DPLYR Bisection–Evaluate Many Unknown Nonlinear Equations Jointly, Solve Roots for Strictly Monotonic Functions with Single Zero-Crossing

Go back to fan's R4Econ Repository or Intro Stats with R Repository.

Issue and Goal

We want evaluate linear function $0 = f(z_{ij}, x_i, y_i, \mathbf{X}, \mathbf{Y}, c, d)$. There are i functions that have i specific x and y. For each i function, we evaluate along a grid of feasible values for z, over $j \in J$ grid points, potentially looking for the j that is closest to the root. $\mathbf{X}$ and $\mathbf{Y}$ are arrays common across the i equations, and c and d are constants.

The evaluation strategy is the following, given min and max for z that are specific for each j, and given common number of grid points, generate a matrix of z_{ij}. Suppose there the number of i is I, and the number of grid points for j is J.

1. Generate a $J \cdot I$ by 3 matrix where the columns are z, x, y as tibble
2. Follow this Mutate to evaluate the $f(\cdot)$ function.
3. Add two categorical columns for grid levels and wich i, i and j index. Plot Mutate output evaluated column categorized by i as color and j as x-axis.

Set Up

```r
rm(list = ls(all.names = TRUE))
options(knitr.duplicate.label = 'allow')
```

```r
library(tidyverse)
library(tidyr)
library(knitr)
library(kableExtra)
# file name
st_file_name = 'fs_func_graph_eval'
# Generate R File
purl(paste0(st_file_name, ".Rmd"), output=paste0(st_file_name, ".R"), documentation = 2)
# Generate PDF and HTML
# rmarkdown::render("C:/Users/fan/R4Econ/support/function/fs_funceval.Rmd", "pdf_document")
# rmarkdown::render("C:/Users/fan/R4Econ/support/function/fs_funceval.Rmd", "html_document")
```

Set up Input Arrays

There is a function that takes $M = Q + P$ inputs, we want to evaluate this function N times. Each time, there are M inputs, where all but Q of the M inputs, meaning P of the M inputs, are the same. In particular, $P = Q * N$.

$$M = Q + P = Q + Q * N$$

Now we need to expand this by the number of choice grid. Each row, representing one equation, is expanded by the number of choice grids. We are graphically searching, or rather brute force searching, which means if we have 100 individuals, we want to plot out the nonlinear equation for each of these lines, and show graphically where each line crosses zero. We achieve this, by evaluating the equation for each of the 100 individuals along a grid of feasible choices.

In this problem here, the feasible choices are shared across individuals.

```r
# Parameters
fl_rho = 0.20
svr_id_var = 'INDI_ID'

# it_child_count = N, the number of children
it_N_child_cnt = 4
# it_heter_param = Q, number of parameters that are heterogeneous across children
it_Q_hetpa_cnt = 2

# P fixed parameters, nN is N dimensional, nP is P dimensional
ar_nN_A = seq(-2, 2, length.out = it_N_child_cnt)
ar_nN_alpha = seq(0.1, 0.9, length.out = it_N_child_cnt)
ar_nP_A_alpha = c(ar_nN_A, ar_nN_alpha)

# N by Q varying parameters
mt_nN_by_nQ_A_alpha = cbind(ar_nN_A, ar_nN_alpha)

# Choice Grid for nutritional feasible choices for each
fl_N_agg = 100
fl_N_min = 0
it_N_choice_cnt_ttest = 3
it_N_choice_cnt_dense = 100
ar_N_choices_ttest = seq(fl_N_min, fl_N_agg, length.out = it_N_choice_cnt_ttest)
ar_N_choices_dense = seq(fl_N_min, fl_N_agg, length.out = it_N_choice_cnt_dense)

# Mesh Expand
tb_states_choices <- as_tibble(mt_nN_by_nQ_A_alpha) %>% rowid_to_column(var=svr_id_var)
tb_states_choices_ttest <- tb_states_choices %>% expand_grid(choices = ar_N_choices_ttest)
tb_states_choices_dense <- tb_states_choices %>% expand_grid(choices = ar_N_choices_dense)

# display
summary(tb_states_choices_dense)
```

```
##     INDI_ID        ar_nN_A    ar_nN_alpha     choices
##  Min.   :1.00   Min.   :-2   Min.   :0.1   Min.   :  0
##  1st Qu.:1.75   1st Qu.:-1   1st Qu.:0.3   1st Qu.: 25
##  Median :2.50   Median : 0   Median :0.5   Median : 50
##  Mean   :2.50   Mean   : 0   Mean   :0.5   Mean   : 50
##  3rd Qu.:3.25   3rd Qu.: 1   3rd Qu.:0.7   3rd Qu.: 75
##  Max.   :4.00   Max.   : 2   Max.   :0.9   Max.   :100
```

```r
kable(tb_states_choices_ttest) %>%
  kable_styling(bootstrap_options = c("striped", "hover", "condensed", "responsive"))
```

