Title

stata.com

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

Description Syntax Remarks and examples References Also see

Description

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

Syntax

Compute sample size

`ciwidth` *usermethod* `...,` `width(`*numlist*`)` [`probwidth(`*numlist*`)` *ciwidthopts* *useropts*]

Compute CI width

`ciwidth` *usermethod* `...,` *nspec* [`probwidth(`*numlist*`)` *ciwidthopts* *useropts*]

Compute probability of CI width

`ciwidth` *usermethod* `...,` *nspec* `width(`*numlist*`)` [*ciwidthopts* *useropts*]

usermethod is the name of the method you would like to add to the `ciwidth` command. When naming your `ciwidth` 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 14 characters.

useropts are the options supported by your method *usermethod*.

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

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

Remarks and examples

stata.com

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

1. write an r-class program called `ciwidth_cmd_mymethod` that computes sample size, probability of CI width, or CI width and follows `ciwidth`'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 `ciwidth` like any other official `ciwidth` method.

Remarks are presented under the following headings:

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 CI width for a one-mean normal-based CI given sample size, standard deviation, and confidence level. For simplicity, we assume a two-sided CI. We will call our new method `mymean`. (Note that this method is available in the official `ciwidth onemean` command when you specify the `knownsd` option.)

We create an ado-file called `ciwidth_cmd_mymean.ado` that contains the following Stata program:

```
// evaluator
program ciwidth_cmd_mymean, rclass
        version 18.0
        /* parse options */
        syntax, n(integer)          /// sample size
                [ Level(cilevel)    /// confidence level
                  Stddev(real 1) ]  /// standard deviation
        /* compute CI width */
        tempname width
        scalar `width' = 2*invnormal(1/2+`level'/200)*`stddev'/sqrt(`n')
        /* store results */
        return scalar level   = `level'
        return scalar N       = `n'
        return scalar width   = `width'
        return scalar stddev  = `stddev'
end
```

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

The `ciwidth_cmd_mymean` program has two of `ciwidth`'s common options, `level()` for confidence level and `n()` for sample size, and it has its own option, `stddev()`, with the minimum abbreviation `s()` and default value of 1, to specify a standard deviation.

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

Finally, the resulting CI width and other results are stored in return scalars. Following `ciwidth`'s convention for naming commonly returned results, the confidence level is stored in `r(level)`, the sample size in `r(N)`, and the computed CI width in `r(width)`. The program additionally stores the standard deviation in `r(stddev)`.

We can now use `mymean` within `ciwidth` as we would any other existing method of `ciwidth`:

```
. ciwidth mymean, level(95) n(10) stddev(0.25)
Estimated width
Two-sided CI
```

level	N	width
95	10	.3099

We can check our result using the official `ciwidth onemean`:

```
. ciwidth onemean, level(95) n(10) sd(0.25) knownsd
Estimated width for a one-mean CI
Normal two-sided CI
Study parameters:
        level =     95.00
            N =        10
           sd =    0.2500
Estimated width:
        width =    0.3099
```

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

```
. ciwidth mymean, level(95) n(10 20) stddev(0.25)
Estimated width
Two-sided CI
```

level	N	width
95	10	.3099
95	20	.2191

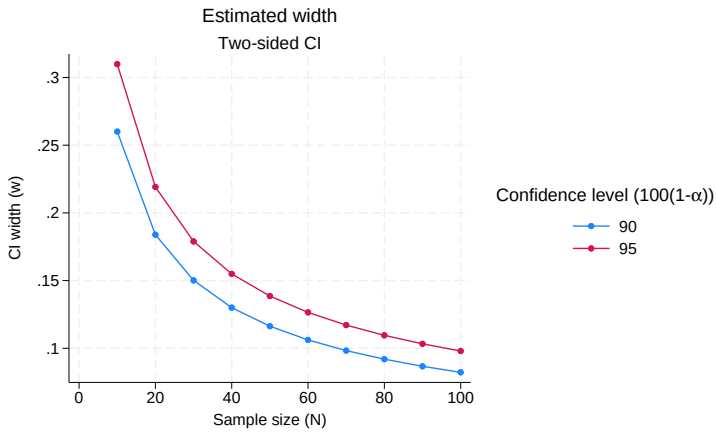
We can also compute results for multiple sample sizes and confidence levels without any additional effort on our part:

```
. ciwidth mymean, level(90 95) n(10 20) stddev(0.25)
Estimated width
Two-sided CI
```

