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

Description Syntax Remarks and examples References Also see

# Description

The power command allows you to add your own methods to power and produce tables and graphs of results automatically.

## Syntax

Compute sample size

power usermethod ... [, power (numlist) poweropts useropts]

Compute power

power usermethod ..., nspec [poweropts useropts]

Compute effect size

power usermethod ..., nspec power(numlist) [poweropts useropts]

*usermethod* is the name of the method you would like to add to the power command. When naming your power 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 official methods. The length of *usermethod* may not exceed 16 characters.

useropts are the options supported by your method usermethod.

*nspec* contains n(*numlist*) for a one-sample test or any of the sample-size options of *poweropts* such as n1(*numlist*) and n2(*numlist*) for a two-sample test.

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

## **Remarks and examples**

Adding your own methods to power is easy. Suppose you want to add a method called mymethod to power. Simply

- write an r-class program called power\_cmd\_mymethod that computes power, sample size, or effect size and follows power's convention for naming common options and storing results; and
- 2. place the program where Stata can find it.

You are done. You can now use mymethod within power like any other official power method.

Remarks are presented under the following headings:

A quick example Steps for adding a new method to the power command Convention for naming options and storing results Allowing multiple values in method-specific options Customizing default tables Setting supported columns Modifying the default table columns Modifying the look of the default table Example continued Customizing default graphs Other settings Handling parsing more efficiently More examples: Adding two-sample methods Initializer's s() return settings

#### A quick example

Before we discuss the technical details in the following sections, let's try an example. Let's write a program to compute power for a one-sample z test given sample size, standardized difference, and significance level. For simplicity, we assume a two-sided test. We will call our new method myztest.

We create an ado-file called power\_cmd\_myztest.ado that contains the following Stata program:

```
// evaluator
program power_cmd_myztest, rclass
       version 19.5
                        // (or version 19 if you do not have StataNow)
                       /* parse options */
                               /// sample size
       syntax, n(integer)
                STDDiff(real) /// standardized difference
                Alpha(string) /// significance level
/* compute power */
tempname power
scalar 'power' = normal('stddiff'*sqrt('n') - invnormal(1-'alpha'/2))
/* return results */
                     = 'power'
return scalar power
                     = 'n'
return scalar N
return scalar alpha = 'alpha'
return scalar stddiff = 'stddiff'
end
```

Our ado-program consists of three sections: the syntax command for parsing options, the power computation, and stored or returned results. The three sections work as follows:

The power\_cmd\_myztest program has two of power's common options, n() for sample size and alpha() for significance level, and it has its own option, stddiff(), to specify a standardized difference.

After the options are parsed, the power is computed and stored in a temporary scalar 'power'.

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

We can now use myztest within power as we would any other existing method of power:

```
. power myztest, alpha(0.05) n(10) stddiff(0.25)
Estimated power
Two-sided test
```

alpha	power	Ν
.05	.1211	10

We can compute results for multiple sample sizes by specifying multiple values in the n() option. Note that our power\_cmd\_myztest program accepts only one value at a time in n(). When a numlist is specified in the power command's n() option, power automatically handles that *numlist* for us.

```
. power myztest, alpha(0.05) n(10 20) stddiff(0.25)
Estimated power
Two-sided test
```

alpha	power	N
.05	.1211	10
.05	.1999	20

We can also compute results for multiple sample sizes and significance levels without any additional effort on our part:

```
. power myztest, alpha(0.01 0.05) n(10 20) stddiff(0.25)
Estimated power
Two-sided test
```

alpha	power	N
.01	.03711	10
.01	.07245	20
.05	.1211	10
.05	.1211	10
.05	.1999	20

We can even produce a graph by merely specifying the graph option:

. power myztest, alpha(0.01 0.05) n(10(10)100) stddiff(0.25) graph



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 tables and graphs with a little extra effort.

