

Package: PhysioStream (via r-universe)

October 3, 2026

Title Governed Real-Time Streams for Physiological Data

Version 0.9.4

Description Provides validated stream metadata, source and sink contracts, a bounded single-producer/single-consumer ring buffer, deterministic in-process loopback streams, capability-gated Lab Streaming Layer inlets and outlets, marker-event capture, governed Extensible Data Format read/write, and conversion of buffered physiological samples to PhysioExperiment objects, with explicit robust clock-offset, drift, jitter, multi-stream synchronization, transactional causal processing pipelines, explicit backpressure, monotonic latency measurement, and governed loopback, MQTT, and TTL stimulation-command outlets with arm, dead-man, maximum, duplicate-ID, and refractory interlocks, plus causal EMG-onset, EEG-ERD, and phase-target detectors bound to commit-before-stimulation closed-loop session control and provenance, with a bounded live biofeedback scope and marker-anchored video-time synchronization.

Depends R (>= 4.2), PhysioExperiment (>= 2.0.0)

Imports methods, Rcpp, S4Vectors, SummarizedExperiment, PhysioPreprocess, digest, jsonlite, xml2

LinkingTo Rcpp

Suggests PhysioCore, reticulate, shiny, testthat (>= 3.2.0), knitr, rmarkdown

VignetteBuilder knitr

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

Collate 'PhysioStream-package.R' 'class-stream-info.R'
'class-stream.R' 'RcppExports.R' 'ring-buffer.R' 'loopback.R'
'lsl-backend.R' 'lsl-info.R' 'lsl-endpoints.R' 'lsl-markers.R'
'xdf-backend.R' 'xdf-writer.R' 'io-xdf.R' 'clock-model.R'
'sync.R' 'sync-diagnostics.R' 'processor-state.R' 'adaptive.R'

'incremental-decomp.R' 'streaming-spectral.R' 'pipeline.R'
 'pipeline-operations.R' 'benchmark-latency.R'
 'trigger-command.R' 'trigger-backend.R' 'trigger-loopback.R'
 'trigger-mqtt.R' 'trigger-ttl.R' 'closed-loop-detectors.R'
 'closed-loop-phase.R' 'closed-loop.R' 'biofeedback-contract.R'
 'biofeedback.R' 'video-sync.R' 'stream-to-pe.R' 'zzz.R'

Config/testthat/edition 3

SystemRequirements C++17

URL <https://github.com/x-biosignal/PhysioStream>,
<https://x-biosignal.r-universe.dev/PhysioStream>,
<https://x-biosignal.github.io/PhysioStream/>

BugReports <https://github.com/x-biosignal/PhysioStream/issues>

Config/roxygen2/version 8.0.0

RoxygenNote 7.3.3

Config/pak/sysreqs libxml2-dev zlib1g-dev

Repository <https://x-biosignal.r-universe.dev>

Date/Publication 2026-10-03 13:11:42 UTC

RemoteUrl <https://github.com/x-biosignal/PhysioStream>

RemoteRef HEAD

RemoteSha 197f7b9569072931fab44185e75c44c9ec657b27

Contents

armTrigger	4
bandpassRmsOp	5
biofeedback-lifecycle	6
biofeedback-state	6
biofeedbackScope	7
biofeedbackUpdate	9
clockOffset	10
closed-loop-lifecycle	11
closed-loop-state	12
closedLoop	13
closedLoopProvenance	14
dejitter	15
driftCorrect	16
emgOnsetOp	17
erdIntentOp	18
incrementalPCA	19
lmsFilter	20
loopbackFeed	20
loopbackSource	21
LoopbackSource-class	22

loopbackTrigger	22
lslAvailable	23
lslBackendInfo	24
lslInlet	24
LSLInlet-class	25
lslMarkerEvents	26
lslOutlet	27
LSLOutlet-class	28
lslPull	28
lslPush	29
lslResolveStreams	30
measureLatency	31
mqttAvailable	32
mqttTrigger	32
nlmsFilter	34
onChunk	35
onlineICA	36
phaseTargetOp	37
pipeline-state	38
pipelineEnqueue	38
pipelineRun	39
pipelineStep	40
processorState	41
readXDF	41
ring-read	42
ring-state	43
ringBuffer	44
RingBuffer-class	45
ringPush	45
ringReset	46
rlsFilter	46
sendStim	47
slidingSTFT	48
stream-metadata	49
stream-operations	50
StreamEndpoint-class	52
streamInfo	53
StreamInfo-class	54
streamPipeline	55
StreamSink-class	56
streamSnapshot	56
StreamSource-class	57
syncDiagnostics	57
syncStreams	58
triggerOpen	59
triggerState	60
ttlTrigger	60
video-sync-map	62

videoSync	63
videoSyncViewer	64
welchOnline	65
writeXDF	66
xdfAvailable	67
xdfBackendInfo	68
Index	69

armTrigger	<i>Arm, heartbeat, disarm, or emergency-stop a trigger</i>
------------	--

Description

Every trigger opens unarmed. Dead-man expiry disarms locally and requires a new explicit arm with a different session ID within the open lifecycle. Emergency stop latches locally even if the transport stop attempt fails. These software gates do not replace device hardware limits or an independent emergency stop.

Usage

```
armTrigger(trigger, session_id, now_ns = NULL)

triggerHeartbeat(trigger, now_ns = NULL)

disarmTrigger(trigger, reason = "caller", now_ns = NULL)

emergencyStop(trigger, reason = "caller", now_ns = NULL)
```

Arguments

trigger	A TriggerBackend.
session_id	Unique non-empty session identifier.
now_ns	Optional exact process-local monotonic nanosecond value.
reason	Bounded non-sensitive reason code.

Value

The trigger, invisibly.

Examples

```
trigger <- loopbackTrigger(allowed_channels = "left", max_intensity = 20,
                           intensity_unit = "mA", max_duration_ms = 500,
                           refractory_ms = 0, deadman_ms = 1000)

triggerOpen(trigger)
armTrigger(trigger, "session-1", now_ns = 0)
triggerState(trigger)$arm$status
```

bandpassRmsOp

*Construct the compiled causal SOS plus rolling-RMS operation***Description**

The operation uses direct-form-II-transposed second-order sections followed by a per-channel rolling root-mean-square. It is a descriptor for `onChunk()` and performs no processing until registered and stepped.

Usage

```
bandpassRmsOp(
  sos,
  window_samples,
  warmup = c("partial", "complete"),
  name = "bandpass_rms"
)
```

Arguments

<code>sos</code>	Finite second-order-section matrix with columns <code>b0</code> , <code>b1</code> , <code>b2</code> , <code>a0</code> , <code>a1</code> , <code>a2</code> .
<code>window_samples</code>	Exact positive RMS window length.
<code>warmup</code>	Whether to emit partial-window RMS values or only rows after a complete window exists.
<code>name</code>	Default operation name.

Value

A `PipelineOperation` descriptor.

Examples

```
# A pass-through SOS keeps the example self-contained; use
# PhysioPreprocess::sosDesign() for a real band definition.
sos <- matrix(c(1, 0, 0, 1, 0, 0), 1L, 6L,
              dimnames = list(NULL, c("b0", "b1", "b2", "a0", "a1", "a2")))
op <- bandpassRmsOp(sos, window_samples = 4L)
pipeline <- streamPipeline(chunk_size = 8L)
onChunk(pipeline, op)
x <- matrix(sin(seq_len(16)), 8, 2, dimnames = list(NULL, c("C3", "C4")))
pipelineEnqueue(pipeline, x, ingest_time_ns = 0)
pipelineStep(pipeline)$results[[1]]$output$samples
```

`biofeedback-lifecycle` *Start, step, and stop a biofeedback runtime*

Description

Start, step, and stop a biofeedback runtime

Usage

```
biofeedbackStart(scope)

biofeedbackStep(scope, max_chunks = 16L)

biofeedbackStop(scope)
```

Arguments

`scope` A BiofeedbackScope.
`max_chunks` Exact per-step work bound.

Value

Lifecycle functions return scope invisibly. `biofeedbackStep()` returns a plain update summary.

Examples

```
info <- streamInfo("demo", type = "EEG",
  channel_names = c("left", "right"), nominal_srate = 100,
  channel_units = c("uV", "uV"))
source <- streamOpen(loopbackSource(info, capacity = 4096L))
scope <- biofeedbackScope(source, update_hz = 20, window_seconds = 1,
  max_points = 64L, launch = FALSE)
biofeedbackStart(scope)
loopbackFeed(source, matrix(as.double(1:40), 20, 2), seq_len(20) / 100)
invisible(biofeedbackStep(scope))
biofeedbackStop(scope)
biofeedbackState(scope)$lifecycle
```

`biofeedback-state` *Inspect a biofeedback frame or portable state*

Description

Inspect a biofeedback frame or portable state

Usage

```
biofeedbackFrame(scope)
```

```
biofeedbackState(scope)
```

Arguments

scope A BiofeedbackScope.

Value

A deep plain-list copy, or NULL before the first frame.

Examples

```
info <- streamInfo("demo", type = "EEG",
  channel_names = c("left", "right"), nominal_srate = 100,
  channel_units = c("uV", "uV"))
source <- loopbackSource(info, capacity = 256L)
scope <- biofeedbackScope(source, source_lifecycle = "own", launch = FALSE)
biofeedbackState(scope)$lifecycle
biofeedbackFrame(scope)
```

biofeedbackScope	<i>Construct and optionally launch a governed live biofeedback scope</i>
------------------	--

Description

Construction is side-effect-free. With `launch = FALSE`, the returned runtime is advanced synchronously with `biofeedbackStart()` and `biofeedbackStep()`. Shiny is required only for `launch = TRUE`.

Usage

```
biofeedbackScope(
  source,
  pipeline = NULL,
  channels = NULL,
  derived = list(),
  window_seconds = 10,
  update_hz = 20,
  max_points = 2000L,
  gain = 1,
  threshold = NULL,
  target_range = NULL,
  source_lifecycle = c("borrow", "own"),
  launch = interactive(),
  host = "127.0.0.1",
  port = NULL,
```

```

    browser = interactive(),
    clock = NULL
  )

```

Arguments

source	A regular-rate numeric StreamSource.
pipeline	Optional source-backed StreamPipeline.
channels	Exact source channel names to display.
derived	Strictly named derived/external trace descriptors.
window_seconds	Positive visible signal-time window.
update_hz	Requested display update rate.
max_points	Maximum plotted representatives per trace.
gain	Positive display-only gain.
threshold	Optional finite display threshold.
target_range	Optional increasing display target range.
source_lifecycle	Whether the scope borrows an open source or owns its open/close lifecycle.
launch	Whether to run the installed Shiny app immediately.
host	Local Shiny bind host.
port	Optional exact TCP port.
browser	Whether Shiny should launch a browser.
clock	Optional monotonic nanosecond clock for deterministic tests.

Value

A BiofeedbackScope when launch = FALSE, otherwise the result of shiny::runApp().

