Title

stata.com

***gsdesign usermethod* —** Add your own methods to the gsdesign command

Description

The `gsdesign` *usermethod* command allows you to add your own methods to create a group sequential design (GSD) and produce tables and graphs of the stopping boundaries.

Syntax

`gsdesign` *usermethod* ... [`,` *designopts boundopts*]

where *usermethod* is the name of the method you would like to add to the `gsdesign` command, *designopts* are options controlling the sample-size calculation, and *boundopts* are options controlling the calculation of the stopping boundaries.

When naming your `gsdesign` methods, you should follow the same convention as for naming the programs you add to Stata—do not pick "nice" names that may later be used by Stata's built-in methods. The length of *usermethod* may not exceed 16 characters.

designopts	Description
Main	
usermethodopts	method-specific options for user-defined method
* `alpha(#)`	overall significance level for all tests; default is `alpha(0.05)`
* `power(#)`	overall power for all tests; default is `power(0.8)`
`beta(#)`	overall probability of type II error for all tests; default is `beta(0.2)`
`onesided`	request a one-sided test; default is two-sided
* `nfractional`	report fractional sample size

*User-written sample-size evaluators must allow options `alpha()`, `power()`, and `nfractional`.

`collect` is allowed; see [U] **11.1.10 Prefix commands**.

boundopts	Description
Bounds	
`efficacy(`*boundary*`)`	boundary for efficacy stopping; if neither `efficacy()` nor `futility()` is specified, the default is `efficacy(obfleming)`
`futility(`*boundary*`[ , binding ])`	boundary for futility stopping; use `binding` to request binding futility bounds (default is nonbinding)
`nlooks(#[ , equal ])`	total number of analyses (`nlooks()` − 1 interim analyses and one final analysis); use `equal` to enforce equal information increments; if neither `nlooks()` nor `information()` is specified, the default is `nlooks(2)`
`information(`*numlist*`)`	sequence of information levels for analyses; default is evenly spaced
`nopvalues`	suppress p-values
Graph	
`graphbounds[ (`*graphopts*`) ]`	graph boundaries
`matlistopts(`*general_options*`)`	control the display of boundaries and sample size; seldom used
optimopts	optimization options for boundary calculations; seldom used

boundary	Description
`obfleming`	classical O'Brien–Fleming bound
`pocock`	classical Pocock bound
`wtsiatis(#)`	classical Wang–Tsiatis bound with specified parameter value
`errpocock`	error-spending Pocock-style bound
`errobfleming`	error-spending O'Brien–Fleming-style bound
`kdemets(#)`	error-spending Kim–DeMets bound with specified parameter value
`hsdecani(#)`	error-spending Hwang–Shih–de Cani bound with specified parameter value

graphopts	Description
`xdimsampsize`	label the x axis with the sample size collected (default)
`xdiminformation`	label the x axis with the information fraction; use information levels if `information()` specified
`xdimlooks`	label the x axis with the number of each look
`noshade`	do not shade the rejection, acceptance, and continuation regions
`rejectopts(`*area_options*`)`	change the appearance of the rejection region
`acceptopts(`*area_options*`)`	change the appearance of the acceptance region
`continueopts(`*area_options*`)`	change the appearance of the continuation region
`efficacyopts(`*connected_options*`)`	change the appearance of the efficacy bound
`futilityopts(`*connected_options*`)`	change the appearance of the futility bound
`nolooklines`	do not draw vertical reference lines at each look
`looklinesopts(`*added_line_suboptions*`)`	change the appearance of the reference lines marking each look
`nofixed`	do not label critical values from a fixed study design
`fixedopts(`*marker_options*`)`	change the appearance of the fixed-study critical values
twoway_options	any options other than `by()` documented in [G-3] ***twoway_options***

optimopts	Description
`intpointsscale(#)`	scaling factor for number of quadrature points; default is `intpointsscale(20)`
`initinfo(`*initinfo_spec*`)`	initial value(s) for maximum information
`initscale(#)`	initial value for scaling factor C of classical bounds
`infotolerance(#)`	tolerance for bisection search for maximum information of error-spending bounds with futility stopping; default is `infotol(1e-6)`
`marquardt`	use the Marquardt stepping algorithm in nonconcave regions; default is to use a mixture of steepest descent and Newton
`technique(`*algorithm_spec*`)`	maximization technique
`iterate(#)`	perform maximum of # iterations; default is `iterate(300)`
`[no]log`	display an iteration log; default is `nolog`
`trace`	display current parameter vector in iteration log
`gradient`	display current gradient vector in iteration log
`showstep`	report steps within an iteration in iteration log
`hessian`	display current negative Hessian matrix in iteration log
`showtolerance`	report the calculated result that is compared with the effective convergence criterion
`tolerance(#)`	tolerance for the parameter being optimized; default is `tolerance(1e-12)`
`ftolerance(#)`	tolerance for the objective function; default is `ftolerance(1e-10)`
`nrtolerance(#)`	tolerance for the scaled gradient; default is `nrtolerance(1e-16)`
`nonrtolerance`	ignore the `nrtolerance()` option

Options

Main

`alpha(#)` sets the overall significance level, which is the familywise type I error rate for all analyses (interim and final). `alpha()` must be in $(0, 0.5)$. The default is `alpha(0.05)`.

`power(#)` sets the overall power for all analyses. `power()` must be in $(0.5, 1)$. The default is `power(0.8)`. If `beta()` is specified, `power()` is set to be $1 -$ `beta()`. Only one of `power()` or `beta()` may be specified.

`beta(#)` sets the overall probability of a type II error. `beta()` must be in $(0, 0.5)$. The default is `beta(0.2)`. If `power()` is specified, `beta()` is set to be $1 -$ `power()`. Only one of `beta()` or `power()` may be specified.

`onesided` requests a study design for a one-sided test. The direction of the test is inferred from the effect size.

`nfractional` specifies that fractional sample sizes be reported.

`nratio(#)` specifies the sample-size ratio of the experimental group relative to the control group, $N2/N1$. The default is `nratio(1)`, meaning equal allocation between the two groups.

Bounds

`efficacy(`*boundary*`)` specifies the boundary for efficacy stopping. If neither `efficacy()` nor `futility()` is specified, the default is `efficacy(obfleming)`.

`futility(`*boundary*`[ , binding ])` specifies the boundary for futility stopping.

`binding` specifies binding futility bounds. With binding futility bounds, if the result of an interim analysis crosses the futility boundary and lies in the acceptance region, the trial must end or risk overrunning the specified type I error. With nonbinding futility bounds, the trial does not need to stop if the result of an interim analysis crosses the futility boundary; the familywise type I error rate is controlled even if the trial continues. By default, futility bounds are nonbinding.

`nlooks(# [ , equal ])` specifies the total number of analyses to be performed (`nlooks()` $- 1$ interim analyses and one final analysis). If neither `nlooks()` nor `information()` is specified, the default is `nlooks(2)`.

`equal` indicates that equal information increments be enforced, which is to say that the same number of new observations will be collected at each look. The default behavior is to start by dividing information evenly among looks, then proceed by rounding up to a whole number of observations at each look. This can cause slight differences in the information collected at each look.

