Stata

Setup

We will need several Stata packages to draw a violin plot. We can use "ssc install [package_name]" to install them.

```
clear all
// ssc install violinplot, replace     // module to draw violin plots
// ssc install dstat, replace          // violinplot's dependency, module to compute summary statistics
// ssc install moremata, replace       // violinplot's dependency, module (Mata) to provide various fun
// ssc install palettes, replace       // violinplot's dependency, module to provide color palettes
// ssc install colrspace, replace      // violinplot's dependency, module providing a class-based color
```

We will also set the pseudo-random number generator seed to `02138` to make the stochastic components of our simulations reproducible (this is similar to the process in R and Python).

```
set seed 02138
```

Data simulation step by step

To give an overview of the power simulation task, we will simulate data from a design with crossed random factors of subjects and songs (see Power of What? for design details), fit a model to the simulated data, recover from the model output the parameter values we put in, calculate power, and finally automate the whole process so that we can calculate power for different effect sizes. ### Establish the simulation parameters

Before we start, let's set some global parameters for our power simulations.

```
// number of simulation replications for power calculation
global reps = 30

// specified alpha for power calculation
global alpha = 0.05
```

Establish the data-generating parameters

The first thing to do is to set up the parameters that govern the process we assume gave rise to the data - the *data-generating process*, or DGP. We previously decided upon the the data-generating parameters (see Power of What?), so we just need to code them here.

Note: There is a difference between Stata and R and the Python: We decrease the data-generating parameters to simplify our model, and we delete some parameters: by-song random intercept `omega_0`, by-subject random slope sd `tau_1`, and the correlation between intercept and slope `rho`.

```
// set all data-generating parameters
global beta_0 = 60   // intercept; i.e., the grand mean
global beta_1 = 5    // slope; i.e., effect of category
global tau_0 = 7     // by-subject random intercept sd
global sigma = 8     // residual (error) sd
```

Simulate the sampling process

Next, we will simulate the sampling process for the data. First, let's define parameters related to the number of observations.

```
// Set number of subjects and songs
global n_subj = 25   // number of subjects
global n_pop = 15    // number of songs in pop category
global n_rock = 15   // number of songs in rock category
global n_all = $n_pop + $n_rock
```

Simulate the sampling of songs We need to create a table listing each song i, which category it is in (`rock` or `pop`).

```
// simulate a sample of songs
quietly {
  clear
  set obs $n_all

  // Generate a sequence of song ids
  gen song_id = _n

  // Generate the category variable
  gen category = "pop"
  replace category = "rock" if song_id > $n_pop

  // Generate the genre variable
  gen genre_i = 0
  replace genre_i = 1 if song_id > $n_pop

  gen key = 1

  save "./data/songs.dta", replace
}
list in 1/10
```

```
     | song_id   category   genre_i   key |
     |------------------------------------|
  1. |       1        pop         0     1 |
  2. |       2        pop         0     1 |
  3. |       3        pop         0     1 |
  4. |       4        pop         0     1 |
  5. |       5        pop         0     1 |
     |------------------------------------|
  6. |       6        pop         0     1 |
  7. |       7        pop         0     1 |
  8. |       8        pop         0     1 |
  9. |       9        pop         0     1 |
 10. |      10        pop         0     1 |
     +------------------------------------+
```

Simulate the sampling of subjects Now we simulate the sampling of participants, which results in table listing each individual and their random effect (a random intercept). To do this, we must sample t_0 from a normal distribution.

We will use the function `rnormal`, which generates a simulated value from a univariate normal distribution with a mean of 0 and a standard deviations of `tau_0` of each variable.

```
// simulate a sample of subjects
quietly {
  clear
  set obs $n_subj

  // Generate the by-subject random intercept
  gen t0 = rnormal(0, $tau_0)

  // Generate a sequence of subject ids
  gen subj_id = _n

  gen key = 1

  save "./data/subjects.dta", replace
}
list in 1 / 10
```

```
     |        t0   subj_id   key |
     |---------------------------|
  1. |  4.356949         1     1 |
  2. |  .0887434         2     1 |
  3. |  .1867903         3     1 |
  4. |  11.24607         4     1 |
  5. | -1.842066         5     1 |
     |---------------------------|
  6. |  1.966723         6     1 |
  7. |  2.544997         7     1 |
  8. | -9.950144         8     1 |
  9. | -8.176116         9     1 |
 10. | -4.609569        10     1 |
     +---------------------------+
```