Examples

```

# Construction is device-free; Shiny is required only for launch = TRUE.
info <- streamInfo("demo", type = "EEG",
  channel_names = c("left", "right"), nominal_srate = 100,
  channel_units = c("uV", "uV"))
source <- loopbackSource(info, capacity = 256L)
scope <- biofeedbackScope(source, source_lifecycle = "own", launch = FALSE)
biofeedbackState(scope)$lifecycle

```

biofeedbackUpdate	<i>Publish a bounded external feedback update</i>
-------------------	---

Description

This is the sink used by downstream live metric adapters. It updates only declared external display traces and has no pipeline or stimulation effect.

Usage

```
biofeedbackUpdate(
  scope,
  values,
  timestamp,
  names = base::names(values),
  units = NULL,
  sequence = NULL
)
```

Arguments

scope	A running BiofeedbackScope.
values	Finite named values.
timestamp	One signal-domain timestamp.
names	Exact names corresponding to values.
units	Optional exact units corresponding to values.
sequence	Optional exact contiguous update sequence.

Value

An immutable plain receipt.

Examples

```
info <- streamInfo("demo", type = "EEG",
  channel_names = c("left", "right"), nominal_srate = 100,
  channel_units = c("uV", "uV"))
source <- streamOpen(loopbackSource(info, capacity = 4096L))
scope <- biofeedbackScope(source, derived = list(
  score = list(type = "external", unit = "ratio", gain = 1)
), launch = FALSE)
biofeedbackStart(scope)
receipt <- biofeedbackUpdate(scope, c(score = 0.8), timestamp = 1,
  units = "ratio", sequence = 0)

receipt$names
biofeedbackStop(scope)
```

clockOffset	<i>Estimate a governed stream-clock mapping</i>
-------------	---

Description

Fits recorded clock offsets under the sign convention $\text{master_time} = \text{device_time} + \text{offset}(\text{device_time})$.
 Fits are centered within each reset segment.

Usage

```
clockOffset(
  clock_times,
  clock_values,
  method = c("huber", "ols", "ransac"),
  origin = c("median", "first"),
  segments = NULL,
  huber_k = 1.345,
  ransac_threshold = NULL,
  ransac_trials = 200L,
  seed = 1L
)
```

Arguments

clock_times	Finite non-decreasing device-clock observation times.
clock_values	Finite observed offsets in seconds.
method	Exact fitting method.
origin	Exact centering rule.
segments	Optional positive labels or zero-based inclusive ranges.
huber_k	Positive Huber tuning constant.
ransac_threshold	Optional positive inlier threshold in seconds.
ransac_trials	Positive number of deterministic RANSAC trials.
seed	Non-negative deterministic RANSAC seed.

Value

A serializable StreamClockModel.

Examples

```
device <- seq(0, 10, by = 0.5)
offset <- -0.4 + 0.005 * (device - median(device))
model <- clockOffset(device, offset, method = "ols")
model$segments$drift_ppm
```

closed-loop-lifecycle *Run a governed closed-loop session*

Description

The controller is synchronous and terminal. Start explicitly opens and arms its trigger. Step processes one committed chunk and then dispatches due actions. Flush dispatches due delayed actions without ingesting samples. Stop and emergency stop suppress pending actions and close the trigger.

Usage

```
closedLoopStart(controller, session_id, now_ns = NULL)

closedLoopStep(controller, samples, timestamps = NULL, now_ns = NULL)

closedLoopFlush(controller, now_ns = NULL)

closedLoopStop(controller, reason = "caller", now_ns = NULL)

closedLoopEmergencyStop(controller, reason = "caller", now_ns = NULL)
```

Arguments

controller	A ClosedLoopController.
session_id	New bounded trigger session identifier.
now_ns	Optional exact test monotonic nanosecond value.
samples	Finite sample-by-channel chunk.
timestamps	Optional signal-domain timestamps.
reason	Bounded non-sensitive stop reason.

Value

Start/stop functions return controller invisibly. Step and flush return bounded plain result lists.

Examples

```
set.seed(1)
pipeline <- streamPipeline(chunk_size = 2048L)
trigger <- loopbackTrigger(allowed_channels = "left", max_intensity = 20,
                           intensity_unit = "mA", max_duration_ms = 500,
                           refractory_ms = 0, deadman_ms = 1000)
detector <- emgOnsetOp("emg", sampling_rate = 1000,
                      baseline_samples = 30, rms_window_samples = 5)
controller <- closedLoop(pipeline, trigger, detector, intensity = 2,
                         stim_channel = "left", duration_ms = 10)
closedLoopStart(controller, "demo-session", now_ns = 0)
emg <- rnorm(100, sd = 0.08)
```

```

result <- closedLoopStep(controller,
  matrix(emg, ncol = 1, dimnames = list(NULL, "emg")),
  timestamps = (seq_along(emg) - 1) / 1000, now_ns = 1e6)
length(result$actions)
closedLoopStop(controller, now_ns = 2e6)

```

closed-loop-state	<i>Inspect governed closed-loop state and audit</i>
-------------------	---

Description

Returned values are deep portable copies. They contain no runtime object, trigger credential, patient identifier, or raw physiological samples.

Usage

```

closedLoopState(controller)

closedLoopLog(controller)

```

Arguments

controller A ClosedLoopController.

Value

closedLoopState() returns the sealed state; closedLoopLog() returns its bounded structured session log.

Examples

```

pipeline <- streamPipeline(chunk_size = 2048L)
trigger <- loopbackTrigger(allowed_channels = "left", max_intensity = 20,
  intensity_unit = "mA", max_duration_ms = 500,
  refractory_ms = 0, deadman_ms = 1000)
detector <- emgOnsetOp("emg", sampling_rate = 1000,
  baseline_samples = 30, rms_window_samples = 5)
controller <- closedLoop(pipeline, trigger, detector, intensity = 2,
  stim_channel = "left", duration_ms = 10)
closedLoopState(controller)$lifecycle$status
closedLoopLog(controller)

```

closedLoop

*Construct a governed detector-to-stimulation controller***Description**

closedLoop() is side-effect free with respect to the trigger transport. It registers one governed detector operation in the supplied pipeline and owns that graph and trigger for a terminal session lifecycle. Detector events are decoded only after [pipelineStep\(\)](#) commits; callbacks never stimulate.

Usage

```
closedLoop(
    pipeline,
    trigger,
    detector,
    intensity,
    stim_channel,
    duration_ms,
    delay_ms = 0,
    event_refractory_ms = 0,
    pending_capacity = 128L,
    log_capacity = 4096L,
    clock = NULL
)
```

Arguments

pipeline	An empty-queue StreamPipeline.
trigger	A closed, disarmed TriggerBackend.
detector	An emgOnsetOp() , erdIntentOp() , or phaseTargetOp() descriptor.
intensity, stim_channel, duration_ms	Explicit stimulation dose fields.
delay_ms	Detection-commit to attempt delay. Positive delays require a later step or closedLoopFlush() ; no sleep or background task is used.
event_refractory_ms	Controller-level detection refractory period.
pending_capacity	Maximum delayed actions.
log_capacity	Maximum retained audit records and detector identities.
clock	Optional deterministic monotonic clock, for loopback tests.

Value

A closed ClosedLoopController.

Examples

```

pipeline <- streamPipeline(chunk_size = 2048L)
trigger <- loopbackTrigger(allowed_channels = "left", max_intensity = 20,
                           intensity_unit = "mA", max_duration_ms = 500,
                           refractory_ms = 0, deadman_ms = 1000)
detector <- emgOnsetOp("emg", sampling_rate = 1000,
                      baseline_samples = 30, rms_window_samples = 5)
controller <- closedLoop(pipeline, trigger, detector, intensity = 2,
                        stim_channel = "left", duration_ms = 10)
closedLoopState(controller)$lifecycle$status

```

closedLoopProvenance	<i>Append a stopped closed-loop session to PhysioExperiment provenance</i>
----------------------	--

Description

The bounded log records detection, queue, and acknowledgement evidence. Acknowledgement is explicitly not a physical-delivery claim. Raw samples and trigger runtime values are excluded.

Usage

```
closedLoopProvenance(controller, x, input_assay = NA_character_)
```

Arguments

controller	A stopped ClosedLoopController.
x	A PhysioExperiment.
input_assay	Exact existing assay name or NA_character_.

Value

A modified PhysioExperiment with one provenance activity.

Examples

```

pipeline <- streamPipeline(chunk_size = 2048L)
trigger <- loopbackTrigger(allowed_channels = "left", max_intensity = 20,
                           intensity_unit = "mA", max_duration_ms = 500,
                           refractory_ms = 0, deadman_ms = 1000)
detector <- emgOnsetOp("emg", sampling_rate = 1000,
                      baseline_samples = 30, rms_window_samples = 5)
controller <- closedLoop(pipeline, trigger, detector, intensity = 2,
                        stim_channel = "left", duration_ms = 10)
closedLoopStart(controller, "demo-session", now_ns = 0)
closedLoopStep(controller,
  matrix(rnorm(100, sd = 0.08), ncol = 1, dimnames = list(NULL, "emg")),
  now_ns = 1e6)

```

```

closedLoopStop(controller, now_ns = 2e6)
pe <- PhysioExperiment(assays = list(raw = matrix(as.double(1:20), 10, 2)),
                      samplingRate = 100)
result <- closedLoopProvenance(controller, pe, "raw")
length(S4Vectors::metadata(result)$provenance)

```

 dejitter

Regularize stream timestamps on a nominal grid

Description

This changes timestamps only. It never interpolates or reorders samples.

Usage

```

dejitter(
  timestamps,
  nominal_srate,
  segments = NULL,
  anchor = c("least_squares", "first"),
  max_residual_seconds = Inf
)

```

Arguments

timestamps	Finite non-decreasing timestamps.
nominal_srate	Positive nominal sampling rate.
segments	Optional positive labels or zero-based inclusive ranges.
anchor	Exact grid anchoring rule.
max_residual_seconds	Maximum permitted absolute jitter residual.

Value

Regular timestamps with a plain `dejitter` diagnostics attribute.

Examples

```

raw <- c(10.001, 10.010, 10.021, 10.029)
regular <- dejitter(raw, nominal_srate = 100)
as.numeric(regular)

```

driftCorrect	<i>Apply a governed stream-clock model</i>
--------------	--

Description

Apply a governed stream-clock model

Usage

```
driftCorrect(  
  timestamps,  
  model,  
  segments = NULL,  
  extrapolate = c("bounded", "error"),  
  max_extrapolation_seconds = 30  
)
```

Arguments

timestamps	Finite device-clock timestamps.
model	A valid StreamClockModel.
segments	Optional positive labels or zero-based inclusive ranges.
extrapolate	Exact extrapolation policy.
max_extrapolation_seconds	Non-negative bounded extrapolation limit.

Value

Corrected timestamps with a plain clock_correction attribute.

Examples

```
device <- seq(0, 10, by = 0.5)  
offset <- -0.4 + 0.005 * (device - median(device))  
model <- clockOffset(device, offset, method = "ols")  
corrected <- driftCorrect(seq(0, 10, by = 1), model)  
as.numeric(corrected)
```

emgOnsetOp

*Construct a causal streaming EMG-onset detector***Description**

The detector uses a trailing RMS envelope, a frozen warm-up baseline, and explicit enter/release hysteresis. It is a pure `onChunk()` operation: stimulation is attempted only by `closedLoopStep()` after pipeline commit. Explicit signal-timestamp intervals must remain within five percent of the declared sampling interval.

