Value

A data set with the calculated features

## Examples

```
file = actiread::acti_example_gt3x()
res = actiread::acti_read_gt3x(file, verbose = FALSE)
res = res[1:12000, ]
```

```
measures = acti_calculate_measures(
  res,
  dynamic_range = NULL,
  calculate_mims = FALSE,
  calculate_ac = FALSE,
  flag_data = FALSE
)
auc = acti_calculate_auc(res)

mims = acti_calculate_mims(res, dynamic_range = NULL)

if (requireNamespace("data.table", quietly = TRUE)) {
   dt = data.table::as.data.table(res)
   out = acti_calculate_measures(dt, calculate_mims = FALSE, flag_data = FALSE,
     calculate_ac = FALSE)
}
```

---

`acti_calculate_sdt` *Calculate Step Counts via SDT*

---

## Description

Calculate Step Counts via SDT

## Usage

```
acti_calculate_sdt(data, sample_rate = NULL, ...)
```

## Arguments

| | |
|---|---|
| `data` | A `data.frame`, `AccData` object, or GT3X file with `X`, `Y`, `Z`, and `time` |
| `sample_rate` | Sample rate in Hz. If omitted, it is taken from the input object when available. |
| `...` | Additional arguments passed to `walking::sdt_count_steps()` |

## Value

A tibble with minute-level `time`, `steps` columns.

## Examples

```
data = actiread::acti_read_gt3x(actiread::acti_example_gt3x())
steps = acti_calculate_sdt(data)
```

`acti_calculate_stepcount` *Calculate Step Counts via* `stepcount`

## Description

Use the `stepcount` package to estimate steps from raw accelerometer data and summarize them to minute-level epochs (as opposed to 10s default)

## Usage

```
acti_calculate_stepcount(
  data,
  sample_rate = NULL,
  ...,
  epoch = "1 minute",
  tz = NULL
)
```

## Arguments

| | |
|---|---|
| `data` | A `data.frame`, `AccData` object, or GT3X file with `X`, `Y`, `Z`, and `time` |
| `sample_rate` | Sample rate in Hz. If omitted, it is taken from the input object when available. |
| `...` | Additional arguments passed to `stepcount::stepcount()` |
| `epoch` | epoch to aggregate the data to, in seconds or a string such as `"1 minute"` (see `actibase::acti_standardize_epoch()`); original output is at 10-seconds |
| `tz` | Time zone of the returned times. `stepcount` reports local clock times without a time zone; by default they are given the time zone of `data$time` (so they line up with the input), or `"UTC"` when `data` is a file path. Set `tz` to the recording's time zone for file input. |

## Value

A tibble with minute-level `time`, `steps`, and `walking` columns.

## Examples

```
# reticulate::py_require("stepcount==3.11.0", python_version = "3.10", action = "add")
# sc = try({ reticulate::import("stepcount") })
if (stepcount::have_stepcount()) {
  data = actiread::acti_read_gt3x(actiread::acti_example_gt3x())
  steps = acti_calculate_stepcount(data, sample_rate = 100)
  steps = acti_calculate_stepcount(data, model_type = "rf")
}
```

`acti_calculate_verisense` *Calculate Step Counts via Verisense*

## Description

Calculate Step Counts via Verisense

## Usage

```
acti_calculate_verisense(
  data,
  resample_to_15hz = TRUE,
  method = c("original", "revised"),
  ...
)
```

## Arguments

| | |
|---|---|
| `data` | A `data.frame`, `AccData` object, or GT3X file with `X`, `Y`, `Z`, and `time` |
| `resample_to_15hz` | resample data to 15Hz, passed to `walking::estimate_steps_verisense()` |
| `method` | parameters to estimate walking, either original or revised, passed to `walking::estimate_steps_verise` |
| `...` | Additional arguments passed to `walking::estimate_steps_verisense()` |

## Value

A tibble with minute-level `time`, `steps` columns.

## Examples

```
data = actiread::acti_read_gt3x(actiread::acti_example_gt3x())
steps = acti_calculate_verisense(data)
```

---

`acti_calculate_wear_raw` *Estimate Non-wear from Raw Accelerometry*

## Description

Detects non-wear from raw tri-axial acceleration. Unlike `acti_calculate_wear()`, which works on 60-second activity counts, this function takes the raw `X`, `Y`, `Z` signal (in g) as input. It calls `acti_calculate_wear_raw_ggir()` or `acti_calculate_wear_raw_actipy()`; see those functions for the algorithms and their arguments. Neither GGIR nor actipy is required.

## Usage

```
acti_calculate_wear_raw(data, method = c("ggir", "actipy"), ...)
```

## Arguments

| | |
|---|---|
| data | A data.frame of raw accelerometer data with a time column and X, Y, Z columns, or an AccData object. Data should be calibrated (e.g. with acti_calibrate()). |
| method | Method for detecting non-wear, either "ggir" or "actipy", corresponding to acti_calculate_wear_raw_ggir() or acti_calculate_wear_raw_actipy(). |
| ... | additional arguments to pass to the method function, such as sample_rate, approach, or sd_threshold for "ggir", or window, patience, or sd_threshold for "actipy". |

## Value

A tibble with one row per epoch, containing at least time (start of the epoch), nonwear, and wear. The epoch length and the other columns depend on method.

## See Also

acti_calculate_wear_raw_ggir(), acti_calculate_wear_raw_actipy()

## Examples

```
n = 3 * 3600 * 10
time = as.POSIXct("2020-01-01 00:00:00", tz = "UTC") + (seq_len(n) - 1) / 10
moving = rep(c(TRUE, FALSE, TRUE), each = n / 3)
set.seed(1)
data = data.frame(
  time = time,
  X = ifelse(moving, stats::rnorm(n, sd = 0.3), 0),
  Y = ifelse(moving, stats::rnorm(n, sd = 0.3), 0),
  Z = ifelse(moving, stats::rnorm(n, sd = 0.3), 1)
)
ggir = acti_calculate_wear_raw(data, sample_rate = 10)
actipy = acti_calculate_wear_raw(data, method = "actipy", patience = 3000)
```

---

# acti_calculate_wear_raw_actipy

*Estimate Non-wear from Raw Accelerometry with the actipy Method*

---

## Description

Detects non-wear from raw tri-axial acceleration with the stationary-episode definition of Doherty et al. (2017), as implemented in actipy's detect_nonwear() (Chan and Doherty). This is a standalone port and does not require actipy or Python. Usually called through acti_calculate_wear_raw().

