Writing behavioral feedback modules in Tax-Simulator

Tax-Simulator allows users to write custom behavioral feedback modules. These modules can be called in Tax-Simulator to simulate micro-level responses to policy changes — that is, to account for behavioral feedback in revenue estimates. This document describes how Tax-Simulator's behavioral modules work.

Quickly, a bit of terminology up front. *Static* revenue estimates contain no behavioral feedback: taxpayer behavior is held fixed across policy scenarios. *Conventional* revenue estimates allow for certain types of behavioral responses that reflect tax avoidance rather than a change of economic substance. Examples include business income shifting across legal entity form and the timing of capital gains realizations. *Partial dynamic* estimates further expand the range of incorporated behaviors, allowing for first-order economic changes such as labor supply changes. *Dynamic*, or full dynamic, refers to "true" general equilibrium modeling in which macroeconomic factor prices like interest rates are allowed to vary. **Tax-Simulator can be used to generate conventional and partial dynamic estimates only.** Full dynamic requires a notion of general equilibrium available only in computational macro models.

Below, I describe how behavioral feedback modules work and how to write your own.

Some architecture basics

This section describes where behavioral feedback fits into Tax-Simulator.

Types of functions

Tax-Simulator can be thought of as one big function mapping *input* attributes of tax microdata (variables like number of children, income, and expenses) to *output* attributes (variables like deductions, tax liability, and credits). It is made of many smaller functions. These smaller functions take one of two forms:

- **Tax calculation functions.** These functions calculate output attributes for an arbitrary set of inputs. For example, `calc_ctc()` returns the Child Tax Credit for a set of input tax units. This is the TurboTax portion of the code.
- **Economic functions.** These functions act on the input attributes of tax microdata.

Behavioral feedback modules fall into the latter category. A behavioral feedback module is, literally, an R function expressing input attributes (e.g. capital gains) as a function of tax policy (e.g. the marginal effective tax rate on capital gains). When tax policy changes, these modules are called and update the input attributes of tax microdata to reflect behavioral feedback.

Static mode

At runtime, if told to execute a behavioral feedback module as part of a simulation of a policy reform, Tax-Simulator will run non-baseline scenarios twice. First, it runs in *static mode.* This means all input attributes are held fixed at their baseline levels when calculating taxes under the policy reform. Critically, this step allows us to assess the first-order effects of a policy — for example, effective marginal tax rates (EMTRs) under the new policy regime.

Then Tax-Simulator runs the policy reform scenario again, this time in *non-static* mode. **Behavioral feedback modules are executed at the beginning of non-static mode.** These modules modify input attributes according to their specific logic, then Tax-Simulator re-calculates taxes. The result is a policy reform simulation which accounts for behavioral feedback.

Organization and naming conventions

In the Tax-Simulator repository, code and data are organized into one of two folders. The */src* folder contains scenario-invariant model code — which is to say, most of it. The */config* folder contains configuration files which are used to define tax policy scenarios, economic assumptions, and other specifications about execution. Despite being made of code, behavioral feedback modules fall into the */config* bucket: these modules *configure* the assumptions about how taxpayers react to tax reforms. In other words, along with tax law, behavioral feedback assumptions *define* a scenario.

As such, behavioral feedback modules are .R files stored in */config/scenarios/behavior*. Within this directly, modules are organized into subfolders by the type of behavior modeled.

Let's take */charity* as an example. This subfolder contains modules that modify charitable contributions in response to policy reforms that directly or indirectly affect the tax subsidy for giving. There are two modules in */charity*. 100.R simulates a tax-price elasticity of charitable giving of -1; 50.R assumes a value of -0.5. Module names should be brief and describe what the module does. We will look at the contents of these modules below.

Critically, both modules contain a functional called `do_charity()` ; "charity" in "do_charity" is a keyword which communicates specific information to Tax-Simulator when it looks for functions to execute. A behavioral feedback module in the subfolder */config/scenarios/behavior/X* must contain a function called `do_X()` .

Configuring the model to execute with specific behavioral feedback modules

A *runscript* is a CSV file that defines scenarios for Tax-Simulator to run. Runscripts are located in */config/runscripts*. Every row represents a scenario to run; columns reflect parameter values defining each scenario.

One column is called "behavior". This is where you list the names of behavioral feedback modules to run as part of a scenario. Modules are referred to by their filename relative to the */behavior* directory. If running more than one module in a single scenario, separate the filenames with a single space.