Usage

```
emgOnsetOp(
  channel,
  sampling_rate,
  baseline_samples,
  rms_window_samples,
  enter_z = 5,
  release_z = 2,
  min_on_samples = 3L,
  min_off_samples = 3L,
  min_baseline_sd = 1e-08
)
```

Arguments

<code>channel</code>	Exact input channel name.
<code>sampling_rate</code>	Sampling rate in hertz.
<code>baseline_samples</code>	Number of complete-window envelope samples used for the frozen baseline.
<code>rms_window_samples</code>	Trailing causal RMS window length.
<code>enter_z, release_z</code>	Baseline-standardized enter/release thresholds.
<code>min_on_samples, min_off_samples</code>	Consecutive confirmation counts.
<code>min_baseline_sd</code>	Minimum accepted baseline scale.

Value

A governed PipelineOperation detector descriptor.

Examples

```
set.seed(1)
emg <- c(rnorm(80, sd = 0.08), rep(1.5, 12), rnorm(30, sd = 0.08))
detector <- emgOnsetOp("emg", sampling_rate = 1000,
                      baseline_samples = 50, rms_window_samples = 8)
pipeline <- streamPipeline(chunk_size = length(emg))
onChunk(pipeline, detector)
pipelineEnqueue(pipeline,
               matrix(emg, ncol = 1, dimnames = list(NULL, "emg")),
               seq_along(emg) / 1000)
length(pipelineStep(pipeline)$results[[1]]$events)
```

erdIntentOp

Construct a causal streaming EEG ERD detector

Description

A fixed causal biquad and trailing power window estimate band-power change from a frozen baseline. The result is an ERD proxy, not a clinical movement-intention diagnosis. Explicit signal-timestamp intervals must remain within five percent of the declared sampling interval.

Usage

```
erdIntentOp(
  channel,
  sampling_rate,
  band = c(8, 13),
  baseline_samples,
  power_window_samples,
  enter_percent = -30,
  release_percent = -15,
  min_on_samples = 3L,
  min_off_samples = 3L,
  min_baseline_power = 1e-12
)
```

Arguments

channel	Exact input channel name.
sampling_rate	Sampling rate in hertz.
band	Two strictly increasing band edges in hertz.
baseline_samples	Number of complete power windows in the baseline.
power_window_samples	Trailing causal mean-square window length.
enter_percent, release_percent	ERD enter/release percentages.

min_on_samples, min_off_samples
 Consecutive confirmation counts.
 min_baseline_power
 Minimum accepted frozen baseline power.

Value

A governed PipelineOperation detector descriptor.

Examples

```
detector <- erdIntentOp("eeg", sampling_rate = 1000, band = c(8, 13),
                        baseline_samples = 40, power_window_samples = 10)
class(detector)
```

incrementalPCA	<i>Incremental principal-component analysis</i>
----------------	---

Description

For forgetting = 1, the processor retains Chan-Golub-LeVeque sufficient statistics and is batch-equivalent without retaining samples. Smaller forgetting factors produce an explicitly exponentially weighted covariance.

Usage

```
incrementalPCA(n_components, n_features = NULL, forgetting = 1, center = TRUE)
```

Arguments

n_components Number of retained components.
 n_features Optional feature count to bind at construction.
 forgetting Per-sample covariance forgetting factor in (0, 1].
 center Whether to estimate and subtract a running mean.

Value

A mutable IncrementalPCA streaming processor.

Examples

```
set.seed(1)
latent <- matrix(rnorm(300), ncol = 3)
samples <- latent %*% matrix(rnorm(12), 3, 4)
pca <- incrementalPCA(n_components = 2L)
result <- update(pca, samples)
result$diagnostics$explained_variance
```

<code>lmsFilter</code>	<i>Stateful least-mean-squares adaptive filter</i>
------------------------	--

Description

Adaptive filters estimate the part of each signal channel linearly predictable from its paired reference. The returned residual is not inherently a clean physiological signal; that interpretation requires a caller-owned reference-noise assumption.

Usage

```
lmsFilter(n_taps, step_size, leakage = 0, initial_weights = NULL)
```

Arguments

<code>n_taps</code>	Exact adaptive-filter order.
<code>step_size</code>	Positive LMS step size.
<code>leakage</code>	Weight leakage in $[0, 1)$.
<code>initial_weights</code>	Optional finite <code>n_taps</code> by channel matrix.

Value

A mutable `LMSFilter` streaming processor.

Examples

```
# Cancel a reference-correlated component from a short synthetic signal.
set.seed(1)
reference <- sin(2 * pi * 5 * seq_len(200) / 100)
signal <- 0.4 * reference + rnorm(200, sd = 0.05)
filt <- lmsFilter(n_taps = 4L, step_size = 0.05)
result <- update(filt, matrix(signal, ncol = 1),
                reference = matrix(reference, ncol = 1))
str(result$output)
```

<code>loopbackFeed</code>	<i>Feed a loopback source</i>
---------------------------	-------------------------------

Description

Feed a loopback source

Usage

```
loopbackFeed(x, samples, timestamps)
```

Arguments

x	An open LoopbackSource.
samples	A finite sample-by-channel real matrix.
timestamps	Strictly increasing finite timestamps.

Value

x, invisibly. The native buffer is modified by reference.

Examples

```
info <- streamInfo("demo", type = "EEG",
  channel_names = c("C3", "C4"), nominal_srate = 100)
src <- streamOpen(loopbackSource(info, capacity = 32L))
loopbackFeed(src, matrix(as.double(1:6), 3, 2), c(0.01, 0.02, 0.03))
streamPull(src)$count
```

loopbackSource	<i>Construct a loopback source</i>
----------------	------------------------------------

Description

Construct a loopback source

Usage

```
loopbackSource(info, capacity = 1024L)
```

Arguments

info	Numeric StreamInfo.
capacity	Exact positive ring capacity.

Value

A closed-over loopback source in the "created" state.

Examples

```
info <- streamInfo("demo", type = "EEG",
  channel_names = c("C3", "C4"), nominal_srate = 100)
src <- streamOpen(loopbackSource(info, capacity = 32L))
streamState(src)
```

LoopbackSource-class	<i>Deterministic in-process stream source</i>
----------------------	---

Description

Deterministic in-process stream source

Slots

buffer Live native ring buffer.

Examples

```
info <- streamInfo("demo", type = "EEG",
                  channel_names = c("C3", "C4"), nominal_srate = 100)
src <- loopbackSource(info, capacity = 16L)
is(src, "LoopbackSource")
```

loopbackTrigger	<i>Construct a deterministic governed loopback trigger</i>
-----------------	--

Description

The loopback backend records commands and timing without opening a device or network connection. It exercises the same arm, dead-man, maximum, duplicate ID, and refractory gates as live transports.

Usage

```
loopbackTrigger(
  allowed_channels,
  max_intensity,
  intensity_unit,
  max_duration_ms,
  refractory_ms,
  deadman_ms,
  audit_capacity = 4096L,
  clock = NULL
)
```

Arguments

allowed_channels	Unique allowed channel identifiers.
max_intensity	Positive scalar or exactly channel-named maxima.
intensity_unit	Explicit device intensity unit; no conversion occurs.
max_duration_ms	Positive scalar or channel-named duration maxima.
refractory_ms	Non-negative scalar or channel-named refractory times.
deadman_ms	Positive heartbeat expiry interval.
audit_capacity	Bounded command/audit capacity.
clock	Optional deterministic monotonic clock for tests.

Value

A closed, disarmed LoopbackTrigger.

Examples

```
trigger <- loopbackTrigger(allowed_channels = "left", max_intensity = 20,
                           intensity_unit = "mA", max_duration_ms = 500,
                           refractory_ms = 0, deadman_ms = 1000)
triggerState(trigger)$lifecycle$status
```

lslAvailable	<i>Test Lab Streaming Layer backend availability</i>
--------------	--

Description

The default probe is conservative and does not initialize Python. Explicit transport operations initialize only the already configured reticulate interpreter; no function installs or selects an environment.

Usage

```
lslAvailable(backend = c("auto", "pylsl"), initialize = FALSE)
```

Arguments

backend	Exact backend, currently "auto" or "pylsl".
initialize	Whether the configured Python interpreter may be initialized to verify that pylsl and liblsl load.

Value

One logical value.

Examples

```
# Reports FALSE unless pylsl and liblsl are installed and loadable.
lslAvailable()
```

lslBackendInfo

Report the configured LSL backend

Description

Report the configured LSL backend

Usage

```
lslBackendInfo(backend = c("auto", "pylsl"))
```

Arguments

backend Exact backend, currently "auto" or "pylsl".

Value

A serializable named list of backend and runtime versions.

Examples

```
# Resolving the backend version requires pylsl/liblsl to be installed.
if (lslAvailable()) str(lslBackendInfo())
```

lslInlet

Construct an LSL inlet

Description

Construction validates R state only. Assign the result of streamOpen() to open the already configured backend.

Usage

```
lslInlet(
  info,
  capacity = 4096L,
  max_chunk = 1024L,
  recover = TRUE,
  processing = c("none", "clocksync"),
  marker_capacity = 4096L,
  backend = getOption("PhysioStream.lsl_backend", "auto")
)
```

Arguments

info	A resolved StreamInfo.
capacity	Numeric ring capacity.
max_chunk	Maximum rows per backend pull.
recover	Whether liblsl may recover by non-empty source id.
processing	Exact timestamp processing mode.
marker_capacity	String marker queue capacity.
backend	Exact backend, currently "auto" or "pylsl".

Value

An LSLInlet in state "created".

Examples

```
# Requires a running LSL stream on the local network.
if (lslAvailable()) {
  found <- lslResolveStreams(timeout = 0.2)
  if (length(found)) {
    inlet <- streamOpen(lslInlet(found[[1]]))
    streamClose(inlet)
  }
}
```

LSLInlet-class

Lab Streaming Layer inlet

Description

Live backend and buffer state are intentionally reference-like and do not survive serialization.

Slots

runtime Private live backend state.
buffer Numeric ring buffer or bounded marker queue.
max_chunk Maximum rows requested per pull.
recover Whether liblsl source recovery is enabled.
processing Exact timestamp processing mode.
marker_capacity Marker queue capacity.

Examples

```
# Concrete inlet objects come from lslInlet(); see ?lslInlet.
isVirtualClass("LSLInlet")
```

lslMarkerEvents	<i>Convert buffered LSL markers to PhysioEvents</i>
-----------------	---

Description

Marker timestamps remain in the LSL clock domain unless an explicit `time_origin` is subtracted. Cross-stream alignment is handled separately.