INDI_ID

ar_nN_A

ar_nN_alpha

```
choices
1
-2.0000000
0.1000000
0
1
-2.0000000
0.1000000
50
1
-2.0000000
0.1000000
100
2
-0.6666667
0.3666667
0
2
-0.6666667
0.3666667
50
2
-0.6666667
0.3666667
100
3
0.6666667
0.6333333
0
3
0.6666667
0.6333333
50
3
0.6666667
0.6333333
```

100

4

2.0000000

0.9000000

0

4

2.0000000

0.9000000

50

4

2.0000000

0.9000000

100

Apply Same Function all Rows, Some Inputs Row-specific, other Shared

There are two types of inputs, row-specific inputs, and inputs that should be applied for each row. The Function just requires all of these inputs, it does not know what is row-specific and what is common for all row. Dplyr recognizes which parameter inputs already existing in the piped dataframe/tibble, given rowwise, those will be row-specific inputs. Additional function parameters that do not exist in dataframe as variable names, but that are pre-defined scalars or arrays will be applied to all rows.

- @param string variable name of input where functions are evaluated, these are already contained in the dataframe, existing variable names, row specific, rowwise computation over these, each rowwise calculation using different rows: *fl_A*, *fl_alpha*, *fl_N*
- @param scalar and array values that are applied to every rowwise calculation, all rowwise calculations using the same scalars and arrays:*ar_A*, *ar_alpha*, *fl_N_agg*, *fl_rho*
- @param string output variable name

The function looks within group, finds min/max etc that are relevant.

3 Points and Denser Dataframs and Define Function

```r
# Convert Matrix to Tibble
ar_st_col_names = c(svr_id_var,'fl_A', 'fl_alpha')
tb_states_choices <- tb_states_choices %>% rename_all(~c(ar_st_col_names))
ar_st_col_names = c(svr_id_var,'fl_A', 'fl_alpha', 'fl_N')
tb_states_choices_ttest <- tb_states_choices_ttest %>% rename_all(~c(ar_st_col_names))
tb_states_choices_dense <- tb_states_choices_dense %>% rename_all(~c(ar_st_col_names))

# Define Implicit Function
ffi_nonlin_dplyrdo <- function(fl_A, fl_alpha, fl_N, ar_A, ar_alpha, fl_N_agg, fl_rho){
  # scalar value that are row-specific, in dataframe already: *fl_A*, *fl_alpha*, *fl_N*
  # array and scalars not in dataframe, common all rows: *ar_A*, *ar_alpha*, *fl_N_agg*, *fl_rho*

  # Test Parameters
  # ar_A = ar_nN_A
  # ar_alpha = ar_nN_alpha
  # fl_N = 100
  # fl_rho = -1
  # fl_N_q = 10

  # Apply Function
  ar_p1_s1 = exp((fl_A - ar_A)*fl_rho)
  ar_p1_s2 = (fl_alpha/ar_alpha)
  ar_p1_s3 = (1/(ar_alpha*fl_rho - 1))
  ar_p1 = (ar_p1_s1*ar_p1_s2)^ar_p1_s3
  ar_p2 = fl_N^((fl_alpha*fl_rho-1)/(ar_alpha*fl_rho-1))
  ar_overall = ar_p1*ar_p2
  fl_overall = fl_N_agg - sum(ar_overall)

  return(fl_overall)
}
```

Evaluate at Three Choice Points and Show Table

In the example below, just show results evaluating over three choice points and show table.

```
# fl_A, fl_alpha are from columns of tb_nN_by_nQ_A_alpha
tb_states_choices_ttest_eval = tb_states_choices_ttest %>% rowwise() %>%
                        mutate(dplyr_eval = ffi_nonlin_dplyrdo(fl_A, fl_alpha, fl_N,
                                                               ar_nN_A, ar_nN_alpha,
                                                               fl_N_agg, fl_rho))
# Show
kable(tb_states_choices_ttest_eval) %>%
  kable_styling(bootstrap_options = c("striped", "hover", "condensed", "responsive"))
```