level	N	width
90	10	.2601
90	20	.1839
95	10	.3099
95	20	.2191

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

```
. ciwidth mymean, level(90 95) n(10(10)100) stddev(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; see *More examples: Compute probability of CI width for a one-proportion CI*. You can also customize your tables and graphs with a little extra effort.

Steps for adding a new method to the ciwidth command

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

1. Create the evaluator, an r-class program called `ciwidth_cmd_`*usermethod* and defined by the ado-file `ciwidth_cmd_`*usermethod*`.ado`, that performs precision and sample-size computations and follows `ciwidth`'s convention for naming options and storing results.
2. Optionally, create an initializer, an s-class program called `ciwidth_cmd_`*usermethod*`_init` and defined by the ado-file `ciwidth_cmd_`*usermethod*`_init.ado`, that specifies information about table columns, options that may allow a numlist, and so on.
3. Optionally, create a parser, a program called `ciwidth_cmd_`*usermethod*`_parse` and defined by the ado-file `ciwidth_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 `ciwidth`:

```
. ciwidth usermethod ...
```

You may also use programs within `ciwidth` 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 `ciwidth` command to automatically recognize its common options, you must ensure that you follow `ciwidth`'s naming convention for these options in your program. For example, `ciwidth` specifies the confidence level in the `level()` option with minimum abbreviation of `l()`. You need to ensure that you use the same option with the same abbreviation in your evaluator to specify the confidence level. The same applies to all of `ciwidth`'s common options described in [PSS-3] **ciwidth**.

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

Unlike `ciwidth`'s official methods, user-defined methods do not require specifying the `probwidth()` option by default because some computations, such as our earlier normal-based one-mean CI example, may not need the probability of CI width. The probability of CI width is often needed when the computation of the width depends on unknown parameters that are themselves estimated from the data. For instance, if the standard deviation is not known a priori, the computation of the CI width for a one-mean CI incorporates the uncertainty about the specified standard deviation because its estimate may vary from one sample to another. The specified probability of CI width is used to ensure that the estimated CI width is no larger than the desired width of a future CI with the prespecified probability. This is the default method of `ciwidth onemean`. Also see *More examples: Compute probability of CI width for a one-proportion CI* for an example of computing probability of CI width.

To produce tables and graphs of results, you must ensure that your evaluator follows `ciwidth`'s convention for storing results. `ciwidth`'s commonly stored results are described in *Stored results* of [PSS-3] **ciwidth**. For example, the value for a confidence level should be stored in the scalar `r(level)`, the value for a total sample size in the scalar `r(N)`, the value for CI width in the scalar `r(width)`, the value for probability of CI width, if available, in the scalar `r(Pr_width)`, and so on.

You can also store additional method-specific results, but `ciwidth` 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 `ciwidth` 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 `ciwidth` 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(prss_numopts)` within the `ciwidth_cmd_`*usermethod*`_init` program.

Recall our earlier example in which we added the `mymean` method, calculating the CI width of a two-sided normal CI for one-sample mean, to `ciwidth`. We computed CI widths for multiple values of confidence level and sample size. What if we would also like to specify multiple values of standard deviation in the `stddev()` option of `mymean`? If we do this now, we will receive an error message,

```
. ciwidth mymean, level(95) n(10) stddev(0.25 0.5)
option stddev() incorrectly specified
r(198);
```

because the `stddev()` option is not allowed to include *numlist* by the evaluator and is not one of `ciwidth`'s common options. To make `ciwidth` recognize this option as one allowing *numlist*, we need to specify this in the initializer. Following the guidelines, we create an ado-file called `ciwidth_cmd_mymean_init.ado` and specify the name of the `stddev()` option (with the corresponding abbreviation) in the s-class macro `s(prss_numopts)` within the `ciwidth_cmd_mymean_init` program.