Usage

```
lslMarkerEvents(x, n = NULL, consume = FALSE, time_origin = 0, type = NULL)
```

Arguments

<code>x</code>	An open string/irregular-rate LSLInlet.
<code>n</code>	Optional newest marker count.
<code>consume</code>	Whether to consume the complete oldest selection.
<code>time_origin</code>	Finite scalar subtracted from LSL timestamps.
<code>type</code>	Optional event type; defaults to the stream type.

Value

A valid `PhysioExperiment::PhysioEvents`.

Examples

```
# Requires an open marker inlet bound to a live LSL marker stream.
if (lslAvailable()) {
  found <- lslResolveStreams(property = "type", value = "Markers",
                             timeout = 0.2)
  if (length(found)) {
    inlet <- streamOpen(lslInlet(found[[1]]))
    events <- lslMarkerEvents(inlet)
    streamClose(inlet)
  }
}
```

lslOutlet	<i>Construct an LSL outlet</i>
-----------	--------------------------------

Description

Construct an LSL outlet

Usage

```
lslOutlet(  
  info,  
  chunk_size = 0L,  
  max_buffered = 360L,  
  backend = getOption("PhysioStream.lsl_backend", "auto")  
)
```

Arguments

info	A valid numeric regular-rate or string irregular-rate StreamInfo.
chunk_size	Exact non-negative LSL chunk preference.
max_buffered	Exact positive LSL sender buffer bound.
backend	Exact backend, currently "auto" or "pylsl".

Value

An LSLOutlet in state "created".

Examples

```
# Publishing requires the LSL runtime (pylsl/liblsl).  
if (lslAvailable()) {  
  info <- streamInfo("demo", type = "EEG",  
    channel_names = c("C3", "C4"), nominal_srate = 100)  
  outlet <- streamOpen(lslOutlet(info))  
  lslPush(outlet, matrix(as.double(1:4), 2, 2))  
  streamClose(outlet)  
}
```

LSLOutlet-class	<i>Lab Streaming Layer outlet</i>
-----------------	-----------------------------------

Description

Lab Streaming Layer outlet

Slots

runtime Private live backend state.
 chunk_size Preferred LSL chunk size.
 max_buffered LSL sender buffer bound.

Examples

```
# Concrete outlet objects come from lslOutlet(); see ?lslOutlet.  
isVirtualClass("LSLOutlet")
```

lslPull	<i>Pull one LSL chunk into an inlet buffer</i>
---------	--

Description

Pull one LSL chunk into an inlet buffer

Usage

```
lslPull(x, max_samples = x@max_chunk, timeout = 0)
```

Arguments

x	An open LSLInlet.
max_samples	Exact number of samples requested, at most the inlet chunk bound.
timeout	Finite non-negative backend timeout.

Value

A serializable pull summary. Payload remains in the bounded inlet buffer or marker queue.

Examples

```
# Requires an open inlet bound to a live LSL stream.
if (lslAvailable()) {
  found <- lslResolveStreams(timeout = 0.2)
  if (length(found)) {
    inlet <- streamOpen(lslInlet(found[[1]]))
    summary <- lslPull(inlet, max_samples = 32)
    streamClose(inlet)
  }
}
```

lslPush

*Push samples or markers through an LSL outlet***Description**

Push samples or markers through an LSL outlet

Usage

```
lslPush(x, samples, timestamps = NULL, pushthrough = TRUE)
```

Arguments

x	An open LSLOutlet.
samples	A strict sample-by-channel numeric or character matrix.
timestamps	Optional explicit timestamp per row.
pushthrough	Whether the final backend operation flushes the chunk.

Value

A serializable push summary.

Examples

```
# Requires the LSL runtime (pylsl/liblsl).
if (lslAvailable()) {
  info <- streamInfo("demo", type = "EEG",
    channel_names = c("C3", "C4"), nominal_srate = 100)
  outlet <- streamOpen(lslOutlet(info))
  lslPush(outlet, matrix(as.double(1:4), 2, 2))
  streamClose(outlet)
}
```

lslResolveStreams	<i>Resolve visible Lab Streaming Layer streams</i>
-------------------	--

Description

pylsl resolution returns short descriptors. PhysioStream therefore opens a bounded, non-recovering metadata inlet for each result, retrieves its full descriptor, and closes the temporary inlet before returning. No samples are pulled.

Usage

```
lslResolveStreams(
  property = NULL,
  value = NULL,
  minimum = 0L,
  timeout = 1,
  backend = getOption("PhysioStream.lsl_backend", "auto")
)
```

Arguments

property, value	Either both NULL, or an exact core LSL property and one non-empty value.
minimum	Exact minimum number of descriptors required.
timeout	Finite non-negative resolver timeout in seconds.
backend	Exact backend, currently "auto" or "pylsl".

Value

A list of validated StreamInfo objects in backend order.

Examples

```
# Requires a running LSL network with at least one stream.
if (lslAvailable()) {
  streams <- lslResolveStreams(timeout = 0.2)
}
```

measureLatency	<i>Measure governed ingest-to-emit pipeline latency</i>
----------------	---

Description

The default benchmark processes 60 seconds of deterministic 32-channel, 512-Hz data in 32-sample chunks through the compiled causal bandpass plus rolling-RMS operation. Input generation and warmup are outside the measured interval. Results describe empirical performance on the recorded hardware, not a hard real-time guarantee.

Usage

```
measureLatency(
  pipeline = NULL,
  n_channels = 32L,
  sampling_rate = 512,
  hop_samples = 32L,
  duration_s = 60,
  warmup_chunks = 64L,
  seed = 1L
)
```

Arguments

pipeline	Optional empty StreamPipeline. NULL constructs the reference bandpass-plus-RMS graph.
n_channels	Exact positive channel count.
sampling_rate	Finite positive sampling rate.
hop_samples	Exact positive samples per chunk.
duration_s	Finite positive duration whose sample count is divisible by hop_samples.
warmup_chunks	Exact non-negative warmup chunk count.
seed	Exact non-negative local random seed.

Value

A plain benchmark result with summary, latency rows, pipeline state, hardware, configuration, and reference hash.

Examples

```
# Tiny deterministic benchmark; sampling_rate * duration_s must divide
# evenly by hop_samples.
bench <- measureLatency(n_channels = 2L, sampling_rate = 100,
                        hop_samples = 10L, duration_s = 0.5,
                        warmup_chunks = 1L, seed = 1L)

bench$summary$count
```

mqttAvailable	<i>Test or describe the configured MQTT backend</i>
---------------	---

Description

The conservative availability probe does not select or install Python. Initialization is explicit and uses only the interpreter already configured for reticulate.

Usage

```
mqttAvailable(initialize = FALSE)

mqttBackendInfo()
```

Arguments

initialize	Whether the configured Python interpreter may be initialized to verify pinned Paho.
------------	---

Value

mqttAvailable() returns one logical value. mqttBackendInfo() returns a plain capability list.

Examples

```
# Reports FALSE unless the pinned Paho MQTT backend is installed.
mqttAvailable()
```

mqttTrigger	<i>Construct a governed MQTT stimulation-command trigger</i>
-------------	--

Description

Uses MQTT v5, Paho callback API VERSION2, a caller-thread manual network loop, QoS 0 or 1, and retain = FALSE. QoS 1 can duplicate a command, so the receiver must deduplicate exact command IDs. No exactly-once or physical delivery claim is made.

Usage

```
mqttTrigger(
  host,
  topic,
  client_id,
  allowed_channels,
  max_intensity,
  intensity_unit,
```

```

    max_duration_ms,
    refractory_ms,
    deadman_ms,
    port = 8883L,
    qos = 1L,
    tls = c("required", "disabled"),
    allow_insecure_localhost = FALSE,
    username = NULL,
    password = NULL,
    connect_timeout_ms = 5000,
    publish_timeout_ms = 1000,
    audit_capacity = 4096L,
    clock = NULL
)

```

Arguments

host	Broker hostname or IP literal.
topic	Concrete publish topic without wildcards.
client_id	Bounded MQTT client identifier.
allowed_channels	Unique allowed channel identifiers.
max_intensity	Positive scalar or exactly channel-named maxima.
intensity_unit	Explicit device intensity unit; no conversion occurs.
max_duration_ms	Positive scalar or channel-named duration maxima.
refractory_ms	Non-negative scalar or channel-named refractory times.
deadman_ms	Positive heartbeat expiry interval.
port	Integer broker port.
qos	Exact MQTT QoS, 0 or 1.
tls	Exact "required" or "disabled".
allow_insecure_localhost	Permit plaintext only for a loopback host.
username	Optional broker username, excluded from portable state.
password	Optional broker password, excluded from portable state.
connect_timeout_ms	Positive bounded connect timeout.
publish_timeout_ms	Positive bounded acknowledgement timeout.
audit_capacity	Bounded command/audit capacity.
clock	Optional deterministic monotonic clock for tests.

Details

TLS is required by default. Plaintext is restricted to an explicitly acknowledged loopback-only test broker.

Value

A closed, disarmed MqttTrigger.

Examples

```
## Not run:
# Requires a reachable MQTT broker; see loopbackTrigger() for an
# offline equivalent with the same interlocks.
trigger <- mqttTrigger(host = "127.0.0.1", port = 1883,
  topic = "stim/commands",
  allowed_channels = "left", max_intensity = 20,
  intensity_unit = "mA", max_duration_ms = 500,
  refractory_ms = 0, deadman_ms = 1000)

## End(Not run)
```

nlmsFilter

Stateful normalized LMS adaptive filter

Description

Stateful normalized LMS adaptive filter

Usage

```
nlmsFilter(
  n_taps,
  step_size = 0.5,
  epsilon = 1e-08,
  leakage = 0,
  initial_weights = NULL
)
```

Arguments

n_taps	Exact adaptive-filter order.
step_size	Positive normalized step size; values in (0, 2) are the conventional stable range.
epsilon	Positive denominator regularizer.
leakage	Weight leakage in [0, 1).
initial_weights	Optional finite n_taps by channel matrix.

Value

A mutable NLMSFilter streaming processor.

Examples

```

set.seed(1)
reference <- sin(2 * pi * 8 * seq_len(200) / 100)
signal <- 0.3 * reference + rnorm(200, sd = 0.05)
filt <- nlmsFilter(n_taps = 4L, step_size = 0.5)
result <- update(filt, matrix(signal, ncol = 1),
                 reference = matrix(reference, ncol = 1))
tail(result$output)

```

onChunk

*Register a causal per-chunk operation***Description**

The callback is invoked as `callback(chunk, state, context)` and must return a list containing output, state, events, and diagnostics. Operation states are committed only after the full graph succeeds.

Usage

```
onChunk(pipeline, callback, state = NULL, name = NULL, kind = "filter")
```

Arguments

pipeline	A StreamPipeline.
callback	A callback function or a PipelineOperation descriptor.
state	Initial bounded plain-list state for a custom callback.
name	Unique non-empty operation name.
kind	Exact operation kind.

Value

pipeline, invisibly.