### Steps for adding a new method to the power command

Suppose you want to add your own method, *usermethod*, to the power command. Here is an outline of the steps to follow:

- Create the evaluator, an r-class program called power\_cmd\_usermethod and defined by the ado-file power\_cmd\_usermethod.ado, that performs power and sample-size computations and follows power's convention for naming options and storing results.
- 2. Optionally, create an initializer, an s-class program called power\_cmd\_usermethod\_init and defined by the ado-file power\_cmd\_usermethod\_init.ado, that specifies information about table columns, options that may allow a numlist, and so on.
- Optionally, create a parser, a program called power\_cmd\_usermethod\_parse and defined by the ado-file power\_cmd\_usermethod\_parse.ado, that checks the syntax of user-specific options, useropts.
- 4. Place all of your programs where Stata can find them.

You can now use your *usermethod* with power:

. power usermethod ...

You may also use programs within power that are not defined by an ado-file (that is, they were defined in a do-file or by hand).

## Convention for naming options and storing results

For the power command to automatically recognize its common options, you must ensure that you follow power's naming convention for these options in your program. For example, power specifies the significance level in the alpha() option with minimum abbreviation of a(). You need to ensure that you use the same option with the same abbreviation in your evaluator to specify the significance level. The same applies to all of power's common options described in [PSS-2] power.

You can specify additional method-specific options, but power will not know about them unless you make it aware of them; see Allowing multiple values in method-specific options for details.

To produce tables and graphs of results, you must ensure that your evaluator follows power's convention for storing results. power's commonly stored results are described in *Stored results* of [PSS-2] **power**. For example, the value for power should be stored in the scalar r(power), the value for a total sample size in the scalar r(N), the value for a significance level in r(alpha), and so on.

You can also store additional method-specific results, but power will not know about them unless you make it aware of them; see *Customizing default tables* for details.

## Allowing multiple values in method-specific options

By default, the power command accepts multiple values only within its common options. If you want to allow multiple values in the method-specific options *useropts*, you need to let power know about them. This is done via the initializer.

To allow the specification of multiple values, or a numlist, in method-specific options, you need to list the names of the options with proper abbreviations in an s-class macro s(pss\_numopts) within the power\_cmd\_usermethod\_init program.

Recall our earlier example in which we added the myztest method, calculating the power of a twosided one-sample z test, to power. We computed powers for multiple values of significance level and sample size. What if we would also like to specify multiple values of standardized differences in the stddiff() option of myztest? If we do this now, we will receive an error message,

```
. power myztest, alpha(0.05) n(10) stddiff(0.25 0.5)
option stddiff() invalid
r(198);
```

because the stddiff() option is not allowed to include *numlist* by the evaluator and is not one of power's common options. To make power recognize this option as one allowing *numlist*, we need to specify this in the initializer. Following the guidelines, we create an ado-file called power\_cmd\_myztest\_init.ado and specify the name of the stddiff() option (with the corresponding abbreviation) in the s-class macro s(pss\_numopts) within the power\_cmd\_myztest\_init program.

```
// initializer
program power_cmd_myztest_init, sclass
    version 19.5 // (or version 19 if you do not have StataNow)
    sreturn clear
    sreturn local pss_numopts "STDDiff"
end
```

We now can specify multiple standardized differences:

```
. power myztest, alpha(0.05) n(10) stddiff(0.25 0.5)
Estimated power
Two-sided test
```

alpha	power	N
.05	.1211	10
.05	.3524	10

### Customizing default tables

The power command with user-defined methods always displays results in a table. By default, it displays columns alpha, power or beta (whichever is specified), and N, which contain the significance level, the power, and the sample size, respectively. See Setting supported columns and Modifying the default table columns for details on how to customize the default table columns.

The default column labels are the column names, and the default formats are %7.4g for alpha and power and %7.0gc for N. These and other settings controlling the look of the default table can be changed as described in *Modifying the look of the default table*.