For example, let's say I wanted to run a policy experiment where I raised ordinary tax rates by 2 percentage points, a policy which increases the implicit tax subsidy for charitable contributions, and account for the reforms' impact on contributions. I want to assume an elasticity of -1 under the functional form contained in the */charity/100.R* module. I would specify the following runscript (extraneous columns are left hidden):

ID	…	tax_law	behavior	…	mtr_vars	mtr_type
baseline	…	baseline		…	char_cash	nextdollar
2_pp	…	tests/charity/2_pp	charity/100	…	char_cash	nextdollar

The "behavior" attribute is left blank for the baseline scenario, since behavioral feedback is a concept that applies only to counterfactual reforms. For the reform scenario, titled "2_pp", I specify *charity/100* which tells Tax-Simulator to execute the code contained in the 100.R module when running this scenario. Because the code contained in 100.R module depends on knowing the marginal tax rate on cash contributions (variable "char_cash" in the data) current current law and under the reform, I include *char_cash* as in column "mtr_vars" and *nextdollar* in the column "mtr_type", which tells Tax-Simulator to compute the next-dollar EMTR on whatever variables are specified. (The other option for “mtr_type” is *extensive*, in which the average effective tax rate over the realized value is calculated by setting the value to $0.) If my behavioral feedback module did not depend on any EMTR calculations, I could leave this column blank.

Module structure

A behavioral feedback module contains a “do” function, as described above. This function must have the following signature:

```
do_some_behavior = function(tax_units, ...) {

  # (function contents)

}
```

The parameter `tax_units` is the dataframe containing the tax microdata. It contains all tax records and their attributes, including all tax law attributes. For example, a record in `tax_units` contains variables like marital status, capital gains, and the top tax bracket.

The parameter `…` simply means that any number of other, arbitrary arguments can be passed to the function. The reason for this structure is beyond the scope of this how-to guide. But in practice, your function will also have access to two additional arguments: `baseline_mtrs` and `static_mtrs`. These are dataframes, equal in length to `tax_units`, containing marginal tax rate variables requested at runtime. If your scenario does not calculate marginal tax rates, these arguments will be of type `NULL`.

For example, if I submitted “wages char_cash” as variables for which to calculate marginal tax rates in my runscripts CSV, the `baseline_mtrs` dataframe will look something like this:

year	id	mtr_wages	mtr_char_cash
2024	1	0.24	0
2024	2	0.32	-0.32
2024	3	0.408	-0.408
…	…	…	…

…and `static_mtrs` will look identical, just with (potentially) different values.

Then, the content of the function is completely arbitrary. **This is the key idea behind our design of behavioral feedback modules.** Rather than be confined to expressing behavior feedback as a specific instance of a single generic elasticity function, users are free to impose whatever logic you’d like.

That’s not to say there are no out-of-the-box helper functions. Many behavioral feedback functions will simply take the form of applying an elasticity to a change in EMTRs for all tax units. Rather than having to re-invent the wheel, users have `apply_mtr_elasticity()`, a helper function defined in */src/sim/behavior.R*, at their disposal. We will look at an example of its use below.

Example modules

Let’s make things concrete by reviewing two examples of behavioral feedback modules.

Charitable giving

By offering an itemized charitable deduction, the tax code subsidizes donations for those who itemize. There is a large body of empirical research measuring the responsiveness of charitable activity to the “tax price” of giving, i.e. the after-tax cost to the taxpayer of a marginal dollar of giving. Let’s say we believe the tax price elasticity of charitable giving – that is, the percent change in charitable contributions for a one percent increase in the tax price — takes a value of -1. We wish to account the revenue implications of increased giving, and thus increased deductions, as a result of an increase in ordinary tax rates (which lower the tax price of giving). We can write a behavioral feedback module to model this behavior.

One major benefit of microsimulation as a methodological approach is the ability to precisely measure EMTRs on any activity via computation. By adding $1 to some variable Y for each record, re-calculating taxes, and looking at the change in tax liability, we obtain the *next-dollar* EMTR on activity Y. For example, if we add $1 to an itemizer’s charitable contributions, re-calculate their taxes, and their taxes fall by 37 cents, the next-dollar EMTR on charitable contributions is 0.37 and the tax price of giving is 1 + -0.37 = 0.63. We can calculate this EMTR for all filers under baseline and under the policy reform then use this information in our behavioral feedback calculation.