INDI_ID

fl_A

fl_alpha

fl_N

dplyr_eval

1

-2.0000000

0.1000000

0

100.00000

1

-2.0000000

0.1000000

50

-5666.95576

1

```
-2.0000000
 0.1000000
 100
-12880.28392
 2
-0.6666667
 0.3666667
 0
 100.00000
 2
-0.6666667
 0.3666667
 50
-595.73454
 2
-0.6666667
 0.3666667
 100
-1394.70698
 3
 0.6666667
 0.6333333
 0
 100.00000
 3
 0.6666667
 0.6333333
 50
-106.51058
 3
 0.6666667
 0.6333333
 100
-323.94216
 4
 2.0000000
```

0.9000000

0

100.00000

4

2.0000000

0.9000000

50

22.55577

4

2.0000000

0.9000000

100

-51.97161

Evaluate at Many Choice Points and Show Graphically

Same as above, but now we evaluate the function over the individuals at many choice points so that we can graph things out.

```
# fl_A, fl_alpha are from columns of tb_nN_by_nQ_A_alpha
tb_states_choices_dense_eval = tb_states_choices_dense %>% rowwise() %>%
                        mutate(dplyr_eval = ffi_nonlin_dplyrdo(fl_A, fl_alpha, fl_N,
                                                               ar_nN_A, ar_nN_alpha,
                                                               fl_N_agg, fl_rho))
```

```
# Show
dim(tb_states_choices_dense_eval)
```

```
## [1] 400   5
```