```
// initializer
program ciwidth_cmd_mymean_init, sclass
        version 18.0
        sreturn clear
        sreturn local prss_numopts "Stddev"
end
```

We now can specify multiple standard deviations:

```
. ciwidth mymean, level(95) n(10) stddev(0.25 0.5)
Estimated width
Two-sided CI
```

level	N	width
95	10	.3099
95	10	.6198

Customizing default tables

The `ciwidth` command with user-defined methods always displays results in a table. By default, it displays columns `level` or `alpha` (whichever is specified), `N`, and `width`, which contain the confidence level, the sample size, and the CI width, respectively. If option `probwidth()` or both options `n()` and `width()` are specified, the `Pr_width` column is also shown in the default table following the `N` column. 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 `level`, `alpha`, `width`, and `Pr_width` 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

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

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

If you want to reset `ciwidth`'s list of supported columns, that is, to specify all the supported columns manually, you should use the `s(prss_allcolnames)` macro. The supported columns will then include only the ones you listed in the macro. If you specify `s(prss_allcolnames)`, you

must remember to include ciwidth's main columns N, width, level, Pr_width (if applicable) in your list. Otherwise, you may not be able to use some of ciwidth's functionality, such as default graphs. If s(prss_colnames) is specified together with s(prss_allcolnames), the former will be ignored. The specified supported columns will be automatically displayed in the default table unless s(prss_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 level is stored in r(level), the value for column width is stored in r(width), and the value for column N is stored in r(N).

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

Modifying the default table columns

By default, ciwidth displays the specified supported columns. If you want to display only a subset of those columns, you can use either s(prss_tabcolnames) or s(prss_alltabcolnames) to specify the columns to be displayed. You specify additional columns to be displayed in s(prss_tabcolnames) and a complete list of the displayed columns in s(prss_alltabcolnames). If you specify s(prss_tabcolnames), the displayed columns will include level or alpha (whichever is specified with the command), N, Pr_width (if applicable), width and the additional columns you specified. If you specify s(prss_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(prss_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(prss_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(prss_colnames) or s(prss_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(prss_colformats) macro. The formats must be quoted and specified for all columns listed in s(prss_colnames) or s(prss_allcolnames).

The default column widths are the widths of the default formats plus one. You can specify your own column widths in the s(prss_colwidths) macro. The widths must be specified for all columns listed in s(prss_colnames) or s(prss_allcolnames).

Example continued

Continuing our mymean example, we want to add a column containing the specified standard deviation to the list of supported columns. The specified standard deviation is stored in r(stddev) in the mymean evaluator, so the name of our column is stddev. We specify it in s(prss_colnames) in our initializer as follows:

```
// initializer
program drop ciwidth_cmd_mymean_init
program ciwidth_cmd_mymean_init, sclass
        version 18.0
        sreturn clear
        sreturn local prss_numopts "Stddev"
        sreturn local prss_colnames "stddev"  // <-- new line
end
```

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

```
. ciwidth mymean, level(95) n(10) stddev(0.25)
Estimated width
Two-sided CI
```

level	N	width	stddev
95	10	.3099	.25

We can also change the default column label of the `stddev` column to "Std. dev.". Note that we can do this within `ciwidth`'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(prss_collabels)` macro.

```
// initializer
program drop ciwidth_cmd_mymean_init
program ciwidth_cmd_mymean_init, sclass
        version 18.0
        sreturn clear
        sreturn local prss_numopts "sd"
        sreturn local prss_colnames "stddev"
        sreturn local prss_collabels `""Std. dev.""'  // <-- new line
end
```

The column containing standard deviation now has the new label