Here is what a behavior feedback module that leverages EMTRs on charitable contributions looks like:

```
do_charity = function(tax_units, ...) {

  #----------------------------------------------------------------------------
  # Adjusts cash charitable contributions along the intensive margin with a
  # tax price elasticity of -1.
  #
  # Parameters:
  #   - tax_units (df)     : tibble of tax units with calculated variables
  #   - baseline_mtrs (df) : year-id indexed tibble of MTRs under the baseline
  #   - static_mtrs (df)   : year-id indexed tibble of MTRs under the static
  #                          counterfactual scenario
  #
  # Returns: tibble of tax units with post-adjustment cash charitable
  #          contribution values.
  #----------------------------------------------------------------------------

  # Set elasticity
  e = -1

  # Apply elasticities and calculate new values
  new_values = tax_units %>%
    mutate(e_char_cash      = e,
           e_char_cash_type = 'taxprice') %>%
    apply_mtr_elasticity('char_cash', baseline_mtrs, static_mtrs, 1)

  # Replace old values with new and return
  tax_units %>%
    select(-char_cash) %>%
    bind_cols(new_values) %>%
    return()
}
```

First: note the formatted function documentation. We require the author of a module follow the style laid out above. The idea is to describe in plain English which parameters the module depends on, what it does, and what it returns. It helps the reader orient themselves before diving into the code.

The first thing we do is define a variable `e = -1` to be our elasticity. Then we assign this variable, as well as a string indicating its type, to each record in a copy of `tax_units`. Next we feed the result into a function called `apply_mtr_elasticity()`. This is a helper function that adjusts a specific variable in a dataframe based on specified elasticity information, returning a one-column dataframe of the post-adjustment variable. For a full look at this function, please see the */src/sim/behavior.R* file. But here's a rundown on what each argument in our function call does:

- `tax_units = (.)` … This argument, omitted by convention in the dplyr chain, is the dataframe of tax units. Crucially, it contains three columns with specific names: "char_cash", "e_char_cash", and "e_char_cash_type". These are required because the next argument is…
- `var = 'char_cash'` … This argument communicates that we wish to adjust the variable called "char_cash". It also tells the function that there are two associated columns in the `tax_units` dataframe: "e_char_cash", which is a filer-level elasticity with respect to "char_cash", and "e_char_cash_type", which describes the functional form of the elasticity. Earlier in the module, we set `e_char_cash = -1` and `e_char_cash_type = 'taxprice'`. Here, "taxprice" is a keyword. There are four possible options for elasticity type:
    - "semi": a log-lin semi-elasticity, i.e. it gives percent change in Y for a percentage point change in the EMTR.
    - "arc": a full log-log elasticity evaluated at the midpoint, i.e. it gives percent change in Y for a percent change in EMTR, where the latter is calculated at the midpoint of the two EMTRs.
    - "netoftax": a full log-log elasticity on 1 minus the EMTR.
    - "taxprice": a full log-log elasticity on 1 plus the EMTR.

Here are the actual calculations for each option:

```
    pct_chg = case_when(
      e_type == "semi"     ~ exp((mtr - mtr_baseline) * e) - 1,
      e_type == "arc"      ~ (e * (mtr / ((mtr + mtr_baseline) / 2) - 1)),
      e_type == "netoftax" ~ (e * ((1 - mtr) / (1 - mtr_baseline) - 1)),
      e_type == "taxprice" ~ (e * ((1 + mtr) / (1 + mtr_baseline) - 1)),
      TRUE                 ~ NA
    )
```

- `baseline_mtrs = baseline_mtrs` … This argument supplies a dataframe of baseline EMTRs, which in this case must contain a variable called "mtr_char_cash".
- `static_mtrs = static_mtrs` … This argument supplies a dataframe of EMTRs under the static simulation of the policy reform, which again in this case must contain a variable called "mtr_char_cash".
- `max_adj = 1` … This argument limits the absolute value of the resulting percent change for any record to 100%. It prevents unreasonable adjustments resulting from edge cases in EMTRs, which in rare cases can be extreme when a taxpayer is stuck at a notch in the code.