Check the simulated values Let's do a quick sanity check by comparing our simulated values to the parameters we used as inputs. Because the sampling process is stochastic, we shouldn't expect that these will exactly match for any given run of the simulation.

```
quietly {
  use "./data/subjects.dta"

  qui summarize t0
  egen tau_0_s = sd(t0)
}
display "tau_0, " $tau_0 ", " tau_0_s
```

```
tau_0, 7, 7.4337502
```

Simulate trials Since all subjects rate all songs (i.e., the design is fully crossed) we can set up a table of trials by including every possible combination of the rows in the `subjects` and `songs` tables. Each trial has random error associated with it, reflecting fluctuations in trial-by-trial ratings due to unkown factors. We simulate this by sampling values from a univariate normal distribution with a mean of 0 and a standard deviation of `sigma`.

```
// cross subject and song IDs; add an error term
quietly {
  use "./data/subjects.dta"
  cross using "./data/songs.dta"
  drop key
  sort subj_id song_id

  gen e_ij = rnormal(0, $sigma)

  save "./data/data_sim_tmp.dta", replace
}
list in 1 / 10
```

```
     |       t0   subj_id   song_id   category   genre_i        e_ij |
     |---------------------------------------------------------------|
  1. | 4.356949         1         1        pop         0    4.979371 |
  2. | 4.356949         1         2        pop         0    .1014211 |
  3. | 4.356949         1         3        pop         0    .2134746 |
  4. | 4.356949         1         4        pop         0    12.85266 |
  5. | 4.356949         1         5        pop         0   -2.105218 |
     |---------------------------------------------------------------|
  6. | 4.356949         1         6        pop         0    2.247684 |
  7. | 4.356949         1         7        pop         0    2.908567 |
  8. | 4.356949         1         8        pop         0   -11.37159 |
  9. | 4.356949         1         9        pop         0   -9.344132 |
 10. | 4.356949         1        10        pop         0   -5.268079 |
     +---------------------------------------------------------------+
```

Calculate response values With this resulting `trials` table, in combination with the constants `beta_0` and `beta_1`, we have the full set of values that we need to compute the response variable `liking_ij` according the linear model we defined previously (see Power of What?).

```
quietly {
  use "./data/data_sim_tmp.dta"

  gen liking_ij = $beta_0 + t0 +  $beta_1  * genre_i + e_ij
  keep subj_id song_id category genre_i liking_ij

  save "./data/data_sim.dta", replace
}
list in 1 / 10
```

```
     | subj_id   song_id   category   genre_i   liking~j |
     |---------------------------------------------------|
  1. |       1         1        pop         0   69.33632 |
  2. |       1         2        pop         0   64.45837 |
  3. |       1         3        pop         0   64.57043 |
  4. |       1         4        pop         0    77.2096 |
```