## Usage

```
acti_calculate_wear_raw_actipy(
  data,
  window = 10,
  patience = 5400,
  sd_threshold = 0.015
)
```

## Arguments

| | |
|---|---|
| `data` | A `data.frame` of raw accelerometer data with a time column and `X`, `Y`, `Z` columns, or an `AccData` object. Data should be calibrated (e.g. with `acti_calibrate()`). |
| `window` | Length, in seconds, of the windows used to test for stationarity. |
| `patience` | Minimum span, in seconds, of a stationary run for it to be non-wear. |
| `sd_threshold` | Standard deviation (g) below which an axis is considered still. |

## Details

The signal is split into windows of `window` seconds, anchored at the first sample. A window is stationary when the standard deviation of every axis is below `sd_threshold`. A run of stationary windows is non-wear when its span, from the start of its first window to the start of its last window, is longer than `patience` seconds. Windows with fewer than two samples (for example, during time gaps) are not stationary.

The defaults are actipy's (10 seconds, 90 minutes, 15 mg). Doherty et al. (2017) used 60 minutes and 13 mg (`patience = 3600, sd_threshold = 0.013`). actipy sets samples to `NaN` from the start of the first window to the start of the last window of a run; this function labels whole windows, so the last window of a run is labelled non-wear. actipy runs this step after low-pass filtering and gravity calibration.

## Value

A `tibble` with one row per window: `time` (start of the window), `stationary`, `nonwear`, and `wear`.

## References

Doherty A, et al. (2017). Large scale population assessment of physical activity using wrist worn accelerometers: the UK Biobank study. PLoS ONE 12(2): e0169649. doi:10.1371/journal.pone.0169649

Chan S, Doherty A. actipy: a Python package to process wearable sensor data. doi:10.5281/zenodo.14514488

## See Also

`acti_calculate_wear_raw()`, `acti_calculate_wear_raw_ggir()`

## Examples

```
n = 3 * 3600 * 10
time = as.POSIXct("2020-01-01 00:00:00", tz = "UTC") + (seq_len(n) - 1) / 10
moving = rep(c(TRUE, FALSE, TRUE), each = n / 3)
set.seed(1)
data = data.frame(
  time = time,
  X = ifelse(moving, stats::rnorm(n, sd = 0.3), 0),
  Y = ifelse(moving, stats::rnorm(n, sd = 0.3), 0),
  Z = ifelse(moving, stats::rnorm(n, sd = 0.3), 1)
)
wear = acti_calculate_wear_raw_actipy(data, patience = 3000)
table(wear$wear)
```

---

`acti_calculate_wear_raw_ggir`

*Estimate Non-wear from Raw Accelerometry with the GGIR Method*

---

## Description

Detects non-wear and signal clipping from raw tri-axial acceleration with the algorithm in GGIR (van Hees et al. 2011, 2013), as implemented in GGIR's `detect_nonwear_clipping()` and `g.weardec()`. This is a standalone port and does not require GGIR. Usually called through `acti_calculate_wear_raw()`.

## Usage

```
acti_calculate_wear_raw_ggir(
  data,
  sample_rate = NULL,
  medium_epoch = 900,
  long_epoch = 3600,
  approach = c("2023", "2013"),
  sd_threshold = 0.013,
  range_threshold = 0.15,
  dynamic_range = NULL,
  clip_threshold = NULL,
  wear_threshold = 2,
  clipping_threshold = 0.3,
  edge_correction = TRUE,
  filter_max_hours = NULL,
  filter_window = NULL,
  align_start = TRUE
)
```

## Arguments

| | |
|---|---|
| `data` | A `data.frame` of raw accelerometer data with a time column and `X`, `Y`, `Z` columns, or an `AccData` object. Data should be calibrated (e.g. with `acti_calibrate()`) and regularly sampled. |

`sample_rate` Sampling frequency (Hz). If `NULL`, taken from the `sample_rate` attribute of `data`.

`medium_epoch` Length, in seconds, of the epoch the output is reported in (GGIR `windowsizes[2]`).

`long_epoch` Length, in seconds, of the window used to evaluate non-wear (GGIR `windowsizes[3]`).

`approach` Non-wear window approach (GGIR `nonwear_approach`). `"2023"` (GGIR default) uses a forward-looking long window, evaluated on a subsample of `ceiling(sample_rate / 5)`, and labels every medium epoch it covers. `"2013"` uses a long window centred on each medium epoch and labels only that epoch.

`sd_threshold` Standard deviation (g) below which an axis is considered still. GGIR uses 0.013, or `1.2 * rmc.noise` for ad-hoc formats.

`range_threshold` Range (g) below which an axis is considered still (GGIR `nonwear_range_threshold`, in mg, divided by 1000). GGIR uses 0.15, or 0.2 for Verisense devices.

`dynamic_range` Dynamic range of the device (g), e.g. `c(-8, 8)`. If `NULL`, it is taken from the attributes or header of `data` when available. Used to set `clip_threshold`.

`clip_threshold` Absolute acceleration (g) above which a sample is counted as clipped. Defaults to the dynamic range minus 0.5, or 7.5 when the dynamic range is unknown, as in GGIR.

`wear_threshold` Minimum number of axes that must be still for an epoch to be non-wear. Fixed at 2 in GGIR.

`clipping_threshold` Fraction of clipped samples above which an epoch is flagged as clipping. Fixed at 0.3 in GGIR.

`edge_correction` Should non-wear starting (ending) within the first (last) 3 hours of the recording be extended to the start (end)? GGIR `nonWearEdgeCorrection`.

`filter_max_hours`, `filter_window` If `filter_max_hours` is not `NULL`, non-wear bouts shorter than `filter_max_hours` hours occurring inside the clock window `filter_window = c(start_hour, end_hour)` are relabelled as wear (GGIR `nonwearFiltermaxHours` and `nonwearFilterWindow`). The window may wrap midnight, e.g. `c(22, 7)`.

