Package 'gmsp'

July 18, 2026

Title Ground Motion Signal Processing

Version 0.7.1

Description Implements short-time Fourier transform (STFT) based processing of strong-motion time series: time-grid regularisation, STFT-window and anti-alias-resampling strategy selection, edge tapering, and frequency-domain integration and differentiation, mapping a single input (acceleration, velocity, or displacement) to a consistent triplet under a chosen analysis bandwidth. Also provides intrinsic-mode-function decomposition via empirical mode decomposition (EMD), ensemble EMD (EEMD), and variational mode decomposition (VMD) with optional band-rule filtering; elastic single-degree-of-freedom (SDOF) domain-normalised acceleration, velocity, and displacement response spectra from their corresponding time-series domains by exact state-space integration; intensity measures including peak, root-mean-square (RMS), Arias intensity, significant-duration, cumulative absolute velocity, mean period, and the derived indices earthquake destructiveness potential (EPI) and power-of-input (PDI); and D50 and D100 horizontal response spectra. Methods: Huang et al. (1998) <doi:10.1098/rspa.1998.0193>, Wu and Huang (2009) <doi:10.1142/S1793536909000047>, Dragomiretskiy and Zosso (2014) <doi:10.1109/TSP.2013.2288675>, Boore (2010) <doi:10.1785/0120090179>. An optional indexing layer parses provider files in formats including 'PEER' 'NGA-West2' 'AT2', 'CESMD' 'V2'/'V2c', 'NWZ' 'V2A', Geological Survey of Canada 'TR', 'IGP'/'UCR' 'AC' variants, and generic two-column ASCII text, normalises components, writes per-record CSV (comma-separated values) and JSON (JavaScript Object Notation) pairs, and computes per-record intensity tables.

License MIT + file LICENSE

Encoding UTF-8

Language en-US

URL https://averriK.github.io/gmsp/

Depends R (>= 4.1.0)

Imports data.table, digest, EMD, expm, hht, jsonlite, openssl, pracma, purrr, seewave, signal, spectral, stats, stringr, utils, VMDecomp

Suggests knitr, Matrix, osqp, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Alejandro Verri Kozlowski [aut, cre, cph] (ORCID: <https://orcid.org/0000-0002-8535-1170>)

Maintainer Alejandro Verri Kozlowski <averri@fi.uba.ar>

Repository CRAN

Date/Publication 2026-07-18 16:40:02 UTC

Contents

`alignComponents` *Equalize NP across components of a parsed record.*

Description

Pads shorter OCIDs with trailing zeros (align = "max") or truncates longer OCIDs (align = "min") so all components share the same NP. Operates on a classified LONG table with columns `t`, `OCID`, `s`, and `DIR`.

Usage

```
alignComponents(DT, align = "max")
```

Arguments

`DT`	LONG `data.table(t, OCID, s, DIR)` for ONE record.
`align`	`"max"` (default) or `"min"`.

Value

Named list with two elements:

- `DT` : aligned LONG `data.table` with the same columns as the input.
- `NP` : integer, final number of samples per component after alignment (the same for every OCID). When every OCID already shares the same NP the function returns without padding or truncation; the input is passed through as `DT` unchanged.

Examples

```
x <- data.table::rbindlist(list(
  data.table::data.table(t = c(0, 0.01, 0.02), OCID = "H1",
                         DIR = "H1", s = c(1, 2, 3)),
  data.table::data.table(t = c(0, 0.01), OCID = "H2",
                         DIR = "H2", s = c(1, 2)),
  data.table::data.table(t = c(0, 0.01, 0.02), OCID = "UP",
                         DIR = "UP", s = c(0, 1, 0))
))
aligned <- alignComponents(x, align = "max")
aligned$NP
```

---

archiveRawOwner *Compress* raw.owner/ *to* raw.owner.tar.gz *and delete the directory.*

Description

Saves disk space after the parser has succeeded. Returns silently if nothing to do (no raw.owner/ directory) or if the archive already exists. Only deletes raw.owner/ after the tar archive is verified (re-readable). On any failure, leaves both intact.

Usage

```
archiveRawOwner(path)
```

Arguments

path	Absolute path to the station folder containing raw.owner/.

Value

Logical, TRUE if archive newly written, FALSE if no-op, NA on failure.

Examples

```
station <- file.path(tempdir(), "gmsp-archive-example")
unlink(station, recursive = TRUE)
dir.create(file.path(station, "raw.owner"), recursive = TRUE)
writeLines("provider bytes", file.path(station, "raw.owner", "record.txt"))
archiveRawOwner(station)
file.exists(file.path(station, "raw.owner.tar.gz"))
```

AT2TS *Convert acceleration time series into AT/VT/DT bundles*

Description

End-to-end workflow that takes acceleration time histories and produces a consistent set of acceleration, velocity, and displacement time series. It optionally regularizes sampling, converts units (for raw data), selects optimal STFT parameters and resampling strategy, applies robust edge tapering, performs spectral-domain integration, and provides post-tapering/optional trimming.

The function is designed for seismic/structural records but is agnostic to the physical origin provided `Fmax` reflects the analysis band of interest.

Usage

```
AT2TS(
  .x,
  units.source,
  time = "t",
  Fmax = 16,
  kNyq = 3.125,
  resample = TRUE,
  units.target = "mm",
  NW = 128,
  OVLP = 75,
  flatZeros = FALSE,
  Astop0 = 0.001,
  Apass0 = 0.001,
  AstopLP = 0.001,
  ApassLP = 0.98,
  trimZeros = FALSE,
  detrend = FALSE,
  regularize = FALSE,
  output = "TSL",
  verbose = FALSE,
  audit = TRUE,
  isRaw = TRUE
)
```

Arguments

`.x`	data.table. Input acceleration records with a time column and one or more signal columns (e.g., H1, H2, V).
`units.source`	character. Source units for input acceleration when `isRaw = TRUE`. Supported: "mm", "cm", "m", "gal", "g". If different from `units.target`, a scale factor is applied per channel.
`time`	character. Name of the time column in the input (default `"t"`). Internally and in `TSL` output, time is canonicalized to `t`.

Fmax	numeric. Maximum frequency of interest (Hz). Used to set STFT strategy and low-pass regularization in integration. Default: 16.
kNyq	numeric. Target Nyquist multiplier (Fs_target ~= kNyq * Fmax) if the user forces it. If not provided, an automatic grid is searched. Default: 3.125.
resample	logical. Kept for compatibility; the actual decision is made by the internal STFT strategy based on Fmax and constraints. Default: TRUE.
units.target	character. Output target units for acceleration records. Default: "mm".
NW	integer. Nominal STFT window length (samples); may be adjusted by the strategy. Default: 128.
OVLP	numeric. Window overlap percent. Default: 75.
flatZeros	logical. If TRUE, flatten external low-amplitude tails to exact zeros using per-channel normalized acceleration amplitude before optional trimming. If isRaw = TRUE, the historical smooth taper is still applied even when flatZeros = FALSE. Default: FALSE.
Astop0	numeric. Normalized stop threshold 0..1 for taper/flatten; relative to per-channel max amplitude. Astop0 = 1e-3 is equivalent to threshold 10 after normalizing PGA to 10000. Default: 1e-3.
Apass0	numeric. Normalized pass threshold 0..1 for taper/flatten; relative to per-channel max amplitude. Default: 1e-3.
AstopLP	numeric. Stopband attenuation for anti-alias LP (resampling). Default: 1e-3.
ApassLP	numeric. Passband for anti-alias LP (resampling). Default: 0.98.
trimZeros	logical. If TRUE, trims rows outside the final nonzero window support after taper/flatten. Multi-component records use the union of component supports. Default: FALSE.
detrend	logical. Remove mean before/after each main stage. Default: FALSE.
regularize	logical. Force time regularization of input if needed. Default: FALSE.
output	character. Short-circuit outputs (default: "TSL"): "ATo": early wide-frame after units; "AT"/"VT"/"DT": final wide; "TSW": combined wide; "TSL": long table.
verbose	logical. Print diagnostic logs. Default: FALSE.
audit	logical. If TRUE, runs auditSTFT() to validate STFT/resampling strategy and emit warnings for risky configurations. Default: TRUE.
isRaw	logical. If TRUE, perform unit conversion and robust pre/post tapering by default. If FALSE, the input is assumed to be already in units.target and the units.source argument is silently overridden to match; no scale factor is applied. Default: TRUE.

Value

Returns the requested object based on output (no other element is returned alongside it):

- "ATo": wide table with ts (time starting at 0), Units and channels, before any tapering or integration.
- "AT" / "VT" / "DT": wide table with the channels only (no ts column).

- "TSW": wide table with columns ts, AT.<OCID>, VT.<OCID>, DT.<OCID>.
- "TSL" (default): long table with columns t, s, ID (one of "AT", "VT", "DT"), and OCID. Sampling interval and row count are not returned separately; recover them from the output via 1 / diff(ts)[1] and nrow(.).

Examples

```
t <- seq(0, 2, by = 0.02)
x <- data.table::data.table(
  t = t,
  H1 = sin(2 * pi * t),
  H2 = 0.5 * cos(2 * pi * t),
  UP = 0.25 * sin(4 * pi * t)
)
tsl <- AT2TS(x, units.source = "mm", Fmax = 4, NW = 16,
             audit = FALSE, isRaw = FALSE)
head(tsl)
```

---

auditDistances *Audit distances in a record metadata table.*

Description

Flags each row with the FIRST applicable reason from a fixed precedence: lat/lon NA, lat/lon out of range, depth negative, Repi above outlier threshold, Rhyp < Repi (geometric impossibility). Rows with no issue are dropped from the output.