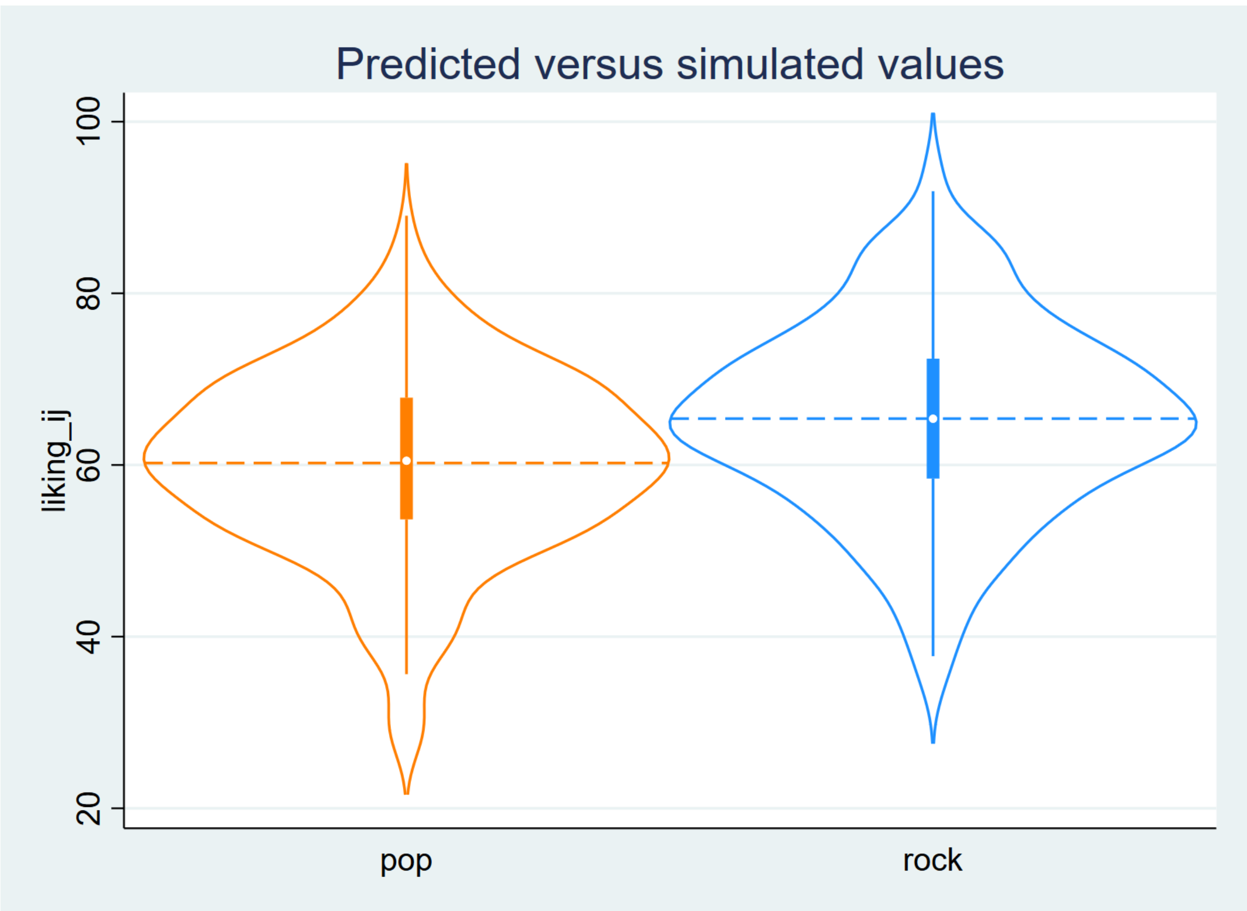
```
  5. |        1         5        pop         0   62.25173 |
     |---------------------------------------------------|
  6. |        1         6        pop         0   66.60463 |
  7. |        1         7        pop         0   67.26552 |
  8. |        1         8        pop         0   52.98536 |
  9. |        1         9        pop         0   55.01282 |
 10. |        1        10        pop         0   59.08887 |
     +---------------------------------------------------+
```

Plot the data Let's visualize the distribution of the response variable for each of the two song genres and superimpose the simulated parameter estimates for the means of these two groups.

```
quietly {
  use "./data/data_sim.dta"

  // Set the palette colors
  local palette "orange dodgerblue"

  // Create a violin plot for actual data
  violinplot liking_ij, over(category) colors(`palette') vertical mean(type(line) lp(dash) stat(mean))
  graph export "./figures/violin.png", replace
}
```



Analyze the simulated data

Now we can analyze our simulated data in a linear mixed effects model using the function `mixed`. The model formula in `mixed` maps onto how we calculated our `liking_ij` outcome variable above.

```
quietly use "./data/data_sim_tmp.dta"
mixed liking_ij genre_i || subj_id:
quietly estimates save "./data/data_sim_estimates.ster", replace
```

```
variable liking_ij not found
r(111);

end of do-file
r(111);
```

The terms in formula are as follows:

- `liking_ij` is the response.
- `genre_i` is the dummy coded variable identifying whether song i belongs to the pop or rock genre.
- `|| subj_id` specified a subject-specific random intercept (`t0`)

Now we can estimate the model.

```
quietly estimates use "./data/data_sim_estimates.ster"
quietly matrix b = e(b)
matrix list b
```

```
b[1,4]
     liking_ij:  liking_ij:   lns1_1_1:    lnsig_e:
       genre_i       _cons       _cons       _cons
y1   5.1710617   60.229571   2.0205456   2.0902013
```

Data simulation automated

Now that we've tested the data generating code, we can put it into a function so that it's easy to run it repeatedly.

```
capture program drop sim_data
program define sim_data
    args n_subj n_pop n_rock beta_0 beta_1 tau_0 sigma

  // simulate a sample of songs
    clear
    local n_all = `n_pop' + `n_rock'
    set obs `n_all'
    gen song_id = _n
    gen category = "pop"
    replace category = "rock" if song_id > `n_pop'
    gen genre_i = 0
    replace genre_i = 1 if song_id > `n_pop'
    gen key = 1
```

```
    save "./data/songs.dta", replace

  // simulate a sample of subjects
    clear
    set obs `n_subj'

    gen t0 = rnormal(0, `tau_0')
    gen subj_id = _n
    gen key = 1
    save "./data/subjects.dta", replace

  // cross subject and song IDs
    use "./data/subjects.dta"
    cross using "./data/songs.dta"
    drop key
    sort subj_id song_id
    gen e_ij = rnormal(0, `sigma')

    gen liking_ij = `beta_0' + t0 + `beta_1' * genre_i + e_ij
    keep subj_id song_id category genre_i liking_ij
end
```