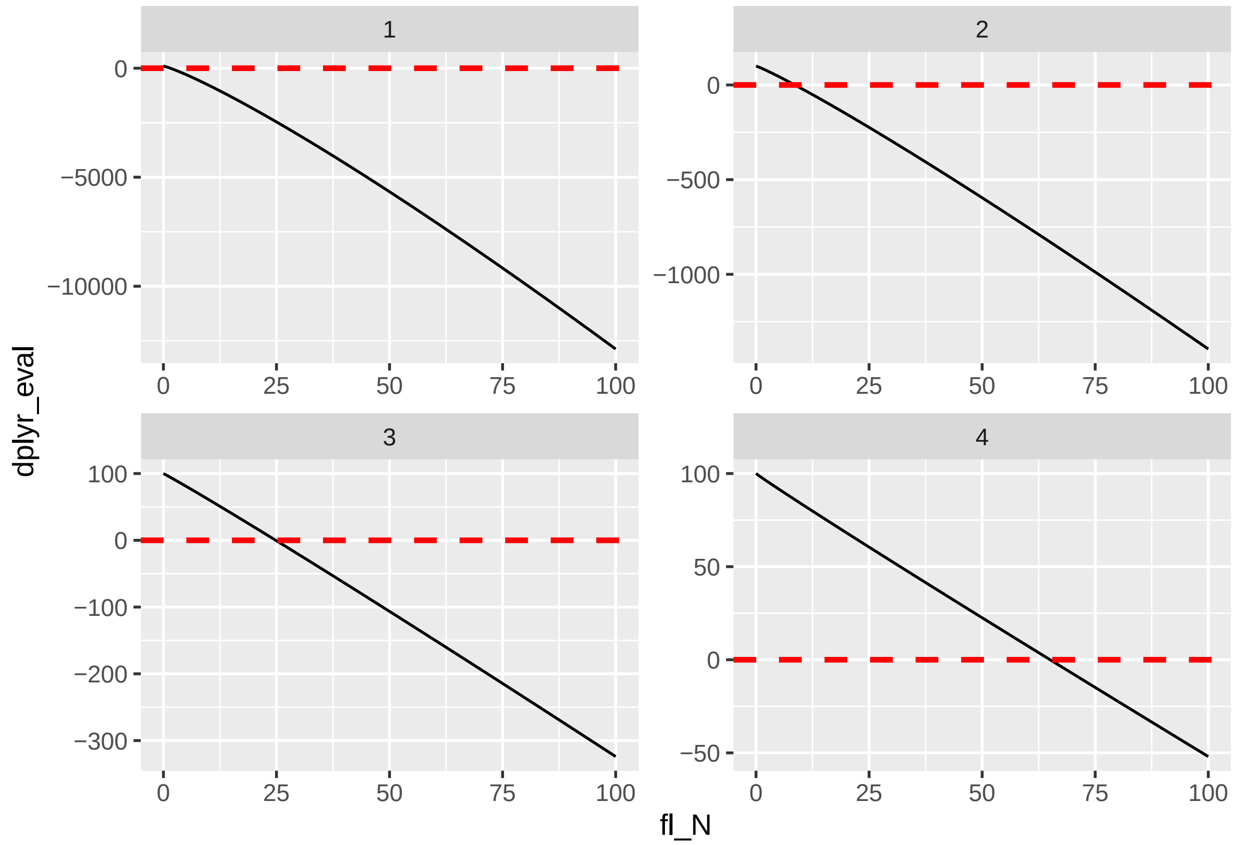
```
summary(tb_states_choices_dense_eval)
```

```
##     INDI_ID          fl_A       fl_alpha        fl_N       dplyr_eval
##  Min.   :1.00   Min.   :-2   Min.   :0.1   Min.   :  0   Min.   :-12880.28
##  1st Qu.:1.75   1st Qu.:-1   1st Qu.:0.3   1st Qu.: 25   1st Qu.: -1167.29
##  Median :2.50   Median : 0   Median :0.5   Median : 50   Median :  -202.42
##  Mean   :2.50   Mean   : 0   Mean   :0.5   Mean   : 50   Mean   : -1645.65
##  3rd Qu.:3.25   3rd Qu.: 1   3rd Qu.:0.7   3rd Qu.: 75   3rd Qu.:     0.96
##  Max.   :4.00   Max.   : 2   Max.   :0.9   Max.   :100   Max.   :   100.00
```

```
lineplot <- tb_states_choices_dense_eval %>%
    ggplot(aes(x=fl_N, y=dplyr_eval)) +
        geom_line() +
        facet_wrap( . ~ INDI_ID, scales = "free") +
        geom_hline(yintercept=0, linetype="dashed",
                color = "red", size=1) +
        labs(title = 'Evaluate Non-Linear Functions to Search for Roots',
             x = 'X values',
             y = 'f(x)',
             caption = 'Evaluating the Function')
```

```
## $x
## [1] "X values"
##
## $y
## [1] "f(x)"
##
## $title
## [1] "Evaluate Non-Linear Functions to Search for Roots"
##
## $caption
## [1] "Evaluating the Function"
##
## attr(,"class")
## [1] "labels"
```

```r
print(lineplot)
```



Bisection Solve Optimal Choice for Each Individual

The bisection specific code does not need to do much.

- @param list variables in file for grouping, each group is an individual for whom we want to calculate optimal choice for using bisection.
- @param string variable name of input where functions are evaluated, these are already contained in the dataframe, existing variable names, row specific, rowwise computation over these, each rowwise calculation using different rows.

- @param scalar and array values that are applied to every rowwise calculation, all rowwise calculations using the same scalars and arrays.
- @param string output variable name

Bisection Algorithm

This is how I implement the bisection algorithm, when we know the bounding minimum and maximum to be below and above zero already.

1. Evaluate $f^0_a = f(a^0)$ and $f^0_b = f(b^0)$, min and max points.
2. Evaluate at $f^0_p = f(p^0)$, where $p_0 = \frac{a^0+b^0}{2}$.
3. if $f^i_a \cdot f^i_p < 0$, then $b_{i+1} = p_i$, else, $a_{i+1} = p_i$ and $f^{i+1}_a = p_i$.
4. iteratre until convergence.

DPLYR Implementation of Bisection

Generate New columns of a and b as we iteratre, do not need to store p, p is temporary. Evaluate the function below which we have already tested, but now, in the dataframe before generating all permutations, *tb_states_choices*, now the *fl_N* element will be changing with each iteration, it will be row specific. *fl_N* are first min and max, then each subsequent ps.