Examples

```

sos <- matrix(c(1, 0, 0, 1, 0, 0), 1L, 6L,
              dimnames = list(NULL, c("b0", "b1", "b2", "a0", "a1", "a2")))
pipeline <- streamPipeline(chunk_size = 8L)
onChunk(pipeline, bandpassRmsOp(sos, window_samples = 4L))
length(pipelineState(pipeline)$operations)

```

onlineICA

*Online recursive independent-component analysis***Description**

The processor combines exponentially weighted whitening with a natural gradient update and symmetric decorrelation. Reported components have a deterministic order and sign, but physiological source identity is not inferred. Only complete update blocks are emitted; an incomplete block is retained with its timestamps and sequence identity until a later update.

Usage

```
onlineICA(
  n_components = NULL,
  n_features = NULL,
  learning_rate = 0.01,
  forgetting = 0.995,
  nonlinearity = c("tanh", "extended"),
  block_size = 1L,
  orthogonalize_every = 1L,
  seed = 1L
)
```

Arguments

<code>n_components</code>	Optional number of sources; defaults to all features.
<code>n_features</code>	Optional feature count to bind at construction.
<code>learning_rate</code>	Positive natural-gradient step size.
<code>forgetting</code>	Per-sample whitening forgetting factor in $(0, 1]$.
<code>nonlinearity</code>	Exact tanh or extended score rule.
<code>block_size</code>	Exact positive update block size.
<code>orthogonalize_every</code>	Exact positive number of blocks between symmetric decorrelations.
<code>seed</code>	Exact integer used only for deterministic initialization.

Value

A mutable OnlineICA streaming processor.

Examples

```
set.seed(1)
sources <- cbind(sin(seq_len(200) / 5), sign(sin(seq_len(200) / 7)))
samples <- sources %*% t(matrix(c(1, 0.4, 0.3, 1), 2, 2))
ica <- onlineICA(n_components = 2L, block_size = 10L)
result <- update(ica, samples)
dim(result$output)
```

phaseTargetOp	<i>Construct a causal phase-target detector</i>
---------------	---

Description

The detector fits a sinusoid at a fixed frequency using only the trailing window ending at each current sample. `closedLoop()` binds its configured detection-to-stimulation delay before registration so the event reports the predicted phase at the due time. Explicit signal-timestamp intervals must remain within five percent of the declared sampling interval.

Usage

```
phaseTargetOp(
  channel,
  sampling_rate,
  frequency,
  target_degrees,
  window_cycles = 3,
  tolerance_degrees = 20,
  min_amplitude = 0,
  min_fit = 0.8
)
```

Arguments

<code>channel</code>	Exact input channel name.
<code>sampling_rate</code>	Sampling rate in hertz.
<code>frequency</code>	Target oscillation frequency in hertz.
<code>target_degrees</code>	Requested phase in $[0, 360)$, modulo 360.
<code>window_cycles</code>	Number of trailing cycles in the causal fit.
<code>tolerance_degrees</code>	Maximum accepted predicted signed phase error.
<code>min_amplitude</code>	Minimum fitted oscillation amplitude.
<code>min_fit</code>	Minimum coefficient of determination for the sinusoid fit.

Value

A governed PipelineOperation detector descriptor.

Examples

```
sr <- 500
t <- seq_len(800) / sr
eeg <- 1.7 + 2.5 * cos(2 * pi * 9 * t + 0.7)
detector <- phaseTargetOp("eeg", sampling_rate = sr, frequency = 9,
  target_degrees = 90)
```

```

pipeline <- streamPipeline(chunk_size = length(eeg))
onChunk(pipeline, detector)
pipelineEnqueue(pipeline,
  matrix(eeg, ncol = 1, dimnames = list(NULL, "eeg")), t)
length(pipelineStep(pipeline)$results[[1]]$events)

```

pipeline-state	<i>Inspect or reset governed pipeline state</i>
----------------	---

Description

Inspect or reset governed pipeline state

Usage

```

pipelineState(pipeline)

pipelineReset(pipeline, keep_operations = TRUE)

```

Arguments

pipeline	A StreamPipeline.
keep_operations	Whether reset retains the registered graph and resets each operation to its initial state.

Value

pipelineState() returns a deep plain list; pipelineReset() returns the pipeline invisibly.

Examples

```

pipeline <- streamPipeline(chunk_size = 8L)
pipelineState(pipeline)$configuration$chunk_size
pipelineReset(pipeline)

```

pipelineEnqueue	<i>Enqueue one bounded input chunk</i>
-----------------	--

Description

Full-queue behavior is governed by the pipeline's exact backpressure policy. Empty chunks are no-ops and consume neither capacity nor sequence identity.

Usage

```

pipelineEnqueue(pipeline, samples, timestamps = NULL, ingest_time_ns = NULL)

```

Arguments

pipeline A StreamPipeline.
 samples Finite sample-by-channel values.
 timestamps Optional strictly increasing row-matched timestamps.
 ingest_time_ns Optional process-local monotonic nanosecond stamp.

Value

pipeline, invisibly.

Examples

```

pipeline <- streamPipeline(chunk_size = 4L)
x <- matrix(1:8, 4L, 2L, dimnames = list(NULL, c("a", "b")))
pipelineEnqueue(pipeline, x, c(1, 2, 3, 4))
pipelineState(pipeline)$channel_names

```

pipelineRun

Pull from a configured source and process synchronously

Description

The source must already be open. The function never starts a background thread and stops on an empty pull rather than busy-spinning.

Usage

```
pipelineRun(pipeline, max_chunks = Inf, timeout = 0)
```

Arguments

pipeline A source-backed StreamPipeline.
 max_chunks Maximum chunks to process; Inf is allowed.
 timeout Finite non-negative overall run time in seconds. Zero means no wall-time limit for the current call.

Value

A plain list of processed results and stop diagnostics.

Examples

```

info <- streamInfo("demo", type = "EEG",
                  channel_names = "C3", nominal_srate = 100)
src <- streamOpen(loopbackSource(info, capacity = 64L))
loopbackFeed(src, matrix(sin(seq_len(16))), 16, 1, seq_len(16) / 100)
sos <- matrix(c(1, 0, 0, 1, 0, 0), 1L, 6L,
             dimnames = list(NULL, c("b0", "b1", "b2", "a0", "a1", "a2")))
pipeline <- streamPipeline(source = src, chunk_size = 8L)
onChunk(pipeline, bandpassRmsOp(sos, window_samples = 4L))
run <- pipelineRun(pipeline)
run$n_processed

```

pipelineStep

Process queued chunks synchronously

Description

Process queued chunks synchronously

Usage

```
pipelineStep(pipeline, n = 1L)
```

Arguments

pipeline	A StreamPipeline.
n	Maximum whole chunks to process.

Value

A plain list containing per-chunk results and counts. Each result includes event_sources, a plain operation identity parallel to events.

Examples

```

sos <- matrix(c(1, 0, 0, 1, 0, 0), 1L, 6L,
             dimnames = list(NULL, c("b0", "b1", "b2", "a0", "a1", "a2")))
pipeline <- streamPipeline(chunk_size = 8L)
onChunk(pipeline, bandpassRmsOp(sos, window_samples = 4L))
x <- matrix(sin(seq_len(16))), 8, 2, dimnames = list(NULL, c("C3", "C4")))
pipelineEnqueue(pipeline, x, ingest_time_ns = 0)
pipelineStep(pipeline)$n_processed

```

processorState	<i>Inspect or reset governed streaming processor state</i>
----------------	--

Description

processorState() returns a deep, portable copy of the numeric processor state. processorReset() clears learned and delay state, increments the reset counter, and can retain the bound channel identity.

Usage

```
processorState(object, ...)

processorReset(object, keep_channels = TRUE, ...)
```

Arguments

object	A StreamProcessor.
...	Reserved for methods.
keep_channels	Whether to retain channel identity across reset.

Value

processorState() returns a plain list. processorReset() returns object invisibly.

Examples

```
filt <- lmsFilter(n_taps = 3L, step_size = 0.05)
update(filt, matrix(1:8, 4L, 2L), reference = matrix(1:4, 4L, 1L))
processorState(filt)$n_samples
processorReset(filt)
processorState(filt)$n_samples
```

readXDF	<i>Read an Extensible Data Format file</i>
---------	--

Description

XDF timestamps are authoritative in rowData(x)\$xdf_time; the regular streamTimeIndex() grid of the returned multi-rate container is only a nominal convenience for jittered or irregular streams. Files and converted payloads are also bounded by a private 512 MiB in-memory ceiling even when max_file_bytes is larger.

Usage

```
readXDF(  
  path,  
  streams = NULL,  
  synchronize = TRUE,  
  dejitter = TRUE,  
  keep_raw_timestamps = TRUE,  
  max_file_bytes = 2^31 - 1,  
  backend = getOption("PhysioStream.xdf_backend", "auto")  
)
```

Arguments

path	Existing local .xdf file.
streams	Optional unique stream ids or unambiguous stream names.
synchronize	Whether pyxdf clock synchronization is applied.
dejitter	Whether pyxdf timestamp de-jittering is applied.
keep_raw_timestamps	Whether a second uncorrected pass is retained.
max_file_bytes	Exact positive file-size ceiling.
backend	Exact configured backend.

Value

A valid [PhysioExperiment::MultiPhysioExperiment](#).

Examples

```
# Reading XDF requires the pyxdf backend (via reticulate).  
if (xdfAvailable()) {  
  path <- system.file("extdata", "xdf-minimal.xdf",  
                      package = "PhysioStream")  
  container <- readXDF(path)  
}
```

ring-read	<i>Pull or inspect buffered samples</i>
-----------	---

Description

Pull or inspect buffered samples

Usage

```
ringPull(x, n = ringFill(x))

ringPeek(x, n = ringFill(x), from = "oldest")
```

Arguments

x A live RingBuffer.

n Non-negative exact number of rows. Requests larger than the current fill return all available rows.

from Exact peek side, either "oldest" or "latest".

Value

A named list with samples, timestamps, sequence, count, and stats_after.

Examples

```
info <- streamInfo("demo", type = "EEG",
  channel_names = c("C3", "C4"), nominal_srate = 100)
buffer <- ringBuffer(info, capacity = 8L)
ringPush(buffer, matrix(as.double(1:6), 3, 2), c(0.01, 0.02, 0.03))
ringPeek(buffer, 2L)$samples
ringPull(buffer)$sequence
```

ring-state	<i>Ring-buffer capacity and occupancy</i>
------------	---

Description

Ring-buffer capacity and occupancy

Usage

```
ringCapacity(x)

ringFill(x)

ringStats(x)
```

Arguments

x A live RingBuffer.

Value

ringCapacity() and ringFill() return integer scalars. ringStats() returns a serializable named list.

Examples

```

info <- streamInfo("demo", type = "EEG",
                  channel_names = c("C3", "C4"), nominal_srate = 100)
buffer <- ringBuffer(info, capacity = 8L)
ringPush(buffer, matrix(as.double(1:6), 3, 2), c(0.01, 0.02, 0.03))
ringCapacity(buffer)
ringFill(buffer)
ringStats(buffer)$fill

```

ringBuffer	<i>Construct a native ring buffer</i>
------------	---------------------------------------

Description

Construct a native ring buffer

Usage

```
ringBuffer(info, capacity)
```

Arguments

info	Numeric StreamInfo.
capacity	Exact positive sample capacity.