Power calculation single run

We can wrap the data generating function and modeling code in a new function `single_run()` that returns the analysis results for a single simulation run. We'll suppress warnings and messages from the modeling fitting process, as these sometimes occur with simulation runs that generate extreme realized values for parameters.

```
capture program drop single_run
program define single_run, rclass
    args n_subj n_pop n_rock beta_0 beta_1 tau_0 sigma

    clear
    sim_data `n_subj' `n_pop' `n_rock' `beta_0' `beta_1' `tau_0' `sigma'
    mixed liking_ij genre_i || subj_id:, noretable nofetable noheader nogroup

  estimates clear
    estimates store model_results

  // calculate analysis results
    matrix coefficients = e(b)
    matrix std_errors = e(V)
    matrix p_values = e(p)

    return scalar coef = coefficients[1, 1]
    return scalar std_err = std_errors[1, 1]
    return scalar p_value = p_values[1, 1]
end
```

Let's test that our new `single_run()` function performs as expected.

```
// run one model with default parameters
quietly single_run 25 15 15 60 5 7 8
return list
```

```
scalars:
            r(p_value) =  6.15504528649e-25
            r(std_err) =  .3467247369073159
               r(coef) =  6.072637593587305
```

```
// run one model with new parameters
quietly single_run 25 10 50 60 2 7 8
return list
```

```
scalars:
            r(p_value) =  .0005213058910955
            r(std_err) =  .3180056524570669
               r(coef) =  1.956555587768618
```

Power calculation automated

To get an accurate estimation of power, we need to run the simulation many times. Here we use a matrix `results` to store the analysis results of each run.

We can finally calculate power for our parameter of interest `beta_1` by filtering to keep only that term and the calculating the proportion of times the p-value is below the `alpha` threshold.

```
quietly {
  clear
  matrix results = J($reps, 3, .)
  forval i = 1/$reps {
    quietly single_run 25 15 15 60 5 7 8
    matrix results[`i', 1] = r(coef)
    matrix results[`i', 2] = r(std_err)
    matrix results[`i', 3] = r(p_value)
  }

  clear
  svmat results, names(x)

  // calculate mean estimates and power for specified alpha
  gen power = 0
  replace power = 1 if x3 < $alpha

  egen coef_mean = mean(x1)
  egen std_err_mean = mean(x2)
  egen power_mean = mean(power)
}

di "Coef. Mean: " coef_mean
di "Std.Err. Mean: " std_err_mean
di "Power Mean: " power_mean
```

```
Coef. Mean: 5.1327767

Std.Err. Mean: .34294486

Power Mean: 1
```

Check false positive rate

We can do a sanity check to see if our simulation is performing as expected by checking the false positive rate (Type I error rate). We set the effect of `genre_ij` (`beta_1`) to 0 to calculate the false positive rate, which is the probability of concluding there is an effect when there is no actual effect in the population.

```
// run simulations and calculate the false positive rate
quietly {
  clear
  matrix results = J($reps, 3, .)
  forval i = 1/$reps {
    quietly single_run 25 15 15 60 0 7 8
    matrix results[`i', 1] = r(coef)
    matrix results[`i', 2] = r(std_err)
    matrix results[`i', 3] = r(p_value)
  }

  clear
  svmat results, names(x)

  // calculate power for specified alpha
  gen power = 0
  replace power = 1 if x3 < $alpha

  egen power_mean = mean(power)
}

di "Power Mean: " power_mean
```

```
Power Mean: .03333334
```

Ideally, the false positive rate will be equal to `alpha`, which we set at 0.05.