`information(`*numlist*`)` specifies a sequence of information levels for interim and final analyses. This must be a sequence of increasing positive numbers, but the scale is unimportant because the information sequence will be automatically rescaled to ensure the maximum information is reached at the final look. By default, analyses are evenly spaced.

`nopvalues` suppresses the p-values from being reported in the table of boundaries for each look.

Graph

`graphbounds` and `graphbounds(`*graphopts*`)` produce graphical output showing the stopping boundaries.

graphopts are the following:

`xdimsampsize` labels the x axis with the sample size collected (the default).

`xdiminformation` labels the x axis with the information fraction unless `information()` is specified, in which case information levels will be used.

`xdimlooks` labels the x axis with the number of each look.

`noshade` suppresses shading of the rejection, acceptance, and continuation regions of the graph.

`rejectopts(`*area_options*`)` affects the rendition of the rejection region. See [G-3] ***area_options***.

`acceptopts(`*area_options*`)` affects the rendition of the acceptance region. See [G-3] ***area_options***.

`continueopts(`*area_options*`)` affects the rendition of the continuation region. See [G-3] ***area_options***.

`efficacyopts(`*connected_options*`)` affects the rendition of the efficacy bound. See [G-3] ***cline_options*** and [G-3] ***marker_options***.

`futilityopts(`*connected_options*`)` affects the rendition of the futility bound. See [G-3] ***cline_options*** and [G-3] ***marker_options***.

`nolooklines` suppresses the vertical reference lines drawn at each look.

`looklinesopts(`*added_line_suboptions*`)` affects the rendition of reference lines marking each look. See *suboptions* in [G-3] ***added_line_options***.

`nofixed` suppresses the fixed-study critical values in the plot.

`fixedopts(`*marker_options*`)` affects the rendition of the fixed-study critical values. See [G-3] ***marker_options***.

twoway_options are any of the options documented in [G-3] ***twoway_options***, excluding `by()`. These include options for titling the graph (see [G-3] ***title_options***) and for saving the graph to disk (see [G-3] ***saving_option***).

`matlistopts(`*general_options*`)` affects the display of the matrix of boundaries and sample sizes. *general_options* are `title()`, `tindent()`, `rowtitle()`, `showcoleq()`, `coleqonly`, `colorcoleq()`, `aligncolnames()`, and `linesize()`; see *general_options* in [P] **matlist**. This option is seldom used.

optimopts control the iterative algorithm used to calculate stopping boundaries:

`intpointsscale(#)` specifies the scaling factor for the number of quadrature points used during the numerical evaluation of stopping probabilities at each look. The default is `intpointsscale(20)`. See *Methods and formulas* in [ADAPT] **gsbounds**.

`initinfo(`*initinfo_spec*`)` specifies either one or two initial values to be used in the iterative calculation of the maximum information.

The syntax `initinfo(#)` is applicable when using classical group sequential boundaries (Pocock bounds, O'Brien–Fleming bounds, and Wang–Tsiatis bounds), as well as with efficacy-only stopping when using error-spending boundaries (error-spending Pocock-style efficacy bounds, error-spending O'Brien–Fleming-style efficacy bounds, Kim–DeMets efficacy bounds, and Hwang–Shih–de Cani efficacy bounds). The default is to use the information from a fixed study design; see *Methods and formulas* in [ADAPT] **gsbounds**.

The syntax `initinfo(# #)` is applicable when using error-spending group sequential boundaries with futility stopping (error-spending Pocock-style bounds, error-spending O'Brien–Fleming-style bounds, Kim–DeMets bounds, and Hwang–Shih–de Cani bounds). With this syntax, the first and second numbers specify the lower and upper starting values, respectively, for the bisection algorithm estimating the maximum information. The default is to use the information from a fixed study design for the lower initial value and the information corresponding to a Bonferroni correction for the upper initial value; see *Methods and formulas* in [ADAPT] **gsbounds**. To specify just the lower starting value, use `initinfo(# .)`, and to specify just the upper starting value, use `initinfo(. #)`.

`initscale(#)` specifies the initial value to be used during the iterative calculation of scaling factor C for classical group sequential boundaries (Pocock bounds, O'Brien–Fleming bounds, and Wang–Tsiatis bounds). The default is to use the z-value corresponding to the specified value of `alpha()`. See *Methods and formulas* in [ADAPT] **gsbounds**.

`infotolerance(#)` specifies the tolerance for the bisection algorithm used in the iterative calculation of the maximum information of error-spending group sequential boundaries with futility stopping (error-spending Pocock-style bounds, error-spending O'Brien–Fleming-style bounds, Kim–DeMets bounds, and Hwang–Shih–de Cani bounds). The default is `infotolerance(1e-6)`. See *Methods and formulas* in [ADAPT] **gsbounds**.

`marquardt` specifies that the optimizer should use the modified Marquardt algorithm when, at an iteration step, it finds that H is singular. The default is to use a mixture of steepest descent and Newton, which is equivalent to the `difficult` option in [R] **ml**.

`technique(`*algorithm_spec*`)` specifies how the objective function is to be maximized. The following algorithms are allowed. For details, see Gould, Pitblado, and Poi (2010).

`technique(bfgs)` specifies the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm.

`technique(nr)` specifies Stata's modified Newton–Raphson (NR) algorithm.

`technique(dfp)` specifies the Davidon–Fletcher–Powell (DFP) algorithm.

The default is `technique(bfgs)` when using classical group sequential boundaries (Pocock bounds, O'Brien–Fleming bounds, and Wang–Tsiatis bounds) and also for the second optimization step used to estimate the maximum information with efficacy-only stopping when using error-spending boundaries (error-spending Pocock-style efficacy bounds, error-spending O'Brien–Fleming-style efficacy bounds, Kim–DeMets efficacy bounds, and Hwang–Shih–de Cani efficacy bounds). The default is `technique(nr)` for the sequential optimization steps used to estimate critical values for error-spending boundaries. You can also switch between two algorithms by specifying the technique name followed by the number of iterations. For example, specifying `technique(nr 10 bfgs 20)` requests 10 iterations with the NR algorithm followed by 20 iterations with the BFGS algorithm, and then back to NR for 10 iterations, and so on. The process continues until convergence or until the maximum number of iterations is reached.

`iterate(#)` specifies the maximum number of iterations. If convergence is not declared by the time the number of iterations equals `iterate()`, an error message is issued. The default value of `iterate(#)` is the number set using `set maxiter`, which is 300 by default.

[`no`]`log` requests an iteration log showing the progress of the optimization. The default is `nolog`.

`trace` adds to the iteration log a display of the current parameter vector.

`gradient` adds to the iteration log a display of the current gradient vector.

`showstep` adds to the iteration log a report on the steps within an iteration. This option was added so that developers at StataCorp could view the stepping when they were improving the `ml` optimizer code. At this point, it mainly provides entertainment.

`hessian` adds to the iteration log a display of the current negative Hessian matrix.

`showtolerance` adds to the iteration log the calculated value that is compared with the effective convergence criterion at the end of each iteration. Until convergence is achieved, the smallest calculated value is reported. `shownrtolerance` is a synonym of `showtolerance`.

Below, we describe the three convergence tolerances. Convergence is declared when the `nrtolerance()` criterion is met and either the `tolerance()` or the `ftolerance()` criterion is also met.