```
. ciwidth mymean, level(95) n(10) stddev(0.25)
Estimated width
Two-sided CI
```

level	N	width	Std. dev.
95	10	.3099	.25

Customizing default graphs

By default, `ciwidth` plots the estimated CI width 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 CI width on the x axis. See [PSS-3] **ciwidth, graph** for details about other default settings.

You can overwrite the default column labels displayed on the graph by specifying the `s(prss_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(prss_colgrsymbols)`. The symbols must be specified for all columns listed in `s(prss_colnames)` or `s(prss_allcolnames)`.

Other settings

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

`ciwidth` *usermethod* uses the following generic titles: "Estimated sample size" for sample-size determination, "Estimated width" for CI width determination, and "Estimated probability of width" for probability of CI width determination. You can extend these titles to be more specific to your method by adding text in the `s(prss_title)` macro. For example, if `s(prss_title)` contains "for my CI", the resulting titles will be "Estimated sample size for my CI", "Estimated width for my CI", and "Estimated probability of width for my CI".

`ciwidth` *usermethod* uses the following generic subtitles: "Two-sided CI" for a two-sided CI, "One-sided upper CI" when the `upper` option is specified, and "One-sided lower CI" when the `lower` option is specified. You can change the default subtitle by specifying the `s(prss_subtitle)` macro.

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(prss_samples)` macro to contain `twosample` in the initializer. If any of the two-sample options `n1()`, `n2()`, and `nratio()` are specified, `ciwidth` 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(prss_samples)` to `twosample`. If you do not want `ciwidth` to respect the two-sample options, you should set `s(prss_samples)` to `onesample`.

Handling parsing more efficiently

The `ciwidth` command checks its common options, but you are responsible for checking your method-specific options, *useropts*, or their interaction with `ciwidth`'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: Compute probability of CI width for a one-proportion CI

For some CIs, the expressions for the required sample size or CI width may not be available or difficult to compute. In such cases, you can use simulation to obtain the results. And you can turn your simulation program into a user-defined `ciwidth` method. Huber (2019a) and Huber (2019b) describe how to compute power by simulation and integrate the simulation program in the `power` command. The same principles apply to the simulation of CI width or probability of CI width and its integration in the `ciwidth` command.

The `ciwidth` command does not provide precision and sample-size analysis for CIs for proportions. The width of CIs for proportions depends on the estimates of proportions. Its estimation thus needs to account for the uncertainty in the proportion estimates. There are no closed-form solutions to compute the required sample size or width for CIs for proportions that incorporate the probability of CI width. But we can compute the probability of CI width for a given sample size and target width using simulation. We can then vary the sample sizes to see which ones correspond to high values of the probability of CI width for the desired CI width. Let's do this for the binomial CI for one proportion.

We can estimate the probability of CI width as the proportion of times the width of the CI computed from simulated samples is less than or equal to the desired CI width for a given sample size and probability of success. Our steps are to 1) create a program that simulates the data and computes the width of the estimated CI; 2) run the program multiple times and compute the probability of CI width; and 3) add our computations to `ciwidth` as a new method. We can actually compute the probability of CI width exactly, without the simulation, for the binomial CI. So we compare our simulation results with the exact computation in step 4.

Step 1: Program to simulate the data and compute the CI width

We start with a simple program `myonepropsim` below.

```
program myonepropsim, rclass
        version 18.0
        args n p level
        clear
        set obs `n'
        generate byte y = rbinomial(1, `p')
        ci proportions y, level(`level')
        return scalar w = r(ub)-r(lb)
end
```

Our program requires three arguments: `n` for sample size, `p` for proportion estimate, and `level` for confidence level. It generates `` `n' `` observations for the binary outcome `y` from a Bernoulli distribution with a specified probability of success `` `p' ``. (`` `' `` refers to the specified values for the arguments.) It uses the `ci proportions` command ([R] **ci**) to estimate the proportion of successes (`y==1`) and its binomial CI. It then computes and stores in the return scalar `r(w)` the estimated CI width—the difference between the upper and lower CI bounds stored by `ci proportions` in return scalars `r(ub)` and `r(lb)`, respectively.

Let's run our program. Suppose that we want to simulate 50 Bernoulli observations with a low success probability of 0.1 and compute the width of the corresponding 95% two-sided binomial CI for the proportion of successes. Because we randomly generate the data, we use `set seed` prior to calling `myonepropsim` for reproducibility.

```
. set seed 1234
. myonepropsim 50 0.1 95
Number of observations (_N) was 0, now 50.
```

Variable	Obs	Proportion	Std. err.	Binomial exact [95% conf.	interval]
y	50	.18	.0543323	.0857621	.3143694

```
.
. return list
scalars:
                  r(w) =  .2286073331759869
```

From the stored results, the estimated CI width, `r(w)`, is 0.23.

Step 2: Estimating probability of CI width using simulation

Suppose that our target CI width is 0.2. To estimate the probability of CI width, we need to call our `myonepropsim` program multiple times and compute the proportion of times the estimated CI width was less than or equal to our target width of 0.2.

Stata has a handy command to run simulations—`simulate` ([R] **simulate**).