Power for different effect sizes

In real life, we will not know the effect size of our quantity of interest and so we will need to repeatedly perform the power analysis over a range of different plausible effect sizes. Perhaps we might also want to calculate power as we vary other data-generating parameters, such as the number of pop and rock songs sampled and the number of subjects sampled. We can create a table that combines all combinations of the parameters we want to vary in a grid.

```
// grid of parameter values of interest
quietly matrix define params = (10, 10, 10, 1 \ 10, 10, 10, 2 \ 10, 10, 10, 3 \ 10, 10, 10, 4 \ 10, 10,
\ 10, 10, 40, 1 \ 10, 10, 40, 2 \ 10, 10, 40, 3 \ 10, 10, 40, 4 \ 10, 10, 40, 5 ///
\ 10, 40, 10, 1 \ 10, 40, 10, 2 \ 10, 40, 10, 3 \ 10, 40, 10, 4 \ 10, 40, 10, 5 ///
```

```
\ 10, 40, 40, 1 \ 10, 40, 40, 2 \ 10, 40, 40, 3 \ 10, 40, 40, 4 \ 10, 40, 40, 5 ///
\ 25, 10, 10, 1 \ 25, 10, 10, 2 \ 25, 10, 10, 3 \ 25, 10, 10, 4 \ 25, 10, 10, 5 ///
\ 25, 10, 40, 1 \ 25, 10, 40, 2 \ 25, 10, 40, 3 \ 25, 10, 40, 4 \ 25, 10, 40, 5 ///
\ 25, 40, 10, 1 \ 25, 40, 10, 2 \ 25, 40, 10, 3 \ 25, 40, 10, 4 \ 25, 40, 10, 5 ///
\ 25, 40, 40, 1 \ 25, 40, 40, 2 \ 25, 40, 40, 3 \ 25, 40, 40, 4 \ 25, 40, 40, 5 ///
\ 50, 10, 10, 1 \ 50, 10, 10, 2 \ 50, 10, 10, 3 \ 50, 10, 10, 4 \ 50, 10, 10, 5 ///
\ 50, 10, 40, 1 \ 50, 10, 40, 2 \ 50, 10, 40, 3 \ 50, 10, 40, 4 \ 50, 10, 40, 5 ///
\ 50, 40, 10, 1 \ 50, 40, 10, 2 \ 50, 40, 10, 3 \ 50, 40, 10, 4 \ 50, 40, 10, 5 ///
\ 50, 40, 40, 1 \ 50, 40, 40, 2 \ 50, 40, 40, 3 \ 50, 40, 40, 4 \ 50, 40, 40, 5)
```

We can now wrap our `single_run()` function within a more general function `parameter_search()` that takes the grid of parameter values as input and uses a matrix `results` to store analysis results of each `single_run()`.

```
capture program drop parameter_search
program define parameter_search, rclass
    args params

    local rows = rowsof(params)
    matrix results = J(`rows', 7, .)

    forval i = 1/`rows' {
        local n_subj = params[`i', 1]
        local n_pop = params[`i', 2]
        local n_rock = params[`i', 3]
        local beta_1 = params[`i', 4]

        single_run `n_subj' `n_pop' `n_rock' 60 `beta_1' 7 8
        matrix results[`i', 1] = `n_subj'
        matrix results[`i', 2] = `n_pop'
        matrix results[`i', 3] = `n_rock'
        matrix results[`i', 4] = `beta_1'
        matrix results[`i', 5] = r(coef)
        matrix results[`i', 6] = r(std_err)
        matrix results[`i', 7] = r(p_value)
    }

    return matrix RE results
end
```

If we call `parameter_search()` it will return a single replication of simulations for each combination of parameter values in `params`.

```
quietly parameter_search params
return list
matrix list r(RE)
```

```
matrices:
                  r(RE) :  60 x 7


r(RE)[60,7]
             c1          c2          c3          c4          c5          c6
```