`align_start` Should samples before the first clock time aligned to `medium_epoch` be dropped, as GGIR does? If `FALSE`, epochs start at the first sample.

## Details

For each medium epoch (default 15 minutes), a long window (default 60 minutes) is evaluated per axis. An axis is scored as non-wear when both its standard deviation is below `sd_threshold` and its range is below `range_threshold`. The non-wear score is the number of axes (0–3) that meet both criteria, and an epoch is non-wear when the score is at least `wear_threshold`. An epoch is clipped when more than `clipping_threshold` of its samples on any axis exceed `clip_threshold` (or, if any sample exceeds `1.5 * clip_threshold`, the whole epoch is clipped). Short wear bouts surrounded by long non-wear bouts are then relabelled as non-wear, following GGIR part 2.

GGIR reads files in chunks (24 hours for GT3X files) and scores each chunk separately. With `approach = "2023"`, a window that starts in one chunk cannot label epochs in the next, so GGIR's `nonwearscore` can be lower for the first epochs after a chunk boundary. This function scores the whole recording at once. GGIR also imputes time gaps before scoring; this function does not.

## Value

A `tibble` with one row per medium epoch:

`time` Start of the epoch.

`nonwear_score` Number of axes (0–3) meeting the non-wear criteria (GGIR `nonwearscore`).

`clipping_score` Maximum fraction of clipped samples across axes (GGIR `clippingscore`).

`nonwear` Non-wear from the score (GGIR `r1`).

`clipping` Clipping (GGIR `r2`).

`nonwear_additional` Additional non-wear from the bout rules (GGIR `r3`).

`wear` `TRUE` if none of `nonwear`, `clipping`, or `nonwear_additional` is flagged.

## References

van Hees VT, et al. (2011). Estimation of daily energy expenditure in pregnant and non-pregnant women using a wrist-worn tri-axial accelerometer. PLoS ONE 6(7): e22922. doi:10.1371/journal.pone.0022922

van Hees VT, et al. (2013). Separating movement and gravity components in an acceleration signal and implications for the assessment of human daily physical activity. PLoS ONE 8(4): e61691. doi:10.1371/journal.pone.0061691

## See Also

`acti_calculate_wear_raw()`, `acti_calculate_wear_raw_actipy()`

## Examples

```
n = 3 * 3600 * 10
time = as.POSIXct("2020-01-01 00:00:00", tz = "UTC") + (seq_len(n) - 1) / 10
moving = rep(c(TRUE, FALSE, TRUE), each = n / 3)
set.seed(1)
data = data.frame(
  time = time,
  X = ifelse(moving, stats::rnorm(n, sd = 0.3), 0),
  Y = ifelse(moving, stats::rnorm(n, sd = 0.3), 0),
  Z = ifelse(moving, stats::rnorm(n, sd = 0.3), 1)
)
acti_calculate_wear_raw_ggir(data, sample_rate = 10)
```

---

`acti_calibrate` *Calibrate Accelerometer Data using* `agcounts`

---

## Description

This uses the van Hees calibration method typically exposed through GGIR, implemented here via `agcounts::agcalibrate()`.

### Usage

```
acti_calibrate(
  data,
  verbose = TRUE,
  fill_zeroes = TRUE,
  round_after_calibration = TRUE,
  ...
)
```

### Arguments

| | |
|---|---|
| data | Either a GT3X file, AccData object, or data.frame with X/Y/Z and time |
| verbose | print diagnostic messages, higher number result in higher verbosity |
| fill_zeroes | Should actibase::acti_fill_zeros() be run before calculating the measures? This trims zero values from the beginning and the end of the time course using last observation carried forward behavior. |
| round_after_calibration | Should the data be rounded after calibration? Will round to 3 digits |
| ... | Additional arguments to pass to agcounts::agcalibrate() |

### Value

A calibrated dataset with the same columns as the input data, but with the X/Y/Z values calibrated using the van Hees method.

### Examples

```
res = acti_calibrate(data = actiread::acti_example_gt3x())
```

---

## acti_count_data *Example Actigraphy/Activity Count Data*

---

### Description

Example Actigraphy/Activity Count Data

### Usage

```
acti_count_data
```

## Format

A `data.frame` with the columns

**time** time at the minute level

**axis1** axis1 (Y) counts

**axis2** axis2 (X) counts

**axis3** axis3 (Z) counts

**counts** vector magnitude of all 3 axes column

This data was taken from running agcounts::calculate_counts via acti_calculate_counts on actibase::acti_raw_data.

---

`acti_process` *Process Raw data to Counts*

---

## Description

Process Raw data to Counts

## Usage

```
acti_process(
  data,
  lfe_select = FALSE,
  method = c("choi", "troiano"),
  use_magnitude = TRUE,
  calibrate = FALSE,
  verbose = TRUE,
  ...,
  min_required = 1368L,
  round_digits = NULL
)
```

## Arguments

| | |
|---|---|
| `data` | A `data.frame` from `acti_calculate_counts` that has columns `axis1-3` and `counts` |
| `lfe_select` | Apply the Actigraph Low Frequency Extension filter. See agcounts::calculate_counts |
| `method` | Method for detecting non-wear, either "choi" or "troiano", corresponding to `actigraph.sleepr::apply_choi()` or `actigraph.sleepr::apply_troiano()` |
| `use_magnitude` | If `TRUE`, the magnitude of the vector (axis1, axis2, axis3) is used to measure activity; otherwise the axis1 value is used. |
| `calibrate` | Logical. If `TRUE`, the data will be calibrated using `acti_calibrate()` |
| `verbose` | print diagnostic messages. Either logical or integer, where higher values are higher levels of verbosity. |

| | |
|---|---|
| ... | additional arguments to pass to actigraph.sleepr function |
| min_required | Number of minutes required in a day to be called included. No day inclusion is run if NULL. |
| round_digits | Number of decimal places X/Y/Z are rounded to (half away from zero) after resampling to 30Hz and before counting. The default NULL uses 3 for GT3X data (read with actiread::acti_read_gt3x()) and does not round other data. Use NA to never round. |

## Details

To match ActiLife for GT3X files, a GT3X path is read with digits = 8 (unless digits is given in ...), resampled, and then rounded to 3 digits. ActiLife rounds the scaled samples to 3 digits, half away from zero. Reading at 8 digits and then rounding gives exactly those values, but rounding values already read at 5 digits does not (e.g. 0.1234996 -> 0.12350 -> 0.124).

