Package 'STMr'

November 2, 2023

Title Strength Training Manual R-Language Functions

Version 0.1.6

Description Strength training prescription using percent-based approach requires numerous computations and assumptions. 'STMr' package allow users to estimate individual reps-max relationships, implement various progression tables, and create numerous set and rep schemes. The 'STMr' package is originally created as a tool to help writing Jovanović M. (2020) Strength Training Manual <ISBN:979-8604459898>.

License MIT + file LICENSE

Encoding UTF-8

LazyData true

RoxygenNote 7.2.3

URL https://mladenjovanovic.github.io/STMr/

BugReports https://github.com/mladenjovanovic/STMr/issues

Imports dplyr, ggfittext, ggplot2, magrittr, minpack.lm, nlme, quantreg, stats, tidyr

Suggests testthat (>= 3.0.0)

Depends R (>= 2.10)

Config/testthat/edition 3

NeedsCompilation no

Author Mladen Jovanović [aut, cre]

Maintainer Mladen Jovanović <coach.mladen.jovanovic@gmail.com>

Repository CRAN

Date/Publication 2023-11-02 10:50:02 UTC

R topics documented:

`+.STMr_scheme` *Method for adding set and rep schemes*

Description

Method for adding set and rep schemes

Usage

```
## S3 method for class 'STMr_scheme'
lhs + rhs
```

Arguments

`lhs`	`STMr_scheme` object
`rhs`	`STMr_scheme` object

Value

`STMr_scheme` object

Examples

```
scheme1 <- scheme_wave()
warmup_scheme <- scheme_perc_1RM()
plot(warmup_scheme + scheme1)
```

`adj_perc_1RM` *Family of functions to adjust %1RM*

Description

Family of functions to adjust %1RM

Usage

```
adj_perc_1RM_RIR(
  reps,
  adjustment = 0,
  mfactor = 1,
  max_perc_1RM_func = max_perc_1RM_epley,
  ...
)

adj_perc_1RM_DI(
  reps,
  adjustment = 0,
  mfactor = 1,
  max_perc_1RM_func = max_perc_1RM_epley,
  ...
)

adj_perc_1RM_rel_int(
  reps,
  adjustment = 1,
  mfactor = 1,
  max_perc_1RM_func = max_perc_1RM_epley,
  ...
)

adj_perc_1RM_perc_MR(
  reps,
  adjustment = 1,
  mfactor = 1,
  max_perc_1RM_func = max_perc_1RM_epley,
  ...
)
```

Arguments

reps	Numeric vector. Number of repetition to be performed
adjustment	Numeric vector. Adjustment to be implemented
mfactor	Numeric vector. Default is 1 (i.e., no adjustment). Use `mfactor = 2` to generate ballistic adjustment and tables
max_perc_1RM_func	Max %1RM function to be used. Default is `max_perc_1RM_epley`
...	Forwarded to `max_perc_1RM_func`. Usually the parameter value. For example `klin = 36` when using `max_perc_1RM_linear` as `max_perc_1RM_func` function

Value

Numeric vector. Predicted perc 1RM

Functions

- `adj_perc_1RM_RIR()`: Adjust max %1RM using the Reps In Reserve (RIR) approach
- `adj_perc_1RM_DI()`: Adjust max %1RM using the Deducted Intensity (DI) approach. This approach simple deducts `adjustment` from estimated %1RM
- `adj_perc_1RM_rel_int()`: Adjust max perc 1RM using the Relative Intensity (RelInt) approach. This approach simple multiplies estimated perc 1RM with `adjustment`
- `adj_perc_1RM_perc_MR()`: Adjust max perc 1RM using the %Max Reps (%MR) approach. This approach simple divides target reps with `adjustment`

Examples

```
# ------------------------------------------
# Adjustment using Reps In Reserve (RIR)
adj_perc_1RM_RIR(5)

# Use ballistic adjustment (this implies doing half the reps)
adj_perc_1RM_RIR(5, mfactor = 2)

# Use 2 reps in reserve
adj_perc_1RM_RIR(5, adjustment = 2)

# Use Linear model
adj_perc_1RM_RIR(5, max_perc_1RM_func = max_perc_1RM_linear, adjustment = 2)

# Use Modifed Epley's equation with a custom parameter values
adj_perc_1RM_RIR(
  5,
  max_perc_1RM_func = max_perc_1RM_modified_epley,
  adjustment = 2,
  kmod = 0.06
)
# ------------------------------------------
# Adjustment using Deducted Intensity (DI)
```

```
adj_perc_1RM_DI(5)

# Use ballistic adjustment (this implies doing half the reps)
adj_perc_1RM_DI(5, mfactor = 2)

# Use 10 perc deducted intensity
adj_perc_1RM_DI(5, adjustment = -0.1)

# Use Linear model
adj_perc_1RM_DI(5, max_perc_1RM_func = max_perc_1RM_linear, adjustment = -0.1)

# Use Modifed Epley's equation with a custom parameter values
adj_perc_1RM_DI(
  5,
  max_perc_1RM_func = max_perc_1RM_modified_epley,
  adjustment = -0.1,
  kmod = 0.06
)
# ------------------------------------------
# Adjustment using Relative Intensity (RelInt)
adj_perc_1RM_rel_int(5)

# Use ballistic adjustment (this implies doing half the reps)
adj_perc_1RM_rel_int(5, mfactor = 2)

# Use 90 perc  relative intensity
adj_perc_1RM_rel_int(5, adjustment = 0.9)

# Use Linear model
adj_perc_1RM_rel_int(5, max_perc_1RM_func = max_perc_1RM_linear, adjustment = 0.9)

# Use Modifed Epley's equation with a custom parameter values
adj_perc_1RM_rel_int(
  5,
  max_perc_1RM_func = max_perc_1RM_modified_epley,
  adjustment = 0.9,
  kmod = 0.06
)
# ------------------------------------------
# Adjustment using % max reps (%MR)
adj_perc_1RM_perc_MR(5)

# Use ballistic adjustment (this implies doing half the reps)
adj_perc_1RM_perc_MR(5, mfactor = 2)

# Use 70 perc max reps
adj_perc_1RM_perc_MR(5, adjustment = 0.7)

# Use Linear model
adj_perc_1RM_perc_MR(5, max_perc_1RM_func = max_perc_1RM_linear, adjustment = 0.7)

# Use Modifed Epley's equation with a custom parameter values
adj_perc_1RM_perc_MR(
```

```
  5,
  max_perc_1RM_func = max_perc_1RM_modified_epley,
  adjustment = 0.7,
  kmod = 0.06
)
```

---

adj_reps *Family of functions to adjust number of repetition*

Description

These functions are reverse version of the adj_perc_1RM family of functions. Use these when you want to estimate number of repetitions to be used when using the known %1RM and level of adjustment

Usage

```
adj_reps_RIR(
  perc_1RM,
  adjustment = 0,
  mfactor = 1,
  max_reps_func = max_reps_epley,
  ...
)

adj_reps_DI(
  perc_1RM,
  adjustment = 1,
  mfactor = 1,
  max_reps_func = max_reps_epley,
  ...
)

adj_reps_rel_int(
  perc_1RM,
  adjustment = 1,
  mfactor = 1,
  max_reps_func = max_reps_epley,
  ...
)

adj_reps_perc_MR(
  perc_1RM,
  adjustment = 1,
  mfactor = 1,
  max_reps_func = max_reps_epley,
  ...
)
```

Arguments

perc_1RM	Numeric vector. %1RM used (use 0.5 for 50%, 0.9 for 90%)
adjustment	Numeric vector. Adjustment to be implemented
mfactor	Numeric vector. Default is 1 (i.e., no adjustment). Use mfactor = 2 to generate ballistic adjustment and tables
max_reps_func	Max reps function to be used. Default is max_reps_epley
...	Forwarded to max_reps_func. Usually the parameter value. For example klin = 36 when using max_reps_linear as max_reps_func function

Value

Numeric vector. Predicted number of repetitions to be performed

Functions

- adj_reps_RIR(): Adjust number of repetitions using the Reps In Reserve (RIR) approach
- adj_reps_DI(): Adjust number of repetitions using the Deducted Intensity (DI) approach
- adj_reps_rel_int(): Adjust number of repetitions using the Relative Intensity (RelInt) approach
- adj_reps_perc_MR(): Adjust number of repetitions using the % max reps (%MR) approach