Usage

```
auditDistances(DT, repiOutlier = 5000)
```

Arguments

DT	record metadata data.table.
repiOutlier	threshold in km above which Repi is flagged (default 5000).

Details

Does NOT read record.json or provider flatfiles. Comparison against raw/flatfile distances is deferred to v2.

Value

data.table of flagged rows with column Reason.

Examples

```
x <- data.table::data.table(
  EventLatitude = c(0, 95),
  EventLongitude = c(0, 0),
  StationLatitude = c(0.1, 0.1),
  StationLongitude = c(0.1, 0.1),
  EventDepth = c(10, 10),
  Repi = c(15, 20),
  Rhyp = c(18, 25)
)
auditDistances(x)
```

---

auditParsers *Audit parsers: dry-run* parseRecord *on every record of an owner.*

Description

Iterates a record metadata subset for owner, calls parseRecord per unique (EventID, StationID), catches errors, and returns a status table.

Usage

```
auditParsers(.x, owner, path)
```

Arguments

.x	record metadata data.table.
owner	one OwnerID string.
path	Absolute path to the records root passed through to parseRecord(). Required – no default.

Value

data.table(OwnerID, EventID, StationID, status, reason).

Examples

```
root <- file.path(tempdir(), "gmsp-auditparsers-example")
unlink(root, recursive = TRUE)
raw <- file.path(root, "ESM", "E1", "S1", "raw.owner")
dir.create(raw, recursive = TRUE)
writeLines(c("0 1", "0.01 2", "0.02 3"), file.path(raw, "N_acc.txt"))
writeLines(c("0 2", "0.01 3", "0.02 4"), file.path(raw, "E_acc.txt"))
writeLines(c("0 0", "0.01 1", "0.02 0"), file.path(raw, "Z_acc.txt"))
rows <- data.table::data.table(
  OwnerID = "ESM", EventID = "E1", StationID = "S1",
  FileID = c("N_acc.txt", "E_acc.txt", "Z_acc.txt")
```

```
)
auditParsers(rows, owner = "ESM", path = root)
```

`auditSite` *Audit site / station information in a record metadata table.*

Description

Flags rows with the FIRST applicable reason: `StationVs30` NA, below low cutoff, or above high cutoff. Coord checks live in `auditDistances`.

Usage

```
auditSite(DT, vs30Low = 50, vs30High = 3000)
```

Arguments

`DT`	record metadata `data.table`.
`vs30Low`	lower physical cutoff for `StationVs30` (m/s).
`vs30High`	upper physical cutoff (m/s).

Value

`data.table` of flagged rows with column `Reason`.

Examples

```
x <- data.table::data.table(StationVs30 = c(30, 760, NA))
auditSite(x)
```

`buildRawFileTable` *Build the per-owner RawFileTable CSV (provider file inventory, post-archive safe).*

Description

For each station, reads `raw.owner/record.json` and emits one row per provider file (one per ComponentID x FileID). Handles both states:

- `raw.owner/record.json` present on disk (not yet archived).
- `raw.owner.tar.gz` archive (extracts record.json via streaming `tar -xzOf` to avoid touching disk).

Usage

```
buildRawFileTable(path.records, path.index, owners = NULL)
```

Arguments

path.records	Absolute path to the records root. Required – no default.
path.index	Absolute path to the index root where per-owner CSVs are written. Required – no default.
owners	Character vector of OwnerIDs. NULL = scan all.

Details

Schema:

```
OwnerID, EventID, StationID, ComponentID, FileID,
  NP, dt, Fs, Units, HP, LP, isArray
```

PGA is intentionally not emitted here. Pre-parse PGAs from record.json are in heterogeneous provider units; canonical post-parse PGAs (in mm/s^2) live in RawIntensityTable.<Owner>.csv. Missing provider fields in the canonical schema are emitted as typed NA columns instead of changing the output schema.

isArray = nComponentID > 3 (heuristic per legacy convention).

Value

Invisibly, the per-owner row counts.

Examples

```
root <- file.path(tempdir(), "gmsp-raw-file-example")
index <- file.path(tempdir(), "gmsp-raw-file-index")
unlink(c(root, index), recursive = TRUE)
dir.create(file.path(root, "AAA", "E1", "S1", "raw.owner"),
           recursive = TRUE)
dir.create(index)
record <- list(
  Event = list(EventID = "E1"),
  Station = list(StationID = "S1"),
  Record = list(
    list(ComponentID = "H1", FileID = "H1.txt", NP = 4, dt = 0.01,
         Fs = 100, Units = "cm", HP = NA, LP = NA),
    list(ComponentID = "H2", FileID = "H2.txt", NP = 4, dt = 0.01,
         Fs = 100, Units = "cm", HP = NA, LP = NA),
    list(ComponentID = "UP", FileID = "UP.txt", NP = 4, dt = 0.01,
         Fs = 100, Units = "cm", HP = NA, LP = NA)
  )
)
jsonlite::write_json(
  record,
```

```
  file.path(root, "AAA", "E1", "S1", "raw.owner", "record.json"),
  auto_unbox = TRUE
)
suppressMessages(buildRawFileTable(root, index, owners = "AAA"))
data.table::fread(file.path(index, "RawFileTable.AAA.csv"))
```

`buildRawIntensityTable` *Build the canonical RawIntensityTable for one or more owners (WIDE).*

Description

For each station with `raw/AT.<RID>.csv` / `.json`, calls `getRawIntensities()` to compute the 20 AT-derivable scalars per direction via `getIntensity()`. Emits one row per `(RecordID, DIR)` - three rows per record.

Usage

```
buildRawIntensityTable(
  path.records,
  path.index,
  owners = NULL,
  incremental = TRUE,
  force = FALSE
)
```

Arguments

`path.records`	Absolute path to the records root. Required – no default.
`path.index`	Absolute path to the index root where per-owner CSVs are written. Required – no default.
`owners`	Character vector of `OwnerIDs`. `NULL` = scan all.
`incremental`	Logical. If `TRUE`, skip records already present in `RawIntensityTable.<O>.csv`. Default: `TRUE`.
`force`	Logical. If `TRUE`, ignore incremental cache and recompute everything. Default: `FALSE`.

Details

Schema:

```
RecordID, DIR, OCID,
  AI, AId, AIu, ARMS, ATn, ATo, AZC, CAV, CAV5,
  D0575, D0595, D2080, Dmax, EPI, Fs, NP, PDI,
  PGA, TmA, dt
```

All amplitude scalars are in TARGET_UNITS = "mm"-derived units (mm/s2, mm/s, etc. per getIntensity() contract).

Value

Invisibly, the per-owner row counts.

Examples

```
root <- file.path(tempdir(), "gmsp-raw-intensity-table-example")
index <- file.path(tempdir(), "gmsp-raw-intensity-table-index")
unlink(c(root, index), recursive = TRUE)
raw <- file.path(root, "AAA", "E1", "S1", "raw")
dir.create(raw, recursive = TRUE)
dir.create(index)
t <- seq(0, 1, by = 0.01)
data.table::fwrite(
  data.table::data.table(
    H1 = sin(2 * pi * t),
    H2 = 0.5 * cos(2 * pi * t),
    UP = 0.25 * sin(4 * pi * t)
  ),
  file.path(raw, "AT.R1.csv")
)
jsonlite::write_json(
  list(RecordID = "R1", OwnerID = "AAA", EventID = "E1",
       StationID = "S1", NetworkID = "NW",
       DIR = c("H1", "H2", "UP"), OCID = c("H1", "H2", "UP"),
       NP = rep(length(t), 3), dt = 0.01, Fs = 100, Units = "mm"),
  file.path(raw, "AT.R1.json"), auto_unbox = TRUE
)
suppressMessages(buildRawIntensityTable(root, index, owners = "AAA",
                                        incremental = FALSE))
data.table::fread(file.path(index, "RawIntensityTable.AAA.csv"))
```

---

buildRawRecordTable *Build the canonical RawRecordTable for one or more owners.*

Description

Scans <path.records>/<OwnerID>/<EventID>/<StationID>/raw/AT.*.json and emits one row per RecordID to <path.index>/RawRecordTable.<OwnerID>.csv.

Usage

```
buildRawRecordTable(path.records, path.index, owners = NULL)
```

Arguments

path.records	Absolute path to the records root. Required – no default.
path.index	Absolute path to the index root where per-owner CSVs are written. Required – no default.
owners	Character vector of OwnerIDs. NULL = scan all subdirs of path.records.

Details

Schema:

```
RecordID, EventID, StationID, OwnerID, NP, Fs, pad
```

where NP = max(json$NP) (post-align) and pad = max(json$NP) - min(json$NP).

This table carries zero per-direction metadata: directional detail (PGA, AI, ARMS, ...) lives in RawIntensityTable.<Owner>.csv. The per-layer Raw* prefix leaves room for downstream processors to emit their own <ProcessID>RecordTable.<Owner>.csv.

Value

Invisibly, the per-owner row counts.

Examples

```
root <- file.path(tempdir(), "gmsp-raw-record-example")
index <- file.path(tempdir(), "gmsp-raw-record-index")
unlink(c(root, index), recursive = TRUE)
raw <- file.path(root, "AAA", "E1", "S1", "raw")
dir.create(raw, recursive = TRUE)
dir.create(index)
jsonlite::write_json(
  list(RecordID = "R1", OwnerID = "AAA", EventID = "E1",
       StationID = "S1", NP = c(4, 4, 4), Fs = 100),
  file.path(raw, "AT.R1.json"), auto_unbox = TRUE
)
suppressMessages(buildRawRecordTable(root, index, owners = "AAA"))
data.table::fread(file.path(index, "RawRecordTable.AAA.csv"))
```

DT2TS *Convert displacement time series into AT/VT/DT bundles*

Description

End-to-end workflow that takes displacement time histories and produces a consistent set of velocity and acceleration, along with the displacement processed outputs. It regularizes sampling if needed, converts units (for raw data), chooses STFT parameters/resampling, applies robust edge tapering, performs spectral/time derivatives, and applies post-tapering/optional trimming.

