Optimal climate policy when damages are unknown

Ivan Rudik

American Economic Journal: Economic Policy

Top level instructions

IPSCR Number / Link

Data can be found here (https://www.openicpsr.org/openicpsr/project/111185/view).

The ICPSR number is 111185.

Expected run time

The expected runtime for reproducing all results is >20,000 core-hours. Expected run time for reproducing results in the paper from intermediate data files is about 10 minutes.

What to do and know before reproduction

1. The code to solve and simulate the models is written in Julia (https://julialang.org), the code to estimate the damage parameters is written in Stata and can be found in `/estimate_damage_parameters`, and the code to generate the figures is written in R and can be found in `/generate_plots`.
2. The Julia version used in the paper is v1.1.0. Versions prior to v1.0 (except for possibly v0.7) will not work because of major changes in language syntax, and later versions may not be backwards compatible with third-party packages.
3. The operating system used to produce the results in the paper is Linux CentOS 7.6.1810.
4. You do not need to declare a seed for the random draws of d_2, d_1, ω_{t+1} in the simulations. The draws used in the paper are contained in `/results` because regenerating the draws for each set of simulations tends to be more time-costly than simply reading them from a pre-existing file. A comment in the function `simulate_paths` shows example code for generating random draws from the distributions.

5. The code requires several packages. These can be installed with the following commands. If the `using` command to load the packages in `MAIN_SCRIPT.jl` throws an error it is usually because you need to build a particular package after installing: `Pkg.build(package_here)`.

```
import Pkg
Pkg.add("BSON")
Pkg.add("Roots")
Pkg.add("DataFrames")
Pkg.add("CSV")
Pkg.add("Distributed")
Pkg.add("CompEcon")
Pkg.add("BasisMatrices")
Pkg.add("NLopt")
Pkg.add("Distributions")
Pkg.add("Dierckx")
Pkg.add("LinearAlgebra")
```

How to reproduce the results

Main results: full execution of code

1. Change directory to the folder containing the Julia code: `cd("directory_path_here")`
2. Install required packages
3. Run `MAIN_SCRIPT.jl`

Main results: from intermediate data files

1. Change directory to the folder containing the Julia code: `cd("directory_path_here")`
2. Install required packages
3. Run `results_from_intermediate.jl`

Damage function estimates (Table 1)

1. Change directory to the folder containing the Stata code: `cd "directory_path_here"`
2. Run `/estimate_damage_parameters/table_1.do`

Plot figures

1. Change directory to the folder containing the R code: `setwd("directory_path_here")`
2. Run `/generate_plots/make_plots.r`

Files and functions

Below each file name is a list of functions in the file and a short description of what they do. Note that multiple functions with the same name may be defined within the same file to handle different sets of input arguments.

MAIN_SCRIPT.jl

The master script. It loads packages and calls the functions in `model_solvers.jl` to solve the models. Comments above each function call indicate the corresponding figures and tables in the paper.

results_from_intermediate.jl

Skips over solving the models and generates results reported in the paper from the pre-made intermediate data files.

model_solvers.jl

Functions that call `compute_value_function` in `model.jl` and solve the different versions of the model using the preallocated `inputs` dictionary defined below.

- `solve_uncert`, `solve_learn`, `solve_rc`, `solve_rcl` : solve the base frameworks and saves results to a bson file, if solving a robust control-using framework it takes in the penalty parameter `theta` as an argument, and each of the functions takes in a subset of the following keyword arguments (non-RC frameworks take in an argument `na = 1` that is a dummy just so we can use optional keyword arguments):
  - `quad` : integer for the number of quadrature points for d_2 and $\{d_1, \omega_{t+1}\}$ (default = 11)
  - `expand` : float that determines the expansion of the domain size for Table A4 (default = 1.)
  - `adapt_iters` : integer for the number of adaptive grid iterations (default = 20)
  - `load` : if true then the function will load the pre-solved learning model to use its adaptive collocation bounds and skip the 20 iteration adaptive procedure (default = false)
- `simulate_alt_damage_params` : simulates a framework with saved bson file located at `model_path` with alternate damage parameters `d1` and `d2`
- `simulate_misspecified_damages` : simulates a framework with saved bson file located at `model_path` with a misspecified damage function given by `damage_type` which is a symbol equal to `:weitzman`, `:dietz`, or `:extreme`
- `solve_alt_penalty_parameter_rc` : solves the robust control framework under the alternate penalty parameters shown in Figure 6a
- `solve_alt_penalty_parameter_rcl` : solves the robust control + learning framework under the alternate penalty parameters shown in Figure 6b
- `solve_uncert_error_coll`, `solve_learn_error_coll`, `solve_rc_error_coll`, `solve_rcl_error_coll` : solve the different frameworks with the varying number of collocation points for the error analysis
- `solve_learn_previous_iter` : solves the learning framework with only 19 adaptive iterations to get the second-to-last set of adaptive collocation bounds for Figure A2

model.jl

Functions to perform value function iteration.

