README

Monetary Policy, Segmentation, and the Term Structure

Rohan Kekre[*] Moritz Lenel[†] Federico Mainardi[‡]

1 Overview

The files in this folder contain the programs to replicate the model results presented in the paper. This is an active working paper and the provided codes should be considered as work in progress. We plan to provide updates to the code as we continue to work on the paper, the most recent version can be found on GitHub. Please feel free to reach out to us if you have any questions regarding the code or the paper.

The files are split into two folders:

1. `bin`: files to run the code on either a local desktop or a remote cluster
2. `src`: files that solve and simulate the model and create the output files

In addition, the folder `data` contains empirical results used to calibrate and validate the model results. After running the code, the folder `output` will contain the output files.

2 Computational requirements

2.1 Software requirements

- Julia (code was last run with Julia version 1.9.1)
- Julia packages which can be directly installed by running the script `src/install_packages.jl`

[*]Chicago Booth and NBER. Email: rohan.kekre@chicagobooth.edu.
[†]Princeton and NBER. Email: lenel@princeton.edu.
[‡]Chicago Booth. Email: fmainard@chicagobooth.edu.

2.2 Hardware requirements and runtime

We have produced the results in the paper on a remote computer cluster using 16 cores and 64GB of RAM per node. The code also solves well on a local desktop with only 4 cores and 32 GB of RAM but the simulation of the model is memory intensive. In case the code does not run on a local desktop, we suggest to reduce the parameter NN_sim in `src/params.jl` to, for example, 2.

The actual solution of the model (`calcP()` in `src/solution.jl`) is relatively quick and takes about 20 minutes on our cluster. The solution of unexpected (MIT) monetary and QE shocks takes longer to solve, so that the full solution of all model results for the benchmark calibration takes about 1.2 hours. Parallelizing across calibrations is therefore recommended when solving comparative model specifications.

If one is not interested in the QE results or the monetary shock impulse responses, setting `calc_QE = false` and `calc_IRM = false` in `getParams()` in `src/params.jl` reduces the runtime and the code still produces all tables as well as the impulse responses to expected (non-MIT) shocks.

3 Description of key program files

- `src`: folder containing the Julia source code to solve and simulate the model, as well as to create the output files
  - `main.jl`: main programs to solve and simulate the model
  - `solution.jl`: containing the functions to solve the model
  - `params.jl`: containing the model parameters and the functions for the numerical setup
  - `analytical.jl`: containing the functions to compute the analytical solution of the model under exogenous wealth
  - `empirical.jl`: containing the functions to import the empirical results stored in the `data` folder
  - `results.jl`: containing the functions to simulate the model and store the necessary inputs used by the script `output.jl` to produce the output files
  - `packages.jl`: loading all packages used throughout the code

  - `figures.jl`: containing the functions to produce the figures
  - `tables.jl`: containing the functions to produce the tables
  - `install_packages.jl`: script that installs all packages used throughout the code

- `bin`: folder containing the scripts to compile the code
  - `run.jl`: run this file directly in Julia REPL to solve and simulate the model for all calibrations and create output files
  - `bin/run_parallel.sh`: bash script to run the code on a cluster, distributing the solution of different parameterizations across nodes
  - `bin/get_results.sh`: bash script to solve a parameterization on a cluster node (called by `run_parallel.sh`)
  - `bin/make_output.sh`: script to create output files from the collected solutions (called by `run_parallel.sh`)

3.1 Instructions

To run the code locally and create the output files, change to the `bin` directory, open the Julia REPL and include `run.jl` after selecting the desired calibrations in `src/pramas.jl`. Since the computationally intensive routines are parallelized using Julia's multi-threading features, it is recommended to start the Julia environment with multiple threads (e.g `Julia -t 4` on a quad-core system).

Alternatively, the selected calibrations can be run in parallel on a cluster by executing the bash script `run_parallel.sh` file in the `bin` folder. Note that the `run_parallel.sh` file has been written for a specific Unix system and will need to be adapted for other environments. In `src/run.jl`, the output folder for temporary result files is set depending on the second input argument specified in `bin/get_results.sh` and `bin/make_output.sh`. These arguments and the output folder specifications need to be adapted to point to the scratch folder in the current environment. Alternatively, the second input argument (`della`) can be removed in both `bin/get_results.sh` and `bin/make_output.sh` to also store temporary result files in the local `output` folder.

3.1.1 Understanding the code

To understand which steps the code performs, open the `main.jl` file in the `bin` folder. This files imports all relevant files in the `src` folder and provides the `main()` function that solves and simulates a specific calibration. Before studying `main()` in detail, it is useful to understand the content of file `params.jl`, which defines a structure `Params` that contains a benchmark parameterization of the model. The function `getParams()` in that same file collects all comparative calibrations that are to be solved by the code, and which are all derived from the default structure `Params`.

The model is solved by functions contained in the file `solution.jl`. The `main()` function initializes the solution by calling `calcP()` within that file. We hope that the description of the solution algorithm in the file `SolutionAlgorithm.pdf` will be helpful in understanding the steps in `calcP()`.

3.1.2 Modifying the code

To run alternative calibrations, adjust the parameters defined in `params.jl`. The `params` structure defines a benchmark set of parameters. See function `getParams()` in `params.jl` for examples of how to add additional parameterizations. To solve alternative model setups which also permit a solution using the Feynman-Kac formula, take a look at `getP!` in `solution.jl`, which contains the core routine for the Monte Carlo simulations.

Beyond the tables and figures produced for the paper, we also include more general output that can be adjusted to develop new results. For example, `makeAddlFigures` in `figures.jl` produces individual impulse responses for each calibration to shocks to r, β and W. The resulting figures are stored in `output/figures/addl_figures`.