Usage

```
DT2TS(
  .x,
  units.source,
  time = "t",
  Fmax = 16,
  kNyq = 3.125,
  resample = TRUE,
  derivate = "freq",
  units.target = "mm",
  NW = 128,
  OVLP = 75,
  flatZeros = FALSE,
  Astop0 = 1e-04,
  Apass0 = 0.001,
  AstopLP = 0.001,
  ApassLP = 0.98,
  trimZeros = FALSE,
  detrend = FALSE,
  regularize = FALSE,
  output = "TSL",
  verbose = FALSE,
  audit = TRUE,
  isRaw = TRUE,
  lowPass = TRUE
)
```

Arguments

`.x`	data.table. Input displacement records with a time column and one or more signal columns.
`units.source`	character. Source units for the input displacement when `isRaw = TRUE`. Same set as `AT2TS` / `VT2TS`: `"mm"`, `"cm"`, `"m"`, plus `"gal"` (treated as cm scale) and `"g"` (multiplied by `g_mms2`). Practical displacement records are virtually always in `"mm"`, `"cm"` or `"m"`; the acceleration-flavoured entries are accepted for symmetry with `.getSF()` but are unusual here. If different from `units.target`, a scale factor is applied per channel.
`time`	character. Name of the time column in the input (default `"t"`). Internally and in `TSL` output, time is canonicalized to `t`.
`Fmax`	numeric. Maximum frequency of interest (Hz). Guides STFT strategy and low-pass regularization during integration.
`kNyq`	numeric. Target Nyquist multiplier (`Fs_target ~= kNyq * Fmax`) when forced by the user. Otherwise an automatic grid is searched.
`resample`	logical. Kept for compatibility; decision is made by the internal STFT strategy.
`derivate`	character. Derivative method for `DT -> VT / AT` (`"time"` or `"freq"`).
`units.target`	character. Target units for acceleration-related outputs.

NW	integer. Nominal STFT window length (samples). May be adjusted.
OVLP	numeric. Window overlap percent.
flatZeros	logical. Apply edge tapering; if isRaw = TRUE, tapering is applied regardless.
Astop0, Apass0	numeric. Normalized thresholds 0..1 for taper/flatten; relative to the per-channel max amplitude.
AstopLP, ApassLP	numeric. Anti-alias LP specs for resampling.
trimZeros	logical. If TRUE, trims leading/trailing zeros by the final window.
detrend	logical. Remove mean before/after stages.
regularize	logical. Force time regularization of input if needed.
output	character. Early/short-circuit outputs (default: "TSL"): "DTo", "AT", "VT", "DT", "TSW", "TSL".
verbose	logical. Print diagnostic logs.
audit	logical. If TRUE, runs auditSTFT() to validate STFT/resampling strategy and emit warnings for risky configurations. Default: TRUE.
isRaw	logical. If TRUE, handle unit conversion and default tapering.
lowPass	logical. If TRUE, multiply the spectral derivative kernel by an additional Butterworth-like low-pass at Fmax to suppress high-frequency amplification of the numerical derivative. Default: TRUE.

Value

Returns the requested object based on output.

Examples

```
t <- seq(0, 2, by = 0.02)
x <- data.table::data.table(
  t = t,
  H1 = sin(2 * pi * t),
  H2 = 0.5 * cos(2 * pi * t),
  UP = 0.25 * sin(4 * pi * t)
)
tsl <- DT2TS(x, units.source = "mm", Fmax = 4, NW = 16,
             audit = FALSE, isRaw = FALSE)
head(tsl)
```

extractRecord *Extract one record to* raw/<KIND>.<RecordID>.csv + <KIND>.<RecordID>.json.

Description

Pipeline: parseRecord -> mapComponents(rotate = FALSE) -> capture pre-align NP and DIR/OCID mapping -> alignComponents -> pivot WIDE by provider OCID -> md5-16 hash -> write. The function returns NULL and writes nothing in three skip paths:

1. Component classification cannot map the record (arrays with > 3 OCIDs, 2-component records, or records with no vertical channel in the vertical-component vocabulary).
2. The provider Units string cannot be normalised (.parseUnits returns NA_character_, so the scale factor K is NULL).
3. kind is left at its default NULL and .parseKind(Key$Units) returns NA_character_ (kind cannot be derived from Units).

Usage

```
extractRecord(.x, path, align = "max", kind = NULL)
```

Arguments

.x	data.table subset for ONE record.
path	Absolute path to the records root. The function writes outputs under <path>/<OwnerID>/<EventID>/<St Required – no default.
align	"max" (pad to longest, default) or "min" (truncate).
kind	Optional "AT"|"VT"|"DT" override. When NULL (default) KIND is derived from Key$Units via .parseKind().

Details

KIND is one of "AT" (acceleration), "VT" (velocity), "DT" (displacement). By default it is derived from Key$Units via .parseKind(). Pass kind explicitly to override – e.g. kind = "VT" for blasting records whose Units may not be machine-parseable.

Old contents of raw/ are unlinked before writing (idempotent).

JSON sidecar schema:

```
RecordID, OwnerID, EventID, StationID, NetworkID,
FileID            (scalar = "<KIND>.<RID>.csv"),
DIR               (array, ["H1","H2","UP"]),
OCID              (array of 3, provider channels in DIR order),
NP                (array of 3, pre-align NP per DIR),
PGA / PGV / PGD   (array of 3, peak |s| per DIR, post-align;
                   field name derived from KIND[1] -> PGA/PGV/PGD),
dt, Fs, Units     (scalars).
```

Value

Absolute path to the written <KIND>.<RecordID>.csv, or NULL (skip).

Examples

```
root <- file.path(tempdir(), "gmsp-extract-example")
unlink(root, recursive = TRUE)
raw <- file.path(root, "ESM", "E1", "S1", "raw.owner")
dir.create(raw, recursive = TRUE)
writeLines(c("0 1", "0.01 2", "0.02 3"), file.path(raw, "N_acc.txt"))
writeLines(c("0 2", "0.01 3", "0.02 4"), file.path(raw, "E_acc.txt"))
writeLines(c("0 0", "0.01 1", "0.02 0"), file.path(raw, "Z_acc.txt"))
rows <- data.table::data.table(
  OwnerID = "ESM", EventID = "E1", StationID = "S1",
  NetworkID = "NW", Units = "cm",
  FileID = c("N_acc.txt", "E_acc.txt", "Z_acc.txt")
)
path <- extractRecord(rows, path = root, kind = "AT")
basename(path)
```

---

fitModalFactor *Fit the modal shaping factors of one record*

Description

Per-record modal shaping of the spectral-match workflow: released modal amplitudes are fitted against a target response-spectrum band (shape.modalFactor in the batch JSON contract), with energy restoration applied after the modal objective.

Usage

```
fitModalFactor(
  .x,
  target,
  source,
  key,
  ocid = "H1",
  xi = 0.05,
  fmax = 25,
  kNyq = 4,
  factorMin,
  factorMax = Inf,
  remove = NULL,
  includeResidue = TRUE,
  unitsSource = "mm",
  unitsTarget = "mm"
)
```

Arguments

.x	data.table. IMF rows for the suite (or the record): RecordID, OwnerID, EventID, StationID, OCID, t, signal, IMF* mode columns, and optionally residue. Rows are filtered to key's record and ocid.
target	data.table. Target band with columns Tn, Sa.low, Sa.mean, Sa.high; must include Tn == 0 (PGA) and positive periods, in the PSA units of .x.
source	data.table. Source TSW rows for the suite (columns RecordID, t, AT.<ocid>, ...) used for the source spectra.
key	data.table. One row carrying RecordID, OwnerID, EventID, StationID of the record to fit.
ocid	character. Component to shape ("H1", "H2", "UP").
xi	numeric. Damping ratio.
fmax	numeric. Analysis bandwidth (Hz).
kNyq	numeric. Nyquist multiple.
factorMin	numeric. Lower bound of the modal factors (shape.modalFactor.min), in (0, 1].
factorMax	numeric. Upper bound of the modal factors (shape.modalFactor.max); must contain 1. Default Inf.
remove	character or integer. IMF modes excluded from the reconstruction (imf.remove).
includeResidue	logical. Keep the residue in the reconstruction.
unitsSource	character. Units of .x signals.
unitsTarget	character. Units of the shaped products.

Value

list with TSW (shaped time histories of the fitted component), PSW (shaped spectra), PSA (long table with kind = source/start/final curves), coeff (per-mode factors: b, energyFactor, stageFactor), metrics (one-row fit diagnostics), and eval (objective evaluations).

Examples

```
## Not run:
Fit <- fitModalFactor(.x = IMF, target = Band, source = TSW,
                      key = Selection[1L], factorMin = 0.001)

## End(Not run)
```

`fitRecordFactor` *Fit the suite record-scaling factors*

Description

Suite-level record scaling of the spectral-match workflow: one scale coefficient per record (`shape.recordFactor` in the batch JSON contract), solved as a bounded quadratic program against the target band (global band-conditioned objective).