You can always use the table() option to customize your table. However, if you want to modify how the table looks by default, you need to follow the steps described in the following sections:

Setting supported columns Modifying the default table columns Modifying the look of the default table Example continued

#### Setting supported columns

The power command automatically supports a number of columns, such as alpha, beta, power, N, etc. The supported columns are the columns that can be accessed within power's options table() and graph().

Most of the time, you will have additional columns, *usercolnames*, which you will want power to support. To make power recognize the columns as supported columns, you must list the names of the additional columns, *usercolnames*, in an s-class macro s(pss\_colnames) in the initializer. Columns *usercolnames* will then be added to power's list of supported columns. Columns *usercolnames* will also be displayed in the default table unless s(pss\_tabcolnames) or s(pss\_alltabcolnames) is set.

If you want to reset power's list of supported columns, that is, to specify all the supported columns manually, you should use the s(pss\_allcolnames) macro. The supported columns will then include only the ones you listed in the macro. If you specify s(pss\_allcolnames), you must remember to include power's main columns N, power, and alpha in your list. Otherwise, you may not be able to use some of power's functionality, such as default graphs. If s(pss\_colnames) is specified together with s(pss\_allcolnames), the former will be ignored. The specified supported columns will be automatically displayed in the default table unless s(pss\_alltabcolnames) is set.

The values corresponding to the specified columns must be stored by the evaluator in r() scalars with the same names as the column names. For example, the value for column alpha is stored in r(alpha), the value for column power is stored in r(power), and the value for column N is stored in r(N).

Any column not listed in s(pss\_colnames) or s(pss\_allcolnames) will not be available within the power command. For example, any reference to such a column within power's options table() and graph() will result in an error.

#### Modifying the default table columns

By default, power displays the specified supported columns. If you want to display only a subset of those columns, you can use either s(pss\_tabcolnames) or s(pss\_alltabcolnames) to specify the columns to be displayed. You specify additional columns to be displayed in s(pss\_tabcolnames) and a complete list of the displayed columns in s(pss\_alltabcolnames). If you specify s(pss\_tabcolnames), the displayed columns will include alpha, power, or beta (whichever is specified with the command), N, and the additional columns you specified. If you specify s(pss\_alltabcolnames), only the columns listed in this macro will be displayed. One situation when you may want to do this is if you want to change the order in which the columns are displayed by default. If you specify both macros, s(pss\_tabcolnames) will be ignored. You can specify only the names of supported columns in these macros.

#### Modifying the look of the default table

The default table column labels are the column names. You can change this by specifying your own column labels in the s(pss\_collabels) macro. The labels must be properly quoted if they contain spaces or quotes. The labels must be specified for all columns listed in s(pss\_colnames) or s(pss\_allcolnames).

The default column formats are %7.0gc for sample sizes and %7.4g for all other columns. You can change this by specifying your own column formats in the s(pss\_colformats) macro. The formats must be quoted and specified for all columns listed in s(pss\_colnames) or s(pss\_allcolnames).

The default column widths are the widths of the default formats plus one. You can specify your own column widths in the s(pss\_colwidths) macro. The widths must be specified for all columns listed in s(pss\_colnames) or s(pss\_allcolnames).

#### Example continued

Continuing our myztest example, we want to add a column containing the specified standardized differences to the list of supported columns. The specified standardized difference is stored in r(stddiff) in the myztest evaluator, so the name of our column is stddiff. We specify it in s(pss\_colnames) in our initializer as follows:

```
// initializer
program drop power_cmd_myztest_init
program power_cmd_myztest_init, sclass
        version 19.5 // (or version 19 if you do not have StataNow)
        sreturn clear
        sreturn local pss_numopts "STDDiff"
        sreturn local pss_colnames "stddiff" // <-- new line
end</pre>
```

We rerun our command now and see that the stddiff column is added to the default table:

```
. power myztest, alpha(0.05) n(10) stddiff(0.25)
Estimated power
Two-sided test
alpha power N stddiff
```