Initialize Matrix

First, initialize the matrix with a_0 and b_0, the initial min and max points:

```
# common prefix to make reshaping easier
st_bisec_prefix <- 'bisec_'
svr_a_lst <- paste0(st_bisec_prefix, 'a_0')
svr_b_lst <- paste0(st_bisec_prefix, 'b_0')
svr_fa_lst <- paste0(st_bisec_prefix, 'fa_0')
svr_fb_lst <- paste0(st_bisec_prefix, 'fb_0')

# Add initial a and b
tb_states_choices_bisec <- tb_states_choices %>%
                           mutate(!!sym(svr_a_lst) := fl_N_min, !!sym(svr_b_lst) := fl_N_agg)

# Evaluate function f(a_0) and f(b_0)
tb_states_choices_bisec <- tb_states_choices_bisec %>% rowwise() %>%
                           mutate(!!sym(svr_fa_lst) := ffi_nonlin_dplyrdo(fl_A, fl_alpha, !!sym(svr_a_
                                                                          ar_nN_A, ar_nN_alpha,
                                                                          fl_N_agg, fl_rho),
                                  !!sym(svr_fb_lst) := ffi_nonlin_dplyrdo(fl_A, fl_alpha, !!sym(svr_b_
                                                                          ar_nN_A, ar_nN_alpha,
                                                                          fl_N_agg, fl_rho))
# Summarize
dim(tb_states_choices_bisec)
```

```
## [1] 4 7
```

```
summary(tb_states_choices_bisec)
```

```
##     INDI_ID          fl_A       fl_alpha     bisec_a_0   bisec_b_0
##  Min.   :1.00   Min.   :-2   Min.   :0.1   Min.   :0   Min.   :100
##  1st Qu.:1.75   1st Qu.:-1   1st Qu.:0.3   1st Qu.:0   1st Qu.:100
##  Median :2.50   Median : 0   Median :0.5   Median :0   Median :100
##  Mean   :2.50   Mean   : 0   Mean   :0.5   Mean   :0   Mean   :100
##  3rd Qu.:3.25   3rd Qu.: 1   3rd Qu.:0.7   3rd Qu.:0   3rd Qu.:100
##  Max.   :4.00   Max.   : 2   Max.   :0.9   Max.   :0   Max.   :100
##    bisec_fa_0    bisec_fb_0
##  Min.   :100   Min.   :-12880.28
##  1st Qu.:100   1st Qu.: -4266.10
##  Median :100   Median :  -859.33
##  Mean   :100   Mean   : -3662.73
##  3rd Qu.:100   3rd Qu.:  -255.95
##  Max.   :100   Max.   :   -51.97
```

Iterate and Solve for f(p), update f(a) and f(b)