## Value

A data frame containing activity counts and wear-time indicators.

## Note

For acti_process, the ... argument are passed to actiread::acti_read_gt3x()

---

# acti_process_study *Process all supported actigraphy files in a directory*

---

## Description

Files are read according to their extension, passed to acti_process(), and saved as individual RDS files in outdir. Supported extensions are GT3X, CWA, GeneActiv BIN, and CSV, including .gz and .xz compressed suffixes for the supported formats described below.

## Usage

```
acti_process_study(directory, outdir, verbose = TRUE, ...)
```

## Arguments

| | |
|---|---|
| directory | Directory containing raw actigraphy files. |
| outdir | Directory in which processed RDS files are saved. Created if needed. |
| verbose | Logical; report progress for each file. Failures are always reported with a warning. |
| ... | Arguments passed to acti_process(). |

### Value

Invisibly, a character vector of paths to the saved RDS files. Files that failed are listed in its `failed` attribute: a named character vector of error messages, named by input file. Each RDS contains the processed data, with the raw reader header attached as the `header` attribute when available.

---

## `acti_sleep_diary` *Identify a diary-defined sleep window*

---

### Description

Identify a diary-defined sleep window

### Usage

```
acti_sleep_diary(data, onset, wakeup)
```

### Arguments

| | |
|---|---|
| `data` | A data frame containing timestamps. |
| `onset, wakeup` | POSIXct values defining a diary sleep window. |

### Value

An `acti_sleep_guider` object. Missing diary times return an empty window rather than imputing either boundary.

### Examples

```
data <- actimetrics::acti_count_data
diary_window <- acti_sleep_diary(
  data, onset = data$time[5], wakeup = data$time[30]
)
```

---

## `acti_sleep_ensemble` *Run a sleep-label ensemble from raw or epoch data*

---

### Description

This convenience workflow detects or accepts sustained inactivity bouts, runs compatible sleep guiders, creates one sleep label per guider, then produces a consensus. Raw triaxial data use `HDCZA`, `HorAngle`, and a fixed 22:00–08:00 window by default; `L5` is included only when the data span its requested duration. Epoch data use `NotWorn` and the fixed window, adding `L5` when possible. Set `guiders` to choose a subset explicitly.

## Usage

```
acti_sleep_ensemble(
  data,
  guiders = NULL,
  epoch = "5 seconds",
  sib_method = "vanHees2015",
  fusion = c("staple", "majority"),
  sleepwindow_type = c("SPT", "TimeInBed"),
  ...
)
```

## Arguments

| | |
|---|---|
| data | Raw triaxial or regular epoch-level data. Epoch data should have a sib column or a conventional activity column. |
| guiders | Guider names. Defaults depend on whether raw acceleration is available. |
| epoch | Raw-data aggregation epoch. |
| sib_method | SIB method, passed to acti_sleep_sib(). |
| fusion | Consensus rule: "staple" or "majority". |
| sleepwindow_type | SPT or TimeInBed overlap rule. |
| ... | Passed to the individual guider functions. |

## Value

A list of time-indexed data. epoch_data includes the van Hees sib; each guider has a label_data element with time and window; labels has time plus one logical sleep_<guider> column per guider; fused has time, probability, and consensus label; and label_data combines all these analysis columns.

## Examples

```
raw <- actiread::acti_read_gt3x(actiread::acti_example_gt3x(), verbose = FALSE)
result <- acti_sleep_ensemble(raw)
```

---

# acti_sleep_fuse_labels

*Fuse multiple sleep/wake labels with STAPLE*

---

## Description

STAPLE estimates a latent consensus sleep label and each input labeler's sensitivity and specificity using expectation-maximisation.

### Usage

```
acti_sleep_fuse_labels(
  labels,
  threshold = 0.5,
  tolerance = 1e-06,
  max_iterations = 200L
)
```

### Arguments

| | |
|---|---|
| labels | A data frame, matrix, or tibble of logical/binary sleep-label columns, such as the output of acti_sleep_labels(). |
| threshold | Posterior sleep-probability threshold for the fused label. |
| tolerance, max_iterations | STAPLE convergence controls. |

### Value

A tibble with sleep_probability and binary sleep columns. If labels includes time or epoch, that index is retained. The fitted STAPLE parameters are attached as the staple attribute.

### Examples

```
labels <- data.frame(
  sleep_l5 = c(FALSE, TRUE, TRUE, FALSE),
  sleep_fixed = c(FALSE, TRUE, FALSE, FALSE)
)
acti_sleep_fuse_labels(labels)
```

---

## acti_sleep_guider *Dispatch a sleep-period-time guider from data*

---

### Description

For "HDCZA" and "HorAngle", supply raw triaxial acceleration; the selected function derives a regular epoch-level posture angle. The remaining methods use regular epoch-level activity, SIB, or timestamp data. Use acti_sleep_guider_requirements() for a method-by-method summary.

### Usage

```
acti_sleep_guider(
  data,
 method = c("HDCZA", "HorAngle", "L5", "setwindow", "HLRB", "NotWorn", "sleeplog"),
  ...
)
```

### Arguments

| | |
|---|---|
| data | A raw or epoch-level data frame, according to method. |
| method | One of "HDCZA", "HorAngle", "L5", "setwindow", "HLRB", "NotWorn", or "sleeplog". |
| ... | Method-specific arguments, such as threshold, start_hour, onset, and wakeup. |

### Value

An acti_sleep_guider object.