Value

A live RingBuffer.

Examples

```

info <- streamInfo("demo", type = "EEG",
                  channel_names = c("C3", "C4"), nominal_srate = 100)
buffer <- ringBuffer(info, capacity = 16L)
ringCapacity(buffer)

```

RingBuffer-class	<i>Live bounded ring buffer</i>
------------------	---------------------------------

Description

Copying a RingBuffer wrapper aliases the same native buffer. The native queue is not serialized; a restored wrapper fails on its first operation.

Slots

ptr Protected native pointer.
 info Immutable stream metadata.
 capacity Fixed sample capacity.
 schema_version Wrapper schema.

Examples

```
info <- streamInfo("demo", type = "EEG",
                  channel_names = c("C3", "C4"), nominal_srate = 100)
buffer <- ringBuffer(info, capacity = 8L)
ringPush(buffer, matrix(as.double(1:4), 2, 2), c(0.01, 0.02))
ringFill(buffer)
```

ringPush	<i>Push samples into a ring buffer</i>
----------	--

Description

The push is transactional. When unread rows are overwritten, the oldest rows are dropped and counted. Matrices are always sample by channel.

Usage

```
ringPush(x, samples, timestamps)
```

Arguments

x	A live RingBuffer.
samples	A finite real matrix, sample by channel.
timestamps	Strictly increasing finite timestamps.

Value

Updated serializable ring statistics, invisibly.

Examples

```
info <- streamInfo("demo", type = "EEG",
                  channel_names = c("C3", "C4"), nominal_srate = 100)
buffer <- ringBuffer(info, capacity = 8L)
ringPush(buffer, matrix(as.double(1:4), 2, 2), c(0.01, 0.02))
ringFill(buffer)
```

ringReset	<i>Reset unread ring-buffer state</i>
-----------	---------------------------------------

Description

Reset clears unread rows and the last-timestamp constraint while preserving lifetime counters and monotonic sequence identity.

Usage

```
ringReset(x)
```

Arguments

x A live RingBuffer.

Value

Updated serializable ring statistics, invisibly.

Examples

```
info <- streamInfo("demo", type = "EEG",
                  channel_names = c("C3", "C4"), nominal_srate = 100)
buffer <- ringBuffer(info, capacity = 8L)
ringPush(buffer, matrix(as.double(1:4), 2, 2), c(0.01, 0.02))
ringReset(buffer)
ringFill(buffer)
```

rlsFilter	<i>Stateful recursive-least-squares adaptive filter</i>
-----------	---

Description

Stateful recursive-least-squares adaptive filter

Usage

```
rlsFilter(n_taps, forgetting = 0.99, delta = 1, initial_weights = NULL)
```

Arguments

n_taps	Exact adaptive-filter order.
forgetting	Forgetting factor in $(0, 1]$.
delta	Positive initial inverse-covariance scale.
initial_weights	Optional finite n_taps by channel matrix.

Value

A mutable RLSFilter streaming processor.

Examples

```
set.seed(1)
reference <- cos(2 * pi * 6 * seq_len(150) / 100)
signal <- 0.5 * reference + rnorm(150, sd = 0.05)
filt <- rlsFilter(n_taps = 3L, forgetting = 0.995)
result <- update(filt, matrix(signal, ncol = 1),
                 reference = matrix(reference, ncol = 1))
length(result$output)
```

sendStim

Send a governed stimulation command

Description

Validation, arm/dead-man, channel maximum, duration, duplicate-ID, and refractory gates run before transport invocation. Once invocation begins, command identity and refractory time are conservatively reserved even when acknowledgement is unknown. Commands are never retried automatically.

Usage

```
sendStim(trigger, intensity, channel, duration_ms, command_id, now_ns = NULL)
```

Arguments

trigger	A TriggerBackend.
intensity	Finite non-negative intensity in the configured unit.
channel	Exact configured channel identifier.
duration_ms	Finite positive command duration in milliseconds.
command_id	Unique bounded command identifier.
now_ns	Optional exact process-local monotonic nanosecond value.

Details

MQTT acknowledgement means broker/client acceptance; TTL acknowledgement means adapter completion. Neither status proves physical stimulation. Receipts and audit records bind life-cycle and arm generations, validation, attempt, and acknowledgement times, and bounded error class/code. A caller interrupt after transport invocation first finalizes an unknown audit record and then re-signals the interrupt.

Value

A deep plain-list command receipt.

Examples

```
trigger <- loopbackTrigger(allowed_channels = "left", max_intensity = 20,
                           intensity_unit = "mA", max_duration_ms = 500,
                           refractory_ms = 0, deadman_ms = 1000)
triggerOpen(trigger)
armTrigger(trigger, "session-1", now_ns = 0)
receipt <- sendStim(trigger, intensity = 2.5, channel = "left",
                   duration_ms = 125, command_id = "cmd-1", now_ns = 10)
receipt$status
```

slidingSTFT

Sliding short-time Fourier transform

Description

Sliding short-time Fourier transform

Usage

```
slidingSTFT(
  sampling_rate,
  window_samples,
  hop_samples,
  n_fft = window_samples,
  window = c("hann", "hamming", "rectangular"),
  detrend = c("mean", "none"),
  output = c("power", "complex"),
  bands = NULL,
  causal_filter = NULL,
  n_features = NULL
)
```

Arguments

sampling_rate	Positive samples per second.
window_samples	Exact frame length.
hop_samples	Exact frame advance, no larger than the window.
n_fft	Exact FFT length, at least the frame length.
window	Exact governed window definition.
detrend	Per-frame mean removal or no detrending.
output	Return one-sided density power or normalized complex coefficients whose squared modulus equals that density.
bands	Optional uniquely named two-column Hz matrix.
causal_filter	Optional independent PhysioPreprocess::StreamFilter objects or factory.
n_features	Optional channel count to bind at construction.

Value

A mutable SlidingSTFT streaming processor.

Examples

```
sr <- 100
signal <- sin(2 * pi * 10 * seq_len(200) / sr)
stft <- slidingSTFT(sr, window_samples = 64L, hop_samples = 16L,
                    output = "power")
result <- update(stft, signal)
dim(result$output)
```

stream-metadata	<i>Stream metadata accessors</i>
-----------------	----------------------------------

Description

Stream metadata accessors

Usage

```
streamName(x)

streamType(x)

streamChannels(x)

streamRate(x)

streamDtype(x)
```

Arguments

`x` A StreamInfo, endpoint, or ring buffer.

Value

The requested metadata field.

Examples

```
info <- streamInfo("eeg-demo", type = "EEG",
                  channel_names = c("C3", "C4"), nominal_srate = 250,
                  dtype = "float32")
streamName(info)
streamType(info)
streamChannels(info)
streamRate(info)
streamDtype(info)
```

stream-operations

Stream endpoint lifecycle and transport operations

Description

Transport packages extend these generics. Base virtual endpoints provide no implicit transport.

Usage

```
streamOpen(x, ...)

streamClose(x, ...)

streamPull(x, ...)

streamPush(x, ...)

streamState(x)

## S4 method for signature 'StreamEndpoint'
streamState(x)

## S4 method for signature 'StreamEndpoint'
streamOpen(x, ...)

## S4 method for signature 'StreamEndpoint'
streamClose(x, ...)

## S4 method for signature 'StreamEndpoint'
```

```
streamPull(x, ...)

## S4 method for signature 'StreamEndpoint'
streamPush(x, ...)

## S4 method for signature 'LoopbackSource'
streamOpen(x, ...)

## S4 method for signature 'LoopbackSource'
streamClose(x, ...)

## S4 method for signature 'LoopbackSource'
streamPull(x, ...)

## S4 method for signature 'LoopbackSource'
streamPush(x, ...)

## S4 method for signature 'LSLInlet'
streamState(x)

## S4 method for signature 'LSLOutlet'
streamState(x)

## S4 method for signature 'LSLInlet'
streamOpen(x, timeout = 2, ...)

## S4 method for signature 'LSLOutlet'
streamOpen(x, ...)

## S4 method for signature 'LSLInlet'
streamClose(x, ...)

## S4 method for signature 'LSLOutlet'
streamClose(x, ...)

## S4 method for signature 'LSLInlet'
streamPull(x, ...)

## S4 method for signature 'LSLOutlet'
streamPush(x, ...)
```

Arguments

x	A stream endpoint.
...	Backend-specific arguments.
timeout	Optional finite non-negative LSL inlet open timeout.

Value

Backend-specific output.

Examples

```
# Drive the lifecycle on a device-free loopback source.
info <- streamInfo("demo", type = "EEG",
                  channel_names = c("C3", "C4"), nominal_srate = 100)
src <- streamOpen(loopbackSource(info, capacity = 16L))
streamState(src)
loopbackFeed(src, matrix(as.double(1:4), 2, 2), c(0.01, 0.02))
streamPull(src)$samples
src <- streamClose(src)
streamState(src)
```

StreamEndpoint-class *Virtual stream endpoint*

Description

Virtual stream endpoint

Slots

`info` Immutable StreamInfo.

`state` Exact lifecycle state.

`audit` Append-only state transition records.

Examples

```
# StreamEndpoint is virtual; a loopback source is a concrete endpoint.
info <- streamInfo("demo", type = "EEG",
                  channel_names = "C3", nominal_srate = 100)
src <- loopbackSource(info)
is(src, "StreamEndpoint")
streamState(src)
```

streamInfo	<i>Construct or retrieve stream metadata</i>
------------	--

Description

With a character first argument, constructs validated stream metadata. With a stream endpoint or ring buffer, retrieves its immutable StreamInfo.

Usage

```
streamInfo(name, ...)

## S4 method for signature 'character'
streamInfo(
  name,
  type,
  channel_names,
  nominal_srate,
  dtype = c("float64", "float32", "int32", "int16", "int8", "string"),
  source_id = "",
  clock_domain = "local",
  channel_units = NULL,
  metadata = list(),
  ...
)

## S4 method for signature 'StreamInfo'
streamInfo(name, ...)

## S4 method for signature 'StreamEndpoint'
streamInfo(name, ...)

## S4 method for signature 'RingBuffer'
streamInfo(name, ...)
```

Arguments

name	A non-empty stream name, or a stream object when used as an accessor.
...	Reserved for methods.
type	A non-empty stream type.
channel_names	Unique non-empty channel labels.
nominal_srate	A finite nominal sampling rate greater than or equal to zero.
dtype	Exact storage type.
source_id	Optional stable source identifier.
clock_domain	Non-empty clock-domain label.

channel_units Optional units matching channel_names.
 metadata Named recursively serializable metadata, at most 1 MiB.

Value

A StreamInfo object, or an endpoint's StreamInfo.

Examples

```
info <- streamInfo("eeg-demo", type = "EEG",
  channel_names = c("C3", "Cz", "C4"),
  nominal_srate = 250, channel_units = rep("uV", 3))
info
streamChannels(info)
```

StreamInfo-class	<i>Validated stream metadata</i>
------------------	----------------------------------

Description

Validated stream metadata

Usage

```
## S4 method for signature 'StreamInfo'
show(object)
```

Arguments

object A StreamInfo object to display.