10

We can also change the default column label of the stddiff column to "Std. difference". Note that we can do this within power's option table(), but if we want this label to show up automatically in the default table, we should specify it in the initializer. We specify the column label in the s(pss\_collabels) macro.

```
// initializer
program drop power_cmd_myztest_init
program power_cmd_myztest_init, sclass
    version 19.5 // (or version 19 if you do not have StataNow)
    sreturn clear
    sreturn local pss_numopts "STDDiff"
    sreturn local pss_colnames "stddiff"
    sreturn local pss_collabels '""Std. difference""' // <-- new line
end</pre>
```

.25

The column containing standardized differences now has the new label

```
. power myztest, alpha(0.05) n(10) stddiff(0.25)
Estimated power
Two-sided test
```

alpha	power	N Std.	difference
.05	.1211	10	.25

### Customizing default graphs

.05

.1211

By default, power plots the estimated power on the y axis and the specified sample size on the x axis or the estimated sample size on the y axis and the specified power on the x axis. If  $s(pss_target)$  is specified, the estimated sample size is plotted against the specified target parameter. For effect-size computation, the target parameter must be specified in  $s(pss_target)$ , and it is plotted on the x axis against the specified sample size. See [PSS-2] power, graph for details about other default settings.

You can overwrite the default column labels displayed on the graph by specifying the s(pss\_colgrlabels) macro. The specification of the graph labels is the same as the specification of table column labels.

You can also overwrite the default symbols that are used to label the results on the graph by specifying the new symbols in the macro s(pss\_colgrsymbols). The symbols must be specified for all columns listed in s(pss\_colnames) or s(pss\_allcolnames).

## Other settings

When you add your own method to power, the effect-size parameter is not defined. You can define it yourself by specifying the name of the column containing the values of the effect-size parameter in the s(pss\_delta) macro. The effect-size parameter can then be accessed using the column name delta and will be displayed in the default table as delta unless the s(pss\_notabdelta) macro is set to notabdelta.

The target parameter is not set by power for newly added methods. You can set it yourself by specifying the name of the column containing the values of the target parameter in the s(pss\_target) macro. You must set this macro if you want to obtain default graphs for effect-size determination. The target parameter can then be accessed using the column name target.

If the target parameter is set in the s(pss\_target) macro, you can also specify its label in s(pss\_targetlabel). This label will be used in the title for the effect-size determination and as the axis label for the graph column target.

If your method supports command arguments, the arguments specified directly following the method name, you can specify their corresponding column names in the s(pss\_argnames) macro. You can then refer to these arguments as arg1, arg2, and so on, when producing tables or graphs.

power usermethod uses the following generic titles: "Estimated sample size" for sample-size determination, "Estimated power" for power determination, and "Estimated target parameter" for effect-size determination. You can extend these titles to be more specific to your method by adding text in the s(pss\_title) macro. For example, if s(pss\_title) contains "for my test", the resulting titles will be "Estimated sample size for my test", "Estimated power for my test", and "Estimated target parameter for my test". Also see s(pss\_targetlabel) for how to include a label for the target parameter in the title.

power *usermethod* uses the following generic subtitles: "Two-sided test" for a two-sided test or "Onesided test" for a one-sided test when the onesided option is specified. You can change the default subtitle by specifying the s(pss\_subtitle) macro.

Optionally, power *usermethod* can display a hypothesis statement if macros s(pss\_hyp\_lhs) and s(pss\_hyp\_rhs) are specified. s(pss\_hyp\_lhs) must contain the parameter of interest, and s(pss\_hyp\_rhs) will typically contain the null or comparison value. For example, if s(pss\_hyp\_lhs) contains beta1 and s(pss\_hyp\_rhs) contains 0, power *usermethod* will display

Ho: beta1 = 0 versus Ha: beta1 != 0

for a two-sided test and

Ho: beta1 = 0, one-sided alternative

for a one-sided test. The same hypotheses will appear on the graph, unless s(pss\_grhyp\_lhs) and s(pss\_grhyp\_rhs) are specified. These macros are useful if you want to include parameters as symbols on the graph. In our example, we could have defined s(pss\_grhyp\_lhs) as {&beta}{sub:1} and s(pss\_grhyp\_rhs) as 0 to include "beta1" as the corresponding symbol on the graph; see [G-4] *text*.