Usage

```
fitRecordFactor(
  .x,
  target,
  records,
  factorMin = 0,
  factorMax = Inf,
  bandWeight = 2,
  deficitWeight = 1
)
```

Arguments

`.x`	data.table. Suite PSA long table with columns `RecordID`, `Tn`, `Sa`, `kind`; the `kind == "final"` curves are scaled.
`target`	data.table. Target band with columns `Tn`, `Sa.low`, `Sa.mean`, `Sa.high`, in the same units as `.x`.
`records`	character. RecordIDs of the completed records to scale (at least two).
`factorMin`	numeric. Lower bound of the record factors (`shape.recordFactor.min`). Default `0`.
`factorMax`	numeric. Upper bound of the record factors (`shape.recordFactor.max`). Default `Inf`.
`bandWeight`	numeric. Weight of the band-violation term. Default `2`.
`deficitWeight`	numeric. Extra weight of the suite-mean deficit below the target. Default `1`.

Value

list with `scale` (data.table `RecordID`, `scaleFactor`) and `stage` (one-row solver diagnostics; the `stage` column keeps the frozen product label `"B"`).

Examples

```
## Not run:
Suite <- fitRecordFactor(.x = PSA, target = Band,
                         records = Metrics[status == "done", RecordID],
                         factorMin = 0.5, factorMax = 1.6)
```

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

---

getIntensity *Compute intensity measures from a long time-series table.*

Description

`getIntensity()` is a compatibility wrapper around `TSL2IM()`. It accepts canonical long `TSL` input only; convert wide `TSW` input with `TSW2TSL()` first.

Usage

```
getIntensity(.x, units.source, units.target = "mm", output = c("IML", "IMW"))
```

Arguments

`.x`	`data.table`. Long-format TSL with at least `OCID`, `ID`, `t`, `s` columns plus optional record-identifier metadata columns.
`units.source`	character. Source units of the `s` column (`"mm"`, `"cm"`, `"m"`, `"gal"`, `"g"`).
`units.target`	character. Target units for the returned intensities. Default `"mm"`.
`output`	character. `"IML"` returns long intensity rows; `"IMW"` returns one row per metadata and `OCID`, with intensity measures as columns.

Value

See `TSL2IM()`.

Examples

```
t <- seq(0, 1, by = 0.01)
tsl <- data.table::data.table(t = t, s = sin(2 * pi * t),
                              ID = "AT", OCID = "H1")
getIntensity(tsl, units.source = "mm")
```

getRawIntensities *Compute KIND-derivable intensities for one raw record.*

Description

Reads raw/<KIND>.<RID>.csv (WIDE provider OCID columns) plus raw/<KIND>.<RID>.json, builds a long TSL with ID set to the file's KIND (AT, VT, or DT), and calls getIntensity() to compute the per-direction intensity scalars.

Usage

```
getRawIntensities(path)
```

Arguments

path	Absolute path to a station folder containing raw/.

Details

Output (WIDE): one row per (RecordID, DIR) with the KIND-derivable intensity columns.

DIR (H1/H2/UP) is recovered from the JSON sidecar's explicit DIR/OCID mapping. The CSV column names remain provider OCID values such as HHE, HNZ, or L/T/V.

Assumes signal already in TARGET_UNITS = "mm" (per extractRecord).

Value

Wide data.table with intensity columns (e.g. RecordID, DIR, OCID, AI, AIu, AId, PGA, ARMS, AZC, ATo, ATn, D059
for AT inputs). Returns NULL if the station has no raw/<KIND>.<RID>.csv / .json.

Examples

```
station <- file.path(tempdir(), "gmsp-raw-intensity-example")
unlink(station, recursive = TRUE)
raw <- file.path(station, "raw")
dir.create(raw, recursive = TRUE)
t <- seq(0, 1, by = 0.01)
data.table::fwrite(
  data.table::data.table(
    H1 = sin(2 * pi * t),
    H2 = 0.5 * cos(2 * pi * t),
    UP = 0.25 * sin(4 * pi * t)
  ),
  file.path(raw, "AT.R1.csv")
)
jsonlite::write_json(
  list(RecordID = "R1", OwnerID = "AAA", EventID = "E1",
       StationID = "S1", NetworkID = "NW",
       DIR = c("H1", "H2", "UP"), OCID = c("H1", "H2", "UP"),
```

```
      NP = rep(length(t), 3), dt = 0.01, Fs = 100, Units = "mm"),
  file.path(raw, "AT.R1.json"), auto_unbox = TRUE
)
getRawIntensities(station)
```

IML2IMW *Convert long intensity tables to wide form.*

Description

IML2IMW() casts long intensity output from TSL2IM() / getIntensity() to one row per metadata and OCID, with intensity measures as columns.

Usage

```
IML2IMW(.x, by = "auto")
```

Arguments

.x	Long intensity table with columns OCID, IM, and value.
by	Metadata columns to keep as row keys. The default "auto" uses all columns except OCID, ID, IM, value, and units.

Value

A wide intensity data.table.

Examples

```
iml <- data.table::data.table(RecordID = "R1", OCID = "H1",
                              ID = "AT", IM = "PGA",
                              value = 1, units = "mm /s2")
IML2IMW(iml)
```

---

mapComponents *Map provider components to canonical processed components.*

Description

Classifies a single three-component record, preserves provider channel names in OCID, and records canonical processed directions in DIR (H1, H2, UP). This helper is for processed products; raw extraction continues to preserve provider OCID values.

Usage

```
mapComponents(DT, rotate = TRUE, output = c("long", "wide"))
```

Arguments

DT	LONG data.table(t, OCID, s) or WIDE data.table(t, <OCID1>, <OCID2>, <OCID3>) for ONE record.
rotate	Logical scalar. If TRUE, rotate the two horizontal components to their principal axes and store attr(out, "theta").
output	"long" (default) returns provider OCID, canonical DIR, and signal s; "wide" returns data.table(t, H1, H2, UP).

Value

A data.table in the requested output shape, or NULL when the record cannot be mapped. Both shapes carry attr(out, "componentMap") and attr(out, "rotate"); rotated outputs also carry attr(out, "theta").

See Also

extractRecord()

Examples

```
t <- seq(0, 1, by = 0.1)
x <- data.table::rbindlist(list(
  data.table::data.table(t = t, OCID = "N", s = sin(2 * pi * t)),
  data.table::data.table(t = t, OCID = "E", s = 0.5 * cos(2 * pi * t)),
  data.table::data.table(t = t, OCID = "Z", s = 0.1 * sin(2 * pi * t))
))
mapped <- mapComponents(x, rotate = FALSE)
head(mapped)
```

---

matchCompatibility *Compatibility diagnostics of a matched suite*

Description

Record-level and suite-level compatibility of the scaled spectra against the target band: PGA position, band containment, and RMSE diagnostics per record, plus their suite aggregate.

Usage

```
matchCompatibility(
  .x,
  target,
  scale,
  ocid = "H1",
  factorMin = 0,
  factorMax = Inf
)
```

Arguments

.x	data.table. Final PSW rows (after record scaling) with the record key columns, Tn, and PSA.<ocid>.
target	data.table. Target band with columns Tn, Sa.low, Sa.mean, Sa.high; must include exactly one Tn == 0 row.
scale	data.table. Record scaling table (RecordID, scaleFactor) from fitRecordFactor().
ocid	character. Component evaluated ("H1", "H2", "UP").
factorMin	numeric. Lower record-factor bound used to flag saturated records. Default 0.
factorMax	numeric. Upper record-factor bound used to flag saturated records. Default Inf.

Value

list with record (per-record diagnostics, one row per record) and summary (one-row suite aggregate).

Examples

```
## Not run:
QA <- matchCompatibility(.x = PSW, target = Band, scale = Suite$scale,
                         factorMin = 0.5, factorMax = 1.6)

## End(Not run)
```

---

normalizeTS *Normalize a long time-series table to unit amplitude per channel.*

Description

Divides the signal column s by the peak amplitude of a reference quantity (norm) for every (metadata, OCID) group. The same scale factor is applied to all ID values within each group so that the physical relationship between AT, VT, and DT is preserved.

The default norm = "PGA" scales by 1 / max(abs(s)) computed from ID == "AT" rows, making max(abs(AT)) = 1 for every channel. The same SF is then applied to the corresponding VT and DT rows.

Modifies .x in place; returns .x[].

Usage

```
normalizeTS(.x, norm = "PGA")
```

Arguments

.x	Canonical TSL data.table with columns t, s, ID, and OCID, plus optional metadata columns.
norm	character. Reference quantity used to derive the scale factor. "PGA" (default) uses max(abs(AT)), "PPV" uses max(abs(VT)), "PGD" uses max(abs(DT)).

Value

.x with s scaled in place.

See Also

VT2TS(), mapComponents()

Examples

```
t <- seq(0, 1, by = 0.01)
tsl <- data.table::rbindlist(list(
  data.table::data.table(t = t, s = 2 * sin(2 * pi * t),
                         ID = "AT", OCID = "H1"),
  data.table::data.table(t = t, s = cos(2 * pi * t),
                         ID = "VT", OCID = "H1"),
  data.table::data.table(t = t, s = sin(pi * t),
                         ID = "DT", OCID = "H1")
))
normalizeTS(tsl)
max(abs(tsl[ID == "AT", s]))
```

---

parseRecord — *Parse one record (event x station x owner) into a LONG time-series table.*

Description

Reads <path>/<OwnerID>/<EventID>/<StationID>/raw.owner/ according to the owner's format and returns LONG (t, OCID, s). NPTS divergence between components is not enforced.

Usage

```
parseRecord(.x, path)
```

Arguments

.x	data.table subset of master for ONE record (same OwnerID, EventID, StationID).
path	Absolute path to the records root. The function expects per-station files under <path>/<OwnerID>/<EventID>/<StationID>/raw.owner/. Required – no default.

Details

Quantity (ID = "AT"|"VT"|"DT") is NOT set here.

Value

LONG data.table(t, OCID, s).