```
. set seed 1234
. simulate w=r(w), reps(100): myonepropsim 50 0.1 95
      Command: myonepropsim 50 0.1 95
            w: r(w)
Simulations (100): .........10.........20.........30.........40.........50.....
> ....60.........70.........80.........90.........100 done
. count if w <= 0.2
  75
. display r(N)/100
.75
```

`simulate` runs `myonepropsim` 100 times, as specified by `simulate`'s `reps()` option, and stores the computed CI widths in the `w` variable, as requested by the `w=r(w)` specification. We then count the number of observations of `w` that are less than or equal to 0.2 and estimate the probability of CI width to be 0.75.

Step 3: Adding probability of CI width computation to ciwidth

Our final step is to combine all of our computations in a single program and integrate it into the `ciwidth` command. Following `ciwidth`'s convention, we call our new program `ciwidth_cmd_myonepropsim`.

```
program ciwidth_cmd_myonepropsim, rclass
        version 18.0
        /* parse command arguments and options */
        syntax anything(name=p),          /// proportion estimate
                        n(integer)        /// sample size
                        Width(real)       /// target CI width
                        [ Level(cilevel) /// confidence level
                        reps(integer 100) qui ]
        /* compute probability of CI width using simulation */
        display as txt _n "Computing Pr(width) for n=`n' and width=`w' ..."
        `qui' simulate w=r(w), reps(`reps'): myonepropsim `n' `p' `level'
        quietly count if w <= `width'
        /* store results */
        return scalar Pr_width = r(N)/`reps'
        return scalar level = `level'
        return scalar N = `n'
        return scalar width = `width'
        return scalar p = `p'
end
```

As in *A quick example*, we use `syntax` to parse options. We have three new options. The `width()` option, with the minimum abbreviation `w()`, is one of `ciwidth`'s common options; it specifies the target CI width. The `reps()` option is specific to our method—it specifies the number of replications for the simulation, with the default of 100 replications. Finally, `qui` suppresses the output from the `simulate` command that we display by default.

In addition to options, our program requires that the proportion estimate be specified as a command argument. We could have specified it as an option, say, `proportion(real)`, but here we wanted to demonstrate how to handle arguments with user-defined `ciwidth` methods. Also, official `ciwidth` methods typically specify estimates of parameters of interest, such as proportion, as command arguments, which are specified following the command name.

The block after `syntax` includes our earlier `simulate` command to which we now pass the content of the specified command argument and options instead of the hard-coded values. We also compute the estimate of probability of CI width and store it in the `r(Pr_width)` scalar, following `ciwidth`'s naming convention for the common stored results; see *Stored results* of [PSS-3] **ciwidth**. We also store other results in the corresponding return scalars.

Let's recompute the probability of CI width from *Step 2: Estimating probability of CI width using simulation* but now using `ciwidth myonepropsim`.

```
. set seed 1234
. ciwidth myonepropsim 0.1, n(50) width(0.2)
Computing Pr(width) for n=50 and width=.2 ...
      Command: myonepropsim 50 .1 95
            w: r(w)
Simulations (100): .........10.........20.........30.........40.........50.....
> ....60.........70.........80.........90.........100 done
Estimated probability of width
Two-sided CI
```

level	N	Pr_width	width
95	50	.75	.2

We obtain the same estimate of 0.75. We used the default value of the `level()` option, which is `level(95)` or as set by `set level`; see [R] **level**.

Notice that our default table now contains the `Pr_width` column. Because we specified both `n()` and `width()`, `ciwidth` recognized this as the case for computing probability of CI width and automatically added its column to the default table. However, we are missing the proportion estimate in our default table. `ciwidth` is not aware of user-defined command arguments until we specify them in the initializer.

```
program ciwidth_cmd_myonepropsim_init, sclass
        version 18.0
        sreturn clear
        sreturn local prss_argnames = "p"
        sreturn local prss_colnames = "p"
        sreturn local prss_subtitle = "Two-sided binomial CI"
end
```

We list the name of the stored result containing the proportion estimate in macro `prss_argnames` to allow the specification of multiple values for the proportion and in macro `prss_colnames` to add the proportion column to the default table. We also specify a more descriptive subtitle to be used in the output.

If we rerun our previous command (with the `qui` option to suppress the output from the `simulate` command),

```
. set seed 1234
. ciwidth myonepropsim 0.1, n(50) width(0.2) qui
Computing Pr(width) for n=50 and width=.2 ...
Estimated probability of width
Two-sided binomial CI
```

level	N	Pr_width	width	p
95	50	.75	.2	.1

we will now see the proportion column in the table and the new subtitle.

The estimated probability of CI width of 0.75 is somewhat low. We can specify multiple sample sizes to find an acceptable value of probability of CI width.