- `compute_value_function` : outer function for computing value functions, called by functions in `model_solvers.jl`

- `backwards_induce` : backwards induces the dynamic programming problem from the terminal year to year 0
- `iterate_nodes` : iterates on the collocation points and calls `maximize_bellman` to maximize the Bellman at each collocation point for a given year
- `iterate_adaptive_grid` : iterates to adapt the grid based on simulated trajectories

initialization.jl

Function to initialize the model.

- `parameterization` : creates dictionaries containing the model parameters (`mparams`), exogenous state trajectories (`exog_states`), boolean variables denoting the model options (`opts`), and the approximation parameters (`space`); creates a pre-inverted Vandermonde matrix of the Chebyshev polynomials (`Binv`); and creates the collocation grid (`states`)

maxbell.jl

Functions to maximize the Bellman.

- `maximize_bellman` : maximizes the Bellman at a particular state and year
- `value_func` : the value function for learning models
- `value_func_unc` : the value function for non-learning models
- `resource_constraint` : the amount of slack in the resource constraint

transitions.jl

State transition functions.

- `transitions` : compute state transitions for simulations

adaptive_grid.jl

Function that generates the adapted grid bounds.

- `time_adaptive_grid` : adapt grid to simulated trajectories

simulation.jl

Functions to simulate a solved model.

- `simulate_paths` : outer function for initializing the simulations, it takes in the following keyword arguments:
    - `misspec_damages` : symbol determining the damage function (default = `:base`)
    - `alt_damages` : boolean equal to true if simulating with alternate damage parameters (default = `false`)
    - `d1` : alternate damage coefficient if `alt_damages == true`
    - `d2` : alternate damage exponent if `alt_damages == true`
- `simulate_paths_adapt` : outer function for initializing the simulations for the adaptive algorithm

- `simulate_inner` : inner function that simulates the model from year 0 to the end simulation year
- `simulate_exog_vars` : simulates the exogenous variables

tax_calc.jl

Function to compute the optimal tax channels.

- `compute_tax` : computes the optimal tax and performs the tax decomposition

custom_types.jl

Defines composite types to facilitate passing variables and containers to functions.

- `stateVariables` : variables to be passed to the Bellman maximizer
- `simParameters` : variables for the simulations
- `taxParameters` : variables used to compute optimal taxes and decompose the tax into the seven uncertainty channels

tools.jl

Auxiliary functions for manipulating data and computing welfare.

- `extract` : extract variables from saved dictionaries into the workspace
- `reduce_dict` : reduces the exogenous states dictionary to only time t variables
- `get_bounds` : calculates percentile bounds for the simulated trajectories
- `extract_welfare_data` : pull out welfare variables from a saved model
- `ex_ante_welfare` : compare *ex ante* welfare between two models
- `ex_post_welfare` : compute *ex post* welfare for a model and horizon

RESULTS.jl

Generates results reported in the paper from intermediate output bson files in `/results` . Comments indicate which sets of code generates which figures and tables.

- `generate_results` : calculates results for the figures and tables and writes them to csv files in `/generate_plots/data`

/estimate_damage_parameters/table_1.do

Estimates the distributions for d_1, d_2, ω_{t+1} .

/generate_plots/make_plots.r

Makes the figures in the paper from the csv files made by `generate_results` .

Definitions for `input` dictionary entries

`input` is a dictionary that contains all the relevant inputs into the model that are changed for different parts of the paper. In each function in `model_solvers.jl`, they have already been preallocated so that the functions will generate output saved to bson files that are used to reproduce the results in `RESULTS.jl`. Below I group the inputs by type and whether they are general model inputs or inputs specific to the adaptive grid procedure.