Examples

```
# ------------------------------------------
# Adjustment using Reps In Reserve (RIR)
adj_reps_RIR(0.75)

# Use ballistic adjustment (this implies doing half the reps)
adj_reps_RIR(0.75, mfactor = 2)

# Use 2 reps in reserve
adj_reps_RIR(0.75, adjustment = 2)

# Use Linear model
adj_reps_RIR(0.75, max_reps_func = max_reps_linear, adjustment = 2)

# Use Modifed Epley's equation with a custom parameter values
adj_reps_RIR(
  0.75,
  max_reps_func = max_reps_modified_epley,
  adjustment = 2,
  kmod = 0.06
)
# ------------------------------------------
# Adjustment using Deducted Intensity (DI)
adj_reps_DI(0.75)

# Use ballistic adjustment (this implies doing half the reps)
adj_reps_DI(0.75, mfactor = 2)
```

```
# Use 10% deducted intensity
adj_reps_DI(0.75, adjustment = -0.1)

# Use Linear model
adj_reps_DI(0.75, max_reps_func = max_reps_linear, adjustment = -0.1)

# Use Modifed Epley's equation with a custom parameter values
adj_reps_DI(
  0.75,
  max_reps_func = max_reps_modified_epley,
  adjustment = -0.1,
  kmod = 0.06
)
# ------------------------------------------
# Adjustment using Relative Intensity (RelInt)
adj_reps_rel_int(0.75)

# Use ballistic adjustment (this implies doing half the reps)
adj_reps_rel_int(0.75, mfactor = 2)

# Use 85% relative intensity
adj_reps_rel_int(0.75, adjustment = 0.85)

# Use Linear model
adj_reps_rel_int(0.75, max_reps_func = max_reps_linear, adjustment = 0.85)

# Use Modifed Epley's equation with a custom parameter values
adj_reps_rel_int(
  0.75,
  max_reps_func = max_reps_modified_epley,
  adjustment = 0.85,
  kmod = 0.06
)
# ------------------------------------------
# Adjustment using % max reps (%MR)
adj_reps_perc_MR(0.75)

# Use ballistic adjustment (this implies doing half the reps)
adj_reps_perc_MR(0.75, mfactor = 2)

# Use 85% of max reps
adj_reps_perc_MR(0.75, adjustment = 0.85)

# Use Linear model
adj_reps_perc_MR(0.75, max_reps_func = max_reps_linear, adjustment = 0.85)

# Use Modifed Epley's equation with a custom parameter values
adj_reps_perc_MR(
  0.75,
  max_reps_func = max_reps_modified_epley,
  adjustment = 0.85,
  kmod = 0.06
```

```
)
```

`create_example` *Create Example*

Description

This function create simple example using `progression_table`

Usage

```
create_example(
  progression_table,
  reps = c(3, 5, 10),
  volume = c("intensive", "normal", "extensive"),
  type = c("grinding", "ballistic"),
  ...
)
```

Arguments

`progression_table`	Progression table function
`reps`	Numeric vector. Default is `c(3, 5, 10)`
`volume`	Character vector. Default is `c("intensive", "normal", "extensive")`
`type`	Character vector. Type of max rep table. Options are grinding (Default) and ballistic
`...`	Extra arguments forwarded to `progression_table`

Value

Data frame with the following structure

type Type of the set and rep scheme

reps Number of reps performed

volume Volume type of the set and rep scheme

Step 1 First progression step %1RM

Step 2 Second progression step %1RM

Step 3 Third progression step %1RM

Step 4 Fourth progression step %1RM

Step 2-1 Diff Difference in %1RM between second and first progression step

Step 3-2 Diff Difference in %1RM between third and second progression step

Step 4-3 Diff Difference in %1RM between fourth and third progression step

Examples

```
create_example(progression_RIR)

# Create example using specific reps-max table and k value
create_example(
  progression_RIR,
  max_perc_1RM_func = max_perc_1RM_modified_epley,
  kmod = 0.0388
)
```

`estimate_functions` *Estimate relationship between reps and %1RM (or weight)*

Description

By default, target variable is the reps performed, while the predictors is the `perc_1RM` or `weight`. To reverse this, use the `reverse = TRUE` argument

Usage

```
estimate_k_generic(
  perc_1RM,
  reps,
  eRIR = 0,
  k = 0.0333,
  reverse = FALSE,
  weighted = "none",
  ...
)

estimate_k_generic_1RM(
  weight,
  reps,
  eRIR = 0,
  k = 0.0333,
  reverse = FALSE,
  weighted = "none",
  ...
)

estimate_k(perc_1RM, reps, eRIR = 0, reverse = FALSE, weighted = "none", ...)

estimate_k_1RM(weight, reps, eRIR = 0, reverse = FALSE, weighted = "none", ...)

estimate_kmod(
  perc_1RM,
  reps,
```

```
  eRIR = 0,
  reverse = FALSE,
  weighted = "none",
  ...
)

estimate_kmod_1RM(
  weight,
  reps,
  eRIR = 0,
  reverse = FALSE,
  weighted = "none",
  ...
)

estimate_klin(
  perc_1RM,
  reps,
  eRIR = 0,
  reverse = FALSE,
  weighted = "none",
  ...
)

estimate_klin_1RM(
  weight,
  reps,
  eRIR = 0,
  reverse = FALSE,
  weighted = "none",
  ...
)

get_predicted_1RM_from_k_model(model)
```

Arguments

`perc_1RM`	%1RM
`reps`	Number of repetitions done
`eRIR`	Subjective estimation of reps-in-reserve (eRIR)
`k`	Value for the generic Epley's equation, which is by default equal to 0.0333
`reverse`	Logical, default is `FALSE`. Should reps be used as predictor instead as a target?
`weighted`	What weighting should be used for the non-linear regression? Default is "none". Other options include: "reps" (for 1/reps weighting), "load" (for using weight or %1RM), "eRIR" (for 1/(eRIR+1) weighting), "reps x load", "reps x eRIR", "load x eRIR", and "reps x load x eRIR"
`...`	Forwarded to `nlsLM` function

weight	Weight used
model	Object returned from the estimate_k_1RM function

Value

nlsLM object

Functions

- estimate_k_generic(): Provides the model with generic k parameter
- estimate_k_generic_1RM(): Provides the model with generic k parameter, as well as estimated 1RM. This is a novel estimation function that uses the absolute weights.
- estimate_k(): Estimate the parameter k in the Epley's equation
- estimate_k_1RM(): Estimate the parameter k in the Epley's equation, as well as 1RM. This is a novel estimation function that uses the absolute weights.
- estimate_kmod(): Estimate the parameter kmod in the modified Epley's equation
- estimate_kmod_1RM(): Estimate the parameter kmod in the modified Epley's equation, as well as 1RM. This is a novel estimation function that uses the absolute weights
- estimate_klin(): Estimate the parameter klin using the Linear/Brzycki model
- estimate_klin_1RM(): Estimate the parameter klin in the Linear/Brzycki equation, as well as 1RM. This is a novel estimation function that uses the absolute weights
- get_predicted_1RM_from_k_model(): Estimate the 1RM from estimate_k_1RM function

  The problem with Epley's estimation model (implemented in estimate_k_1RM function) is that it predicts the 1RM when nRM = 0. Thus, the estimated parameter in the model produced by the estimate_k_1RM function is not 1RM, but 0RM. This function calculates the weight at nRM = 1 for both the normal and reverse model. See Examples for code

Examples

```
# ---------------------------------------------------------
# Generic Epley's model
m1 <- estimate_k_generic(
  perc_1RM = c(0.7, 0.8, 0.9),
  reps = c(10, 5, 3)
)

coef(m1)
# ---------------------------------------------------------
# Generic Epley's model that also estimates 1RM
m1 <- estimate_k_generic_1RM(
  weight = c(70, 110, 140),
  reps = c(10, 5, 3)
)

coef(m1)
# ---------------------------------------------------------
# Epley's model
m1 <- estimate_k(
```

```
  perc_1RM = c(0.7, 0.8, 0.9),
  reps = c(10, 5, 3)
)

coef(m1)
# ----------------------------------------------------------
# Epley's model that also estimates 1RM
m1 <- estimate_k_1RM(
  weight = c(70, 110, 140),
  reps = c(10, 5, 3)
)

coef(m1)
# ----------------------------------------------------------
# Modified Epley's model
m1 <- estimate_kmod(
  perc_1RM = c(0.7, 0.8, 0.9),
  reps = c(10, 5, 3)
)

coef(m1)
# ----------------------------------------------------------
# Modified Epley's model that also estimates 1RM
m1 <- estimate_kmod_1RM(
  weight = c(70, 110, 140),
  reps = c(10, 5, 3)
)

coef(m1)
# ----------------------------------------------------------
# Linear/Brzycki model
m1 <- estimate_klin(
  perc_1RM = c(0.7, 0.8, 0.9),
  reps = c(10, 5, 3)
)

coef(m1)
# ----------------------------------------------------------
# Linear/Brzycki model thal also estimates 1RM
m1 <- estimate_klin_1RM(
  weight = c(70, 110, 140),
  reps = c(10, 5, 3)
)

coef(m1)
# ----------------------------------------------------------
# Estimating 1RM from Epley's model
m1 <- estimate_k_1RM(150 * c(0.9, 0.8, 0.7), c(3, 6, 12))
m2 <- estimate_k_1RM(150 * c(0.9, 0.8, 0.7), c(3, 6, 12), reverse = TRUE)

# Estimated 0RM values from both model
c(coef(m1)[[1]], coef(m2)[[1]])
```

```
# But these are not 1RMs!!!
# Using the "reverse" model, where nRM is the predictor (in this case m2)
# makes it easier to predict 1RM
predict(m2, newdata = data.frame(nRM = 1))

# But for the normal model it involve reversing the formula
# To spare you from the math pain, use this
get_predicted_1RM_from_k_model(m1)

# It also works for the "reverse" model
get_predicted_1RM_from_k_model(m2)
```