`tolerance(#)` specifies the tolerance for the parameter vector. When the relative change in the parameter vector from one iteration to the next is less than or equal to `tolerance()`, the `tolerance()` convergence criterion is satisfied. The default is `tolerance(1e-12)`.

`ftolerance(#)` specifies the tolerance for the objective function. When the relative change in the objective function from one iteration to the next is less than or equal to `ftolerance()`, the `ftolerance()` convergence is satisfied. The default is `ftolerance(1e-10)`.

`nrtolerance(#)` specifies the tolerance for the scaled gradient. Convergence is declared when $\mathbf{g}\mathbf{H}^{-1}\mathbf{g}' <$ `nrtolerance()`. The default is `nrtolerance(1e-16)`.

`nonrtolerance` specifies that the default `nrtolerance()` criterion be turned off.

boundary

`obfleming` specifies a classical O'Brien–Fleming design for efficacy or futility bounds (O'Brien and Fleming 1979). O'Brien–Fleming efficacy bounds are characterized by being extremely conservative at early looks. The O'Brien–Fleming design is a member of the Wang–Tsiatis family and is equivalent to specifying a *boundary* of `wtsiatis(0)`.

`pocock` specifies a classical Pocock design for efficacy or futility bounds (Pocock 1977). Pocock efficacy bounds are characterized by using the same critical value at all looks. The Pocock design is a member of the Wang–Tsiatis family and is equivalent to specifying a *boundary* of `wtsiatis(0.5)`.

`wtsiatis(#)` specifies a classical Wang–Tsiatis design for efficacy or futility bounds (Wang and Tsiatis 1987). The shape of Wang–Tsiatis bounds is determined by parameter $\Delta \in [-10, 0.7]$, where smaller values of Δ yield bounds that are more conservative at early looks.

`errpocock` specifies an error-spending Pocock-style design for efficacy or futility bounds (Lan and DeMets 1983). The critical values from error-spending Pocock-style bounds are very similar to those of classic Pocock bounds, but they are obtained using an error-spending function.

`errobfleming` specifies an error-spending O'Brien–Fleming-style design for efficacy or futility bounds (Lan and DeMets 1983). The critical values from error-spending O'Brien–Fleming-style bounds are very similar to those of classic O'Brien–Fleming bounds, but they are obtained using an error-spending function.

`kdemets(#)` specifies an error-spending Kim–DeMets design for efficacy or futility bounds (Kim and DeMets 1987). The shape of Kim–DeMets bounds is determined by power parameter $\rho \in (0, 10]$, where larger values of ρ yield bounds that are more conservative at early looks.

`hsdecani(#)` specifies an error-spending Hwang–Shih–de Cani design for efficacy or futility bounds (Hwang, Shih, and de Cani 1990). The shape of Hwang–Shih–de Cani bounds is determined by parameter $\gamma \in [-30, 3]$, where smaller values of γ yield bounds that are more conservative at early looks.

For a design with both efficacy and futility stopping boundaries, if you specify a classical boundary (that is, in the Wang–Tsiatis family) for one, then you must specify a classical boundary for the other. So, you could not specify a boundary in the Wang–Tsiatis family for one boundary and an error-spending boundary for the other. When specifying efficacy and futility boundaries from the same family, the efficacy parameter does not need to be the same as the futility parameter.

Boundaries that are conservative at early looks, such as the O'Brien–Fleming bound, offer little chance of early stopping unless the true effect size is quite large (in the case of efficacy bounds) or quite small (in the case of futility bounds). A trial employing a conservative bound is more likely to continue to the final look, yielding an expected sample size that is not dramatically smaller than the sample size required by an equivalent fixed-sample trial. However, the maximum sample size (that is, the sample size at the final look) of a trial with a conservative bound is generally not much greater than the sample size required by an equivalent fixed trial. Another direct result of specifying conservative bounds is that the critical value at the final look tends to be close to the critical value employed by an equivalent fixed design. In contrast, anticonservative boundaries such as the Pocock bound offer a much better shot at early stopping (often yielding a small expected sample size) at the cost of a larger maximum sample size and final critical values that are considerably larger than the critical value of an equivalent fixed design.

Remarks and examples

stata.com

Remarks are presented under the following headings:

This entry describes the use of the `gsdesign` command with a user-defined sample-size evaluator. See [ADAPT] **GSD intro** for a general introduction to GSDs for clinical trials; see [ADAPT] **gsbounds** for information about group sequential bounds; and see [ADAPT] **gsdesign** for information about designing group sequential clinical trials with the `gsdesign` command. Also see [PSS-2] **Intro (power)** for a general introduction to power and sample-size analysis, and see [PSS-2] **power usermethod** for additional details about how to write your own sample-size evaluator.

Steps for adding a new method to the gsdesign command

`gsdesign` works by combining stopping boundaries calculated by `gsbounds` with the fixed-design sample size calculated by `power`. If the sample-size calculation you want does not exist as a built-in `power` method, you can write your own sample-size evaluator and use it with `gsdesign`.

Adding your own methods to `gsdesign` is easy. Suppose you want to add your own method, *usermethod*, to `gsdesign`:

1. Create the evaluator, an r-class program called `power_cmd_`*usermethod* that computes the sample size that would be required for a fixed study design. Save the program as ado-file `power_cmd_`*usermethod*`.ado`.

    A. Be sure your program accepts the `nfractional` option. This is necessary because `gsdesign` uses fractional sample sizes when calculating the sample size required at each look.

B. Store the resulting sample size following `power`'s simple naming conventions. Store the total sample size in `r(N)`. For two-sample methods, additionally store control-group and experimental-group sample sizes in `r(N1)` and `r(N2)`, respectively. For time-to-event methods, additionally store the number of events in `r(E)` and store local macro `r(endpoint)` as "`survival`".

C. If your method allows one-sided tests, store local macro `r(direction)` as "`upper`" for an upper one-sided test and as "`lower`" for a lower one-sided test.

2. Optionally, create an initializer or a parser, s-class programs called, respectively, `power_cmd_`*usermethod*`_init` (defined by ado-file `power_cmd_`*usermethod*`_init.ado`) and `power_cmd_`*usermethod*`_parse` (defined by ado-file `power_cmd_`*usermethod*`_parse.ado`). This step is not necessary but can be used to customize the titles and parameters displayed by `gsdesign`. See *Initializer and parser* for more details.

3. Place all of your programs where Stata can find them.

You are done. You can now use `gsdesign` *usermethod* like any other `gsdesign` method.

All user-defined methods for `gsdesign` are, by construct, also user-defined methods for the `power` command. This means that your evaluator can be used to calculate the sample size for a fixed study design by running command `power` *usermethod*. This ability can be exploited, as we do in our second example. However, it bears mentioning that the `power` command allows user-defined evaluators to calculate power and effect size in addition to sample size, but `gsdesign` only supports sample-size calculations.

A quick example

Before we discuss the technical details in the following sections, let's try an example to show how easy this all is. We will write a program to compute sample size for a fixed-study one-sample z test given standardized difference, significance level, and power. For simplicity, we assume a two-sided test.