To summarize, we set the tax-price charitable contributions elasticity to -1 for each record. Then we pass our data, including EMTRs on charitable contributions under the baseline and under the policy reform, to a helper function. This function multiplies the result of the following calculation to cash charitable contributions for each record…

$$1 + e\left(\frac{1 + EMTR_{policy}}{1 + EMTR_{baseline}}\right)$$

…and returns the resulting new values of cash charitable contributions. We assign these values to an intermediate dataframe called `new_values` . Finally, in the next block of code, we replace the old values of cash charitable contributions with these new values. (Note that we return the whole tax units dataframe, not just the new column. The result of a behavioral feedback module should always be the identical `tax_units` dataframe input except with new values for the variable(s) we are simulating feedback for.)

And we're done! The module has applied the logic in our behavioral feedback module to the simulated tax records. Tax-Simulator will go on to calculate taxes based on this adjusted data, and our end results will reflect our assumptions about behavioral responses.

One last note on EMTR elasticity-based behavioral feedback modules. This example was a comparatively crude example: We are applying the same elasticity to all tax units in all years. In real life, elasticities may vary with observable attributes like income (rich people are more concerned with tax optimization). We could just as easily have assigned specific elasticities to different types of records in the `mutate()` call, and the logic afterwards would be unchanged.

Employment

The prior module assumed that behavioral responses occur only at the intensive margin: charitable giving was increased or decreased only among those who already gave under the baseline. But many behavioral responses we wish to model are extensive-margin phenomena, in which nonzero values can become zeros and vice versa. Low-income employment and its relationship to tax policy is one such example.

In 2021, the Child Tax Credit (CTC) was expanded such that the credit no longer phased in with earnings — meaning that the reform increased EMTRs on low-income workers. This reduction in the "return to work" may, via substitution effects, cause some employment loss among marginally attached low-income workers. A partial dynamic budget cost estimate will reflect the budgetary impact of this change in employment, which should reduce payroll taxes and income taxes. We can write a behavioral feedback module to account for this effect in our cost estimate.

Below is a behavioral feedback module implementing the logic and parameters of Bastian (2023) (file:///C:/Users/jar335/Downloads/Bastian_CTCexpansion_2023.pdf), a paper which estimates employment loss caused by the 2021 CTC expansion. Bastian calculates a "return-to-work" metric, defined as 1 minus the average effective tax rate on working. By demographic group, he calculates the change in return-to-work caused by the CTC reform and then applies demographic-specific labor supply elasticities to obtain employment loss estimates.

```r
do_employment = function(tax_units, ...) {

  #----------------------------------------------------------------------------
  # Adjusts wage earnings at the extensive margin, per Bastian (2023). Only
  # suitable to analyze an *increase* in EMTRs at the low end -- there is no
  # symmetric effect in which nonworkers become workers in response to a cut in
  # EMTRs. Used in our analysis of the employment effects of the 2021 CTC.
  #
  # Parameters:
  #   - tax_units (df)     : tibble of tax units with calculated variables
  #   - baseline_mtrs (df) : year-id indexed tibble of extensive-margin MTRs on
  #                          wages1 and wages2 under the baseline
  #   - static_mtrs (df)   : year-id indexed tibble of extensive-margin MTRs on
  #                          wages1 and wages2 under the static counterfactual
  #
  # Returns: tibble of tax units with post-adjustment wage earnings values.
  #----------------------------------------------------------------------------

  # Set random seed
  set.seed(globals$random_seed)

  # Set elasticities
  e_mothers_poor  = 0.4
  e_mothers_other = 0.2
  e_else          = 0.05


  tax_units %>%

    # Join MTRs
    left_join(baseline_mtrs %>%
                rename_with(.cols = -c(id, year),
                            .fn   = ~ paste0(., '_baseline')),
              by = c('id', 'year')) %>%
    left_join(static_mtrs, by = c('id', 'year')) %>%

    mutate(

      # Calculate tax unit-level income (roughly AGI)
      income = wages + txbl_int + div_ord + div_pref + state_ref +
               txbl_ira_dist + txbl_pens_dist + kg_lt + kg_st + other_gains +
               sole_prop + part_active + part_passive - part_active_loss -
               part_passive_loss - part_179 + scorp + scorp_active +
               scorp_passive - scorp_active_loss - scorp_passive_loss -
               scorp_179 + rent - rent_loss + estate - estate_loss + farm + ui +
               gross_ss + other_inc,

      #-----------------
      # Set elasticities
      #-----------------

      # First earner
      e1 = case_when(

        # Low-income single mothers
        (male1 == 0) & (n_dep_ctc > 0) & (wages1 < eitc.po_thresh_1) & (filing_status != 2) ~ e_
mothers_poor,

        # All other mothers with family income below $80,000
        (male1 == 0) & (n_dep_ctc > 0) & (income < 80000) ~ e_mothers_other,

        # Others below $80,000
        (income < 80000 & n_dep_ctc > 0) ~ e_else,

        # Everyone else
        TRUE ~ 0
      ),

      # Second earner
      e2 = case_when(

        # Low-income single mothers
        (male1 == 0) & (n_dep_ctc > 0) & (wages1 < eitc.po_thresh_1) & (filing_status != 2) ~ e_
mothers_poor,

        # All other mothers with family income below $80,000
        (male1 == 0) & (n_dep_ctc > 0) & (income < 80000) ~ e_mothers_other,

        # Others below $80,000
        (income < 80000 & n_dep_ctc > 0) ~ e_else,

        # Everyone else
        TRUE ~ 0
      ),


      #---------------------------
      # Simulate labor force exit
      #---------------------------

      # Calculate percent change in return-to-work
      delta_rtw1 = ((1 - mtr_wages1) - (1 - mtr_wages1_baseline)) / (1 - mtr_wages1_baseline),
      delta_rtw2 = ((1 - mtr_wages2) - (1 - mtr_wages2_baseline)) / (1 - mtr_wages2_baseline),

      # Calculate probability of remaining employed defined as 1 plus the
      # implied percent change in employment
      pr_emp1 = 1 + (e1 * delta_rtw1),
      pr_emp2 = 1 + (e2 * delta_rtw2),

      # Simulate outcomes
      emp1 = runif(nrow(.)) < pr_emp1,
      emp2 = runif(nrow(.)) < pr_emp2,

      # Adjust wages
      wages1 = if_else(wages1 == 0, 0, wages1 * emp1),
      wages2 = if_else(wages2 == 0, 0, wages2 * emp2),
      wages  = wages1 + wages2

    ) %>%

    # Remove intermediate calculation variables and return
    select(-income, -e1, -e2, -delta_rtw1, -delta_rtw2,
           -pr_emp1, -pre_emp2, -emp1, -emp2) %>%
    return()
}
```