```
. set seed 1234
. ciwidth myonepropsim 0.1, n(50 70 100) width(0.2) qui
Computing Pr(width) for n=50 and width=.2 ...
Computing Pr(width) for n=70 and width=.2 ...
Computing Pr(width) for n=100 and width=.2 ...
Estimated probability of width
Two-sided binomial CI
```

level	N	Pr_width	width	p
95	50	.75	.2	.1
95	70	1	.2	.1
95	100	1	.2	.1

We can further explore the sample sizes between 50 and 70:

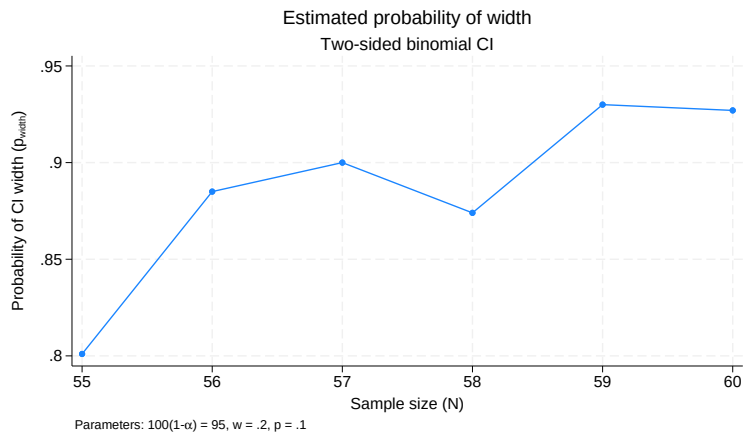
```
. set seed 1234
. ciwidth myonepropsim 0.1, n(50(5)70) width(0.2) qui
Computing Pr(width) for n=50 and width=.2 ...
Computing Pr(width) for n=55 and width=.2 ...
Computing Pr(width) for n=60 and width=.2 ...
Computing Pr(width) for n=65 and width=.2 ...
Computing Pr(width) for n=70 and width=.2 ...
Estimated probability of width
Two-sided binomial CI
```

level	N	Pr_width	width	p
95	50	.75	.2	.1
95	55	.78	.2	.1
95	60	.97	.2	.1
95	65	.98	.2	.1
95	70	1	.2	.1

Finally, we can compute probability of CI widths for sample sizes between 55 and 60 and plot the results in addition to the table. We also specify more replications to obtain more precise estimates of the probability of CI width.

```
. set seed 1234
. ciwidth myonepropsim 0.1, n(55(1)60) width(0.2) qui reps(1000) table graph
Computing Pr(width) for n=55 and width=.2 ...
Computing Pr(width) for n=56 and width=.2 ...
Computing Pr(width) for n=57 and width=.2 ...
Computing Pr(width) for n=58 and width=.2 ...
Computing Pr(width) for n=59 and width=.2 ...
Computing Pr(width) for n=60 and width=.2 ...
Estimated probability of width
Two-sided binomial CI
```

level	N	Pr_width	width	p
95	55	.801	.2	.1
95	56	.885	.2	.1
95	57	.9	.2	.1
95	58	.874	.2	.1
95	59	.93	.2	.1
95	60	.927	.2	.1



For example, the sample size of 57 corresponds to the probability of 0.9 that the width of a future 95% two-sided binomial CI for one proportion will not exceed 0.2 given the proportion estimate of 0.1.

The probability of CI width is typically a monotonically increasing function of the sample size. However, similarly to the power of the binomial test, the probability of CI width for the binomial CI may not be monotonic with respect to the sample size, as we see in this example, because of the discrete nature of the binomial distribution.

Of course, because we use simulation, if we rerun `ciwidth myonepropsim` with a different seed, we will get different results. The results, however, should be comparable provided the number of replications is sufficiently large.

Step 4: Computing exact probability of CI width

We can compute the probability of CI width for the binomial CI more easily by using the exact formula

$$\Pr(w) = \sum_{k=0}^{n} \texttt{binomialp}(n, k, p) \times I(w_{k,n} \leq w_{\text{target}})$$

where $\texttt{binomialp}(n, k, p)$ is the probability of observing k successes in n trials with the success probability p; w_{target} is the target CI width; $w_{k,n}$ is the width of the binomial CI computed given k observed successes in n trials; and the indicator function

$$I(w_{k,n} \leq w_{\text{target}}) = \begin{cases} 1, \text{ if } w_{k,n} \leq w_{\text{target}} \\ 0, \text{ otherwise} \end{cases}$$

The `ciwidth_cmd_myoneprop` program below uses the formula above to compute the probability of CI width.