### Examples

```
data <- actimetrics::acti_count_data
data <- data[rep(seq_len(nrow(data)), length.out = 24 * 60), ]
data$time <- data$time[1] + (seq_len(nrow(data)) - 1) * 60
guider <- acti_sleep_guider(data, method = "L5")
```

---

## acti_sleep_guider_requirements

*Describe data requirements for sleep guiders*

---

### Description

HDCZA and HorAngle take raw triaxial acceleration and create their own regular epoch-level angle. All other guiders take a regularly spaced, epoch-level data frame and do not require raw acceleration data.

### Usage

```
acti_sleep_guider_requirements()
```

### Value

A tibble describing the required data for each guider.

### Examples

```
acti_sleep_guider_requirements()
```

`acti_sleep_hdcza` *Identify a sleep window with HDCZA*

## Description

Implements the HDCZA sleep-period-time guider for wrist accelerometry. It finds prolonged periods of little change in the supplied z-axis angle, joins interruptions shorter than an hour, and returns the longest resulting block. The default threshold is the 0.2 degrees specified in the GGIR guider documentation.

## Usage

```
acti_sleep_hdcza(
  data,
  epoch = "5 seconds",
  threshold = 0.2,
  ignore_invalid = FALSE,
  min_block_minutes = 30,
  max_gap_minutes = 60
)
```

## Arguments

| | |
|---|---|
| `data` | A data frame of raw accelerometer observations. |
| `epoch` | Epoch used to average raw angles, passed to `lubridate::floor_date()`. |
| `threshold` | Maximum five-minute rolling median absolute angle change. |
| `ignore_invalid` | Whether invalid observations are treated as movement; use `NA` to treat them as no movement. |
| `min_block_minutes` | Minimum low-movement block duration. |
| `max_gap_minutes` | Maximum interruption bridged between blocks. |

## Value

An `acti_sleep_guider` object. `window` is the selected SPT guider; `crude_window` retains all candidate blocks before choosing the longest.

## Examples

```
# The bundled actiread recording contains raw X, Y, and Z acceleration.
raw <- actiread::acti_read_gt3x(actiread::acti_example_gt3x(), verbose = FALSE)
hdcza <- acti_sleep_hdcza(raw, epoch = "5 seconds")
```

## acti_sleep_hlrb *Identify the longest rest bout from sustained inactivity bouts*

### Description

Identify the longest rest bout from sustained inactivity bouts

### Usage

```
acti_sleep_hlrb(data)
```

### Arguments

| | |
|---|---|
| data | A data frame of regular epochs. |

### Examples

```
data <- actimetrics::acti_count_data
data <- data[rep(seq_len(nrow(data)), length.out = 24 * 60), ]
data$time <- data$time[1] + (seq_len(nrow(data)) - 1) * 60
# In practice, use a validated SIB classification rather than this example.
sib <- data$counts == 0
data$sib <- data$counts == 0
longest_rest <- acti_sleep_hlrb(data)
```

## acti_sleep_horangle *Identify a sleep window from horizontal posture*

### Description

Identify a sleep window from horizontal posture

### Usage

```
acti_sleep_horangle(
  data,
  epoch = "5 seconds",
  longitudinal_axis = "Z",
  horizontal_threshold = 45,
  ignore_invalid = FALSE,
  min_block_minutes = 30,
  max_gap_minutes = 60
)
```

### Arguments

| | |
|---|---|
| `data` | A data frame of raw accelerometer observations. |
| `epoch` | Epoch used to average raw angles, passed to `lubridate::floor_date()`. |
| `longitudinal_axis` | Raw axis used as the longitudinal axis: `"X"`, `"Y"`, or `"Z"`. |
| `horizontal_threshold` | Absolute-angle limit defining horizontal posture. |
| `ignore_invalid` | Whether invalid observations are treated as movement; use `NA` to treat them as no movement. |
| `min_block_minutes` | Minimum low-movement block duration. |
| `max_gap_minutes` | Maximum interruption bridged between blocks. |

### Examples

```
raw <- actiread::acti_read_gt3x(actiread::acti_example_gt3x(), verbose = FALSE)
horizontal <- acti_sleep_horangle(raw, epoch = "5 seconds")
```

---

## `acti_sleep_l5` *Identify a sleep window from the least-active five hours*

---

### Description

Identify a sleep window from the least-active five hours

### Usage

```
acti_sleep_l5(data, l5_hours = 5, window_hours = 12)
```

### Arguments

| | |
|---|---|
| `data` | A data frame of regular activity epochs. |
| `l5_hours` | Duration of the least-active period. |
| `window_hours` | Duration of the guider centred on L5. |

### Examples

```
# Repeat the bundled minute-level count data to form a complete example day.
data <- actimetrics::acti_count_data
data <- data[rep(seq_len(nrow(data)), length.out = 24 * 60), ]
data$time <- data$time[1] + (seq_len(nrow(data)) - 1) * 60
l5 <- acti_sleep_l5(data)
```

## acti_sleep_label *Convert a sleep guider to epoch-level sleep/wake labels*

### Description

A guider defines the likely main sleep window; it is not a sleep/wake classifier itself. This function applies GGIR's final overlap step to a sustained-inactivity-bout (`sib`) classification. With the default `"SPT"` rule, an entire SIB is sleep when any part overlaps the guider. With `"TimeInBed"`, an SIB must be wholly inside the guider.

### Usage

```
acti_sleep_label(sib, guider, sleepwindow_type = c("SPT", "TimeInBed"))
```

### Arguments

| | |
|---|---|
| `sib` | Logical or 0/1 sustained-inactivity-bout classification at the same regular epochs as `guider$window`. |
| `guider` | An `acti_sleep_guider` object. |
| `sleepwindow_type` | Whether the guider approximates sleep-period time (`"SPT"`) or time in bed (`"TimeInBed"`). |

### Value

A logical vector: `TRUE` for sleep and `FALSE` for wake.