r1	10	10	10	1	2.0767703	1.3290591
r2	10	10	10	2	.62064529	1.1411167
r3	10	10	10	3	3.2051462	1.3930413
r4	10	10	10	4	2.5571848	1.3892114
r5	10	10	10	5	6.6856051	1.4763299
r6	10	10	40	1	1.5987428	.83724974
r7	10	10	40	2	2.6229231	.79420778
r8	10	10	40	3	3.1869321	.7857458
r9	10	10	40	4	3.6704681	.7473238
r10	10	10	40	5	5.4704365	.77949362
r11	10	40	10	1	2.2633134	.89868119
r12	10	40	10	2	2.6807894	.84989613
r13	10	40	10	3	4.5684848	.78929499
r14	10	40	10	4	4.0980586	.82437518
r15	10	40	10	5	4.7150011	.74640408
r16	10	40	40	1	.81983679	.35818376
r17	10	40	40	2	1.5840674	.32607948
r18	10	40	40	3	3.1854328	.32937222
r19	10	40	40	4	4.519555	.30852233
r20	10	40	40	5	5.364586	.33542361
r21	25	10	10	1	1.3821919	.50011368
r22	25	10	10	2	.90685745	.48654647
r23	25	10	10	3	2.2015396	.54055836
r24	25	10	10	4	4.0424014	.51098122
r25	25	10	10	5	4.7957371	.50273713
r26	25	10	40	1	1.2478584	.33050679
r27	25	10	40	2	2.0108924	.30820396
r28	25	10	40	3	2.6370676	.31890292
r29	25	10	40	4	4.8110402	.3149082
r30	25	10	40	5	4.8646795	.32617716
r31	25	40	10	1	.62202797	.30721533
r32	25	40	10	2	2.4312455	.33297659
r33	25	40	10	3	3.0815587	.31753019
r34	25	40	10	4	4.358412	.32801022
r35	25	40	10	5	4.7545508	.31512113
r36	25	40	40	1	1.1845128	.12694719
r37	25	40	40	2	1.9192859	.12666322
r38	25	40	40	3	3.1466376	.12789099
r39	25	40	40	4	4.011289	.12794187
r40	25	40	40	5	4.6528025	.13191323
r41	50	10	10	1	1.0968133	.25114203
r42	50	10	10	2	1.8459916	.24536956
r43	50	10	10	3	2.5134481	.27698685
r44	50	10	10	4	3.9934688	.26265935
r45	50	10	10	5	5.550354	.27162128
r46	50	10	40	1	.8059052	.15170351
r47	50	10	40	2	2.1912051	.16397648
r48	50	10	40	3	2.8441802	.15442357
r49	50	10	40	4	3.7818822	.15977438
r50	50	10	40	5	5.0424808	.15956679
r51	50	40	10	1	-.06306	.15992571
r52	50	40	10	2	2.0112829	.15796861
r53	50	40	10	3	3.1048406	.16241303
r54	50	40	10	4	4.4309891	.15874512

r55	50	40	10	5	4.697969	.16744383
r56	50	40	40	1	1.0014537	.06765406
r57	50	40	40	2	2.0857039	.06470816
r58	50	40	40	3	2.6670335	.06178508
r59	50	40	40	4	3.8384164	.06603957
r60	50	40	40	5	4.8219271	.06276692

	c7
r1	.07163583
r2	.56123837
r3	.00661557
r4	.03003782
r5	3.747e-08
r6	.08059674
r7	.00324848
r8	.00032405
r9	.00002177
r10	5.789e-10
r11	.01696379
r12	.00363862
r13	2.715e-07
r14	6.376e-06
r15	4.828e-08
r16	.17073231
r17	.00553657
r18	2.850e-08
r19	4.059e-16
r20	1.993e-20
r21	.05064301
r22	.19356659
r23	.00275014
r24	1.558e-08
r25	1.345e-11
r26	.0299632
r27	.00029213
r28	3.016e-06
r29	1.006e-17
r30	1.626e-17
r31	.26175742
r32	.00002517
r33	4.535e-08
r34	2.741e-14
r35	2.459e-17
r36	.00088573
r37	6.937e-08
r38	1.382e-18
r39	3.464e-29
r40	1.430e-37
r41	.0286235
r42	.00019404
r43	1.791e-06
r44	6.591e-15
r45	1.749e-26
r46	.03853462

```
r47  6.261e-08
r48  4.564e-13
r49  3.039e-21
r50  1.571e-36
r51  .87470376
r52  4.183e-07
r53  1.316e-14
r54  9.897e-29
r55  1.646e-30
r56  .00011802
r57  2.419e-16
r58  7.385e-27
r59  1.906e-50
r60  1.506e-82
```

Then we just repeatedly call `parameter_search()` for the number of times specified by `reps` and store the result in a matrix `final_results`. Fair warning, this will take some time if you have set a high number of replications!

```
// replicate the parameter search many times
quietly {
  clear
  matrix final_results = J(1, 7, .)
  forval i = 1/$reps {
    quietly parameter_search params

    matrix final_results = final_results \ r(RE)
  }

  // rename the columns
  clear
  svmat final_results, names(final_results)
  rename final_results1 n_subj
  rename final_results2 n_pop
  rename final_results3 n_rock
  rename final_results4 beta_1
  rename final_results5 mean_estimate
  rename final_results6 mean_se
  rename final_results7 p_value
  drop in 1

  save "./data/final_results.dta", replace
}
```