Examples

```
root <- file.path(tempdir(), "gmsp-parse-example")
unlink(root, recursive = TRUE)
raw <- file.path(root, "ESM", "E1", "S1", "raw.owner")
dir.create(raw, recursive = TRUE)
writeLines(c("0 1", "0.01 2", "0.02 3"), file.path(raw, "N_acc.txt"))
writeLines(c("0 2", "0.01 3", "0.02 4"), file.path(raw, "E_acc.txt"))
writeLines(c("0 0", "0.01 1", "0.02 0"), file.path(raw, "Z_acc.txt"))
rows <- data.table::data.table(
  OwnerID = "ESM", EventID = "E1", StationID = "S1",
  FileID = c("N_acc.txt", "E_acc.txt", "Z_acc.txt")
)
parseRecord(rows, path = root)
```

---

pipelineRunners *Run a gmsp batch pipeline*

Description

gmsp helpers such as AT2TS(), TSL2PS(), TS2IMF() and getIntensity() work directly on R objects. The run*() functions are optional batch runners for the same kind of work when the inputs and outputs are files controlled by a project JSON file.

Usage

```
runSelect(file = file.path("gmsp", "select.json"), root = getwd())

runGMSP(file = file.path("gmsp", "gmsp.json"), root = getwd())

runTrim(file = file.path("gmsp", "trim.json"), root = getwd())

runIMF(file = file.path("gmsp", "imf.json"), root = getwd())

runPSA(file = file.path("gmsp", "psa.json"), root = getwd())

runProcess(file = file.path("gmsp", "process.json"), root = getwd())

runStage0(file = file.path("gmsp", "match.json"), root = getwd())

runMatch(file = file.path("gmsp", "match.json"), root = getwd())

runPlot(file = file.path("gmsp", "plot.json"), root = getwd())

runExport(file = file.path("gmsp", "export.json"), root = getwd())
```

Arguments

file	character. Path to the pipeline JSON file.
root	character. Project root used to resolve relative JSON paths.

Details

Use the direct helpers for one record, one table, or interactive analysis. Use a runner when the same operation must be repeated over a declared selection of records and the output must be reproducible from a saved JSON contract.

Value

Invisibly returns the runner exit status.

Runner map

`runSelect()` Builds a record selection table. The helper-level contract is `selectRecords()` plus project-owned metadata.

`runStage0()` Builds a spectrally compatible record selection from the processed candidate pool: scores PSA against the target compatibility band declared in the match JSON, applies optional seismological screens (`candidate.pool`, `candidate.filter`, `candidate.exclude`, `candidate.oneRecordPerEvent`), and writes the top-`stage0.N` selection to `path.selection` together with the score table and a reusable spectra cache.

`runGMSP()` Processes raw acceleration, velocity, or displacement records into canonical time histories. The helper-level contracts are `AT2TS()`, `VT2TS()` and `DT2TS()`.

`runTrim()` Applies an approved time-windowing contract to existing time histories.

`runIMF()` Builds intrinsic-mode-function products from processed time histories. The helper-level contract is `TS2IMF()`.

`runPSA()` Builds elastic response spectra from processed time histories. The helper-level contract is `TSL2PS()`.

`runProcess()` Runs an approved composed processing contract for a project record set.

`runMatch()` Runs an approved spectral-match contract for a selected record set. This is an advanced batch workflow, not a replacement for the response-spectrum helpers.

`runPlot()` Builds optional raw QA/debug HTML widgets from existing products. It is not an SRK publication renderer.

`runExport()` Packages existing products, metadata, and approved audit files into a delivery folder.

JSON and paths

Each runner reads one JSON file. Paths inside that JSON are resolved relative to `root`, normally the project root. The JSON file is a project input, not a global package option.

Installed template JSON files live under `system.file("scripts", package = "gmsp")`. A project may keep simpler names such as `gmsp/select.json` or pass an explicit template-style path such as `gmsp/runSelect.json`.

Plotting and SRK reports

runPlot() writes raw self-contained QA/debug widgets. SRK client plots are rendered outside gmsp by the report layer from materialized gmsp CSV products. Do not use runPlot() output as final SRK report output.

Examples

```
## Not run:
# Direct helper use for one in-memory table:
TSL <- AT2TS(DT, units.source = "mm", Fmax = 25, output = "TSL")

# Batch equivalent for a project-controlled record set:
runGMSP(file = "gmsp/gmsp.json", root = ".")

# Compute spectra for an existing batch product:
runPSA(file = "gmsp/psa.json", root = ".")

## End(Not run)
```

PSL2PSW *Convert long response spectra to wide form.*

Description

PSL2PSW() casts canonical long spectra rows to wide columns such as PSA.H1, PSV.H1, and SD.H1.

Usage

```
PSL2PSW(.x, by = "auto")
```

Arguments

.x	Long spectra table with columns OCID, Tn, ID, and S.
by	Metadata columns to keep as row keys. The default "auto" uses all columns except OCID, Tn, ID, and S.

Value

A wide spectra data.table.

Examples

```
psl <- data.table::data.table(RecordID = "R1", OCID = "H1",
                              Tn = 0.1, ID = "PSA", S = 1)
PSL2PSW(psl)
```

PSW2PSL *Convert wide response spectra to long form.*

Description

PSW2PSL() melts spectra columns named <ID>.<OCID> back to canonical long spectra rows. Derived RotD components such as D50 and D100 are returned as ordinary OCID values.

Usage

```
PSW2PSL(.x, by = "auto", ids = c("PSA", "PSV", "SD"))
```

Arguments

.x	Wide spectra table with a Tn column and spectra columns named <ID>.<OCID>.
by	Metadata columns to keep as row keys. The default "auto" uses all non-spectra columns except Tn.
ids	Preferred order for spectra IDs. Other IDs present in .x are kept after these values.

Value

A canonical long spectra data.table.

Examples

```
psw <- data.table::data.table(RecordID = "R1", Tn = 0.1, PSA.H1 = 1)
PSW2PSL(psw)
```

readAC *Read a 3D-COL acceleration record (ACA, ACB, LIS).*

Description

One file holds 3 components as parallel columns:

- ACA (IGP Peru): cols Z N E after a header row.
- ACB (CISMID Peru): cols T EW NS UD; the time column T is dropped.
- LIS (UCR Costa Rica): cols N00E UPDO N90E after a ===DATA=== line.

Usage

```
readAC(file, type)
```

Arguments

file	Path to the file.
type	One of "ACA", "ACB", "LIS".

Details

OCIDs come from the file's column header line.

Value

LONG data.table(t, OCID, s).

Examples

```
file <- tempfile()
writeLines(c(
  "MUESTREO : 100",
  "  Z N E",
  "1 2 3",
  "4 5 6"
), file)
readAC(file, type = "ACA")
```

---

readAT *Read acceleration records via* readTS() *with* kind = "AT".

Description

Thin wrapper around readTS(); see there for full semantics.

Usage

```
readAT(.x, path)
```

Arguments

.x	data.table with columns RecordID, OwnerID, EventID, StationID (one row per record). Output of selectRecords().
path	Absolute path to the records root. The function reads per-station files under <path>/<OwnerID>/<EventID>/<StationID>/raw/. Required – no default.

Value

See readTS().

Examples

```
root <- file.path(tempdir(), "gmsp-readat-example")
unlink(root, recursive = TRUE)
raw <- file.path(root, "AAA", "E1", "S1", "raw")
dir.create(raw, recursive = TRUE)
data.table::fwrite(data.table::data.table(H1 = c(1, 2)),
                   file.path(raw, "AT.R1.csv"))
jsonlite::write_json(list(dt = 0.01), file.path(raw, "AT.R1.json"),
                     auto_unbox = TRUE)
selection <- data.table::data.table(
  RecordID = "R1", OwnerID = "AAA", EventID = "E1", StationID = "S1"
)
readAT(selection, path = root)
```

---

readAT2 *Read a PEER NGA-West2 AT2 acceleration record.*

Description

AT2 has a 4-line header ending with `NPTS=/DT=`. Line 2 holds the direction as the last comma-separated token (e.g., `Helena Montana-01, 10/31/1935, Carroll College, 180`). Body has up to 8 values per row in scientific notation; "stuck" negatives (`1.234-5.678`) are split before parsing. Truncated at NPTS.

Usage

```
readAT2(file)
```

Arguments

`file`	Path to the .AT2 file.

Value

LONG `data.table(t, OCID, s)`.

Examples

```
file <- tempfile(fileext = ".AT2")
writeLines(c(
  "header",
  "Event, date, station, H1",
  "units",
  "NPTS= 4, DT= 0.01 SEC",
  "1.0 2.0 3.0 4.0"
), file)
readAT2(file)
```

readDT *Read displacement records via* readTS() *with* kind = "DT".

Description

Thin wrapper around readTS(); see there for full semantics.

Usage

```
readDT(.x, path)
```

Arguments

.x	data.table with columns RecordID, OwnerID, EventID, StationID (one row per record). Output of selectRecords().
path	Absolute path to the records root. The function reads per-station files under <path>/<OwnerID>/<EventID>/<StationID>/raw/. Required – no default.

Value

See readTS().

Examples

```
root <- file.path(tempdir(), "gmsp-readdt-example")
unlink(root, recursive = TRUE)
raw <- file.path(root, "AAA", "E1", "S1", "raw")
dir.create(raw, recursive = TRUE)
data.table::fwrite(data.table::data.table(H1 = c(1, 2)),
                   file.path(raw, "DT.R1.csv"))
jsonlite::write_json(list(dt = 0.01), file.path(raw, "DT.R1.json"),
                     auto_unbox = TRUE)
selection <- data.table::data.table(
  RecordID = "R1", OwnerID = "AAA", EventID = "E1", StationID = "S1"
)
readDT(selection, path = root)
```

readISEE *Read a Micromate ISEE blasting record.*

Description

Parses the TXT output of an ISEE-compliant blasting seismograph (Micromate, Vibra-Tech, GeoSonics). The format is the *International Society of Explosives Engineers* (ISEE) Performance Specifications for Blasting Seismographs and is declared in the header line Version : V 10-90 Micromate ISEE.