---

estimate_functions_mixed
Estimate relationship between reps and weight using the non-linear mixed-effects regression

---

Description

These functions provide estimated 1RM and parameter values using the mixed-effect regression. By default, target variable is the reps performed, while the predictor is the `perc_1RM` or `weight`. To reverse this, use the `reverse = TRUE` argument

Usage

```
estimate_k_mixed(athlete, perc_1RM, reps, eRIR = 0, reverse = FALSE, ...)

estimate_k_generic_1RM_mixed(
  athlete,
  weight,
  reps,
  eRIR = 0,
  k = 0.0333,
  reverse = FALSE,
  random = zeroRM ~ 1,
  ...
)

estimate_k_1RM_mixed(
  athlete,
  weight,
  reps,
  eRIR = 0,
  reverse = FALSE,
  random = k + zeroRM ~ 1,
  ...
)
```

```
estimate_kmod_mixed(athlete, perc_1RM, reps, eRIR = 0, reverse = FALSE, ...)

estimate_kmod_1RM_mixed(
  athlete,
  weight,
  reps,
  eRIR = 0,
  reverse = FALSE,
  random = kmod + oneRM ~ 1,
  ...
)

estimate_klin_mixed(athlete, perc_1RM, reps, eRIR = 0, reverse = FALSE, ...)

estimate_klin_1RM_mixed(
  athlete,
  weight,
  reps,
  eRIR = 0,
  reverse = FALSE,
  random = klin + oneRM ~ 1,
  ...
)
```

Arguments

athlete	Athlete identifier
perc_1RM	%1RM
reps	Number of repetitions done
eRIR	Subjective estimation of reps-in-reserve (eRIR)
reverse	Logical, default is FALSE. Should reps be used as predictor instead as a target?
...	Forwarded to nlme function
weight	Weight used
k	Value for the generic Epley's equation, which is by default equal to 0.0333
random	Random parameter forwarded to nlme function. Default is k + zeroRM ~ 1 for, estimate_k_mixed function, or k + oneRM ~ 1 for estimate_kmod_mixed and estimate_klin_mixed functions

Value

nlme object

Functions

- estimate_k_mixed(): Estimate the parameter k in the Epley's equation
- estimate_k_generic_1RM_mixed(): Provides the model with generic k parameter, as well as estimated 1RM. This is a novel estimation function that uses the absolute weights

- estimate_k_1RM_mixed(): Estimate the parameter k in the Epley's equation, as well as 1RM. This is a novel estimation function that uses the absolute weights
- estimate_kmod_mixed(): Estimate the parameter kmod in the Modified Epley's equation
- estimate_kmod_1RM_mixed(): Estimate the parameter kmod in the Modified Epley's equation, as well as 1RM. This is a novel estimation function that uses the absolute weights
- estimate_klin_mixed(): Estimate the parameter klin in the Linear/Brzycki's equation
- estimate_klin_1RM_mixed(): Estimate the parameter klin in the Linear/Brzycki equation, as well as 1RM. This is a novel estimation function that uses the absolute weights

Examples

```
# ---------------------------------------------------------
# Epley's model
m1 <- estimate_k_mixed(
  athlete = RTF_testing$Athlete,
  perc_1RM = RTF_testing$`Real %1RM`,
  reps = RTF_testing$nRM
)

coef(m1)
# ---------------------------------------------------------
# Generic Epley's model that also estimates 1RM
m1 <- estimate_k_generic_1RM_mixed(
  athlete = RTF_testing$Athlete,
  weight = RTF_testing$`Real Weight`,
  reps = RTF_testing$nRM
)

coef(m1)
# ---------------------------------------------------------
# Epley's model that also estimates 1RM
m1 <- estimate_k_1RM_mixed(
  athlete = RTF_testing$Athlete,
  weight = RTF_testing$`Real Weight`,
  reps = RTF_testing$nRM
)

coef(m1)
# ---------------------------------------------------------
# Modifed Epley's model
m1 <- estimate_kmod_mixed(
  athlete = RTF_testing$Athlete,
  perc_1RM = RTF_testing$`Real %1RM`,
  reps = RTF_testing$nRM
)

coef(m1)
# ---------------------------------------------------------
# Modified Epley's model that also estimates 1RM
m1 <- estimate_kmod_1RM_mixed(
  athlete = RTF_testing$Athlete,
```

```
  weight = RTF_testing$`Real Weight`,
  reps = RTF_testing$nRM
)

coef(m1)
# -----------------------------------------------------------
# Linear/Brzycki model
m1 <- estimate_klin_mixed(
  athlete = RTF_testing$Athlete,
  perc_1RM = RTF_testing$`Real %1RM`,
  reps = RTF_testing$nRM
)

coef(m1)
# -----------------------------------------------------------
# Linear/Brzycki model that also estimates 1RM
m1 <- estimate_klin_1RM_mixed(
  athlete = RTF_testing$Athlete,
  weight = RTF_testing$`Real Weight`,
  reps = RTF_testing$nRM
)

coef(m1)
```

---

estimate_functions_quantile

Estimate relationship between reps and weight using the non-linear quantile regression

---

Description

These functions provide estimate 1RM and parameter values using the quantile regression. By default, target variable is the reps performed, while the predictors is the `perc_1RM` or `weight`. To reverse this, use the `reverse = TRUE` argument

Usage

```
estimate_k_quantile(
  perc_1RM,
  reps,
  eRIR = 0,
  tau = 0.5,
  reverse = FALSE,
  control = quantreg::nlrq.control(maxiter = 10^4, InitialStepSize = 0),
  ...
)

estimate_k_generic_1RM_quantile(
  weight,
```

```
    reps,
    eRIR = 0,
    k = 0.0333,
    tau = 0.5,
    reverse = FALSE,
    control = quantreg::nlrq.control(maxiter = 10^4, InitialStepSize = 0),
    ...
  )

  estimate_k_1RM_quantile(
    weight,
    reps,
    eRIR = 0,
    tau = 0.5,
    reverse = FALSE,
    control = quantreg::nlrq.control(maxiter = 10^4, InitialStepSize = 0),
    ...
  )

  estimate_kmod_quantile(
    perc_1RM,
    reps,
    eRIR = 0,
    tau = 0.5,
    reverse = FALSE,
    control = quantreg::nlrq.control(maxiter = 10^4, InitialStepSize = 0),
    ...
  )

  estimate_kmod_1RM_quantile(
    weight,
    reps,
    eRIR = 0,
    tau = 0.5,
    reverse = FALSE,
    control = quantreg::nlrq.control(maxiter = 10^4, InitialStepSize = 0),
    ...
  )

  estimate_klin_quantile(
    perc_1RM,
    reps,
    eRIR = 0,
    tau = 0.5,
    reverse = FALSE,
    control = quantreg::nlrq.control(maxiter = 10^4, InitialStepSize = 0),
    ...
  )
```

```
estimate_klin_1RM_quantile(
  weight,
  reps,
  eRIR = 0,
  tau = 0.5,
  reverse = FALSE,
  control = quantreg::nlrq.control(maxiter = 10^4, InitialStepSize = 0),
  ...
)
```

Arguments

perc_1RM	%1RM
reps	Number of repetitions done
eRIR	Subjective estimation of reps-in-reserve (eRIR)
tau	Vector of quantiles to be estimated. Default is 0.5
reverse	Logical, default is FALSE. Should reps be used as predictor instead as a target?
control	Control object for the nlrq function. Default is: quantreg::nlrq.control(maxiter = 10^4, InitialStepSize = 0)
...	Forwarded to nlrq function
weight	Weight used
k	Value for the generic Epley's equation, which is by default equal to 0.0333

Value

nlrq object

Functions

- estimate_k_quantile(): Estimate the parameter k in the Epley's equation
- estimate_k_generic_1RM_quantile(): Provides the model with generic k parameter, as well as estimated 1RM. This is a novel estimation function that uses the absolute weights
- estimate_k_1RM_quantile(): Estimate the parameter k in the Epley's equation, as well as 1RM. This is a novel estimation function that uses the absolute weights
- estimate_kmod_quantile(): Estimate the parameter kmod in the modified Epley's equation
- estimate_kmod_1RM_quantile(): Estimate the parameter kmod in the modified Epley's equation, as well as 1RM. This is a novel estimation function that uses the absolute weights
- estimate_klin_quantile(): Estimate the parameter klin in the Linear/Brzycki equation
- estimate_klin_1RM_quantile(): Estimate the parameter klin in the Linear/Brzycki equation, as well as 1RM. This is a novel estimation function that uses the absolute weights