### Examples

```
data <- actimetrics::acti_count_data
data <- data[rep(seq_len(nrow(data)), length.out = 24 * 60), ]
data$time <- data$time[1] + (seq_len(nrow(data)) - 1) * 60
guider <- acti_sleep_l5(data)
sib <- data$counts == 0
sleep <- acti_sleep_label(sib, guider)
```

## acti_sleep_labels *Create sleep/wake labels for multiple guiders*

### Description

Create sleep/wake labels for multiple guiders

### Usage

```
acti_sleep_labels(sib, guiders, sleepwindow_type = c("SPT", "TimeInBed"))
```

### Arguments

| | |
|---|---|
| sib | Logical or 0/1 sustained-inactivity-bout classification. |
| guiders | A named list of acti_sleep_guider objects. All guider windows must use the same epoch grid as sib. |
| sleepwindow_type | The SPT or TimeInBed overlap rule. |

### Value

A tibble with one logical sleep_<guider> column per guider.

### Examples

```
data <- actimetrics::acti_count_data
data <- data[rep(seq_len(nrow(data)), length.out = 24 * 60), ]
data$time <- data$time[1] + (seq_len(nrow(data)) - 1) * 60
guiders <- list(L5 = acti_sleep_l5(data), fixed = acti_sleep_setwindow(data))
labels <- acti_sleep_labels(data$counts == 0, guiders)
```

---

## acti_sleep_notworn *Identify a low-activity window for no-night-wear protocols*

---

### Description

Identify a low-activity window for no-night-wear protocols

### Usage

```
acti_sleep_notworn(data, min_block_minutes = 30, max_gap_minutes = 60)
```

### Arguments

| | |
|---|---|
| data | A data frame of regular activity epochs. |
| min_block_minutes | Minimum duration of a low-activity block. |
| max_gap_minutes | Maximum interruption bridged between blocks. |

### Examples

```
data <- actimetrics::acti_count_data
data <- data[rep(seq_len(nrow(data)), length.out = 24 * 60), ]
data$time <- data$time[1] + (seq_len(nrow(data)) - 1) * 60
# `counts` is an epoch-level activity metric.
no_night_wear <- acti_sleep_notworn(data)
```

## acti_sleep_setwindow *Identify a fixed daily sleep window*

### Description

Identify a fixed daily sleep window

### Usage

```
acti_sleep_setwindow(data, start_hour = 22, end_hour = 8)
```

### Arguments

| | |
|---|---|
| `data` | A data frame containing timestamps. |
| `start_hour, end_hour` | Start and end clock hours (0–24). |

### Value

An `acti_sleep_guider` object.

### Examples

```
data <- actimetrics::acti_count_data
fixed_window <- acti_sleep_setwindow(data, start_hour = 22, end_hour = 8)
```

## acti_sleep_sib *Detect sustained inactivity bouts (SIBs)*

### Description

For raw triaxial accelerometer data, the default `"vanHees2015"` method derives a longitudinal-axis angle at epoch resolution and identifies periods with no posture change above `angle_threshold`. For epoch-level data, provide a precomputed `sib` column; alternatively, `"low_activity"` marks epochs whose conventional activity metric is at or below `activity_threshold`.

### Usage

```
acti_sleep_sib(
  data,
  method = c("vanHees2015", "low_activity"),
  epoch = "5 seconds",
  time_threshold_minutes = 5,
  angle_threshold = 5,
  activity_threshold = 0
)
```

### Arguments

| | |
|---|---|
| data | Raw triaxial or regular epoch-level data. |
| method | SIB method: "vanHees2015" or "low_activity". |
| epoch | Raw-data aggregation epoch. |
| time_threshold_minutes, angle_threshold | van Hees posture-change parameters. |
| activity_threshold | Low-activity cutoff for epoch data. |

### Value

A tibble with time, sib, and supporting epoch-level variables.

### Examples

```
raw <- actiread::acti_read_gt3x(actiread::acti_example_gt3x(), verbose = FALSE)
sib <- acti_sleep_sib(raw)
```

---

## acti_threshold_counts *Classify activity counts using intensity thresholds*

---

### Description

Applies count thresholds to a count-data set after verifying that its timestamps have the requested, regular epoch. Thresholds must be expressed in counts per epoch seconds. Published cut-points should only be used at the epoch and placement for which they were developed.

### Usage

```
acti_threshold_counts(
  data,
  thresholds,
  labels,
  name = "activity_intensity",
  epoch = 60
)
```

### Arguments

| | |
|---|---|
| data | A data frame with time and counts columns. |
| thresholds | Increasing numeric lower bounds for successive intensity categories, expressed in counts per epoch seconds. |
| labels | Labels for the categories. There must be one more label than threshold. |
| name | Name of the factor column to add to data. |
| epoch | Required epoch, in seconds. |

## Value

`data` with an ordered factor column named by `name`.

## Examples

```
counts <- data.frame(
  time = as.POSIXct("2020-01-01", tz = "UTC") + 60 * 0:2,
  counts = c(0, 3000, 5000)
)
acti_threshold_counts(
  counts,
  thresholds = c(2860, 3941),
  labels = c("sedentary", "light", "mvpa")
)
```

---

# acti_threshold_enmo *Classify ENMO using activity-intensity thresholds*

---

## Description

Applies thresholds to Euclidean Norm Minus One (ENMO) values after checking the regular epoch in `time`. ENMO values must be in milli-g (mg), as is conventional for published ENMO cut-points. The output of `acti_calculate_enmo()` is in g; multiply `ENMO_t` by 1000 before using this function.

## Usage

```
acti_threshold_enmo(
  data,
  thresholds,
  labels,
  name = "activity_intensity",
  epoch,
  enmo = "enmo"
)
```

## Arguments

| | |
|---|---|
| `data` | A data frame with a `time` and an ENMO column. |
| `thresholds` | Increasing numeric lower bounds for successive intensity categories, in mg. |
| `labels` | Labels for the categories. There must be one more label than threshold. |
| `name` | Name of the factor column to add to `data`. |
| `epoch` | Required epoch, in seconds. |
| `enmo` | Name of the ENMO column, in mg. |

## Value

`data` with an ordered factor column named by `name`.

`acti_threshold_hildebrand` *Classify adult ENMO with Hildebrand cut-points*

## Description

Classifies one-second, raw ActiGraph GT3X+ ENMO values in mg from adults as light-or-less, moderate, or vigorous physical activity. The source used a 60-Hz device at the right hip or non-dominant wrist; select the matching placement and do not apply these adult thresholds to children.