We will call our new method `myztest` and save it as `power_cmd_myztest.ado`.

```
program power_cmd_myztest, rclass
        version 18.0

        /* parse syntax */
        syntax, STDDiff(real)      /// standardized difference (effect size)
                [ Alpha(real 0.05) /// significance level
                  Power(real 0.8)  /// power
                  NFRACtional      /// report fractional sample size
                ]

        /* calculate sample size for a fixed study */
        tempname N
        scalar `N' = ((invnormal(`power') + invnormal(1 - `alpha' / 2)) / `stddiff')^2
        if ("`nfractional'" == "") {
                scalar `N' = ceil(`N')
        }

        /* return stored results */
        return scalar N       = `N'
        return scalar alpha   = `alpha'
        return scalar power   = `power'
        return scalar stddiff = `stddiff'
end
```

Our program consists of three sections: the `syntax` command for parsing options, the sample-size computation, and returning the stored results. The three sections work as follows:

Parse: The `power_cmd_myztest` program accepts three of `gsdesign`'s *designopts*: `alpha()` for significance level, `power()` for power, and `nfractional` to compute fractional sample size. It also has its own option, `stddiff()`, to specify a standardized difference.

Compute: After parsing options, sample size is computed and stored in temporary scalar `'N'`.

Return: Finally, the resulting sample size and other results are returned as scalars. Following `power`'s convention for naming commonly returned results, the computed sample size is stored in `r(N)`, the significance level in `r(alpha)`, and the power in `r(power)`. The program additionally stores the standardized difference in `r(stddiff)`.

We save our program as `power_cmd_myztest.ado` and place the program where Stata can find it. Now we can use `myztest` within `gsdesign` as we would any other existing method of `gsdesign`.

To design a group sequential trial using `myztest` with a standardized difference of 0.7 and default specifications of O'Brien–Fleming efficacy bounds with two evenly spaced looks, power of 0.8, and two-sided significance level of 0.05, we run

```
. gsdesign myztest, stddiff(0.7)
Group sequential design for myztest
Two-sided test
Efficacy: O'Brien-Fleming
Study parameters:
      alpha = 0.0500  (two-sided)
      power = 0.8000
Expected sample size:
         H0 =  16.96
         Ha =  15.06
Info. ratio = 1.0078
    N fixed =     17
      N max =     17
Fixed-study crit. values = ±1.9600
Critical values, p-values, and sample sizes for a group sequential design
```

Look	Info. frac.	Efficacy Lower	Efficacy Upper	p-value	Sample size N
1	0.50	-2.7965	2.7965	0.0052	9
2	1.00	-1.9774	1.9774	0.0480	17

```
Notes: Critical values are for z statistics; otherwise,
       use p-value boundaries.
       Requested information fraction not attained.
```

We can use any type of boundary allowed by `gsdesign`, and we can even display the bounds on a graph. For a four-look design with Wang–Tsiatis efficacy bounds with efficacy parameter $\Delta_e = 0.25$ and O'Brien–Fleming nonbinding futility bounds, we run

```
. gsdesign myztest, stddiff(0.7) efficacy(wtsiatis(0.25))  futility(obfleming)
> nlooks(4) graphbounds

Group sequential design for myztest
Two-sided test

Efficacy: Wang-Tsiatis, Delta = 0.2500
Futility: O'Brien-Fleming, nonbinding

Study parameters:
      alpha = 0.0500  (two-sided)
      power = 0.8000

Expected sample size:
         H0 =  12.56
         Ha =  14.09

Info. ratio = 1.2141
    N fixed =     17
      N max =     20

Fixed-study crit. values = ±1.9600

Critical values, p-values, and sample sizes for a group sequential design
```

Look	Info. frac.	Efficacy Lower	Efficacy Upper	Efficacy p-value	Futility Lower	Futility Upper	Futility p-value
1	0.25	-2.9887	2.9887	0.0028	.	.	.
2	0.50	-2.5132	2.5132	0.0120	-0.8059	0.8059	0.4203
3	0.75	-2.2709	2.2709	0.0232	-1.5492	1.5492	0.1213
4	1.00	-2.1133	2.1133	0.0346	-2.1133	2.1133	0.0346

```
Note: Critical values are for z statistics; otherwise, use p-value
      boundaries.
```

Look	Sample size N
1	5
2	10
3	15
4	20

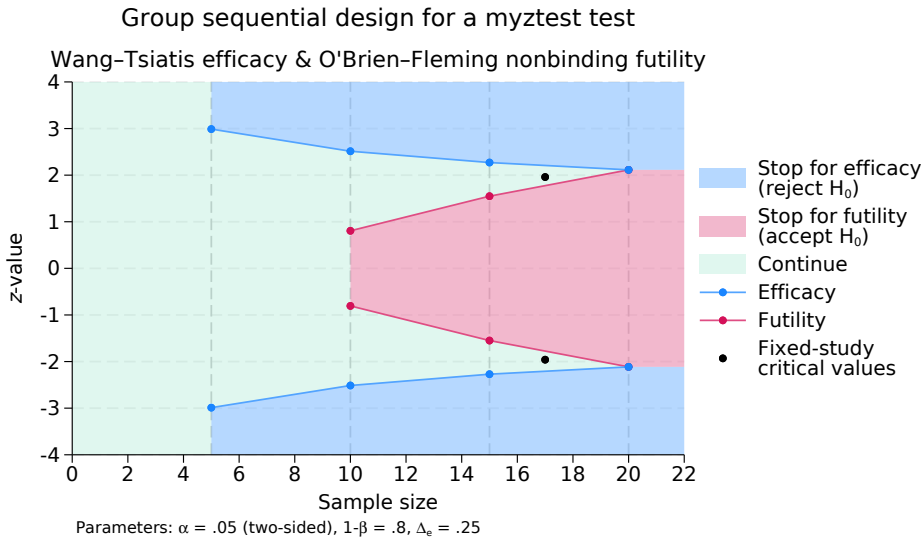

Figure 1. User-written one-sample z test with efficacy and futility bounds

The above is just a simple example. Your program can be as complicated as you would like; you can even use simulations to compute your results. You can also customize your output and graphs with an initializer or parser.

Convention for naming options and storing results

You can specify any method-specific options you want, but for the `gsdesign` command to automatically recognize its common design options, you must ensure that you follow `gsdesign`'s naming convention for *designopts* in your program. For example, `gsdesign` specifies the significance level in the `alpha()` option with minimum abbreviation of `a()`. You need to ensure that you use the same option name with the same abbreviation in your evaluator to specify the significance level. The same applies to all the *designopts* described in the *Syntax* section.

To be compatible with `gsdesign`, you must ensure that your sample-size evaluator stores the total sample size in scalar `r(N)`. For two-sample methods, you must additionally store the control-group sample size in scalar `r(N1)` and the experimental-group sample size in scalar `r(N2)`.

For time-to-event methods, your evaluator must store local macro `r(endpoint)` as "`survival`" and store the number of events in scalar `r(E)`. If your method allows for censoring, store the survival probability of the control group in scalar `r(s1)` and the survival probability of the experimental group in scalar `r(s2)`, store the overall probability of experiencing a failure event in scalar `r(Pr_E)`, and store the probability of withdrawal in scalar `r(Pr_w)`.

If your method allows one-sided tests, it should store local macro `r(direction)` as "`upper`" when an upper one-sided test is specified and as "`lower`" when a lower one-sided test is specified.

If you want to display additional parameters in the `gsdesign` output, you must store them as scalars and let `gsdesign` know to display them through the use of an initializer or parser. However, the full functionality of the `gsdesign` command is available without the use of an initializer or parser.