## Handling parsing more efficiently

The power command checks its common options, but you are responsible for checking your methodspecific options, *useropts*, or their interaction with power's common options. You can certainly do this in your evaluator. However, the checks will then be performed each time your evaluator is called. You can instead perform all of your checks once within the parser.

Your parser may be an s-class command and may set any of the s() results typically specified in the initializer. This may be useful, for example, for building the columns displayed in the default table dynamically, depending on which options were specified. If all desired s() results are set in the parser, you do not need an initializer.

#### More examples: Adding two-sample methods

All the examples so far showed how to add a one-sample method to power. We now demonstrate how to add a two-sample method. (Support for multiple-sample methods is not yet available.)

The steps for adding your own two-sample methods are the same as those for adding one-sample methods, except you may need to set the s(pss\_samples) macro to contain twosample in the initializer. If any of the two-sample options n1(), n2(), and nratio() are specified, power automatically recognizes the method as a two-sample method. If these options are not used and you need the method to be recognized as a two-sample method, you must set s(pss\_samples) to twosample. If you do not want power to respect the two-sample options, you should set s(pss\_samples) to onesample.

For illustration, let's add a method comparing two independent proportions using a large-sample  $\chi^2$  test. (Note that this method is available in the official power twoproportions command.) For simplicity, we will compute the power of a two-sided test. We will call our new method powertwoprop.

We write our evaluator and save it as power\_cmd\_powertwoprop.ado.

```
// evaluator
program power_cmd_powertwoprop, rclass
                         // (or version 19 if you do not have StataNow)
       version 19.5
        //parse command arguments and options
        syntax anything(id="proportions"),
                                                 111
               [ Alpha(real 0.05) /// significance level
                 n(string)
                                       /// total sample size
                 n1(string) n2(string) /// group sample sizes
                 NRATio(real 1)
                                       /// N2/N1
                 Power(string)
                                       111
               ٦
        //parse specification of proportions
        gettoken p1 rest : anything
        gettoken p2 rest : rest
        if ('"'p2'"'=="") {
               di as err "Experimental-group proportion must be specified"
                exit 198
        }
        if ('"'rest'"'!="") {
               di as err "Only two proportions may be specified"
                exit 198
        }
        //sample size must be specified to compute power
        if ('"'n''n1''n2'"'=="") {
                di as err "One of {bf:n()}, {bf:n1()}, or {bf:n2()} " ///
                          "is required to compute power"
                exit 198
        }
```

```
//handle some sample-size specifications
 if ('"'n'"'=="") {
         tempname n
         if ('"'n2'"'=="") {
                 tempname n2
                 scalar 'n2' = ceil('nratio'*'n1')
         }
         else if ('"'n1'"'=="") {
                 tempname n1
                 scalar 'n1' = ceil('n2'/'nratio')
         }
         scalar 'n' = 'n1'+'n2'
         local nratio = 'n2'/'n1'
 }
 else {
         if ('"'n1'"'!="") {
                 tempname n2
                 scalar 'n2' = 'n' - 'n1'
         }
         else if ('"'n2'"'!="") {
                 tempname n1
                 scalar 'n1' = 'n' - 'n2'
         }
         else {
                 tempname n1 n2
                 scalar 'n1' = ceil('n'/(1+'nratio'))
                 scalar 'n2' = 'n'-'n1'
         }
 }
//compute power
tempname diff pbar sigma_D sigma_p crv power
          'diff' = 'p2' - 'p1'
scalar
          'pbar' = ('n1'*'p1'+'n2'*'p2')/'n'
scalar
scalar 'sigma_D' = sqrt('p1'*(1-'p1')/'n1'+'p2'*(1-'p2')/'n2')
scalar 'sigma_p' = sqrt('pbar'*(1-'pbar')*(1/'n1'+1/'n2'))
scalar
           'crv' = invnormal(1-'alpha'/2)*'sigma_p'
         'power' = normal(('diff'-'crv')/'sigma_D') ///
scalar
                   + normal((-'diff'-'crv')/'sigma_D')
//return results
return scalar alpha = 'alpha'
return scalar power = 'power'
                     = 'n'
return scalar N
return scalar N1
                      = 'n1'
                      = 'n2'
return scalar N2
return scalar nratio = 'nratio'
return scalar p1
                  = 'p1'
return scalar p2
                     = 'p2'
```