## Usage

```
acti_threshold_hildebrand(
  data,
  placement = c("wrist", "hip"),
  name = "hildebrand_intensity",
  enmo = "enmo"
)

acti_threshold_hildebrand_wrist(..., placement = "wrist")

acti_threshold_hildebrand_hip(..., placement = "hip")
```

## Arguments

| | |
|---|---|
| `data` | A data frame with a `time` and an ENMO column. |
| `placement` | Body placement used in the original calibration: `"hip"` or `"wrist"`. |
| `name` | Name of the added ordered factor column. |
| `enmo` | Name of the ENMO column, in mg. |
| `...` | not used, but used to pass arguments to acti_threshold_hildebrand from acti_threshold_hildebrand_wrist and acti_threshold_hildebrand_hip |

## Details

Hildebrand et al. report lower bounds of 69.1 and 258.7 mg at the hip, and 100.6 and 428.8 mg at the wrist, for moderate and vigorous activity, respectively doi:10.1249/MSS.0000000000000289.

## Value

`data` with an ordered factor column named by `name`.

## References

Hildebrand M, van Hees VT, Hansen BH, Ekelund U. Age group comparability of raw accelerometer output from wrist- and hip-worn monitors. *Medicine & Science in Sports & Exercise*. 2014;46(9):1816-1824. doi:10.1249/MSS.0000000000000289.

`acti_threshold_montoye` *Classify wrist ActiGraph counts with Montoye cut-points*

## Description

Classifies 60-second ActiGraph vector-magnitude counts from a non-dominant wrist as sedentary, light activity, or moderate-to-vigorous physical activity (MVPA). These cut-points were developed in free-living adults with a GT9X Link and should not be used for another placement, device, or epoch.

## Usage

```
acti_threshold_montoye(data, name = "montoye_intensity")
```

## Arguments

| | |
|---|---|
| `data` | A data frame with `time` and `counts` columns. |
| `name` | Name of the added ordered factor column. |

## Details

Montoye et al. recommend `<2860` counts/minute for sedentary activity, `2860–3940` for light activity, and `>=3941` for MVPA doi:10.1080/02640414.2020.1794244.

## Value

`data` with an ordered factor column named by `name`.

## References

Montoye AH, Clevenger KA, Pfeiffer KA, et al. Development of cut-points for determining activity intensity from a wrist-worn ActiGraph accelerometer in free-living adults. *Journal of Sports Sciences*. 2020;38(22):2569-2578. doi:10.1080/02640414.2020.1794244.

`acti_threshold_romanzini` *Classify adolescent hip ActiGraph counts with Romanzini cut-points*

## Description

Classifies 15-second ActiGraph GT3X vector-magnitude counts from an adolescent hip-worn monitor as sedentary, light, moderate, or vigorous activity. The cut-points must not be rescaled to another epoch.

### Usage

```
acti_threshold_romanzini(data, name = "romanzini_intensity")
```

### Arguments

| | |
|---|---|
| data | A data frame with time and counts columns. |
| name | Name of the added ordered factor column. |

### Details

Romanzini et al. recommend <=180 counts/15 seconds for sedentary, 181–756 for light, 757–1111 for moderate, and >=1112 for vigorous activity doi:10.1080/17461391.2012.732614.

### Value

data with an ordered factor column named by name.

### References

Romanzini M, Petroski EL, Ohara D, Dourado AC, Reichert FF. Calibration of ActiGraph GT3X, Actical and RT3 accelerometers in adolescents. *European Journal of Sport Science*. 2014;14(1):91-99. doi:10.1080/17461391.2012.732614.

---

## acti_threshold_sasaki *Classify hip ActiGraph counts with Sasaki cut-points*

---

### Description

Classifies 60-second ActiGraph vector-magnitude counts from a hip-worn GT3X as less-than-moderate activity, moderate, hard, or very hard activity. The source provides no sedentary/light boundary, so the first category is deliberately named light_or_less. Do not use these hip-specific cut-points for wrist data.

### Usage

```
acti_threshold_sasaki(data, name = "sasaki_intensity")
```

### Arguments

| | |
|---|---|
| data | A data frame with time and counts columns. |
| name | Name of the added ordered factor column. |

### Details

Sasaki et al. recommend 2690–6166 counts/minute for moderate, 6167–9642 for hard, and >9642 for very hard activity doi:10.1016/j.jsams.2011.04.003.

## Value

`data` with an ordered factor column named by `name`.

## References

Sasaki JE, John D, Freedson PS. Validation and comparison of ActiGraph activity monitors. *Journal of Science and Medicine in Sport*. 2011;14(5):411-416. doi:10.1016/j.jsams.2011.04.003.

---

`mims_default_extrapolation`

*Default MIMS worker functions*

---

## Description

Default MIMS worker functions

## Usage

```
mims_default_extrapolation(data, dynamic_range = NULL)

mims_default_interpolation(data)

mims_default_filtering(data)
```

## Arguments

| | |
|---|---|
| `data` | data set of data, usually time and X/Y/Z. Usually from `actiread::acti_read_gt3x()` |
| `dynamic_range` | dynamic range of the data. Will be passed to `actibase::get_dynamic_range()` |

## Value

A data set of data

---

`mims_default_processing`

*Default MIMS Pre-processing*

---

## Description

Default MIMS Pre-processing

### Usage

```
mims_default_processing(
  data,
  use_extrapolation = TRUE,
  use_filtering = TRUE,
  verbose = TRUE,
  dynamic_range = NULL,
  round_after_processing = FALSE
)
```

### Arguments

| | |
|---|---|
| data | Data set of raw accelerometry values, usually time and X/Y/Z. Usually from `actiread::acti_read_gt3x()` |
| use_extrapolation | If `TRUE` the function will apply extrapolation algorithm to the input signal, otherwise it will skip extrapolation but only linearly interpolate the signal to 100Hz. |
| use_filtering | If `TRUE` the function will apply bandpass filtering to the input signal, otherwise it will skip the filtering. |
| verbose | print diagnostic messages |
| dynamic_range | the dynamic ranges of the input signal. Passed to `actimetrics::mims_default_extrapolation()`. Only needed if `use_extrapolation = TRUE` |
| round_after_processing | Should the result be rounded to 3 decimal values after processing, to make similar to standard accelerometry? |