Examples

```
# ---------------------------------------------------------
# Epley's model
m1 <- estimate_k_quantile(
  perc_1RM = c(0.7, 0.8, 0.9),
  reps = c(10, 5, 3)
)

coef(m1)
# ---------------------------------------------------------
# Epley's model that also estimates 1RM
m1 <- estimate_k_generic_1RM_quantile(
  weight = c(70, 110, 140),
  reps = c(10, 5, 3)
)

coef(m1)
# ---------------------------------------------------------
# Epley's model that also estimates 1RM
m1 <- estimate_k_1RM_quantile(
  weight = c(70, 110, 140),
  reps = c(10, 5, 3)
)

coef(m1)
# ---------------------------------------------------------
# Modified Epley's model
m1 <- estimate_kmod_quantile(
  perc_1RM = c(0.7, 0.8, 0.9),
  reps = c(10, 5, 3)
)

coef(m1)
# ---------------------------------------------------------
# Modified Epley's model that also estimates 1RM
m1 <- estimate_kmod_1RM_quantile(
  weight = c(70, 110, 140),
  reps = c(10, 5, 3)
)

coef(m1)
# ---------------------------------------------------------
# Linear/Brzycki model
m1 <- estimate_klin_quantile(
  perc_1RM = c(0.7, 0.8, 0.9),
  reps = c(10, 5, 3)
)

coef(m1)
# ---------------------------------------------------------
# Linear/Brzycki model thal also estimates 1RM
m1 <- estimate_klin_1RM_quantile(
```

```
  weight = c(70, 110, 140),
  reps = c(10, 5, 3)
)

coef(m1)
```

---

estimate_rolling_1RM *Estimate the rolling profile and 1RM*

Description

Estimate the rolling profile and 1RM

Usage

```
estimate_rolling_1RM(
  weight,
  reps,
  eRIR = 0,
  day_index,
  window = 14,
  estimate_function = estimate_k_1RM,
  ...
)
```

Arguments

`weight`	Weight used
`reps`	Number of repetitions done
`eRIR`	Subjective estimation of reps-in-reserve (eRIR)
`day_index`	Day index used to estimate rolling window
`window`	Width of the rolling window. Default is 14
`estimate_function`	Estimation function to be used. Default is `estimate_k_1RM`
`...`	Forwarded to `estimate_function` function

Value

Data frame with day index and coefficients returned by the `estimate_function` function

Examples

```
estimate_rolling_1RM(
  weight = strength_training_log$weight,
  reps = strength_training_log$reps,
  eRIR = strength_training_log$eRIR,
  day_index = strength_training_log$day,
  window = 10,
  estimate_function = estimate_k_1RM_quantile,
  tau = 0.9
)
```

---

generate_progression_table

Family of functions to create progression tables

Description

Family of functions to create progression tables

Usage

```
generate_progression_table(
  progression_table,
  type = c("grinding", "ballistic"),
  volume = c("intensive", "normal", "extensive"),
  reps = 1:12,
  step = seq(-3, 0, 1),
  ...
)

progression_DI(
  reps,
  step = 0,
  volume = "normal",
  adjustment = 0,
  type = "grinding",
  mfactor = NULL,
  step_increment = -0.025,
  volume_increment = step_increment,
  ...
)

progression_RIR(
  reps,
  step = 0,
  volume = "normal",
  adjustment = 0,
```

```
  type = "grinding",
  mfactor = NULL,
  step_increment = 1,
  volume_increment = step_increment,
  ...
)

progression_RIR_increment(
  reps,
  step = 0,
  volume = "normal",
  adjustment = 0,
  type = "grinding",
  mfactor = NULL,
  ...
)

progression_perc_MR(
  reps,
  step = 0,
  volume = "normal",
  adjustment = 0,
  type = "grinding",
  mfactor = NULL,
  step_increment = -0.1,
  volume_increment = -0.2,
  ...
)

progression_perc_MR_variable(
  reps,
  step = 0,
  volume = "normal",
  adjustment = 0,
  type = "grinding",
  mfactor = NULL,
  ...
)

progression_perc_drop(
  reps,
  step = 0,
  volume = "normal",
  adjustment = 0,
  type = "grinding",
  mfactor = NULL,
  ...
)
```

```
progression_rel_int(
  reps,
  step = 0,
  volume = "normal",
  adjustment = 0,
  type = "grinding",
  mfactor = NULL,
  step_increment = -0.05,
  volume_increment = -0.075,
  ...
)
```

Arguments

`progression_table`	Progression table function to use
`type`	Character vector. Type of max rep table. Options are grinding (Default) and ballistic.
`volume`	Character vector: 'intensive', 'normal' (Default), or 'extensive'
`reps`	Numeric vector. Number of repetition to be performed
`step`	Numeric vector. Progression step. Default is 0. Use negative numbers (i.e., -1, -2)
`...`	Extra arguments forwarded to `adj_perc_1RM` family of functions Use this to supply different parameter value (i.e., `k = 0.035`), or model function (i.e., `max_perc_1RM_func = max_perc_1RM_linear`)
`adjustment`	Numeric vector. Additional post adjustment applied to sets. Default is none (value depends on the method).
`mfactor`	Numeric vector. Factor to adjust max rep table. Used instead of `type` parameter, unless `NULL`
`step_increment, volume_increment`	Numeric vector. Used to adjust specific progression methods

Value

List with two elements: `adjustment` and `perc_1RM`

Functions

- `generate_progression_table()`: Generates progression tables
- `progression_DI()`: Deducted Intensity progression table. This simplest progression table simply deducts intensity to progress. Adjust this deducted by using the `deduction` parameter (default is equal to -0.025)
- `progression_RIR()`: Constant RIR Increment progression table. This variant have constant RIR increment across reps from phases to phases and RIR difference between extensive, normal, and intensive schemes. Use `step_increment` and `volume_increment` parameters to utilize needed increments

- `progression_RIR_increment()`: RIR Increment progression table (see Strength Training Manual)
- `progression_perc_MR()`: Constant %MR Step progression table. This variant have constant %MR increment across reps from phases to phases and %MR difference between extensive, normal, and intensive schemes. Use `step_increment` and `volume_increment` parameters to utilize needed increments
- `progression_perc_MR_variable()`: Variable %MR Step progression table
- `progression_perc_drop()`: Perc Drop progression table (see Strength Training Manual)
- `progression_rel_int()`: Relative Intensity progression table. Use `step_increment` and `volume_increment` parameters to utilize needed increments

References

Jovanović M. 2020. Strength Training Manual: The Agile Periodization Approach. Independently published.

Jovanović M. 2020. Strength Training Manual: The Agile Periodization Approach. Independently published.

Examples

```
generate_progression_table(progression_RIR)

generate_progression_table(
  progression_RIR,
  type = "grinding",
  volume = "normal",
  step_increment = 2
)

# Create progression table using specific reps-max table and k value
generate_progression_table(
  progression_RIR,
  max_perc_1RM_func = max_perc_1RM_modified_epley,
  kmod = 0.0388
)
# ---------------------------------------------
# Progression Deducted Intensity
progression_DI(10, step = seq(-3, 0, 1))
progression_DI(10, step = seq(-3, 0, 1), volume = "extensive")
progression_DI(5, step = seq(-3, 0, 1), type = "ballistic", step_increment = -0.05)
progression_DI(
  5,
  step = seq(-3, 0, 1),
  type = "ballistic",
  step_increment = -0.05,
  volume_increment = -0.1
)

# Generate progression table
```

```
generate_progression_table(progression_DI, type = "grinding", volume = "normal")

# Use different reps-max model
generate_progression_table(
  progression_DI,
  type = "grinding",
  volume = "normal",
  max_perc_1RM_func = max_perc_1RM_linear,
  klin = 36
)

# ----------------------------------------
# Progression RIR Constant
progression_RIR(10, step = seq(-3, 0, 1))
progression_RIR(10, step = seq(-3, 0, 1), volume = "extensive")
progression_RIR(5, step = seq(-3, 0, 1), type = "ballistic", step_increment = 2)
progression_RIR(
  5,
  step = seq(-3, 0, 1),
  type = "ballistic",
  step_increment = 3
)

# Generate progression table
generate_progression_table(progression_RIR, type = "grinding", volume = "normal")

# Use different reps-max model
generate_progression_table(
  progression_RIR,
  type = "grinding",
  volume = "normal",
  max_perc_1RM_func = max_perc_1RM_linear,
  klin = 36
)

# Plot progression table
plot_progression_table(progression_RIR)
plot_progression_table(progression_RIR, "adjustment")
# ----------------------------------------
# Progression RIR Increment
progression_RIR_increment(10, step = seq(-3, 0, 1))
progression_RIR_increment(10, step = seq(-3, 0, 1), volume = "extensive")
progression_RIR_increment(5, step = seq(-3, 0, 1), type = "ballistic")

# Generate progression table
generate_progression_table(progression_RIR_increment, type = "grinding", volume = "normal")

# Use different reps-max model
generate_progression_table(
  progression_RIR_increment,
  type = "grinding",
  volume = "normal",
  max_perc_1RM_func = max_perc_1RM_linear,
```

```
  klin = 36
)
# -----------------------------------------
# Progression %MR Step Const
progression_perc_MR(10, step = seq(-3, 0, 1))
progression_perc_MR(10, step = seq(-3, 0, 1), volume = "extensive")
progression_perc_MR(5, step = seq(-3, 0, 1), type = "ballistic", step_increment = -0.2)
progression_perc_MR(
  5,
  step = seq(-3, 0, 1),
  type = "ballistic",
  step_increment = -0.15,
  volume_increment = -0.25
)

# Generate progression table
generate_progression_table(progression_perc_MR, type = "grinding", volume = "normal")

# Use different reps-max model
generate_progression_table(
  progression_perc_MR,
  type = "grinding",
  volume = "normal",
  max_perc_1RM_func = max_perc_1RM_linear,
  klin = 36
)

# -----------------------------------------
# Progression %MR Step Variable
progression_perc_MR_variable(10, step = seq(-3, 0, 1))
progression_perc_MR_variable(10, step = seq(-3, 0, 1), volume = "extensive")
progression_perc_MR_variable(5, step = seq(-3, 0, 1), type = "ballistic")
# Generate progression table
generate_progression_table(progression_perc_MR_variable, type = "grinding", volume = "normal")

# Use different reps-max model
generate_progression_table(
  progression_perc_MR_variable,
  type = "grinding",
  volume = "normal",
  max_perc_1RM_func = max_perc_1RM_linear,
  klin = 36
)
# -----------------------------------------
# Progression Perc Drop
progression_perc_drop(10, step = seq(-3, 0, 1))
progression_perc_drop(10, step = seq(-3, 0, 1), volume = "extensive")
progression_perc_drop(5, step = seq(-3, 0, 1), type = "ballistic")

# Generate progression table
generate_progression_table(progression_perc_drop, type = "grinding", volume = "normal")

# Use different reps-max model
```

```
generate_progression_table(
  progression_perc_drop,
  type = "grinding",
  volume = "normal",
  max_perc_1RM_func = max_perc_1RM_linear,
  klin = 36
)
# ------------------------------------------
# Progression Relative Intensity
progression_rel_int(10, step = seq(-3, 0, 1))
progression_rel_int(10, step = seq(-3, 0, 1), volume = "extensive")
progression_rel_int(5, step = seq(-3, 0, 1), type = "ballistic")

# Generate progression table
generate_progression_table(progression_rel_int, type = "grinding", volume = "normal")
generate_progression_table(progression_rel_int, step_increment = -0.1, volume_increment = 0.15)

# Use different reps-max model
generate_progression_table(
  progression_rel_int,
  type = "grinding",
  volume = "normal",
  max_perc_1RM_func = max_perc_1RM_linear,
  klin = 36
)
```