Example: A log-rank test for substantial superiority

Many aspects of the COVID-19 pandemic were unprecedented, including the speed with which vaccines were developed. Unlike the yearslong development process of previous vaccines, the first COVID-19 vaccines began phase 3 clinical trials for efficacy less than half a year after COVID-19 was declared a global pandemic. One of these vaccines was produced by the company Sinovac, and Palacios et al. (2020) describe the PROFISCOV phase 3 clinical trial of the Sinovac COVID-19 vaccine among healthcare workers in Brazil.

The primary endpoint, or outcome of interest, was the incidence of symptomatic COVID-19. Rather than merely recording whether study participants caught COVID-19, the researchers monitored how long it took each participant to catch COVID-19, making this a survival study. Participants who had not experienced symptomatic COVID-19 by the end of the study's one-year follow-up period were considered to have been censored. This type of censoring is known as administrative censoring.

The PROFISCOV study measured vaccine efficacy as $1 - \text{HR}$, where HR is the hazard ratio of the experimental to the control participants (Palacios et al. 2020, Study Protocol). The alternative hypothesis of the PROFISCOV study was a vaccine efficacy of 60%, corresponding to a hazard ratio of 0.4. However, the null hypothesis was not a vaccine efficacy of 0% (which would correspond to a hazard ratio of 1 and indicate no treatment effect); instead, the null hypothesis was a vaccine efficacy of 30% (corresponding to a hazard ratio of 0.7). To declare the Sinovac COVID-19 vaccine effective, the planners of the PROFISCOV study required it to beat the control by more than 30%. This type of study is known as a superiority trial or, more specifically, a substantial superiority trial with a superiority margin of 30%.

`gsdesign` does not have a built-in method for calculating sample size for a substantial superiority test of two survivor functions, so we will write our own. We assume a log-rank test will be used to compare the two survivor functions, so we model our command after `power logrank`. We write a sample-size evaluator based on the *Methods and formulas* described in [PSS-2] **power logrank**, but we follow the example of Julious (2010, 264) to modify the formulas to accommodate a superiority margin, provided in the form of a hazard ratio under the null hypothesis.

We will call our new method `superlogrank`. It will compute the number of events and sample size for a fixed-design substantial superiority trial using a log-rank test to compare two survivor functions. Sample-size evaluator `power_cmd_superlogrank` accepts the standard `gsdesign` *designopts* of `alpha()`, `power()`, `nfractional`, and `onesided`, but we decide to make `onesided` a required option because we are only interested in testing a one-sided alternative hypothesis: that the Sinovac COVID-19 vaccine is substantially better than the placebo.

Like `power logrank`, our command `power_cmd_superlogrank` performs sample-size calculations for a test of the hazard ratio using the Freedman method (the default) or for a test of the log hazard-ratio using the Schoenfeld method (with option `schoenfeld`). Most of the remaining syntax for `power_cmd_superlogrank` is akin to a simplified version of the `power logrank` syntax: the survival probability in the control group is provided as an optional argument to the command (specified before the comma), the hazard ratio under the alternative hypothesis is specified with option `hratio()`, the probability of withdrawal is specified with option `wdprob()`, and the ratio of experimental-group sample size to control-group sample size is specified with option `nratio()`.

Command `power_cmd_superlogrank` accepts the additional option `hr0()`, the hazard ratio under the null hypothesis. We set the default to be `hr0(1)`, which corresponds to a superiority margin of 0 (that is, the vaccine efficacy under the null hypothesis is 0%). If `hr0()` is left at its default, the substantial superiority test reduces to a standard log-rank test.

```
program power_cmd_superlogrank, rclass
        version 18.0

        /* parse syntax and check for valid options */
        syntax [anything(name=s1)]  /// P(survival) of control group (optional)
               , ONESIDed           /// one-sided test (required option)
                 [ HRatio(real 0.5) /// hazard ratio under Ha
                   hr0(real 1)      /// hazard ratio under H0
                   WDProb(real 0)   /// P(nonadministrative censoring)
                   NRATio(real 1)   /// ratio of experimental/controls
                   SCHoenfeld       /// use Schoenfeld calculation
                   Alpha(real 0.05) /// significance level
                   Power(real 0.8)  /// power
                   NFRACtional      /// report fractional sample size
                 ]

        /* assume 0% survival if s1 is not specified */
        if (`"`s1'"' != "") {
                confirm number `s1'
                assert (`s1' >= 0) & (`s1' < 1)
        }
        else {
                local s1 = 0
        }
        assert (`hratio'  > 0)
        assert (`hr0'     > 0)
        assert (`nratio'  > 0)
        assert (`wdprob' >= 0) & (`wdprob' < 1)
        assert (`alpha'   > 0) & (`alpha'  < 1)
        assert (`power'   > 0) & (`power'  < 1)

        /* calculate number of failures (events) & fixed-study sample size */
        tempname zalpha zbeta Dratio lhs rhs E s2 prE N N1 N2
        scalar `zalpha' = invnormal(1 - `alpha')
        scalar `zbeta'  = invnormal(`power')
        scalar `Dratio' = `hratio' / `hr0'
        scalar `lhs'    = (`zalpha' + `zbeta')^2 / `nratio'
        if ("`schoenfeld'" != "") {
                /* Schoenfeld calculation */
                scalar `rhs' =  ((`nratio' + 1) / log(`Dratio'))^2
        }
        else {
                /* Freedman calculation */
                scalar `rhs' = ((`nratio' * `Dratio' + 1) / (`Dratio' - 1))^2
        }
        scalar `E'   = `lhs' * `rhs'
        scalar `s2'  = `s1'^`hratio'
        scalar `prE' = 1 - (`s1' + `nratio' * `s2') / (`nratio' + 1)
        scalar `N'   = `E' / (`prE' * (1 - `wdprob'))
        scalar `N1'  = `N' * `nratio' / (`nratio' + 1)
        scalar `N2'  = `N' / (`nratio' + 1)

        if ("`nfractional'" == "") {
                /* round up to a whole number */
                scalar `E'  = ceil(`E')
                scalar `N1' = ceil(`N1')
                scalar `N2' = ceil(`N2')
                scalar `N'  = `N1' + `N2'
        }
```

```
        /* return stored results */
        return scalar E           = `E'
        return scalar N           = `N'
        return scalar N1          = `N1'
        return scalar N2          = `N2'
        return scalar hratio      = `hratio'
        return scalar hr0         = `hr0'
        return scalar nratio      = `nratio'
        return scalar s1          = `s1'
        return scalar s2          = `s2'
        return scalar Pr_E        = `prE'
        return scalar Pr_w        = `wdprob'
        return scalar alpha       = `alpha'
        return scalar power       = `power'
        return scalar nfractional = ("`nfractional'" != "")
        return local  direction   = cond(`Dratio' > 1, "upper", "lower")
        return local  endpoint    = "survival"
end
```

While this program is considerably more complicated than our previous program, `power_cmd_myztest`, it contains the same three basic parts: it starts by parsing the syntax, then it calculates the sample size, and finally it returns the stored results. The three sections work as follows:

Parse: The `power_cmd_superlogrank` program accepts four common `gsdesign` *designopts* (`onesided`, `alpha()`, `power()`, and `nfractional`), as well as several of its own options. To match the syntax of `power logrank`, program `power_cmd_superlogrank` reads the survival probability of the control group as an argument (before the comma) rather than as an option. The syntax is parsed with the `syntax` command and checked for validity.