end

We can now use powertwoprop with the power command. We specify the two proportions following the command name and group sample sizes in the n1() and n2() options.

```
. power powertwoprop 0.1 0.3, n1(40) n2(60)
Estimated power
Two-sided test
alpha power N
.05 .6743 100
```

As with one-sample methods, we can use an initializer (saved in power\_cmd\_powertwoprop\_init.ado) to include additional columns in our default table.

```
// initializer
program power_cmd_powertwoprop_init, sclass
                          // (or version 19 if you do not have StataNow)
        version 19.5
        sreturn clear
        sreturn local pss_colnames "N1 N2 nratio p1 p2"
        sreturn local pss_samples "twosample"
end
. power powertwoprop 0.1 0.3, n1(40) n2(60)
Estimated power
Two-sided test
      alpha
                          Ν
                                  N1
                                          N2 nratio
                                                                   p2
              power
                                                           p1
```

40

60

1.5

.3

. 1

## Initializer's s() return settings

.05

.6743

The following s() results may be set by the initializer or parser:

100

Macros

```
s(pss_samples)
                            onesample for a one-sample test or twosample for a two-sample test
                            columns to be added to the default supported columns
s(pss_colnames)
s(pss_allcolnames)
                            all supported columns
s(pss_tabcolnames)
                            columns to be added to the default table
                            all columns to be displayed in the default table
s(pss_alltabcolnames)
s(pss_collabels)
                            labels for the specified columns
                            formats for the specified columns
s(pss_colformats)
                            widths for the specified columns
s(pss_colwidths)
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
                            column name containing the effect-size parameter
s(pss_delta)
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
```

## References

- Cain, M. 2021. Calculating power using Monte Carlo simulations, part 5: Structural equation models. The Stata Blog: Not Elsewhere Classified. https://blog.stata.com/2021/08/19/calculating-power-using-monte-carlo-simulations-part-5-structural-equation-models/.
- Huber, C. 2019. Calculating power using Monte Carlo simulations, part 2: Running your simulation using power. The Stata Blog: Not Elsewhere Classified. https://blog.stata.com/2019/01/29/calculating-power-using-monte-carlosimulations-part-2-running-your-simulation-using-power/.

# Also see

- [PSS-2] power Power and sample-size analysis for hypothesis tests
- [PSS-2] Intro (power) Introduction to power and sample-size analysis for hypothesis tests
- [PSS-5] Glossary
- [ADAPT] gsdesign usermethod Add your own methods to the gsdesign command

Stata, Stata Press, Mata, NetCourse, and NetCourseNow are registered trademarks of StataCorp LLC. Stata and Stata Press are registered trademarks with the World Intellectual Property Organization of the United Nations. StataNow is a trademark of StataCorp LLC. Other brand and product names are registered trademarks or trademarks of their respective companies. Copyright © 1985–2025 StataCorp LLC, College Station, TX, USA. All rights reserved.



For suggested citations, see the FAQ on citing Stata documentation.