`get_perc_1RM` *Get %1RM*

Description

Function `get_perc_1RM` represent a wrapper function

Usage

```
get_perc_1RM(reps, method = "RIR", model = "epley", ...)
```

Arguments

`reps`	Numeric vector. Number of repetition to be performed
`method`	Character vector. Default is "RIR". Other options are "DI", "RelInt", "%MR"
`model`	Character vector. Default is "epley". Other options are "modified epley", "linear"
`...`	Forwarded to selected `adj_perc_1RM` function

Value

Numeric vector. Predicted %1RM

Examples

```
get_perc_1RM(5)

# # Use ballistic adjustment (this implies doing half the reps)
get_perc_1RM(5, mfactor = 2)

# Use perc MR adjustment method
get_perc_1RM(5, "%MR", adjustment = 0.8)

# Use linear model with use defined klin values
get_perc_1RM(5, "%MR", model = "linear", adjustment = 0.8, klin = 36)
```

---

get_reps *Get Reps*

Description

Function `get_reps` represent a wrapper function. This function is the reverse version of the `get_perc_1RM` function. Use it when you want to estimate number of repetitions to be used when using the known %1RM and level of adjustment

Usage

```
get_reps(perc_1RM, method = "RIR", model = "epley", ...)
```

Arguments

`perc_1RM`	Numeric vector. %1RM used (use 0.5 for 50 perc, 0.9 for 90 perc)
`method`	Character vector. Default is "RIR". Other options are "DI", "RelInt", "%MR"
`model`	Character vector. Default is "epley". Other options are "modified epley", "linear"
`...`	Forwarded to selected `adj_reps` function

Value

Numeric vector Predicted repetitions

Examples

```
get_reps(0.75)

# # Use ballistic adjustment (this implies doing half the reps)
get_reps(0.75, mfactor = 2)

# Use %MR adjustment method
get_reps(0.75, "%MR", adjustment = 0.8)

# Use linear model with use defined klin values
get_reps(0.75, "%MR", model = "linear", adjustment = 0.8, klin = 36)
```

max_perc_1RM *Family of functions to estimate max %1RM*

Description

Family of functions to estimate max %1RM

Usage

```
max_perc_1RM_epley(reps, k = 0.0333)

max_perc_1RM_modified_epley(reps, kmod = 0.0353)

max_perc_1RM_linear(reps, klin = 33)
```

Arguments

reps	Numeric vector. Number of repetition to be performed
k	User defined k parameter in the Epley's equation. Default is 0.0333
kmod	User defined kmod parameter in the Modified Epley's equation. Default is 0.0353
klin	User defined klin parameter in the Linear equation. Default is 33

Value

Numeric vector. Predicted %1RM

Functions

- `max_perc_1RM_epley()`: Estimate max %1RM using the Epley's equation
- `max_perc_1RM_modified_epley()`: Estimate max %1RM using the Modified Epley's equation
- `max_perc_1RM_linear()`: Estimate max %1RM using the Linear (or Brzycki's) equation

Examples

```
# ----------------------------------------
# Epley equation
max_perc_1RM_epley(1:10)
max_perc_1RM_epley(1:10, k = 0.04)
# ----------------------------------------
# Modified Epley equation
max_perc_1RM_modified_epley(1:10)
max_perc_1RM_modified_epley(1:10, kmod = 0.05)
# ----------------------------------------
# Linear/Brzycki equation
max_perc_1RM_linear(1:10)
max_perc_1RM_linear(1:10, klin = 36)
```

max_reps — *Family of functions to estimate max number of repetition (nRM)*

Description

Family of functions to estimate max number of repetition (nRM)

Usage

```
max_reps_epley(perc_1RM, k = 0.0333)

max_reps_modified_epley(perc_1RM, kmod = 0.0353)

max_reps_linear(perc_1RM, klin = 33)
```

Arguments

perc_1RM	Numeric vector. % 1RM used (use 0.5 for 50 %, 0.9 for 90 %)
k	User defined k parameter in the Epley's equation. Default is 0.0333
kmod	User defined kmod parameter in the Modified Epley's equation. Default is 0.0353
klin	User defined klin parameter in the Linear equation. Default is 33

Value

Numeric vector. Predicted maximal number of repetitions (nRM)

Functions

- `max_reps_epley()`: Estimate max number of repetition (nRM) using the Epley's equation
- `max_reps_modified_epley()`: Estimate max number of repetition (nRM) using the Modified Epley's equation
- `max_reps_linear()`: Estimate max number of repetition (nRM) using the Linear/Brzycki's equation

Examples

```
# ------------------------------------------
# Epley equation
max_reps_epley(0.85)
max_reps_epley(c(0.75, 0.85), k = 0.04)
# ------------------------------------------
# Modified Epley equation
max_reps_modified_epley(0.85)
max_reps_modified_epley(c(0.75, 0.85), kmod = 0.05)
# ------------------------------------------
# Linear/Brzycki's equation
max_reps_linear(0.85)
max_reps_linear(c(0.75, 0.85), klin = 36)
```

plot.STMr_release *Plotting of the Release*

Description

Function for creating `ggplot2` plot of the Release `STMr_release` object

Usage

```
## S3 method for class 'STMr_release'
plot(x, font_size = 14, load_1RM_agg_func = max, ...)
```

Arguments

`x`	`STMr_release` object
`font_size`	Numeric. Default is 14
`load_1RM_agg_func`	Function to aggregate step `load_1RM` from multiple sets. Default is `max`
`...`	Forwarded to `geom_bar_text` and `geom_fit_text` functions. Can be used to se the highest labels size, for example, using `size=5`. See documentation for these two packages for more info

Value

`ggplot2` object

Examples

```
scheme1 <- scheme_step(vertical_planning = vertical_constant)
scheme2 <- scheme_step(vertical_planning = vertical_linear)
scheme3 <- scheme_step(vertical_planning = vertical_undulating)

release_df <- release(
  scheme1, scheme2, scheme3,
  additive_1RM_adjustment = 2.5
)

plot(release_df)
```

plot.STMr_scheme — *Plotting of the Set and Reps Scheme*

Description

Functions for creating ggplot2 plot of the Set and Reps Scheme

Usage

```
## S3 method for class 'STMr_scheme'
plot(x, type = "bar", font_size = 14, perc_str = "%", ...)
```

Arguments

x	STMr_scheme object. See examples
type	Type of plot. Options are "bar" (default), "vertical", and "fraction"
font_size	Numeric. Default is 14
perc_str	Percent string. Default is "%". Use "" to have more space on graph
...	Forwarded to geom_bar_text and geom_fit_text functions. Can be used to se the highest labels size, for example, using size=5. See documentation for these two packages for more info

Value

ggplot2 object

Examples

```
scheme <- scheme_wave(
  reps = c(10, 8, 6, 10, 8, 6),
  # Adjusting sets to use lower %1RM (RIR Inc method used, so RIR adjusted)
  adjustment = c(4, 2, 0, 6, 4, 2),
  vertical_planning = vertical_linear,
  vertical_planning_control = list(reps_change = c(0, -2, -4)),
  progression_table = progression_RIR_increment,
  progression_table_control = list(volume = "extensive")
)

plot(scheme)
plot(scheme, type = "vertical")
plot(scheme, type = "fraction")
```

plot_progression_table *Plotting of the Progression Table*

Description

Function for creating ggplot2 plot of the Progression Table

Usage

```
plot_progression_table(
  progression_table,
  plot = "%1RM",
  signif_digits = 3,
  adjustment_multiplier = 1,
  font_size = 14,
  ...
)
```