Slots

name,type,source_id,clock_domain Scalar stream identity strings.
 n_channels Exact positive channel count.
 channel_names,channel_units Channel labels and optional units.
 nominal_srate Nominal samples per second; zero denotes an irregular stream reserved for later transports.
 dtype Exact storage type.
 metadata Named recursively serializable metadata.
 schema_version Metadata schema identifier.

Examples

```
info <- streamInfo("eeg-demo", type = "EEG",
  channel_names = c("C3", "C4"), nominal_srate = 250)
info
```

streamPipeline	<i>Construct a governed synchronous stream-processing pipeline</i>
----------------	--

Description

A pipeline owns a bounded FIFO of whole chunks and an ordered callback graph. Processing occurs synchronously on the caller's R thread. The runtime environment is not portable; use [pipelineState\(\)](#) for a hashed plain-state snapshot.

Usage

```
streamPipeline(  
  source = NULL,  
  chunk_size = 32L,  
  queue_capacity = 64L,  
  backpressure = c("error", "drop_oldest", "drop_newest"),  
  latency_budget_ms = 50  
)
```

Arguments

source	Optional StreamSource. Caller-driven pipelines use NULL.
chunk_size	Maximum samples per enqueued chunk.
queue_capacity	Maximum number of whole chunks in the ingress FIFO.
backpressure	Exact full-queue policy.
latency_budget_ms	Finite positive empirical latency budget.

Value

A StreamPipeline environment.

Examples

```
pipeline <- streamPipeline(chunk_size = 8L)  
pipelineState(pipeline)$configuration$chunk_size
```

StreamSink-class	<i>Virtual stream sink</i>
------------------	----------------------------

Description

Transport backends extend this class and the stream-operation generics.

Examples

```
# StreamSink is virtual; transport outlets such as lslOutlet() extend it.
isVirtualClass("StreamSink")
```

streamSnapshot	<i>Snapshot an open numeric stream</i>
----------------	--

Description

Original timestamps and monotonic stream sequence identity are retained. Selection is non-consuming unless consume = TRUE; consuming a newest subset while older unread rows remain is rejected.

Usage

```
streamSnapshot(
  x,
  n = NULL,
  duration_seconds = NULL,
  consume = FALSE,
  assay_name = "stream"
)
```

Arguments

- x An open buffered StreamSource.
- n Optional number of newest rows.
- duration_seconds Optional duration of the newest closed timestamp interval.
- consume Whether to consume the selected oldest contiguous unread rows.
- assay_name Non-empty assay name.

Value

A valid [PhysioExperiment::PhysioExperiment](#).

Examples

```
info <- streamInfo("demo", type = "EEG",
                   channel_names = c("C3", "C4"), nominal_srate = 100)
src <- streamOpen(loopbackSource(info, capacity = 32L))
loopbackFeed(src, matrix(sin(seq_len(10)), 5, 2),
              c(0.01, 0.02, 0.03, 0.04, 0.05))
pe <- streamSnapshot(src)
dim(pe)
```

StreamSource-class	<i>Virtual stream source</i>
--------------------	------------------------------

Description

Transport backends extend this class and the stream-operation generics.

Examples

```
# StreamSource is virtual; loopbackSource() returns a concrete instance.
info <- streamInfo("demo", type = "EEG",
                   channel_names = "C3", nominal_srate = 100)
is(loopbackSource(info), "StreamSource")
```

syncDiagnostics	<i>Summarize governed stream synchronization</i>
-----------------	--

Description

Returns stored diagnostics for a synchronized container. For an unsynchronized container, models or shared events are evaluated through syncStreams() on a copy and only the diagnostics table is returned.

Usage

```
syncDiagnostics(x, models = NULL, shared_events = NULL)
```

Arguments

x	A <code>PhysioExperiment::MultiPhysioExperiment</code> .
models	Optional uniquely named <code>StreamClockModel</code> list.
shared_events	Optional exact event_id, stream, timestamp table.

Value

One plain data-frame row per stream/reset segment.

Examples

```
## Not run:
# Requires a multi-stream container (MultiPhysioExperiment), e.g. from
# readXDF().
diagnostics <- syncDiagnostics(container)

## End(Not run)
```

syncStreams	<i>Synchronize a multi-rate stream container</i>
-------------	--

Description

Applies explicit clock models or recorded XDF/shared-event evidence without interpolating sample values. Authoritative raw and master-domain timestamps are appended to each stream's row data.

Usage

```
syncStreams(
  x,
  master = NULL,
  models = NULL,
  shared_events = NULL,
  dejitter = FALSE,
  max_residual_seconds = NULL
)
```

Arguments

<code>x</code>	A <code>PhysioExperiment::MultiPhysioExperiment</code> .
<code>master</code>	Exact master stream key, or <code>NULL</code> for deterministic choice.
<code>models</code>	Optional uniquely named <code>StreamClockModel</code> list.
<code>shared_events</code>	Optional exact <code>event_id</code> , <code>stream</code> , <code>timestamp</code> table.
<code>dejitter</code>	Whether to append a nominal-grid timestamp column.
<code>max_residual_seconds</code>	Optional non-negative residual gate.

Value

A `synchronized PhysioExperiment::MultiPhysioExperiment`.

Examples

```
## Not run:
# Requires a multi-stream container (MultiPhysioExperiment), e.g. from
# readXDF(); `master` names the reference stream key.
synced <- syncStreams(container, master = "eeg")

## End(Not run)
```

triggerOpen	<i>Open or close a governed stimulation trigger</i>
-------------	---

Description

Trigger constructors are side-effect free. `triggerOpen()` explicitly opens the configured transport and leaves it disarmed. `triggerClose()` performs a bounded best-effort stop/deassert before closing. Software interlocks are additional safeguards only; independently fail-safe stimulation hardware is required.

Usage

```
triggerOpen(trigger)

triggerClose(trigger)
```

Arguments

trigger A TriggerBackend.

Value

The trigger, invisibly.

Examples

```
trigger <- loopbackTrigger(allowed_channels = "left", max_intensity = 20,
                           intensity_unit = "mA", max_duration_ms = 500,
                           refractory_ms = 0, deadman_ms = 1000)

triggerOpen(trigger)
triggerState(trigger)$lifecycle$status
triggerClose(trigger)
```

triggerState	<i>Inspect governed trigger state</i>
--------------	---------------------------------------

Description

The returned deep plain-list snapshot contains no connection, Python object, callback, or credential. Runtime environments are not portable.

Usage

```
triggerState(trigger)
```

Arguments

trigger	A TriggerBackend.
---------	-------------------

Value

A sealed plain list.

Examples

```
trigger <- loopbackTrigger(allowed_channels = "left", max_intensity = 20,
                           intensity_unit = "mA", max_duration_ms = 500,
                           refractory_ms = 0, deadman_ms = 1000)

triggerOpen(trigger)
triggerState(trigger)$arm$status
```

ttlTrigger	<i>Construct a governed TTL stimulation-command trigger</i>
------------	---

Description

TTL transport is an adapter boundary, not a vendor stimulator protocol. Serial mode writes a versioned ASCII frame to a cooperating independently fail-safe adapter. Callback mode requires one atomic timed pulse operation; PhysioStream does not synthesize pulse timing with an R sleep or busy loop. Loopback mode records the exact frame without hardware.

Usage

```
ttlTrigger(
  transport = c("serial", "callback", "loopback"),
  allowed_channels,
  max_intensity,
  intensity_unit,
  max_duration_ms,
```

```

    refractory_ms,
    deadman_ms,
    port = NULL,
    baud = 115200L,
    line = 1L,
    pulse_width_ms = 5,
    write_timeout_ms = 100,
    writer = NULL,
    audit_capacity = 4096L,
    clock = NULL
)

```

Arguments

transport	Exact "serial", "callback", or "loopback".
allowed_channels	Unique allowed channel identifiers.
max_intensity	Positive scalar or exactly channel-named maxima.
intensity_unit	Explicit device intensity unit; no conversion occurs.
max_duration_ms	Positive scalar or channel-named duration maxima.
refractory_ms	Non-negative scalar or channel-named refractory times.
deadman_ms	Positive heartbeat expiry interval.
port	Serial port path for serial mode.
baud	Positive integer serial baud rate.
line	Positive integer adapter line identifier.
pulse_width_ms	Positive pulse width requested from the adapter.
write_timeout_ms	Positive bounded serial write timeout.
writer	Callback implementing atomic "pulse" and fail-safe "stop" actions in callback mode.
audit_capacity	Bounded command/audit capacity.
clock	Optional deterministic monotonic clock for tests.

Details

Opening some serial ports can momentarily affect RTS/DTR control lines. Hardware must remain fail-safe despite port open, process failure, malformed frames, and communication loss.

Value

A closed, disarmed TtlTrigger.

Examples

```
## Not run:
# Serial transport requires a device exposing a TTL line; see
# loopbackTrigger() for an offline equivalent with the same interlocks.
trigger <- ttlTrigger(transport = "serial", port = "/dev/ttyUSB0",
                      allowed_channels = "left", max_intensity = 20,
                      intensity_unit = "mA", max_duration_ms = 500,
                      refractory_ms = 0, deadman_ms = 1000)

## End(Not run)
```

video-sync-map

Map between synchronized signal and video time

Description

Map between synchronized signal and video time

Usage

```
videoSyncTime(sync, signal_time)
```

```
signalSyncTime(sync, video_time)
```

Arguments

sync	A validated VideoSync.
signal_time	Finite signal-domain times.
video_time	Finite media times.

Value

A numeric vector preserving names.

Examples

```
sync <- videoSync(c(a = 100, b = 110, c = 120), c(a = 1, b = 11, c = 21),
                 frame_rate = 30, method = "offset", clock_domain = "demo")
videoSyncTime(sync, c(x = 105, y = 115))
signalSyncTime(sync, c(x = 6, y = 16))
```

videoSync

Fit a governed marker-anchored video/signal time map

Description

The mapping convention is $\text{video_seconds} = \text{intercept} + \text{rate} * \text{signal_seconds}$. Evaluation uses an equivalent centered representation to retain precision for large clocks.

Usage

```
videoSync(
  signal_time,
  video_time,
  frame_rate,
  method = c("offset", "affine"),
  clock_domain,
  tolerance_frames = 1,
  max_rate_error_ppm = 5000
)
```

Arguments

signal_time	Strictly increasing signal-domain marker times.
video_time	Strictly increasing corresponding media times.
frame_rate	Positive media frames per second.
method	Exact offset-only or affine fit.
clock_domain	Exact signal clock-domain label.
tolerance_frames	Maximum accepted anchor residual in frames.
max_rate_error_ppm	Maximum affine rate deviation from one.

Value

A sealed portable VideoSync.

Examples

```
sync <- videoSync(
  signal_time = c(a = 100, b = 110, c = 120),
  video_time = c(a = 1, b = 11, c = 21),
  frame_rate = 30, method = "offset", clock_domain = "demo")
videoSyncTime(sync, c(x = 105))
```

videoSyncViewer	<i>Launch a synchronized video and signal viewer</i>
-----------------	--

Description

launch = FALSE returns a side-effect-free BiofeedbackScope carrying a validated video descriptor. Local media are copied and exposed only when the Shiny viewer starts.