Compute: The required number of events (failures) is calculated and stored in temporary scalar `` `E' ``, and the control-group sample size, experimental-group sample size, and total sample size are calculated and stored in temporary scalars `` `N1' ``, `` `N2' ``, and `` `N' ``, respectively. Additional temporary scalars hold the probability of survival in the experimental group (`` `s2' ``) and the overall probability of failure (`` `prE' ``).

Return: The design parameters specified to `power_cmd_superlogrank` are returned as scalars, as are indicators that a one-sided test was conducted and that the fractional sample size was calculated. The overall sample size is returned as `r(N)`. By returning the control- and experimental-group sample sizes as `r(N1)` and `r(N2)`, `power_cmd_superlogrank` tells `gsdesign` that method `superlogrank` performs a two-sample test.

Because local macro `r(endpoint)` is returned as `"survival"`, `gsdesign` will recognize `superlogrank` as a survival method and know to look for returned results `r(E)`, `r(Pr_E)`, `r(s1)`, `r(s2)`, and `r(Pr_w)`. Additionally, `gsdesign` will know the direction of the one-sided test because `power_cmd_superlogrank` stores local macro `r(direction)` as either `"upper"` or `"lower"`.

Any user-defined method for `gsdesign` is, by design, also a user-defined method for the `power` command. This enables us to perform a simple sanity check of our new program: if `superlogrank` is used as a `power` method and option `hr0()` is left at its default value of `1`, it should yield the same sample size as `power logrank` with the same options. For this sanity check, we arbitrarily choose a control-group survival probability of 83%, hazard ratio of 0.8, withdrawal probability of 12%, significance level of 2.5% for a one-sided test using the Schoenfeld method, and power of 90%, and we allocate 1.5 times as many participants to the experimental arm as to the control arm. We verify:

```
. power logrank 0.83, hratio(0.8) wdprob(0.12) nratio(1.5) schoenfeld
> onesided alpha(0.025) power(0.9)

Estimated sample sizes for two-sample comparison of survivor functions
Log-rank test, Schoenfeld method
H0: ln(HR) = 0  versus  Ha: ln(HR) < 0

Study parameters:

        alpha =    0.0250
        power =    0.9000
        delta =   -0.2231  (log hazard-ratio)
       hratio =    0.8000
        N2/N1 =    1.5000

Censoring and withdrawal:

           s1 =    0.8300
           s2 =    0.8615
         Pr_E =    0.1511
         Pr_w =    0.1200

Estimated number of events and sample sizes:

            E =       880
            N =     6,614
           N1 =     2,646
           N2 =     3,968
        N2/N1 =    1.4996

. power superlogrank 0.83, hratio(0.8) wdprob(0.12) nratio(1.5) schoenfeld
> onesided alpha(0.025) power(0.9)

Estimated sample sizes
One-sided test

  alpha   power       N
------------------------
   .025      .9   6,614
------------------------
```

The output of `power superlogrank` is stark compared with the detailed output of `power logrank`, but the sample-size calculation is identical. The output of `power superlogrank` can be improved through the addition of an initializer or parser, but the functionality of `gsdesign superlogrank` does not require an initializer or parser.

Returning to the design of the PROFISCOV trial, Palacios et al. (2020) report that the study was designed to have 90% power with a one-sided significance level of 2.5%, and it used error-spending Hwang–Shih–de Cani efficacy and futility bounds with parameter $\gamma_e = \gamma_f = -4$ and a single interim look once 40% of the total number of events had been observed. We assume that all participants in the clinical trial will be followed until they develop symptomatic COVID-19, so we omit the command argument specifying the control-group survival probability. Using a hazard ratio of 0.7 under the null hypothesis and 0.4 under the alternative hypothesis, we calculate the stopping boundaries and sample sizes using `gsdesign superlogrank`.

```
. gsdesign superlogrank, hratio(0.4) hr0(0.7) onesided alpha(0.025) power(0.9)
> efficacy(hsdecani(-4)) futility(hsdecani(-4)) information(0.4 1)

Group sequential design for superlogrank
One-sided test

Efficacy: Error-spending Hwang-Shih-de Cani, gamma = -4.0000
Futility: Error-spending Hwang-Shih-de Cani, nonbinding, gamma = -4.0000

Study parameters:
      alpha = 0.0250  (lower one-sided)
      power = 0.9000

Censoring:
         s1 = 0.0000
         s2 = 0.0000
       Pr_E = 1.0000

Expected number of events:
         H0 = 113.41
         Ha = 126.11

Info. ratio = 1.0142
    E fixed =    142
    N fixed =    142
      N max =    144
     N1 max =     72
     N2 max =     72

Fixed-study crit. value = -1.9600

Critical values, p-values, and sample sizes for a group sequential design
```

	Info.	Efficacy		Futility		Events
Look	frac.	Lower	p-value	Upper	p-value	E
1	0.40	-2.9037	0.0018	0.3739	0.6457	58
2	1.00	-1.9753	0.0241	-1.9753	0.0241	144

```
Note: Critical values are for z statistics; otherwise, use p-value
      boundaries.
```

`gsdesign` begins by displaying the study parameters and, because it knows that `superlogrank` is a survival method, details about censoring.

The next section of the output displays the expected number of events, which is the average number of events if the group sequential trial were to be repeated many times. The following section reports the information ratio, the sample size for a fixed study with an equivalent significance level and power (`N fixed`), the maximum sample size of the GSD (`N max`), and the maximum sample sizes for each group (`N1 max` and `N2 max`). The information ratio is the ratio of the number of failures at the final look of the GSD to the number of failures in a fixed study design.

In this case, the maximum sample size is the same as the maximum number of events because we omitted information about censoring, so `gsdesign superlogrank` assumes that all participants are followed until they contract symptomatic COVID-19.

The table at the end of the output displays the stopping boundaries and sample sizes at each look, where sample size is reported as the number of events observed. Boundary critical values are reported on the z scale and are designed to be compared against the z statistic from a log-rank test for substantial superiority.

Graphing boundaries

It is unrealistic to assume, as we did above, that all participants in the clinical trial will be followed until they develop symptomatic COVID-19. Here we assume that only 1% of participants in the control group develop symptomatic COVID-19 during the follow-up period, and we assume that 10% of all participants withdraw from the study before contracting COVID-19. We leave the rest of the design parameters at their previous values, but we add `gsdesign` option `graphbounds` to display the boundaries visually.

```
. gsdesign superlogrank 0.99, hratio(0.4) hr0(0.7) wdprob(0.1) onesided
> alpha(0.025) power(0.9) efficacy(hsdecani(-4))
> futility(hsdecani(-4)) information(0.4 1) graphbounds

Group sequential design for superlogrank
One-sided test

Efficacy: Error-spending Hwang-Shih-de Cani, gamma = -4.0000
Futility: Error-spending Hwang-Shih-de Cani, nonbinding, gamma = -4.0000

Study parameters:
      alpha = 0.0250  (lower one-sided)
      power = 0.9000

Censoring and withdrawal:
         s1 = 0.9900
         s2 = 0.9960
       Pr_E = 0.0070
       Pr_w = 0.1000

Expected number of events:
         H0 = 113.41
         Ha = 126.11

Info. ratio = 1.0142
    E fixed =    142
    N fixed = 22,404
      N max = 22,722
     N1 max = 11,361
     N2 max = 11,361

Fixed-study crit. value = -1.9600

Critical values, p-values, and sample sizes for a group sequential design
```