Usage

```
readISEE(file)
```

Arguments

file	Path to the ISEE TXT.

Details

Two firmware variants are supported transparently:

- **v10 /** .TXT: header lines "<key> : <value>" (space-colon-space), body tab-separated Tran Vert Long MicL.
- **v11 /** .CSV: header lines "<key>","<value>" (CSV pair), body comma-separated quoted "Tran","Vert","Long" (no MicL present in the body).

Variant is auto-detected from the file content – the dispatch is not by extension. Sample rate is read from a line matching Sample\\s*Rate\\s*[: ,"]+<Fs>\\s*sps, which captures both header styles. The body header is located by the regex Tran[^A-Za-z]+Vert[^A-Za-z]+Long, also style-agnostic.

Column-to-OCID mapping (ISEE convention):

- Tran (transverse) -> T
- Vert (vertical) -> V
- Long (longitudinal, radial blast-to-station) -> L
- MicL (microphone, dB(L)) -> dropped when present (only in v10).

Velocity is in mm/s; dt = 1 / Fs is derived from the header. No metadata other than Fs is read here – per-event scalars (charge, distance to centroid, location) live in the blast flatfile, not in the parser's contract.

To process an ISEE record end-to-end:

```
DT <- readISEE("UM12780_23_10_2025_15_22_8.TXT")
# DT now has columns (t, OCID, s) with OCIDs T/V/L; s in mm/s
```

From parseRecord() (the canonical entry point), the dispatch happens via .OWNER_FORMAT["ISEE"] = "ISEE". To get a sidecar record (raw/VT.<RID>.csv + JSON), call extractRecord(.x, path) – the provider-string parser (.parseUnits / .parseKind) accepts "mm/s" on the master row and derives KIND = "VT" automatically, or pass kind = "VT" explicitly to bypass derivation.

Value

LONG data.table(t, OCID, s) with OCID in T/V/L and s in mm/s. Three rows per sample, 3 * NP rows total where NP is the per-channel sample count.

See Also

parseRecord(), extractRecord(), readAT2(), readV2(), readAC().

Examples

```
file <- tempfile(fileext = ".TXT")
writeLines(c(
  "Version : V 10-90 Micromate ISEE",
  "Sample Rate : 100 sps",
  "Tran\tVert\tLong\tMicL",
  "1\t2\t3\t90",
  "4\t5\t6\t91"
), file)
readISEE(file)
```

`readTR` *Read a TRA/TRZ/TRB/TRC acceleration record (GSC and SGC families).*

Description

TRA/TRZ have a multi-column body after `END_HEADER`; the last column is the corrected acceleration. TRB/TRC have a single-column body after the `Unidades:` (TRB) or `USER5` (TRC) line. OCID lives in the header: `Component:` (TRA/TRZ), `Componente:` (TRB), `STREAM:` (TRC).

Usage

```
readTR(file, type)
```

Arguments

`file`	Path to the file.
`type`	One of `"TRA"`, `"TRZ"`, `"TRB"`, `"TRC"`.

Value

LONG `data.table(t, OCID, s)`.

Examples

```
file <- tempfile()
writeLines(c(
  "Component: HNZ",
  "rate: 100",
  "END_HEADER",
  "skip1",
  "skip2",
  "1 2 3",
  "4 5 6"
), file)
readTR(file, type = "TRA")
```

readTS *Read parsed time-series records into the shape* AT2TS() / VT2TS() / DT2TS() *expect.*

Description

Stacks raw/<KIND>.<RID>.csv from each record in .x, builds the time axis t from dt (in raw/<KIND>.<RID>.json), and returns a single data.table keyed at (RecordID, OwnerID, EventID, StationID, t).

Usage

```
readTS(.x, path, kind = c("AT", "VT", "DT"))
```

Arguments

.x	data.table with columns RecordID, OwnerID, EventID, StationID (one row per record). Output of selectRecords().
path	Absolute path to the records root. The function reads per-station files under <path>/<OwnerID>/<EventID>/<StationID>/raw/. Required – no default.
kind	One of "AT", "VT", "DT". Selects the sidecar file prefix (<kind>.<RID>.csv / <kind>.<RID>.json).

Details

The sidecar shape produced by extractRecord() is identical across KINDs; KIND only selects the file prefix. The CSV columns are provider OCID values preserved by extractRecord(). Direction labels (H1/H2/UP) live in the JSON sidecar mapping, not in the CSV header. Use the thin wrappers readAT() / readVT() / readDT() at call sites where the KIND is fixed.

```
Selection <- selectRecords(M, EventID = "...")
TS  <- readTS(.x = Selection, path = "/path/to/records", kind = "VT")
# Sidecar declares units.source as the length base ("mm"); KIND is set by kind=.
TS[, VT2TS(.SD, units.source = "mm"), by = .(RecordID, OwnerID, EventID, StationID)]
```

Value

data.table with columns RecordID, OwnerID, EventID, StationID, t, <OCID columns>. Records whose sidecars are missing are skipped.

Examples

```
root <- file.path(tempdir(), "gmsp-readts-example")
unlink(root, recursive = TRUE)
raw <- file.path(root, "AAA", "E1", "S1", "raw")
dir.create(raw, recursive = TRUE)
data.table::fwrite(
  data.table::data.table(H1 = c(1, 2), H2 = c(0, 1)),
```

```
  file.path(raw, "AT.R1.csv")
)
jsonlite::write_json(list(dt = 0.01), file.path(raw, "AT.R1.json"),
                     auto_unbox = TRUE)
selection <- data.table::data.table(
  RecordID = "R1", OwnerID = "AAA", EventID = "E1", StationID = "S1"
)
readTS(selection, path = root, kind = "AT")
```

`readTwoCol` *Read a 2-column whitespace-delimited ASCII record.*

Description

Time in column 1 (seconds), signal in column 2. No header. Separator (space or tab) is auto-detected by `fread`. Trailing all-NA columns (some providers leave a trailing tab -> phantom column) are dropped.

Usage

```
readTwoCol(file)
```

Arguments

`file`	Path to the file.

Details

OCID extracted from filename. Three known patterns:

- SEED-like: `NET.STA.LOC.CHA__...` -> CHA
- CENA: `<date>_<time>_NET.STA.CHA_AccTH` -> CHA
- CLSMD: `<X>_acc.txt` -> X

Value

LONG `data.table(t, OCID, s)`.

Examples

```
dir <- tempfile()
dir.create(dir)
file <- file.path(dir, "N_acc.txt")
writeLines(c("0 1", "0.01 2"), file)
readTwoCol(file)
```

readV2 *Read a CESMD V2 acceleration record (multi-channel V2 or single-channel V2c).*

Description

Two CESMD variants:

- Multi-channel V2 (`.v2`): blocks marked `^Corrected accelerogram`, `8f10.6` body, ends at next `points of veloc data`. 1+ blocks.
- Single-channel V2c (`.V2c`): line 1 = `Corrected acceleration`, `1E15.6` body (1 col/row). Marker `acceleration pts` carries NPTS. `samples/sec` (last occurrence – DECIMATE > RESAMPLE) carries dt. Body ends at `End-of-data` or EOF. OCID from `Sta Chan ...:` line.

Usage

```
readV2(file)
```

Arguments

`file`	Path to the .v2 / .V2c file.

Value

LONG `data.table(t, OCID, s)`.

Examples

```
file <- tempfile(fileext = ".V2c")
writeLines(c(
  "Corrected acceleration",
  "Sta Chan 1: HNZ",
  "100 samples/sec",
  "4 acceleration pts approx 0.04 secs",
  "1", "2", "3", "4",
  "End-of-data"
), file)
readV2(file)
```

readV2A *Read a NWZ V2A acceleration record (3D-BLOCK, 1 file = 3 components).*

Description

Each component is a sequential block opened by `Corrected accelerogram` (case-insensitive). Within a block, header ends 10 lines after `Displacement:`. Body has 10 values per row in fixed-width form; "stuck" negatives are split before parsing. Each component vector is truncated at its own `Number of points` value.

Usage

```
readV2A(file)
```

Arguments

`file`	Path to the .V2A file.

Details

OCIDs come from `Component <X>` lines in each block.

Value

LONG `data.table(t, OCID, s)`.

Examples

```
file <- tempfile(fileext = ".V2A")
writeLines(c(
  "Corrected Accelerogram", "Component H1", "at 0.01 sec intervals",
  "Number of points 4", "Displacement:", rep("header", 10), "1 2 3 4",
  "Corrected Accelerogram", "Component H2",
  "Number of points 4", "Displacement:", rep("header", 10), "2 3 4 5",
  "Corrected Accelerogram", "Component UP",
  "Number of points 4", "Displacement:", rep("header", 10), "3 4 5 6"
), file)
readV2A(file)
```

readVT — *Read velocity records via* `readTS()` *with* `kind = "VT"`.

Description

Thin wrapper around `readTS()`; see there for full semantics.

Usage

```
readVT(.x, path)
```

Arguments

`.x`	`data.table` with columns `RecordID, OwnerID, EventID, StationID` (one row per record). Output of `selectRecords()`.
`path`	Absolute path to the records root. The function reads per-station files under `<path>/<OwnerID>/<EventID>/<StationID>/raw/`. Required – no default.

Value

See `readTS()`.