Usage

```
videoSyncViewer(
  source,
  video,
  sync,
  pipeline = NULL,
  channels = NULL,
  window_seconds = 10,
  update_hz = 20,
  max_points = 2000L,
  source_lifecycle = c("borrow", "own"),
  launch = interactive(),
  host = "127.0.0.1",
  port = NULL,
  browser = interactive(),
  clock = NULL
)
```

Arguments

source	A regular-rate numeric StreamSource.
video	Local mp4/webm/ogg path, HTTPS URL, or "webcam".
sync	A VideoSync in the source clock domain.
pipeline	Optional source-backed StreamPipeline.
channels	Exact source channel names to display.
window_seconds	Positive visible signal-time window.
update_hz	Requested display update rate.
max_points	Maximum plotted representatives per trace.
source_lifecycle	Whether the scope borrows an open source or owns its open/close lifecycle.
launch	Whether to run the installed Shiny app immediately.
host	Local Shiny bind host.
port	Optional exact TCP port.
browser	Whether Shiny should launch a browser.
clock	Optional monotonic nanosecond clock for deterministic tests.

Value

A BiofeedbackScope or the result of shiny::runApp().

Examples

```
## Not run:
# Requires a Shiny session and a local video file for display.
info <- streamInfo("demo", type = "EEG",
  channel_names = c("left", "right"), nominal_srate = 100,
  channel_units = c("uV", "uV"))
source <- streamOpen(loopbackSource(info, capacity = 256L))
sync <- videoSync(c(a = 100, b = 110, c = 120), c(a = 1, b = 11, c = 21),
  frame_rate = 30, method = "offset", clock_domain = "local")
viewer <- videoSyncViewer(source, "session.mp4", sync, launch = FALSE)

## End(Not run)
```

welchOnline

Exponentially weighted online Welch spectrum

Description

Complete windows are converted to one-sided power spectral density and combined as $\alpha * \text{newest} + (1 - \alpha) * \text{previous}$.

Usage

```
welchOnline(
  sampling_rate,
  window_samples,
  hop_samples = floor(window_samples/2),
  n_fft = window_samples,
  alpha = 0.1,
  window = c("hann", "hamming", "rectangular"),
  detrend = c("mean", "none"),
  bands = NULL,
  causal_filter = NULL,
  n_features = NULL
)
```

Arguments

sampling_rate	Positive samples per second.
window_samples	Exact frame length.
hop_samples	Exact frame advance, no larger than the window.
n_fft	Exact FFT length, at least the frame length.
alpha	Weight assigned to the newest periodogram.

window	Exact governed window definition.
detrend	Per-frame mean removal or no detrending.
bands	Optional uniquely named two-column Hz matrix.
causal_filter	Optional independent PhysioPreprocess::StreamFilter objects or factory.
n_features	Optional channel count to bind at construction.

Value

A mutable OnlineWelch streaming processor.

Examples

```
sr <- 100
signal <- sin(2 * pi * 10 * seq_len(256) / sr)
welch <- welchOnline(sr, window_samples = 64L, hop_samples = 32L)
result <- update(welch, signal)
dim(result$output)
```

writeXDF	<i>Write an Extensible Data Format file</i>
----------	---

Description

The writer implements the governed XDF 1.0 subset natively and does not require Python. Output is scanned before an atomic same-directory rename.

Usage

```
writeXDF(
  x,
  path,
  overwrite = FALSE,
  chunk_samples = 256L,
  timestamps = c("output", "raw")
)
```

Arguments

x	One PhysioExperiment::MultiPhysioExperiment or PhysioExperiment::PhysioExperiment .
path	Destination .xdf path in an existing directory.
overwrite	Whether an existing regular destination may be replaced.
chunk_samples	Exact positive samples per XDF Samples chunk.
timestamps	Exact timestamp domain, "output" or "raw".

Value

The normalized output path, invisibly.

Examples

```
## Not run:
# Writing XDF requires the pyxdf backend and a multi-stream container,
# e.g. one returned by readXDF().
writeXDF(container, tempfile(fileext = ".xdf"))

## End(Not run)
```

xdfAvailable	<i>Test XDF backend availability</i>
--------------	--------------------------------------

Description

The default probe is conservative and does not initialize Python. Explicit read operations initialize only the already configured reticulate interpreter. `PhysioStream` never installs or selects a Python environment.

Usage

```
xdfAvailable(backend = c("auto", "pyxdf"), initialize = FALSE)
```

Arguments

backend	Exact backend, currently "auto" or "pyxdf".
initialize	Whether the configured Python interpreter may be initialized to verify pyxdf and NumPy.

Value

One non-missing logical value.

Examples

```
# Reports FALSE unless the pyxdf backend is installed.
xdfAvailable()
```

xdfBackendInfo	<i>Report the configured XDF backend</i>
----------------	--

Description

Report the configured XDF backend

Usage

```
xdfBackendInfo(backend = c("auto", "pyxdf"))
```

Arguments

backend Exact backend, currently "auto" or "pyxdf".

Value

A serializable named list of backend and runtime versions.

Examples

```
# Resolving the backend version requires the pyxdf backend.  
if (xdfAvailable()) str(xdfBackendInfo())
```

Index

armTrigger, 4

bandpassRmsOp, 5

biofeedback-lifecycle, 6

biofeedback-state, 6

biofeedbackFrame (biofeedback-state), 6

biofeedbackScope, 7

biofeedbackStart
 (biofeedback-lifecycle), 6

biofeedbackStart(), 7

biofeedbackState (biofeedback-state), 6

biofeedbackStep
 (biofeedback-lifecycle), 6

biofeedbackStep(), 7

biofeedbackStop
 (biofeedback-lifecycle), 6

biofeedbackUpdate, 9

clockOffset, 10

closed-loop-lifecycle, 11

closed-loop-state, 12

closedLoop, 13

closedLoop(), 37

closedLoopEmergencyStop
 (closed-loop-lifecycle), 11

closedLoopFlush
 (closed-loop-lifecycle), 11

closedLoopFlush(), 13

closedLoopLog (closed-loop-state), 12

closedLoopProvenance, 14

closedLoopStart
 (closed-loop-lifecycle), 11

closedLoopState (closed-loop-state), 12

closedLoopStep (closed-loop-lifecycle),
 11

closedLoopStep(), 17

closedLoopStop (closed-loop-lifecycle),
 11

dejitter, 15

disarmTrigger (armTrigger), 4

driftCorrect, 16

emergencyStop (armTrigger), 4

emgOnsetOp, 17

emgOnsetOp(), 13

erdIntentOp, 18

erdIntentOp(), 13

incrementalPCA, 19

lmsFilter, 20

loopbackFeed, 20

loopbackSource, 21

LoopbackSource-class, 22

loopbackTrigger, 22

lslAvailable, 23

lslBackendInfo, 24

lslInlet, 24

LSLInlet-class, 25

lslMarkerEvents, 26

lslOutlet, 27

LSLOutlet-class, 28

lslPull, 28

lslPush, 29

lslResolveStreams, 30

measureLatency, 31

mqttAvailable, 32

mqttBackendInfo (mqttAvailable), 32

mqttTrigger, 32

nlmsFilter, 34

onChunk, 35

onChunk(), 5, 17

onlineICA, 36

phaseTargetOp, 37

phaseTargetOp(), 13

PhysioExperiment::MultiPhysioExperiment,
 42, 66
 PhysioExperiment::PhysioExperiment, 56,
 66
 PhysioPreprocess::StreamFilter, 49, 66
 pipeline-state, 38
 pipelineEnqueue, 38
 pipelineReset (pipeline-state), 38
 pipelineRun, 39
 pipelineState (pipeline-state), 38
 pipelineState(), 55
 pipelineStep, 40
 pipelineStep(), 13
 processorReset (processorState), 41
 processorState, 41

 readXDF, 41
 ring-read, 42
 ring-state, 43
 ringBuffer, 44
 RingBuffer-class, 45
 ringCapacity (ring-state), 43
 ringFill (ring-state), 43
 ringPeek (ring-read), 42
 ringPull (ring-read), 42
 ringPush, 45
 ringReset, 46
 ringStats (ring-state), 43
 rlsFilter, 46

 sendStim, 47
 show, StreamInfo-method
 (StreamInfo-class), 54
 signalSyncTime (video-sync-map), 62
 slidingSTFT, 48
 stream-metadata, 49
 stream-operations, 50
 streamChannels (stream-metadata), 49
 streamClose (stream-operations), 50
 streamClose, LoopbackSource-method
 (stream-operations), 50
 streamClose, LSLInlet-method
 (stream-operations), 50
 streamClose, LSLOutlet-method
 (stream-operations), 50
 streamClose, StreamEndpoint-method
 (stream-operations), 50
 streamDtype (stream-metadata), 49
 StreamEndpoint-class, 52

streamInfo, 53
 streamInfo, character-method
 (streamInfo), 53
 streamInfo, RingBuffer-method
 (streamInfo), 53
 streamInfo, StreamEndpoint-method
 (streamInfo), 53
 streamInfo, StreamInfo-method
 (streamInfo), 53
 StreamInfo-class, 54
 streamName (stream-metadata), 49
 streamOpen (stream-operations), 50
 streamOpen, LoopbackSource-method
 (stream-operations), 50
 streamOpen, LSLInlet-method
 (stream-operations), 50
 streamOpen, LSLOutlet-method
 (stream-operations), 50
 streamOpen, StreamEndpoint-method
 (stream-operations), 50
 streamPipeline, 55
 streamPull (stream-operations), 50
 streamPull, LoopbackSource-method
 (stream-operations), 50
 streamPull, LSLInlet-method
 (stream-operations), 50
 streamPull, StreamEndpoint-method
 (stream-operations), 50
 streamPush (stream-operations), 50
 streamPush, LoopbackSource-method
 (stream-operations), 50
 streamPush, LSLOutlet-method
 (stream-operations), 50
 streamPush, StreamEndpoint-method
 (stream-operations), 50
 streamRate (stream-metadata), 49
 StreamSink-class, 56
 streamSnapshot, 56
 StreamSource-class, 57
 streamState (stream-operations), 50
 streamState, LSLInlet-method
 (stream-operations), 50
 streamState, LSLOutlet-method
 (stream-operations), 50
 streamState, StreamEndpoint-method
 (stream-operations), 50
 streamType (stream-metadata), 49
 syncDiagnostics, 57

syncStreams, [58](#)

triggerClose (triggerOpen), [59](#)

triggerHeartbeat (armTrigger), [4](#)

triggerOpen, [59](#)

triggerState, [60](#)

ttlTrigger, [60](#)

video-sync-map, [62](#)

videoSync, [63](#)

videoSyncTime (video-sync-map), [62](#)

videoSyncViewer, [64](#)

welchOnline, [65](#)

writeXDF, [66](#)

xdfAvailable, [67](#)

xdfBackendInfo, [68](#)