Implement the DPLYR based Concurrent bisection algorithm.

```
# fl_tol = float tolerance criteria
# it_tol = number of interations to allow at most
fl_tol <- 10^-2
it_tol <- 100

# fl_p_dist2zr = distance to zero to initalize
fl_p_dist2zr <- 1000
it_cur <- 0
while (it_cur <= it_tol && fl_p_dist2zr >= fl_tol ) {

  it_cur <- it_cur + 1

  # New Variables
  svr_a_cur <- paste0(st_bisec_prefix, 'a_', it_cur)
  svr_b_cur <- paste0(st_bisec_prefix, 'b_', it_cur)
  svr_fa_cur <- paste0(st_bisec_prefix, 'fa_', it_cur)
  svr_fb_cur <- paste0(st_bisec_prefix, 'fb_', it_cur)

  # Evaluate function f(a_0) and f(b_0)
  # 1. generate p
  # 2. generate f_p
  # 3. generate f_p*f_a
  tb_states_choices_bisec <- tb_states_choices_bisec %>% rowwise() %>%
                             mutate(p = ((!!sym(svr_a_lst) + !!sym(svr_b_lst))/2)) %>%
                             mutate(f_p = ffi_nonlin_dplyrdo(fl_A, fl_alpha, p,
                                                             ar_nN_A, ar_nN_alpha,
                                                             fl_N_agg, fl_rho)) %>%
                             mutate(f_p_t_f_a = f_p*!!sym(svr_fa_lst))
  # fl_p_dist2zr = sum(abs(p))
  fl_p_dist2zr <- mean(abs(tb_states_choices_bisec %>% pull(f_p)))

  # Update a and b
  tb_states_choices_bisec <- tb_states_choices_bisec %>%
                             mutate(!!sym(svr_a_cur) :=
                                      case_when(f_p_t_f_a < 0 ~ !!sym(svr_a_lst),
                                                TRUE ~ p)) %>%
                             mutate(!!sym(svr_b_cur) :=
                                      case_when(f_p_t_f_a < 0 ~ p,
                                                TRUE ~ !!sym(svr_b_lst)))
  # Update f(a) and f(b)
  tb_states_choices_bisec <- tb_states_choices_bisec %>%
                              mutate(!!sym(svr_fa_cur) :=
                                       case_when(f_p_t_f_a < 0 ~ !!sym(svr_fa_lst),
                                                 TRUE ~ f_p)) %>%
                              mutate(!!sym(svr_fb_cur) :=
                                       case_when(f_p_t_f_a < 0 ~ f_p,
                                                 TRUE ~ !!sym(svr_fb_lst)))
  # Save from last
  svr_a_lst <- svr_a_cur
  svr_b_lst <- svr_b_cur
  svr_fa_lst <- svr_fa_cur
  svr_fb_lst <- svr_fb_cur

  # Summar current round
  print(paste0('it_cur:', it_cur, ', fl_p_dist2zr:', fl_p_dist2zr))
  summary(tb_states_choices_bisec %>% select(one_of(svr_a_cur, svr_b_cur, svr_fa_cur, svr_fb_cur)))
}
```

```
## [1] "it_cur:1, fl_p_dist2zr:1597.93916362849"
## [1] "it_cur:2, fl_p_dist2zr:676.06602535902"
## [1] "it_cur:3, fl_p_dist2zr:286.850590132782"
## [1] "it_cur:4, fl_p_dist2zr:117.225493866655"
## [1] "it_cur:5, fl_p_dist2zr:37.570593471664"
## [1] "it_cur:6, fl_p_dist2zr:4.60826664896022"
## [1] "it_cur:7, fl_p_dist2zr:14.4217689135683"
## [1] "it_cur:8, fl_p_dist2zr:8.38950830086659"
## [1] "it_cur:9, fl_p_dist2zr:3.93347761455868"
## [1] "it_cur:10, fl_p_dist2zr:1.88261338941038"
## [1] "it_cur:11, fl_p_dist2zr:0.744478952222305"
## [1] "it_cur:12, fl_p_dist2zr:0.187061801237917"
## [1] "it_cur:13, fl_p_dist2zr:0.117844913432613"
## [1] "it_cur:14, fl_p_dist2zr:0.0275365951418891"
## [1] "it_cur:15, fl_p_dist2zr:0.0515488156908255"
## [1] "it_cur:16, fl_p_dist2zr:0.0191152349149135"
## [1] "it_cur:17, fl_p_dist2zr:0.00385372194545752"
```