Arguments

progression_table	Function for creating progression table
plot	Character string. Options include "%1RM" (default) and "adjustment"
signif_digits	Rounding numbers for plotting. Default is 3
adjustment_multiplier	Factor to multiply the adjustment. Useful when converting to percentage. Default is 1
font_size	Numeric. Default is 14
...	Forwarded to the generate_progression_table function

Value

ggplot2 object

Examples

```
plot_progression_table(progression_RIR_increment, "%1RM", reps = 1:5)
plot_progression_table(progression_RIR_increment, "adjustment", reps = 1:5)

# Create progression pot by using specific reps-max table and klin value
plot_progression_table(
  progression_RIR,
  reps = 1:5,
  max_perc_1RM_func = max_perc_1RM_linear,
  klin = 36
)
```

plot_scheme — *Plotting of the Set and Reps Scheme*

Description

Functions for creating ggplot2 plot of the Set and Reps Scheme

Usage

```
plot_scheme(scheme, font_size = 8, perc_str = "%")
```

Arguments

scheme	Data Frame create by one of the package functions. See examples
font_size	Numeric. Default is 8
perc_str	Percent string. Default is "%". Use "" to have more space on graph

Value

ggplot2 object

Examples

```
scheme <- scheme_wave(
  reps = c(10, 8, 6, 10, 8, 6),
  # Adjusting sets to use lower %1RM (RIR Inc method used, so RIR adjusted)
  adjustment = c(4, 2, 0, 6, 4, 2),
  vertical_planning = vertical_linear,
  vertical_planning_control = list(reps_change = c(0, -2, -4)),
  progression_table = progression_RIR_increment,
  progression_table_control = list(volume = "extensive")
)

plot_scheme(scheme)
```

plot_vertical — *Plotting of the Vertical Planning*

Description

Function for creating ggplot2 plot of the Vertical Planning function

Usage

```
plot_vertical(vertical_plan, reps = c(5, 5, 5), font_size = 14, ...)
```

Arguments

`vertical_plan`	Vertical Plan function
`reps`	Numeric vector
`font_size`	Numeric. Default is 14
`...`	Forwarded to `vertical_plan` function

Examples

```
plot_vertical(vertical_block_undulating, reps = c(8, 6, 4))
```

`release` *Create a Release period*

Description

Release combines multiple schemes together with `prescription_1RM`, `additive_1RM_adjustment`, and `multiplicative_1RM_adjustment` parameters to calculate working `weight`, `load_1RM`, and `buffer`

Usage

```
release(
  ...,
  prescription_1RM = 100,
  additive_1RM_adjustment = 2.5,
  multiplicative_1RM_adjustment = 1,
  rounding = 2.5,
  max_perc_1RM_func = max_perc_1RM_epley
)
```

Arguments

`...`	`STMr_scheme` objects create by `scheme_` functions
`prescription_1RM`	Initial prescription planning 1RM to calculate weight Default is 100
`additive_1RM_adjustment`	Additive 1RM adjustment across phases. Default is 2.5
`multiplicative_1RM_adjustment`	multiplicative 1RM adjustment across phases. Default is 1 (i.e., no adjustment)
`rounding`	Rounding for the calculated weight. Default is 2.5
`max_perc_1RM_func`	Max Perc 1RM function to use when calculating `load_1RM`. Default is `max_perc_1RM_epley`

Value

`STMr_relase` data frame

Examples

```
scheme1 <- scheme_step(vertical_planning = vertical_constant)
scheme2 <- scheme_step(vertical_planning = vertical_linear)
scheme3 <- scheme_step(vertical_planning = vertical_undulating)

release_df <- release(
  scheme1, scheme2, scheme3,
  additive_1RM_adjustment = 2.5
)

plot(release_df)
```

RTF_testing *Reps to failure testing of 12 athletes*

Description

A dataset containing reps to failure testing for 12 athletes using 70, 80, and 90% of 1RM

Usage

```
RTF_testing
```

Format

A data frame with 36 rows and 6 variables:

Athlete Name of the athlete; ID

1RM Maximum weight the athlete can lift correctly for a single rep

Target %1RM %1RM we want to use for testing; 70, 80, or 90%

Target Weight Estimated weight to be lifted

Real Weight Weight that is estimated to be lifted, but rounded to closest 2.5

Real %1RM Recalculated %1RM after rounding the weight

nRM Reps-to-failure (RTF), or the number of maximum repetitions (nRM) performed set_and_reps_schemes *Set and Rep Schemes*

Description

Set and Rep Schemes

Usage

```
scheme_generic(
  reps,
  adjustment,
  vertical_planning,
  vertical_planning_control = list(),
  progression_table,
  progression_table_control = list()
)

scheme_wave(
  reps = c(10, 8, 6),
  adjustment = -rev((seq_along(reps) - 1) * 5)/100,
  vertical_planning = vertical_constant,
  vertical_planning_control = list(),
  progression_table = progression_perc_drop,
  progression_table_control = list(volume = "normal")
)

scheme_plateau(
  reps = c(5, 5, 5),
  vertical_planning = vertical_constant,
  vertical_planning_control = list(),
  progression_table = progression_perc_drop,
  progression_table_control = list(volume = "normal")
)

scheme_step(
  reps = c(5, 5, 5),
  adjustment = -rev((seq_along(reps) - 1) * 10)/100,
  vertical_planning = vertical_constant,
  vertical_planning_control = list(),
  progression_table = progression_perc_drop,
  progression_table_control = list(volume = "intensive")
)

scheme_step_reverse(
  reps = c(5, 5, 5),
  adjustment = -((seq_along(reps) - 1) * 10)/100,
```

```
  vertical_planning = vertical_constant,
  vertical_planning_control = list(),
  progression_table = progression_perc_drop,
  progression_table_control = list(volume = "intensive")
)

scheme_wave_descending(
  reps = c(6, 8, 10),
  adjustment = -rev((seq_along(reps) - 1) * 5)/100,
  vertical_planning = vertical_constant,
  vertical_planning_control = list(),
  progression_table = progression_perc_drop,
  progression_table_control = list(volume = "normal")
)

scheme_light_heavy(
  reps = c(10, 5, 10, 5),
  adjustment = c(-0.1, 0)[(seq_along(reps)%%2) + 1],
  vertical_planning = vertical_constant,
  vertical_planning_control = list(),
  progression_table = progression_perc_drop,
  progression_table_control = list(volume = "normal")
)

scheme_pyramid(
  reps = c(12, 10, 8, 10, 12),
  adjustment = 0,
  vertical_planning = vertical_constant,
  vertical_planning_control = list(),
  progression_table = progression_perc_drop,
  progression_table_control = list(volume = "extensive")
)

scheme_pyramid_reverse(
  reps = c(8, 10, 12, 10, 8),
  adjustment = 0,
  vertical_planning = vertical_constant,
  vertical_planning_control = list(),
  progression_table = progression_perc_drop,
  progression_table_control = list(volume = "extensive")
)

scheme_rep_acc(
  reps = c(10, 10, 10),
  adjustment = 0,
  vertical_planning_control = list(step = rep(0, 4)),
  progression_table = progression_perc_drop,
  progression_table_control = list(volume = "normal")
```

```
)

scheme_ladder(
  reps = c(3, 5, 10),
  adjustment = 0,
  vertical_planning = vertical_constant,
  vertical_planning_control = list(),
  progression_table = progression_perc_drop,
  progression_table_control = list(volume = "normal")
)

scheme_manual(
  index = NULL,
  step,
  sets = 1,
  reps,
  adjustment = 0,
  perc_1RM = NULL,
  progression_table = progression_perc_drop,
  progression_table_control = list(volume = "normal")
)

scheme_perc_1RM(reps = c(5, 5, 5), perc_1RM = c(0.4, 0.5, 0.6), n_steps = 4)
```

Arguments

`reps`	Numeric vector indicating reps prescription
`adjustment`	Numeric vector indicating adjustments. Forwarded to `progression_table`.
`vertical_planning`	Vertical planning function. Default is `vertical_constant`
`vertical_planning_control`	Arguments forwarded to the `vertical_planning` function
`progression_table`	Progression table function. Default is `progression_perc_drop`
`progression_table_control`	Arguments forwarded to the `progression_table` function
`index`	Numeric vector. If not provided, index will be create using sequence of `step`
`step`	Numeric vector
`sets`	Numeric vector. Used to replicate reps and adjustments
`perc_1RM`	Numeric vector of user provided 1RM percentage
`n_steps`	How many progression steps to generate? Default is 4

Value

Data frame with the following columns: `reps`, `index`, `step`, `adjustment`, and `perc_1RM`.