General entries

Booleans

- `perfect_info` : boolean equal to true if doing a perfect information model
- `learn_e` : boolean equal to true if allowing the policymaker to learn
- `robust` : boolean equal to true if allowing the policymaker to use robust control

Numbers, containers, symbols

- `scaling_factor` : value used to rescale the utility function (and terminal value function) so $\theta = 0.27$ is next to the breakdown point for all frameworks; *the rescaling does not directly affect decisionmaking in any way since it performs an affine transform of the utility function, and it has been correctly accounted for in welfare calculations between models with different scaling factors*
    - The scaling factor is needed because the exponential term in the RC operator causes overflow at different θs due to differences in the *levels* of the value function, the scaling factor addresses this issue in a way that preserves the preference ordering
- `initial_approximation_level` : the initial number of collocation points for each dimension in order of effective capital, cumulative emissions, location parameter, scale parameter, fractional net output
- `increase_level` : the years in which to increase the approximation level by `increase_dims` on the corresponding states
- `increase_dims` : how many collocation points to add to each dimension, each row corresponds to the same row in `increase_level`
- `time_horizon` : time horizon for the model
- `sim_length` : time horizon for the simulated trajectories
- `theta` : robust control penalty parameter value
- `damages` : symbol equal to `:base` , `:stern` , `:dietz` , or `:extreme` to select the simulation damage function
- `num_runs` : number of simulations
- `nqnodes` : number of quadrature points for the d_2 and the joint d_1, ω distributions

Adaptive procedure entries

Booleans

- `adapt` : boolean equal to true if you want to adapt the collocation grid to simulated trajectories
- `load` : boolean equal to true if you want to load previous adaptive bounds that should be stored in `smax` and `smin` instead of starting from a time-invariant grid

Numbers, containers, symbols

- `num_adapts` : how many times to perform the adaptive procedure
- `num_runs_adapt` : how many runs to simulate for each iteration of the adaptive procedure
- `sim_length_adapt` : the length of each simulation in years for the adaptive procedure
- `increase_adapt_iter` : at which adaptive iteration you want to begin increasing the approximation level
- `increase_level_adapt` : the years in which to increase the approximation level by `increase_dims` on the corresponding states when the adaptive procedure is on at least iteration `increase_adapt_iter`
- `increase_dims_adapt` : how many collocation points to add to each dimension, each row corresponds to the same row in `increase_level_adapt`
- `new_bounds_weight_up` : the weight placed on the new upper bounds calculated by the most recent adaptive procedure, 1 minus this value is placed on the old upper bounds, there is one value for each state
- `new_bounds_weight_down` : the weight placed on the new lower bounds calculated by the most recent adaptive procedure, 1 minus this value is placed on the old lower bounds, there is one value for each state
- `smax/smin` : bounds to load, set equal to an array of zeros if not loading

Codebook

The key variables are:

- `tax_channels` : dictionary containing *ex ante* expected tax trajectories
- `channels_out_samples` : dictionary containing the actual tax trajectories for each simulation
- `consumption` : vector of effective consumption trajectories for each simulation
- `capital` : vector of effective capital state trajectories for each simulation
- `co2` : vector of cumulative emissions trajectories for each simulation
- `prior_loc` : vector of location parameter trajectories for each simulation
- `prior_scale` : vector of scale parameter trajectories for each simulation
- `damage` : vector of fractional net output trajectories for each simulation
- `mparams` : dictionary of time-invariant model parameters
- `opts` : dictionary of booleans of model options
- `space` : dictionary of numerics and containers defining the approximation space
- `c_store` : vector of the vectors of basis function coefficients for each year-specific value function approximation
- `smax` : time-variant upper bounds of the collocation grid for the learning frameworks
- `smin` : time-variant lower bounds of the collocation grid for the learning frameworks
- `sim_values` : damage coefficient, exponent, and shock draws specific to each simulation

Results

The function `generate_results` in `RESULTS.jl` is called at the end of `MAIN_SCRIPT.jl` . It takes the solved models in the intermediate output bson files and writes the results to generate the figures and tables in the paper to csv files in `/generate_plots/data` (e.g. figure_6a.csv, table_3.csv, figure_a2_bounds.csv, figure_a2_states.csv).

The pre-made csv files containing the final results used in the paper can be found in `/generate_plots/data`.

The R script `make_plots.R` in `/generate_plots` reads in these csv files and creates the figures in the paper.