```
program ciwidth_cmd_myoneprop, rclass
        version 18.0
        /* parse command arguments and options */
        syntax anything(name=p),            /// proportion estimate
                        n(integer)          /// sample size
                        Width(real)         /// target CI width
                        [ Level(cilevel) ] /// confidence level
        /* compute probability of CI width using exact formula */
        tempname Pr_width
        scalar `Pr_width' = 0
        forvalues k = 0/`n' {
                quietly cii proportions `n' `k', level(`level')
                if (r(ub)-r(lb) <= `width') {
                        scalar `Pr_width' = `Pr_width' + binomialp(`n',`k',`p')
                }
        }
        /* store results */
        return scalar Pr_width = `Pr_width'
        return scalar level = `level'
        return scalar N = `n'
        return scalar width = `width'
        return scalar p = `p'
end
```

This program is similar to our earlier `ciwidth_cmd_myonepropsim` program but without the `reps()` and `qui` options and using the exact formula instead of simulation to compute the probability of CI width. Also, instead of using `ci proportions`, which estimates the binomial CI from the data, we use its immediate version, `cii proportions`, which uses the numbers supplied in `` `n' `` and `` `k' `` to compute the CI; see **[U] 19 Immediate commands** for a general discussion of immediate commands.

We followed `ciwidth`'s naming convention for the `ciwidth_cmd_myoneprop` program, so we can use `myoneprop` with `ciwidth`. Let's compute the exact probability of CI width for sample sizes, n, between 55 and 60 given the target CI width, w_{target}, of 0.2 and probability of success, p, of 0.1, using the default 95% confidence level.

```
. ciwidth myoneprop 0.1, n(55(1)60) width(0.2)
Estimated probability of width
Two-sided CI
```

level	N	Pr_width	width
95	55	.8196	.2
95	56	.897	.2
95	57	.888	.2
95	58	.8785	.2
95	59	.9334	.2
95	60	.9269	.2

Our results are similar to the simulation results from the last table in *Step 3: Adding probability of CI width computation to ciwidth*. We can match the exact results even more closely if we use more replications, say, 10,000, during simulation.

Initializer's s() return settings

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

Macros

`s(prss_samples)`	`onesample` for a one-sample CI or `twosample` for a two-sample CI
`s(prss_colnames)`	columns to be added to the default supported columns
`s(prss_allcolnames)`	all supported columns
`s(prss_tabcolnames)`	columns to be added to the default table
`s(prss_alltabcolnames)`	all columns to be displayed in the default table
`s(prss_collabels)`	labels for the specified columns
`s(prss_colformats)`	formats for the specified columns
`s(prss_colwidths)`	widths for the specified columns
`s(prss_colgrlabels)`	labels to be used to label columns on the graph
`s(prss_colgrsymbols)`	symbols to be used to label columns on the graph
`s(prss_argnames)`	column names containing command arguments
`s(prss_title)`	method-specific title
`s(prss_subtitle)`	subtitle

References

Huber, C. 2019a. Calculating power using Monte Carlo simulations, part 1: The basics. *The Stata Blog: Not Elsewhere Classified*. https://blog.stata.com/2019/01/10/calculating-power-using-monte-carlo-simulations-part-1-the-basics/.

——. 2019b. 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-carlo-simulations-part-2-running-your-simulation-using-power/.

Also see

[PSS-3] **ciwidth** — Precision and sample-size analysis for CIs

[PSS-3] **Intro (ciwidth)** — Introduction to precision and sample-size analysis for confidence intervals

[PSS-5] **Glossary**