Now, as before, we can calculate power. But this time we'll group by all of the parameters we manipulated in `pgrid`, so that we can get power estimates for all combinations of parameter values.

```
quietly {
  use "./data/final_results.dta"

  gen power = 0
  replace power = 1 if p_value < $alpha
```

```
  drop p_value

  collapse (mean) mean_estimate mean_se power, by(n_subj n_pop n_rock beta_1)

  save "./data/sims_table.dta", replace
}
list
```

```
     | n_subj   n_pop   n_rock   beta_1    mean_e~e     mean_se      power |
     |--------------------------------------------------------------------|
  1. |     10      10       10        1    1.268973    1.238123   .2333333 |
  2. |     10      10       10        2    1.889742    1.235582         .4 |
  3. |     10      10       10        3    2.683959    1.239035   .6333333 |
  4. |     10      10       10        4    4.400897    1.289458          1 |
  5. |     10      10       10        5    5.275087    1.243694          1 |
     |--------------------------------------------------------------------|
  6. |     10      10       40        1    1.014274    .7960495   .1666667 |
  7. |     10      10       40        2    1.988628    .8059257         .6 |
  8. |     10      10       40        3     2.97399    .8001856          1 |
  9. |     10      10       40        4    3.996796    .7847906          1 |
 10. |     10      10       40        5    4.943249    .8100024          1 |
     |--------------------------------------------------------------------|
 11. |     10      40       10        1    1.049238    .7958079         .2 |
 12. |     10      40       10        2    1.768396    .8269665   .5333334 |
 13. |     10      40       10        3    3.041028    .7862483   .8666667 |
 14. |     10      40       10        4    4.262085    .8037483          1 |
 15. |     10      40       10        5    5.404449    .7965609          1 |
     |--------------------------------------------------------------------|
 16. |     10      40       40        1    .9648004    .3186095         .4 |
 17. |     10      40       40        2    2.164498      .31894          1 |
 18. |     10      40       40        3    2.994144    .3153042          1 |
 19. |     10      40       40        4    4.117634    .3245635          1 |
 20. |     10      40       40        5    4.947803    .3222156          1 |
     |--------------------------------------------------------------------|
 21. |     25      10       10        1    1.009122    .5089009   .2666667 |
 22. |     25      10       10        2     2.01162    .5161048   .7666667 |
 23. |     25      10       10        3    2.849991    .5214418          1 |
 24. |     25      10       10        4    4.013868     .510223          1 |
 25. |     25      10       10        5    5.058398    .5018378          1 |
     |--------------------------------------------------------------------|
 26. |     25      10       40        1    1.091378    .3221848   .5333334 |
 27. |     25      10       40        2    1.758242    .3150428   .8333333 |
 28. |     25      10       40        3    2.941761    .3189749          1 |
 29. |     25      10       40        4    4.100368    .3209341          1 |
 30. |     25      10       40        5    4.905522    .3202905          1 |
     |--------------------------------------------------------------------|
 31. |     25      40       10        1    .9354665    .3251629   .3333333 |
 32. |     25      40       10        2    2.089087    .3190893   .9666666 |
 33. |     25      40       10        3    3.008885    .3194447          1 |
 34. |     25      40       10        4    4.040331    .3203337          1 |
 35. |     25      40       10        5    4.991682    .3226803          1 |
     |--------------------------------------------------------------------|
 36. |     25      40       40        1    1.077446    .1273965   .8666667 |
 37. |     25      40       40        2     1.97645    .1276043          1 |
```