In terms of function inputs, this module uses extensive-margin tax rates — i.e., average effective tax rates — on primary-earner wages ("wages1") and, in the case of joint returns, secondary-earner wages ("wages2"). These variables are calculated earlier in Tax-Simulator's execution and are passed to our function in `baseline_mtrs` and `static_mtrs`.

At the top, because this module involves stochastic simulation, we reset the random number generator seed — a requirement in Tax-Simulator before any call to a random number generator. We also list our elasticity assumptions at the top per convention.

Then we join our EMTR dataframes into tax units, making sure to distinguish baseline from policy reform variables by renaming the former.

Next, we assign labor supply elasticities to each nondependent adult. As per Tax-Simulator naming convention, primary earner variables are appended with "1" and secondary variables are appended with "2". Before assignment, we calculate a helper variable, income, since elasticities are heterogeneous by demographic and income. The `case_when()` function calls implement Bastian's logic: EITC-qualifying single mothers have an elasticity of 0.4; other mothers with income below $80K have an elasticity of 0.2; others below $80K have an elasticity of 0.05; those above $80K are assumed not to be responsive. The important takeaway from these lines is that we can build highly detailed heterogeneity into our behavioral feedback assumptions.

The next step is to calculate the policy reforms's effect on the return to work, again defined as 1 minus the average effective tax rate (somewhat confusingly stored with an "mtr" prefix per Tax-Simulator convention). We calculate the percent change in this metric for primary and secondary earners.

Applying the labor supply elasticity to this quantity gets us the percent change in employment, which is uninterpretable at the micro level. Instead, we convert the result into a *probability of remaining employed* by adding 1. For example, if the reform reduces someone's return to work by 10%, and we assume a 0.4 labor supply elasticity, we'd say that the percent change in employment is -4%, but the probability of remaining employed is 96%, i.e.

$$1 + (-0.1 * 0.4) = 1 - 0.04 = 0.96$$

.

At this point, we have employment probabilities for all workers in our data. The final step is to simulate labor force exit using a random number generator. Wages are set to 0 for those who drop out of the labor force, and *viola* — we're done.