Look	Info. frac.	Efficacy Lower	Efficacy p-value	Futility Upper	Futility p-value	Events E
1	0.40	-2.9037	0.0018	0.3739	0.6457	58
2	1.00	-1.9753	0.0241	-1.9753	0.0241	144

```
Note: Critical values are for z statistics; otherwise, use p-value
      boundaries.
```

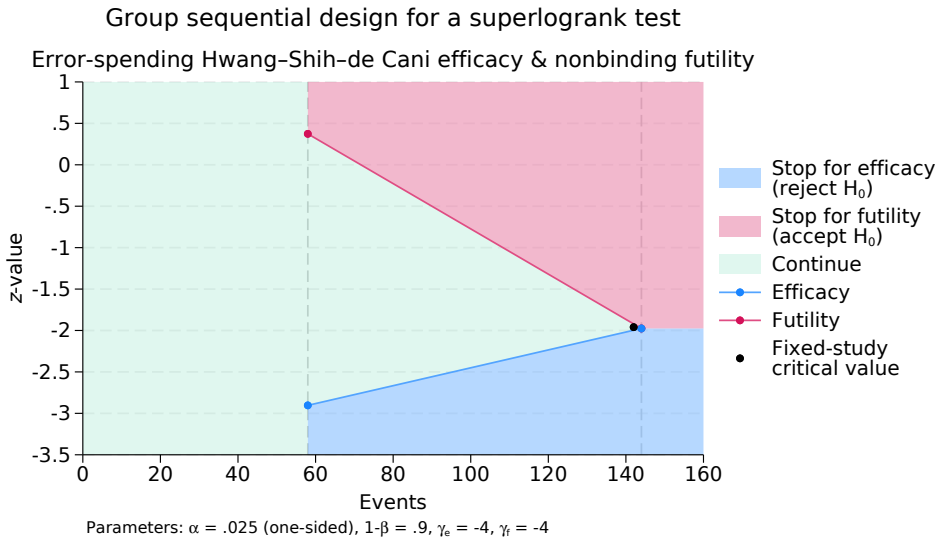

Figure 2. Log-rank test for substantial superiority with efficacy and futility bounds

The required number of events is unchanged from its previous value, but incorporating information about censoring has increased the number of participants we need in order to observe those failures. After taking into account participant withdrawal as well as administrative censoring, we anticipate requiring 22,722 participants to observe 144 failures.

Examining the graph, we see that the entire region from 0 to 58 events is shaded green, the color of the continuation region. This is because the data have not yet been analyzed, so the trial cannot be stopped. The first look will be conducted once 58 participants have contracted symptomatic COVID-19, and a log-rank test for substantial superiority will be performed. If the test statistic, z_1, is ≤ -2.904, we say that z_1 lies in the rejection region (shaded blue on the graph) and we reject H_0, terminating the trial early due to treatment efficacy. If $z_1 > 0.374$, it lies in the acceptance region and we can accept H_0, terminating the trial early for futility. Because the futility bound is nonbinding, if we continue the trial despite z_1 crossing the futility bound, the familywise type I error is still controlled. If $-2.904 < z_1 \leq 0.374$, we say that z_1 lies in the continuation region, and the trial must proceed to the second and final look.

At the final look, there is no continuation region; the futility critical value equals the efficacy critical value of -1.975, so H_0 must be rejected or accepted. While accepting the null hypothesis is taboo in many disciplines, it has a long history in the context of sequential trials (see *Origins of GSD* for a history of GSDs). If test statistic $z_2 \leq -1.975$, we reject H_0; if $z_2 > -1.975$, we accept H_0.

Initializer and parser

The initializer and parser are optional s-class programs named `power_cmd_`*usermethod*`_init` and `power_cmd_`*usermethod*`_parse`, respectively. Initializers and parsers are more important for user-defined `power` commands than for user-defined `gsdesign` commands, but they can still be useful tools to customize the output and graphs produced by `gsdesign`.

The option to provide both an initializer and a parser is provided as a convenience to the user, but in practice only one is ever needed because the `s()` returned values can be set by either an initializer or a parser. In fact, it is generally counterproductive to use both an initializer and a parser because the `s()` returned values are collected by `gsdesign` (or by `power`, in the case of `power` *usermethod*) after first running the parser and then the initializer. This means that if the initializer executes `sreturn clear`, it will clear any `s()` returned values set by the parser.

The difference between the initializer and the parser is that the parser is executed with all the arguments and options specified to `gsdesign` (or to `power`, in the case of `power` *usermethod*), while only options are passed to the initializer, not arguments. This is done to enable the parser to parse the full command specification (instead of the evaluator program), should you so desire. A side effect is that a parser can be more useful than an initializer if your user-defined method accepts arguments as well as options.

Using an initializer and parser

Using our user-defined method `superlogrank` as an example, we define a parser, `power_cmd_superlogrank_parse`, to set `s()` results and customize the output and graph produced by `gsdesign superlogrank`. We choose a parser over an initializer because program `power_cmd_superlogrank` accepts the control-group survival probability as an argument (before the comma), not an option, so it will only be passed to a parser, not an initializer. We write our parser and save it as `power_cmd_superlogrank_parse.ado`.

```
program power_cmd_superlogrank_parse, sclass
        version 18.0

        /* parse relevant syntax */
        syntax [anything(name=s1)] ///
               , [WDProb(string)   ///
                  NRATio(string)   ///
                  SCHoenfeld       ///
                  *                /// asterisk (*) captures all other options
                 ]

        /* identify parameters to display */
        local diparam hratio hr0
        local grparam HR{sub:a} HR{sub:0}
        if (`"`nratio'"' != "") {
                local diparam `diparam' nratio
                local grparam `grparam' N{sub:2}/N{sub:1}
        }
        if (`"`s1'"' != "") {
                local diparam `diparam' s1 s2 Pr_E
                local grparam `grparam' S{sub:1}(T) S{sub:2}(T) p{sub:E}
        }
        if (`"`wdprob'"' != "") {
                local diparam `diparam' Pr_w
                local grparam `grparam' p{sub:w}
        }

        /* return stored results */
        sreturn clear
        local suptest  = "Log-rank test for substantial superiority"
        local testtype = cond("`schoenfeld'" == "", "Freedman", "Schoenfeld")
        sreturn local pss_subtitle = "`suptest', `testtype' method"
        sreturn local pss_title "for two-sample comparison of survivor functions"
        sreturn local pss_colnames `diparam'
        sreturn local pss_colgrsymbols `grparam'
end
```