### Value

A processed data set

---

## py_acti_calculate_forest

*Perform step count calculation in a separate Python environment*

---

### Description

Perform step count calculation in a separate Python environment

### Usage

```
py_acti_calculate_forest(
  ...,
  pyenv_function = function() {
     reticulate::import("forest")
 },
  show = FALSE
)
```

### Arguments

| | |
|---|---|
| ... | arguments passed to acti_calculate_forest() |
| pyenv_function | function that loads the forest Python package. By default, it uses reticulate::py_import("forest") to import the package. If this function has an args argument, the output of pyenv_function will be re-assigned to args. |
| show | Logical, whether to show the standard output on the screen while the child process is running, passed to callr::r() |

### Value

The output from acti_calculate_forest().

### Examples

```
Sys.setenv("SSQ_PARALLEL" = 0)
if (reticulate::py_module_available("forest")) {
  data = actiread::acti_read_gt3x(actiread::acti_example_gt3x())
  steps = py_acti_calculate_forest(data, sample_rate = 100)
}
```

---

## py_acti_calculate_stepcount

*Perform step count calculation in a separate Python environment*

---

### Description

Perform step count calculation in a separate Python environment

### Usage

```
py_acti_calculate_stepcount(
  ...,
  pyenv_function = function() {
     stepcount::py_require_stepcount()
 },
  show = FALSE
)
```

### Arguments

| | |
|---|---|
| ... | arguments passed to acti_calculate_stepcount() |
| pyenv_function | function that loads the stepcount Python package. By default, it uses stepcount::py_require_stepcou to set up the package. If this function has an args argument, the output of pyenv_function will be re-assigned to args. |
| show | Logical, whether to show the standard output on the screen while the child process is running, passed to callr::r() |

## Value

The output from acti_calculate_stepcount(). A tibble with minute-level time, steps, and walking columns.

## Examples

```
if (stepcount::have_stepcount()) {
  data = actiread::acti_read_gt3x(actiread::acti_example_gt3x())
  steps = py_acti_calculate_stepcount(data, sample_rate = 100)
}
```

# Index


∗ **datasets**
acti_count_data, 19

acti_apply_cole_kripke
(acti_calculate_counts), 6
acti_apply_cole_kripke(), *3*
acti_apply_sadeh
(acti_calculate_counts), 6
acti_apply_tudor_locke, 3
acti_apply_tudor_locke(), *3*
acti_apply_tudor_locke_diary, 3
acti_apply_tudor_locke_diary(), *3*
acti_calculate_activity_index
(acti_calculate_ai), 4
acti_calculate_ai, 4
acti_calculate_ai_defined
(acti_calculate_measures), 8
acti_calculate_auc
(acti_calculate_measures), 8
acti_calculate_counts, 6, *20*
acti_calculate_enmo
(acti_calculate_measures), 8
acti_calculate_enmo(), *33*
acti_calculate_fast_mims
(acti_calculate_measures), 8
acti_calculate_flags
(acti_calculate_measures), 8
acti_calculate_forest, 8
acti_calculate_forest(), *39*
acti_calculate_mad
(acti_calculate_measures), 8
acti_calculate_mad(), *5*
acti_calculate_measures, 8
acti_calculate_mims
(acti_calculate_measures), 8
acti_calculate_n_idle
(acti_calculate_measures), 8
acti_calculate_nonwear
(acti_calculate_counts), 6
acti_calculate_sdt, 11
acti_calculate_stepcount, 12
acti_calculate_stepcount(), *39, 40*
acti_calculate_verisense, 13
acti_calculate_wear
(acti_calculate_counts), 6
acti_calculate_wear(), *13*
acti_calculate_wear_raw, 13
acti_calculate_wear_raw(), *14–16*, *18*
acti_calculate_wear_raw_actipy, 14
acti_calculate_wear_raw_actipy(), *13,*
*14*, *18*
acti_calculate_wear_raw_ggir, 16
acti_calculate_wear_raw_ggir(), *13–15*
acti_calibrate, 18
acti_calibrate(), *14–16*, *20*
acti_count_data, 19
acti_process, 20
acti_process(), *21*
acti_process_study, 21
acti_sleep_diary, 22
acti_sleep_ensemble, 22
acti_sleep_fuse_labels, 23
acti_sleep_guider, 24
acti_sleep_guider_requirements, 25
acti_sleep_hdcza, 26
acti_sleep_hlrb, 27
acti_sleep_horangle, 27
acti_sleep_l5, 28
acti_sleep_label, 29
acti_sleep_labels, 29
acti_sleep_notworn, 30
acti_sleep_setwindow, 31
acti_sleep_sib, 31
acti_threshold_counts, 32
acti_threshold_enmo, 33
acti_threshold_hildebrand, *34*, 34
acti_threshold_hildebrand_hip, *34*
acti_threshold_hildebrand_hip
(acti_threshold_hildebrand), 34

acti_threshold_hildebrand_wrist, *34*
acti_threshold_hildebrand_wrist
(acti_threshold_hildebrand), 34
acti_threshold_montoye, 35
acti_threshold_romanzini, 35
acti_threshold_sasaki, 36
actibase::acti_raw_data, *20*
actibase::acti_resample, *6*
actibase::acti_standardize_epoch(), *6*, *12*
actiread::acti_read_gt3x(), *21*
agcounts::calculate_counts, *6, 7*, *20*

base::round(), *7*

callr::r(), *39*

lubridate::floor_date(), *5*, *26*, *28*

mims_default_extrapolation, 37
mims_default_filtering
(mims_default_extrapolation), 37
mims_default_interpolation
(mims_default_extrapolation), 37
mims_default_processing, 37

py_acti_calculate_forest, 38
py_acti_calculate_stepcount, 39

stepcount::stepcount(), *12*

walking::estimate_steps_verisense(), *13*
walking::find_walking(), *8*
walking::sdt_count_steps(), *11*