Reshape Wide to long to Wide

To view results easily, how iterations improved to help us find the roots, convert table from wide to long. Pivot twice. This allows us to easily graph out how bisection is working out iterationby iteration.

```
# New variables
svr_bisect_iter <- 'biseciter'
svr_abfafb_long_name <- 'varname'
svr_number_col <- 'value'
svr_id_bisect_iter <- paste0(svr_id_var, '_bisect_ier')

# Pivot wide to very long
tb_states_choices_bisec_long <- tb_states_choices_bisec %>%
  pivot_longer(
    cols = starts_with(st_bisec_prefix),
    names_to = c(svr_abfafb_long_name, svr_bisect_iter),
    names_pattern = paste0(st_bisec_prefix, "(.*)_(.*)"),
    values_to = svr_number_col
  )
```

```r
# # Generate ID column based on INDI ID and ITER
# tb_states_choices_bisec_long <- tb_states_choices_bisec_long %>%
#   mutate(!!sym(svr_id_bisect_iter) := paste0(!!sym(svr_id_var), '_', !!sym(svr_bisect_iter)))

# Pivot wide to very long to a little wide
tb_states_choices_bisec_wider <- tb_states_choices_bisec_long %>%
  pivot_wider(
    names_from = !!sym(svr_abfafb_long_name),
    values_from = svr_number_col
  )

# Print
summary(tb_states_choices_bisec_wider)
```

```
##     INDI_ID          fl_A       fl_alpha         p         
##  Min.   :1.00   Min.   :-2   Min.   :0.1   Min.   : 1.545  
##  1st Qu.:1.75   1st Qu.:-1   1st Qu.:0.3   1st Qu.: 6.824  
##  Median :2.50   Median : 0   Median :0.5   Median :16.710  
##  Mean   :2.50   Mean   : 0   Mean   :0.5   Mean   :25.000  
##  3rd Qu.:3.25   3rd Qu.: 1   3rd Qu.:0.7   3rd Qu.:34.886  
##  Max.   :4.00   Max.   : 2   Max.   :0.9   Max.   :65.037  
##       f_p                 f_p_t_f_a           biseciter               a         
##  Min.   :-0.0076372   Min.   :-3.800e-04   Length:72          Min.   : 0.000  
##  1st Qu.:-0.0058251   1st Qu.:-1.128e-04   Class :character   1st Qu.: 1.440  
##  Median :-0.0034186   Median :-1.313e-05   Mode  :character   Median : 8.582  
##  Mean   :-0.0038537   Mean   :-1.016e-04                      Mean   :21.442  
##  3rd Qu.:-0.0014473   3rd Qu.:-1.945e-06                      3rd Qu.:24.835  
##  Max.   :-0.0009405   Max.   :-1.884e-07                      Max.   :65.036  
##        b                  fa                  fb          
##  Min.   :  1.545   Min.   :  0.00020   Min.   :-12880.284  
##  1st Qu.:  8.592   1st Qu.:  0.04976   1st Qu.:   -10.180  
##  Median : 24.854   Median :  1.89887   Median :    -0.686  
##  Mean   : 32.553   Mean   : 25.69212   Mean   :  -354.393  
##  3rd Qu.: 65.039   3rd Qu.: 29.42716   3rd Qu.:    -0.047  
##  Max.   :100.000   Max.   :100.00000   Max.   :    -0.001
```

```r
head(tb_states_choices_bisec_wider %>% select(-one_of('p','f_p','f_p_t_f_a')), 30)
```

```
## # A tibble: 30 x 8
##    INDI_ID  fl_A fl_alpha biseciter     a      b    fa        fb
##      <int> <dbl>    <dbl> <chr>     <dbl>  <dbl> <dbl>     <dbl>
##  1       1    -2      0.1 0         0     100    100   -12880. 
##  2       1    -2      0.1 1         0      50    100    -5667. 
##  3       1    -2      0.1 2         0      25    100    -2465. 
##  4       1    -2      0.1 3         0      12.5  100    -1042. 
##  5       1    -2      0.1 4         0       6.25 100     -409. 
##  6       1    -2      0.1 5         0       3.12 100     -127. 
##  7       1    -2      0.1 6         0       1.56 100       -1.33
##  8       1    -2      0.1 7         0.781   1.56  54.7     -1.33
##  9       1    -2      0.1 8         1.17    1.56  27.5     -1.33
## 10       1    -2      0.1 9         1.37    1.56  13.2     -1.33
## # ... with 20 more rows
```