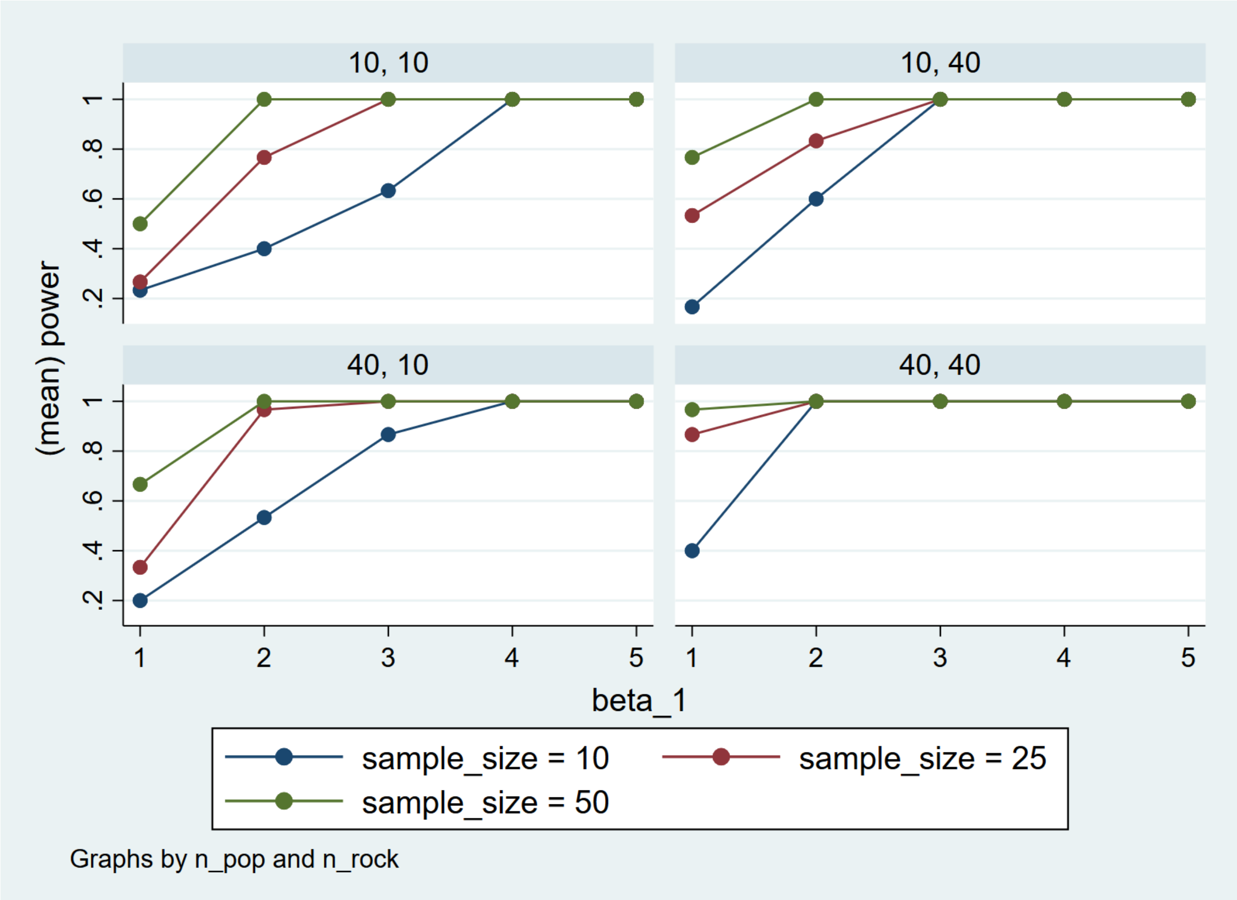
```
38. |     25      40       40        3   2.928479   .1292938          1 |
39. |     25      40       40        4   4.038676   .1287591          1 |
40. |     25      40       40        5   5.002239   .1278136          1 |
    |-----------------------------------------------------------------|
41. |     50      10       10        1   .9126882   .2530858         .5 |
42. |     50      10       10        2    2.00245   .2595248          1 |
43. |     50      10       10        3    3.04136    .258838          1 |
44. |     50      10       10        4   3.974287   .2553087          1 |
45. |     50      10       10        5   5.205029   .2573906          1 |
    |-----------------------------------------------------------------|
46. |     50      10       40        1   1.030129   .1595967   .7666667 |
47. |     50      10       40        2   1.991586   .1598501          1 |
48. |     50      10       40        3   3.040545   .1619374          1 |
49. |     50      10       40        4   3.960105    .159726          1 |
50. |     50      10       40        5   4.958652    .159858          1 |
    |-----------------------------------------------------------------|
51. |     50      40       10        1   .9377397   .1599206   .6666667 |
52. |     50      40       10        2   2.048419   .1587613          1 |
53. |     50      40       10        3   2.945178   .1593941          1 |
54. |     50      40       10        4    4.02575     .15889          1 |
55. |     50      40       10        5   4.963776   .1594315          1 |
    |-----------------------------------------------------------------|
56. |     50      40       40        1    1.04931    .064062   .9666666 |
57. |     50      40       40        2   2.004742   .0637963          1 |
58. |     50      40       40        3   2.927157   .0643362          1 |
59. |     50      40       40        4    3.90757   .0640075          1 |
60. |     50      40       40        5   5.028903   .0638838          1 |
    +-----------------------------------------------------------------+
```

Here's a graph that visualizes the output of the power simulation.

```
quietly {
  use "./data/sims_table.dta"

  twoway (connected power beta_1 if n_subj == 10, sort) (connected power beta_1 if n_subj == 25, sort)

  graph export "./figures/twoway.png", replace
}
```


10, 10
10, 40
40, 10
40, 40
(mean) power
.2
.4
.6
.8
1
1
2
3
4
5
beta_1
sample_size = 10
sample_size = 25
sample_size = 50
Graphs by n_pop and n_rock