We rerun the same `gsdesign superlogrank` command specification as before, but this time the parser sets s-class returned values to customize the output and graph.

```
. gsdesign superlogrank 0.99, hratio(0.4) hr0(0.7) wdprob(0.1) onesided
> alpha(0.025) power(0.9) efficacy(hsdecani(-4))
> futility(hsdecani(-4)) information(0.4 1) graphbounds

Group sequential design for two-sample comparison of survivor functions
Log-rank test for substantial superiority, Freedman method

Efficacy: Error-spending Hwang-Shih-de Cani, gamma = -4.0000
Futility: Error-spending Hwang-Shih-de Cani, nonbinding, gamma = -4.0000

Study parameters:
      alpha = 0.0250  (lower one-sided)
      power = 0.9000
     hratio = 0.4000
        hr0 = 0.7000

Censoring and withdrawal:
         s1 = 0.9900
         s2 = 0.9960
       Pr_E = 0.0070
       Pr_w = 0.1000

Expected number of events:
         H0 = 113.41
         Ha = 126.11

Info. ratio = 1.0142
    E fixed =    142
    N fixed = 22,404
      N max = 22,722
     N1 max = 11,361
     N2 max = 11,361

Fixed-study crit. value = -1.9600

Critical values, p-values, and sample sizes for a group sequential design
```

Look	Info. frac.	Efficacy Lower	Efficacy p-value	Futility Upper	Futility p-value	Events E
1	0.40	-2.9037	0.0018	0.3739	0.6457	58
2	1.00	-1.9753	0.0241	-1.9753	0.0241	144

```
Note: Critical values are for z statistics; otherwise, use p-value
      boundaries.
```

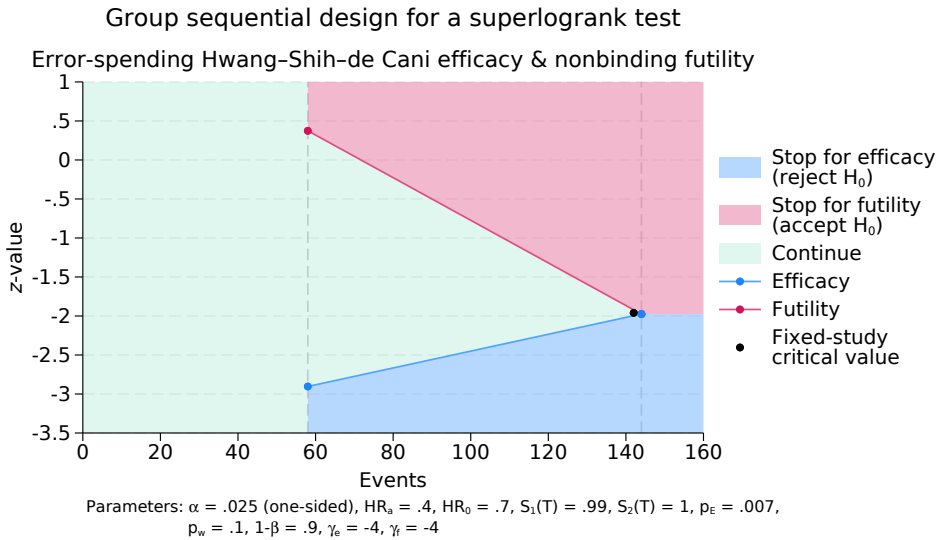

Figure 3. Customized graph of log-rank test for substantial superiority

With the addition of the parser, `gsdesign superlogrank` displays the values of study parameters `hratio` and `hr0`. Also, our additional parameters and their custom symbols now appear in the "Parameters:" note on the graph.

Initializer's s() return settings

The following `s()` results may be set by the initializer or parser. See [PSS-2] **power usermethod** for more details.

Macros

`s(pss_samples)`	`onesample` for a one-sample test or `twosample` for a two-sample test
`s(pss_colnames)`	columns to be added to the default supported columns
`s(pss_allcolnames)`	all supported columns
`s(pss_tabcolnames)`	columns to be added to the default table
`s(pss_alltabcolnames)`	all columns to be displayed in the default table
`s(pss_collabels)`	labels for the specified columns
`s(pss_colformats)`	formats for the specified columns
`s(pss_colwidths)`	widths for the specified columns
`s(pss_colgrlabels)`	labels to be used to label columns on the graph
`s(pss_colgrsymbols)`	symbols to be used to label columns on the graph
`s(pss_delta)`	column name containing the effect-size parameter
`s(pss_target)`	column name containing the target parameter
`s(pss_targetlabel)`	label for the target parameter
`s(pss_argnames)`	column names containing command arguments
`s(pss_title)`	method-specific title
`s(pss_subtitle)`	subtitle
`s(pss_hyp_lhs)`	left-hand-side parameter or value for the hypothesis
`s(pss_hyp_rhs)`	right-hand-side parameter or value for the hypothesis
`s(pss_grhyp_lhs)`	left-hand-side parameter or value for the hypothesis on the graph
`s(pss_grhyp_rhs)`	right-hand-side parameter or value for the hypothesis on the graph

Stored results

Stored results include those stored by the user-defined method and the standard results from `gsdesign`; see *Stored results* in [ADAPT] **gsdesign**.

References

Gould, W. W., J. S. Pitblado, and B. P. Poi. 2010. *Maximum Likelihood Estimation with Stata*. 4th ed. College Station, TX: Stata Press.

Hwang, I. K., W. J. Shih, and J. S. de Cani. 1990. Group sequential designs using a family of type I error probability spending functions. *Statistics in Medicine* 9: 1439–1445. https://doi.org/10.1002/sim.4780091207.

Julious, S. A. 2010. *Sample Sizes for Clinical Trials*. Boca Raton, FL: Chapman and Hall/CRC.

Kim, K., and D. L. DeMets. 1987. Design and analysis of group sequential tests based on the type I error spending rate function. *Biometrika* 74: 149–154. https://doi.org/10.1093/biomet/74.1.149.

Lan, K. K. G., and D. L. DeMets. 1983. Discrete sequential boundaries for clinical trials. *Biometrika* 70: 659–663. https://doi.org/10.1093/biomet/70.3.659.

O'Brien, P. C., and T. R. Fleming. 1979. A multiple testing procedure for clinical trials. *Biometrics* 35: 549–556. https://doi.org/10.2307/2530245.

Palacios, R., E. G. Patiño, R. de Oliveira Piorelli, M. T. R. P. Conde, A. P. Batista, G. Zeng, Q. Xin, E. G. Kallas, J. Flores, C. F. Ockenhouse, and C. Gast. 2020. Double-blind, randomized, placebo-controlled phase III clinical trial to evaluate the efficacy and safety of treating healthcare professionals with the adsorbed COVID-19 (inactivated) vaccine manufactured by Sinovac—PROFISCOV: A structured summary of a study protocol for a randomised controlled trial. *Trials* 21: 853. https://doi.org/10.1186/s13063-020-04775-4.

Pocock, S. J. 1977. Group sequential methods in the design and analysis of clinical trials. *Biometrika* 64: 191–199. https://doi.org/10.1093/biomet/64.2.191.

Wang, S. K., and A. A. Tsiatis. 1987. Approximately optimal one-parameter boundaries for group sequential trials. *Biometrics* 43: 193–199. https://doi.org/10.2307/2531959.

Also see

[ADAPT] **GSD intro** — Introduction to group sequential designs

[ADAPT] **gs** — Introduction to commands for group sequential design

[ADAPT] **gsbounds** — Boundaries for group sequential trials

[ADAPT] **gsdesign** — Study design for group sequential trials

[ADAPT] **Glossary**

[PSS-2] ***power usermethod*** — Add your own methods to the power command