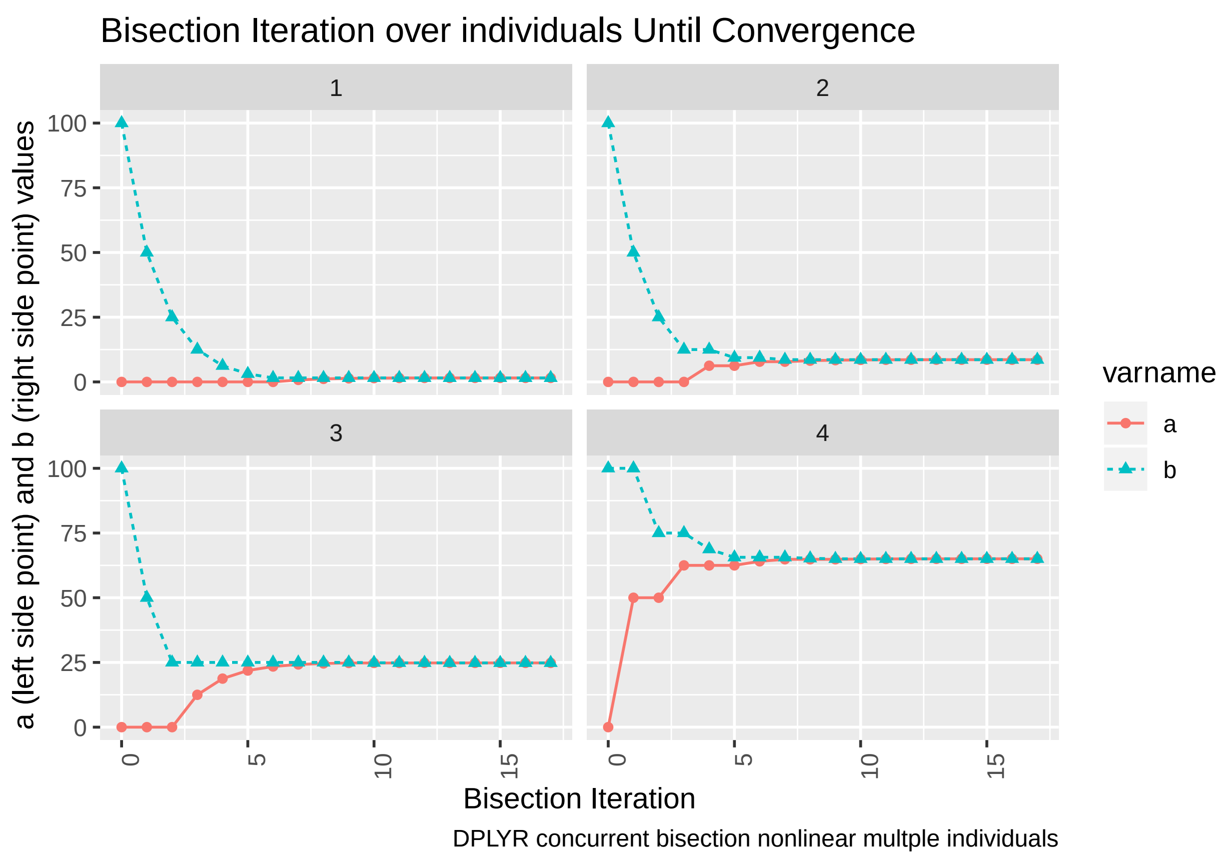
```r
tail(tb_states_choices_bisec_wider %>% select(-one_of('p','f_p','f_p_t_f_a')), 30)
```

```
## # A tibble: 30 x 8
##    INDI_ID  fl_A fl_alpha biseciter     a     b      fa      fb
##      <int> <dbl>    <dbl> <chr>     <dbl> <dbl>   <dbl>   <dbl>
##  1       3 0.667    0.633 6          23.4  25   5.82    -0.686
##  2       3 0.667    0.633 7          24.2  25   2.57    -0.686
##  3       3 0.667    0.633 8          24.6  25   0.943   -0.686
##  4       3 0.667    0.633 9          24.8  25   0.129   -0.686
##  5       3 0.667    0.633 10         24.8  24.9 0.129   -0.278
##  6       3 0.667    0.633 11         24.8  24.9 0.129   -0.0748
##  7       3 0.667    0.633 12         24.8  24.9 0.0270  -0.0748
##  8       3 0.667    0.633 13         24.8  24.8 0.0270  -0.0239
##  9       3 0.667    0.633 14         24.8  24.8 0.00157 -0.0239
## 10       3 0.667    0.633 15         24.8  24.8 0.00157 -0.0112
## # ... with 20 more rows
```

Graph Bisection Iteration Results

Actually we want to graph based on the long results, not the wider. Wider easier to view in table.

```
# Graph results
lineplot <- tb_states_choices_bisec_long %>%
    mutate(!!sym(svr_bisect_iter) := as.numeric(!!sym(svr_bisect_iter))) %>%
    filter(!!sym(svr_abfafb_long_name) %in% c('a', 'b')) %>%
    ggplot(aes(x=!!sym(svr_bisect_iter), y=!!sym(svr_number_col),
               colour=!!sym(svr_abfafb_long_name),
               linetype=!!sym(svr_abfafb_long_name),
               shape=!!sym(svr_abfafb_long_name))) +
        facet_wrap( ~ INDI_ID) +
        geom_line() +
        geom_point() +
        labs(title = 'Bisection Iteration over individuals Until Convergence',
             x = 'Bisection Iteration',
             y = 'a (left side point) and b (right side point) values',
             caption = 'DPLYR concurrent bisection nonlinear multple individuals') +
      theme(axis.text.x = element_text(angle = 90, hjust = 1))
print(lineplot)
```


Bisection Iteration over individuals Until Convergence
1
2
3
4
a (left side point) and b (right side point) values
100
75
50
25
0
0
5
10
15
Bisection Iteration
varname
a
b
DPLYR concurrent bisection nonlinear multple individuals