Functions

- `scheme_generic()`: Generic set and rep scheme. `scheme_generic` is called in all other set and rep schemes - only the default parameters differ to make easier and quicker schemes writing and groupings
- `scheme_wave()`: Wave set and rep scheme
- `scheme_plateau()`: Plateau set and rep scheme
- `scheme_step()`: Step set and rep scheme
- `scheme_step_reverse()`: Reverse Step set and rep scheme
- `scheme_wave_descending()`: Descending Wave set and rep scheme
- `scheme_light_heavy()`: Light-Heavy set and rep scheme. Please note that the `adjustment` column in the output will be wrong, hence set to `NA`
- `scheme_pyramid()`: Pyramid set and rep scheme
- `scheme_pyramid_reverse()`: Reverse Pyramid set and rep scheme
- `scheme_rep_acc()`: Rep Accumulation set and rep scheme
- `scheme_ladder()`: Ladder set and rep scheme. Please note that the `adjustment` column in the output will be wrong, hence set to `NA`
- `scheme_manual()`: Manual set and rep scheme
- `scheme_perc_1RM()`: Manual %1RM set and rep scheme

Examples

```
scheme_generic(
  reps = c(8, 6, 4, 8, 6, 4),
  # Adjusting using lower %1RM (RIR Increment method used)
  adjustment = c(4, 2, 0, 6, 4, 2),
  vertical_planning = vertical_linear,
  vertical_planning_control = list(reps_change = c(0, -2, -4)),
  progression_table = progression_RIR_increment,
  progression_table_control = list(volume = "extensive")
)

# Wave set and rep schemes
# --------------------------
scheme_wave()

scheme_wave(
  reps = c(8, 6, 4, 8, 6, 4),
  # Second wave with higher intensity
  adjustment = c(-0.25, -0.15, 0.05, -0.2, -0.1, 0),
  vertical_planning = vertical_block,
  progression_table = progression_perc_drop,
  progression_table_control = list(type = "ballistic")
)

# Adjusted second wave
# and using 3 steps progression
scheme_wave(
```

```
  reps = c(8, 6, 4, 8, 6, 4),
  # Adjusting using lower %1RM (progression_perc_drop method used)
  adjustment = c(0, 0, 0, -0.1, -0.1, -0.1),
  vertical_planning = vertical_linear,
  vertical_planning_control = list(reps_change = c(0, -2, -4)),
  progression_table = progression_perc_drop,
  progression_table_control = list(volume = "extensive")
)

# Adjusted using RIR inc
# This time we adjust first wave as well, first two sets easier
scheme <- scheme_wave(
  reps = c(8, 6, 4, 8, 6, 4),
  # Adjusting using lower %1RM (RIR Increment method used)
  adjustment = c(4, 2, 0, 6, 4, 2),
  vertical_planning = vertical_linear,
  vertical_planning_control = list(reps_change = c(0, -2, -4)),
  progression_table = progression_RIR_increment,
  progression_table_control = list(volume = "extensive")
)
plot(scheme)

# Plateau set and rep schemes
# --------------------------
scheme_plateau()

scheme <- scheme_plateau(
  reps = c(3, 3, 3),
  progression_table_control = list(type = "ballistic")
)
plot(scheme)

# Step set and rep schemes
# --------------------------
scheme_step()

scheme <- scheme_step(
  reps = c(2, 2, 2),
  adjustment = c(-0.1, -0.05, 0),
  vertical_planning = vertical_linear_reverse,
  progression_table_control = list(type = "ballistic")
)
plot(scheme)

# Reverse Step set and rep schemes
#- -------------------------
scheme <- scheme_step_reverse()
plot(scheme)

# Descending Wave set and rep schemes
# --------------------------
scheme <- scheme_wave_descending()
plot(scheme)
```

```
# Light-Heavy set and rep schemes
# --------------------------
scheme <- scheme_light_heavy()
plot(scheme)

# Pyramid set and rep schemes
# --------------------------
scheme <- scheme_pyramid()
plot(scheme)

# Reverse Pyramid set and rep schemes
# --------------------------
scheme <- scheme_pyramid_reverse()
plot(scheme)

# Rep Accumulation set and rep schemes
# --------------------------
scheme_rep_acc()

# Generate Wave scheme with rep accumulation vertical progression
# This functions doesn't allow you to use different vertical planning
# options
scheme <- scheme_rep_acc(reps = c(10, 8, 6), adjustment = c(-0.1, -0.05, 0))
plot(scheme)

# Other options is to use `.vertical_rep_accumulation.post()` and
# apply it after
# The default vertical progression is `vertical_const()`
scheme <- scheme_wave(reps = c(10, 8, 6), adjustment = c(-0.1, -0.05, 0))

.vertical_rep_accumulation.post(scheme)

# We can also create "undulating" rep decrements
.vertical_rep_accumulation.post(
  scheme,
  rep_decrement = c(-3, -1, -2, 0)
)

# `scheme_rep_acc` will not allow you to generate `scheme_ladder()`
# and `scheme_scheme_light_heavy()`
# You must use `.vertical_rep_accumulation.post()` to do so
scheme <- scheme_ladder()
scheme <- .vertical_rep_accumulation.post(scheme)
plot(scheme)

# Please note that reps < 1 are removed. If you do not want this,
# use `remove_reps = FALSE` parameter
scheme <- scheme_ladder()
scheme <- .vertical_rep_accumulation.post(scheme, remove_reps = FALSE)
plot(scheme)

# Ladder set and rep schemes
```

```
# --------------------------
scheme <- scheme_ladder()
plot(scheme)

# Manual set and rep schemes
# --------------------------
scheme_df <- data.frame(
  index = 1, # Use this just as an example
  step = c(-3, -2, -1, 0),
  # Sets are just an easy way to repeat reps and adjustment
  sets = c(5, 4, 3, 2),
  reps = c(5, 4, 3, 2),
  adjustment = 0
)

# Step index is estimated to be sequences of steps
# If you want specific indexes, use it as an argument (see next example)
scheme <- scheme_manual(
  step = scheme_df$step,
  sets = scheme_df$sets,
  reps = scheme_df$reps,
  adjustment = scheme_df$adjustment
)

plot(scheme)

# Here we are going to provide our own index
scheme <- scheme_manual(
  index = scheme_df$index,
  step = scheme_df$step,
  sets = scheme_df$sets,
  reps = scheme_df$reps,
  adjustment = scheme_df$adjustment
)

plot(scheme)

# More complicated example
scheme_df <- data.frame(
  step = c(-3, -3, -3, -3, -2, -2, -2, -1, -1, 0),
  sets = 1,
  reps = c(5, 5, 5, 5, 3, 2, 1, 2, 1, 1),
  adjustment = c(0, -0.05, -0.1, -0.15, -0.1, -0.05, 0, -0.1, 0, 0)
)

scheme_df

scheme <- scheme_manual(
  step = scheme_df$step,
  sets = scheme_df$sets,
  reps = scheme_df$reps,
  adjustment = scheme_df$adjustment,
```

```
    # Select another progression table
    progression_table = progression_DI,
    # Extra parameters for the progression table
    progression_table_control = list(
      volume = "extensive",
      type = "ballistic",
      max_perc_1RM_func = max_perc_1RM_linear,
      klin = 36
    )
  )

  plot(scheme)

  # Provide %1RM manually

  scheme_df <- data.frame(
    index = rep(c(1, 2, 3, 4), each = 3),
    reps = rep(c(5, 5, 5), 4),
    perc_1RM = rep(c(0.4, 0.5, 0.6), 4)
  )

  warmup_scheme <- scheme_manual(
    index = scheme_df$index,
    reps = scheme_df$reps,
    perc_1RM = scheme_df$perc_1RM
  )

  plot(warmup_scheme)
  # Manual %1RM set and rep schemes
  # -------------------------
  warmup_scheme <- scheme_perc_1RM(
    reps = c(10, 8, 6),
    perc_1RM = c(0.4, 0.5, 0.6),
    n_steps = 3
  )

  plot(warmup_scheme)
```

---

strength_training_log *Strength Training Log*

Description

A dataset containing strength training log for a single athlete. Strength training program involves doing two strength training sessions, over 12 week (4 phases of 3 weeks each). Session A involves linear wave-loading pattern starting with 2x12/10/8 reps and reaching 2x8/6/4 reps. Session B involves constant wave-loading pattern using 2x3/2/1. This dataset contains `weight` being used, as well as estimated reps-in-reserve (eRIR), which represent subjective rating of the proximity to failure

Usage

```
strength_training_log
```

Format

A data frame with 144 rows and 8 variables:

phase Phase index number. Numeric from 1 to 4

week Week index number (within phase). Numeric from 1 to 3

day Day (total) index number. Numeric from 1 to 3

session Name of the session. Can be "Session A" or "Session B"

set Set index number. Numeric from 1 to 6

weight Weight in kg being used

reps Number of reps being done

eRIR Estimated reps-in-reserve

---

vertical_planning_functions
Vertical Planning Functions

Description

Functions for creating vertical planning (progressions)