Examples

```
root <- file.path(tempdir(), "gmsp-readvt-example")
unlink(root, recursive = TRUE)
raw <- file.path(root, "AAA", "E1", "S1", "raw")
dir.create(raw, recursive = TRUE)
data.table::fwrite(data.table::data.table(H1 = c(1, 2)),
                   file.path(raw, "VT.R1.csv"))
jsonlite::write_json(list(dt = 0.01), file.path(raw, "VT.R1.json"),
                     auto_unbox = TRUE)
selection <- data.table::data.table(
  RecordID = "R1", OwnerID = "AAA", EventID = "E1", StationID = "S1"
)
readVT(selection, path = root)
```

selectRecords — *Select records from a record metadata table.*

Description

Filters a record metadata table by any combination of `RecordID`, `EventID`, `StationID`, and `OwnerID`, then deduplicates to one row per record. Output is the canonical selection shape consumed by the `readTS()` family (`readAT()` / `readVT()` / `readDT()`) and `writeSelection()`.

Usage

```
selectRecords(
  M,
  RecordID = NULL,
  EventID = NULL,
  StationID = NULL,
  OwnerID = NULL
)
```

Arguments

M	record metadata data.table.
RecordID	character. Filter by RecordID. Default NULL.
EventID	character. Filter by EventID. Default NULL.
StationID	character. Filter by StationID. Default NULL.
OwnerID	character. Filter by OwnerID. Default NULL.

Details

Filter args are character vectors (length 1+). NULL means "no restriction on this dimension". With all NULL, returns every record in M – protect with explicit filters for non-trivial work.

For richer filters (magnitude, distance, intensity), filter M first and pass the subset:

```
selectRecords(M[EventMagnitude > 7 & Repi < 100 & PGA > 600 & DIR == "H1"])
```

Value

data.table(RecordID, OwnerID, EventID, StationID).

Examples

```
master <- data.table::data.table(
  RecordID = c("R1", "R1", "R2"),
  OwnerID = c("AAA", "AAA", "BBB"),
  EventID = c("E1", "E1", "E2"),
  StationID = c("S1", "S1", "S2"),
  DIR = c("H1", "H2", "H1")
)
selectRecords(master, OwnerID = "AAA")
```

TS2IMF *Decompose one time series into intrinsic mode functions.*

Description

Orchestrates VMD/EMD/EEMD decomposition for one canonical t/s signal and returns either IMFs, a recomposed time series, or long/wide tables depending on output.

TS2IMF() is a worker. It does not detect or own record/component grouping. Use data.table grouping at the call site for TSL tables.

Usage

```
TS2IMF(
  .x,
  method = "vmd",
  K = 12,
  alpha = 2000,
  tau = 0,
  DC = TRUE,
  init = 0,
  tol = 1e-07,
  output = NULL,
  verbose = FALSE,
  boundary = "wave",
  stop.rule = "type5",
  noise.type = "gaussian",
  noise.amp = 5e-08,
  trials = 10,
  imf.remove = NULL,
  remove.Fo = NULL,
  keep.Fo = NULL,
  keep.Residue = TRUE
)
```

Arguments

.x	data.table with canonical t and s columns.
method	character. One of "vmd", "emd", "eemd" (default "vmd").
K	integer. Number of IMFs (default 12).
alpha, tau, DC, init, tol	numeric. Parameters for VMD.
output	character or NULL. One of "TSL", "TSW", "IMF" (default NULL).
verbose	logical. Engine verbosity (default FALSE).
boundary, stop.rule	character. EMD/EEMD parameters.

noise.type, noise.amp, trials
: parameters for EEMD.

imf.remove
: character or integer. IMF selection (optional). Character values remove explicit mode names such as "IMF1". Numeric values select IMF indices to remove: positive indices count from the first IMF, negative indices count from the last IMF, and both sets are unioned. For example, c(1L, -1L) removes IMF1 and the last IMF. Zero, non-finite, and out-of-range numeric values are ignored.

remove.Fo, keep.Fo
: numeric length-2 (Hz) frequency band rules (optional).

keep.Residue
: logical. If TRUE (default), include residue in the reconstruction of signal.R. If FALSE, signal.R is built only from the kept IMFs.

Value

Depending on output, returns TSL, TSW, IMF or a list with the decomposition.

Examples

```
t <- seq(0, 1, by = 0.01)
x <- data.table::data.table(
  t = t,
  s = sin(2 * pi * t) + 0.1 * sin(10 * pi * t)
)
imf <- TS2IMF(x, method = "vmd", K = 2, output = "IMF")
imf
```

TSL2AI *Arias-intensity ratio history from canonical acceleration time series.*

Description

TSL2AI() computes the cumulative Arias-intensity history from acceleration rows in a canonical TSL table and returns the relative value AI(t) / max(AI).

Usage

```
TSL2AI(.x, units.source, units.target = "mm", output = c("wide", "long"))
```

Arguments

.x
: data.table. Long-format TSL with at least OCID, ID, t, and s columns plus optional metadata columns. Only rows with ID == "AT" are used.

units.source
: character. Source length units of the s column ("mm", "cm", or "m"). Canonical TSL acceleration is interpreted as length units per second squared.

units.target
: character. Target length units used for intermediate calculations. Default "mm".

output
: character. "wide" returns one row per metadata and t, with one ratio column per OCID; "long" returns columns <metadata>, OCID, t, ratio.

Value

A data.table. Values are ratios in [0, 1] when the final cumulative value is positive; otherwise ratios are NA_real_.

Examples

```
t <- seq(0, 1, by = 0.01)
tsl <- data.table::data.table(RecordID = "R1", OCID = "H1",
                              ID = "AT", t = t,
                              s = 1000 * sin(2 * pi * t))
TSL2AI(tsl, units.source = "mm")
```

TSL2CAV *CAV ratio history from canonical acceleration time series.*

Description

TSL2CAV() computes the cumulative absolute velocity history from acceleration rows in a canonical TSL table and returns the relative value CAV(t) / max(CAV).

Usage

```
TSL2CAV(.x, units.source, units.target = "mm", output = c("wide", "long"))
```

Arguments

.x	data.table. Long-format TSL with at least OCID, ID, t, and s columns plus optional metadata columns. Only rows with ID == "AT" are used.
units.source	character. Source length units of the s column ("mm", "cm", or "m"). Canonical TSL acceleration is interpreted as length units per second squared.
units.target	character. Target length units used for intermediate calculations. Default "mm".
output	character. "wide" returns one row per metadata and t, with one ratio column per OCID; "long" returns columns <metadata>, OCID, t, ratio.

Value

See TSL2AI().

Examples

```
t <- seq(0, 1, by = 0.01)
tsl <- data.table::data.table(RecordID = "R1", OCID = "H1",
                              ID = "AT", t = t,
                              s = 1000 * sin(2 * pi * t))
TSL2CAV(tsl, units.source = "mm")
```

TSL2CAV5 *CAV5 ratio history from canonical acceleration time series.*

Description

TSL2CAV5() computes the cumulative absolute velocity ratio using only acceleration samples with absolute value at least 0.05 g.

Usage

```
TSL2CAV5(.x, units.source, units.target = "mm", output = c("wide", "long"))
```

Arguments

.x	data.table. Long-format TSL with at least OCID, ID, t, and s columns plus optional metadata columns. Only rows with ID == "AT" are used.
units.source	character. Source length units of the s column ("mm", "cm", or "m"). Canonical TSL acceleration is interpreted as length units per second squared.
units.target	character. Target length units used for intermediate calculations. Default "mm".
output	character. "wide" returns one row per metadata and t, with one ratio column per OCID; "long" returns columns <metadata>, OCID, t, ratio.

Value

See TSL2AI().

Examples

```
t <- seq(0, 1, by = 0.01)
tsl <- data.table::data.table(RecordID = "R1", OCID = "H1",
                              ID = "AT", t = t,
                              s = 1000 * sin(2 * pi * t))
TSL2CAV5(tsl, units.source = "mm")
```

TSL2IM *Compute intensity measures from a canonical long time-series table.*

Description

Given a long time-series table with columns for the record identifier, OCID, ID, t, and s, converts amplitudes to units.target and computes standard intensity measures grouped by record x OCID x ID. Valid ID content is either AT only, or a complete time-series set with AT, VT, and DT.

Usage

```
TSL2IM(.x, units.source, units.target = "mm", output = c("IML", "IMW"))
```

Arguments

.x	data.table. Long-format TSL with at least OCID, ID, t, s columns plus optional record-identifier metadata columns.
units.source	character. Source units of the s column ("mm", "cm", "m", "gal", "g").
units.target	character. Target units for the returned intensities. Default "mm".
output	character. "IML" returns long intensity rows; "IMW" returns one row per metadata and OCID, with intensity measures as columns.

Value

A data.table. output = "IML" returns long rows with columns <metadata cols>, OCID, ID, IM, value, units; output = "IMW" returns wide intensity columns.

Examples

```
t <- seq(0, 1, by = 0.01)
tsl <- data.table::rbindlist(list(
  data.table::data.table(t = t, s = sin(2 * pi * t),
                         ID = "AT", OCID = "H1"),
  data.table::data.table(t = t, s = cos(2 * pi * t),
                         ID = "VT", OCID = "H1"),
  data.table::data.table(t = t, s = sin(pi * t),
                         ID = "DT", OCID = "H1")
))
im <- TSL2IM(tsl, units.source = "mm")
head(im)
```

TSL2PS *Convert canonical long time series to response spectra.*

Description

TSL2PS() is the spectra helper for canonical TSL tables produced by AT2TS(), VT2TS(), and DT2TS(). It derives grouping keys from the TSL schema instead of exposing BY or column-name arguments.

