Package 'STMr'

March 16, 2022

Title Strength Training Manual R-Language Functions

Version 0.1.3

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 7.1.2

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

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

Imports dplyr, ggplot2, ggstance, magrittr, 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 2022-03-16 13:10:09 UTC

R topics documented:

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

Arguments

`progression_table`	Progression table function. Default is `progression_RIR_increment`
`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(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)
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 nls function
weight	Weight used
model	Object returned from the estimate_k_1RM function

Value

`nls` object

Functions

- `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

```
# ------------------------------------------------------------
# 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_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

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_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)
# ------------------------------------------------------------
# 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_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

Value

nlrq object

Functions

- estimate_k_quantile: Estimate the parameter k in the Epley's equation
- 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_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)
```

`generate_progression_table` *Family of functions to create progression tables*

Description

Family of functions to create progression tables

Usage

```
generate_progression_table(
  progression_table = progression_RIR_increment,
  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. Default is progression_RIR_increment
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
)

# Plot progression table
plot_progression_table(progression_DI)
plot_progression_table(progression_DI, "adjustment")
# ----------------------------------------
# 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
)

# Plot progression table
plot_progression_table(progression_RIR_increment)
plot_progression_table(progression_RIR_increment, "adjustment")
# ----------------------------------------
# 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
)

# Plot progression table
plot_progression_table(progression_perc_MR)
plot_progression_table(progression_perc_MR, "adjustment")
# ----------------------------------------
# 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
)

# Plot progression table
plot_progression_table(progression_perc_MR_variable)
plot_progression_table(progression_perc_MR_variable, "adjustment")
# ----------------------------------------
# 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
)

# Plot progression table
plot_progression_table(progression_perc_drop)
plot_progression_table(progression_perc_drop, "adjustment")
# ----------------------------------------
# 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
)

# Plot progression table
plot_progression_table(progression_rel_int)
plot_progression_table(progression_rel_int, "adjustment")
```

`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_progression_table *Plotting of the Progression Table*

Description

Functions for creating ggplot2 plot of the Progression Table

Usage

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

Arguments

progression_table	Function for creating progression table. Default is progression_RIR_increment
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
...	Forwarder to the generate_progression_table function

Value

ggplot2 object

Examples

```
plot_progression_table(progression_RIR_increment, "%1RM")
plot_progression_table(progression_RIR_increment, "adjustment")

# Create progression pot by using specific reps-max table and klin value
plot_progression_table(
  progression_RIR,
  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,
  label_size = 3,
  signif_digits = 3,
  adjustment_multiplier = 1
)
```

Arguments

scheme	Data Frame create by one of the package functions. See examples
label_size	Numeric. Default is 3
signif_digits	Rounding numbers for plotting. Default is 3
adjustment_multiplier	Factor to multiply the adjustment. Useful when converting to percentage. Default is 1

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)
```

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 = c(5, 5, 5),
  adjustment = c(0, 0, 0),
  vertical_planning = vertical_linear,
  vertical_planning_control = list(),
  progression_table = progression_RIR_increment,
  progression_table_control = list()
)

scheme_wave(
  reps = c(10, 8, 6, 10, 8, 6),
  adjustment = c(4, 2, 0, 6, 4, 2),
```

```
  vertical_planning = vertical_linear,
  vertical_planning_control = list(),
  progression_table = progression_RIR_increment,
  progression_table_control = list(volume = "extensive")
)

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

scheme_step(
  reps = c(5, 5, 5, 5),
  adjustment = c(-0.3, -0.2, -0.1, 0),
  vertical_planning = vertical_constant,
  vertical_planning_control = list(),
  progression_table = progression_perc_drop,
  progression_table_control = list(volume = "normal")
)

scheme_step_reverse(
  reps = c(10, 10, 10, 10),
  adjustment = c(0, 3, 6, 9),
  vertical_planning = vertical_constant,
  vertical_planning_control = list(),
  progression_table = progression_RIR_increment,
  progression_table_control = list(volume = "normal")
)

scheme_wave_descending(
  reps = c(6, 8, 10, 6, 8, 10),
  adjustment = c(4, 2, 0, 6, 4, 2),
  vertical_planning = vertical_linear,
  vertical_planning_control = list(),
  progression_table = progression_RIR_increment,
  progression_table_control = list(volume = "extensive")
)

scheme_light_heavy(
  reps = c(6, 3, 6, 3, 6, 3),
  adjustment = c(0, -0.2, 0, -0.2, 0, -0.2),
  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, 8, 10, 12),
  adjustment = 0,
  vertical_planning = vertical_linear,
  vertical_planning_control = list(reps_change = c(0, -2, -4, -6)),
  progression_table = progression_RIR_increment,
  progression_table_control = list(volume = "extensive")
)

scheme_pyramid_reverse(
  reps = c(8, 10, 12, 12, 10, 8),
  adjustment = 0,
  vertical_planning = vertical_linear,
  vertical_planning_control = list(reps_change = c(0, -2, -4, -6)),
  progression_table = progression_RIR_increment,
  progression_table_control = list(volume = "extensive")
)

scheme_rep_acc(
  reps = c(7, 7, 7),
  adjustment = 0,
  vertical_planning_control = list(step = rep(-3, 4)),
  progression_table = progression_RIR_increment,
  progression_table_control = list(volume = "extensive")
)
```