Usage

```
vertical_planning(reps, reps_change = NULL, step = NULL)

vertical_constant(reps, n_steps = 4)

vertical_linear(reps, reps_change = c(0, -1, -2, -3))

vertical_linear_reverse(reps, reps_change = c(0, 1, 2, 3))

vertical_block(reps, step = c(-2, -1, 0, -3))

vertical_block_variant(reps, step = c(-2, -1, -3, 0))

vertical_rep_accumulation(
  reps,
  reps_change = c(-3, -2, -1, 0),
  step = c(0, 0, 0, 0)
)

vertical_set_accumulation(
```

```
    reps,
    step = c(-2, -2, -2, -2),
    reps_change = rep(0, length(step)),
    accumulate_set = length(reps),
    set_increment = 1,
    sequence = TRUE
  )

  vertical_set_accumulation_reverse(
    reps,
    step = c(-3, -2, -1, 0),
    reps_change = rep(0, length(step)),
    accumulate_set = length(reps),
    set_increment = 1,
    sequence = TRUE
  )

  vertical_undulating(reps, reps_change = c(0, -2, -1, -3))

  vertical_undulating_reverse(reps, reps_change = c(0, 2, 1, 3))

  vertical_block_undulating(
    reps,
    reps_change = c(0, -2, -1, -3),
    step = c(-2, -1, -3, 0)
  )

  vertical_volume_intensity(reps, reps_change = c(0, 0, -3, -3))

  .vertical_rep_accumulation.post(
    scheme,
    rep_decrement = c(-3, -2, -1, 0),
    remove_reps = TRUE
  )
```

Arguments

`reps`	Numeric vector indicating reps prescription
`reps_change`	Change in `reps` across progression steps
`step`	Numeric vector indicating progression steps (i.e. -3, -2, -1, 0)
`n_steps`	Number of progression steps. Default is 4
`accumulate_set`	Which set (position in `reps`) to accumulate
`set_increment`	How many sets to increase each step? Default is 1
`sequence`	Should the sequence of accumulated sets be repeated, or individual sets?
`scheme`	Scheme generated by `scheme_` functions
`rep_decrement`	Rep decrements across progression step
`remove_reps`	Should < 1 reps be removed?

Value

Data frame with reps, index, and step columns

Functions

- vertical_planning(): Generic Vertical Planning
- vertical_constant(): Constants Vertical Planning
- vertical_linear(): Linear Vertical Planning
- vertical_linear_reverse(): Reverse Linear Vertical Planning
- vertical_block(): Block Vertical Planning
- vertical_block_variant(): Block Variant Vertical Planning
- vertical_rep_accumulation(): Rep Accumulation Vertical Planning
- vertical_set_accumulation(): Set Accumulation Vertical Planning
- vertical_set_accumulation_reverse(): Set Accumulation Reverse Vertical Planning
- vertical_undulating(): Undulating Vertical Planning
- vertical_undulating_reverse(): Undulating Vertical Planning
- vertical_block_undulating(): Block Undulating Vertical Planning
- vertical_volume_intensity(): Volume-Intensity Vertical Planning
- .vertical_rep_accumulation.post(): Rep Accumulation Vertical Planning POST treatment This functions is to be applied AFTER scheme is generated. Other options is to use scheme_rep_acc function, that is flexible enough to generate most options, except for the scheme_ladder and scheme_light_heavy. Please note that the adjustment column in the output will be wrong, hence set to NA

Examples

```
# Generic vertical planning function
# ---------------------------------

# Constant
vertical_planning(reps = c(3, 2, 1), step = c(-3, -2, -1, 0))

# Linear
vertical_planning(reps = c(5, 5, 5, 5, 5), reps_change = c(0, -1, -2))

# Reverse Linear
vertical_planning(reps = c(5, 5, 5, 5, 5), reps_change = c(0, 1, 2))

# Block
vertical_planning(reps = c(5, 5, 5, 5, 5), step = c(-2, -1, 0, -3))

# Block variant
vertical_planning(reps = c(5, 5, 5, 5, 5), step = c(-2, -1, -3, 0))

# Undulating
vertical_planning(reps = c(12, 10, 8), reps_change = c(0, -4, -2, -6))
```

```
# Undulating + Block variant
vertical_planning(
  reps = c(12, 10, 8),
  reps_change = c(0, -4, -2, -6),
  step = c(-2, -1, -3, 0)
)

# Rep accumulation
# If used with `scheme_generic()` (or any other `scheme_`) it will provide wrong set and rep scheme.
# Use `scheme_rep_acc()` instead, or apply `.vertical_rep_accumulation.post()`
# function AFTER generating the scheme
vertical_planning(
  reps = c(10, 8, 6),
  reps_change = c(-3, -2, -1, 0),
  step = c(0, 0, 0, 0)
)


# Constant
# ----------------------------------
vertical_constant(c(5, 5, 5), 4)
vertical_constant(c(3, 2, 1), 2)

plot_vertical(vertical_constant)

# Linear
# ----------------------------------
vertical_linear(c(10, 8, 6), c(0, -2, -4))
vertical_linear(c(5, 5, 5), c(0, -1, -2, -3))

plot_vertical(vertical_linear)

# Reverse Linear
# ----------------------------------
vertical_linear_reverse(c(6, 4, 2), c(0, 1, 2))
vertical_linear_reverse(c(5, 5, 5))

plot_vertical(vertical_linear_reverse)

# Block
# ----------------------------------
vertical_block(c(6, 4, 2))

plot_vertical(vertical_block)

# Block Variant
# ----------------------------------
vertical_block_variant(c(6, 4, 2))

plot_vertical(vertical_block_variant)

# Rep Accumulation
```

```
# ----------------------------------
# If used with `scheme_generic()` (or any other `scheme_`) it will provide wrong set and rep scheme.
# Use `scheme_rep_acc()` instead, or apply `.vertical_rep_accumulation.post()`
# function AFTER generating the scheme
vertical_rep_accumulation(c(10, 8, 6))

plot_vertical(vertical_rep_accumulation)

# Set Accumulation
# ----------------------------------
# Default is accumulation of the last set
vertical_set_accumulation(c(3, 2, 1))

# We can have whole sequence being repeated
vertical_set_accumulation(c(3, 2, 1), accumulate_set = 1:3)

# Or we can have accumulation of the individual sets
vertical_set_accumulation(c(3, 2, 1), accumulate_set = 1:3, sequence = FALSE)

# We can also have two or more sequences
vertical_set_accumulation(c(10, 8, 6, 4, 2, 1), accumulate_set = c(1:2, 5:6))

# And also repeat the individual sets
vertical_set_accumulation(
  c(10, 8, 6, 4, 2, 1),
  accumulate_set = c(1:2, 5:6),
  sequence = FALSE
)
plot_vertical(vertical_set_accumulation)

# Reverse Set Accumulation
# ----------------------------------
# Default is accumulation of the last set
vertical_set_accumulation_reverse(c(3, 2, 1))

# We can have whole sequence being repeated
vertical_set_accumulation_reverse(c(3, 2, 1), accumulate_set = 1:3)

# Or we can have accumulation of the individual sets
vertical_set_accumulation_reverse(c(3, 2, 1), accumulate_set = 1:3, sequence = FALSE)

# We can also have two or more sequences
vertical_set_accumulation_reverse(c(10, 8, 6, 4, 2, 1), accumulate_set = c(1:2, 5:6))

# And also repeat the individual sets
vertical_set_accumulation_reverse(
  c(10, 8, 6, 4, 2, 1),
  accumulate_set = c(1:2, 5:6),
  sequence = FALSE
)

plot_vertical(vertical_set_accumulation_reverse)
```

```
# Undulating
# ----------------------------------
vertical_undulating(c(8, 6, 4))

# Reverse Undulating
# ----------------------------------
vertical_undulating_reverse(c(8, 6, 4))

# Block Undulating
# ----------------------------------
# This is a combination of Block Variant (undulation in the steps) and
# Undulating (undulation in reps)
vertical_block_undulating(c(8, 6, 4))

# Volume-Intensity
# ----------------------------------
vertical_volume_intensity(c(6, 6, 6))

# Rep Accumulation
# --------------------------
scheme_rep_acc()

# Generate Wave scheme with rep accumulation vertical progression
# This functions doesn't allow you to use different vertical planning
# options
scheme <- scheme_rep_acc(reps = c(10, 8, 6), adjustment = c(-0.1, -0.05, 0))
plot(scheme)

# Other options is to use `.vertical_rep_accumulation.post()` and
# apply it after
# The default vertical progression is `vertical_const()`
scheme <- scheme_wave(reps = c(10, 8, 6), adjustment = c(-0.1, -0.05, 0))

.vertical_rep_accumulation.post(scheme)

# We can also create "undulating" rep decrements
.vertical_rep_accumulation.post(
  scheme,
  rep_decrement = c(-3, -1, -2, 0)
)

# `scheme_rep_acc` will not allow you to generate `scheme_ladder()`
# and `scheme_scheme_light_heavy()`
# You must use `.vertical_rep_accumulation.post()` to do so
scheme <- scheme_ladder()
scheme <- .vertical_rep_accumulation.post(scheme)
plot(scheme)

# Please note that reps < 1 are removed. If you do not want this,
# use `remove_reps = FALSE` parameter
scheme <- scheme_ladder()
scheme <- .vertical_rep_accumulation.post(scheme, remove_reps = FALSE)
plot(scheme)
```

Index