Usage

```
TSL2PS(
  .x,
  xi = 0.05,
  Tn = NULL,
```

```
  output = "PSL",
  D50 = FALSE,
  D100 = FALSE,
  nTheta = 180L
)
```

Arguments

.x	Canonical TSL data.table with columns t, s, ID, and OCID, plus optional metadata columns.
xi	numeric. Damping ratio(s) 0..1. Scalar input preserves the historical output schema; vector input adds an xi column.
Tn	numeric vector. Natural periods in seconds. Must not include 0; the Tn = 0 peak-value anchor is prepended internally.
output	character. One of "PSL" or "PSW". Default "PSL".
D50	logical scalar. If TRUE, add the median rotated horizontal component as OCID = "D50" for every metadata group with H1 and H2.
D100	logical scalar. If TRUE, add the maximum rotated horizontal component as OCID = "D100" for every metadata group with H1 and H2.
nTheta	integer. Number of rotation angles in [0, 180) for D50 = TRUE or D100 = TRUE. Default 180L (1-degree step).

Details

For each source domain d in acceleration (AT), velocity (VT), and displacement (DT), the function solves z'' + 2 xi omega z' + omega^2 z = x_d independently and reports omega^2 max(abs(z)). The resulting PSA, PSV, and SD ordinates retain the units of AT, VT, and DT, respectively. They are domain-normalized gmsp spectra, not classical transformations of one acceleration-driven oscillator state. The internally prepended Tn = 0 rows are the matching peak-value anchors PGA, PGV, and PGD.

Value

A data.table.

- output = "PSL" returns a long table with metadata columns, OCID, Tn, spectral ID ("PSA", "PSV", "SD"), and S. If xi is a vector, the output also includes xi.
- output = "PSW" returns a wide table with metadata columns, Tn, and spectral component columns such as PSA.H1, PSV.H1, and SD.H1. If xi is a vector, the output also includes xi. If requested, D50 and D100 appear as ordinary component suffixes such as PSA.D50 and PSA.D100.

See Also

AT2TS(), VT2TS(), DT2TS()

Examples

```
t <- seq(0, 1, by = 0.01)
tsl <- data.table::rbindlist(list(
  data.table::data.table(t = t, s = sin(2 * pi * t),
                         ID = "AT", OCID = "H1"),
  data.table::data.table(t = t, s = cos(2 * pi * t),
                         ID = "VT", OCID = "H1"),
  data.table::data.table(t = t, s = sin(pi * t),
                         ID = "DT", OCID = "H1"),
  data.table::data.table(t = t, s = sin(2 * pi * t + 0.3),
                         ID = "AT", OCID = "H2"),
  data.table::data.table(t = t, s = cos(2 * pi * t + 0.4),
                         ID = "VT", OCID = "H2"),
  data.table::data.table(t = t, s = sin(pi * t + 0.2),
                         ID = "DT", OCID = "H2")
))
ps <- TSL2PS(tsl, Tn = c(0.1, 0.2))
head(ps)
rot <- TSL2PS(tsl, Tn = 0.1, D50 = TRUE, D100 = TRUE, nTheta = 6L)
rot[OCID %in% c("D50", "D100")]
```

TSL2TSW *Convert canonical long time-series tables to wide form.*

Description

`TSL2TSW()` casts a canonical long `TSL` table with `t`, `s`, `ID`, and `OCID` columns to wide `TSW` form with columns such as `AT.H1`, `VT.H1`, and `DT.H1`.

Usage

```
TSL2TSW(.x, by = "auto", ids = c("AT", "VT", "DT"))
```

Arguments

`.x`	Canonical `TSL` `data.table` with columns `t`, `s`, `ID`, and `OCID`, plus optional metadata columns.
`by`	Metadata columns to keep as row keys. The default `"auto"` uses all columns except `t`, `s`, `ID`, and `OCID`.
`ids`	Preferred order for signal IDs in the output columns. Other IDs present in `.x` are kept after these values.

Value

A wide `data.table` keyed by `<metadata>, t`.

Examples

```
tsl <- data.table::data.table(RecordID = "R1", OCID = "H1",
                              ID = "AT", t = c(0, 0.01), s = c(1, 2))
TSL2TSW(tsl)
```

TSW2TSL *Convert wide time-series tables to canonical long form.*

Description

`TSW2TSL()` melts wide time-series columns named `<ID>.<OCID>` back to canonical `TSL` rows.

Usage

```
TSW2TSL(.x, by = "auto", ids = c("AT", "VT", "DT"))
```

Arguments

`.x`	Wide `TSW` `data.table` with a `t` column, or a legacy constructor `ts` time column, plus signal columns named `<ID>.<OCID>`.
`by`	Metadata columns to keep as row keys. The default `"auto"` uses all columns except `t`, `s`, `ID`, and `OCID`.
`ids`	Preferred order for signal IDs in the output columns. Other IDs present in `.x` are kept after these values.

Value

A canonical long `data.table` with metadata columns, `OCID`, `ID`, `t`, and `s`.

Examples

```
tsw <- data.table::data.table(RecordID = "R1", t = c(0, 0.01),
                              AT.H1 = c(1, 2))
TSW2TSL(tsw)
```

VT2TS *Convert velocity time series into AT/VT/DT bundles*

Description

End-to-end workflow that takes velocity time histories and produces a consistent set of acceleration, velocity, and displacement time series. It optionally regularizes sampling, converts units (for raw data), selects optimal STFT parameters and resampling strategy, applies robust edge tapering, performs spectral-domain derivation and integration, yields post-tapering and optional trimming.

Usage

```
VT2TS(
  .x,
  units.source,
  time = "t",
  Fmax = 16,
  kNyq = 3.125,
  resample = TRUE,
  derivate = "freq",
  units.target = "mm",
  NW = 128,
  OVLP = 75,
  flatZeros = FALSE,
  Astop0 = 1e-04,
  Apass0 = 0.001,
  AstopLP = 0.001,
  ApassLP = 0.98,
  trimZeros = FALSE,
  detrend = FALSE,
  regularize = FALSE,
  verbose = FALSE,
  audit = TRUE,
  output = "TSL",
  isRaw = TRUE,
  lowPass = TRUE
)
```

Arguments

`.x`	data.table. Input velocity records with a time column and one or more signal columns.
`units.source`	character. Source units for input velocity when `isRaw = TRUE`. Supported: "mm", "cm", "m", "gal", "g" (interpreted for acceleration scaling on derivative/integral paths). If different from `units.target`, a scale factor is applied per channel.
`time`	character. Name of the time column in the input (default `"t"`). Internally and in `TSL` output, time is canonicalized to `t`.

Fmax	numeric. Maximum frequency of interest (Hz). Guides STFT strategy.
kNyq	numeric. Target Nyquist multiplier (Fs_target ~= kNyq * Fmax) if the user forces it; otherwise an automatic grid is searched.
resample	logical. Kept for compatibility; the actual decision is made by the internal STFT strategy based on Fmax and constraints.
derivate	character. Derivative method for VT -> AT ("time" or "freq").
units.target	character. Target units for output acceleration. Default: "mm".
NW	integer. Nominal STFT window length (samples). May be adjusted.
OVLP	numeric. Window overlap percent.
flatZeros	logical. Apply edge tapering. If isRaw = TRUE, tapering is applied regardless.
Astop0, Apass0	numeric. Normalized thresholds 0..1 for taper/flatten; relative to the per-channel max amplitude.
AstopLP, ApassLP	numeric. Anti-alias LP specifications for resampling.
trimZeros	logical. If TRUE, trims leading/trailing zeros by the final window.
detrend	logical. Remove mean before/after stages.
regularize	logical. Force time regularization of input if needed.
verbose	logical. Print diagnostic logs.
audit	logical. If TRUE, runs auditSTFT() to validate STFT/resampling strategy and emit warnings for risky configurations. Default: TRUE.
output	character. Early/short-circuit outputs (default: "TSL"): "VTo", "AT", "VT", "DT", "TSW", "TSL".
isRaw	logical. If TRUE, handle unit conversion and default tapering.
lowPass	logical. If TRUE, multiply the spectral derivative kernel by an additional Butterworth-like low-pass at Fmax to suppress high-frequency amplification of the numerical derivative. Default: TRUE.

Value

Returns the requested object based on output.

Examples

```
t <- seq(0, 2, by = 0.02)
x <- data.table::data.table(
  t = t,
  H1 = sin(2 * pi * t),
  H2 = 0.5 * cos(2 * pi * t),
  UP = 0.25 * sin(4 * pi * t)
)
tsl <- VT2TS(x, units.source = "mm", Fmax = 4, NW = 16,
             audit = FALSE, isRaw = FALSE)
head(tsl)
```

writeSelection *Write a selection table to* selection/<name>.csv / .json.

Description

Deduplicates (OwnerID, EventID, StationID) so the CSV carries one row per station folder (= one record). The JSON sidecar captures audit metadata: name, timestamp, total hits, hits per owner.

Usage

```
writeSelection(DT, name, path)
```

Arguments

DT	record metadata data.table after user filters.
name	identifier for the selection, used as filename stem.
path	Absolute path to the directory where <name>.csv and <name>.json will be written. Required – no default.

Details

The CSV is the canonical input contract for any downstream orchestrator that iterates over a selection: each row identifies one (OwnerID, EventID, StationID) station folder under whichever records root the orchestrator was given.

Value

Invisibly, the deduplicated selection data.table.

Examples

```
x <- data.table::data.table(
  OwnerID = c("AAA", "AAA"),
  EventID = c("E1", "E1"),
  StationID = c("S1", "S1"),
  DIR = c("H1", "H2")
)
path <- file.path(tempdir(), "gmsp-selection-example")
unlink(path, recursive = TRUE)
suppressMessages(writeSelection(x, name = "demo", path = path))
list.files(path)
```

Index