Arguments

`reps`	Numeric vector indicating reps prescription
`adjustment`	Numeric vector indicating adjustments. Forwarded to `progression_table`. If the `progression_table` is `progression_RIR_increment`, `adjustment` will be done using RIR. On the other hand, if `progression_perc_drop` is used, `adjustment` will be done using 1RM percentage
`vertical_planning`	Vertical planning function. Default is `vertical_linear`
`vertical_planning_control`	Arguments forwarded to the `vertical_planning` function
`progression_table`	Progression table function. Default is `progression_RIR_increment`
`progression_table_control`	Arguments forwarded to the `progression_table` function

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
- `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

Examples

```
scheme_generic()

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

scheme_wave(
  reps = c(8, 6, 4, 8, 6, 4),
  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_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")
)

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

scheme_plateau(
  reps = c(3, 3, 3),
  progression_table_control = list(type = "ballistic")
)

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

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")
)

# Reverse Step set and rep schemes
--------------------------
  scheme_step_reverse()

# Descending Wave set and rep schemes
--------------------------
  scheme_wave_descending()

# Light-Heavy set and rep schemes
--------------------------
  scheme_light_heavy()

# Pyramid set and rep schemes
--------------------------
  scheme_pyramid()

# Reverse Pyramid set and rep schemes
--------------------------
  scheme_pyramid_reverse()

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

`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 7 variables:

phase Phase index number. Numeric from 1 to 4

week Week 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),
  accumulate_rep = length(reps),
  set_increment = 1
)

vertical_set_accumulation_reverse(
  reps,
  step = c(-3, -2, -1, 0),
  accumulate_rep = length(reps),
  set_increment = 1
)

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

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

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

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_rep	Which rep (position in reps) to accumulate
set_increment	How many sets to increase each step? Default is 1

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_volume_intensity`: Volume-Intensity Vertical Planning

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
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)

# Linear
```

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

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

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

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

# Rep Accumulation
vertical_rep_accumulation(c(19, 8, 6))

# Set Accumulation
vertical_set_accumulation(c(5, 5, 5))
vertical_set_accumulation(c(3, 2, 1), step = c(-1, -1, -1))
vertical_set_accumulation(
  c(3, 2, 1),
  step = c(-1, -1, -1),
  accumulate_rep = 1
)
vertical_set_accumulation(
  c(3, 2, 1),
  step = c(-1, -1, -1),
  accumulate_rep = 2,
  set_increment = 2
)

# Set Accumulation Reverse
vertical_set_accumulation_reverse(c(5, 5, 5))
vertical_set_accumulation_reverse(c(3, 2, 1), step = c(-1, -1, -1))
vertical_set_accumulation_reverse(
  c(3, 2, 1),
  step = c(-1, -1, -1),
  accumulate_rep = 1
)
vertical_set_accumulation_reverse(
  c(3, 2, 1),
  step = c(-4, -2, 0),
  accumulate_rep = 2,
  set_increment = 2
)

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

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

# Volume-Intensity
```

```
vertical_volume_intensity(c(6, 6, 6))
```

Index